Adaptação Dinâmica de Sistemas Distribuídos

Francisco José da Silva e Silva

Tese apresentada ao
Instituto de Matemática e Estatística da
Universidade de São Paulo
para obtenção do grau de Doutor
em Ciência da Computação.

Orientador: Prof. Dr. Markus Endler

Co-Orientador: Prof. Dr. Fabio Kon

São Paulo, Janeiro de 2003

Adaptação Dinâmica de Sistemas Distribuídos

Este exemplar corresponde à redação final da tese, devidamente corrigida e defendida por Francisco José da Silva e Silva e aprovada pela comissão julgadora.

São Paulo, 17 de março de 2003.

Banca Examinadora:

Prof. Dr. Markus Endler	PUC-Rio
Prof. Dr. Fabio Kon	IME-USP
Prof. Dr. Edmundo R. M. Madeira	UNICAMP
Prof. Dr. Renato Fontoura de G. Cerqueira	PUC-Rio
Prof. Dr. Fabio M. Costa	UFG

Aos meus avós

Francisco e Rita

João Benedito e Antônia

Agradecimentos

Este trabalho contou com a colaboração de muitas pessoas, às quais gostaria de registrar meu muito obrigado:

Aos meus orientadores, Markus e Fabio, pela dedicação na orientação deste trabalho e pelo apoio e compreensão que recebi nos momentos difíceis.

À Dilma por inúmeros motivos: pelo incentivo à minha vinda para o IME, pela orientação no meu primeiro ano de doutorado e ao apoio dado durante estes quatro anos.

Aos meus pais, Arteiro e Ozanira, por todo auxílio ao meu desenvolvimento tanto profissional quanto pessoal ao longo de todos estes anos.

À minha noiva Cláudia e meu filho Daniel, pela compreensão por todo este tempo que estivemos longe sem, no entanto, deixarmos de lembrar um do outro a cada dia.

Ao meu irmão e sua esposa, Paulo e Silvina, pelo companheirismo e afeto.

Aos amigos Robson, Ângela, Heitor e Maria do Carmo, pelos bons momentos de convivência.

A Alexandre, pela consultoria em UML e apoio nestes últimos meses de trabalho.

A todos os meus colegas de pós-graduação do IME, com os quais tive o prazer de conviver durante estes anos.

Ao Pinho e demais funcionários da CPG.

Aos amigos Anselmo e Auxiliadora pelo apoio e por me representarem junto à UFMA.

À Cristiane e Márcio Henrique por terem ajudado a encurtar a distância entre São Paulo e São Luís.

À FAPESP pelo apoio financeiro ao projeto SIDAM, através do qual foram adquiridas as máquinas utilizadas nos experimentos realizados.

À CAPES e à UFMA pelo suporte financeiro.

Resumo

Esta tese aborda o desenvolvimento de software adaptativo, capaz de modificar o seu comportamento dinamicamente em resposta a variações detectadas em seu ambiente de execução. A demanda por software adaptativo tem sido impulsionada por recentes avanços nas áreas de computação distribuída, móvel e ubíqua caracterizadas por um ambiente de execução com um alto grau de dinamismo.

O desenvolvimento de aplicações adaptativas é bastante complexo. Os desenvolvedores devem levar em consideração diversas questões além da implementação do comportamento funcional da aplicação. Estas questões incluem a escolha de quais elementos do ambiente de execução devem ser monitorados, como realizar este monitoramento, que ações de adaptação devem ser realizadas e quando elas devem ser executadas.

A complexidade inerente ao processo de desenvolvimento de aplicações distribuídas adaptativas é abordada através da proposição de um modelo para construção desta classe de aplicações. Tendo por base este modelo, foi implementado um arcabouço que simplifica o processo de desenvolvimento, disponibilizando ferramentas para a monitoração do ambiente de execução, detecção de mudanças no mesmo e reconfiguração dos componentes da aplicação.

Descreve-se o modelo e a implementação do arcabouço, bem como resultados experimentais obtidos através da construção de uma aplicação distribuída adaptativa para a disseminação de informações a clientes móveis. Os resultados dos testes reforçam a aplicabilidade do arcabouço, além de evidenciar a importância de levarmos em consideração a análise das interações entre os componentes da aplicação como forma de identificar a necessidade de reconfiguração da mesma.

