

Universidade Federal Fluminense

DIEGO ASTH SCHUENCK LEAL

Suportando a adaptação de aplicações pervasivas pelo
uso de funções utilidade

NITERÓI

2007

DIEGO ASTH SCHUENCK LEAL

Suportando a adaptação de aplicações pervasivas pelo uso de funções utilidade

Dissertação de Mestrado submetida ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do título de Mestre. Área de concentração: Processamento Paralelo e Distribuído.

Orientador:

ORLANDO GOMES LOQUES FILHO

UNIVERSIDADE FEDERAL FLUMINENSE

NITERÓI

2007

Ficha Catalográfica elaborada pela Biblioteca da Escola de Engenharia e Instituto de Computação da UFF

L435 Leal, Diego Asth Schuenck.
Suportando a adaptação de aplicações pervasivas pelo uso de
funções utilidade / Diego Asth Schuenck Leal. – Niterói, RJ : [s.n.],
2007.
130 f.

Orientador: Orlando Gomes Loques Filho.
Dissertação (Mestrado em Computação) - Universidade Federal
Fluminense, 2007.

1. Processamento distribuído. 2. Computação pervasiva. 3.
Computação móvel. 4. Arquitetura de computadores. I. Título.

CDD 004.36

Suportando a adaptação de aplicações pervasivas pelo uso de funções
utilidade

Diego Asth Schuenck Leal

Dissertação de Mestrado submetida ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do título de Mestre.

Aprovada por:

Prof. Orlando Gomes Loques Filho, Ph.D. / IC-UFF
(Presidente)

Prof. Célio V. N. Albuquerque, Ph.D. / IC-UFF

Prof. Alexandre Sztajnberg, D.Sc. / UERJ

Niterói/RJ - Brasil

09 de julho de 2007

Às minhas pessoas queridas.

Agradecimentos

É muito reconfortante olhar para trás agora e ver que a missão foi cumprida. Faz parecer que as noites mal-dormidas e os artigos intermináveis nunca existiram. Porém, o fato de apenas o meu nome aparecer na capa dessa dissertação, dá a impressão de que só eu fui responsável por ela. Não é verdade. Esse trabalho simplesmente não seria possível sem a ajuda de diversas pessoas.

Gostaria de agradecer às minhas chefes na Embratel, no período de 2004 a 2006, Miriam Rumiantzeff e Asunta Guardia pela liberação de um bom número de horas semanais para que eu pudesse frequentar as aulas e as reuniões do curso. Sem seu apoio eu não conseguiria fazer como aquele chinês do circo que equilibra vários pratos ao mesmo tempo, girando-os na ponta de varetas para não deixá-los cair. Um de cada vez.

Ao meu orientador, prof. Orlando Loques, por cada dificuldade que ele colocou no meu caminho. As dificuldades sempre foram dosadas¹ de forma que eu pudesse, com a sua ajuda e paciência, crescer e superá-las.

Aos meus pais, por tudo.

Aos meus amigos pela paciência com que ouviram um único motivo para as minhas recusas durante todos esses anos todos.

Ao meus amigos de curso, Jacques da Silva, Renatha Capua e Cristiane Ferreira pelo apoio quando muito parecia fadado ao fracasso e os prazos inatingíveis.

Aos meus amigos do grupo de pesquisas, Alexsandro Corradi, André Santos, Leonardo Cardoso, Glauco Freitas, Alessandro Copetti e Jonivan Lisboa pelas discussões que melhoraram o conteúdo deste trabalho.

A Deus por nunca ter me faltado. Quando em uma certa fase eu me perguntava, todos os dias, porque eu estava cursando esse Mestrado, Ele me mostrava a resposta.

¹Embora num primeiro momento não parecem ser :-)

“Cada um pensa como pode.”

Mário Quintana

Resumo

As aplicações cada vez mais operam em ambientes dinâmicos, seja pela variação da disponibilidade de recursos, seja pela variação das preferências do usuário. Para manter a satisfação do usuário durante a utilização dessas aplicações, os custos de administração têm crescido muito. Neste cenário, aplicações pervasivas percebem essas variações no contexto de operação e se adaptam, independentemente da ação do usuário ou de um administrador. Além disso, se o mecanismo utilizado para a adaptação puder ser fatorado da aplicação, então ele poderá ser reusado em diversos casos.

Este trabalho objetiva relacionar questões que devem ser resolvidas para suportar a adaptação de aplicações pervasivas, dentre elas: descrição da arquitetura da aplicação, descoberta de recursos no ambiente, monitoração, elicitación das preferências do usuário e, principalmente, a escolha do melhor recurso disponível. Buscando resolver essa última questão, propomos - como principal contribuição deste trabalho - o emprego de uma função utilidade por entender que ela constitui uma técnica que leva em conta as preferências dos usuários (que ponderam dimensões de QoS frequentemente conflitantes) e a disponibilidade dos recursos que impactam a qualidade da aplicação. Outra virtude dessa técnica é a facilidade de implementação e a pouca quantidade de processamento exigida, propiciando ser utilizada em pequenos dispositivos com recursos computacionais e de energia restritos, tais como PDAs.

Para validar nossa proposta, apresentamos alguns exemplos de aplicação onde empregamos uma função utilidade para realizar a escolha do melhor recurso. No exemplo em que um cliente sem-fio escolhe o melhor ponto de acesso, analisamos *traces* reais do tráfego de uma rede sem-fio e *traces* sintetizados por um simulador construído e parametrizado com distribuições realísticas de probabilidade. Dessa análise, verificamos a qualidade e a robustez da solução baseada em uma função utilidade e concluímos que a principal métrica disponível em produtos comerciais para a escolha do melhor AP, a intensidade do sinal, não deve guiar exclusivamente essa escolha.

Palavras-chave:

Auto-adaptação, dimensões de QoS, adaptação em tempo de execução, CR-RIO, monitoração, requisitos não-funcionais, seleção de recursos, função utilidade, contexto.

Abstract

Applications operate in dynamic environments, due to the variation of the availability of the resources or due to the variation of the user's preferences. To keep user's satisfaction during the operation of such applications, administration costs have grown up. In this scenario, pervasive applications are aware of such variations in the operation context and adapt themselves without the action of users or administrators.

This work aims to list the questions that have to be solved to make adaptation possible, among them: application's architecture, resource discovery, monitoring, user's preferences elicitation and, mainly, best available resource selection. Trying to solve this last question, we propose - as our main contribution - to use a utility function, as a technique which takes into account user's preferences (to weigh conflicting QoS dimensions) and the availability of the resources needed to provide the proper quality. An extra feature is the ease of implementation and the low processing costs, what makes it possible to be used in small devices, like PDAs.

In order to validate our proposal, we present some examples where we used a utility function to select the best resource. In the example where a client selects the best AP, we analyzed real traffic traces of a corporate wireless network and traces generated by a simulator parameterized by realistic probability distributions. We verified the quality and robustness of a solution based in a utility function and concluded that the main QoS dimension commercially available to select the best AP, the signal strength, should not guide this selection exclusively.

Keywords:

Auto-configuration, QoS dimensions, run-time adaptation, CR-RIO, monitoring, non-functional requirements, resource selection, utility function, context.

Glossário

ADL	: <i>Architecture Description Language</i>
AP	: <i>Access Point</i>
API	: <i>Application Programming Interface</i>
AR	: <i>Anycast Resolver</i>
CR-RIO	: <i>Contractual Reflective - Reconfigurable Interconnectable Objects</i>
DNS	: <i>Domain Name System</i>
JPEG	: <i>Joint Photographic Experts Group</i>
HTTP	: <i>Hyper Text Transfer Protocol</i>
IP	: <i>Internet Protocol</i>
ISO	: <i>International Organization for Standardization</i>
MeCo	: <i>Metric Collector</i>
PDA	: <i>Personal Digital Assistant</i>
QoS	: <i>Quality of Service</i>
SNMP	: <i>Simple Network Management Protocol</i>
SQL	: <i>Structured Query Language</i>
OSI	: <i>Open Systems Interconnect</i>
XML	: <i>Extensible Markup Language</i>
URI	: <i>Uniform Resource Identifier</i>
URL	: <i>Uniform Resource Locator</i>

Sumário

Lista de Figuras	xi
Lista de Tabelas	xiii
1 Introdução	1
2 Conceitos preliminares	6
2.1 Razões para a adaptação dinâmica	6
2.2 Características desejáveis em um <i>framework</i>	9
2.3 Descrição arquitetural	10
2.4 Mapeamento físico de componentes	13
2.5 Descoberta de recursos	14
2.6 Monitoração de recursos	19
2.6.1 Meta-dimensões de monitoração	19
2.6.2 Coleta das medições	21
2.6.3 Localização dos componentes coletores	23
2.6.4 Pré-processamento das dimensões de QoS monitoradas	28
2.6.5 Lógica <i>fuzzy</i>	34
2.6.6 Ortogonalidade entre dimensões de QoS	35
2.7 Gerência de ações conflitantes	36
2.8 Conclusão do capítulo	38
3 Trabalhos correlatos	39

3.1	<i>Synthesizer</i> e representação por receitas	39
3.2	CR-RIO	43
3.3	Proposta de Cheng	47
3.4	Aura	50
3.5	Olympus	53
3.6	Conclusão do capítulo	55
4	Exemplos de aplicação	56
4.1	Escolha do melhor ponto de acesso	56
4.1.1	Integração com o CR-RIO	62
4.2	Escolha do melhor servidor <i>web</i>	64
4.3	Parametrizando um componente	68
4.4	Considerações gerais sobre a instabilidade	71
4.5	Conclusão do capítulo	72
5	Estudo de caso	73
5.1	Análise do tráfego de uma rede sem-fio real	75
5.1.1	Resultados	77
5.1.1.1	Dinâmica das medições e influência da janela deslizante	77
5.1.1.2	Comparação entre políticas de escolha do melhor AP	79
5.1.1.3	Correlação entre dimensões de QoS	83
5.1.1.4	Descorrelação entre o número de usuários e o tráfego	84
5.2	Experimentos baseados em simulação	85
5.2.1	Arquitetura centralizada	86
5.2.2	Arquitetura distribuída	88
5.2.3	Descrição dos experimentos para a arquitetura centralizada	91
5.2.4	Resultados	96

5.3	Conclusão do capítulo	105
6	Conclusões e trabalhos futuros	106
6.1	Conclusões	107
6.2	Trabalhos Futuros	108
	Referências	111

Lista de Figuras

1.1	<i>Loop</i> fechado dos <i>frameworks</i>	3
2.1	<i>Cluster</i> típico de servidores	8
2.2	Descrição de um serviço simples baseado na arquitetura cliente-servidor . .	11
2.3	Entidades do banco de dados virtual	16
2.4	Protocolo de descoberta de recursos	18
2.5	Arquitetura proposta para o <i>Anycast Resolver</i>	22
2.6	A forma de coleta sugerida na proposta	26
2.7	Aplicação da estrutura de coleta	28
2.8	Limiares para definição dos servidores equivalentes	29
2.9	Representação gráfica dos conceitos apresentados	31
2.10	Possibilidades de valoração	32
2.11	<i>State Compatibility Value</i> de vários <i>QoSParams</i>	32
2.12	Estados intermediários	34
2.13	Políticas para resolução de conflitos	38
3.1	Comparação entre meta-heurísticas na busca do ótimo global	41
3.2	Elicitação de requisitos e preferências	50
3.3	Hierarquia ontológica da entidade virtual <i>Device</i>	53
4.1	Esquemático de uma rede sem-fio	57
4.2	Função de satisfação para a elicitacão de preferências	61
4.3	Descrição de uma rede sem-fio	62
5.1	Utilização desbalanceada de dois APs de uma mesma rede	74
5.2	Quadro seleção de redes sem-fio do MS Windows	75

5.3	Influência da janela deslizante (A)	78
5.4	Influência da janela deslizante (B)	78
5.5	Variação do tráfego cursado	84
5.6	Variação do número de usuários conectados	85
5.7	Proposta de arquitetura centralizada	89
5.8	Proposta de arquitetura distribuída	90
5.9	Diagrama de classes do simulador	92

Lista de Tabelas

2.1	Alguns métodos presentes na API	12
4.1	Camadas do modelo ISO/OSI	65
4.2	Recursos consumidos com diversas parametrizações	71
5.1	<i>Roamings</i> realizados pelo cliente	80
5.2	Escolhas através de diferentes políticas	81
5.3	Resultados do cenário Rodada01 (clientes homogêneos)	98
5.4	Resultados do cenário Rodada02 (clientes heterogêneos) (A)	101
5.5	Resultados do cenário Rodada02 (clientes heterogêneos) (B)	104

Lista de Códigos

2.1	Descrição arquitetural utilizando CBabel	11
2.2	Consulta SQL- <i>like</i> de alto nível	17
2.3	Descritor de recursos	18
3.1	Especificação da função utilidade	40
3.2	Um contrato do <i>framework</i> CR-RIO	44
3.3	Uma categoria do <i>framework</i> CR-RIO	45
3.4	Perfil utilizando uma função utilidade	46
3.5	Descrição da função utilidade em XML	51
3.6	Uso de entidades virtuais	54
3.7	Restrições em nível de classe	54
4.1	Descrição arquitetural e contrato CR-RIO	63
4.2	Função utilidade expressa no CR-RIO	63
5.1	Trecho do arquivo de <i>trace</i> Rodada01.trace	93

Capítulo 1

Introdução

O conceito de computação distribuída está cada vez mais difundido entre os desenvolvedores. As aplicações não são mais criadas utilizando-se apenas módulos e conectores [Corradi 2005] locais responsáveis por atender, respectivamente, seus requisitos funcionais e não-funcionais. Na verdade, as aplicações são montadas através do arranjo desses componentes que podem ter sido fabricados por diferentes empresas e executarem em diferentes nós da rede. Além disso, a aplicação pode perceber durante sua operação que um componente necessário não está disponível localmente e que, portanto, deve ser feito seu *download* do *site* do fabricante. A natureza distribuída do processamento somada ao fato do desenvolvedor não saber, em tempo de projeto, quais condições sua aplicação enfrentará durante a execução, torna sua tarefa bastante complexa.

Assim, é interessante notar que o funcionamento das aplicações distribuídas é fortemente impactado pelos recursos disponíveis no ambiente onde ela é executada e, uma vez que os recursos são compartilhados, a qualidade oferecida por elas aos seus usuários pode variar extremamente.

A “flutuação” dos recursos foi aumentada ainda mais com o advento da computação móvel. Com ela, um usuário pode, por exemplo, enviar seus e-mails a qualquer instante, seja a partir do seu *desktop* no escritório da empresa onde os recursos de processamento e comunicação são abundantes, seja portando seu computador de mão (PDA) a caminho de casa. No PDA, onde esses recursos são mais escassos, o usuário possivelmente usa uma versão mais simples do *software* que ele tem no seu *desktop*. Assim, as aplicações são obrigadas a se ajustarem de forma a não violarem os requisitos não-funcionais desejados pelo usuário.

A aplicação que leva em conta o contexto onde opera para realizar adaptações é

denominada pervasiva. Ela descobre quais recursos estão disponíveis no ambiente para o seu funcionamento, realiza seu monitoramento constantemente e se adapta de forma a maximizar a satisfação do usuário. Em [Santos et al. 2006] esse conceito é exercitado e os autores criam uma aplicação sensível ao contexto no qual ela é executada, adaptando-se, por exemplo, à movimentação do usuário pelo ambiente.

Porém, há uma outra fonte de motivos para realizar essas adaptações: as necessidades do usuário também variam ao longo do tempo. No exemplo anterior, enquanto o usuário estiver em seu escritório, ele deseja máxima *performance* para executar sua tarefa, diferentemente de quando ele porta seu PDA e abre mão da *performance* em troca da economia da energia da bateria.

Essa preocupação com as preferências do usuário e em como elas devem guiar o processo de adaptação das aplicações amplia o conceito proposto pela IBM em que os sistemas autônomos devem ser *self-configuring* (se adaptam automaticamente a ambientes dinâmicos), *self-healing* (descobrem, fazem o diagnóstico e reagem à falhas), *self-optimizing* (monitoram e se ajustam aos recursos disponíveis automaticamente) e *self-protecting* (se antecipam e se protegem de ataques) [Cheng et al. 2004, Huebscher e McCann 2004a].

Atualmente, as técnicas de adaptação embutidas nas linguagens de programação como, por exemplo o tratamento de exceções, são específicas e extremamente acopladas ao código da aplicação. Como consequência, elas têm uma grande eficiência mas impossibilitam o reuso e dificultam a separação de interesses, encarecendo a manufatura do *software*. O desenvolvedor da aplicação é responsável por garantir os requisitos funcionais e também os não-funcionais e, geralmente, acaba mesclando-os de maneira desorganizada [Garlan et al. 2004].

Por outro lado, têm surgido recentemente na literatura alternativas que propõem soluções baseadas em *frameworks*. Essa abordagem busca um mecanismo comum às diversas aplicações para implementar a adaptação: todos os componentes que visam garantir os requisitos não-funcionais são fatorados das aplicações e reunidos no *framework*. E o que é intrínseco à cada aplicação é descrito através de políticas de operação que configuram o *framework*, como será explicado adiante. A idéia é aumentar o reuso, barateando custos de desenvolvimento e desonerando o desenvolvedor da aplicação de conhecer técnicas de garantia de requisitos não-funcionais.

Com esse objetivo, foram definidos em [Loques et al. 2004] o *framework* CR-RIO (*Contractual Reflective - Reconfigurable Interconnectable Objects*) e os conceitos de contrato, níveis diferenciados de qualidade da aplicação, categorias e perfis. Resumidamente,

o CR-RIO é uma infra-estrutura que garante a imposição dos requisitos não-funcionais por meio da descrição arquitetural da aplicação e de um contrato que contém a política de operação. A descrição arquitetural informa ao CR-RIO como módulos (requisitos funcionais) e conectores (requisitos não-funcionais) são acoplados para que a aplicação seja disponibilizada. O contrato é um acordo entre as partes envolvidas: provedores e consumidores. A aplicação é oferecida em níveis diferenciados de qualidade, conforme a disponibilidade dos recursos consumidos por ela e das preferências do usuário. Para definir esses diversos níveis, os autores lançam mão de perfis de funcionamento. Eles são uma valoração das categorias que reúnem as dimensões de QoS que impactam na qualidade da aplicação. Há uma descrição detalhada na Seção 3.2 e em [Corradi 2005].

Esses *frameworks* geralmente operam ao lado da aplicação em um *loop* fechado de Monitoração-Controle-Adaptação como na Figura 1.1 [Cheng et al. 2004] [Garlan et al. 2004]. A Monitoração deve verificar a disponibilidade dos recursos do ambiente que afetam a qualidade da aplicação, como por exemplo largura de banda de um enlace de comunicação, seu atraso, a quantidade de servidores disponíveis no *cluster* e etc¹. O Controle avalia se os perfis requeridos no oferecimento da aplicação foram violados durante a sua operação, ou seja, se é necessário realizar uma adaptação. E para isso, o *framework* deve conhecer as políticas (contrato) que governam o funcionamento da aplicação. O próximo passo é especificar qual deve ser essa adaptação: quais componentes deverão receber novos parâmetros de funcionamento, quais deverão ser substituídos e etc. A última parte tem a responsabilidade, geralmente exclusiva, de executar essas adaptações, selecionando os melhores recursos ou parametrizações disponíveis para o oferecimento da aplicação no nível desejado de qualidade [Garlan et al. 2004]. Para auxiliar o *framework* nessa tarefa, este trabalho propõe o uso de funções utilidade, como será argumentado mais adiante.

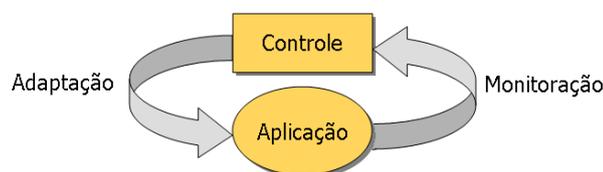


Figura 1.1: *Loop fechado dos frameworks*

Conforme dito anteriormente, é necessário que seja informada ao *framework* uma descrição de como a aplicação deve ser montada (quais componentes - módulos e co-

¹Definir quais são essas dimensões de QoS impactantes é uma tarefa muito importante, de responsabilidade do desenvolvedor pois depende do seu conhecimento a respeito do serviço sendo oferecido (*service-specific knowledge* [Huang 2004]).

nectores - são necessários para o seu estabelecimento), ou seja, sua descrição arquitetural. Trata-se de uma descrição de alto nível que dá ao desenvolvedor a possibilidade de postergar preocupações de mais baixo nível tais como: a localização os recursos necessários e qual sua disponibilidade, por exemplo. Com isso, o *framework* pode suportar o desenvolvedor não só na manutenção da qualidade da aplicação em operação [Capra et al. 2005, Poladian et al. 2006, Molina-Jimenez et al. 2004], mas também no momento de oferecê-la pela primeira vez [Huang 2004, Ranganathan et al. 2005]. Mais detalhes são dados nas Seções 2.3 e 2.4. Essa forma de compor as aplicações se ajustará perfeitamente ao ambiente *web*, onde elas deverão ser montadas no futuro pela interligação de diversos componentes, de procedência e localizações diferentes. Além disso, a descrição arquitetural fornece um conhecimento amplo (não-pontual) da composição da aplicação e permite ao *framework* otimizar sua qualidade globalmente, conforme será mostrado na Seção 3.1.

Sumarizando, para possibilitarmos a adaptação de aplicações, um conjunto de questões deve ser resolvido, a saber:

- deve ser possível a representação da aplicação em termos arquiteturais (descrição arquitetural), ou seja, o desenvolvedor deve informar ao *framework* como ele deseja que os serviços por ela oferecidos sejam montados;
- o *framework* deve conhecer as necessidades do usuário, em outras palavras, deve saber como adaptar a aplicação de forma a melhor satisfazer seus requisitos não-funcionais. Elicitar essas preferências tem se mostrado complexo na literatura;
- o *framework* deve conhecer os recursos exigidos pela aplicação (perfis) e como eles afetam sua qualidade. Além disso, deve conhecer a política (contrato) que impõe os níveis diferenciados de qualidade da aplicação;
- o *framework* deve dispor de uma ferramenta representativa e computacionalmente barata para expressar as preferências do cliente e a disponibilidade dos recursos. Esta ferramenta tem o objetivo de selecionar os melhores recursos que irão compor a aplicação, realizando essas adaptações sem introduzir um *overhead* de processamento considerável;
- a dinâmica da variação da disponibilidade dos recursos é geralmente alta e frequentemente há a presença de ruído e de outras imprecisões inerentes às medições. O *framework* deve ter um meio robusto de monitorar essa variação;

- uma vez que a aplicação pode ter clientes com diferentes preferências, o *framework* deve oferecer o nível de qualidade (dentre os especificados no contrato pelo desenvolvedor da aplicação) que mais satisfaça ao cliente; e
- quando uma violação de perfil é verificada e um novo nível de qualidade da aplicação deve ser imposto, é importante que o *framework* produza adaptações que não sejam conflitantes entre si.

Este trabalho objetiva relacionar questões que devem ser resolvidas para suportar a adaptação de aplicações pervasivas, dentre elas: descrição da arquitetura da aplicação, descoberta de recursos no ambiente, monitoração, elicitación das preferências do usuário e, principalmente, a escolha do melhor recurso disponível. Buscando resolver essa última questão, propomos - como principal contribuição deste trabalho - o emprego de uma função utilidade por entender que ela constitui uma técnica que leva em conta as preferências dos usuários (que ponderam dimensões de QoS frequentemente conflitantes) e a disponibilidade dos recursos que impactam a qualidade da aplicação. Outra virtude dessa técnica é a facilidade de implementação e a pouca quantidade de processamento exigida, propiciando ser utilizada em pequenos dispositivos com recursos computacionais e de energia restritos, tais como PDAs. Para validar nossa proposta, apresentamos alguns exemplos de aplicação onde empregamos uma função utilidade para realizar a escolha do melhor recurso. Nos concentramos no exemplo em que um cliente escolhe o melhor AP de uma rede sem-fio para estudar como é montada a função utilidade neste caso.

O restante desse trabalho está dividido assim: no Capítulo 2 iremos expor os conceitos e o vocabulário que o leitor deve conhecer antes de ler os capítulos seguintes. No Capítulo 3 mostramos pesquisas importantes, as soluções que elas propõem e que serviram de impulso para este trabalho. No Capítulo 4 listamos exemplos de aplicação onde propomos empregar uma função utilidade para realizar a escolha da melhor alternativa de adaptação. Para validar a qualidade e a robustez da nossa proposta, no Capítulo 5 estudamos em detalhes o exemplo da escolha do melhor AP. Analisamos *traces* reais do tráfego de uma rede sem-fio e *traces* sintetizados por um simulador construído e parametrizado com distribuições realísticas de probabilidade. No Capítulo 6 encerramos este trabalho citando as conclusões que depreendemos e sugerimos novos estudos que podem ser feitos a partir deste.

Capítulo 2

Conceitos preliminares

Neste capítulo iremos expor os conceitos e o vocabulário que o leitor deve conhecer antes de ler os próximos capítulos.

2.1 Razões para a adaptação dinâmica

Aplicações adaptativas são divididas em duas grandes categorias [Poladian et al. 2006, Huebscher e McCann 2004b]. A primeira é populada por sistemas tolerantes à falhas, que reagem ao mau funcionamento de componentes usando técnicas tais como redundância e *graceful degradation*. Tais aplicações são comuns em áreas críticas, onde o custo para o reparo *off-line* é proibitivo ou as aplicações simplesmente não podem parar de operar: telecomunicações, controle aéreo, plantas industriais e de energia e etc. Por exemplo, uma aplicação é hospedada em um *cluster* de N servidores para atender a demanda percebida em condições normais. Para suportar picos de demanda, os projetistas adicionam estaticamente um servidor extra, de modo a prover capacidade de reserva. Porém, o custo dessa solução é um problema: reservar um servidor, permanentemente, implica em uma despesa que muitas empresas não aceitam.

A segunda categoria é composta de aplicações que reagem à variação de recursos, se adaptando de modo a funcionar da melhor forma possível com os recursos disponíveis: CPU, largura de banda dos enlaces de comunicação, energia das baterias e etc. Muitas delas advêm da área de computação móvel visto que esse tipo de computação tornou a dinâmica dos usuários e da disponibilidade dos recursos requeridos ainda maiores. Nesse tipo de cenário, é inviável para o desenvolvedor saber, antes da imposição da aplicação, quais serão as condições de operação no ambiente onde ela será oferecida. Seria interessante se o desenvolvedor conhecesse, *a priori*, quantos acessos simultâneos os clientes

vão demandar, qual será o tipo de requisição mais freqüente dos seus clientes, se eles vão requerer em sua maioria conteúdo estático ou dinâmico (se for o caso de uma aplicação *web*), qual a largura de banda dos enlaces de comunicação, se o servidor vai ou não se degradar em função de problemas de *hardware* e etc.

Essa questão tem um enfoque interessante em [Pinheiro et al. 2003] onde, de acordo com a demanda, ajusta-se o número de servidores ligados em um *cluster* para economizar energia. Naquele trabalho, os autores lembram que manter servidores extras implica em gastos adicionais com *hardware*, licenças de *software*, contratos de manutenção, energia, ar condicionado e seguro, entre outras. A solução proposta cria um mecanismo que, em caso de um pico de demanda, balanceia a carga entre todos os servidores do *cluster* e, nos momentos que o sistema experimentar uma baixa demanda, a carga é concentrada em um número reduzido de servidores de forma que os outros do *cluster* possam ser desligados. Mesmo tendo a preocupação de garantir alguns requisitos de QoS, tal como o tempo de resposta, o trabalho mostra que é possível uma economia de até 86% no consumo de energia.

Em [Norris et al. 2004] uma abordagem mercadológica interessante foi aplicada para ajustar o número de servidores dinamicamente. Lá, os autores modelam o problema de forma que várias aplicações são executadas conjuntamente em um *cluster* e compartilham recursos limitados, como por exemplo, o número de servidores disponível para elas. Antes do início do funcionamento do *cluster*, o administrador dá a cada aplicação um “orçamento” e um certo número de servidores de modo que o somatório desses servidores seja menor ou igual ao total disponível no *cluster*. Quando as aplicações são iniciadas, elas monitoram se o número de servidores que elas têm para utilizar é suficiente. A cada intervalo de tempo τ , as aplicações que perceberam um número insuficiente de servidores têm a oportunidade de comprar servidores daquelas que perceberam que possuem uma quantidade supérflua de servidores. O preço dos servidores é negociado a cada iteração em um leilão do qual participam todas as aplicações com o objetivo de reservar os recursos do ambiente. Veja que mesmo aquelas aplicações que perceberam que têm um número ideal de servidores para funcionar podem participar do leilão: pode ser que elas estejam prevendo um aumento ou uma diminuição de carga no futuro. O trabalho ressalta que essa abordagem mercadológica pode ser sabotada se aplicações maliciosas usarem o leilão para fazer especulação: dependendo da política de negociação elas podem, por exemplo, oferecer ao mercado mais servidores do que elas realmente querem vender apenas para que outras aplicações se beneficiem de uma baixa nos preços.

Em um outro exemplo, temos o cenário descrito na Figura 2.1 onde um cliente está na iminência de enviar sua requisição ao *cluster*. Ele gostaria que seu pedido fosse redirecionado para o melhor servidor, no qual “melhor” é um conceito complexo. A parte simples desse conceito diz respeito às métricas que se pode coletar do ambiente. Algumas delas são a quantidade de memória e de capacidade de processamento livres em cada servidor, além da largura de banda disponível nos enlaces de comunicação. Um algoritmo executado no *web switch* poderia manter essas informações e usá-las para redirecionar as chamadas aos servidores. Essa técnica sensível ao contexto dos servidores já melhora a eficiência do sistema em comparação quando o *cluster* utiliza mecanismos simples de distribuição de carga: aleatórios ou baseados em *Round Robin*. Porém, numa situação típica, os pedidos enviados podem ser diferentes entre si. Os usuários podem solicitar páginas *web* estáticas, dinâmicas, com multimídia, consultas ao banco de dados ou complexos cálculos matemáticos. Portanto, a parte complexa do conceito de “melhor” servidor advém do fato de que, dependendo da requisição, as métricas coletadas devem ter pesos diferenciados nessa escolha. Os algoritmos de redirecionamento têm que ser sensíveis ao contexto, em outras palavras, eles devem analisar a requisição HTTP proveniente do cliente, o que reduz extremamente sua escalabilidade [Cardellini et al. 2002] (ver Seção 4.2).

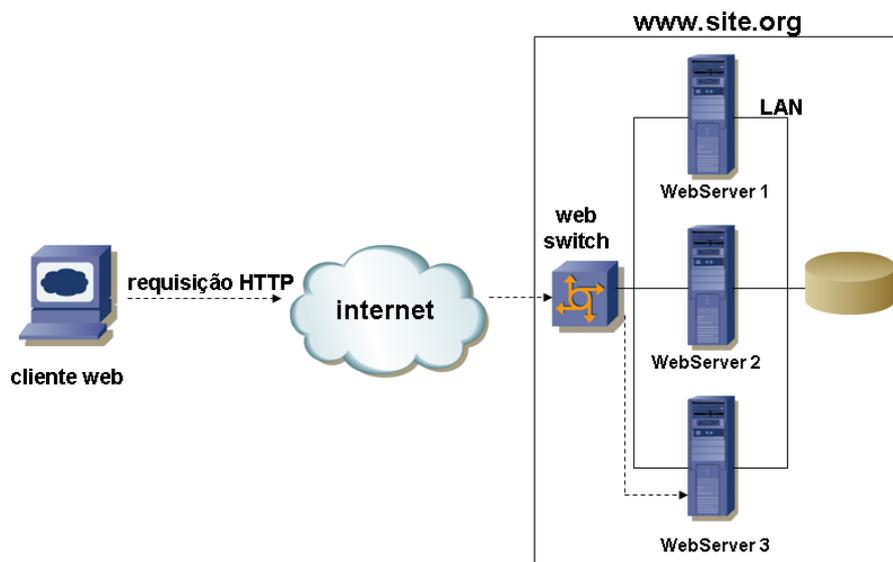


Figura 2.1: *Cluster típico de servidores*

Entretanto, mesmo esses algoritmos mais complexos de redirecionamento podem não satisfazer totalmente ao usuário. Por exemplo, o usuário solicita assistir um vídeo que é uma tarefa consumidora principalmente de recursos de comunicação. Como esses recursos são caros, o usuário gostaria de assistir com alta qualidade apenas os vídeos de esportes, enquanto vídeos do noticiário local ele aceita assistir com menor qualidade para poupar

esses recursos. Assim, duas requisições (por um fluxo de vídeo) similares devem ser tratadas diferentemente pela aplicação. Como medida adaptativa, a aplicação pode instalar um *codec* de vídeo entre a fonte e o cliente, propiciando economizar os recursos de comunicação. Porém, como a aplicação poderia perceber a necessidade de instalar um *codec* de vídeo? Uma alternativa é permitir que o usuário, conhecedor do tipo de tarefa que está prestes a submeter à aplicação e conhecedor do tipo de qualidade esperada, expresse suas preferências, de modo que a aplicação ofereça essa qualidade.

Desta forma, fica claro que para realizar adaptações dinâmicas, a aplicação deve ser sensível ao contexto, ou seja, ser sensível às variações dos recursos do ambiente importantes para o seu estabelecimento e às preferências do seu usuário; deixando claro que o ambiente e o usuário podem ser considerados atores desse processo.

2.2 Características desejáveis em um *framework*

A adaptação deve ser implementada externamente a aplicação através de um *framework*. Com isso, temos a possibilidade de reutilizar a infra-estrutura entre diferentes aplicações, barateando o custo final de desenvolvimento e facilitando a separação dos interesses: o desenvolvedor se preocupa em interligar os componentes de forma a oferecer os requisitos funcionais de sua aplicação e o *framework* cuida de garantir os requisitos não-funcionais (as preferências do usuário são expressas pelos perfis e as políticas que guiam a adaptação são expressas através do contrato que configura o *framework*), ver Seção 2.3 e [Corradi 2005]. A literatura mostra alguns compromissos que idealmente devem ser cumpridos para que se tenha uma boa solução baseada em *framework* [Othman et al. 2004]:

- **Propósito geral:** o *framework* não deve fazer em seu código nenhuma hipótese sobre as aplicações que ele gerencia, reduzindo assim o acoplamento e permitindo o reuso. Ele deve obter o conhecimento sobre o funcionamento das aplicações e sobre as políticas de adaptação na forma de arquivos de configuração. Essas políticas devem guiar o mecanismo de adaptação, garantindo a reusabilidade.
- **Requerer pequena ou nenhuma mudança na aplicação:** se a aplicação tiver de ser alterada para a interação com o *framework*, este é um indício de que os dois estão acoplados uma vez que novas alterações em um poderão interferir no outro. Atualmente as aplicações expõem suas funcionalidades através de APIs que podem ser acessadas pelo *framework*. O problema reside nas aplicações legadas que não foram projetadas para interagir com sistemas externos. Uma alternativa

é a utilização de *wrappers* para o interfaceamento entre o *framework* e a aplicação [Poladian et al. 2006].

- **Transparência:** do ponto de vista do usuário, ele não deve perceber que a aplicação está sendo gerenciada por um *framework*, e se ele é local ou remoto. Isso sugere também que o *overhead* de processamento causado pelo *framework* seja mínimo.
- **Extensibilidade:** o *framework* deve conter elementos genéricos - *hotspots* - que são especializados caso a caso para que ele possa interagir com a aplicação.
- **Manutenibilidade:** o *framework* deve ser composto de componentes com responsabilidades bem definidas, o que facilita sua manutenção.

Essas características são gerais aos *frameworks* e serão particularizadas para a tarefa de adaptação de sistemas ao longo desse trabalho.

2.3 Descrição arquitetural

As soluções presentes na literatura têm mostrado uma forte tendência em fornecer ao *framework* a descrição arquitetural do serviço [Corradi 2005, Huang 2004, Garlan et al. 2004]. Essa abordagem confere vantagens importantes.

A descrição arquitetural evita que o desenvolvedor da aplicação se preocupe precocemente com detalhes de implementação. Ela é uma estrutura que reúne os componentes da aplicação e suas interligações, oferecendo ao *framework* uma visão global. É complementada por um conjunto de políticas (contrato) e de restrições que dirigem o funcionamento da aplicação e sua evolução ao longo do tempo. Utilizando a descrição arquitetural, permite-se descobrir estilos arquiteturais comuns entre si e, conseqüentemente, reusar mecanismos entre aplicações que compartilhem o mesmo estilo [Cheng et al. 2006]. É importante notar que a descrição arquitetural está ligada aos requisitos funcionais, inegociáveis; enquanto o contrato está ligado aos requisitos não-funcionais, negociáveis.

Em outras palavras, antes do estabelecimento da aplicação, a descrição arquitetural permite que o desenvolvedor se ocupe apenas em compô-la interligando componentes. Nesse momento, ele está preocupado apenas com os requisitos funcionais da aplicação: selecionar quais os componentes, de forma abstrata, são necessários e qual arquitetura empregar para que a funcionalidade esperada seja entregue. Essa visão pode ser comum no futuro quando as aplicações serão montadas através de vários componentes distribuídos na Internet. A localização dos componentes, suas disponibilidades e a maneira

como a aplicação deve reagir quando esses recursos variarem não devem fazer parte das preocupações do desenvolvedor nesse estágio inicial e podem ser postergadas para outro momento.

Quanto à portabilidade, se pensarmos que a mesma aplicação pode ser oferecida em diferentes ambientes - como por exemplo o envio de mensagens eletrônicas no *desktop* e no PDA do usuário - então a mesma descrição arquitetural poderá ser reusada nesses vários ambientes e caberá ao *framework* impor a aplicação de forma diferenciada, em cada plataforma, com os recursos disponíveis. O desenvolvedor é desonerado em um primeiro momento de conhecer detalhes de implementação e políticas de cada ambiente, podendo concentrar-se nos requisitos funcionais do serviço [Ranganathan et al. 2005].

Dentre os possíveis estilos arquiteturais, descrevendo as aplicações segundo uma arquitetura cliente-servidor, tem-se uma clara separação entre os componentes que provêm as funcionalidades e aqueles que as utilizam. Segundo [Corradi 2005], outra classificação possível que favorece a separação de interesses, é quanto à implementação dos requisitos: os componentes no CR-RIO são identificados como módulos (elementos **Cliente** e **Servidor** da Figura 2.2), que implementam os requisitos funcionais do serviço; e conectores (elemento **C-S** na mesma figura), que devem suportar os requisitos não-funcionais, tais como segurança, criptografia, *logging* e etc. Conforme dito anteriormente, apenas os requisitos não-funcionais são negociáveis e, por isso, serão o alvo da adaptação promovida pelo *framework*.

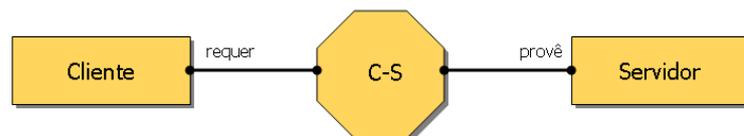


Figura 2.2: Descrição de um serviço simples baseado na arquitetura cliente-servidor

A forma de realizar a descrição arquitetural é encontrada de forma variada na literatura. Alguns trabalhos utilizam arquivos-texto para fornecer essa meta-informação ao *framework*, através de ADLs (*Architecture Description Languages*), como a CBabel por exemplo [Corradi 2005]. Essa opção geralmente usa uma representação mais próxima ao ser humano, mais fácil de ser entendida, verificada e de ser usada pelo desenvolvedor, às custas da implementação de um *parser* desses arquivos. O Código 2.1 exemplifica o uso dessa ADL na descrição do serviço mostrado na Figura 2.2. Pode-se verificar a definição e a instanciação dos módulos e do conector e também a forma como eles são conectados de forma que a aplicação seja estabelecida.

Código 2.1: Descrição arquitetural utilizando CBabel

```

module ClienteServidor {
  port requer, prover;
  module Cliente {
    out port requer;
5  } cliente;
  module Servidor {
    in port prover;
  } servidor;
  connector C-S {
10  in port requer;
    out port prover;
  } cs;
  instantiate servidor at HostServidor;
  instantiate cliente at HostCliente;
15 link cliente to servidor by cs;
} cliente_servidor;

```

Outros trabalhos utilizam-se de uma API (ver Tabela 2.1) exposta pelo *framework* para fazer essa descrição em “receitas” [Huang 2004]. Descrevendo a aplicação através de uma linguagem de programação, como por exemplo Java, os desenvolvedores podem realizar seu trabalho em ambientes integrados de desenvolvimento (IDEs - *Integrated Development Environment*), ver Código 3.1.

Métodos	
addComp(spec)	Adiciona um componente abstrato com a especificação dada à configuração abstrata
addConn(c1, c2)	Adiciona uma conexão entre os componentes c1 e c1 à configuração abstrata
addSubComp(n, spec)	Adiciona n componentes idênticos com a especificação spec a um dado componente
getProperty(prop)	Obtém o valor da propriedade prop de um dado componente

Tabela 2.1: Alguns métodos presentes na API

Entre a descrição arquitetural de uma aplicação e o seu estabelecimento propriamente dito, uma importante questão deve ser resolvida: quais instâncias de componentes estão disponíveis (ou devem ser instaladas) no ambiente a fim de proverem as funcionalidades desejadas pelo desenvolvedor, quando do momento da especificação da aplicação? A opção por utilizar a descrição arquitetural do serviço foi adotada para desonerar o desenvolvedor da preocupação de definir, em tempo de projeto, quais instâncias iriam efetivamente executar a aplicação em um dado ambiente, aumentando a portabilidade. Porém no

momento do estabelecimento da aplicação, essa questão deve ser resolvida (ver Seção 2.4).

Essa técnica é freqüentemente referida como virtualização de componentes: o desenvolvedor informa ao *framework* a arquitetura que deverá ser imposta sem que seus componentes apontem diretamente para recursos reais existentes no ambiente. Ela é interessante pois permite postergar essa resolução para o momento mais propício: a hora do estabelecimento da aplicação quando o *framework* tem detalhes do ambiente onde ela será executada. Essa indireção facilita o trato do problema da heterogeneidade e da dinâmica dos ambientes: heterogeneidade porque uma tarefa deve ser executada em diversos ambientes e dinâmica porque os recursos são compartilhados com freqüência.

Em resumo, essa distinção é feita para chamar atenção de que na descrição arquitetural, usando uma terminologia de orientação a objetos, o desenvolvedor escolhe as classes que irão compor a aplicação, deixando a cargo do *framework* descobrir quais as instâncias irão efetivamente executá-la. Como vantagem, a cada estabelecimento da aplicação (e durante sua operação) o *framework* tem a oportunidade de montá-la da maneira que mais satisfaça ao usuário.

2.4 Mapeamento físico de componentes

Uma vez que o desenvolvedor, em tempo de projeto, definiu apenas quais os componentes necessários para o estabelecimento da aplicação e suas interligações (topologia), ele postergou a decisão de escolher quais as instâncias desses componentes efetivamente vão executar a aplicação. Isso é necessário porque o desenvolvedor não sabia, em tempo de projeto, quais condições vão existir para o estabelecimento da aplicação: em qual ambiente ela será executada, quais serão os recursos disponíveis e quais serão os requisitos não-funcionais esperados pelo usuário. Na verdade, o ambiente no qual a aplicação vai ser estabelecida pode oferecer várias instâncias alternativas para cada um dos componentes necessários, citados na descrição arquitetural¹. Essa seleção deve ser realizada e é chamada freqüentemente de mapeamento físico [Huang 2004].

É interessante notar que a partir da descrição da aplicação, o desenvolvedor não vai mais interagir com o *framework*, o que significa que o *framework* deve conter todas as informações necessárias para realizar o mapeamento físico e ser capaz de levantar as informações que ele não tiver. Essas informações em tempo de execução vêm de quatro

¹Não estamos interessados neste momento em mecanismos de reserva de recursos.

fontes: do serviço de descoberta de recursos disponíveis, do serviço que monitora esses recursos [Cardoso et al. 2006], do desenvolvedor (que define quais dimensões de QoS impactam na qualidade da aplicação) [Huang 2004] e das preferências do usuário em questão, previamente elicitadas. No CR-RIO, a especificação das dimensões de QoS impactantes se dá através das categorias enquanto que as preferências dos usuários são guardadas nos perfis [Corradi 2005].

Sejam dois cenários onde esse mapeamento deve ser realizado, ou seja, o *framework* deve escolher, dentre as alternativas disponíveis, aquela que irá suportar a descrição arquitetural da aplicação (instância de contrato). No primeiro, o desenvolvedor especificou que a transmissão de mensagens entre dois componentes deve ser encriptada. Através de uma técnica específica de descobrimento de recursos, abordada na Seção 2.5, o *framework* fica ciente que existem vários componentes disponíveis no ambiente com a funcionalidade de encriptação, caracterizados por dimensões de QoS tais como *performance* e custo.

No segundo cenário, já citado na Seção 2.1, um *codec* deve ser utilizado segundo a descrição arquitetural de uma aplicação de transmissão de vídeo. Mesmo na hipótese de existir apenas uma instância disponível no ambiente, essa instância pode ser configurada (parametrizada) de várias formas diferentes. Por exemplo, o vídeo pode ser transmitido com diferentes taxas de compressão, quadros por segundo, tamanho de quadro, qualidade de áudio e número de cores.

Assim, o *framework* deve saber como realizar essa seleção, sempre tentando maximizar a satisfação do usuário. Porém, uma vez que a aplicação é composta geralmente por vários componentes, pode-se verificar facilmente que o número de combinações possíveis cresce não-polinomialmente, podendo atingir tipicamente a ordem das dezenas ou centenas de milhares [Poladian et al. 2006]. Portanto, é claro que uma busca exaustiva do espaço de soluções pela alternativa que mais satisfaça ao usuário é inviável. Afinal, um dos critérios que o desenvolvedor pode usar para decidir qual *framework* utilizar para publicar sua aplicação é o tempo gasto para o seu estabelecimento. Por se tratar de um dos objetivos deste estudo, o mapeamento físico é descrito em maiores detalhes no Capítulo 3.

2.5 Descoberta de recursos

Para que uma aplicação pervasiva possa se adaptar à variação da disponibilidade dos recursos, é essencial que ela conheça a todo instante de quais recursos ela pode lançar mão para promover adaptações que aumentem a qualidade percebida pelos usuários

[Cardoso et al. 2006]. Para resolver esse problema, o *framework* deve poder consultar um serviço em que estejam descritos os recursos possíveis de serem utilizados e, também, precisa saber como estão valorados os atributos desses recursos. Nesta seção, vamos cuidar da primeira parte desse problema: vamos estudar algumas soluções que organizam e descrevem os recursos. Na seção seguinte, vamos estudar os detalhes envolvidos na monitoração dos atributos de um recurso.

Além de soluções de propósito geral para descrever e organizar recursos em diretórios, tais como o JNDI da comunidade Java [JNDI] ou o utilizado em [Huebscher e McCann 2004a]², encontramos na literatura técnicas otimizadas para aplicações pervasivas.

Em [Judd e Steenkiste 2003] vemos uma iniciativa em que um componente chamado *Context Information Service* (CIS) fornece informações sobre o contexto onde uma aplicação opera através de um banco de dados virtual. É mostrado que há a necessidade de desenvolver um repositório único que contenha informações sobre o ambiente para que as aplicações sejam desoneradas de construir, de forma *ad hoc*, os seus próprios sistemas de contexto; uma situação que levaria a sistemas de informação com várias interfaces distintas e que, por isso, não poderiam inter-operar.

Os autores comentam a possibilidade de manter informações sobre o contexto em um banco de dados tradicional: a tecnologia é bastante madura e os clientes poderiam ser leves pois fariam suas consultas através de uma linguagem padronizada, tal como o SQL. Porém as informações sobre o contexto são bastante dinâmicas e as informações frequentemente possuem meta-atributos associados a elas, tais como quando as informações foram coletadas e com que precisão isso aconteceu. Ainda segundo os autores, é importante que os clientes possam especificar esses meta-atributos em suas consultas, por exemplo: um cliente gostaria de saber a localização de uma pessoa com uma precisão de até 50 metros de raio. Bancos de dados típicos não são projetados para manter essas informações.

Assim, o trabalho sugere a criação de uma base de dados virtual em que a informação é guardada e coletada sob demanda, através da consulta de vários sensores, ou *Context Information Providers* (CIPs). Os clientes continuariam sendo leves pois a consulta a essa base de dados virtual seria feita através de uma linguagem *SQL-like* e a solução não teria as desvantagens de um banco de dados tradicional. Cada CIP estaria livre para ter a sua

²Naquele trabalho, o autor define que cada recurso deve se cadastrar no *Directory Service* (DS) quando fica ativo. Esse cadastro prevê o envio de um descritor que contém os atributos e os serviços que o recurso pode executar. Além disso, o recurso envia um sinal de *heartbeat* a cada intervalo de tempo, previamente acordado, para que o DS saiba que ele está operando.

própria implementação: se ele tratar de informações muito dinâmicas, então possivelmente seja mais interessante levantá-las sob demanda, caso contrário, a implementação pode ser através de um banco de dados tradicional; em outras palavras, cada CIP só implementa a parte da interface que lhe diz respeito. Assim, o objetivo do CIS é obter as informações de vêm de diversos CIPs diferentes, de diferentes implementações, e reuni-las para que sejam consultadas pelos clientes, de forma transparente e homogênea.

São definidas quatro entidades distintas, sobre as quais um cliente pode fazer consultas, tal como em um banco de dados típico, a saber: **Devices**, **Networks**, **People** e **Areas**. São definidos também alguns relacionamentos entre essas entidades, como mostra a Figura 2.3.

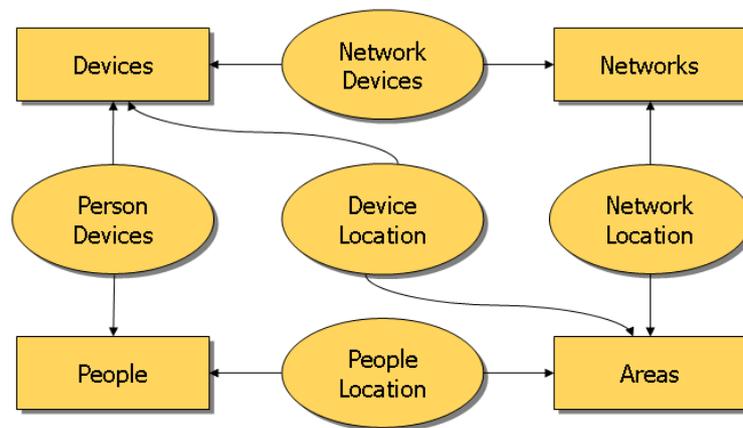


Figura 2.3: *Entidades do banco de dados virtual*

A entidade **Devices** basicamente é a classe de todo objeto sobre o qual um cliente gostaria de obter informações: veículos, impressoras, computadores e etc. A entidade **People** receberá as pessoas, usuárias do sistema ou não. A entidade **Areas** contém elementos de diversas dimensões tais como uma sala ou um *campus* de uma faculdade. A última entidade, **Networks**, foi colocada ali pois o autor percebeu a necessidade natural que esses elementos têm de se comunicar, e isso se dá através de redes, com a única exceção entre a entidade **People** e **Networks**, como denota a falta de um relacionamento entre elas.

Sobre a interface *SQL-like* utilizada pelos clientes para fazer consultas ao CIS, o artigo estabelece as seguintes partes que podem ser definidas em uma consulta:

- **selectedAttributes**: representa uma lista de atributos que devem ser retornados pela consulta. Corresponde à parte **select** de uma consulta SQL.
- **providerNames**: representa os nomes dos CIPs que devem ser consultados. Corresponde à parte **from** de uma consulta SQL. Esse item é importante pois sinaliza ao

CIS que o cliente deseja que ele sintetize as informações de várias fontes antes de retornar a resposta.

- **selectionExpression**: permite criar relações entre os objetos retornados e filtros para que seu número seja diminuído. Corresponde à parte **where** de uma consulta SQL. Neste campo o cliente pode especificar, por exemplo, “`person.name='john'`”.
- **attributeReqs**: reservado para que o cliente defina os meta-atributos que devem nortear a consulta. Por exemplo, ele pode definir que os dados devem ter sido levantados à no máximo 30 minutos. Esse atributo permite que os CIPs realizem *cache* interno, aumentando sua *performance*.
- **timeLimit**: especifica o máximo que o cliente deseja esperar pela resposta. Esse parâmetro pode ser visto pelo CIP como uma dica de quanto esforço ele pode empenhar nessa tarefa.

Um exemplo de consulta utilizando essa interface pode ser vista no Código 2.2 (sob o ponto de vista da implementação ela é mantida em um documento XML transportado entre os clientes e o CIS através de HTTP). No código, um cliente deseja saber a localização de *John*. Para isso, o sistema deve acessar o provedor `PersonLocation`. O cliente deseja receber uma informação que tenha sido coletada há, no máximo, 2 minutos e com uma resolução/precisão de 500 metros. Para isso, o cliente está disposto a esperar 1 minuto no máximo.

Código 2.2: Consulta *SQL-like* de alto nível

```
Select location
from PersonLocation
where PersonID = John's UID
require
5 location.updateTime within 2 minutes of present time
  location.accuracy within 500 meters of actual location
  timeLimit 1 minute
```

Uma outra abordagem é utilizada em [Capra et al. 2005]. Naquele trabalho os autores se preocupam em descrever três elementos distintos, a saber: serviços, que são oferecidos por provedores; sensores, que são responsáveis por levantar informações sobre o ambiente e componentes, que podem ser trazidos de um nó remoto da rede para o nó onde o usuário está conectado antes de serem utilizados.

Além de uma URI única (do tipo `//host/displayVideo`) cada recurso é definido através de um descritor de recursos, que para uma tela de vídeo tem o aspecto mostrado no Código 2.3.

Código 2.3: Descritor de recursos

```
(component , displayVideo)
(code , display800600.jar)
(resolution , 800x600)
(version , 2.1)
5 (platform , JVM2)
(size , 70kB)
(cost , $10)
(memory , 2)
(battery , 4)
```

Podemos verificar que o recurso é um componente do tipo tela de vídeo e que tem entre outros atributos, uma resolução máxima de 800x600 *pixels*. É interessante notar que estão valorados os recursos que este componente consome: se instalado, esse componente consome 2 unidades de memória e 4 unidades de bateria. O artigo explica que estes valores estão normalizados entre 0 e 10 e servem apenas para a comparação entre recursos.

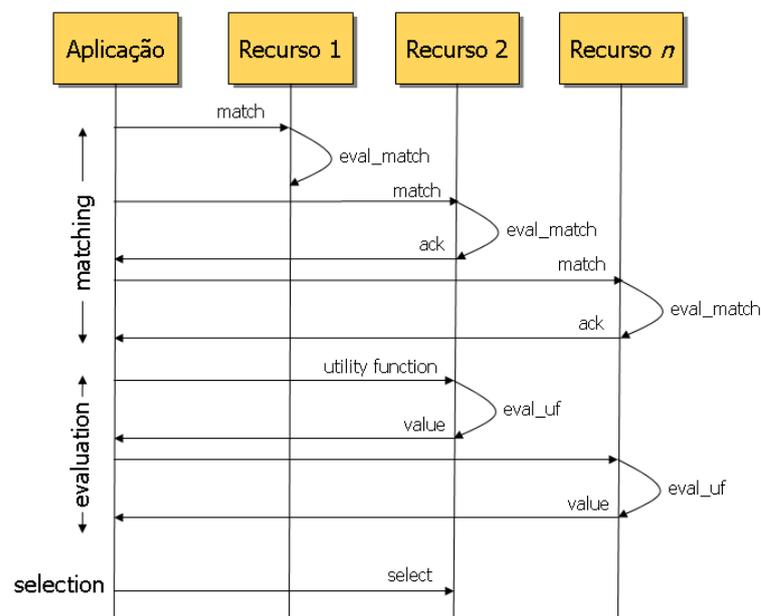


Figura 2.4: Protocolo de descoberta de recursos

Uma vez que cada recurso está descrito de uma forma padronizada, o artigo mostra conceitualmente um protocolo de descobrimento de recursos baseado em *polling* (ver Figura 2.4). Na primeira fase (*matching*), o *framework* envia uma solicitação contendo

a descrição do recurso necessário. O artigo sugere que essa solicitação seja enviada em *broadcast* para os recursos do ambiente. Cada recurso irá confrontar a solicitação com seu próprio descritor de recursos e apenas aqueles que passarem nesse teste responderão à solicitação, em um caráter eliminatório. Na segunda fase (*evaluation*), os recursos que responderam à solicitação operam uma função utilidade enviada pelo *framework* sobre seu descritor de recurso, retornando então um número escalar, em um caráter classificatório. O artigo prevê que, caso o componente não possa calcular a função utilidade porque, por exemplo, não tem recursos computacionais, então ele deve retornar seu próprio descritor para que a função utilidade seja calculada pelo *framework*. No terceiro e último passo (*selection*), o protocolo de escolha de recurso prevê o caso em que os recursos são muito importantes para o funcionamento da aplicação ou demasiadamente caros do ponto de vista monetário, então o *framework* solicita a intervenção humana.

2.6 Monitoração de recursos

Os *frameworks*, tal como descritos no Capítulo 1, operam em um *loop* fechado de Monitoração-Control-Adaptação. Assim, na base da tarefa de adaptar dinamicamente as aplicações deve estar um serviço de monitoração capaz de oferecer dados com qualidade. É importante lembrar que é responsabilidade do desenvolvedor informar ao *framework* quais recursos, e mais especificamente, quais atributos desses recursos impactam na qualidade da aplicação entregue e, que portanto, devem ser monitorados para que os requisitos não-funcionais possam ser garantidos pelo *framework* [Huang 2004]. Em [Corradi 2005], por exemplo, essa especificação é feita através das *QoSCategories* como será visto na Seção 3.2. O *framework* então deve preocupar-se em como medir esses atributos e de fazê-lo com qualidade.

2.6.1 Meta-dimensões de monitoração

Para cada dimensão de QoS sendo monitorada existe um conjunto de meta-dimensões que também podem ser fornecidas pelo serviço de monitoração. A natureza dinâmica dos ambientes, principalmente depois do advento da computação móvel, insere incertezas a essas medições. Características tais como precisão, probabilidade de corretude, resolução, idade da informação e taxa de atualização são intrínsecas à metodologia de medição e fazem parte da meta-informação sobre as dimensões de QoS. Dessa forma, elas podem ser levadas em conta no processo decisório do *framework*, ou seja, podem entrar na com-

posição da função utilidade que escolhe o melhor recurso [Huebscher e McCann 2004a]. Por exemplo, entre dois enlaces de comunicação com mesma capacidade disponível, o *framework* pode escolher aquele que é monitorado mais freqüentemente.

Como vimos na seção anterior, essas meta-dimensões são levadas em conta no trabalho de [Judd e Steenkiste 2003] através da parte `attributeReqs` da interface de consulta de recursos. Naquele trabalho, os autores consideram um serviço de monitoração que possa ser acessado pelos clientes através de uma interface *SQL-like*, tal como mostrado no Código 2.2. Um banco de dados virtual atuaria como um *cache* das informações monitoradas e seria capaz de resolver consultas de alto nível.

Dessa forma, um cliente poderia enviar uma consulta do tipo “João está no prédio?” que deveria ser resolvida pelo serviço de monitoração de forma diferente da pergunta “João está na sala 160?”. Embora a resposta nos dois casos seja *booleana*, no primeiro caso freqüentemente o sistema já terá no seu repositório dados suficientes para responder, enquanto que no segundo caso ele necessitará de informações especializadas e, possivelmente devido à dinâmica do ambiente, sensores terão que ser consultados sob demanda. Assim, o serviço poderia sintetizar informação e não ficaria restrito a um banco de dados estático.

Ainda naquele trabalho, é colocada a preocupação de que existe um custo inerente ao processo de obter e tratar informações sobre o ambiente em que a aplicação será imposta. Espalhar sensores (*probes*) demasiadamente pelo ambiente no intuito de conhecê-lo melhor pode levar a dois problemas. O primeiro diz respeito à dificuldade de processar todas essas informações, reduzindo a escalabilidade da solução e, o segundo, advém do fato que algumas dessas informações podem ser conflitantes e discrepantes, uma vez que são tomadas por sensores de diferentes tecnologias e de diferentes fabricantes.

Os trabalhos de [Huebscher e McCann 2004a] e [Cheng et al. 2004] se concentram em escolher no ambiente o melhor provedor de um determinado tipo de informação e, para isso, se valem das meta-dimensões citadas anteriormente. No segundo trabalho, os autores desejam fazer uma seleção entre provedores que fornecem a localização de uma pessoa em um ambiente (uma casa por exemplo) através de uma função utilidade. Porém ao invés de os provedores de informação responderem ao *framework* com meta-dimensões tais como resolução, taxa de amostragem e etc, a aplicação estabelece qual é o nível desejado de qualidade e a função utilidade calcula a distância entre a qualidade que a aplicação requer e aquela que os provedores oferecem. Os melhores provedores são aqueles que a função utilidade retornar um valor próximo de zero. Naquele trabalho, o algoritmo calcula a

distância apenas se todos os atributos esperados forem satisfeitos.

Os trabalhos que utilizam a distância entre elementos para compará-los freqüentemente usam a distância Euclideana, porém um problema deve ser resolvido antes. Cada parcela da função utilidade (dimensão de QoS) pode expressar uma grandeza diferente: a resolução pode ser especificada em centímetros enquanto a taxa de amostragem é especificada em Hertz. Assim, para que os termos da função utilidade possam ser somados, é necessário o uso de fatores que efetuem essas escalas e tornem as parcelas adimensionais. Porém, a escolha desses fatores de escala deve ser feita com base nos valores esperados para as medições, o que é tedioso, subjetivo e propenso a erros. Como alternativa, é sugerido em [Huebscher e McCann 2004a] o emprego da distância de Mahalanobis (ou distância estatística), onde cada termo leva em conta não a diferença absoluta entre o valor medido por um sensor e o valor desejado pela aplicação, mas essa diferença dividida pelo desvio padrão calculado, levando-se em consideração todos os sensores que fornecem dados sobre aquela meta-dimensão. Dessa forma, as diferentes grandezas são normalizadas.

Neste trabalho (ver Seções 4.1 e 5.2), propomos normalizar as diferentes grandezas através de funções de satisfação. Ao invés das dimensões de QoS serem empregadas diretamente no cálculo de uma função utilidade, elas servem de argumento para as funções de satisfação (que podem ser lineares do tipo mínimo-máximo) e apresentam como saída um número entre 0 e 1, normalizado e adimensional. Cada dimensão de QoS tem uma função de satisfação parametrizada de acordo com seu intervalo de variação e a preferência do cliente por aquela dimensão de QoS.

2.6.2 Coleta das medições

Outro tópico que deve receber atenção é a maneira pela qual as medições são coletadas pelo *framework*, como mostra o trabalho de [Zegura et al. 2000]. Antes de efetivamente relatar como os autores trataram essa questão vamos contextualizar descrevendo o cenário considerado por eles.

Naquele trabalho, é proposta uma arquitetura capaz de receber pedidos de clientes e redirecioná-los para um servidor, escolhido dentre um conjunto. Para que seja feita essa seleção, um componente da arquitetura, o *Anycast Resolver* (AR), guarda métricas (tais como a porcentagem de CPU disponível e a largura de banda dos enlaces de comunicação) em uma tabela, tomadas sobre os vários servidores participantes. O cliente, para usar o serviço desse componente, deve enviar junto com sua requisição um filtro e um nome de domínio que será resolvido para um endereço IP pelo AR, com base no filtro enviado.

O filtro pode ser pensado como uma função de utilidade ou um critério de seleção que é enviado para o AR dinamicamente, dependendo do interesse e das preferências do cliente. De acordo com esse filtro, o AR consulta sua tabela interna e retorna o endereço IP do melhor servidor dentro do domínio solicitado.

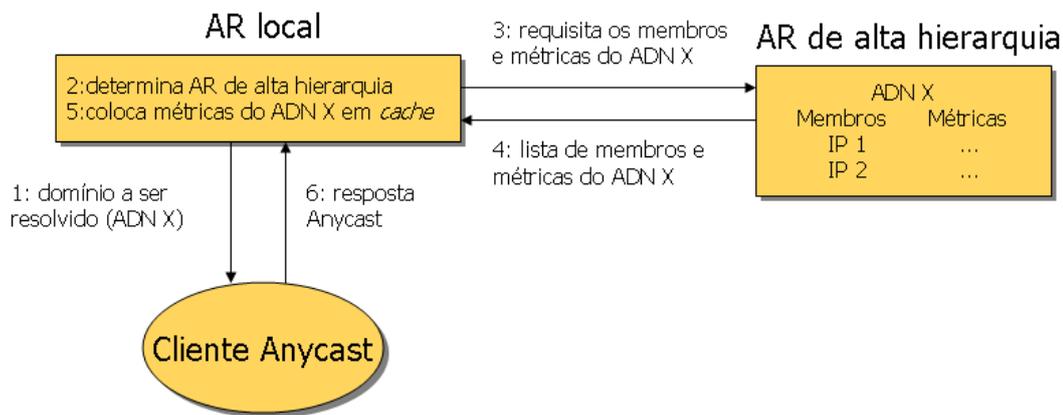


Figura 2.5: Arquitetura proposta para o Anycast Resolver

Na Figura 2.5 estão descritos os passos necessários para a resolução de um nome. No passo 1, o cliente *anycast* envia ao AR local o domínio que ele deseja resolver (converter em um endereço IP) com um filtro que deverá nortear essa seleção. No passo 2, o AR local determina quem é o AR de mais alta hierarquia que deve efetivamente resolver a requisição, tal como é feito na arquitetura DNS. No passo 3, o AR de alta hierarquia recebe o pedido solicitando as métricas especificadas no filtro enviado pelo cliente. O passo 4 representa a resposta à solicitação do passo 3. No passo 5, essas medidas preenchem um *cache* no AR local que servirá para responder solicitações semelhantes no futuro. Sobre esse *cache* é aplicado o filtro enviado pelo cliente. No passo 6, o AR local finalmente responde ao cliente o endereço IP do servidor que melhor se adequou ao seu filtro.

Descrito o cenário, o artigo cita as diversas formas pelas quais as medições podem ser coletadas nos servidores e enviadas para o AR. Uma das possibilidades, chamada de *server push*, consiste em instalar nos servidores *daemons* responsáveis por obter os valores atuais das dimensões de QoS e enviá-los ao AR. Eles fazem isso a cada intervalo de tempo ou quando percebem que uma mudança importante aconteceu e que, por isso, deve ser informada ao AR. É uma técnica escalável, que não exige alteração do código do servidor mas que não leva em conta o atraso de rede, ou seja, funciona apenas para dimensões de QoS que podem ser tomadas localmente ao servidor.

Uma outra técnica é chamada de *server probing*. Agentes co-localizados ao AR fazem permanentemente requisições *dummy* aos servidores. É uma técnica valiosa pois considera

o tempo de processamento da requisição pelo servidor e o tempo de atraso na rede, além disso não implica em alterações no código do servidor. O ponto negativo é que onera o servidor, ou seja, o servidor tem de tratar um pedido falso da mesma forma como ele trata um pedido verdadeiro.

A terceira técnica citada no artigo se baseia na experiência do usuário. A medição é coletada onde mais interessa, ou seja, no cliente a um custo baixo. Para agregar ainda mais valor, o artigo sugere que os clientes podem trocar informações de forma colaborativa, aumentando a visão que o *framework* tem do ambiente. Embora seja interessante, não é de fácil implementação pois os clientes teriam que concordar em hospedar o componente responsável pela monitoração, além disso, a correlação das informações coletadas em diversos clientes não é trivial.

Visando que os clientes tenham o menor tempo de resposta possível quando enviarem suas requisições, o artigo combina os pontos positivos das duas primeiras técnicas, de forma que a medição coletada envolva o atraso de rede e não onere significativamente os servidores. Uma vez que para fazer uma comparação entre servidores não é importante o valor absoluto dessas medições e sim o valor relativo, a opção de coleta adotada pelo projeto Anycast é mista: (a) mais freqüentemente, o servidor envia (*server pushes*) com suas taxas de ocupação até o AR. Ele faz isso quando percebe uma mudança que mereça ser relatada; (b) menos freqüentemente, o AR solicita ao servidor (*server probes*) o envio de um arquivo de tamanho conhecido. Apesar de mais caro computacionalmente, essa modalidade inclui também o atraso de comunicação entre o AR e o servidor indagado. Essa medida é usada para calibrar os *server pushes* de forma que eles representem também, de alguma forma, o atraso de rede.

Por fim, é relevante lembrar que esses tempos de atraso de rede são medidos entre o AR e os servidores. Se os clientes estiverem distantes do AR, o servidor escolhido pode não ser o ideal do ponto de vista dos clientes.

2.6.3 Localização dos componentes coletores

No trabalho de [Molina-Jimenez et al. 2004] a importância da monitoração tem seu *status* elevado. Segundo o artigo, além de possibilitar a criação de aplicações mais flexíveis e imunes às variações do ambiente, a monitoração é também uma ferramenta capaz de aumentar diretamente os lucros do provedor de serviços. Se ele puder provar ao cliente (se possível antes da contratação) que seus serviços satisfazem certos níveis diferenciados de qualidade, ou SLAs (*Service Level Agreements*), então o provedor se destacará dentre seus

concorrentes. Por exemplo, corretoras de ações disponibilizam na Internet o serviço de compra e venda *on-line* e para isso devem refletir as variações de preços imediatamente. Nesse caso, a latência e a segurança das transações são requisitos não-funcionais fundamentais levados em conta pelos clientes no momento da escolha da corretora de ações. Assim, provedores e clientes gostariam de saber se o nível de qualidade do serviço oferecido atende às condições contratadas. É citado no artigo que atualmente a tarefa de alcançar esses SLAs é complexa, a começar pela dificuldade de expressá-los em uma linguagem que não contenha ambigüidades, permitindo que sejam verificados de forma automática pelo *framework*. Como alternativa, provedores de Internet, por exemplo, garantem a qualidade de seus serviços meramente através da modelagem constante da rede (aprovisionamento) e do monitoramento dos padrões de tráfego.

As várias preocupações inerentes à monitoração de recursos são enumeradas no artigo:

- a) Como coletar as métricas? Através de um método passivo (*packet sniffing*) ou ativo (interceptação de pacotes, *probing* com operações sintéticas como citado em [Zegura et al. 2000])?
- b) Onde coletar as métricas: no cliente, no servidor ou em algum ponto intermediário da rede?
- c) Quem é responsável pela coleta?
- d) Qual a qualidade das informações deduzidas das métricas coletadas?

Antes de colocar a proposta daquele trabalho, é preciso explicar que o ambiente é modelado lá em duas partes, e ambas afetam o QoS percebido pelos usuários. São eles, os subsistemas de computação e de comunicação. O primeiro faz o processamento das tarefas, persiste informações em bancos de dados e expõe o serviço que deve ser entregue ao cliente pelo segundo.

À medida que o cliente se afasta do provedor, torna-se mais difícil garantir a qualidade do serviço entregue, visto que diferentes meios podem ser usados, com diferentes características (perdas de pacotes, latência, *jitter*, largura de banda), o que tornaria o custo de gerenciamento dessas redes de comunicação alto demais para esse objetivo. Sendo assim, o artigo define uma região onde o provedor consegue garantir a qualidade do serviço entregue. Na fronteira dessa região ficam os pontos de presença do provedor, ou ISPs (*Internet Service Providers*). Por ser possível garantir o QoS nesses pontos, alguns provedores au-

mentam o número de ISPs dentro de sua área de atuação para que possam atender mais clientes com qualidade, de acordo com seu próprio modelo estratégico de negócio.

Os autores fazem duas importantes suposições: a primeira é de que o cliente não gostaria de ter instalado em sua máquina o componente responsável por realizar a monitoração e, portanto, ela deve ser realizada por terceiros, uma entidade confiada pelo cliente e pelo provedor. A segunda suposição é de que a monitoração é feita com o intuito de verificar se as obrigações do cliente e do provedor estão sendo honradas, e para isso, um componente deve confrontar, de forma automática, as métricas coletadas com os SLAs especificados em um contrato.

Assim, com o objetivo de criar aplicações sensíveis à disponibilidade dos recursos do ambiente (*network-sensitive*), o artigo sugere uma arquitetura de monitoração conforme mostrado na Figura 2.6. Vemos duas instâncias do componente MeCo (*MetricCollector*) que encerra todo o aparelhamento necessário para a coleta das métricas relacionadas ao provimento do serviço, além de duas entidades terceiras que detêm a confiança do provedor e do cliente. A primeira, *Measurement Service* (MS), conhece as métricas que devem ser coletadas e tem a responsabilidade de guardar os resultados obtidos. A segunda, *Evaluation and Violation Detection Service* (E&VDS), deve obter os resultados das bases de dados da primeira, realizar algum processamento necessário para que seja possível comparar essas informações com os níveis pré-acordados dos SLAs e, caso seja detectada uma violação, avisar aos clientes pertinentes da ocorrência. É visível a comparação entre os componentes MeCo e E&VDS propostos pelos autores e os componentes *Resource Agent* e *Contractor* respectivamente, presentes no *framework* CR-RIO [Corradi 2005].

Dois conceitos estão implícitos na arquitetura proposta, ainda em [Molina-Jimenez et al. 2004]. O primeiro diz respeito às notificações de violação de SLAs. Através do paradigma de comunicação *publish/subscribe*, as partes interessadas devem se inscrever em tópicos mantidos pelo E&VDS para receber as notificações - o conhecimento necessário para que as partes inscrevam-se nos tópicos adequados está fora do escopo daquele artigo. O paradigma *publish/subscribe* confere vantagens importantes à arquitetura proposta em termos de escalabilidade: uma mesma notificação de violação pode ser do interesse de várias partes que, portanto, devem se inscrever no mesmo tópico. Além disso, ao receber uma mensagem informando que há uma nova notificação disponível para ser consumida, cada uma delas assincronamente acessa o componente que implementa o E&VDS e a consome.