Abstract

This thesis focus is the development of self-adaptive software, meaning software that can modify its own behavior at runtime to adapt to changes in its execution environment. The demand for self-adaptive software has continuously been growing due to recent advances in distributed, mobile, and ubiquitous computing, which are characterized as a computing environment with a high degree of dynamism.

Building self-adaptive applications is a complex and challenging task. Developers must consider several issues in addition to the implementation of the application's normal functionality. These issues include the selection of which environment elements should be monitored, how to perform the monitoring, which software adaptations should be carried out and when they should take place.

This work aims at reducing the complexity of building adaptive distributed applications by presenting a framework that provides a set of integrated tools for monitoring resource usage as well as application-specific interactions, detecting environmental changes, and reconfiguring distributed applications.

This thesis describes the framework model and its implementation. It presents experimental results obtained from the construction of an adaptive distributed information service for mobile users using the framework. The experimental results provides evidences that confirmed our belief that need for dynamic adaptation is best identified, if besides checking resource availability also the interactions among the application components are monitored, so as to identfy also changes in the pattern of communication and processing requirements.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

1.1 Motivação

O desenvolvimento de diversas tecnologias de computação e comunicação tem transformado o cotidiano de um número cada vez maior de pessoas, causando profundas mudanças na forma, quantidade, qualidade e velocidade com que trocamos informação. O contínuo avanço tecnológico e os esforços realizados em diversas pesquisas em curso apontam para ambientes de computação compostos por uma diversidade cada vez maior de dispositivos, tanto fixos quanto móveis, conectados através de diferentes tecnologias de rede.

A mobilidade tem se firmado como uma importante característica nesta nova infra-estrutura de computação. O usuário não estará mais restrito a pontos de acesso fixos e, através da utilização de diferentes dispositivos, será capaz de acessar recursos computacionais, serviços e informações, muitos destes sensíveis a sua localização no momento do acesso [HHC^{+}99, PS98, ML98].

Entre os avanços na direção destes novos ambientes de computação, destacam-se as recentes pesquisas sobre espaços ativos, como o projeto Gaia [RC00, MRN02]. Espaços físicos como escritórios, salas de aula, residências e hospitais serão cada vez mais equipados com dispositivos de computação, formando um ambiente que integra diversos equipamentos como computadores, projetores de imagem, câmeras de vídeo, quadros brancos eletrônicos, etc. O objetivo é proporcionar informação aos usuários deste espaço, ajudando-os a realizar, de maneira mais fácil, suas atividades ou até mesmo permitindo a realização de novas atividades através da interação de maneira uniforme com o ambiente de computação constante destes espaços físicos.

Uma característica intrínseca aos novos sistemas em rede é o seu dinamismo, decorrente de diversos fatores como a grande variação na disponibilidade de recursos e serviços, heterogeneidade de hardware e software, diversidade das tecnologias de rede empregadas e mobilidade dos usuários [FZ94, Sat96, JHE99].

Entre os requisitos impostos ao desenvolvimento de software para estes ambientes de computação cada vez mais dinâmicos destaca-se a necessidade de adaptação [dSeSE00]. Passa a ser imperativo que o software seja capaz de reconfigurar-se dinamicamente em resposta às constantes mudanças do ambiente de execução de forma a prover funcionalidade e desempenho adequados.

No entanto, o desenvolvimento de software adaptativo é bastante complexo, uma vez que para desenvolvê-lo deve-se prover tanto o código responsável pelo comportamento funcional da aplicação, quanto o código que permita ao software alterar seu comportamento em resposta a variações do ambiente. Esta complexidade é ainda maior se levarmos em consideração o desenvolvimento de aplicações adaptativas distribuídas, compostas por diversos componentes executados em diferentes nós de uma rede.

1.2 Contribuição desta Tese

Esta tese aborda a complexidade inerente ao processo de desenvolvimento de aplicações distribuídas adaptativas e propõe um modelo e mecanismos básicos para construção desta classe de aplicações. Tendo por base este modelo, foi implementado um arcabouço (*framework*), que disponibiliza uma série de mecanismos que permitem (1) a monitoração do ambiente no qual a aplicação é executada, (2) a análise dos dados coletados por esta monitoração, visando a identificação de mudanças no estado do ambiente, e (3) a reconfiguração dos componentes da aplicação distribuída em resposta às variações detectadas.