O segundo conceito é que, por simplicidade, apenas o monitoramento do provedor

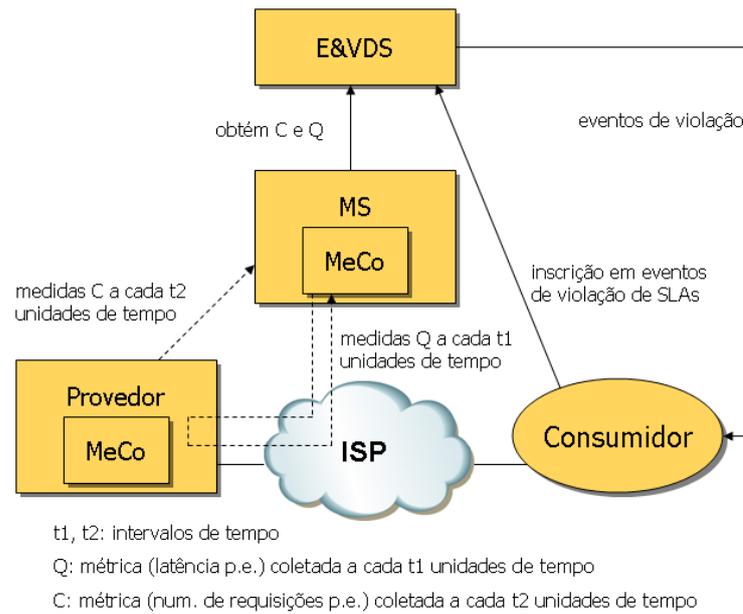


Figura 2.6: A forma de coleta sugerida na proposta

foi representado na figura. Como o contrato é um acordo bilateral entre o provedor e seus clientes, esses últimos também têm responsabilidades a cumprir. Por exemplo, se um cliente, maliciosamente ou não, fizer mais do que um certo limite de requisições por unidade de tempo, ele certamente vai degradar a qualidade do serviço percebido por ele e pelos outros clientes. Caso o mecanismo que verifica se os SLAs estão sendo violados não esteja ciente do comportamento desse cliente, então ele pode gerar desnecessariamente eventos notificando uma violação. Assim, mesmo que o cliente não esteja fornecendo nenhum tipo de serviço, ele também deve ser observado, o que sugere que a monitoração de um sistema deve ser bilateral. Se o cliente estiver preparado para hospedar um MeCo, então o sistema de monitoração seria completo porque teria métricas coletadas também onde o serviço é consumido.

Conforme citado anteriormente, foi suposto que o cliente não deseja ceder recursos e ser aparelhado com um componente coletor de métricas. Dessa forma, um MeCo instalado no componente MS atua como um *probe* e envia requisições ao provedor a cada intervalo de tempo t_1 . Como o contrato garante os níveis de SLAs do serviço oferecido pelo provedor em qualquer ponto dentro do ISP, então essa solução é válida.

Apesar desse coletor de métricas poder inferir bastante sobre a qualidade do serviço entregue ao ISP, o mesmo não pode ser dito sobre a origem de possíveis problemas. Por exemplo, caso os tempos de resposta às requisições feitas ao provedor estiverem altos ou qualquer outro tipo de degradação for percebida, não se pode afirmar que seja devido a um mau funcionamento do provedor. O cliente pode ser culpado também se ele, malicio-

samente ou não, estiver realizando muitas requisições, conforme já discutido. A avaliação se um cliente está ou não dentro da cota contratada pode ser revertida em prêmios ou multas para ele [Cardoso et al. 2006].

Assim, um segundo MeCo é instalado junto ao provedor e envia espontaneamente, a cada intervalo de tempo t_2 , as métricas coletadas para o componente MS. Este MeCo é capaz de coletar informações sobre a quantidade de memória e CPU disponíveis, número de processos alocados e etc, além de poder informar se um cliente está gerando requisições ilegais. É visível a semelhança entre essa técnica e a composta de *server pobres* e *server pushes*, sugerida em [Zegura et al. 2000], conforme mostrado anteriormente.

O artigo cita sistemas especialistas em monitoração de protocolos de rede tais como *Nprobe*, *Windmill* e *EdgeMeter*. Frequentemente tais analisadores requerem alterações em nível de sistema operacional e do *firmware* localizado nas interfaces de rede. Os pacotes coletados na fase de monitoração são analisados em uma segunda fase através de uma trabalhosa reconstrução, *off-line*, de pares *request-response*. Segundo os autores, essa abordagem não se aplica ao problema em questão, uma vez que métricas *end-to-end* de alto nível [Saltzer et al. 1984], como por exemplo o tempo necessário para um usuário fazer uma compra no *site* de uma loja *on-line*, são mais realísticas.

Com o objetivo de demonstrar a aplicação da solução, os autores mostram a integração da arquitetura de monitoração proposta com o servidor de aplicações JBoss [JBoss]. Na Figura 2.7, a aplicação *Web* da Empresa X usa serviços oferecidos pela Empresa Y e são hospedadas em nós diferentes da rede. Um componente que implementa o MS é instalado em cada nó, que pode conter um ou mais componentes MeCo, dependendo da necessidade do desenvolvedor do serviço. As métricas são obtidas através da interceptação das requisições do cliente e suas respectivas respostas e são enviadas para o MS. Uma vez que a solução toda é baseada em Java, isso é feito através de RMI (*Remote Method Invocation*). O MS é responsável por correlacionar as métricas coletadas pelos diversos MeCos (uma vez que eles podem ser instalados em ambientes governados por SLAs diferentes) em uma forma que seja aceita pelo componente *SLAng engine*, que por sua vez, implementa o serviço *E&VDS* já mencionado. A comunicação entre o MS e o *E&VDS* é feita através de mensagens JMS, construídas em XML. Caso seja detectada uma violação de SLA, o *SLAng engine* gera uma mensagem que chega, via JMS, a todas as partes interessadas previamente cadastradas no tópico referente ao SLA violado. A responsabilidade de consumir a mensagem é deixada para a parte interessada. Esse desacoplamento é proporcionado pelo paradigma *publish/subscribe*.

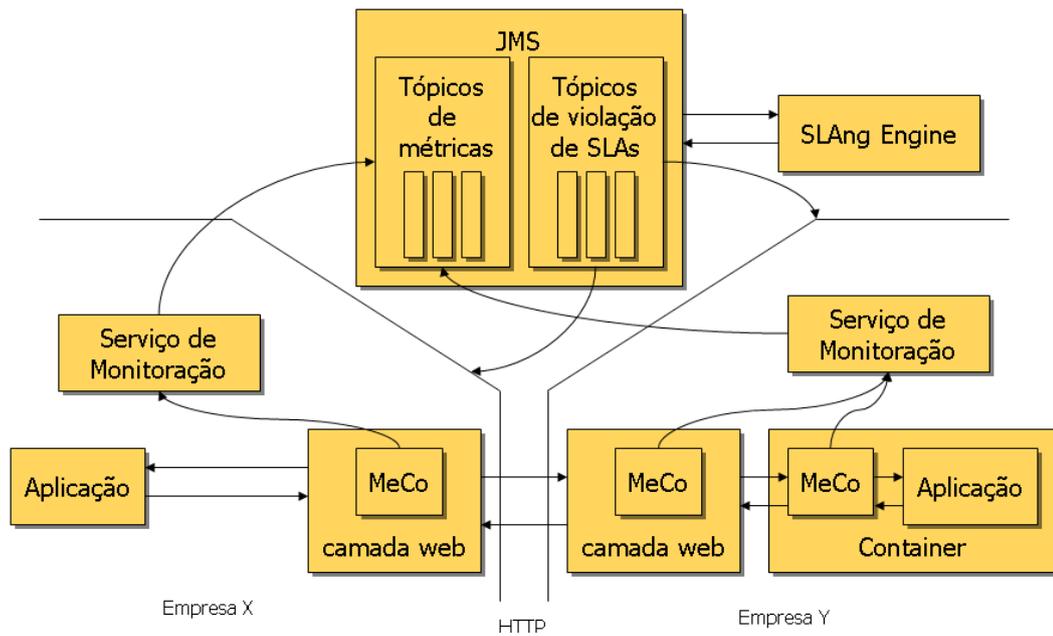


Figura 2.7: Aplicação da estrutura de coleta

Componentes MeCos podem estar localizados dentro de um *container* responsável pela execução de um *bean* (camada de aplicação) ou na camada *Web*. Nesse cenário, interceptadores são colocados na pilha de interceptadores [Leal 2005] do servidor de aplicações JBoss e capturam informações relacionadas à execução do *bean*. MeCos capturam quatro informações básicas: (a) a identificação do chamador, (b) o tipo da invocação, determinado pelo nome do método, (c) o tempo necessário para satisfazer o pedido e (d) o nome do provedor. Com essas informações, o SLAng engine é capaz de determinar qual SLA deve ser verificado.

2.6.4 Pré-processamento das dimensões de QoS monitoradas

Feitas várias considerações sobre a forma de coletar métricas que alimentam o *framework* responsável por promover adaptações, é importante mencionar que essas medidas frequentemente têm muita dinâmica. Por exemplo, um usuário dispõe de um dispositivo móvel com acesso a Internet e ele gostaria que seu dispositivo sempre se conectasse ao melhor AP disponível no ambiente. Supondo que a descoberta dos recursos disponíveis não seja um problema, o *framework* deve escolher o melhor AP para esse usuário baseando sua decisão em dimensões de QoS³ tais como potência do sinal medido e largura de banda. À medida que o usuário se movimenta pelo ambiente, a disponibilidade dos recursos pode variar extremamente, obrigando o *framework* realizar ações adaptativas com uma

³Sistemas disponíveis comercialmente frequentemente escolhem o AP baseando-se apenas na intensidade de sinal, como será mostrado à frente.

grande frequência, aumentando a instabilidade da aplicação. Logicamente, essa situação não é desejada pelo usuário uma vez que pode aumentar sua desatenção (*disruption*) [Poladian et al. 2006, Poladian et al. 2005, Huang 2004], tirando o foco do seu trabalho, principalmente se algumas dessas ações implicarem na interrupção momentânea do funcionamento da aplicação.

Assim, é notório que essas medições devem ser processadas de alguma forma antes de serem consumidas pelo *framework* de maneira a evitar a oscilação da aplicação. Porém, se o mecanismo de monitoração for desenvolvido por uma equipe diferente da equipe responsável pelo consumo das medições, é interessante que esse processamento seja executado pela segunda equipe, pois só ela conhece a aplicação dos dados coletados; uma simples média poderia mascarar um fenômeno. No trabalho de [Cardoso et al. 2006], a mesma equipe teve as duas atribuições e, por isso, implementou em seus experimentos uma técnica chamada de média móvel ou janela deslizante. Naquele trabalho, os recursos são amostrados a cada três segundos e a média das últimas cinco medições é utilizada pelo *framework*. Dessa forma, eventuais picos são amortecidos.

No trabalho de [Zegura et al. 2000] citado anteriormente, um componente chamado *Anycast Resolver* (AR) escolhe o melhor servidor de acordo com um filtro passado pelo cliente. Um fato importante levantado no artigo é que o AR não deve responder para todos os clientes solicitantes o mesmo “melhor” servidor, ainda que eles parametrizem sua solicitação com o mesmo filtro. Se isso acontecer, todos esses clientes passariam a direcionar suas requisições para esse servidor e isso rapidamente degradaria sua *performance* e aumentaria a instabilidade (oscilação) dos seus tempos de resposta. Para evitar isso, os autores propõem um algoritmo que escolhe não um, mas os n melhores servidores, chamados no artigo de servidores equivalentes. Esse conjunto é mantido segundo os seguintes limiares:

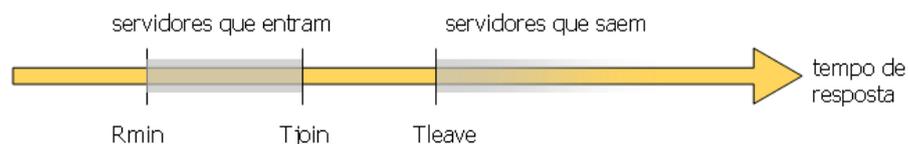


Figura 2.8: *Limiares para definição dos servidores equivalentes*

Os servidores que apresentam tempo de resposta entre R_{min} e T_{join} são admitidos ao grupo de servidores equivalentes enquanto os servidores que têm tempo de resposta maior que T_{leave} saem desse conjunto. Os servidores que se situam entre T_{join} e T_{leave} permanecem no grupo, criando uma espécie de histerese, o que reduz a oscilação da

aplicação. Criado o grupo de servidores equivalentes, o AR escolhe um aleatoriamente e o informa ao cliente.

No trabalho de [Bhatti e Knight 1999] também encontramos a preocupação com a oscilação do *framework*. Naquele artigo, os autores exploram a idéia da compatibilidade entre os recursos monitorados e os vários níveis de qualidade diferenciados (chamados no artigo de estados) que podem ser oferecidos pela aplicação. De forma resumida, será imposto o estado mais compatível com a disponibilidade dos recursos, ou seja, aquele que o ambiente melhor suportar⁴. Para evitar que a aplicação fique oscilando entre estados (*state-flapping*), o usuário pode informar critérios de estabilidade entre suas diversas preferências. Por exemplo: transições para estados de maior qualidade devem ser espaçadas de pelo menos um minuto; mesmo que eles se tornem mais compatíveis com os recursos oferecidos pelo ambiente do que o estado atual. Caso a compatibilidade se reduza, então o sistema deve migrar para um estado de menor qualidade imediatamente. Assim, a máquina de transições é assimétrica, oferecendo mais inércia para migrar para estados de qualidade mais elevada do que na direção contrária.

Uma vez que as preferências dos usuários devem ser levadas em conta para decidir quando e qual transição realizar, o mecanismo encarregado de executar a monitoração - que não tem acesso a essas preferências e, portanto, a uma visão fim-a-fim da tarefa de adaptação [Saltzer et al. 1984, Molina-Jimenez et al. 2004] - não pode participar dessa decisão. Pode apenas produzir relatórios (*QoSReports*) destinados aos componentes que realizam a adaptação, informando a valoração atual das dimensões de QoS monitoradas. Como exemplo de uma dimensão a ser monitorada, os autores sugerem que seja acrescentado o custo, monetário, para o provimento daquele nível de qualidade. Apesar de não conter a dinâmica de dimensões tais como a largura de banda do canal ou a latência de comunicação, o custo deve também ser uma métrica informada pelo sistema de monitoração pois ajudará o mecanismo de adaptação decidir qual o nível de serviço que mais satisfaz ao usuário.

Conforme já citado, aquele trabalho define o grau de compatibilidade de cada nível de qualidade possível de ser oferecido pela aplicação com o ambiente no qual ela é executada. Para isso, devemos estabelecer antes alguns conceitos utilizados no cenário proposto pelos autores:

- **QoSParam:** é um parâmetro ou dimensão de QoS medido pelo sistema de monitoração.

⁴Explicaremos o conceito de compatibilidade criado pelos autores ainda nessa subseção.

- **QoSState**: é um nível de qualidade diferenciado que pode ser oferecido pela aplicação, obviamente, se o ambiente permitir. Tem tantas dimensões quantos forem os **QoSParams** observados.
- **QoSSpace**: trata-se de um espaço multi-dimensional, também tem tantas dimensões quantos forem os **QoSParams** observados, e no qual são mapeadas as medições dos **QoSParams** feitas no ambiente. Os **QoSStates** são subconjuntos, freqüentemente não-disjuntos, do **QoSSpace**.

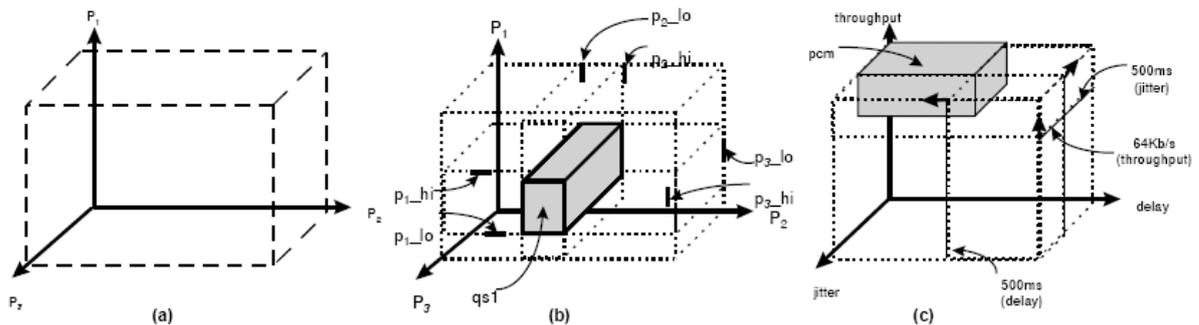


Figura 2.9: Representação gráfica dos conceitos apresentados [Bhatti e Knight 1999]

Na Figura 2.9a, está representado um **QoSSpace** com três dimensões, o que denota que ele é composto por três **QoSParams**: P_1 , P_2 e P_3 . Na Figura 2.9b, vemos um **QoSState** encerrado entre os limites inferior ($_lo$) e superior ($_hi$) de cada uma das dimensões. Na Figura 2.9c, temos um exemplo de um serviço de transmissão de áudio: uma das dimensões é a latência de comunicação e deve ser de, no máximo, 500ms. A segunda dimensão representa a *jitter* e deve ser também de 500ms no máximo. A terceira dimensão representa a largura da banda passante e deve ser de pelo menos 64kbps. É possível notar que os **QoSStates** podem se sobrepor.

O próximo conceito retrata as medições realizadas no ambiente. Uma vez que o trabalho dos autores é focado no comportamento de uma rede de comunicação, ele é batizado de **NetQoSState** e revela a valoração dos **QoSParams**. Teoricamente, ele deveria ser um ponto, uma tupla com quantas dimensões forem as dimensões observadas pelo sistema de monitoração, porém a incerteza, ruídos de medição e a variação inerente às dimensões de QoS fazem que o **NetQoSState** seja uma região do espaço multi-dimensional, sendo visível a comparação com os perfis utilizados em [Corradi 2005].

Assim, para calcular o grau de compatibilidade dos **QoSStates** com o QoS oferecido pelo ambiente, os autores avaliam o grau de compatibilidade de cada dimensão, ou seja, de cada **QoSParam** T separadamente da seguinte forma: seja T_q a variação desse parâmetro

medida no ambiente e T_p a variação aceita pelo sistema para que seja possível impor uma certa qualidade diferenciada de serviço ($QoSState$). Define-se a função $WITHIN$, entre 0 e 1, que calcula a compatibilidade do $QoSParam$ T da forma:

$WITHIN(T_q, T_p) =$ interseção entre o comprimento de T_q e o comprimento de T_p

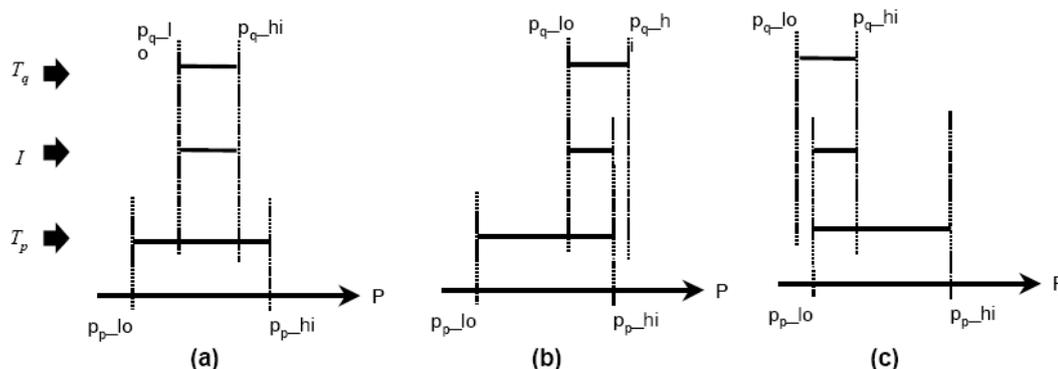


Figura 2.10: Possibilidades de valoração [Bhatti e Knight 1999]

Na Figura 2.10, mostramos três possibilidades de valoração das grandezas T_q e T_p . O serviço aceita que a variação do $QoSParam$ P esteja entre p_{p_lo} e p_{p_hi} , enquanto que o ambiente fornece uma variação desse parâmetro entre p_{q_lo} e p_{q_hi} . Em (a) o parâmetro é 100% compatível com a necessidade do serviço. Em (b) vemos que o limite superior da variação do parâmetro é maior que o aceitado para a imposição do serviço: $p_{q_hi} > p_{p_hi}$. Em (c) vemos que o limite inferior da variação do parâmetro é menor que o aceitado para a imposição do serviço: $p_{q_lo} < p_{p_lo}$. Em (b) e (c), portanto a compatibilidade é menor que 100%. Assim, para calcular a compatibilidade *State Compatibility Value* (SVC) de um $QoSState$ de N $QoSParams$ usa-se a função AND_{fuzzy} . Uma vez que os parâmetros são avaliados independentemente pela função $WITHIN$, eles são tratados como ortogonais pelos autores.

$$\begin{aligned}
 SVC = & WITHIN (T_{q_1}, T_{p_1}) AND_{fuzzy} \\
 & WITHIN (T_{q_2}, T_{p_2}) AND_{fuzzy} \dots \\
 & WITHIN (T_{q_n}, T_{p_n}) AND_{fuzzy}
 \end{aligned}$$

Figura 2.11: *State Compatibility Value* de vários $QoSParams$

Feita esta descrição do cenário utilizado pelos autores, o algoritmo encarregado de realizar as transições se vale de duas variáveis de controle que regulam a estabilidade da aplicação, inserindo uma espécie de histerese: $q_compatibility$ e q_time . Quando o usuário valora essas variáveis com 80% e 60s respectivamente, significa que ele deseja uma política de transições tal que o algoritmo “não deve transitar para um estado de

qualidade superior a menos que ele seja compatível com o QoS oferecido pela rede em 80%, nos últimos 60 segundos”. Caso necessárias, as transições para `QoSStates` de menor qualidade não têm essas restrições e são imediatas. Dessa forma, transições para estados de maior qualidade acontecem espaçadas de, pelo menos, 60 segundos. Para garantir que o cálculo da compatibilidade de um estado não seja prejudicado por ruídos inerentes ao ambiente, ela é calculada através da média tomada sobre uma janela deslizante de 60 segundos.

Com um valor baixo para `q_time`, o algoritmo rapidamente percebe uma melhora do QoS oferecido pelo ambiente e transita para um `QoSState` de melhor qualidade. Por outro lado, aumenta a instabilidade do sistema. Um valor de `q_time` alto, torna o sistema lento para executar essas transições de melhoria, mas aumenta sua estabilidade. Com um valor de `q_compatibility` baixo, o sistema torna-se menos exigente para realizar uma transição de melhoria e, por isso, a realiza com facilidade. Um valor alto é mais difícil de ser atingido e, portanto, é mais difícil para a aplicação transitar para um estado de maior qualidade.

Embora o modelo, conforme descrito até então, consiga perceber quando os `QoSParams` ultrapassam a fronteira da variação aceita pela aplicação, ele não percebe quando um desses parâmetros está prestes a cruzar essa fronteira, ou seja, tem um valor muito próximo ao do limiar. Essa informação seria útil para a aplicação pois pode significar que o `QoSState` atual não poderá ser suportado pelo ambiente em breve e, portanto, uma transição deverá ser feita. Por isso, na Figura 2.12, os autores definem dois estados intermediários `QoSStates`, um superior e outro inferior definidos por `p_qhi` e `p_qlo` respectivamente, internos ao `QoSState`⁵. Quando pelo menos um dos `QoSParams` estiver dentro de um desses dois `QoSStates`, diz-se então que o `QoSState` está operando perto da fronteira, e que uma transição de estado poderá ser necessária em breve. Assim, uma expressão análoga à anterior pode ser usada para o cálculo de `SCV_I`, substituindo o operador `ANDfuzzy` pelo operador `ORfuzzy`.

Dispondo desse ferramental, os autores adotaram uma política simples em que se as métricas monitoradas estiverem muito próximas à fronteira inferior de um estado, ou seja, dentro do `QoSState` inferior, aquele estado não é mais utilizado e o algoritmo transita para o estado de qualidade imediatamente inferior.

A preocupação de construir sistemas adaptáveis, sensíveis às variações dos recursos do ambiente e que sejam estáveis também é tratado em [Poladian et al. 2006]. Naquele

⁵Para a figura, foi retratado um `QoSState` unidimensional, definido por apenas um `QoSParam P`

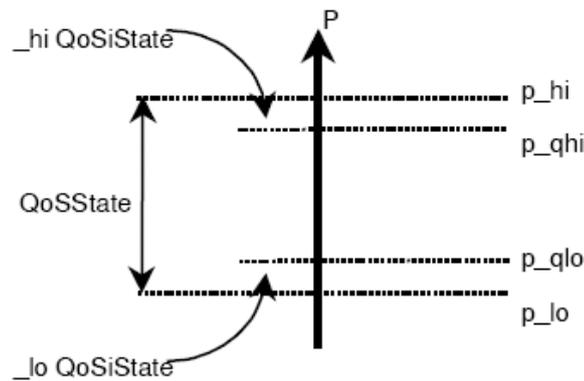


Figura 2.12: *Estados intermediários [Bhatti e Knight 1999]*

trabalho, o *framework* utiliza-se de uma função utilidade para verificar constantemente a existência de uma configuração mais adequada para o provimento da aplicação do que a atual. Essa abordagem facilmente levaria a um mecanismo instável. Para contornar o problema, os autores introduziram o conceito de custo de reconfiguração. Esse custo é uma preferência do usuário e captura sua intolerância à mudanças de configuração no conjunto de componentes que provêm a aplicação, inserindo uma certa histerese a aplicação e evitando sua oscilação entre configurações que competem entre si. Um alto custo de reconfiguração fará a aplicação muito estável mas freqüentemente não-otimizada, ao passo que um custo baixo permitirá a aplicação adaptar-se livremente às variações do ambiente mas aumentará a distração do usuário entre reconfigurações. Essa técnica será utilizada na Seção 5.2.3.

2.6.5 Lógica *fuzzy*

Novas abordagens podem ser usadas para o processamento das métricas coletadas. No trabalho de [de Campos e Saito 2004] vemos a aplicação da lógica *fuzzy* para evitar que ruídos inerentes ao sistema de medição confundam o mecanismo de adaptação. Segundo os autores, a lógica *fuzzy* nasceu da observação de que quando a complexidade de um sistema aumenta, nossa capacidade de concluir fatos e tomar decisões que sejam ao mesmo tempo precisas e significativas, tende a diminuir até um limite a partir do qual precisão e relevância passam a ser características quase excludentes. Ao tratarmos tais sistemas computacionalmente, nós estaremos criando um modelo muito simplificado da realidade uma vez que nossas observações são incompletas e inexatas. Incompletas porque simplificações são necessárias para que seja possível a modelagem e o tratamento computacional. E inexatas porque as informações são tomadas por componentes e, portanto, sempre conterão algum grau de ruído e incerteza.

A lógica *fuzzy* altera o paradigma do controle de processos e, ao invés de modelar o processo em si, tenta modelar as possíveis ações tomadas pelo ser humano controlador do processo. Apesar desse controle requerer alcançar objetivos incompletos, imprecisos e muitas vezes contraditórios; um operador humano consegue fazê-lo. Os subconjuntos *fuzzy* permitem que grandezas tais como a latência de um canal de comunicação não seja classificada apenas em dois estados mutuamente exclusivos, como é na lógica *booleana*. Não se pode definir a partir de quantos mili-segundos a latência deixa de ser baixa e se torna alta. Esses subconjuntos evitam a passagem brusca entre duas classes, permitindo que uma medição possa pertencer parcialmente à várias delas, por causa da imprecisão da própria linguagem humana ou do ruído inerente às métricas coletadas.

Em [Bertini et al. 2006], os autores utilizaram essa técnica. Aquele trabalho trata da replicação de dados em situações de desastre onde um dispositivo móvel de uma rede *ad-hoc* pode se isolar dos outros do seu grupo seja por falta de energia de sua bateria, seja por causa da distância entre ele e o grupo. Quando o dispositivo percebe que vai se isolar, então ele deve replicar seus dados em algum outro do grupo e, para isso, os autores escolhem o dispositivo receptor com base na decisão *bayesiana*, visando aumentar a disponibilidade dos dados e reduzir o número de replicações. Para modelar os vários valores possíveis que as dimensões de QoS relevantes ao estudo (energia das baterias e intensidade do sinal de comunicação entre os dispositivos) podem assumir, os autores se valeram de três níveis *fuzzy*, evitando uma explosão combinações (estados) observáveis e obtendo a estabilidade do mecanismo de decisão.

2.6.6 Ortogonalidade entre dimensões de QoS

Uma última questão importante de ser mencionada quanto à monitoração de recursos é que as dimensões de QoS são tratadas de forma ortogonal (evoluindo independentemente) em diversos trabalhos [Poladian et al. 2006] [Bhatti e Knight 1999] [Huebscher e McCann 2004a], o que facilita a análise do problema. Entretanto [Huang 2004] se preocupa com isso e mostra que, caso contrário, o espaço de soluções cresce exponencialmente. Em seu artigo, o *framework* dispõe de um componente chamado **Synthesizer** que tem a responsabilidade, entre outras, de realizar o mapeamento físico dos componentes abstratos com base nas instâncias disponíveis no ambiente, conforme será verificado em detalhes na Seção 3.1. Essa seleção se dá através em uma função utilidade que é montada a partir das métricas que interferem na qualidade do serviço oferecido, ou seja, os componentes serão avaliados através de suas métricas latência e largura

de banda.

O autor faz uma análise em que devem ser escolhidos m componentes e, cada um deles, tem n representantes (instâncias disponíveis no ambiente e portanto passíveis de serem escolhidas) então: (a) se as métricas forem independentes e os componentes puderem ser escolhidos uma a uma, separadamente (otimização local), então o espaço de busca será de $m \times n$ possibilidades ou (b) se os componentes tiverem de ser escolhidos todos ao mesmo tempo (otimização global) porque suas métricas não são ortogonais, então o espaço de busca aumenta e será, no pior caso, de n^m possibilidades.

2.7 Gerência de ações conflitantes

Uma preocupação importante mas que não aparece com muita frequência nos trabalhos que lidam com sistemas adaptativos é a coordenação de ações de adaptação conflitantes. Pela própria natureza dos sistemas distribuídos que podem obter informações a cerca do ambiente de diversas fontes ou pela dificuldade em determinar a causa raiz de um problema (por exemplo, se o vídeo apresenta uma baixa qualidade pode ser em razão da baixa qualidade do enlace de comunicação ou do servidor que provê o vídeo), o *framework* que monitora a disponibilidade dos recursos do ambiente pode promover ações que, se aplicadas, levam a efeitos contrários.

No trabalho de [Huang 2004] vemos que o autor se preocupa em estender sua API no sentido de permitir ao desenvolvedor do serviço a possibilidade de informar ao *framework* três tipos diferentes problemas que podem surgir entre ações adaptativas. São eles:

- **detecção de conflitos entre propostas:** segundo o autor, os conflitos ocorrem em dois níveis: (a) em nível da ação e (b) em nível de problema. O primeiro é caracterizado quando uma proposta sugerida pelo *framework* deseja substituir o componente A por B enquanto que outra proposta deseja substituir o mesmo componente A por C. Obviamente apenas uma delas poderá ser levada a cabo e, neste caso, o sistema pode automaticamente detectar tais situações. O segundo caso é ilustrado quando um usuário tenta se conectar a um grupo de servidores, o sistema escolhe o melhor servidor para ele de acordo com suas preferências mas, no ato da conexão, aquele servidor apresenta problemas e o *framework* tenta substituí-lo por outro semelhante. A solução deste problema é específica da aplicação sendo oferecida e não pode ser determinada automaticamente pelo *framework*. Dessa forma, o desenvolvedor deve explicitar ao *framework* quais as ações adaptativas (chamadas

no artigo de táticas) são conflitantes. Para isso, o autor estende sua API através do comando `addProblemConflict(T1, T2)`, no qual o desenvolvedor pode informar que as táticas T1 e T2 são conflitantes.

- **resolução de conflitos:** uma vez ciente de quais táticas são conflitantes⁶, o *framework* deve saber como tratá-las, e o autor sugere duas políticas distintas: (a) *First-Come, First-Service (FCFS)* e (b) *Epoch/priority*. A primeira prevê que entre duas propostas conflitantes, apenas a que iniciar primeiro o processamento será executada. Por exemplo, na Figura 2.13a, sabendo-se que as propostas p1 e p2 são conflitantes, o *framework* só permite que a primeira seja executada, enquanto a segunda é deferida. Ainda no mesmo exemplo, as propostas p4 e p5 são conflitantes entre si, enquanto a proposta p6 não. Dessa forma, a proposta p5 foi deferida em favor da proposta p4, enquanto que a proposta p6 iniciou sua execução normalmente. Na segunda política citada, o *framework* acumula a cada janela de tempo as propostas que solicitaram execução. Ao fim da janela, apenas aquelas não-conflitantes entre si são executadas. Na Figura 2.13b, na primeira janela as propostas p1 e p3 são conflitantes entre si e, por isso, apenas a proposta p2 foi executada. Segundo o autor, a segunda política é mais flexível mas reduz a velocidade de reação do sistema, tornando-a inversamente proporcional ao tamanho da janela e, dessa forma, é usada em aplicações com fracas restrições de tempo. Assim, o autor implementou a primeira política pois julgou ser importante a agilidade na resposta do *framework*.
- **identificando incompatibilidades:** algumas táticas tentam adaptar a aplicação em sentidos opostos causando ciclos e oscilações. Por exemplo, sejam duas táticas Ta e Tb que adicionam e removem servidores de um *cluster*, dependendo da carga de tráfego experimentada por ele. Se as condições de disparo⁷ não forem cuidadosamente definidas, a aplicação pode ficar instável. Ao invés de usar mecanismos que controlam as causas e efeitos de cada tática, o autor observa que frequentemente é suficiente indicar que as táticas Ta e Tb possuem causas e efeitos cruzados, o que permite ao *framework* detectar os ciclos causados por sua execução.

É interessante notar que o problema da oscilação é também tratado em [Bhatti e Knight 1999], porém sob outra ótica. Lá, o autor se preocupa nas oscilações decorrentes da dinâmica da disponibilidade dos recursos do ambiente e sugere técnicas

⁶aqui chamadas de propostas, uma vez que nem todas serão efetivamente executadas. Essa nomenclatura é descrita na Seção 3.1.

⁷o autor implementou um mecanismo de condições e regras que é explicado em detalhes na Seção 3.1

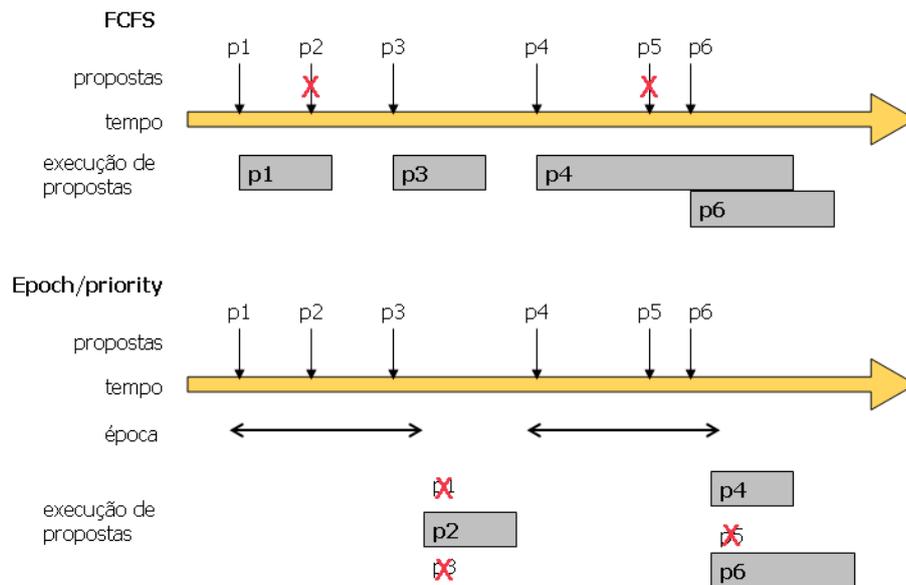


Figura 2.13: Políticas para resolução de conflitos

para minimizar o problema como foi visto na Seção 2.6.4; e não devido a ações de adaptação conflitantes geradas pelo próprio *framework*.

2.8 Conclusão do capítulo

Conforme exposto, a adaptação é um requisito cada vez mais necessário para que as aplicações possam operar em ambientes onde a disponibilidade dos recursos e as preferências do usuário variam ao longo do tempo. Porém, muitos fatores têm de ser resolvidos para que esse objetivo seja atingido. Entre os principais temos a descrição arquitetural, seleção dos recursos (ou mapeamento físico) e a monitoração dos recursos disponíveis. As soluções mais interessantes são aquelas em que os mecanismos de adaptação são fatorados do sistema e reunidos em um *framework* que pode ser reutilizado em várias aplicações semelhantes.

Capítulo 3

Trabalhos correlatos

Estudadas as questões mais importantes que devem ser resolvidas para a adaptação de aplicações, neste capítulo vamos nos concentrar nos principais mecanismos encontrados na literatura para realizar o mapeamento físico, em outras palavras, estudaremos como a descrição arquitetural é resolvida em termos de instâncias de componentes que estão efetivamente disponíveis no ambiente.

3.1 *Synthesizer* e representação por receitas

Conforme citado anteriormente, [Huang 2004] se vale de “receitas”, que se assemelham a *scripts*, para descrever a arquitetura da aplicação. Através do uso de uma API exposta pelo *framework*, o desenvolvedor define quais componentes são necessários e como são interconectados (linhas 1, 4, 5, 6, 7 e 8 no Código 3.1). No mesmo instante, ele define a função utilidade (linhas 2, 3, 9 e 10 no Código 3.1) que deverá ser otimizada quando o mapeamento físico for realizado, ou seja, guiará o *framework* durante sua procura no espaço de soluções. Ao montar a função utilidade, também chamada de função-objetivo, o desenvolvedor tem a oportunidade de usar seu conhecimento particular para definir quais dimensões de QoS são importantes para a qualidade do serviço. No exemplo, apenas a latência de comunicação foi levada em conta, embora o próprio autor ressalte que outras grandezas tais como a carga de processadores, memória disponível e etc, poderiam ser usadas.

É possível notar que a montagem da função utilidade se dá juntamente com a definição arquitetural da aplicação, o que pode ser visto por dois ângulos. Por um lado, o próprio desenvolvedor (conhecedor do funcionamento da aplicação) também conhece quais as dimensões de QoS impactam na qualidade entregue. Assim, ele está apto a de-

finir a função utilidade. Por outro lado, espera-se que durante a descrição arquitetural ele estivesse apenas preocupado com os requisitos funcionais, deixando os não-funcionais para outro momento do processo de desenvolvimento. Não é muito claro naquele trabalho que o desenvolvedor encontra na definição da função utilidade a oportunidade de capturar como o usuário privilegia uma dimensão de QoS em detrimento de outras. Se houvesse outras dimensões de QoS envolvidas na montagem da função utilidade, então elas poderiam participar com pesos diferentes na composição dessa função, conforme será proposta na Seção 4.1.

Código 3.1: Especificação da função utilidade

```

AbsConf conf = new AbsConf();
Term obj = new Term(new Double(0.0));
term partialObj = new Term(new Double(0.0));
AbsComp vgw = conf.addComp("VGW");
5 for (i=0; i<participants.size(); i++) {
    PhyComp pc = (PhyComp) participants.get(i);
    if (pc.getProperty("App").equals("NM")) {
        conf.addConn(vgw, pc);
        partialObj.add(new Term(new LatencyM(vgw, pc)));
10 }
}

```

Uma das vantagens da utilização da descrição arquitetural para o estabelecimento da aplicação é que é dada ao *framework* uma visão global: ele conhece todos os componentes necessários à aplicação e, com isso, tem a possibilidade de realizar uma otimização global. Porém, essa otimização global no intuito de realizar o mapeamento físico pode levar a um grande número de possibilidades e a exploração exaustiva desse espaço de soluções pode ser muito lenta, visto que para cada uma delas a função utilidade deve ser recalculada, ainda que em parte.

Para contornar essa situação, [Huang 2004] sugere dividir em partes a arquitetura que provê a aplicação, seguindo o conceito de “dividir para conquistar”, de forma que cada uma das partes possa ser analisada e otimizada local e independentemente. Essa estratégia parte do princípio que o ótimo global, ou pelo menos algo bem próximo dele, pode ser constituído pela composição de vários ótimos locais; tantos quantos forem as partes no qual a arquitetura da aplicação foi dividida.

O argumento que justifica esse raciocínio é que a função utilidade global é um somatório de termos, mas que pode ser pensada também como um somatório de funções

utilidades locais, de menor abrangência (a abrangência de cada uma das partes no qual a arquitetura da aplicação foi dividida). Otimizando cada uma delas independentemente, tende-se a se chegar a um ótimo bastante próximo do ótimo global.

Porém, ao contrário das métricas que envolvem apenas um componente, como por exemplo a carga de processamento ou o consumo de memória demandados por ele, algumas métricas envolvem dois ou mais componentes concomitantemente. Um exemplo é a medição da latência na comunicação entre dois componentes, o que sugere que eles sejam escolhidos aos pares, e não individualmente. Uma vez que cada um deles pode residir em uma divisão da arquitetura criada para que o mapeamento físico seja viável, então o resultado ficará muito aquém do ótimo global. Para contornar esse fato, em cada otimização local é construída uma lista com as n melhores configurações.

Isto posto, seja o cenário onde o sistema é dividido em duas partes. Na primeira parte, o *framework* lista as n melhores otimizações locais (candidatos) possíveis. Para gerar essa lista, o autor prevê a necessidade de utilizar meta-heurísticas como por exemplo o *Simulated Annealing*. O processo é repetido para a segunda parte e é gerada uma lista com as m melhores otimizações locais (candidatos). A partir daí, o *framework* faz uma busca exaustiva em cada uma das $n \times m$ possibilidades, calculando a função utilidade global. Uma vez que essa técnica utiliza o *Simulated Annealing* localmente e em seguida uma busca exaustiva, ela é referida pelo autor como híbrida.

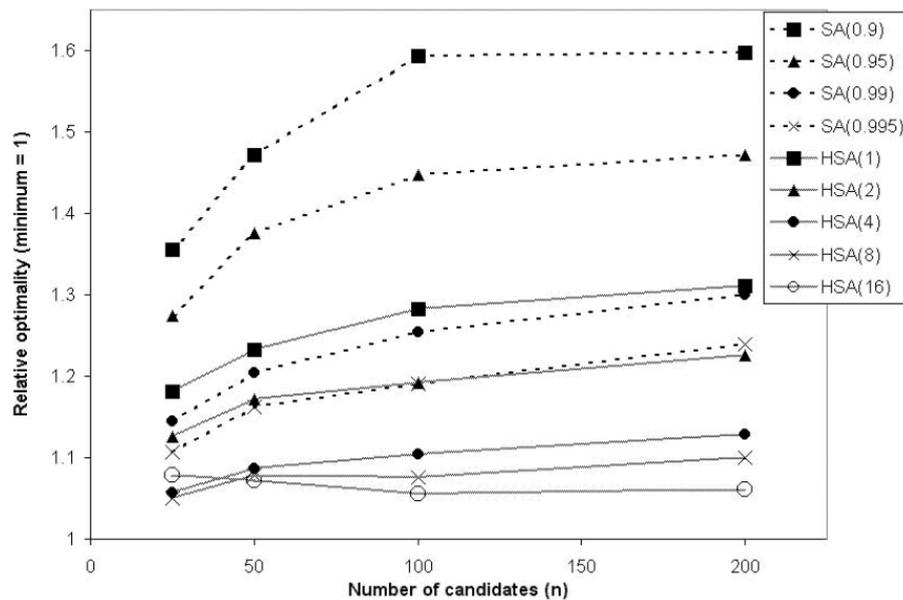


Figura 3.1: Comparação entre meta-heurísticas na busca do ótimo global [Huang 2004]

Na Figura 3.1, vê-se que as técnicas híbridas (HSA(16) significa *Hybrid Simulated Annealing* onde cada otimização local gera uma lista com as 16 melhores configurações)

superam as técnicas puramente meta-heurísticas (SA - *Simulated Annealing*), dado um número restrito de iterações. Uma otimização unitária significa que o método atingiu o ótimo global.

É interessante notar que [Huang 2004] divide o problema da adaptação de aplicações em duas fases distintas com escopos bem definidos. A primeira fase, descrita anteriormente, é chamada de configuração, tem um âmbito global onde a descrição arquitetural do sistema e funções utilidade são levadas em conta durante a imposição da aplicação. A segunda fase é chamada pelo autor de adaptação, na qual é utilizado um modelo orientado a eventos e ocorre durante a execução da aplicação, para a manutenção da sua qualidade. Essa segunda fase será descrita a seguir.

Segundo o autor, há duas correntes na literatura que dizem respeito às adaptações feitas em tempo de execução: soluções que usam funções utilidade e soluções baseadas em regras de eventos-ações. Soluções baseadas em funções utilidade permitem ao desenvolvedor especificar uma função utilidade indicando a configuração desejável da aplicação, e o próprio *framework* se encarrega de ajustá-la de forma que a configuração de maior utilidade seja encontrada. Porém o autor diz que seu objetivo é muito mais amplo pois inclui a troca, remoção e adição de componentes à configuração em funcionamento. Desta forma, o autor diz que as soluções baseadas em funções utilidade não são aplicáveis para manter a qualidade de uma aplicação já em operação.

Ainda segundo o autor, as soluções baseadas em regras de eventos-ações permitem ao desenvolvedor especificar ações de devem ser executadas pelo *framework* quando determinados eventos ocorrem, por exemplo: “quando o componente X ficar sobrecarregado, substitua-o por outro de maior capacidade”. Desta forma, o autor usa uma solução baseada em regras de eventos-ações e, para isso, ele define quatro elementos que são utilizados para promover a adaptação em tempo de execução, a saber:

- **Constraint:** é uma condição que é sempre monitorada pelo *framework* e, quando violada, dispara ações.
- **Problem determination:** a violação de uma condição pode ter várias causas raiz. Por exemplo, o autor cita o caso de um sistema de videoconferência no qual uma condição é violada pois a aplicação apresenta baixa qualidade de vídeo. Uma análise mais profunda é necessária para identificar se a causa raiz é por exemplo o *codec* que está parametrizado incorretamente ou o enlace de comunicação utilizado que apresenta uma pequena largura de banda. O autor lembra que como todos os

elementos do seu *framework* são implementados por classes Java, o desenvolvedor tem bastante flexibilidade para realizar essa análise.

- **Tatic:** uma tática consiste de um conjunto de ações de devem ser executadas quando uma determinada condição é violada. As ações podem ser do tipo: substituir um componente, removê-lo da configuração atual e etc.
- **Strategy:** representa a estratégia em si. É a estrutura de dados que reúne todos os elementos anteriores.

Assim como na primeira fase, de configuração, na fase de adaptação o desenvolvedor também usa de uma API (que é descrita em detalhes no trabalho do autor) para definir os elementos de uma estratégia.

Acreditamos que soluções baseadas em regras de eventos-ações podem ser interessantes mas ainda neste caso as funções utilidade devam ser levadas em conta. Após o *framework* perceber que uma condição foi violada e decidir que deve substituir um componente por outro de maior capacidade ainda resta a tarefa de avaliar qual componente deve entrar no lugar daquele que deixa a configuração, caso haja mais de um disponível no ambiente. Nesse caso, as funções utilidade têm se mostrado uma técnica útil como será mostrado no Capítulo 4. O autor parece não ter percebido essa necessidade.

3.2 CR-RIO

A mescla entre a descrição arquitetural e a definição das dimensões de QoS que impactam a qualidade da aplicação provida não é encontrada no *framework* CR-RIO (*Contractual Reflective - Reconfigurable Interconnectable Objects*), [Loques et al. 2004, Corradi 2005]. Naquele trabalho, a descrição arquitetural da aplicação é feita separadamente das políticas que devem governar seu funcionamento, promovendo uma separação de interesses de forma mais explícita. Conforme citado anteriormente, os módulos, os conectores e sua forma de interconexão são descritos naquele *framework* de acordo com o Código 2.1.

Os requisitos não-funcionais oferecidos pela aplicação são regulados por um contrato em CBabel, conforme o Código 3.2. Um contrato envolve duas ou mais partes, explicitando suas responsabilidades mútuas, e pode ser aplicado a toda arquitetura ou a partes específicas da aplicação, de acordo com a granularidade desejada pelo desenvolvedor. Por exemplo, uma aplicação cliente-servidor pode ter um contrato regendo apenas o funcionamento do cliente e outro para o servidor de maneira que as partes possam evoluir

independentemente. O contrato define níveis diferenciados de qualidade para a imposição da aplicação, de forma que o *framework* possa, através de adaptações motivadas por variações no contexto (violações nos perfis), oferecer o nível de qualidade que maximize a satisfação do usuário. Esses níveis diferenciados de qualidade são modelados como estados de uma máquina de estados, conforme pode ser visto na cláusula *negotiation* do contrato a seguir. Essa máquina especifica uma ordem arbitrária para a imposição dos diferentes níveis de qualidade. Frequentemente, existe um estado falso (*out-of-service*) para explicitar uma transição realizada pela máquina quando nenhum dos níveis de qualidade pôde ser tolerado pelo ambiente. No exemplo a seguir, o contrato é utilizado para instanciar uma aplicação servidora em um nó da rede. Como esse nó é escolhido, será explicado adiante.

Código 3.2: Um contrato do *framework* CR-RIO

```

contract {
  service {
    instantiate server at host1 with ProcMem;
  } prioProc;
5 negotiation {
  prioProc->out-of-service;
  };
}oneServer;
profile {
10 Processing.cpuSlice >=0.25;
  Processing.memReq>=200;
}ProcMem;

```

Um nível diferenciado de qualidade é imposto apenas quando todos os perfis requeridos por ele forem válidos. Quando um desses perfis é violado, o *framework* consulta a máquina de estados para saber qual o nível de qualidade, na sequência determinada pelo desenvolvedor, pode ser oferecido. Este novo nível precisa ter todos os perfis atendidos pela disponibilidade dos recursos do ambiente. No caso deste exemplo, apenas o perfil *ProcMem* é utilizado pelo nível de qualidade *prioProc*.

Perfis são definidos no CR-RIO como valorações/restrições de dimensões de QoS. No caso do exemplo, o perfil *ProcMem* é válido quando o poder de processamento disponível do nó é superior a 25% e a memória disponível é superior a 200Mbytes (ver Código 3.2). Em outras palavras, o contrato deste exemplo fará o *framework* CR-RIO instanciar a aplicação servidora em qualquer nó escolhido aleatoriamente, desde que ele obedeça a esse perfil. A definição dos perfis é uma maneira clara de estabelecer um SLA (*Service*

Level Agreement) entre o cliente e a aplicação e ocupa uma lacuna importante: geralmente linguagens de especificação de SLA são ambíguas [Molina-Jimenez et al. 2004].

As dimensões de QoS de um determinado recurso relevantes à imposição da aplicação são reunidas e organizadas em `QoSCategories`. Elas representam os recursos não-funcionais necessários para o seu estabelecimento. No exemplo, o perfil `ProcMem` usou duas dimensões da `QoSCategory Processing`, a saber: `cpuSlice` e `memReq`. Uma vez que as `QoSCategories` podem ser reusadas entre aplicações diferentes, gerenciadas por contratos diferentes, os perfis podem ser pensados também como instâncias das `QoSCategories` numa linguagem orientada a objetos que permita a herança múltipla.

Código 3.3: Uma categoria do *framework* CR-RIO

```
QoSCategory Processing{
    cpuUse: decreasing numeric % in;
    cpuSlice: increasing numeric % in;
    priority: increasing numeric in;
5 memReq: increasing numeric MBytes in;
};
```

Cada dimensão de QoS presente em uma `QoSCategory` é marcada com a propriedade `in` ou `out`. A primeira significa que o atributo deve ser monitorado e comparado com os valores estipulados no perfil, e a segunda significa que o desenvolvedor deve valorá-lo durante a criação do perfil para o estabelecimento da aplicação. Uma descrição sucinta dos elementos do *framework* CR-RIO pode ser encontrada em [Cardoso et al. 2006].

Neste exemplo, a escolha do nó da rede para abrigar a aplicação servidora foi relativamente simples: qualquer nó que satisfizesse o perfil `ProcMem`. No entanto, essa escolha pode se tornar mais complexa e sofisticada à medida que mais condições forem adicionadas. Seja o exemplo citado na Seção 2.1 e em mais detalhes na Seção 4.2, onde o cliente deseja enviar uma requisição para um servidor organizado em *cluster*. O cliente gostaria que sua requisição fosse atendida dentro do menor tempo possível. Conforme será dito, mecanismos de balanceamento de carga sensíveis ao contexto devem analisar a requisição HTTP até a camada 7 do modelo ISO/OSI e por isso não têm uma boa escalabilidade.

Tal como exposto, os perfis e a máquina de estados guiam a adaptação realizada pelo *framework* e podem ser usados para guardar as preferências do usuário. Por exemplo, se o usuário souber que sua requisição é *I/O Bound* então ele pode valorar a `QoSCategory Transport` de forma a criar um perfil mais exigente quanto a esse recurso. Caso o ambiente não forneça os recursos de comunicação com a qualidade necessária, então outros níveis

diferenciados de qualidade, menos exigentes em termos deste recurso, são oferecidos como de praxe. Seria interessante se o usuário pudesse realizar a configuração do *framework* (definição do contrato e valoração dos perfis) através de uma interface gráfica.

Um passo importante foi dado em [Cardoso et al. 2006] onde o conceito de função utilidade foi adicionado ao CR-RIO. Naquele trabalho, a linguagem de contratos usada em [Corradi 2005] foi estendida de forma a comportar as notações `@ref` e `select*`. A primeira é uma variável de ambiente que permite o contrato ser escrito de forma abstrata, ou em outras palavras, se valendo da técnica de virtualização. Essa variável é iniciada em tempo de execução pelo operador `select*`, que recebe como parâmetros um perfil que parametriza uma função utilidade e um conjunto de objetos sobre os quais deve operar. Apesar dos autores relatarem que a sintaxe ainda está sendo desenvolvida, podemos dizer que:

```
link client to @server=select*(hqProfile, servers@environment)
```

sugere a ligação do cliente a um servidor `@server` resolvido apenas em tempo de execução. O servidor é escolhido pelo operador `select*`. Segundo o 3.4, esse operador utiliza a parametrização `optim` para configurar a função utilidade `selPol` definida pela `QoSCategory Server` e utilizada pelo perfil `hqProfile` para escolher um servidor dentre os disponíveis no domínio `environment`.

Código 3.4: Perfil utilizando uma função utilidade

```
category {
  selPol (random, bestMem, bestCPU, optim);
} Server;

5 profile {
  Server.selPol: optim;
} hqProfile;
```

Uma crítica à abordagem orientada a uma máquina de transições é que os estados, ou seja, os possíveis níveis diferenciados de qualidade da aplicação, devem ser previamente definidos e explicitamente citados no contrato. Isso obriga o desenvolvedor a discretizar o espaço de soluções em poucos pontos, do contrário, seriam inúmeras as cláusulas `service` do contrato e as possíveis transições da máquina de estado. Além disso, o desenvolvedor deve escolher, dentre os estados possíveis, aquele que é prioritário, ou seja, aquele que o *framework* tentará impor em primeiro lugar. Essa solução estática para a máquina de transições é recomendada em [Petrucci e Loques 2007] apenas para serviços simples.

Para serviços mais complexos, chamada pelos autores de negociação dinâmica, a sugestão é dar a função utilidade uma responsabilidade maior (como é sugerido também em [Cheng et al. 2006]): em vez de ser usada apenas para escolher componentes individualmente, ela pode substituir a máquina de estados com transições pré-definidas e ser usada para escolher qual nível de qualidade deve ser oferecido, dadas as disponibilidades dos recursos demandados por ele e as preferências dos usuários.

3.3 Proposta de Cheng

Em [Cheng et al. 2006], as funções utilidade são aplicadas de forma a capturar a experiência do administrador e as relações de compromisso que ele leva em conta quando realiza adaptações em uma aplicação *web*.

Segundo os autores, a experiência obtida no desenvolvimento do *framework* Rainbow [Garlan et al. 2004] mostrou que capturar as preferências dos usuários através de *scripts* é interessante quando apenas uma dimensão de QoS é levada em conta. Porém, quando duas ou mais dimensões, geralmente conflitantes, são importantes para promover a adaptação e satisfazer ao usuário, torna-se muito complexo representar as relações de compromisso, escolhas e a preferência dos usuários através desses *scripts* ou de programação baseada em uma máquina de regras, tais como [JESS].

Os autores ilustram seu argumento com um cenário onde o administrador de um sistema *web* é responsável por promover adaptações de forma que a aplicação agrade aos usuários e aos proprietários daquele *site* de notícias, durante diferentes demandas de tráfego. O *site* frequentemente publica notícias de alta popularidade e isso faz com que os servidores nos quais o *site* está hospedado, experimentem um aumento de requisições.

Observando como o administrador trabalha, os autores observaram que basicamente ele adiciona novos servidores ao *cluster* até o limite do orçamento determinado pelos proprietários. Se ainda assim os clientes estiverem verificando um tempo de resposta grande, então o administrador pode ajustar a aplicação para que ela ofereça conteúdo textual em substituição ao conteúdo gráfico. Quando a demanda diminui, o administrador realiza essas operações no sentido contrário. Desse modo, quatro ações de adaptação são possíveis: (1) mudar o conteúdo de gráfico para textual, (2) mudar o conteúdo de textual par gráfico, (3) adquirir novos servidores e (4) vender servidores.

Visivelmente, os interesses dos usuários e dos proprietários são antagônicos e competem entre si, a saber: (a) o tempo de resposta, (b) a qualidade do conteúdo e (c) o

orçamento. É uma atribuição do administrador cuidar para que todos sejam satisfeitos. Por exemplo, quando ele percebe que o tempo de resposta observado pelos clientes está acima do desejado, o administrador pode emergencialmente ajustar os servidores para que entreguem conteúdo em modo texto. Depois, observando que o *cluster* de servidores está pequeno, ele pode adquirir novos servidores e, só então, ajustá-los para servir conteúdo gráfico novamente.

Os autores enfatizam que o administrador observa a situação da aplicação entre duas adaptações consecutivas, visto que é preciso saber se a última ação teve o efeito esperado e se, durante sua realização, outro problema mais crítico aconteceu. Em cada ponto de decisão, o administrador leva em conta vários fatores: se o orçamento está sendo obedecido, quanto tempo a próxima ação de adaptação deve levar, qual a expectativa de satisfação adicional ela deve trazer e como essa ação se situa em comparação com outras ações alternativas¹.

O primeiro passo realizado pelos autores para resolver o problema é reduzir o espaço observável para um número de estados que possa ser processado, ou seja, o orçamento é dividido em duas faixas (abaixo do orçamento e acima do orçamento) e o tempo de resposta é dividido em três faixas (baixo, médio e alto). Se levarmos em consideração que o sistema pode operar em dois modos (gráfico e textual) então o espaço observável, inicialmente infinito, ficou reduzido a doze estados².

Em seguida, os autores avaliam como cada uma das quatro adaptações (ou táticas) possíveis afetam o ambiente, ou seja, (i) como cada uma fará variar as dimensões de QoS consideradas (o tempo de resposta, a qualidade do conteúdo, o orçamento e a atenção do usuário); e (ii) como essa variação afeta a satisfação dos usuários e proprietários. Para realizar a primeira tarefa os autores valoram estaticamente e de forma antecipada essas dimensões de QoS. Por exemplo, a tática que troca o conteúdo de gráfico para textual é tida como uma ação que aumenta o tempo de resposta, melhora a qualidade, não influi no orçamento e provoca uma desatenção média do usuário (a dimensão de QoS “desatenção” reflete basicamente quanto tempo aquela tática leva para ser executada). Como estamos buscando uma comparação entre táticas, essa previsão/estimativa não precisa ser real em

¹Nesse momento, os autores estabelecem um vocabulário no qual afirmam que tática é um conjunto de ações executadas sem pontos intermediários de observação e decisão; e estratégia é definida como blocos de ações separados por pontos intermediários de observação e decisão. Dessa forma os autores modelam o problema como uma árvore de táticas, onde uma observação é feita antes de ser decidido qual ramo tomar; e não como uma máquina finita de estados.

²A instabilidade do sistema e as possíveis oscilações originadas da inclusão desses limites rígidos parece não ser uma preocupação dos autores. Na Seção 2.6 abordamos detalhadamente as implicações e mostramos as técnicas presentes na literatura para contornar esse problema.

valores absolutos, mas apenas em valores relativos.

Para realizar a segunda tarefa, os autores valem-se de curvas de satisfação, elicitadas previamente com o usuário, que mostram como cada dimensão de QoS influi na sua satisfação. Por exemplo, os clientes podem estar mais interessados na qualidade do conteúdo do que no tempo de resposta; já os proprietários devem se importar principalmente em operar dentro do orçamento. A partir daí, uma função utilidade realiza uma soma ponderada para comparar as táticas e retorna aquela que maximiza a satisfação. Mais detalhes sobre curvas de satisfação serão dados nas Seções 3.4 e 4.1.

É criticável conforme afirmam alguns trabalhos da literatura, a forma como os autores antecipadamente valoram as dimensões de QoS de uma tática antes dela ser executada, por exemplo, afirmando como ela fará variar o tempo de resposta. Neste caso as táticas são claramente ortogonais e assim também são seus atributos. Em cenários mais complexos isso pode não acontecer [Huang 2004].

Para exemplificar, os autores propõem uma situação onde o *site* experimenta um pico de demanda. É observável que o tempo de resposta é alto, o sistema opera abaixo do orçamento e que seu conteúdo é gráfico. Nesse caso, o *framework* deve perceber que condições foram violadas e deve listar quais táticas podem solucionar o problema. Duas táticas são possíveis e uma delas deve ser executada automaticamente pelo *framework*: mudar para conteúdo textual ou incrementar o número de servidores. Assim, sobre esse conjunto restrito de táticas, a função utilidade é aplicada para escolher qual a melhor.

O uso de funções utilidade, segundo o artigo, permite ao *framework* escolher entre táticas possivelmente conflitantes levando em conta quatro dimensões de QoS e relações de compromisso (*trade-offs*) entre elas, além das preferências dos usuários. Como lado negativo, o *framework* deve levantar as curvas de satisfação, pesos para o cálculo da soma ponderada e as dimensões de QoS que influenciam a satisfação dos usuários.

O interessante dessa abordagem é que a função utilidade foi usada para escolher entre táticas (transições) e não entre elementos físicos de um mesmo tipo, como durante o mapeamento físico em [Huang 2004]. Até então, uma função utilidade escolhia, por exemplo, o melhor servidor dentre um conjunto de servidores disponíveis. Uma função utilidade que escolhe uma transição pode ser pensada como um operador arquitetural pré-definido, o *framework* o executa e pode alterar a aplicação de inúmeras formas ao mesmo tempo: adicionando um servidor, alterando um canal de comunicação e trocando os servidores de modo gráfico para modo texto. Em [Petrucci e Loques 2007] essa expansão foi aplicada a linguagem de contratos do CR-RIO para servir de alternativa à máquina de transições,

definida estaticamente, usada por aquele *framework* até então.

3.4 Aura

O trabalho de [Poladian et al. 2006] também é baseado no uso de funções utilidade para guiar dinamicamente o estabelecimento e a adaptação da aplicação. O autor cria o conceito de tarefa de usuário que é elevado a uma entidade de primeira ordem, guardando suas preferências. O usuário tem a oportunidade de informar quais dimensões de QoS influenciam a qualidade e qual a importância relativa de cada uma.

Por exemplo, se a tarefa consistir na tradução simultânea de palavras então as dimensões de QoS importantes podem ser a latência, o tamanho do vocabulário e a acurácia da tradução. Além disso, através de telas como as exibidas na Figura 3.2, o usuário pode quantificar quais dimensões são mais importantes para ele, na execução da tarefa. As réguas (*sliders*), à direita, expressam o peso da dimensão de QoS no cálculo da função utilidade. Se a dimensão for contínua, e.g. a latência de tradução, então o *framework* apresenta uma função sigma onde o usuário pode, deslocando os cursores, qualificar os intervalos. Veja que se a latência for acima de 3 segundos aproximadamente, ele não é satisfeito (ícone X na Figura 3.2a). Se a dimensão for discreta, então o *framework* permite ao usuário especificar intervalos. No exemplo, o usuário definiu os intervalos *small*, *medium* e *large* para a dimensão vocabulário. Veja que um vocabulário médio ou grande satisfazem igual e plenamente ao usuário (ícone “*smiley*” na Figura 3.2b).

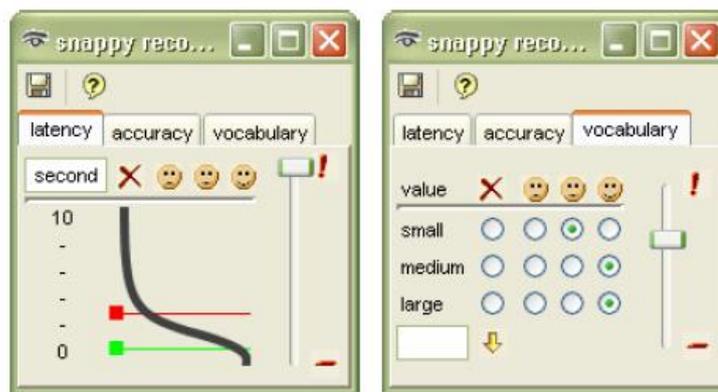


Figura 3.2: Elicitação de requisitos e preferências

Assim, caso o *framework* perceba a indisponibilidade ou falha do componente tradutor, ele irá promover a adaptação privilegiando componentes que tenham principalmente baixa latência, visto que a régua da Figura 3.2a tem grande valor. O usuário pode alterar essas preferências a qualquer instante. Para não desviar a atenção do usuário da

execução de sua tarefa propriamente dita por muito tempo, o autor prevê a existência de configurações (*presets*) de forma a acelerar a parametrização do *framework*. O usuário pode selecionar uma configuração chamada “tradução rápida” já previamente salva no *framework* e, ao selecioná-la, parametrizar a preferência de várias dimensões de QoS concomitantemente.

Enquanto o local onde as preferências do usuário devem ser mantidas pelo *framework* parece ser um consenso entre as propostas que abordam essa preocupação (os autores entendem que a função utilidade é a representação formal que melhor captura as preferências do usuário), elicitá-las sem que o usuário fique entediado ou longe da sua atividade fim tem se mostrado complexo na literatura, como mostram [Poladian et al. 2006, Capra et al. 2005]. No primeiro artigo, esse problema é resolvido através de telas, como mostradas na Figura 3.2. No segundo, a função utilidade é guardada em uma estrutura XML, juntamente com as dimensões de QoS, como mostrada no Código 3.5, mas nenhuma menção é feita sobre como esses parâmetros são levantados.

Código 3.5: Descrição da função utilidade em XML

```
<utility combine="product">
  <QoSDimension name="latency" type="float">
    <function type="sigmoid" weight="1">
      <thresholds good="1" bad="3" unit="second"/>
5    </function>
    </QoSDimension>
  <QoSDimension name="accuracy" type="float">
    <function type="sigmoid" weight="0.7">
      <thresholds good="90" bad="40" unit="percent"/>
10    </function>
    </QoSDimension>
  <QoSDimension name="vocabulary" type="enum">
    <function type="table" weight="0.7">
      <entry x="small" f_x="0.8"/>
15    <entry x="medium" f_x="1"/>
      <entry x="large" f_x="1"/>
    </function>
    </QoSDimension>
</utility>
```

Isto posto, [Poladian et al. 2006] sugere que a função utilidade seja montada através de um produtório entre (i) as preferências do usuário pelos fornecedores de um certo serviço - MS Word, Notepad, Textpad e Vi são aplicações fornecedoras do serviço de

edição de texto - e (ii) as preferências do usuário pelas dimensões de QoS que afetam esse serviço, num total de $a \times b$ possibilidades. A representação através de um produtório é interessante pois tem o significado de um AND lógico. Por exemplo, se um fornecedor citado pelo usuário estiver indisponível no ambiente onde o usuário executar sua tarefa, o valor retornado pela função utilidade calculada sobre as configurações que envolverem este fornecedor ausente retornará zero, independentemente do valor dos outros termos do produtório.

Assim, uma vez que (i) tem menos termos, essa parte é mais fácil de ser calculada e, portanto, pode ser usada como o limite superior do produtório total. Dessa forma, o algoritmo encontra, dentre as a configurações aquela que maximiza a primeira parte do produtório e, em seguida, varre b configurações, testando, no pior caso, $a+b$ possibilidades ao invés $a \times b$ possibilidades.

Estabelecida uma configuração, o *framework* monitora continuamente se existem outras alternativas que ofereçam um valor maior para função utilidade. Para evitar que ele fique continuamente alternando entre configurações, os autores introduziram o conceito de “custo de reconfiguração” que expressa a intolerância do usuário às adaptações feitas na aplicação. Com um pequeno valor, a aplicação impõe rapidamente a melhor configuração assim que ela existir, o que pode levar a uma alta taxa alterações e à distração do usuário. Com um valor alto, a configuração atual continua corrente, embora possam haver outras melhores, ou seja, a aplicação fica altamente estável, mas freqüentemente menos otimizada.

Como alternativa a essa estratégia, o mesmo grupo estuda em um artigo incipiente [Poladian et al. 2005] uma forma do *framework* prever através de processos estocásticos, ao longo de toda a tarefa do usuário, a disponibilidade dos recursos necessários ao estabelecimento da aplicação. Dessa forma, o sistema seria pró-ativo em vez de reativo às mudanças. Segundo o artigo, essa abordagem é interessante porque uma seqüência de adaptações pode não ser a melhor quando consideramos toda a duração da tarefa do usuário, embora cada uma delas seja uma decisão acertada no momento em que foram realizadas. Os processos estocásticos são simplificados à cadeias de Markov, assumindo-se que o valor disponível futuro de uma dimensão de QoS, por exemplo a largura de banda de um enlace de comunicação, depende apenas do seu valor atual. O trabalho termina lembrando que essa é uma abordagem nova e cita dois problemas que podem inviabilizar uma solução computacional escalável: o intervalo pequeno entre medições e a granularidade dos possíveis valores assumidos pelas grandezas, quando modeladas discretamente.

3.5 Olympus

Recursos altamente dinâmicos, ambientes heterogêneos e políticas específicas dificultam o estabelecimento da aplicação, uma vez que esses fatores não podem ser previstos pelo desenvolvedor em tempo de projeto. Com essas premissas, a proposta de [Ranganathan et al. 2005] visa construir um vocabulário comum que garanta a portabilidade em diferentes ambientes, que é uma preocupação das aplicações pervasivas.

Para isso, são criadas hierarquias ontológicas - que podem ser entendidas como diagramas de classes onde cada entidade, exceto a raiz, pode ter múltiplos pais. Nessas hierarquias são definidas as entidades: *Services*, *Applications*, *Devices*, *Physical objects*, *Locations* e *Users*. A hierarquia da entidade *Device* é mostrada na Figura 3.3. Repare que uma tela de plasma pode servir para exibição de dados como também para entrada, sendo sensível ao toque ou a uma caneta especial.

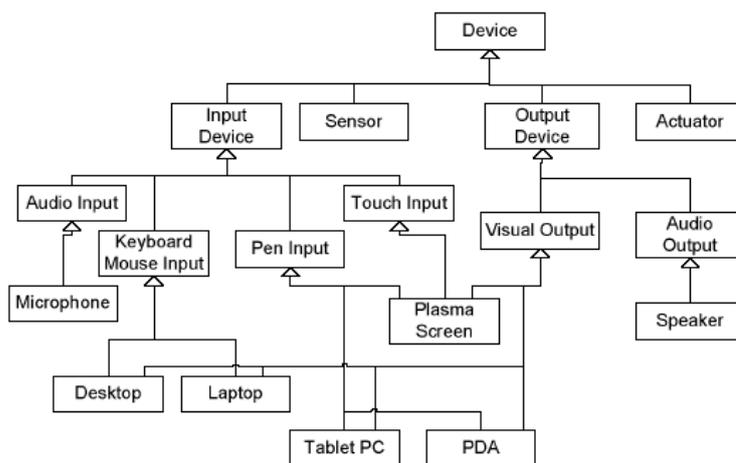


Figura 3.3: Hierarquia ontológica da entidade virtual *Device*

No momento de definir a arquitetura da aplicação, o desenvolvedor tem o auxílio das entidades virtuais e dos métodos de alto nível definidos ontologicamente³. Ele pode dispor, inclusive, de uma aplicação gráfica que o ajude a navegar pela hierarquia e selecionar as entidades mais apropriadas para a sua necessidade. No Código 3.6, vemos que a aplicação é composta de apenas uma classe `SlideShow` que deve exibir o arquivo `Olympus.ppt`. Dependendo do que o ambiente `as1` (cuja a definição não foi mostrada nessa

³Uma entidade virtual tem uma descrição OWL, que é um dos padrões semânticos de descrição da Internet. Uma descrição OWL revela quais atributos uma entidade pode ter, quais tarefas ela pode realizar e quais associações ela tem com outras entidades. A ontologia define relações entre diferentes conceitos. Por exemplo, a associação: `requiresDevice(PowerPointViewer) = PlasmaScreen X Desktop X Laptop X TabletPC` mostra que um `PowerPointerViewer` só pode ser executado caso, pelo menos, um dos quatro recursos esteja disponível. Um outro exemplo pode ressaltar um relacionamento entre uma mesa e suas cadeiras.

listagem) forneça, essa classe será mapeada em um objeto diferente, como por exemplo `MS PowerPoint` ou `InternetExplorer`. A última linha da listagem realiza o mapeamento físico propriamente em dito, que será detalhado a seguir.