O desenvolvimento do arcabouço levou em consideração a experiência obtida através de diversas pesquisas relacionadas ao desenvolvimento de software adaptativo, algumas destas descritas no capítulo 8. Além disto, exploramos novos conceitos no desenvolvimento desta classe de aplicações, como a utilização da monitoração do padrão de interações entre os objetos que compõem a aplicação, levando-se em consideração aspectos semânticos da mesma, como forma de detectar a necessidade de reconfiguração de seus componentes. Um importante requisito de projeto utilizado na construção do arcabouço é que o mesmo deve prover os meios para reconfiguração de aplicações compostas por inúmeros componentes distribuídos em uma rede. O arcabouço proposto foi utilizado para a adição de mecanismos adaptativos a uma aplicação distribuída voltada para a disseminação de informações a clientes móveis, validando-se, assim, os mecanismos propostos em seu modelo.

As principais características do arcabouço são:

1. possui um modelo integrado que contempla a monitoração do ambiente de execução, a detecção de mudanças significativas deste ambiente e o suporte à reconfiguração segura das aplicações em resposta às variações detectadas;

2. pode ser utilizado no desenvolvimento de uma grande variedade de aplicações adaptativas distribuídas;

3. é flexível, podendo contemplar necessidades específicas das aplicações através da extensão de seus componentes;

4. possui mecanismos que permitem o gerenciamento da execução de seus componentes, o que inclui sua suspensão temporária, reinício ou interrupção de sua execução além da alteração de propriedades específicas;

5. permite a reconfiguração dinâmica da infra-estrutura de monitoração e detecção de eventos do ambiente de execução, através do qual pode-se incluir novos componentes de software responsáveis por monitorar ou detectar eventos de mudança no ambiente bem como substituir ou remover componentes já instanciados sem a necessidade de interrupção destes serviços;

6. seu modelo de monitoração permite não somente o acompanhamento do estado de utilização de recursos do ambiente (como taxas de uso da UCP e memória dos nós do ambiente, largura de banda para comunicação entre os mesmos, etc.) bem como a análise das interações realizadas entre os objetos que compõem a aplicação;

7. os componentes do arcabouço executados em processos independentes podem ser compartilhados por diversas aplicações adaptativas e foram projetados para permitir a execução

concorrente destas aplicações;

8. separação do código funcional da aplicação do código responsável pela adaptação;

9. estruturação do código responsável pelo tratamento dos eventos que notificam mudanças significativas no ambiente de execução em estratégias de adaptação [GHVJ94]. Mais de uma estratégia pode ser definida para o tratamento de um dado evento. Pode-se alterar a estratégia a ser utilizada face à ocorrência de um evento em tempo de execução;

10. suporte à reconfiguração dos componentes que compõem a aplicação através do uso de Configuradores de Componentes [KC00, Kon00].

1.3 Organização deste Documento

Esta tese está estruturada da seguinte forma: o capítulo 2 apresenta uma introdução ao desenvolvimento de aplicações adaptativas, identificando seus principais desafios e descrevendo o papel do arcabouço neste processo de desenvolvimento. Em seguida, apresenta-se uma visão geral do modelo utilizado na construção do arcabouço.

Os capítulos 3, 4 e 5 detalham os três aspectos do desenvolvimento de aplicações adaptativas apoiados pelo arcabouço: a monitoração do ambiente de execução, a detecção de variações significativas e a reconfiguração da aplicação de forma a acomodar o novo estado do ambiente.

O capítulo 6 apresenta como o arcabouço foi utilizado para adicionar adaptação dinâmica ao protótipo de uma aplicação para disseminação de informações de trânsito em um ambiente de computação móvel. Nesta aplicação, a geração, consulta e armazenamento dos dados são distribuídos e os clientes são móveis, o que torna os mecanismos de adaptação importantes para garantir um tempo de resposta adequado aos clientes.

O capítulo 7 apresenta os resultados dos experimentos que tiveram por objetivo medir os ganhos de desempenho obtidos através do uso de técnicas adaptativas na aplicação descrita no capítulo 6 bem como avaliar o impacto de ajustes em parâmetros da monitoração e detecção de mudanças no ambiente de execução.

Diversos trabalhos relacionados são descritos no capítulo 8. Realiza-se uma comparação entre estes trabalhos e o arcabouço descrito nesta tese.