Código 3.6: Uso de entidades virtuais

```
Application appl;  
appl.hasProp("class","SlideShow");  
appl.hasProp("file","Olympus.ppt"  
//appl is started in Active Space as1  
5 //in the "best" possible configuration  
appl.start(as1);
```

Para descobrir como uma tarefa pode ser executada da “melhor” forma possível, a proposta se baseia em restrições fornecidas pelo usuário, nos recursos disponíveis no ambiente, nas políticas próprias do ambiente e no contexto de operação. Primeiro, para cada entidade utilizada na descrição da aplicação, o *framework* consulta um servidor encarregado de guardar as hierarquias e de resolver consultas ontológicas. O servidor então retorna uma lista de classes que são semanticamente (graças à abordagem ontológica) semelhantes àquela passada como parâmetro na consulta. No segundo passo, o *framework* aplica filtros de forma a reduzir a lista retornada a um conjunto de classes viáveis. Esses filtros são criados a partir da própria ontologia, da política do ambiente ou por iniciativa do desenvolvedor. Como exemplo de restrições, veja o Código 3.7. Nesse caso, o desenvolvedor espera que o *framework* descubra no ambiente um dispositivo capaz de realizar uma exibição, sendo que ele deve estar localizado próximo ao usuário `user1` e ter resolução de `800x600 pixels`.

Código 3.7: Restrições em nível de classe

```
Device device1;  
device1.hasProp("class","VisualOutput");  
device1.hasProp("location",user1.getProp("location","room"));  
device1.hasProp("resolution","800x600");  
5 device1.hasMetric("distance",user1,"ascending");
```

No terceiro passo, o *framework* deve descobrir quais instâncias estão disponíveis no ambiente, para isso ele consulta um repositório que contém todas as instâncias existentes. No quarto passo, o *framework* checa restrições no nível das instâncias de forma a reduzir ainda mais o número de possibilidades. No quinto passo, caso ainda haja mais de uma possibilidade, para cada uma das instâncias resultantes no passo anterior, é aplicada uma função utilidade de maneira a selecionar aquela que mais satisfaça ao usuário. Na última

linha da listagem anterior, pode ser verificado que a distância entre o dispositivo e o usuário será uma métrica utilizada para escolher o melhor candidato.

Quanto à montagem da função utilidade, o artigo mostra que podem fazer parte da sua definição métricas como: a localização da entidade, as tarefas suportadas por ela, seu estado de operação e o contexto do ambiente. Como exemplo dessa última métrica, o usuário pode preferir assistir um vídeo em um monitor disponível na sala ou na tela de seu PDA, caso ele esteja sozinho ou acompanhado na sala, respectivamente. Algumas dimensões da função utilidade (como a localização de um componente) são quantitativas, enquanto outras (como aquelas relacionadas ao contexto do ambiente) são qualitativas. Para elas, segundo o artigo, o desenvolvedor, o usuário ou o administrador do ambiente deve definir em suas políticas quais entidades são melhores que outras em uma dada dimensão, permitindo ao *framework* realizar uma comparação entre elas. O artigo termina concluindo ser importante usar funções utilidade multi-dimensionais mais expressivas.

3.6 Conclusão do capítulo

Os *frameworks* são uma solução recorrente para garantir os requisitos não-funcionais de operação de uma aplicação. A descrição arquitetural expressa através de receitas ou de uma API, a separação clara entre mecanismos (que operam a adaptação) e políticas (que guiam os mecanismos) e o emprego das funções utilidade para armazenar as preferências dos clientes e selecionar o melhor recurso têm se mostrado um conjunto de técnicas com potencial para impor e manter as aplicações no futuro. Além disso, a função utilidade já aparece em alguns trabalhos selecionando também a melhor transição de uma máquina de estados, onde cada transição pode ser considerada como um operador arquitetural de alto nível. A descrição dos recursos disponíveis no ambiente de forma que possa ser processada pelos *frameworks* têm recebido bastante atenção por parte dos pesquisadores, diferentemente de técnicas que permitam a eliciação das preferências dos usuários: frequentemente os artigos não explicam como foram elicidas as preferências utilizadas em suas propostas.

Capítulo 4

Exemplos de aplicação

Com os conceitos mais importantes expostos e os trabalhos relevantes para este estudo citados, neste capítulo listamos aplicações realísticas em que propomos empregar uma função utilidade para realizar o mapeamento físico, ou seja, resolver um recurso virtual presente na descrição arquitetural em um recurso que efetivamente está disponível no ambiente. Será feita uma análise, caso a caso, mostrando as deficiências dos métodos de escolha empregados atualmente nestas aplicações e como a função utilidade pode resolver o problema de forma flexível e computacionalmente barata. Além disso, mostraremos quais fatores podem influenciar a montagem da função utilidade, ou seja, quais dimensões de QoS são relevantes para a qualidade da aplicação. Preocupações tais como a estabilidade da aplicação e a monitoração dos recursos são comuns a todas as aplicações e serão apresentadas no fim deste capítulo.

4.1 Escolha do melhor ponto de acesso

O objetivo desta aplicação é permitir que um cliente móvel escolha o melhor ponto de acesso (AP) disponível para se conectar em uma rede sem-fio. Essa escolha será realizada com o suporte de uma função utilidade que será calculada sobre as dimensões de QoS e um fator de qualidade reportado individualmente pelos APs ao cliente, conforme será explicado a seguir.

A Figura 4.1 mostra um exemplo de uma rede sem-fio com algumas características que devem ser observadas. A primeira é que os APs são ligados à rede local através de um cabeamento estruturado, diferentemente de uma rede *mesh* [de Albuquerque et al. 2006], significando que a interface aérea só existe entre os APs e os clientes. Isso reduz as possibilidades de interferências eletromagnéticas quando comparadas às que uma rede *mesh* está

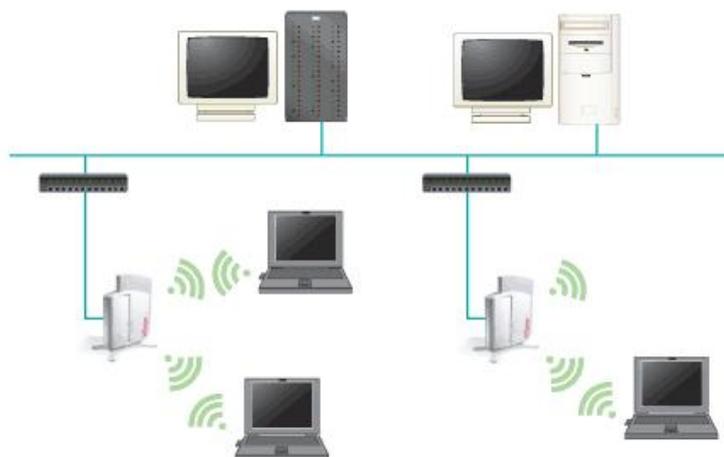


Figura 4.1: *Esquemático de uma rede sem-fio*

sujeita. Outra característica é que, por serem cabeados, os APs são estáticos deixando a mobilidade deste cenário apenas para os clientes. Eles por sua vez, têm tipicamente poucos recursos de processamento e restrições em termos de energia. O protocolo implementado na interface aérea desta rede é geralmente a versão 802.11b ou 802.11g que permite canais de comunicação de até 11Mbps e 54Mbps de velocidade nominal, respectivamente [da Conceição e Kon 2006]. Para garantir que haja cobertura de sinal em todos os locais onde a rede deve operar, freqüentemente há um sobre-dimensionamento do número de APs, o que invariavelmente leva os clientes a receberem o sinal de mais de um AP, dependendo de sua localização. As soluções empregadas comercialmente fazem com que o cliente escolha o AP com base apenas na intensidade do sinal captado pelo primeiro, o que faz com que um pequeno desequilíbrio de intensidades ocasione um forte desbalanceamento de carga entre os APs [Judd e Steenkiste 2002].

Para resolver esse problema, propomos o emprego de uma função utilidade que leve em conta não só a intensidade do sinal mas também outras dimensões de QoS relevantes para a realização da aplicação na qual o cliente está interessado. A definição de quais dimensões influenciam a qualidade da aplicação depende do recurso sendo escolhido e, em geral, o desenvolvedor/provedor da aplicação é o mais capacitado para fazê-la. Dimensões tipicamente utilizadas por clientes nesse cenário são a largura nominal do canal de comunicação, a intensidade do sinal e a porcentagem de bytes retransmitidos (erros). Além dessas dimensões de QoS, poderá ser levado em conta pela função utilidade um fator de qualidade calculado pelo AP de forma sigilosa, com base apenas em informações locais/privadas a ele. O fator de qualidade considera apenas meta-dimensões: a freqüência com que as métricas do AP são coletadas por ele, a última data/hora em que elas

foram coletadas, a probabilidade de corretude, resolução e precisão. Todas essas meta-dimensões de QoS são combinadas pelo AP, que retorna então o fator de qualidade para o cliente¹. Dessa forma, o fator de qualidade pode ser visto como uma forma de o recurso que está sendo escolhido, expressar sua qualidade apenas em termos de meta-dimensões de QoS, ou seja, sem que seja contaminado pelas dimensões de QoS propriamente ditas (especificadas pelo desenvolvedor do serviço) ou pelas preferências do usuário.

De posse do fator de qualidade e de outras dimensões de QoS, o cliente compõe a função utilidade que selecionará o melhor AP através de uma média ponderada, conforme será visto adiante. Esta média deve ser ponderada porque suas parcelas (o fator de qualidade e as dimensões de QoS) podem ter relevância diferenciada para o desempenho da aplicação. Por exemplo, em uma aplicação de *music-on-demand*, a largura de banda é mais importante do que o *jitter* no canal de comunicação, pois esse último é tipicamente contornado com técnicas de *bufferização*.

O fato do cliente se valer do fator de qualidade para compor a função utilidade tem dois aspectos. O primeiro tem a ver com o peso atribuído a esse fator no cálculo da média ponderada: se o cliente se conectar a um AP de alto fator de qualidade e perceber um mau desempenho do serviço, o cliente pode autonomamente reduzir a credibilidade do fator de qualidade diminuindo o peso atribuído a ele nas próximas escolhas que fizer. Dessa forma, caso deseje, o cliente pode correlacionar a qualidade obtida do serviço apenas com o fator de qualidade reportado pelos APs.

O segundo aspecto tem a ver com o sigilo envolvido no cálculo do fator de qualidade feito pelo AP, o que tem dois desdobramentos. O primeiro é que na hipótese de uma atividade de manutenção no AP, o projetista da rede pode arbitrariamente reduzir seu fator de qualidade com o objetivo de desviar o tráfego daquele AP. Desta forma, os clientes que solicitaram o fator de qualidade tenderão a não selecionar aquele AP. O segundo desdobramento é que o desenvolvedor/provedor pode usar de seu conhecimento específico de redes sem-fio (*service-specific knowledge* segundo [Huang 2004]) para manipular o conceito do fator de qualidade colocado anteriormente. Por exemplo, é mostrado em vários trabalhos [Balazinska e Castro 2003, Hernandez-Campos e Papadopouli 2005] que a ocupação da largura de banda de um AP não se dá pelo número de usuários conectados a ele, mas é devido principalmente à identidade dos usuários conectados; em outras

¹Essas meta-dimensões foram introduzidas em [Huebscher e McCann 2004a] na tentativa de definir a Qualidade de Contexto ou, como também encontrado em seus trabalhos, Qualidade da Informação. São muito úteis para comparar a qualidade de componentes que oferecem um mesmo serviço, como por exemplo, diversos componentes espalhados por uma casa que oferecem um serviço de localização de pessoas.

palavras, o número de usuários e o tráfego gerado por eles não são fortemente correlacionados. Ao detectar que um usuário com o perfil de grande consumidor de banda pediu o envio do fator de qualidade, o AP pode reportar um valor baixo com o objetivo de evitar que esse usuário se conecte a ele para o benefício dos usuários já conectados ao AP. E mais, se for interessante em termos de administração da rede, uma entidade central (tomando-se o cuidado de que ela não se torne um gargalo ou um ponto central de falha para o sistema) pode compartilhar as informações de consumo de banda dos clientes entre os APs (ver Seção 5.2.1). Assim, um usuário grande consumidor de banda em um AP será conhecido também pelos outros APs da rede, que também podem reduzir seu fator de qualidade antes de enviá-lo para o cliente. Esse controle de admissão de clientes contribui para o balanceamento da carga do sistema, evitando o aparecimento de *hot-spots* [Balachandran et al. 2002].

Do ponto de vista do cliente, ele recebe o fator de qualidade e as outras dimensões de QoS apenas dos APs que ele conseguir escutar. Em outras palavras, a interface *wireless* do cliente pode ser pensada como a implementação de um serviço de nomes e diretórios onde os recursos disponíveis no ambiente são cadastrados e seus atributos valorados pelo serviço de monitoração². Essas informações deverão ser empregadas em uma função utilidade implementada na forma de uma média ponderada, conforme dito anteriormente. É interessante ressaltar que essa função utilidade será parametrizada com as preferências do cliente. De acordo com a aplicação que ele executar, ele deverá privilegiar algumas dimensões de QoS em detrimento de outras, originando assim os pesos dessa média ponderada. Outro ponto de parametrização será o grau de satisfação obtido com o valor instantâneo de cada uma das dimensões de QoS, que será dado por “funções de satisfação”. Assim, a função utilidade pode ser representada na Equação 4.1³:

$$U = p_1 \times s_1(d_1) + \dots + p_n \times s_n(d_n) + p_{n+1} \times fatQualidade \quad (4.1)$$

onde:

- p_i : preferência do usuário pela dimensão de qualidade i ou, também, a importância que o usuário dá àquela dimensão de QoS, representado por um número real entre

²Ao responder à aplicação quais são os APs ativos naquela localidade e com qual intensidade de sinal estão operando, a interface *wireless* atua como um serviço de descoberta de recursos, tais como o NSSD de [Huang 2004] ou o JNDI da arquitetura Java.

³Ao invés de operar a função utilidade sobre cada um dos APs disponíveis, o cliente pode opcionalmente ordenar os APs segundo cada uma das dimensões de QoS e, então, calcular a utilidade apenas nos $n \times m$ APs, onde n é o número de APs em cada lista ordenada e m é o número de listas ordenadas (dimensões de QoS).

$[0, \max]$.

- **si()**: função de satisfação do usuário pela dimensão de qualidade i . Mapeia um valor de uma dimensão de QoS medido pelo sistema em um número real normalizado entre $[-1, 1]$ ⁴. Se a dimensão de qualidade for contínua, trata-se de um gráfico ou uma tabela caso contrário. Em ambos os casos, sugerimos ser elicitada por interfaces gráficas como é feito no trabalho de [Poladian et al. 2006].
- **di**: valor da dimensão de QoS, tipicamente atraso de comunicação, intensidade do sinal, largura de banda, processamento e memória disponíveis medido pelo serviço de monitoração.
- **fatQualidade**: é o fator de qualidade calculado de forma sigilosa pelos APs, com base em meta-dimensões de QoS.

Dessa forma, para a seleção do melhor AP, propomos o emprego de uma função utilidade por entender que ela constitui uma técnica que leva em conta as preferências dos usuários (que ponderam dimensões de QoS frequentemente conflitantes) e a disponibilidade dos recursos que impactam a qualidade da aplicação. Outra virtude dessa técnica é a facilidade de implementação e a pouca quantidade de processamento exigida, propiciando ser utilizada em pequenos dispositivos com recursos computacionais e de energia restritos, tais como PDAs.

Cada uma das funções de satisfação **si()** pode ser da forma esboçada na Figura 4.2. É interessante notar o ponto no eixo das abscissas em que a função deixa de ser negativa e passa a ser positiva: ele marca exatamente a menor quantidade de recurso disponível que vai satisfazer ao cliente. Por exemplo, o cliente deseja usar um codificador para áudio GSM, que exige pelo menos 13kbps de largura de banda do canal de comunicação [Bhatti e Knight 1999] entre o cliente e o AP. Esse ponto está marcado como **BWreq** no gráfico a seguir. O ponto **BWmax** marca o início do intervalo no qual o gráfico exibe um comportamento assintótico, ou seja, qualquer AP com uma largura de banda maior que **BWmax** irá satisfazer igualmente ao cliente, no que diz respeito a esta dimensão de QoS. Obviamente, dependendo da aplicação a função de satisfação poderá ser representada por uma forma gráfica mais simples, como por exemplo uma reta; um exemplo recorrente é o uso de uma função mínimo-máximo do tipo $si(di) = (di - \min) / (\max - \min)$ onde **max** e **min** são os limites de variação da dimensão **di** de QoS considerada⁵. Assim, acreditamos

⁴ou entre $[0, 1]$ dependendo da implementação.

⁵No caso da satisfação do cliente diminuir com o aumento da dimensão de QoS monitorada, como por exemplo o atraso de rede, a função de satisfação seria decrescente da forma $si(di) = (\max - di) / (\max - \min)$

que o uso de funções de satisfação seja uma solução muito eficiente para capturar as preferências do usuário por uma dimensão de QoS específica. Além disso, seu uso resolve um problema recorrente: dispensa o uso na função utilidade de constantes de ajuste dimensional, visto que uma dimensão de QoS pode ser expressa em metros e a outra em segundos, por exemplo. Também não é necessário o emprego de técnicas como as distâncias Euclideana ou de Mahalanobis, usadas em [Huebscher e McCann 2005]. Naquele trabalho, o cálculo da distância de Mahalanobis foi adaptado para medir quantas vezes o valor de uma dimensão de QoS é maior do que o desvio padrão que ela apresentou dentro de uma janela de amostras de tamanho pré-definido, resultando em um quociente adimensional que entra no cálculo da função utilidade.

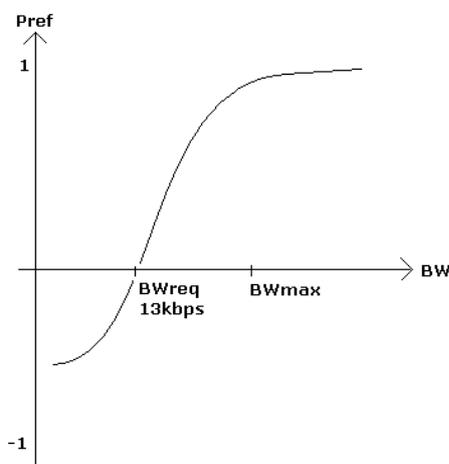


Figura 4.2: *Função de satisfação para a eliciação de preferências*

Vale observar que a satisfação do usuário por uma dimensão de QoS pode resultar em um número negativo. Nesse ponto pode haver duas abordagens e a escolha depende da aplicação. Na primeira, uma satisfação negativa simplesmente decrementa a satisfação total que o usuário tem por aquele recurso. Na segunda abordagem, uma única satisfação negativa acaba por inviabilizar a escolha daquele recurso, não importando se outras satisfações positivas compensem a negativa. Muitas vezes, se um canal de comunicação tiver uma disponibilidade de largura de banda inferior ao necessário, ele não deve ser escolhido, mesmo que apresente um sinal de grande intensidade. As funções de satisfação podem ser alteradas ao longo da utilização da aplicação, como por exemplo, através de uma interface gráfica projetada para elicitar as preferências do usuário, conforme sugerido nos trabalhos de [Poladian et al. 2006], onde apenas duas constantes são necessárias para definir a função sigmóide do gráfico da Figura 4.2.

Assim, acreditamos que uma função utilidade que leve em consideração outras dimensões de QoS além da intensidade do sinal, pode melhorar o desempenho do acesso

a uma rede sem-fio. Para validar nossa proposta, no Capítulo 5 exercitamos esta hipótese, primeiro através do processamento de *traces* reais de utilização de uma grande rede sem-fio e depois através de um simulador desenvolvido orientado a eventos discretos e parametrizado com distribuições realísticas de probabilidade.

4.1.1 Integração com o CR-RIO

Na seção anterior, mostramos como uma função utilidade pode auxiliar diretamente os clientes a escolherem o melhor AP. Agora vamos estender esse exemplo para mostrar como ela pode ser integrada a uma infra-estrutura de seleção.

Seja o mesmo cenário anterior em que clientes desejam acessar uma rede sem-fio. Aqui, eles têm o objetivo de executar uma aplicação de vídeo sob demanda e a rede conta com o CR-RIO como infra-estrutura mantenedora dos requisitos não-funcionais. O *framework*, ao impor o serviço de conexão, verifica que o cliente precisa estar conectado a um AP, disponível dentro de um conjunto de instâncias existentes no domínio do contrato que governa a aplicação. Dessa forma, uma função utilidade deverá selecionar o AP mais adequado com base em uma política que leve em conta as preferências do cliente e os recursos disponíveis no ambiente. A topologia da aplicação pode ser vista na Figura 4.3.

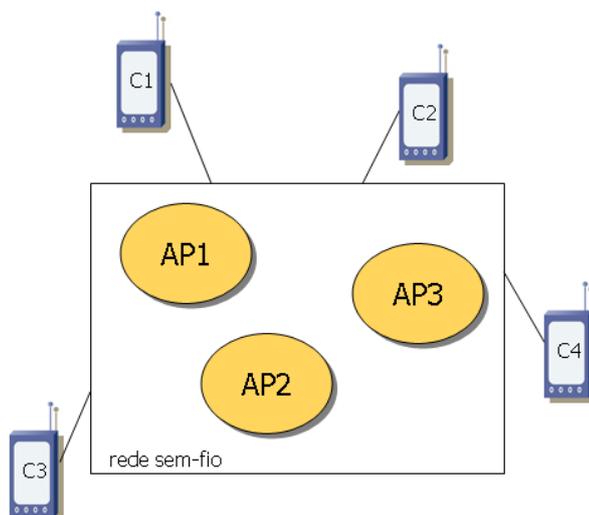


Figura 4.3: Descrição de uma rede sem-fio

Dessa forma, um contrato que seleciona o melhor AP segundo as preferências do cliente e o conecta a esse AP é mostrado no Código 4.1. Supomos que os clientes e os APs disponíveis na rede sem-fio (um módulo chamado aqui de `wirelessNetwork`) são previamente conhecidos pelo *framework*. Neste caso, o contrato prevê a imposição de

apenas um nível diferenciado de qualidade `sClient`, que justamente conecta o cliente ao melhor AP. Por brevidade, não são exibidas as `QoSCategories` e os prefixos.

Código 4.1: Descrição arquitetural e contrato CR-RIO

```

contract {
  service {
    instantiate client with clientProf;
    link client to @ref=select*(apUtil, ap@wirelessNetwork);
5 } sClient;
    negotiation {
      sClient -> out-of-service;
    }
  } cVideoOnDemand;
}

```

A ligação do cliente com o AP leva em consideração dimensões de QoS relevantes para a aplicação em questão, combinadas na forma de uma média ponderada. Neste exemplo, a seleção do recurso se dará através de uma função utilidade que retorna o AP com a melhor combinação de intensidade de sinal e largura de banda, ambas dimensões descritas na categoria `Transport`. O operador `select*` é usado para que de tempos em tempos o *framework* busque o melhor AP (referenciado por `@ref`). Essa busca é realizada no domínio de APs `wirelessNetwork` e parametrizada pela função de utilidade descrita por `apUtil`. Esse contrato utiliza a negociação estática do CR-RIO por ser uma negociação envolvendo poucos níveis diferenciados de qualidade.

Definimos a função `apUtil` com informações sobre as preferências (pesos) em relação às dimensões de QoS envolvidas para a escolha do melhor AP: 0,2 para a intensidade do sinal e 0,8 para a largura de banda. Também são descritos os atributos associados a uma referência para o conjunto de APs disponíveis no domínio `wirelessNetwork` e, portanto, envolvidos na seleção. Esses atributos são utilizados para parametrizar as funções de satisfação que devem ser operadas sobre cada valor instantâneo monitorado das dimensões de QoS, antes de serem utilizados na função utilidade. Neste exemplo, as funções de satisfação são do tipo mínimo-máximo conforme foram descritas na Seção 4.1. A função `apUtil` é descrita no Código 4.2 a seguir:

Código 4.2: Função utilidade expressa no CR-RIO

```

Utility apUtil {
  signal {
    s(signal) = (signal-min)/(max-min)
  } 0.2;
5 bandwidth {

```

```

    s (bandWidth) = (bandWidth-min) / (max-min)
  } 0.8;
  ap.attr =
  signal
10 [min=10dB, max=90dB],
    bandWidth
    min=0Mbps, max=2Mbps];
}

```

Para exemplificar a escolha realizada pelo *framework* CR-RIO, vamos supor que durante o funcionamento da aplicação que provê vídeo sob demanda, temos os seguintes APs disponíveis com as respectivas disponibilidades de recursos: AP1 (sinal: 15dB, bandwidth:1Mbps), AP2 (sinal: 50dB, bandwidth:0,5Mbps) e AP3 (sinal: 80dB, bandwidth:0,2Mbps). A seleção envolve calcular o valor da função utilidade para cada um dos APs disponíveis. Após o cálculo, temos o resultado:

$$U_{AP1} = (0.2 \times 0.0625) + (0.8 \times 0.5000) = 0.4125$$

$$U_{AP2} = (0.2 \times 0.5000) + (0.8 \times 0.2500) = 0.3000$$

$$U_{AP3} = (0.2 \times 0.8750) + (0.8 \times 0.1000) = 0.2550$$

Nesse exemplo, o AP1 foi o escolhido, devido ao peso maior dado a largura de banda do que à intensidade do sinal. Com isso, apesar de AP3 possuir uma forte intensidade de sinal, ele não foi selecionado devido a possuir uma pequena largura de banda. Dessa forma, durante o funcionamento da aplicação, conseguimos escolher o melhor AP com base nas preferências especificadas pelo usuário ou projetista no contrato.

4.2 Escolha do melhor servidor *web*

Nesta seção, consideramos um cenário em que um *hardware* especial, chamado *web switch*, se localiza a frente de um *cluster* de servidores *web* com a responsabilidade de dividir entre eles a carga proveniente das requisições HTTP dos clientes. Esse equipamento atua como um *proxy* entre os clientes e os servidores deixando a arquitetura utilizada pelo *cluster* transparente aos clientes. O artigo [Cardellini et al. 2002] nos oferece um completo estudo sobre arquitetura de servidores *web*.

Segundo os autores, os algoritmos de balanceamento de carga podem ser classificados

de várias formas. A primeira e mais importante classificação diz respeito ao tipo de informações que eles têm disponíveis para realizar o balanceamento, em outras palavras, até qual camada do modelo ISO/OSI eles processam a requisição HTTP para retirar essas informações (ver Tabela 4.1). Alguns simplesmente implementam uma escolha aleatória ou um mecanismo simples do tipo *Round Robin*. Outros, mais evoluídos, processam a requisição até a camada 4 (transporte) e se valem basicamente do endereço IP e da porta do cliente para realizar o balanceamento de carga e, com isso, podem enviar as requisições de um mesmo cliente para o mesmo servidor procurando se valer do princípio da localidade e do *cache* existente em cada servidor. Em razão da sua simplicidade, são extremamente rápidos na escolha do servidor, mas não conseguem balancear a carga igualmente em virtude da complexidade dos serviços e do conteúdo dinâmico da maioria das páginas da *web* atual⁶. Ainda mais complexos e, portanto, com maior potencial para realizar o balanceamento de carga, estão os algoritmos que processam a requisição até a camada 7 (aplicação) do modelo ISO/OSI o que dá a eles a URL que o cliente busca acessar, possibilitando técnicas de balanceamento sensíveis a mais atributos da requisição HTTP. Por essa complexidade adicional que permite uma melhor distribuição de carga, eles pagam o preço de terem um *throughput* cerca de 6 a 7 vezes menor do que aqueles que atuam até a camada 4.

Aplicação	Camada 7
Apresentação	Camada 6
Sessão	Camada 5
Transporte	Camada 4
Rede	Camada 3
Enlace	Camada 2
Física	Camada 1

Tabela 4.1: *Camadas do modelo ISO/OSI*

Algoritmos também podem ser classificados quanto ao contexto a que são sensíveis. Aqueles cegos ao contexto geralmente implementam o mecanismo de *Round Robin*. Aqueles cientes do contexto do cliente, por sua vez abrem a requisição HTTP até a camada 7 para ter a maior quantidade possível de informações conforme dito anteriormente. Por último, estão aqueles algoritmos cientes do contexto do cliente e dos servidores e, para

⁶Ainda nesta seção, discutiremos por que as páginas dinâmicas e a arquitetura dos servidores atuais diminuem a eficiência dos algoritmos que privilegiam a formação de *cache*.

isso, eles levam também em conta a disponibilidade de CPU e de memória dos servidores, por exemplo.

Quanto a localização destes algoritmos, eles podem ser classificados entre centralizados e distribuídos. O artigo de [Cardellini et al. 2002] e esta seção focam especialmente naqueles implementados no *web switch*, sendo portanto uma abordagem centralizada. Porém, o artigo cita também técnicas implementadas nos servidores, que atuam basicamente como filtros para as requisições entrantes e que, portanto, são consideradas constituintes de um algoritmo distribuído⁷.

Alguns algoritmos de balanceamento podem se beneficiar quando o *cluster* conta com servidores especializados/otimizados a um determinado tipo de serviço. Por exemplo, técnicas de *service partitioning* enviam as requisições por arquivos de vídeo para um determinado servidor e as requisições por arquivos de áudio para outro e assim por diante.

Uma preocupação constante dos algoritmos de balanceamento de carga é que sejam totalmente compatíveis com o protocolo HTTP. Desta forma eles podem ser inseridos no ambiente sem que clientes e servidores sofram alterações. Algumas alternativas como encontramos em [Karaul et al. 1998], abrem mão deste requisito e anexam agentes aos clientes e servidores no intuito de trocarem informações necessárias ao seu algoritmo que não poderiam ser transportadas transparentemente pelo protocolo HTTP.

Exposta a taxonomia proposta em [Cardellini et al. 2002], é importante conhecermos mais sobre o serviço de armazenamento de páginas *web*. Nossa intenção é mostrar que alguns detalhes intrínsecos à natureza desse serviço são fundamentais e devem ser levados em conta pelo algoritmo que escolherá o melhor servidor. Na Seção 4.1, citamos trabalhos que demonstraram que o consumo da largura de banda de um AP é determinado principalmente pela identidade dos clientes conectados a ele do que pelo número de clientes em si, como poderia sugerir o senso comum. No caso de hospedagem *web*, o artigo de [Cardellini et al. 2002] argumenta que um pequeno número de objetos *web* é alvo da maioria das requisições. Esse aspecto favorece técnicas que privilegiam a formação de *cache*, por exemplo, CAP (*Content Aware Policy*) e LARD (*Locality-aware Request Distribution*). Essa última utiliza uma tabela *hash* que, pode ser implementada em memória no *web switch*, é responsável por enviar as requisições por um mesmo objeto sempre para o mesmo servidor, se as condições de carregamento deste servidor permitirem. Assim,

⁷Alternativamente, algoritmos de balanceamento podem ser implementados pelo sistema operacional ou então no servidor DNS. Porém, essas soluções estão fora do controle do provedor do serviço que não tem nenhuma gestão sobre elas e não tem condições de garantir a qualidade do serviço de forma fim-a-fim sendo, portanto, não escolhidas atualmente.

vemos que esse algoritmo é ciente tanto do contexto do cliente quanto do servidor⁸.

Porém é importante notar que as técnicas que privilegiam a formação de *cache* citadas anteriormente têm um bom rendimento apenas quando o conteúdo das páginas é estático. No intervalo entre duas requisições sucessivas a mesma URL, o provedor do serviço pode alterar o conteúdo daquela página e tornar sua renderização muito mais custosa do que da primeira vez. Um exemplo é a construção de uma página que retorne os CDs oferecidos por uma loja virtual de música: eles dependem de parâmetros de consulta enviados pelo usuário. Em outro exemplo, temos o caso de uma página que retorna alarmes coletados por um sistema de gerência: a cada vez que for solicitada a consulta traz um número diferente de registros e, portanto, alterando o seu custo de processamento. Além de o conteúdo estático representar uma pequena porcentagem das páginas *web* atuais, os servidores se tornaram mais complexos principalmente devido a arquitetura multi-camadas. Segundo essa abordagem, o servidor *web* foi dividido em três camadas com responsabilidades bem definidas. A primeira cuida principalmente de receber a requisição do usuário. Se ela solicitar uma página estática, então a primeira camada é capaz de retorná-la ao cliente. Se a requisição requerer algum processamento ou alguma regra de negócio para ser respondida, então ela é encaminhada à segunda camada, que contém o servidor de aplicação. Nela é executado o processamento das regras do negócio da empresa necessárias para responder ao cliente. Frequentemente este servidor acessa para consultas ou atualizações a última camada, que contém o banco de dados. Essas três funções tão distintas podem ser executadas na mesma máquina ou em máquinas separadas.

Dessa forma, com o conteúdo dinâmico, com tantas camadas de *software* especializado e com a possibilidade de haver *cache* entre elas torna-se muito difícil para os métodos baseados em gerência de *cache* modelarem essa arquitetura e realizarem o balanceamento da carga de forma razoável e computacionalmente barata. Este é o principal motivo pelo qual soluções profissionais de armazenagem de aplicações *web*, como por exemplo o servidor de aplicações JBoss [JBoss], oferecem nativamente ao provedor do serviço apenas mecanismos simples que implementam o acesso aleatório ou no estilo *Round Robin* aos servidores de um *cluster*.

Como contribuição deste trabalho, propomos o emprego de uma função utilidade para ser executada no *web switch*, cega ao contexto da requisição HTTP e dos clientes (para

⁸Para ser ciente do contexto do servidor, o algoritmo pode lançar mão das técnicas expostas na Subseção 2.6.2, tais como *server pushes*, *server probing* ou requisições *dumb*. O artigo de [Cardellini et al. 2002] ratifica que uma técnica que combine todos estas métricas em um único número que permita a escolha do servidor é ainda um campo a ser explorado. Vemos aqui uma oportunidade para o uso de funções utilidade.

garantir a escalabilidade da solução) mas ciente ao contexto dos servidores que compõem o *cluster*. A função utilidade deve apresentar alto *throughput* devido ao fato do mecanismo de seleção implementado por ela (uma média ponderada) ser barato computacionalmente quando comparado às técnicas mais complexas já vistas nessa seção. Dessa forma, a função utilidade é calculada levando em conta as dimensões de QoS tais como a quantidade de memória/CPU livres em cada servidor e o atraso/largura de banda nos enlaces de comunicação entre o *web switch* e os servidores. Neste exemplo, além de definir as dimensões de QoS que impactam na qualidade do serviço oferecido, o desenvolvedor é responsável por estabelecer os pesos que vão parametrizar a função utilidade, guiando a seleção do servidor que oferece mais recursos para a sua aplicação. Adicionalmente, a função utilidade pode levar em conta um fator de qualidade calculado pelos próprios servidores (similarmente à aplicação da Seção 4.1) que explore o fato de que poucos objetos *web* recebem um grande número de solicitações. Por exemplo, em época de Copa do Mundo de futebol, a página de esportes de um jornal *on-line* recebe muito mais requisições do que normalmente ela receberia. Assim, o desenvolvedor poderia disponibilizar mais de uma réplica desse conteúdo em determinados servidores, e estes retornariam um fator de qualidade maior para o cálculo da função utilidade.

4.3 Parametrizando um componente

Propomos uma terceira aplicação onde acreditamos que o uso de uma função utilidade possa auxiliar a encontrar a melhor parametrização para um recurso .

Neste cenário, o usuário deseja assistir um vídeo de alta qualidade (que está armazenado em um servidor remoto) em um dispositivo que depende de sua localização e da ocupação do ambiente onde ele se encontra [Santos et al. 2006]. Por exemplo, se ele está sozinho na sala, então ele deseja que o vídeo seja exibido no monitor a sua frente, ao passo que se houver mais alguém com ele, o vídeo deve ser exibido em seu PDA. Assim, a disponibilidade dos recursos necessários varia ou não pode ser garantida pelo ambiente. Ou seja, as dimensões de QoS do dispositivo que atuará como *player* do vídeo (tais como memória/CPU livres, quantidade de cores, resolução da tela, carga da bateria) e do canal de comunicação entre ele e o servidor (tais como latência e largura de banda) podem variar sem que o cliente tenha o controle sobre isso. Adicionalmente, o usuário pode ter preferências diferentes dependendo do tipo de vídeo que está assistindo. Por exemplo, quando ele assiste a um vídeo de esportes, ele pode privilegiar uma alta taxa de quadros por segundo em detrimento de uma boa qualidade de áudio. Já, por exemplo, quando

o vídeo traz um noticiário, então o usuário pode escolher ter uma melhor qualidade de áudio monofônico em detrimento do número de cores, seja para poupar largura de banda do canal de comunicação ou a carga da bateria do seu dispositivo.

Para suportar as variações da disponibilidade dos recursos consumidos pela aplicação e as preferências do usuário, torna-se necessária uma reação adaptativa. Em outras palavras, a aplicação deve encontrar no ambiente um componente de vídeo que seja capaz de transcodificar o vídeo armazenado no servidor em uma qualidade que seja compatível com os recursos disponíveis no ambiente e às preferências do usuário. Neste exemplo, para evitar a tarefa de consultar um serviço de descoberta de recursos, podemos admitir que a tarefa da aplicação é definir a parametrização ideal para um componente transcodificador já instalado. A troca de um componente em tempo de execução contém aspectos que são ortogonais a esse estudo, tais como salvar o estado atual para posterior recuperação, interromper as mensagens em trânsito no sistema com destino a esse componente, desfazer as ligações entre esse componente e os outros da arquitetura e possivelmente ter alguma contingência para o período em que a aplicação estiver indisponível. Parametrizar um componente significa:

- a) descobrir o conjunto de parâmetros que mais satisfaçam ao usuário (o que não significa necessariamente impor a aplicação em seu nível diferenciado de maior qualidade) e, concomitantemente;
- b) verificar se a parametrização escolhida na fase anterior evita que o componente consuma mais recursos do que aqueles oferecidos pelo ambiente.

A primeira parte da tarefa pode ser realizada com o auxílio de uma função utilidade que retorna a parametrização que mais se aproxima dos requisitos do usuário. Para isso ela pode usar uma média ponderada de várias funções de satisfação, conforme já descrito anteriormente na Seção 4.1. A ponderação reflete exatamente as preferências do usuário: se ele decidir que naquele instante a resolução do vídeo é mais importante que a taxa de quadros por segundo, então essa dimensão de QoS contará com um peso maior no cálculo da função utilidade. Podem fazer parte dessa função dimensões de QoS que são facilmente identificáveis pelo usuário quando ele assiste a um vídeo, tais como: quanto ao vídeo - número de cores, resolução da tela, taxa de quadros por segundo e etc; e quanto ao áudio - número de bits, taxa de amostragem, estéreo ou monofônico.

A segunda parte da tarefa é mais complexa: ela exige que o sistema seja capaz de prever/estimar a quantidade de recursos consumidos quando o componente adquire uma

determinada parametrização. Em [Narayanan et al. 2000], os autores buscaram esse objetivo através de uma fase inicial de treinamento. Nesta fase, o componente transcodificador é configurado com diversas fidelidades (conforme os autores se referem às parametrizações ou níveis diferenciados de qualidade) e processa um universo representativo de vídeos. Após essa fase (*off-line*), o componente ao processar um novo vídeo seria capaz de prever a quantidade de recursos consumidos e atualizar, de forma iterativa e aditiva (*on-line*), o treinamento realizado anteriormente. Os autores concluem que essa previsão não pode ser feita sempre com precisão porque é difícil encontrar um treinamento que responda bem a todos os tipos de vídeos. Ainda segundo os autores, essa dificuldade advém de dois fatores principais: o primeiro é que algumas mídias exigem um consumo de recursos de forma não-linear para serem processadas. Por exemplo, exibir uma imagem JPEG consome recursos linearmente de acordo com a compressão do arquivo apenas quando ela está entre 5% e 80%. Fora dessa faixa não foi possível modelar com precisão a quantidade de energia gasta para exibir a imagem. O segundo fator é que alguns *softwares* exibidores apresentam um comportamento também não-linear. O artigo cita a experiência que os autores fizeram com o navegador Netscape para exibir imagens JPEG. Segundo o trabalho, a instanciação de *threads* feita pelo navegador diminuiu bastante a capacidade de previsão do método proposto por eles, o que não se repetiu quando eles usaram um exibidor mais simples de imagens.

Desta forma, é interessante notar que a qualidade da previsão dos recursos gastos por um componente é dependente da mídia e do *software* usado para exibí-la. No trabalho [Bhatti e Knight 1999], foram feitas diversas experiências com fluxos contínuos de áudio. Segundo a Tabela 4.2, transcrita daquele trabalho, é possível prever a quantidade de largura de banda e de processamento consumidos por um determinado esquema de compactação. Por exemplo, o esquema PCM (utilizado em telefonia digital) que utiliza 8 bits por amostra e 8 mil amostras por segundo, consome 64kbps de largura de banda (sem compressão) e um custo (normalizado) de CPU de 1 unidade. Outros esquemas utilizam menos largura de banda, porém consomem mais CPU.

É importante mencionar que a previsão de gasto dos recursos de uma dada parametrização não precisa ter uma grande precisão absoluta, mas relativa, uma vez que ela será usada para comparar parametrizações. Uma vez que essas previsões já podem estar disponíveis previamente como na Tabela 4.2, elas podem fazer parte diretamente da função utilidade e com isso o usuário pode expressar sua preferência por parametrizações que gastem menos bateria, por exemplo. Em outras palavras, os itens a) e b) descritos anteriormente são resolvidos em uma única fase.

codificação	fluxo de dados	custo relativo de CPU
PCM	64.0	1
ADM6	48.0	13
ADM4	32.0	11
ADM2	16.0	9
GSM	13.0	1200
LPC	4.8	110

Tabela 4.2: *Recursos consumidos com diversas parametrizações*

Dessa forma, propomos usar uma função utilidade que de posse das preferências do usuário, das previsões de consumo de recursos e da disponibilidade dos recursos oferecidos naquele instante pelo ambiente seja capaz de decidir qual a melhor parametrização possível para ser imposta naquele contexto.

4.4 Considerações gerais sobre a instabilidade

Em qualquer um dos exemplos de aplicação citados, para evitar a instabilidade (oscilação) do *framework* frente às flutuações da disponibilidade das dimensões de QoS, é importante lembrar que algumas medidas devem ser tomadas (ver Seção 2.6). Uma possibilidade é o uso de uma janela deslizante, em que não apenas a última medida é levada em conta, mas a média das n últimas⁹. É interessante notar que essa média pode ser ponderada. Outras técnicas foram sugeridas em [Bhatti e Knight 1999]: uma delas prevê que caso haja fartura de recursos no ambiente, então a aplicação só deve evoluir para um estado de mais alta qualidade se essa fartura perdurar por um certo período (expressando a idéia de histerese). Caso haja falta de recursos, então a aplicação migra imediatamente para um estado de mais baixa qualidade. Outra técnica propõe que se pelo menos uma das dimensões de QoS da aplicação ficar perto do limite da faixa de transição, então o estado de menor qualidade é imposto, como mostrado na Seção 2.6.

Um aspecto importante que deve ser mencionado quando vários clientes escolhem o mesmo tipo de recurso (que no primeiro cenário trata-se de um AP) é que, freqüentemente,

⁹Ao invés de usar uma janela deslizante para amortecer o resultado da função utilidade como um todo, o cliente pode usar uma janela deslizante para cada dimensão de QoS visto que elas podem ter dinâmicas diferentes. Por exemplo, enquanto a largura nominal do canal de um AP é constante (ex 54Mbps), a intensidade do sinal pode variar bastante.

os clientes podem ser modelados como não-cooperativos (teoria dos jogos), podendo apresentar preferências parecidas, o que faria com que cada um identificasse o mesmo AP como sendo o melhor para se conectar [Karaul et al. 1998]. Se isso acontecer, esses clientes se conectariam a este AP e fariam com que seus recursos fossem consumidos demasiadamente, reduzindo a qualidade percebida por eles mesmos. Como efeito, eles buscariam por um novo melhor AP para se conectar e todo o processo se repetiria. Para evitar essas oscilações, pode ser empregado o método da roleta (termo proveniente do estudo dos algoritmos genéticos), no qual quanto maior for o valor retornado pela função utilidade de um AP, maior probabilidade ele terá de ser escolhido pelo cliente. Assim, os piores APs ainda terão uma chance de serem utilizados. Alternativamente ao método da roleta, o trabalho [Zegura et al. 2000] propõe que seja montado um vetor de recursos equivalentes, ou seja, um conjunto dos melhores recursos. Assim, um deles seria escolhido aleatoriamente e retornado ao cliente. O método da roleta será empregado no Capítulo 5.

4.5 Conclusão do capítulo

Neste capítulo, apresentamos três oportunidades em que uma função utilidade pode ser aplicada com benefícios. Entendemos que ela constitui uma técnica que captura as preferências dos usuários (que ponderam dimensões de QoS frequentemente conflitantes) e a disponibilidade dos recursos que impactam na qualidade da aplicação. Outra virtude dessa técnica é a facilidade de implementação e a pouca quantidade de processamento exigida, propiciando ser utilizada em pequenos dispositivos com recursos computacionais e de energia restritos, tais como PDAs.

Cada um desses problemas é resolvido atualmente com outras técnicas que poderiam ser suportadas ou mesmo substituídas por uma função utilidade. Por exemplo, a escolha do melhor AP de uma rede sem-fio é feita comercialmente através apenas da intensidade do sinal medido entre os APs e o cliente, o que pode ocasionar um desbalanceamento de carga conforme já exposto. Dos três exemplos de aplicação mostrados, o último é sem dúvida o mais incipiente por haver falta de estudos modelando o consumo de recursos (ver Seção 6.2), dada uma parametrização. Ainda assim, acreditamos que a função utilidade poderia ser de grande valia para escolher a melhor parametrização de um componente.

Capítulo 5

Estudo de caso

No capítulo anterior, discutimos três oportunidades realísticas em que propomos aplicar funções utilidade. Conforme já exposto, a função utilidade mostra-se na literatura uma ferramenta importante para a escolha de um recurso ou parametrização uma vez que ela pode comportar, ao mesmo tempo, dimensões de QoS possivelmente conflitantes, através das preferências do usuário, valorizando algumas dimensões de QoS em detrimento de outras. Outra virtude dessa técnica é a facilidade de implementação e a pouca quantidade de processamento exigida, propiciando ser utilizada em pequenos dispositivos com recursos computacionais e de energia restritos, tais como PDAs. Neste capítulo, além de empregar uma função utilidade em um dos exemplos citados no capítulo anterior, iremos tratar como o desenvolvedor do serviço (que teoricamente é aquele com mais experiência e propriedade para tanto) pode definir as dimensões de QoS que farão parte da função utilidade.

Para permitir um estudo em profundidade, vamos nos concentrar na primeira aplicação citada no capítulo anterior (ver Seção 4.1). Acreditamos que essa é uma escolha acertada pois oferece uma grande quantidade de variáveis que podemos estudar/controlar, seja analisando *traces* de utilização de redes sem-fio de grande porte, seja através de simulações desenvolvidas para esse trabalho. Além disso, entendemos que a primeira aplicação tem atualmente, nas soluções disponíveis no mercado, a implementação mais ingênua e, portanto, oferece maior potencial de ser beneficiada pelo emprego de uma função utilidade.

Essa ingenuidade é corroborada no trabalho de [Judd e Steenkiste 2002], onde vemos que produtos comerciais frequentemente dotam os clientes de políticas simples para a seleção do melhor AP. Segundo os autores, essas políticas são implementadas no *firmware* das interfaces de rede e levam em conta apenas a intensidade do sinal do AP para selecioná-lo de forma automática. Outros produtos não o fazem de forma automática, mas o cliente

dispõe apenas da intensidade do sinal para fazer manualmente a seleção¹.

Como exemplo, os autores citam a experiência realizada na qual duas salas de aula eram cobertas apenas por um AP que na maioria do tempo experimentava um alto tráfego. Para remediar a situação, um novo AP foi instalado. Cientes de que a escolha dos clientes era baseada apenas na intensidade do sinal, os pesquisadores tomaram cuidado para que elas estivessem equilibradas. Mesmo assim, percebeu-se que o primeiro AP continuava sobrecarregado enquanto o segundo AP cursava uma quantidade mínima de tráfego. Os autores explicam que o fato se deu pela falta de suporte que o protocolo 802.11 oferece para a escolha do AP, informando ao cliente basicamente a intensidade de sinal dos APs disponíveis. Uma pesquisa mais detalhada mostrou que, apesar da cobertura dos dois APs ser praticamente semelhante, na maioria dos pontos onde os clientes se encontravam, o nível de sinal do primeiro era ligeiramente maior do que o segundo. A Figura 5.1 mostra que a utilização (tráfego) dos dois APs era bastante desbalanceada e que um deles continuava bastante sobrecarregado.

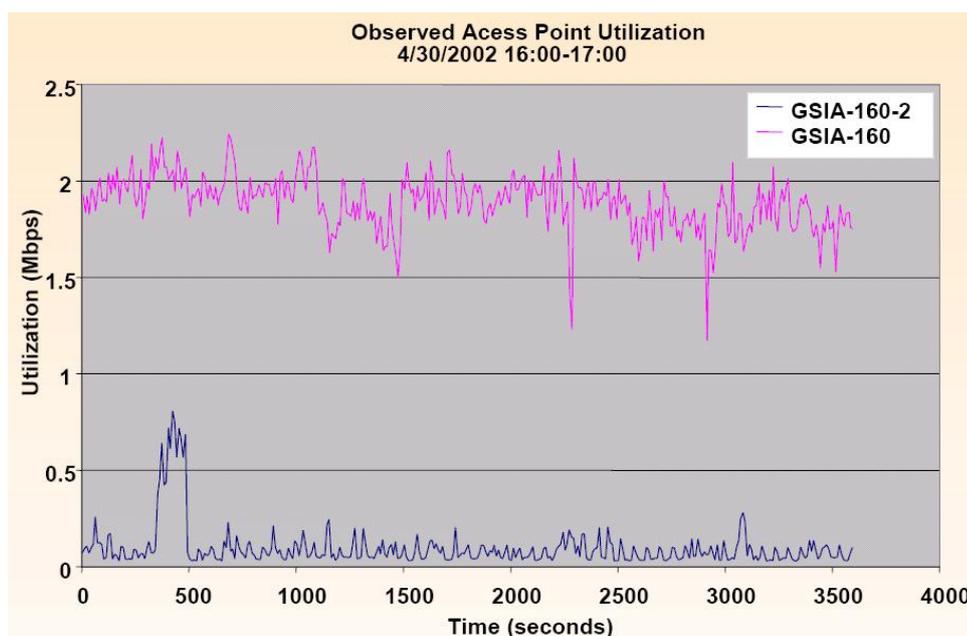


Figura 5.1: *Utilização desbalanceada de dois APs de uma mesma rede [Judd e Steenkiste 2002]*

De fato, é interessante notar que aplicações comerciais como o Microsoft Windows só oferecem a intensidade de sinal para que o usuário escolha a rede² no qual se conectar,

¹Outro fator negativo levantado pelos autores é que, sendo implementada no *firmware*, substituir a política oferecida de fábrica por uma customizada nos vários clientes de uma rede de tamanho considerável é bastante complexo.

²O protocolo 802.11b/g usa o identificador SSID para identificar as diferentes redes sem-fio e não os APs individualmente. É este identificador da rede (que no caso da Figura 5.2 vale `Joni-Net`, `pereiralan`, `gt_mesh_interna` e `gt_mesh`) que é repassado para a aplicação.

sugerindo que ele se conecte àquela que apresenta maior intensidade de sinal. A Figura 5.2 mostra que apenas a intensidade de sinal é mostrada para guiar essa seleção.

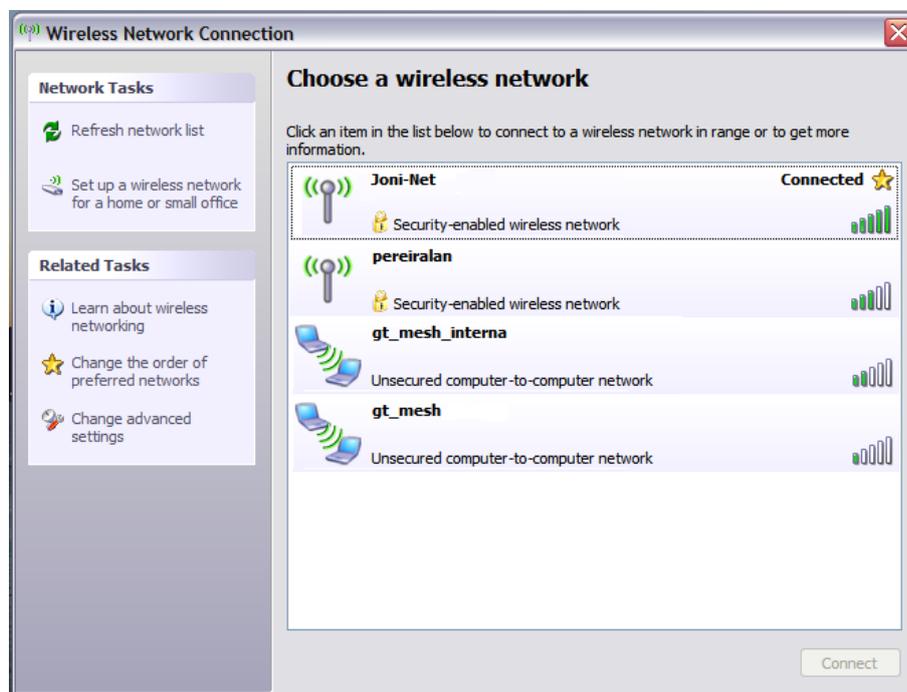


Figura 5.2: Quadro seleção de redes sem-fio do MS Windows

Assim, nas duas seções seguintes, vamos ilustrar a aplicação de uma função utilidade para a escolha do melhor AP. Primeiramente, um cliente hipotético parado deve escolher o melhor AP analisando *traces* reais (coletados através de medições SNMP) da utilização de uma grande rede sem-fio corporativa, operando o protocolo 802.11b. No segundo cenário, um simulador orientado a eventos discretos foi desenvolvido para verificar a influência das diversas parametrizações possíveis da função utilidade na distribuição de carga entre APs e, conseqüentemente, na qualidade do serviço oferecido. Nessa simulação, vamos variar o padrão de movimentação dos clientes e de geração de tráfego para verificar como esses clientes escolhem o melhor AP, ou seja, vamos testar a robustez da solução implementada através de uma função utilidade.

5.1 Análise do tráfego de uma rede sem-fio real

Esta primeira implementação foi motivada por uma pesquisa feita em [Balazinska e Castro 2003] na qual informações sobre a utilização de APs foram coletadas durante aproximadamente trinta dias (de 20 de julho de 2002 a 17 de agosto de 2002). Esses APs encontravam-se distribuídos de forma heterogênea em três edifícios de uma

grande empresa americana, os quais têm tamanhos e utilizações diferentes (laboratórios, auditórios, bibliotecas, cafeterias e etc.). Os APs já tinham sido distribuídos (independentemente da realização destas medições) levando-se em conta principalmente aspectos da geometria dos edifícios, por exemplo, um AP por corredor, e também em salas onde espera-se uma grande utilização, tais como laboratórios. Trata-se de um universo de 177 APs de mesmo modelo e fabricante (Cisco Aironet 350s) utilizados por 1366 clientes portando *laptops*, sem que esses clientes soubessem da realização do teste. Esses APs eram solicitados a cada cinco minutos, através de chamadas SNMP, a enviar informações sobre a quantidade de pacotes enviados/recebidos, a quantidade de bytes enviados/recebidos, a quantidade de retransmissões, o número de usuários conectados e a última intensidade de sinal de cada cliente conectado a ele. Essas dimensões de QoS foram encontradas em um dos três formatos dos arquivos (*traces*) disponibilizados pelos autores³.

Com essa massa de dados realísticos nas mãos, nosso objetivo foi processá-la (conforme será explicado adiante) para verificar a robustez da qualidade da seleção implementada através de uma função utilidade ao longo do tempo, ou seja, variando o número de clientes envolvidos e o tráfego gerado por eles. Para isso, a função utilidade foi comparada a outras técnicas de seleção, como por exemplo, uma escolha baseada no número de usuários conectados ao AP e uma escolha aleatória. Aqui, abordamos apenas o aspecto que se refere ao carregamento dos APs. Em [Hernandez-Campos e Papadopouli 2005], os autores têm preocupações adicionais tais como a mobilidade do usuário (locais onde eles acessam a rede mais frequentemente, quantidade de *roamings* entre APs e etc). Seu objetivo era descobrir similaridades entre redes que operam em diferentes ambientes, tais como *campi* de universidades, empresas e etc, para propiciar a construção de novas ferramentas de modelagem e simulação de redes sem-fio.

Nos *traces* que analisamos, a amostragem dos APs teve alguns problemas tais como: algumas falhas de energia criaram lacunas nas medições de aproximadamente 1 hora, alguns APs (três) não responderam às solicitações SNMP e principalmente APs que, devido ao grande tráfego cursado, estavam demasiadamente ocupados e não conseguiam responder à chamada SNMP a cada cinco minutos. Além disso, os *traces* traziam contadores cumulativos das dimensões de QoS citadas anteriormente, o que é um praxe entre diversos outros fabricantes. Porém, devido a algum problema não documentado, entre duas amostras subseqüentes a quantidade de alguns daqueles contadores diminuía ao invés de aumentar, o que fazia o Δ entre duas amostragens sucessivas ficar negativo. Esse fato pode ser devido ao *firmware* usar contadores de 32 bits o que mostrou-se insufi-

³<http://crawdad.cs.dartmouth.edu/meta.php?name=ibm/watson>

ente em alguns casos, conforme será evidenciado à frente. Outra fonte de erro pode ser associada a *bugs* internos ao *firmware* dos APs, como também foi alertado no trabalho de [Hernandez-Campos e Papadopouli 2005]. Obviamente, contornamos esses problemas durante nossa análise.

Voltando ao artigo de [Balazinska e Castro 2003], é comum nesse tipo de trabalho que o *trace* no qual a pesquisa se baseou seja disponibilizado pelos autores só após os APs e clientes serem “tornados anônimos”. Esse processamento retira a informação de localização (x , y , z) dos APs o que impede que estudos posteriores verifiquem, por exemplo, correlações entre o tráfego cursado pelo AP e sua posição física na planta-baixa do prédio. A falta dessas informações impede também que a intensidade do sinal dos APs seja gerada sinteticamente para utilização em simulações.

Apesar da ausência dessas informações de localização, os quatro primeiros dias de *traces* foram processados e carregados em um banco de dados relacional desenvolvido para este trabalho, de forma que cada registro representasse a utilização de um determinado AP em um determinado instante: cada registro do banco tinha (não de forma cumulativa como estava disponível nos *traces*, mas apenas durante os últimos cinco minutos) o número de bytes enviados/recebidos, de pacotes enviados/recebidos, número de bytes gastos em retransmissões (devido a erros) e o número de clientes conectados aquele AP, responsáveis por gerar tal tráfego. Esse carregamento em um banco de dados facilitou nossa análise pois com a linguagem SQL era imediato saber o estado de um determinado AP em um determinado instante.

5.1.1 Resultados

5.1.1.1 Dinâmica das medições e influência da janela deslizante

A primeira constatação está relacionada à dinâmica das medições. Corroborando a flutuação encontrada na Figura 5.1, mostramos na Figura 5.3 como variou o tráfego (bytes enviados mais bytes recebidos) do AP 31 durante o primeiro dia da experiência. É visível a ocorrência de grandes picos e vales em azul-pontilhado (tráfego não-ponderado - *nonWeighted*) mesmo em um fim de semana. Com o objetivo de filtrar essas variações e estabilizar o sistema, uma janela deslizante de tamanho três foi implementada. Nessa janela, arbitramos que a última e penúltima amostra têm peso dois enquanto que a antepenúltima amostra tem peso 1. Os resultados obtidos estão colocados em lilás-contínuo

(tráfego ponderado - *weighted*) na Figura 5.3⁴.

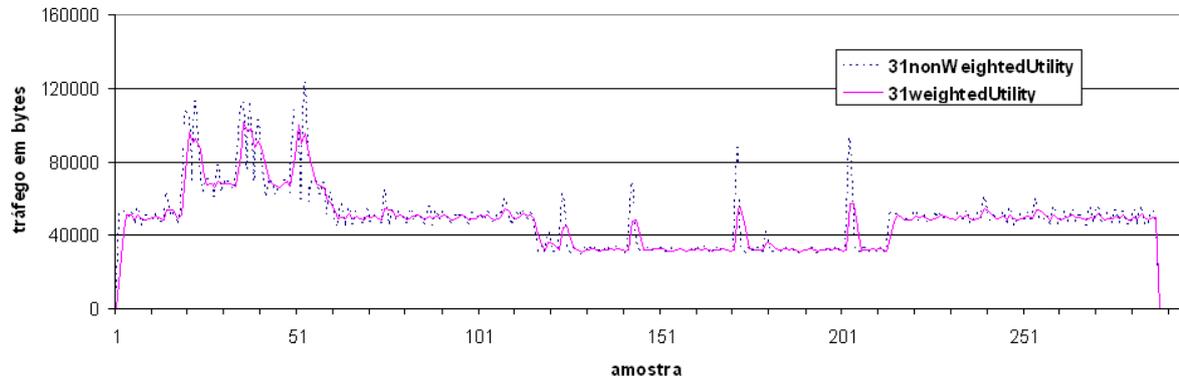


Figura 5.3: *Influência da janela deslizante (A)*

Na Figura 5.4 é mostrado em um gráfico de espalhamento como essa técnica simples é capaz de filtrar resultados de medições que poderiam ser responsáveis por desestabilizar o mecanismo de escolha do melhor AP. No eixo horizontal está mostrado o tráfego não-ponderado do mesmo AP da Figura 5.3 e no eixo vertical está o resultado da razão entre o tráfego ponderado (pela janela deslizante) pelo tráfego não-ponderado. Vê-se que para pequenos valores de tráfego não-ponderado essa razão é maior que a unidade e que para valores grandes a razão é menor que a unidade mostrando o efeito amortecedor que a janela deslizante tem sobre dados monitorados. Dessa forma, verificamos que a janela deslizante adiciona uma certa histerese aos dados monitorados, filtrando os extremos (inferior e superior) como era de se esperar.

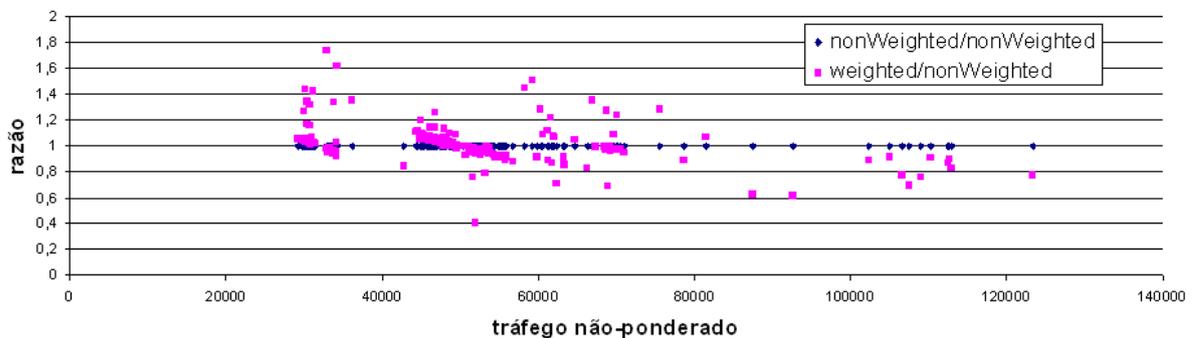


Figura 5.4: *Influência da janela deslizante (B)*

⁴Alguns termos deste capítulo estão em inglês em virtude de um artigo que está sendo escrito concomitantemente

5.1.1.2 Comparação entre políticas de escolha do melhor AP

A seguir, realizamos alguns testes que procuraram refletir o comportamento de um cliente hipotético parado selecionando o melhor AP dentre um grupo de quatro APs escolhidos ao acaso. O cliente realiza 288 escolhas ao longo de um período de 24 horas (uma escolha a cada cinco minutos) baseado em diferentes políticas. A primeira delas consiste de uma função utilidade que usa o tráfego cursado, ou seja, a soma dos bytes enviados e recebidos entre dois instantes de monitoração, para selecionar o AP (de menor tráfego). Para isso, um pequeno *framework* foi implementado em que um cliente consultava cada um dos quatro APs do grupo a cada intervalo de monitoração. Para cada um desses APs, o *framework* calcula sua função utilidade baseando-se apenas nas dimensões de QoS “bytes enviados” e “bytes recebidos”, conforme dito anteriormente. Para representar as preferências do usuário, todos os APs foram instanciados com arquivos de configuração idênticos, contendo dois vetores: o primeiro permite ponderar a função utilidade e o segundo permite ponderar a janela deslizante (utilizada a seguir neste teste)⁵. Assim, o cliente consulta cada um dos APs do grupo enviando basicamente o *timestamp* e cada implementação do AP se incumbia de calcular a função utilidade (acessando o banco de dados citado anteriormente) e retornava seu valor ao cliente, que então tinha o trabalho de compará-los para selecionar o melhor AP.

Resumindo, nossa implementação dessa primeira política de escolha contava com as dimensões de QoS “bytes enviados” e “bytes recebidos” com pesos unitários (a soma destas dimensões representa o tráfego entre o cliente e o AP). Logo, devido às duas dimensões serem expressas em bytes, não foram necessárias constantes de proporcionalidade, tais como descritas em [Huebscher e McCann 2004a]. Ainda assim, as funções de satisfação descritas na Seção 4.1 seriam úteis para capturar uma satisfação do usuário que fosse não-linear com a variação da dimensão de QoS, conforme supusemos neste caso.

Além dessa política, implementamos outras duas: a segunda elege como melhor o AP aquele com menor número de usuários conectados e a terceira política simplesmente escolhe o AP de forma aleatória. É importante mencionar que na avaliação das três políticas, ao escolher o AP, o cliente se conectava a ele mas não injetava tráfego para não alterar o *trace* analisado. Caso contrário, o *trace* não representaria mais a realidade de operação da rede sem-fio monitorada.

⁵Embora todos os APs do grupo sejam instanciados com arquivos idênticos, a implementação do *framework* optou por fazer assim para que em trabalhos futuros, testes com APs calculando sua utilidade com diferentes arquivos de configuração (independentemente) pudessem ser realizados sem nenhuma alteração na implementação.

Foram criados três grupos de quatro APs cada, sendo que nenhum AP pertencia a dois grupos (grupos disjuntos). O grupo de APs de um cliente representa os APs que ele consegue escutar através de sua interface *wireless*. Cada um desses grupos foi estudado durante os quatro primeiros dias do *trace*, que escolhemos propositalmente, por serem sábado, domingo, segunda e terça. Dessa forma, a análise dos dois primeiros dias do *trace* se mostrou bastante discrepante em termos de quantidade de bytes trafegados quando comparados aos terceiro e quarto dias, o que possibilitou criar dois cenários distintos. Na segunda metade do estudo (dias de trabalho) o tráfego cursado é bem superior ao tráfego cursado no fim de semana. Os resultados estão exibidos nas tabelas seguintes.

APs 2, 22, 58, 68	20/7/2002	21/7/2002	22/7/2002	23/7/2002
nonWeightedRoamings	153	142	154	162
weightedRoamings	62	63	84	99
redução	59,48%	55,63%	45,45%	38,89%

APs 19, 31, 42, 71	20/7/2002	21/7/2002	22/7/2002	23/7/2002
nonWeightedRoamings	75	64	76	111
weightedRoamings	28	26	70	97
redução	62,67%	59,38%	7,89%	12,61%

Tabela 5.1: *Roamings* realizados pelo cliente (máximo:288)

A Tabela 5.1 mostra a quantidade de *roamings* (também conhecido por *handoffs* ou reconexões, ou seja, a quantidade de vezes que o cliente se desconectou de um AP e se conectou a outro) realizados em dois dos três grupos de APs estudados (obtivemos resultados semelhantes para o terceiro grupo), utilizando a função utilidade que somava os bytes enviados e recebidos. Esses *roamings* (cujo máximo era 288) aconteceram porque o cliente percebia que no intervalo de medição corrente o AP no qual ele estava conectado apresentava uma utilidade menor do que outros APs de seu grupo. As linhas iniciadas por `nonWeightedRoamings` trazem o número de *roamings* que o cliente experimentou quando não usava uma janela deslizante para amortecer os valores das funções utilidades retornados pelos APs. Já nas linhas iniciadas por `weightedRoamings`, o cliente se valeu de uma janela de tamanho três, onde a última e penúltima amostras têm peso 2 e a antepenúltima amostra tem peso 1. Essa parametrização foi obtida através do arquivo de iniciação dos APs citados anteriormente⁶. É interessante notar como uma técnica simples foi capaz de reduzir sensivelmente a quantidade de *roamings*. Outro fato que deve ser percebido é que essa técnica se mostrou mais eficiente no fim de semana (dias 20 e 21) do que em dias de trabalho. A grande variação da quantidade de tráfego nos dias 22 e 23 pode ter sido superior àquela que esta janela deslizante poderia amortecer.

⁶Na Seção 5.2.3 exercitamos de várias formas diferentes a parametrização da janela deslizante que utilizamos para amortecer as monitorações.

Feita esta primeira análise, na Tabela 5.2 mostramos os resultados mais importantes desta seção, ou seja, a comparação entre três políticas utilizadas para selecionar o melhor AP dentre um grupo de 4 APs (foi mostrado apenas o grupo dos APs 19, 31, 42 e 71 pois com os outros grupos obtivemos resultados semelhantes.). As políticas são:

- **nonWeig** e **weig**: o cliente escolhe o melhor AP utilizando a função utilidade definida anteriormente sem e com (respectivamente) o uso da janela deslizante, também definida anteriormente.
- **minUsers**: o cliente escolhe o melhor AP buscando aquele que tem a menor quantidade de usuários conectados. Nós contabilizamos que o cliente fez uma escolha certa através desta técnica apenas quando ele escolhe o mesmo AP que escolheu através da técnica **nonWeig**. Este critério foi arbitrado dessa forma para que as técnicas pudessem ser comparadas.
- **random**: o cliente escolhe o melhor AP de forma aleatória. Novamente, a escolha é contabilizada como certa apenas quando ela é coincidente com aquela que o cliente fez através da técnica **nonWeig**. Mais uma vez, este critério foi arbitrado dessa forma para que as técnicas pudessem ser comparadas.

política	AP	dia 20/7/2002	AP	dia 21/7/2002	AP	dia 22/7/2002	AP	dia 23/7/2002
nonWeig	19	41,32%	19	0,00%	19	4,86%	19	15,63%
weig		40,97%		0,00%		4,86%		15,97%
minUsers		41,32%		0,00%		4,86%		12,85%
random		12,85%		0,00%		1,39%		3,13%
nonWeig	31	8,68%	31	1,04%	31	0,35%	31	4,17%
weig		13,19%		0,69%		0,00%		4,17%
minUsers		8,68%		0,69%		0,00%		4,17%
random		2,08%		0,00%		0,00%		0,69%
nonWeig	42	14,58%	42	71,18%	42	11,46%	42	37,15%
weig		9,03%		69,10%		11,11%		40,63%
minUsers		3,47%		71,18%		8,68%		32,99%
random		4,17%		19,79%		2,43%		6,25%
nonWeig	71	34,72%	71	27,08%	71	80,90%	71	39,58%
weig		36,11%		29,51%		81,60%		35,76%
minUsers		27,43%		0,69%		33,33%		22,92%
random		5,56%		9,03%		21,53%		7,64%

Tabela 5.2: Escolhas através de diferentes políticas

Na Figura 5.2, as células foram hachuradas de forma que o AP escolhido mais frequentemente (maior porcentagem) por cada técnica durante um determinado dia fosse destacado. Por exemplo, no dia 20 a técnica **nonWeig** escolheu o AP 19 41,32% das vezes, o

AP 31 8,68% das vezes, o AP 42 14,58% das vezes e o AP 71 34,72% das vezes - por isso expressamos que esta técnica escolheu o AP 19 hachurando a célula correspondente. Observa-se que a escolha utilizando-se a função utilidade não-ponderada (**nonWeig**) e ponderada (**weig**) geralmente leva ao mesmo AP, exceto por uma pequena diferença encontrada no último dia: a política **nonWeig** escolheu o AP 71 (39,58% das vezes) enquanto que a política **weig** escolheu o AP 42 (40,63% das vezes). Dessa forma sugerimos usar a política ponderada pois ela levou o cliente, na maioria dos dias, ao mesmo AP escolhido pela política não-ponderada, com a vantagem de provocar um menor número de *roamings*, conforme foi mostrado na Figura 5.1.

Também é possível notar que no fim de semana (dias 20 e 21) a técnica que escolhe o melhor AP pelo menor número de usuários conectados (**minUsers**) obteve um índice de acertos praticamente igual ao obtido pela técnica que faz uso da função utilidade. Porém, esse fato não se manteve nos dias de trabalho (dias 22 e 23) quando mais usuários usam a rede e geram tráfego. Por exemplo, no dia 22, enquanto as técnicas que utilizam a função utilidade escolheram o AP 71 80,90% das vezes, a técnica **minUsers** só acertou essa escolha em cerca de 33,33% das vezes, revelando-se ser uma técnica pouco robusta às variações do ambiente.

Um último fato interessante é observado quando a escolha é realizada de forma aleatória (através da técnica **random**). É preciso lembrar que arbitramos que uma escolha só é contabilizada como correta quando o mesmo AP é escolhido pela técnica em questão e **também** pela técnica que usa a função utilidade não-ponderada (**nonWeig**), que é a técnica mais simples para escolher o AP de menor tráfego. Isto posto, a probabilidade de acerto da técnica **random** pode ser escrita como:

$$P(\text{random}) = P(\text{'escolha aleatoria'} \cap \text{nonWeig})$$

Uma vez que os dois eventos podem ser tomados como independentes, então a equação fica:

$$P(\text{random}) = P(\text{'escolha aleatoria'}) \times P(\text{nonWeig})$$

Por último, como a probabilidade de um AP ser escolhido aleatoriamente, dentro de um grupo de 4 APs, é de 1/4, a equação pode ser reescrita como:

$$P(\text{random}) = \frac{P(\text{nonWeig})}{4}$$

Assim, é interessante notar que a probabilidade da escolha aleatória efetivamente escolher o AP correto (tomado no nosso teste como aquele escolhido pela função utilidade não-ponderada) é inversamente proporcional a n onde n é o número de APs disponíveis no grupo para a escolha. Assim, é mostrada a baixa qualidade (robustez) da técnica baseada na escolha aleatória.

5.1.1.3 Correlação entre dimensões de QoS

Na tentativa de caracterizar dentre as dimensões de QoS disponíveis nos *traces* (quantidade de pacotes enviados/recebidos, a quantidade de bytes enviados/recebidos, a quantidade de retransmissões, o número de usuários conectados e a última intensidade do sinal de cada cliente⁷ conectado a ele) aquelas impactantes na qualidade da aplicação, foi analisada também a correlação estatística entre algumas dessas dimensões. A correlação estatística é uma métrica normalizada (varia entre -1 e 1), que quando tende a -1 significa que as duas grandezas analisadas são altamente correlacionadas mas inversamente proporcionais, ou seja, quando uma aumenta a outra tende a diminuir. Quando a correlação tende a 1 significa que as grandezas também são altamente correlacionadas, porém diretamente proporcionais. Se verificássemos que duas grandezas são altamente correlacionadas, então nossa intenção era montar uma função utilidade em que apenas uma delas participaria (uma vez que uma é dependente da outra).

Inicialmente analisamos a correlação estatística entre a quantidade de bytes retransmitidos e a intensidade do sinal medida em dBm e obtivemos um valor bem próximo a zero com duas casas decimais, indicando que essas duas grandezas não são correlacionadas.

Repetimos o processo agora correlacionando a intensidade do sinal com uma grandeza denominada nos *traces* como “qualidade do sinal”. Não foi possível achar na literatura ou na documentação do fabricante dos APs monitorados como essa grandeza é calculada. Em virtude da ausência de documentação e da subjetividade com que cada fabricante pode compor essa grandeza, não é recomendado utilizá-la com o intuito de comparar alternativas, principalmente quando a rede é composta por APs de diferentes fabricantes⁸.

⁷A intensidade do sinal do cliente percebida pelo AP não foi tomada como idêntica a intensidade do sinal do AP percebido pelo cliente, ou seja, não supomos que os enlaces são simétricos.

⁸Na verdade, a intensidade medida do sinal também pode variar de fabricante para fabricante uma vez que a sensibilidade das antenas e dos circuitos não é padronizada e se constitui uma diferença competitiva entre eles.

Ainda assim, prosseguimos com a análise mas acreditamos que o teste foi inconclusivo uma vez que as correlações variaram entre $-0,6$ e $+0,3$. Assim, não incentivamos colocar um grande peso para ponderar a participação das dimensões de QoS intensidade e “qualidade do sinal” na composição da função utilidade.

5.1.1.4 Descorrelação entre o número de usuários e o tráfego

Nosso próximo teste teve o objetivo de contrariar o senso comum e ratificar o que foi relatado em [Balazinska e Castro 2003] e mostrado na Figura 5.2. Ao escolher um AP seria natural buscar aquele com menos usuários conectados visando maior largura de banda. Porém, o número de usuários conectados a um AP não está fortemente correlacionado com o tráfego gerado por eles. Na Figura 5.5 está colocado o tráfego cursado em ambos os sentidos (bytes enviados mais bytes recebidos) em um AP no primeiro dia da experiência. Este AP tem até aproximadamente a metade da experiência apenas dois usuários conectados (Figura 5.6). Mesmo com poucos usuários é possível notar um grande pico de tráfego (capturado pela escala vertical logarítmica do gráfico da Figura 5.5) que durou certa de 20 amostragens, ou seja, 100 minutos. Esse resultado concorda com os autores quando eles afirmam que o tráfego cursado em um AP não é determinado pelo número de usuários conectados mas pela identidade deles. Uma outra forma de se analisar esse fato é também abordada por aquele artigo quando é mostrado que alguns usuários provocam um aumento de tráfego independente de qual AP eles estejam conectados enquanto outros são notadamente passivos, gerando pouco tráfego. Dessa forma, não incentivamos colocar um grande peso para ponderar a participação da dimensão de QoS “número de usuários conectados” na composição da função utilidade.

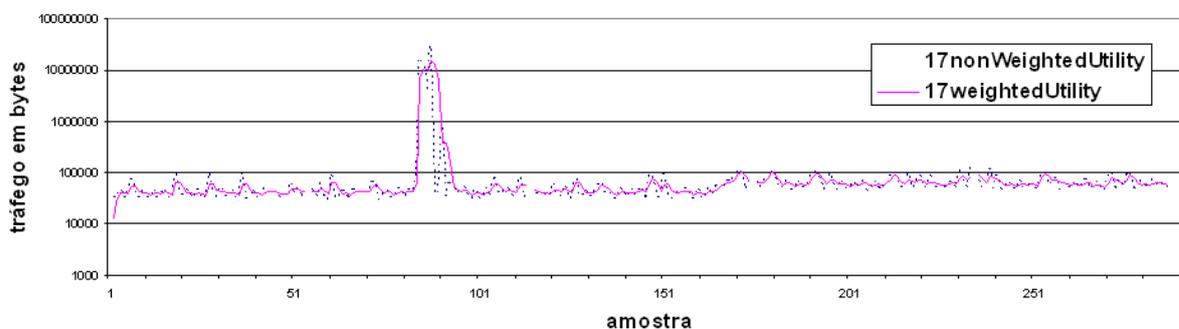


Figura 5.5: *Variação do tráfego cursado*

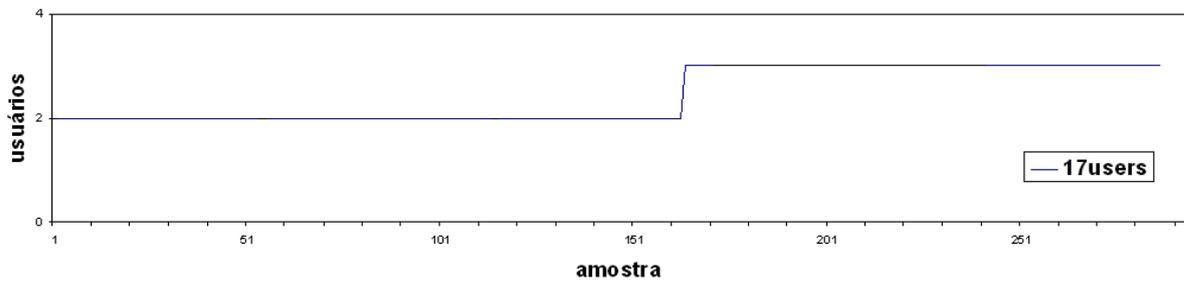


Figura 5.6: *Variação do número de usuários conectados*

5.2 Experimentos baseados em simulação

Nesta seção, iremos discutir os resultados obtidos através de um simulador desenvolvido especialmente para este trabalho. Ao utilizar simulação, o objetivo era aumentar o número de variáveis relevantes sob controle e, assim, compreender melhor o mecanismo de escolha de um AP. Através de simulação, pudemos:

- escolher o número de clientes participantes e o número de APs disponíveis para eles
- variar os padrões de transmissão, movimentação e de escolha dos APs utilizados pelos clientes
- utilizar a intensidade do sinal sintetizado, captado pelos clientes com base na distância entre eles e os APs

Tomamos a decisão de implementar um simulador próprio em uma linguagem orientada a objetos (Java) pois simuladores já estabelecidos na comunidade científica, tais como o [NS-2] ou [SWANS], têm interesse em detalhes de baixo nível do funcionamento dos protocolos de roteamento em redes sem-fio, que são preocupações ausentes neste estudo. Além disso, construindo o nosso próprio simulador, era possível isolar e dar foco aos fatores que realmente nos preocupam: uma arquitetura flexível o suficiente para exercitarmos eficazmente diversas políticas para a escolha do melhor AP, segundo as preferências dos clientes e a disponibilidade dos recursos do ambiente.

Dessa forma, o cenário proposto nessa seção envolve vários clientes móveis que desejam se conectar a um dos diversos APs disponíveis de uma rede sem-fio para obter acesso à Internet. Sobre esses clientes é possível dizer que são bastante heterogêneos tanto no que tange às aplicações que executam quanto na maneira como definem (através de suas preferências) um acesso de boa qualidade. Quanto aos APs, eles são fixos, têm a função de prover acesso sem-fio aos clientes e devem ser ligados por uma rede infra-estruturada.

A seguir, discutimos duas arquiteturas possíveis para a execução desse experimento. Ao final, justificaremos qual arquitetura empregamos.

5.2.1 Arquitetura centralizada

Aqui propomos uma arquitetura em que cada cliente se conecta através de sua interface *wireless* a um dos APs disponíveis. Os APs se conectam entre si e a um *gateway* através de uma rede infra-estruturada. Esse *gateway* efetivamente dá acesso à Internet e tem a função de autenticar os clientes, com o auxílio de um banco de dados, antes que eles consumam recursos da rede. Neste cenário, propomos instalar em algum nó da rede (possivelmente no próprio *gateway*) um componente **Seletor** que, suportado pelo banco de dados hospedado no *gateway*, realizará a seleção do melhor AP. Para permitir esse suporte, o banco de dados no *gateway* será enriquecido com estatísticas de utilização da rede, entre elas quais clientes se conectaram a quais APs e quantos bytes foram escoados em ambos os sentidos dessas conexões. Com esse banco de dados, é possível realizar consultas que retornem, por exemplo, o número de usuários conectados a um determinado AP naquele instante (conforme será mostrado adiante), quais clientes utilizam mais a rede e etc. Essa solução é típica e foi adotada, por exemplo, pelo *software* de código aberto, **Wifidog** utilizado em [de Albuquerque et al. 2006] para a gerência da utilização de uma rede *mesh*.

Do ponto de vista do cliente, o primeiro acesso é feito sempre através do AP que oferecer a ele a maior intensidade de sinal pois essa é, inicialmente, a única dimensão de QoS disponível ao cliente - apesar de sabermos que esse não é um bom critério porque a intensidade de sinal não assegura a qualidade da conexão [Judd e Steenkiste 2002]. Dessa forma, o cliente envia a este AP um vetor contendo todos os atributos (pesos e funções de satisfação) necessários para parametrizar a função utilidade que escolherá o melhor AP. Para cada dimensão de QoS deve ser fornecido um peso e uma função de satisfação. Para isso, deve ser oferecido ao usuário uma interface (fora do escopo deste trabalho) capaz de elicitar suas preferências, conforme encontramos em [Poladian et al. 2006]. Além desse primeiro vetor, um segundo é enviado contendo os IDs dos APs sobre os quais a função utilidade deverá ser calculada. Esse conjunto de APs é montado com base naqueles que o cliente consegue ouvir, ou seja, naqueles APs cujo sinal é captado pelo cliente.

É importante notar que o primeiro vetor pode conter pesos e funções de satisfação referentes tanto a dimensões de QoS como a meta-dimensões de QoS, tais como: resolução, frequência de amostragem; conforme sugeridas em [Huebscher e McCann 2005] e citadas

na Seção 4.1. Aqui partimos do pressuposto que existe entre os clientes e os APs um vocabulário previamente acordado contendo quais (meta)dimensões de QoS são monitoradas pelo sistema [Cardoso et al. 2006]. Esse tipo de acordo é típico em sistemas baseados na arquitetura cliente-servidor.

Prosseguindo, esses dois vetores são encaminhados pelo AP até o **Seletor** que tem a tarefa de operar a função utilidade, parametrizada com as preferências do cliente, sobre os APs solicitados. Para isso, o **Seletor** consulta o banco de dados. O resultado desse cálculo é o ordenamento do vetor de APs segundo a função utilidade. É interessante notar que algumas dimensões de QoS existem apenas entre o cliente e o AP, ou seja, não estão disponíveis no banco de dados, como por exemplo a intensidade do sinal no instante da escolha. Se o cliente desejar que a intensidade de sinal seja uma das dimensões de QoS a serem consideradas na escolha do melhor AP, então essa informação deve ser coletada por ele e enviada também ao **Seletor**.

Implementações ingênuas retornariam ao cliente sempre o primeiro elemento do vetor ordenado de APs. Porém se vários clientes de uma mesma localidade (ou seja, uma sala onde se enxergam basicamente os mesmos APs) enviarem um vetor de preferências semelhante para o **Seletor**, todos esses clientes se conectariam ao mesmo AP concentrando o tráfego. Em suas próximas escolhas, eles se desconectariam deste AP e se conectariam a outro, causando instabilidade e oscilações na rede. Para evitar esse fato, o **Seletor** pode escolher aleatoriamente um entre os primeiros APs do vetor ordenado. Essa escolha pode seguir uma distribuição uniforme ou pode privilegiar os melhores APs (método da roleta), conforme mostrado em [Zegura et al. 2000].

Feita a escolha, um único AP é retornado ao cliente que então deve se desconectar do AP escolhido inicialmente apenas pela intensidade do sinal e se conectar a este outro, escolhido de forma sensível ao contexto pela função utilidade.

Caso as preferências do cliente mudem porque ele passou a executar uma nova aplicação, por exemplo, ou porque ele se locomoveu (fazendo com que os APs ouvidos sejam outros), então o cliente deve reenviar os vetores de preferências e de APs para o **Seletor** e aguardar a escolha do novo melhor AP. Dessa forma, o **Seletor** sempre conhecerá as preferências e os APs nos quais os clientes têm interesse. Essa técnica é chamada de pró-ativa em [Capra et al. 2005].

Opcionalmente, a arquitetura centralizada pode ser enriquecida com uma técnica chamada de reativa em [Capra et al. 2005]. Ela é útil quando as características do ambiente, mais especificamente as dimensões de QoS daquele conjunto de APs que interessa ao cli-

ente, sofrem alterações. Neste caso o **Seletor** escolhe o novo melhor AP (dentro o vetor de APs enviado previamente) e o retorna ao cliente automaticamente. O vetor contendo os APs nos quais o cliente tem interesse pode ser implementado como um tópico de uma arquitetura *publish/subscribe*: ou seja, na técnica reativa, o **Seletor** observa o constantemente o estado desses APs e é capaz de permanentemente escolher o melhor AP dentre aqueles registrados no mesmo vetor (tópico). Por exemplo, se o cliente A tiver interesse nos APs 1, 2 e 3 então o **Seletor** periodicamente informará ao cliente A o melhor dentre os APs 1, 2 e 3.

Em ambas as técnicas (pró-ativa e reativa), caso o cliente se recuse a se conectar ao AP sugerido pelo **Seletor**, o **Seletor** pode alterar o *status* do cliente no banco de dados para inválido, impossibilitando-o de continuar a operar ou pode aplicar uma outra punição. Essa pode ser uma medida necessária pois como lembrou [Molina-Jimenez et al. 2004], o cliente também deve ser monitorado uma vez que ele também pode degradar a qualidade do serviço oferecido pela rede.

Essa arquitetura centralizada permite que a distribuição de carga esteja completamente nas mãos do administrador do sistema. Por exemplo, se ele desejar isolar um AP para manutenção ou se ele perceber que alguém importante⁹ se conectou em um determinado AP, basta que o administrador não retorne aquele AP para os clientes que estão solicitando conexão, conforme sugerimos na Seção 4.1. Para os que já estão conectados naquele AP, o sistema pode enviar uma mensagem ordenando a migração. Da mesma forma, um banco de dados centralizado permitiria o cadastramento de algumas regras que, por exemplo, negariam o acesso de um usuário que anteriormente consumiu muita banda da rede. A Figura 5.7 resume as operações mais importantes descritas no processo de seleção do melhor AP.

5.2.2 Arquitetura distribuída

Outra arquitetura, desta vez com um caráter distribuído, não conta com um banco de dados e um componente **Seletor** únicos. Nesse modelo, cada AP seria responsável por manter um fator de qualidade baseado apenas nas meta-dimensões de QoS que ele coletar localmente, tais como definidas em [Huebscher e McCann 2004a]: resolução, frequência de amostragem, etc. Conforme mencionado na Seção 4.1, esse fator de qualidade é uma média ponderada das meta-dimensões de QoS com pesos definidos pelo desenvolvedor e

⁹Aqui, claramente sugerimos que a implementação de uma política de prioridades para os clientes pode ser feita na arquitetura centralizada.

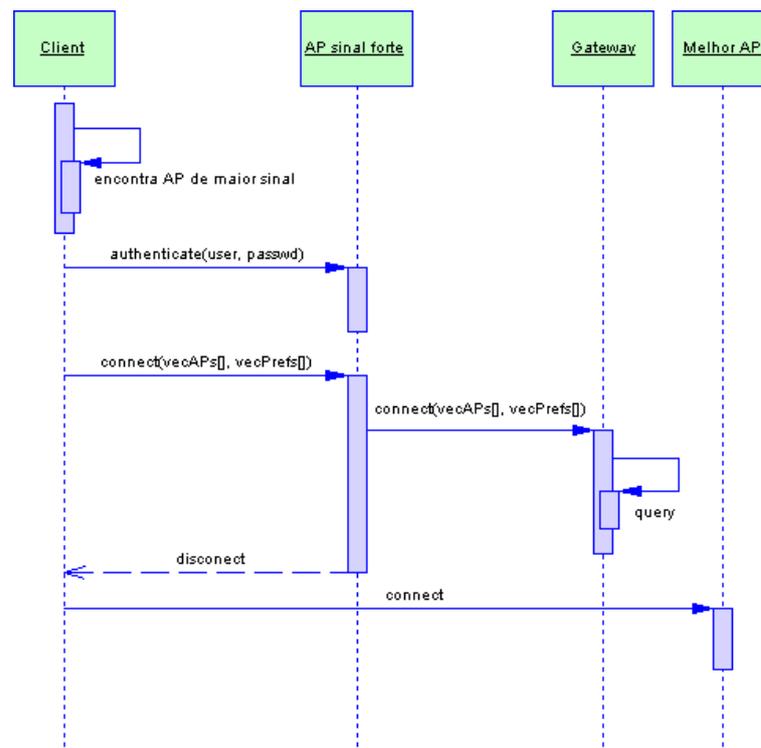


Figura 5.7: Proposta de arquitetura centralizada

não pelo cliente.

Diferentemente do primeiro cenário, em que o cliente se conecta inicialmente ao AP de maior intensidade de sinal, aqui ele solicita esse fator de qualidade a cada um dos APs que ele consegue ouvir. Os APs por sua vez, além de enviarem o fator de qualidade, enviam também as dimensões de QoS nas quais os clientes estejam interessados, com por exemplo o número de usuários conectados aquele AP ou o tráfego cursado naquele instante. O cliente, de posse das dimensões de QoS e do fator de qualidade é capaz de calcular localmente uma função utilidade sobre os APs em que ele tiver interesse, ponderando a importância das dimensões de QoS e do fator de qualidade. Um ponto negativo é que clientes localizados na mesma área e com as mesmas preferências podem se conectar a um mesmo AP causando pontos de grande concentração.

Assim como na arquitetura centralizada, é possível montar um banco de dados, que desta vez deve ser pequeno e localizado no próprio AP. Esse banco será tão rico quanto mais recursos de processamento e memória o AP puder oferecer. Diferentemente do caso centralizado, as informações serão apenas locais ao AP. Por exemplo, se um cliente se conectou a um mesmo AP no qual anteriormente consumiu muita banda, o AP pode implementar um controle de admissão que baixe seu fator de qualidade antes de enviá-lo a este cliente ou que simplesmente negue o acesso a ele, caracterizando que o cliente está

em sua “lista negra”¹⁰. Porém, caso esse cliente se conecte a outro AP, essa reação por parte do sistema não será imediatamente possível em virtude da falta de um banco de dados único¹¹: este AP levaria um tempo aprendendo que aquele cliente é um grande consumidor antes de também inseri-lo em sua “lista negra”.

Outra consequência é que os valores das dimensões de QoS não podem ser mantidos em sigilo pelo AP como no caso anterior. Neste caso, elas devem ser enviadas ao cliente para que ele calcule a função utilidade. Na arquitetura distribuída, são as preferências do cliente que são mantidas em sigilo. A Figura 5.8 resume as etapas mais importantes descritas no processo de seleção do melhor AP.

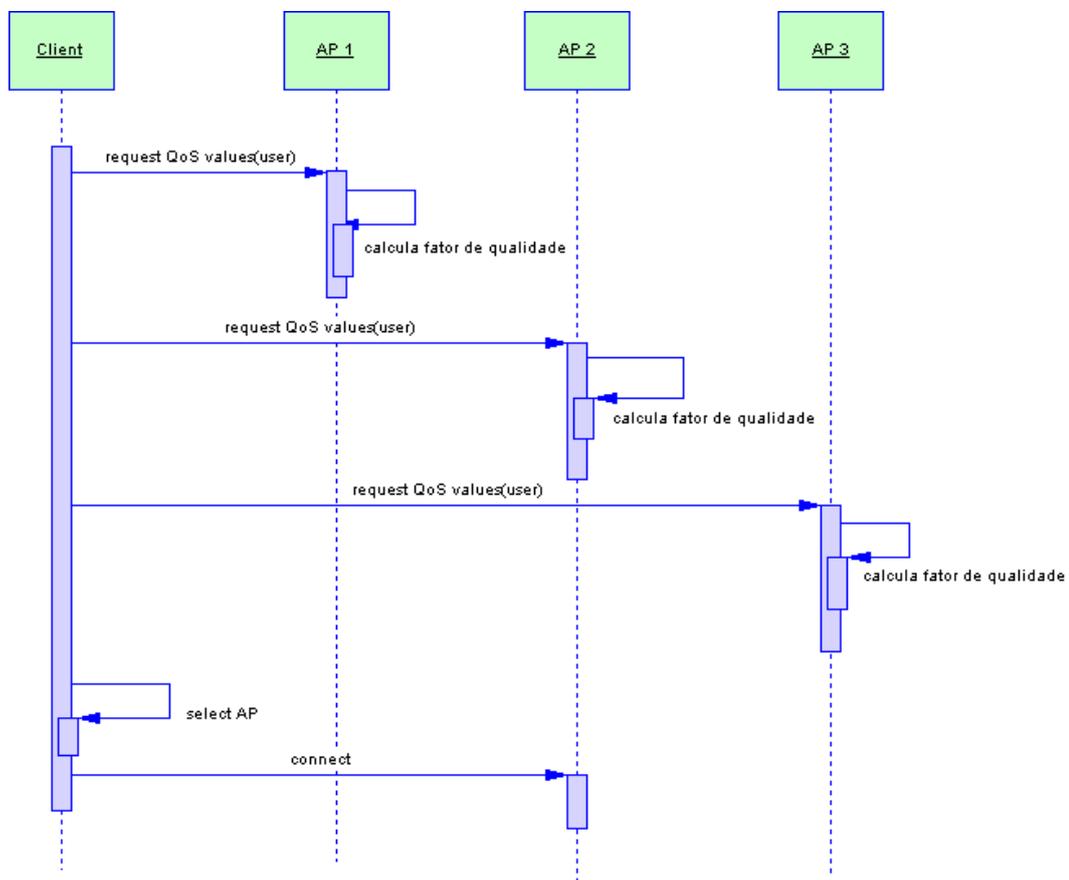


Figura 5.8: Proposta de arquitetura distribuída

¹⁰É importante lembrar que o envio de um fator de qualidade baixo para o cliente não é garantia de que o cliente não vá se conectar aquele AP. O cliente pode desconsiderá-lo se atribuir ao fator de qualidade um peso pequeno.

¹¹Isso pode não ser um problema se for verificado que os clientes dessa rede tem uma baixa taxa de mobilidade, porque, por exemplo, usam *laptops* ao invés de dispositivos PDA.

5.2.3 Descrição dos experimentos para a arquitetura centralizada

A arquitetura centralizada tem a vantagem de ter um conhecimento global da rede e de permitir clientes mais leves em termos de processamento. Adicionalmente, o desenvolvedor tem maior controle sobre a carga de cada AP uma vez que a escolha do melhor AP é coordenada de forma centralizada. O perigo de ter um ponto central de falha (**Seletor** primário) pode ser minimizado com a utilização de redundância (**Seletor** secundário), que além de aumentar o tempo entre falhas, poderia balancear a carga de processamento do **Seletor** primário. Já a arquitetura distribuída necessita de clientes com maior poder de processamento e de APs com alguma memória para manter um banco de dados local, porém espera-se dela uma maior escalabilidade. Ponderando as vantagens e desvantagens, escolhemos a versão centralizada da arquitetura para verificar pontos importantes envolvidos na escolha do melhor AP.

Foi implementado um simulador baseado em eventos discretos e composto basicamente de um módulo **Generator** (que é responsável por gerar *traces* contendo eventos de transmissão, escolha do melhor AP e movimentação dos clientes de acordo com distribuições de probabilidade realísticas) e um módulo **Simulator** que, em uma segunda fase, efetivamente consome e processa esses eventos. Por fim, o módulo **_Analyse** é responsável por analisar os resultados e gerar as estatísticas que servirão para comparar as execuções feitas com diferentes parametrizações, conforme será discutido adiante. Essa montagem é típica nesse tipo de experimento pois é baseada em tempo de simulação que geralmente é diferente do tempo real¹². Além disso, é possível que o módulo **Simulator** processe o mesmo *trace* de eventos com diversas parametrizações diferentes para a função utilidade.

O diagrama de classes da implementação do simulador orientado a eventos discretos está mostrado na Figura 5.9. As classes **Generator**, **Simulator** e **_Analyse** estão exibidas no topo e as classes que efetivamente compõem o modelo de dados (**Event**, **AccessPoint**, **Field** e **Client**) estão mais abaixo. Apenas os métodos/atributos mais importantes foram exibidos. A classe **Event** é compartilhada entre **Generator** e **Simulator**.

Isto posto, o primeiro passo foi gerar *traces* de eventos para que pudessem ser processados em seguida. O módulo **Generator** possui basicamente três rotinas que foram executadas sequencialmente (e independentemente) para cada cliente participante da simulação. As rotinas partiam de um tempo de simulação igual a zero e, ao término da

¹²Simuladores de tempo real têm a quantidade de eventos e objetos simultâneos limitados pela capacidade de processamento e memória das máquinas onde são executados, enquanto aqueles baseados em tempo de simulação não sofrem deste problema.

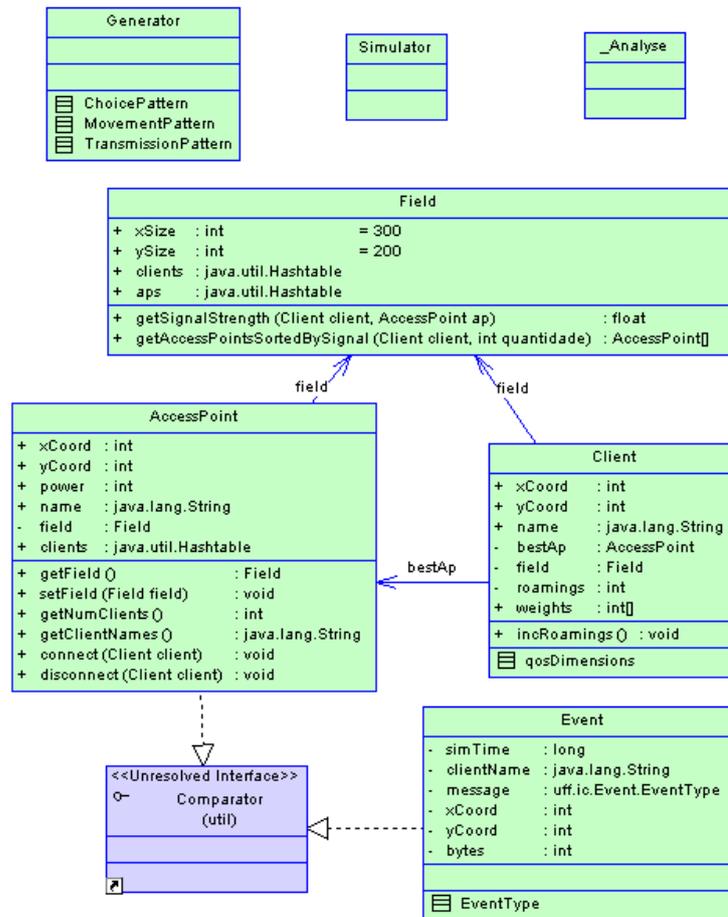


Figura 5.9: Diagrama de classes do simulador

execução com o tempo de simulação arbitrado em 50.000, uma estrutura de dados única era responsável por ordenar os eventos gerados independentemente por cada uma das rotinas para cada cliente e gerar um arquivo *trace* único de saída. As três rotinas estão descritas a seguir:

- **ChoicePattern**: responsável por gerar eventos que indicam o momento em que o cliente deveria escolher o melhor AP (com base em suas preferências, que foram consideradas constantes durante toda a duração das simulações, e na disponibilidade de recursos do ambiente) segundo a técnica pró-ativa citada anteriormente. Neste experimento, arbitramos que os clientes escolhiam o melhor AP a cada intervalo de tempo cuja função de distribuição de probabilidade era Normal de média 500 unidades de tempo e desvio padrão 20 unidades de tempo. Uma vez que a duração total da experiência era de 50.000 unidades de tempo, essa distribuição de probabilidade fez os clientes escolherem o melhor AP cerca de 100 vezes.
- **MovementPattern**: responsável por gerar eventos de movimentação dos clientes.

Neste experimento, foi adotado um padrão em que os clientes se moviam entre zero e 20 unidades de espaço em cada direção X e Y (com função de distribuição de probabilidade uniforme) a cada 50 unidades de tempo de simulação, conforme foi empregado em [Bertini et al. 2006]. A movimentação aconteceu sobre uma área arbitrada de 300x200 unidades de espaço.

- **TransmissionPattern**: responsável por gerar os eventos de transmissão de bytes para o AP que o cliente tinha correntemente como melhor. A quantidade de bytes transmitidos seguiu uma distribuição de Pareto e os intervalos entre transmissões seguiram uma distribuição Exponencial com média 5 unidades de tempo. Essas distribuições foram parametrizadas conforme encontradas no trabalho de [Cicconetti et al. 2006] onde os autores também se valem de simulação para estudar como se comporta o mecanismo de garantia de QoS em sistemas WiMax.

Foram gerados 2 cenários de 2 APs cada. O primeiro (*Rodada01.trace*) contava com 10 clientes homogêneos cuja a distribuição da probabilidade do tamanho das mensagens seguia uma distribuição de Pareto com parâmetros (1.1, 45). O segundo cenário (*Rodada02.trace*) continha clientes heterogêneos: dois dos dez clientes tinham o tamanho das mensagens seguindo uma distribuição de Pareto com parâmetros (1.1, 900); os outros oito clientes eram parametrizados com (1.1, 45). É interessante notar que o volume de tráfego gerado por clientes simulados não precisa ser igual em valores absolutos àquele obtido com clientes reais, visto que estamos interessamos em comparar o tráfego entre APs; em outras palavras, se os parâmetros 45 e 900 das distribuições anteriores fossem dobrados por exemplo, iríamos obter os mesmos resultados quando comparássemos os APs. A seguir, mostramos o aspecto dos arquivos de *trace* gerados pelas três rotinas citadas anteriormente.

Código 5.1: Trecho do arquivo de *trace* Rodada01.trace

```
500 c002 movement 280 48
500 c001 movement 225 1
503 c008 transmission 295 bytes
503 c002 transmission 109 bytes
5 504 c005 transmission 170 bytes
504 c004 transmission 107 bytes
504 c003 transmission 51 bytes
504 c002 transmission 96 bytes
505 c004 transmission 65 bytes
10 506 c010 choice
```

A primeira linha do Código 5.1, mostra que no tempo de simulação 500 ocorreu um evento no qual o cliente c002 se locomoveu do ponto onde estava para a posição (280, 48) do espaço. A linha cinco, mostra um evento de transmissão no qual o cliente c005 enviou 170 bytes para o AP que ele entendia ser o melhor naquele instante. Por fim, a linha 10 mostra um evento no qual o cliente c010 escolhe o melhor AP com base em sua função utilidade. Para os dez clientes simulados, este arquivo contém 101.804 eventos.

Sumário dos cenários considerados

- clientes homogêneos: 10 clientes com mesmo padrão de tráfego, movimentação e escolha deveriam selecionar o melhor AP
- clientes heterogêneos: 2 clientes com um padrão de tráfego cerca de 20 vezes superior aos dos outros 8 clientes participantes. Os padrões de movimentação e escolha dos 10 clientes são idênticos entre si e aos do cenário anterior.

Objetivos do experimento

- gerar dois cenários que permitissem avaliar a dificuldade que a função utilidade teria em balancear a carga de clientes heterogêneos, quando comparado ao cenário só com clientes homogêneos. Essa idéia surgiu da leitura do artigo de [Balazinska e Castro 2003] onde foi constatado que poucos usuários contribuíam para uma grande quantidade de tráfego.
- avaliar como a parametrização do mecanismo decisório (pesos da função utilidade e tamanho das janelas deslizantes, explicadas mais à frente) influenciava a qualidade geral do balanceamento da carga.

Na segunda fase (em que o `Simulator` processa os eventos gerados), os clientes usavam quatro métricas distintas (ou dimensões de QoS) para monitorar o estado de um AP. Cada uma dessas métricas servia de argumento para funções de satisfação que, finalmente, eram ponderadas antes de participar do somatório da função utilidade. As métricas estão descritas a seguir:

- `signalStrength`: representa a intensidade de sinal do AP percebida pelo cliente. Essa intensidade foi sintetizada com base nas posições do cliente e do AP. Tal como

em [Bertini et al. 2006], a intensidade foi modelada como sendo inversamente proporcional ao quadrado da distância que separa o cliente do AP. É importante mencionar que os APs estavam localizados igualmente espaçados na área de simulação que tinha 300x200 unidades, ou seja, nas coordenadas (75, 100) e (225, 100).

- `usersQtd`: representa a quantidade de usuários que têm aquele AP escolhido como melhor.
- `lastK1Messages`: implementa uma janela deslizante contendo o somatório da quantidade de bytes enviados para o AP considerando-se as últimas K1 mensagens. K1 é um parâmetro variado durante o experimento.
- `lastK2TimeUnits`: implementa uma janela deslizante contendo o somatório da quantidade de bytes enviados para o AP considerando-se as últimas K2 unidades de tempo de simulação. K2 é um parâmetro variado durante o experimento. É importante lembrar que essa métrica é diferente da anterior pois o espaçamento entre transmissões é distribuído estatisticamente, conforme já mencionado.

Para a valoração das últimas duas métricas, o `Simulator` consultava o banco de dados que foi enriquecido para persistir, de forma centralizada, estatísticas sobre as mensagens trocadas entre clientes e APs. Este banco de dados foi mencionado anteriormente como sendo mantido pelo *gateway*.

Para a análise dos resultados, o módulo `_Analyse` amostrava o tráfego, em bytes, de cada AP participante da simulação a cada 500 unidades de tempo de simulação, que corresponde basicamente ao intervalo de tempo entre duas escolhas sucessivas de um cliente, e calculava um índice de balanceamento de carga¹³, definido a seguir na Equação 5.1. Este mesmo módulo está preparado para gerar outros relatórios, como por exemplo, um que mostre como variou a utilidade de cada AP, segundo cada cliente ou um que mostre a quantidade de clientes conectados em cada AP.

Conforme dito anteriormente, essas métricas não são utilizadas diretamente na função utilidade. Uma vez que elas têm unidades diferentes (dBs, números de clientes conectados e bytes) e seus valores podem variar dentro de escalas completamente diferentes, nós fizemos que funções de satisfação fossem operadas sobre elas, conforme mostrado na Seção 4.1. Nesta implementação foram usadas funções lineares do tipo mínimo-máximo decrescentes para a quantidade de usuários, os bytes transmitidos nas últimas K1 mensagens

¹³Aqui os termos “tráfego” e “carga” se equivalem.

e os bytes transmitidos nas últimas K2 unidades de tempo de simulação (uma vez que quanto maior o valor dessas métricas, menor é a satisfação do usuário). Para a intensidade do sinal foi utilizada uma função de satisfação linear do tipo mínimo-máximo crescente. Cada uma dessas funções retorna um número entre 0 e 1, normalizando a métrica. Por fim, sobre cada uma dessas funções são aplicados pesos para que o cliente tenha a oportunidade de mostrar ao *framework* a importância relativa de uma métrica em relação às outras. Todos esses parâmetros serão discutidos adiante (ver Tabelas 5.3 e 5.4).

Após o cálculo parametrizado da utilidade de cada AP no qual o cliente tem interesse, duas políticas de escolha foram implementadas. A primeira retorna ao cliente o AP de maior utilidade. Conforme já dito anteriormente, caso clientes com preferências parecidas enxerguem o mesmo conjunto de APs, o sistema pode experimentar uma oscilação: todos esses clientes vão escolher sempre o mesmo AP como o melhor, sobrecarregando-o. A segunda política implementada utiliza o método da roleta, muito conhecido nos algoritmos genéticos: cada AP do conjunto tem uma probabilidade, diretamente proporcional a sua utilidade, de ser escolhido.

Para verificarmos quantitativamente a qualidade da distribuição da carga entre os APs, é utilizado um índice de balanceamento de carga, citado anteriormente e também adotado em [Balachandran et al. 2002]. Este índice é interessante pois se aproxima de $1/n$ (onde n é o número de APs) quando o desequilíbrio é máximo e se aproxima de 1 quando os APs têm a mesma carga. O índice de balanceamento de carga é definido abaixo, onde B_i é a carga (tráfego amostrado em bytes pelo módulo `_Analyse` a cada 500 unidades de tempo de simulação) do i -ésimo AP:

$$\beta = \frac{(\sum B_i)^2}{n * \sum B_i^2} \quad (5.1)$$

5.2.4 Resultados

Inicialmente, foram executadas 40 análises: 20 análises para cada um dos cenários citados parametrizando-se a função utilidade e as janelas deslizantes conforme as Tabelas 5.3 e 5.4, que contêm também os resultados obtidos para cada parametrização. A segunda coluna refere-se aos pesos das métricas `signalStrength`, `usersQtd`, `lastK1Messages` e `lastK2TimeUnits`. As colunas 4, 5, 6 e 7 mostram se os clientes estavam em movimento, se o método da roleta foi empregado e o valor dos parâmetros K1 e K2 respectivamente. A média e o desvio padrão referem-se ao índice de balanceamento descrito na Equação 5.1 e

à quantidade de *roamings* realizados pelos clientes. Vale lembrar que, como a experiência usa dois APs, o índice de balanceamento varia entre 0,5 (completo desbalanceamento) e 1 (completo balanceamento). Outro fato importante é que uma vez que a experiência dura 50.000 unidades de tempo e que cada cliente escolhe o melhor AP a cada 500 unidades de tempo em média (através de uma distribuição normal de média 500 e 20 de desvio padrão), cada cliente poderia fazer, em média, 100 *roamings*. Executar estes testes várias vezes sobre os mesmos arquivos de *traces* `Rodada01.trace` e `Rodada02.trace` não traria um benefício pois encontraríamos os mesmos resultados, uma vez que a parte não-determinística dessa experiência reside apenas na geração destes dois arquivos pelo componente `Generator`.

Cenário com clientes homogêneos

Grupo	Pesos dos clientes {signalStrength, usersQtd, lastK1Messages, lastK2TimeUnits}	Análise	Clientes em movimento	Método da Roleta	lastK1 Messages	lastK2 TimeUnits	Índice de balanceamento		Roamings	
							média	desvio padrão	média	desvio padrão
G1	{1, 0, 0, 0}	1	não	não	10	500	0.842	0.099	0.000	0.000
		2	não	sim			0.864	0.120	26.200	16.498
		3	sim	não			0.900	0.110	13.000	5.696
		4	sim	sim			0.915	0.098	30.500	4.836
G2	{0, 1, 0, 0}	5	não	não	10	500	0.944	0.082	30.800	21.374
		6	não	sim			0.929	0.079	52.900	5.607
		7	sim	não			0.943	0.080	34.200	5.789
		8	sim	sim			0.924	0.090	53.700	4.423
G3	{1, 1, 1, 1}	9	não	não	10	500	0.943	0.075	41.800	17.152
		10	não	sim			0.916	0.094	53.100	4.701
		11	sim	não			0.944	0.081	43.300	4.111
		12	sim	sim			0.911	0.097	53.700	5.078
G4	{1, 1, 3, 3}	13	não	não	10	500	0.898	0.113	61.200	11.998
		14	não	sim			0.909	0.098	52.300	4.739
		15	sim	não			0.907	0.109	56.300	6.977
		16	sim	sim			0.902	0.101	54.100	3.604
G5	{0, 0, 1, 1}	17	não	não	10	500	0.852	0.144	67.300	4.855
		18	não	sim			0.908	0.099	51.900	2.998
		19	sim	não			0.847	0.148	67.600	4.006
		20	sim	sim			0.910	0.091	51.800	5.007

Tabela 5.3: Resultados do cenário Rodada01 (clientes homogêneos)

No primeiro grupo de quatro análises da Tabela 5.3, apenas a intensidade do sinal é levada em consideração e, nos casos em que os clientes se movimentavam, o índice de balanceamento ficou maior que 90%. Esse fato pode ser explicado pela homogeneidade dos clientes: eles geravam o mesmo perfil de tráfego e se movimentavam uniformemente pelo espaço, que tinha dois APs simetricamente distribuídos. Assim, a carga foi distribuída com qualidade razoável.

No segundo grupo de quatro análises a única métrica empregada foi a quantidade de usuários conectados ao AP. Aqui, mais uma vez, como os clientes são idênticos, o número de clientes conectados a um determinado AP permitiu ao sistema inferir a carga gerada por eles e com isso, balancear a carga novamente com qualidade satisfatória.

No terceiro grupo de quatro análises, as métricas `lastK1Messages` e `lastK2TimeUnits` entraram no cálculo da função utilidade e os parâmetros K1 e K2 foram valorados com 10 e 500 respectivamente, o que significa que o total de bytes transmitidos para o AP nas últimas 10 mensagens juntamente com o total de bytes transmitidos para o AP nas últimas 500 unidades de tempo de simulação foram levados em conta.

O primeiro parâmetro foi escolhido dessa forma porque, se temos 10 clientes com mesma distribuição probabilística de transmissão, então estatisticamente cada uma das 10 últimas mensagens recebidas pelo AP é proveniente de um deles. Já o segundo parâmetro foi valorado com 500 porque cada cliente escolhe o melhor AP em média a cada 500 unidades de tempo, ou seja, em uma janela de 500 unidades de tempo estatisticamente todos os clientes terão selecionado o melhor AP. É possível verificar que a qualidade do balanceamento não foi positivamente alterada mas a quantidade de *roamings* sofreu um aumento considerável. Essa instabilidade pode ser explicada porque neste grupo levamos em conta métricas relacionadas ao tráfego cursado pelo AP que inerentemente apresenta uma grande dinâmica, conforme citado anteriormente nas Figuras 5.1 e 5.3 e na Seção 5.1.1.1. Uma alternativa seria dar um peso pequeno a estas métricas ou variar o tamanho das janelas deslizantes¹⁴. Dessa forma, essa parametrização deve ser evitada num cenário de clientes homogêneos.

No quarto e quinto grupos essas métricas recebem ainda mais importância, primeiro recebendo um peso três e depois sendo as únicas levadas em consideração. Pelos motivos acima expostos, é possível explicar porque a quantidade de *roamings* aumentou nestes dois grupos.

¹⁴No cenário com clientes heterogêneos, usamos um tamanho diferente para as janelas deslizantes deste grupo para diminuir a quantidade de *roamings*, conforme será mostrado adiante.

Cenário com clientes heterogêneos

Grupo	Pesos dos clientes {signalStrength, usersQtd, lastK1Messages, lastK2TimeUnits}	Análise	Clientes em movimento	Método da Roleta	lastK1 Messages	lastK2 TimeUnits	Índice de balanceamento		Roamings	
							média	desvio padrão	média	desvio padrão
G1	{1, 0, 0, 0}	21	não	não	10	500	0.573	0.036	0.000	0.000
		22	não	sim			0.640	0.135	26.100	15.509
		23	sim	não			0.731	0.173	10.500	5.169
		24	sim	sim			0.768	0.168	27.600	5.441
G2	{0, 1, 0, 0}	25	não	não	10	500	0.762	0.154	32.200	22.493
		26	não	sim			0.812	0.152	52.000	5.457
		27	sim	não			0.809	0.152	34.900	5.934
		28	sim	sim			0.827	0.150	51.400	4.858
G3	{1, 1, 1, 1}	29	não	não	10	500	0.847	0.139	47.600	16.501
		30	não	sim			0.825	0.151	52.600	4.858
		31	sim	não			0.876	0.127	46.800	5.633
		32	sim	sim			0.834	0.152	52.300	4.523
G4	{1, 1, 3, 3}	33	não	não	10	500	0.842	0.152	62.300	11.615
		34	não	sim			0.826	0.157	52.200	4.940
		35	sim	não			0.858	0.133	59.300	7.088
		36	sim	sim			0.831	0.146	52.100	5.131
G5	{0, 0, 1, 1}	37	não	não	10	500	0.746	0.160	64.300	7.959
		38	não	sim			0.809	0.144	52.000	6.446
		39	sim	não			0.746	0.160	64.300	8.015
		40	sim	sim			0.810	0.151	49.800	3.994
G6	{1, 1, 1, 1}	41	sim	não	30	1500	0.874	0.131	41.400	7.457
		42	sim	sim			0.838	0.148	49.700	5.122

Tabela 5.4: Resultados do cenário Rodada02 (clientes heterogêneos) (A)

No segundo cenário, onde 2 dos 10 clientes tinham um perfil de tráfego cerca de 20 vezes mais pesado que os outros 8 clientes, os resultados obtidos foram diferentes. Este cenário pode ser considerado mais realístico do que o anterior porque, conforme verificado em experiências reais ([Balazinska e Castro 2003]), poucos clientes são responsáveis pela maioria do tráfego.

No primeiro grupo de quatro análises da Tabela 5.4, apenas a intensidade do sinal foi utilizada e a qualidade do balanceamento da carga foi pior do que aquele obtido com esta mesma parametrização no primeiro cenário. Dentro desse grupo esse índice só se tornou razoável quanto os clientes se movimentavam, introduzindo naturalmente um balanceamento de carga entre os APs. Assim, é interessante observar que a única métrica geralmente disponível para os clientes escolherem o melhor AP (a intensidade do sinal) não funciona bem em um cenário realístico com clientes heterogêneos, conforme verificado também em [Judd e Steenkiste 2002].

No segundo grupo de quatro análises, apenas a quantidade de usuários conectados ao AP serviu para tentar balancear a carga e, como pode ser visto, a qualidade atingida não foi tão boa quanto no primeiro cenário com clientes homogêneos para essa mesma parametrização: o sistema não conseguiu inferir a quantidade de tráfego cursado em um determinado AP através do número de clientes conectados a ele.

No terceiro grupo as métricas `lastK1Messages` e `lastK2TimeUnits` foram levadas em consideração e parametrizadas com 10 e 500 respectivamente, pelas mesmas razões explicadas anteriormente. A qualidade do balanceamento foi aumentada novamente às custas de um maior número de *roamings*. Uma parametrização que leve os clientes a realizarem mais *roamings* pode ser ruim para eles caso estejam usando uma aplicação que exija procedimentos computacionalmente caros para adaptar-se, como por exemplo componentes que necessitem ter seu contexto salvo entre adaptações. Outro exemplo pode ser o cliente que assiste a um vídeo e percebe uma pequena interrupção no fluxo multimídia quando ele se conecta a outro AP. Mais adiante, serão feitas análises extras que visam minimizar o aumento de *roamings* introduzido pela utilização das métricas `lastK1Messages` e `lastK2TimeUnits`.

No quarto grupo a importância das métricas acima foi aumentada sem que a qualidade do balanceamento variasse sensivelmente, o que difere do resultado obtido no primeiro cenário onde uma maior importância dada à carga gerada por clientes homogêneos acabou por piorar a qualidade do balanceamento.

No quinto grupo de quatro análises, ao utilizar apenas as métricas `lastK1Messages`

e `lastK2TimeUnits` foi possível perceber uma pequena piora na qualidade do balanceamento, porém não tão grande quanto no primeiro cenário. Esse fato pode ser explicado pois foram levados em conta exclusivamente dados de tráfego que, sabemos, são inerentemente instáveis. É interessante notar que a utilização do método da roleta no quinto grupo contribuiu para melhorar o índice de balanceamento e reduzir o número de *roamings*, apesar de neste grupo levarmos em consideração apenas as dimensões de QoS referentes ao tráfego (`lastK1Messages` e `lastK2TimeUnits`). Parametrizando essas métricas dessa maneira, esperávamos testar ao máximo neste grupo a estabilidade do *framework* pois, como já mencionado, essas dimensões de QoS variam muito uma vez que representam o tráfego cursado pelo AP.

Adicionalmente, para verificar a importância da parametrização das métricas `lastK1Messages` e `lastK2TimeUnits`, repetimos as análises que apresentaram os melhores resultados em termos de balanceamento de carga (análises 31 e 32) parametrizando K1 e K2 com 30 e 1500 respectivamente, ou seja, causando um aumento do tamanho dessas janelas deslizantes. Os resultados podem ser vistos nas análises 41 e 42: a qualidade do balanceamento praticamente não foi alterada. Porém, a média de *roamings* foi reduzida, expressando o caráter estabilizador das janelas deslizantes.

Também buscando reduzir o número de *roamings*, estudamos na Tabela 5.5 o emprego de outro fator: o custo de reconfiguração introduzido em [Poladian et al. 2006]. Conforme discutido anteriormente, esse custo é diretamente proporcional à intolerância do usuário por reconfigurações (adaptações) que o *framework* faz automaticamente. Essa intolerância depende da aplicação executada pelo usuário, por exemplo, quando ele está assistindo um vídeo pode ser que ele prefira terminar de assisti-lo com fluxo de baixa qualidade do que perceber pequenas interrupções devidas à adaptações feitas pelo *framework*. Neste teste, repetimos as análises 31 e 35¹⁵ da Tabela 5.4 introduzindo o custo de reconfiguração entre 5% e 100%. Em outras palavras, na análise 61 por exemplo, o cliente só se realiza o *roaming* se o AP escolhido pelo mecanismo tiver uma utilidade 5% maior do que o AP ao qual ele está conectado. É possível verificar em todas as análises que essa técnica sempre reduziu o número de *roamings* dos clientes, contribuindo para a estabilização do sistema. É interessante notar na análise 65, em que o custo de reconfiguração foi ajustado em 100% (exagerado), que o sistema se tornou menos ótimo (o índice de balanceamento de carga foi reduzido).

¹⁵Essas análises foram escolhidas pois em média apresentaram os melhores índices de balanceamento e não têm a influência do método da roleta que é não-determinístico

Grupo	Pesos dos clientes {signalStrength, usersQtd, lastK1Messages, lastK2TimeUnits}	Análise	Clientes em movimento	Método da Roleta	lastK1 Messages	lastK2 TimeUnits	Reconfiguration Cost	Índice de balanceamento		Roamings	
								média	desvio padrão	média	desvio padrão
G1	{1, 1, 1, 1}	31	sim	não	10	500	0%	0.876	0.127	46.800	5.633
		61						0.882	0.122	33.900	7.534
		62						0.890	0.120	26.600	4.115
		63						0.898	0.125	14.800	2.530
		65						0.793	0.152	1.600	1.075
G2	{1, 1, 3, 3}	35	sim	não	10	500	0%	0.858	0.133	59.300	7.088
		66						0.876	0.130	44.000	6.074
		67						0.894	0.129	39.000	8.233
		68						0.880	0.136	22.800	6.197
		70						0.884	0.143	2.600	2.757

Tabela 5.5: Resultados do cenário Rodada02 (clientes heterogêneos) (B)

5.3 Conclusão do capítulo

Neste capítulo analisamos em detalhes o emprego de uma função utilidade na aplicação em que clientes escolhem o melhor AP de uma rede sem-fio com base em suas preferências e na disponibilidade de recursos do ambiente. Na primeira seção, propomos um cenário em que um cliente hipotético parado processava *traces* de utilização de uma grande rede sem-fio (com clientes reais). Esse estudo nos permitiu concluir que a função utilidade constitui uma solução robusta e de qualidade para selecionar o melhor recurso quando comparada a outras técnicas, como por exemplo, a que escolhe o AP com menos usuários conectados. Verificamos que a janela deslizante é uma técnica simples que pode ser usada em conjunto para amortecer as flutuações da disponibilidade dos recursos do ambiente e evitar oscilações do mecanismo de seleção (que comprovamos através da redução do número de *roamings*). Além disso, verificamos que o número de clientes conectados a um AP não está fortemente correlacionado ao tráfego cursado naquele AP.

Na segunda seção, discutimos duas versões de arquitetura (centralizada e distribuída) para a organização dos APs, citando as vantagens e desvantagens de cada uma. Em seguida, um simulador para a arquitetura centralizada (pois esta permitia maior controle sobre o experimento) foi desenvolvido e parametrizado com distribuições realísticas de probabilidade (de movimentação dos clientes, transmissão de mensagens e escolha de APs), o que nos permitiu mostrar que a intensidade do sinal utilizado nas soluções comerciais para a escolha do melhor AP é pouco indicativa da qualidade deste AP. Em vez disso, sugerimos a aplicação de uma técnica mais representativa que leve em conta as preferências dos clientes, ponderando dimensões de QoS possivelmente conflitantes: mais uma vez a função utilidade se mostrou uma técnica flexível o suficiente tanto no cenário de clientes homogêneos quanto no cenário, mais realístico, de clientes heterogêneos. Ainda na segunda seção, através da redução do número de *roamings* mostramos o efeito estabilizador de técnicas como a janela deslizante e o custo de reconfiguração. A primeira está intrinsecamente ligada à dinâmica da disponibilidade dos recursos monitorados e deve ser parametrizada pelo desenvolvedor que entende dessa dinâmica. A segunda expressa a intolerância do usuário por adaptações e deve ser parametrizada por ele uma vez que depende da aplicação executada.

Capítulo 6

Conclusões e trabalhos futuros

Este foi um estudo exploratório sobre como funções utilidade podem ajudar no suporte à adaptação de aplicações pervasivas. Ele permitiu ao leitor conhecer questões envolvidas neste tema.

No Capítulo 1, ressaltamos a importância das aplicações pervasivas na atualidade. Elas são sensíveis ao contexto onde operam (ou seja, à disponibilidade dos recursos do ambiente e às preferências do usuário) para que seja possível entregar um nível de qualidade adequado e satisfazer ao usuário.

No Capítulo 2, enumeramos os principais aspectos envolvidos e as questões que devem ser resolvidas para que a seleção do melhor recurso seja possível. Entre essas questões, temos a descoberta, descrição e monitoração dos recursos; a descrição arquitetural e o mapeamento físico. A exposição dessas questões serviu também para estabelecermos o vocabulário que usamos ao longo deste trabalho.

No Capítulo 3, ressaltamos que a solução para o problema da adaptação de aplicações pervasivas deveria ser reusável. Por esse motivo, citamos alguns trabalhos baseados em *frameworks* e que serviram também de motivação para a proposta que fizemos nos capítulos seguintes. Através de *frameworks*, o mecanismo de adaptação é fatorado da aplicação e pode ser configurado por políticas definidas externamente, facilitando a separação de interesses.

No Capítulo 4, listamos alguns exemplos de aplicação. Propomos o emprego de uma função utilidade para realizar a adaptação necessária por entender que ela constitui uma técnica que leva em conta as preferências dos usuários (que ponderam dimensões de QoS frequentemente conflitantes) e a disponibilidade dos recursos que impactam na qualidade da aplicação. Outra virtude dessa técnica é a facilidade de implementação e a pouca

quantidade de processamento exigida, propiciando ser utilizada em pequenos dispositivos com recursos computacionais e de energia restritos, tais como PDAs.

No Capítulo 5, analisamos em profundidade a aplicação que selecionava o melhor AP, apesar de haver poucos trabalhos abordando esse assunto. Iniciamos nosso estudo analisando *traces* reais de utilização de uma grande rede sem-fio corporativa. Além disso, propomos duas arquiteturas (centralizada e distribuída) para a aplicação analisada. Com o intuito de ter mais variáveis sob controle, desenvolvemos um simulador orientado a eventos discretos que foi parametrizado com distribuições realísticas de probabilidade.

6.1 Conclusões

Propomos que a política para a escolha do melhor AP deve se valer de uma função multi-variável (função utilidade) em que os clientes possam expressar suas preferências, ponderando as diversas dimensões de QoS envolvidas. Também é importante por parte do desenvolvedor do serviço, explicitar quais dimensões de QoS são importantes de serem levadas em conta para sua qualidade (*service-specific knowledge* segundo [Huang 2004]).

Com os testes realizados no Capítulo 5, pudemos concluir que a dimensão de QoS tipicamente disponível para os usuários de redes móveis realizarem a seleção do melhor AP, a intensidade de sinal, não é representativa para promover o balanceamento de carga entre os APs com igualdade e, conseqüentemente, oferecer o serviço com qualidade. O número de usuários também não deve ser utilizado isoladamente uma vez que uma rede real contém clientes heterogêneos, ou seja, considerando apenas o número de clientes, não se consegue inferir a quantidade de tráfego gerado por eles para buscar o balanceamento.

Mostramos também que um método determinístico que retorna o melhor AP baseado em sua utilidade também não é sempre interessante pois induz o sistema a oscilações. Isso ficou evidenciado principalmente na análise do cenário heterogêneo (ver Tabela 5.4) quando a utilização do método da roleta melhorou o índice de balanceamento e reduziu o número de *roamings*. Dessa forma, quando os clientes são modelados como não-cooperativos, sugerimos uma técnica como o método da roleta ou o vetor de servidores equivalentes como usado em [Zegura et al. 2000].

As arquiteturas (centralizada e distribuída) propostas na Seção 5.2 suportam diversas políticas de seleção de AP e de controle admissão dos clientes. Na versão centralizada, mencionamos que o mesmo banco de dados usado para persistir as estatísticas de utilização dos APs poderia ser enriquecido para manter um histórico de utilização da rede por cliente,

e com isso privilegiar (através de prioridades por exemplo) o acesso a rede de alguns em detrimento de outros.

A função utilidade pôde ser facilmente integrada ao *framework* CR-RIO, por exemplo, dando suporte ao seu mecanismo decisório (ver Seção 4.1.1 e [Cardoso et al. 2006]). Seja para selecionar o melhor recurso, seja para selecionar a melhor configuração admissível de uma arquitetura, a função utilidade comportou as preferências do cliente através de pesos de uma média ponderada e os diferentes intervalos de variação das dimensões de QoS através de funções de satisfação.

Para evitar oscilações do mecanismo de escolha, a monitoração dos recursos do ambiente deve ser filtrada. Nas aplicações deste trabalho, usamos uma janela deslizante que deve ser parametrizada não pelo grupo encarregado da monitoração, mas pelo grupo que vai efetivamente consumir seus dados, visto que apenas o último grupo tem a visão fim-a-fim do serviço que está sendo oferecido. Com o mesmo objetivo de estabilizar o mecanismo de escolha, mas parametrizado desta vez pelo próprio usuário pois expressa sua intolerância em aceitar adaptações, aplicamos também o conceito de custo de reconfiguração que se mostrou uma técnica barata computacionalmente e de fácil aplicação para este fim.

6.2 Trabalhos Futuros

Durante a realização deste estudo, muitas oportunidades para trabalhos futuros foram percebidas. Dessa forma, iremos nos ater ao escopo das análises apresentadas no Capítulo 5.

Alguns trabalhos se preocupam em prever a demanda por recursos do ambiente. No trabalho de [Narayanan et al. 2000], os autores propuseram uma solução em duas fases para prever o consumo de recursos para cada parametrização (ou fidelidade como se referem os autores) com a qual a aplicação era configurada. Na primeira fase, *off-line*, a aplicação é treinada com uma quantidade representativa de fluxos multimídia enquanto suas respectivas quantidades de recursos consumidos do ambiente eram monitoradas. Na segunda fase, *on-line*, a aplicação recebia um fluxo multimídia e podia prever a quantidade de recursos consumidos. Eventuais erros serviam para melhorar iterativamente o mecanismo de previsão. Os autores não obtiveram um bom índice de acertos porque alguns programas exibidores têm muitas *threads*, o que os faz apresentar um comportamento não-linear. Se os recursos demandados por uma configuração pudessem ser previstos com

confiabilidade, a função utilidade poderia levá-los em conta no momento de realizar a seleção da melhor alternativa, conforme sugerimos na Seção 4.3. Ou seja, o mecanismo de seleção não iria impor uma configuração que, em seguida, o serviço de monitoração descobriria que é incompatível com a disponibilidade dos recursos do ambiente.

Já o trabalho de [Poladian et al. 2005] modela as atividades do usuário em uma entidade mais abrangente denominada tarefa. Segundo os autores, é interessante otimizar o sistema levando em conta a duração de toda a tarefa do usuário (vender uma casa) ao invés da duração das pequenas atividades envolvidas (navegar na Internet, ler propostas em um editor de textos e etc); pois otimizar o sistema sucessivamente para cada atividade pode não ser ideal quando pensamos na tarefa como um todo (embora a adaptação para cada tarefa fosse ideal no momento de sua realização). Para isso, os autores usam processos estocásticos para modelar a variação da disponibilidade dos recursos do ambiente. Esses processos são simplificados para cadeias de Markov pois os autores sugerem que apenas o último valor monitorado deve ser levado em conta. Os autores citam que essa estratégia esbarra em duas grandes dificuldades para ser computacionalmente escalável: a granularidade da amostragem temporal e dos valores possíveis para cada recurso. Dessa forma, se os recursos disponíveis no ambiente pudessem ser previstos, a função utilidade poderia realizar uma otimização que seria ideal durante uma janela de tempo, ao invés de ser ideal apenas no instante presente.

A análise de *traces* reais poderia ser aprofundada caso houvesse na Universidade um conjunto de APs mais representativo, com mais tráfego. Dessa forma, os dados seriam disponibilizados para análise sem que previamente fossem tornados anônimos, conforme encontramos nos trabalhos que pesquisamos. Conforme sugerido em [Judd e Steenkiste 2002], poderíamos substituir a política de seleção de APs implementada no *firmware* das interfaces de rede por uma que contasse com uma função utilidade.

Por fim, os testes realizados para a versão centralizada da arquitetura poderiam ser refeitos usando-se a versão distribuída. Naquela arquitetura, não há um banco de dados único para reunir toda a informação de utilização da rede, ao contrário, essa informação é hospedada em cada AP individualmente e diz respeito somente a sua utilização. Em ambas as arquiteturas propostas, novas configurações poderiam ser testadas para aumentar o grau de heterogeneidade: os clientes além de possuírem um perfil de tráfego diferenciado (conforme realizamos), poderiam ser configurados individualmente por arquivos de inicialização (preferências) diferentes. Com isso, poderíamos simular classes de clientes que executam aplicações diferentes e, que por isso, têm objetivos diferentes: enquanto alguns

desejam maior largura de banda, outros desejam menor tempo de atraso por exemplo, conforme foi simulado em [Balachandran et al. 2002].

Referências

- [de Albuquerque et al. 2006] de Albuquerque, C. V. N.; Saade, D. C. M.; Passos, D. G.; Teixeira, D. V.; Leite, J.; Neves, L. E. e Magalhães, L. C. S. **Rede Mesh de Acesso Universitário Faixa Larga Sem Fio**. Universidade Federal Fluminense — Instituto de Computação, 2006.
- [Balachandran et al. 2002] Balachandran, A.; Voelker, G. e Bahl, P. **Hot-Spot Congestion Relief in Public-Area Wireless Networks**. *WMCSA*, p. 70–80, 2002.
- [Balazinska e Castro 2003] Balazinska, M. e Castro, P. **Characterizing Mobility and Network Usage in a Corporate Wireless Local-Area Network**. *Proceedings of MobiSys 2003: The First International Conference on Mobile Systems, Applications and Services*, USENIX Association, p. 303–3016, 2003.
- [Bertini et al. 2006] Bertini, L.; Loques, O. e Leite, J. **Replicação de Dados em Redes Ad Hoc para Sistemas de Apoio em Situações de Desastres**. Preprint submitted to Journal of Systems and Software, 2006.
- [Bhatti e Knight 1999] Bhatti, S. N. e Knight, G. **Enabling QoS adaptation decisions for Internet applications**. *Computer Networks (Amsterdam, Netherlands: 1999)*, v. 31, n. 7, p. 669–692, 1999.
- [de Campos e Saito 2004] de Campos, M. C. M. M. e Saito, K. **Sistemas Inteligentes em Controle e Automação de Processos**. : Ciência Moderna Ltda, 2004.
- [Capra et al. 2005] Capra, L.; Zachariadis, S. e Mascolo, C. **Q-CAD: QoS and Context Aware Discovery Framework for Adaptive Mobile Systems**. *International Conference on Pervasive Services (ICPS '05)*, p. 453–456, 2005.
- [Cardellini et al. 2002] Cardellini, V.; Casalicchio, E.; Colajanni, M. e Yu, P. S. **The state of the art in locally distributed Web-server systems**. *ACM Comput. Surv.*, ACM Press, New York, NY, USA, v. 34, n. 2, p. 263–311, 2002.
- [Cardoso et al. 2006] Cardoso, L.; Loques, O. e Sztajnberg, A. **Provendo aplicações com requisitos não-funcionais dinâmicos através de contratos**. *SBRC*, 2006.
- [Cheng et al. 2006] Cheng, S.-W.; Garlan, D. e Schmerl, B. **Architecture-based self-adaptation in the presence of multiple objectives**. *SEAMS '06: Proceedings of the 2006 international workshop on Self-adaptation and self-managing systems*, ACM Press, New York, NY, USA, p. 2–8, 2006.
- [Cheng et al. 2004] Cheng, S.-W.; Huang, A.-C.; Garlan, D.; Schmerl, B. e Steenkiste, P. **An Architecture for Coordinating Multiple Self-Management Systems**. *WICSA*, IEEE Computer Society, Los Alamitos, CA, USA, v. 00, p. 243, 2004.

- [Cicconetti et al. 2006] Cicconetti, C.; Lenzini, L.; Mingozzi, E. e Eklund, C. **Quality of Service Support in IEEE 802.16 Networks**. *Network IEEE*, p. 50–55, 2006.
- [da Conceição e Kon 2006] da Conceição, A. F. e Kon, F. **Desenvolvimento de aplicações adaptativas para redes IEEE 802.11**. *SBRC*, Instituto de Matemática e Estatística – Universidade de São Paulo (DCC - IME/USP), 2006.
- [Corradi 2005] Corradi, A. **Um Framework de Suporte a Requisitos Não-Funcionais para Serviços de Nível Alto**. Dissertação (Mestrado) — Universidade Federal Fluminense, 2005.
- [Garlan et al. 2004] Garlan, D.; Cheng, S.-W.; Huang, A.-C.; Schmerl, B. e Steenkiste, P. **Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure**. *Computer*, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 37, n. 10, p. 46–54, 2004.
- [Hernandez-Campos e Papadopouli 2005] Hernandez-Campos, F. e Papadopouli, M. **A Comparative Measurement Study of the Workload of Wireless Access Points in Campus Networks**. *16th International Symposium on Personal, Indoor and Mobile Radio Communications*, PIMRC IEEE, Los Alamitos, CA, USA, v. 3, p. 1776–1780, 2005.
- [Huang 2004] Huang, A.-C. **Building Self-Configuring Services Using Service-Specific Knowledge**. Tese (Doutorado) — Carnegie Mellon University, 2004.
- [Huebscher e McCann 2005] Huebscher, M. C. e McCann, J. A. **An adaptive middleware framework for context-aware applications**. *Personal Ubiquitous Comput.*, Springer-Verlag, London, UK, v. 10, n. 1, p. 12–20, 2005.
- [Huebscher e McCann 2004a] Huebscher, M. C. e McCann, J. A. **Adaptive middleware for context-aware applications in smart-homes**. In: *MPAC 04: Proceedings of the 2nd workshop on Middleware for pervasive and ad-hoc computing*, New York, NY, USA. ACM Press, p. 111–116, 2004.
- [Huebscher e McCann 2004b] Huebscher, M. C. e McCann, J. A. **Evaluation issues in autonomic computing**. In: *3rd international conference on grid and cooperative computing (GCC 2004)*, Wuhan, Peoples Republic of China, p. 597, 2004.
- [Huebscher e McCann 2005] Huebscher, M. C. e McCann, J. A. **Using real-time dependability in adaptive service selection**. *ICAS-ICNS*, IEEE Computer Society, Los Alamitos, CA, USA, v. 0, p. 76, 2005.
- [JBoss] JBoss. **JBoss - Servidor de Aplicações**. World Wide Web. Acessado em setembro/2006. Disponível em: <<http://labs.jboss.com/portal>>.
- [JESS] JESS. **JESS - The Rule Engine for the Java Platform**. World Wide Web. Acessado em setembro/2006. Disponível em: <<http://www.jessrules.com>>.
- [JNDI] JNDI. **JNDI - Java Naming and Directory Interface**. World Wide Web. Acessado em setembro/2006. Disponível em: <<http://java.sun.com/products/jndi>>.

- [Judd e Steenkiste 2002] Judd, G. e Steenkiste, P. **Fixing 802.11 access point selection**. *ACM SIGCOMM Computer Communication Review*, ACM Press, New York, NY, USA, v. 32, p. 31, 2002.
- [Judd e Steenkiste 2003] Judd, G. e Steenkiste, P. **Providing Contextual Information to Pervasive Computing Applications**. *percom*, IEEE Computer Society, Los Alamitos, CA, USA, v. 00, p. 133, 2003.
- [Karaul et al. 1998] Karaul, M.; Korilis, Y. A. e Orda, A. **A market-based architecture for management of geographically dispersed, replicated Web servers**. In: *ICE '98: Proceedings of the first international conference on Information and computation economies*, New York, NY, USA. ACM Press, p. 158–165, 1998.
- [Leal 2005] Leal, D. **Proposta de Adaptação Dinâmica utilizando JMX e JBoss**. Universidade Federal Fluminense — Instituto de Computação, 2005.
- [Loques et al. 2004] Loques, O.; Sztajnberg, A.; Cerqueira, R. C. e Ansaloni, S. **A contract-based approach to describe and deploy non-functional adaptations in software architectures**. *Journal of the Brazilian Computer Society*, v. 10, p. 5–18, 2004.
- [Molina-Jimenez et al. 2004] Molina-Jimenez, C.; Shrivastava, S.; Crowcroft, J. e Gevros, P. **TAPAS D10: QoS Monitoring of Service Level Agreements**. Newcastle University, 2004.
- [Narayanan et al. 2000] Narayanan, D.; Flinn, J. e Satyanarayanan, M. **Using history to improve mobile application adaptation**. *wmcsa*, IEEE Computer Society, Los Alamitos, CA, USA, v. 00, p. 31, 2000.
- [Norris et al. 2004] Norris, J.; Coleman, K.; Fox, A. e Candea, G. **OnCall: Defeating Spikes with a Free-Market Application Cluster**. In: *ICAC '04: Proceedings of the First International Conference on Autonomic Computing*, Washington, DC, USA. IEEE Computer Society, p. 198–205, 2004.
- [NS-2] NS-2. **NS-2 - Network Simulator**. World Wide Web. Acessado em setembro/2006. Disponível em: <<http://www.isi.edu/nsnam/ns>>.
- [Othman et al. 2004] Othman, O.; Balasubramanian, J. e Schmidt, D. C. **Performance Evaluation of an Adaptive Middleware Load Balancing and Monitoring Service**. *24th International Conference on Distributed Computing Systems*, ICDCS, 2004.
- [Petrucci e Loques 2007] Petrucci, V. e Loques, O. **Suporte a Adaptação baseado em Arquitetura usando Funções de Utilidade**. Universidade Federal Fluminense — Instituto de Computação, 2007.
- [Pinheiro et al. 2003] Pinheiro, E.; Bianchini, R.; Carrera, E. V. e Heath, T. **Load Balancing and Unbalancing for Power and Performance in Cluster-Based Systems**. In: *IEEE International Symposium on Performance Analysis of Systems and Software*, 2003.
- [Poladian et al. 2005] Poladian, V.; Sousa, J.; Padberg, F. e Shaw, M. **Anticipatory configuration of resource-aware applications**. *SIGSOFT Softw. Eng. Notes*, ACM Press, New York, NY, USA, v. 30, n. 4, p. 1–4, 2005.

- [Poladian et al. 2006] Poladian, V.; Sousa, J. P.; Garlan, D.; Schmerl, B. e Shaw, M. **Task-based Adaptation for Ubiquitous Computing**. *IEEE Transactions on Systems, Man and Cybernetics Part C, Applications and Reviews*, v. 36, n. 3, p. 328–340, 2006.
- [Ranganathan et al. 2005] Ranganathan, A.; Chetan, S.; Al-Muhtadi, J.; Campbell, R. H. e Mickunas, M. D. **Olympus: A High-Level Programming Model for Pervasive Computing Environments**. In: *PERCOM '05: Proceedings of the Third IEEE International Conference on Pervasive Computing and Communications*, Washington, DC, USA. IEEE Computer Society, p. 7–16, 2005.
- [Saltzer et al. 1984] Saltzer, J. H.; Reed, D. P. e Clark, D. D. **End-to-end arguments in system design**. *ACM Trans. Comput. Syst.*, ACM Press, New York, NY, USA, v. 2, n. 4, p. 277–288, 1984.
- [Santos et al. 2006] Santos, A. L. G.; Leal, D. e Loques, O. **Um Suporte para Adaptação Dinâmica de Arquiteturas Pervasivas**. *CLEI - Conferência Latinoamericana de Informática*, Santiago, Chile, v. 0, n. 0, p. 0, 2006.
- [SWANS] SWANS. **SWANS - Scalable Wireless Ad hoc Network Simulator**. World Wide Web. Acessado em novembro/2006. Disponível em: <<http://jist.ece.cornell.edu>>.
- [Zegura et al. 2000] Zegura, E. W.; Ammar, M. H.; Fei, Z. e Bhattacharjee, S. **Application-layer anycasting: a server selection architecture and use in a replicated Web service**. *IEEE/ACM Trans. Netw.*, IEEE Press, Piscataway, NJ, USA, v. 8, n. 4, p. 455–466, 2000.