Finalmente, o capítulo 9 apresenta as conclusões obtidas como resultado de nossa pesquisa bem como diversos trabalhos que podem vir a ser desenvolvidos a partir desta tese.

Capítulo 2

Modelo de Adaptação Proposto

Este capítulo apresenta uma visão geral do modelo proposto para construção de software adaptativo. O modelo é dividido em três partes: (1) monitoração do ambiente de execução, (2) análise dos dados coletados para detecção de mudanças significativas no estado do ambiente e (3) reconfiguração da(s) aplicação(ões) em resposta ao novo estado do ambiente. A partir do modelo, foi implementado um arcabouço (*framework*) [Joh97, FJS99]. Este arcabouço simplifica o desenvolvimento de aplicações distribuídas adaptativas.

2.1 Software Adaptativo

Podemos encontrar na literatura diversas definições para software adaptativo:

- Software adaptativo é aquele que avalia sua própria execução e altera o seu comportamento quando esta avaliação indicar que ele não está atingindo seus objetivos ou quando for possível atingir um melhor desempenho ou funcionalidade [DAR98];
- Software adaptativo é aquele que pode modificar seu comportamento em tempo de execução em resposta à mudanças no ambiente de execução e, assim, atingir seus objetivos [Men00];
- Um programa é dito adaptativo se for capaz de alterar automaticamente seu comportamento de acordo com seu contexto [ACM95];
- Software adaptativo é aquele que modifica seu próprio comportamento em resposta a variações

em seu ambiente de operação [OGT+99]. O ambiente de operação compreende qualquer coisa observável pelo software, como entrada e saída realizada pelo usuário, dispositivos de hardware e sensores externos e instrumentações do programa.

Considerando estas definições, conclui-se que softwares adaptativos devem possuir mecanismos que permitam coletar dados a respeito do estado de seu ambiente de execução, analisar estes dados visando identificar mudanças significativas, e alterar dinamicamente seu comportamento para atingir seus objetivos. A execução destes mecanismos deve ocorrer de forma contínua e concorrente à execução do código funcional da aplicação, conforme ilustrado na figura 2.1.

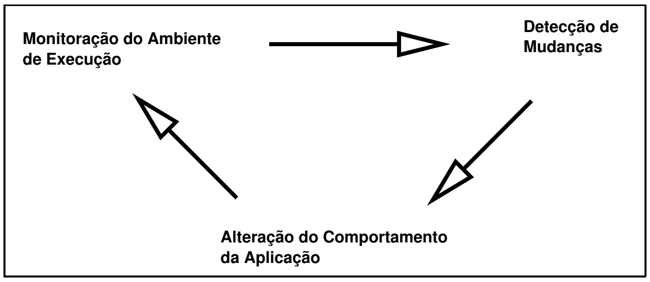

Figura 2.1: Ciclo de execução dos mecanismos adaptativos

2.2 Desenvolvimento de Software Adaptativo

O desenvolvimento de software adaptativo requer do desenvolvedor o cuidado com diversas questões [OGT+99] como:

1. Quais são os objetivos da adaptação? O sistema pode modificar-se, por exemplo, para melhorar o tempo de resposta, recuperar-se de uma falha, otimizar o uso de recursos ou incorporar novas funcionalidades;

2. O software permite a incorporação de novos comportamentos em tempo de execução ou não?

3. O software deve possuir autonomia sobre o processo de adaptação ou certas decisões devem ser tomadas através de intervenções humanas?

4. Qual é o custo operacional de monitorar o ambiente e realizar adaptações dinâmicas? Em que circunstâncias a adaptação apresenta ganho de custo/benefício?

5. Quais variáveis do ambiente de execução devem ser coletadas para que se possa tomar decisões sobre a adaptação?

6. Com que freqüência o sistema deve avaliar a possibilidade de realizar adaptações?

7. Qual é a complexidade de programação necessária para implementar os mecanismos de adaptação?

O desenvolvimento de software adaptativo é, portanto, bastante complexo uma vez que para desenvolvê-lo, deve-se prover tanto o código responsável pelo comportamento funcional da aplicação, como ainda o código que permita ao software acompanhar o estado de seu ambiente de execução e promover reconfigurações de seus componentes sempre que ocorrerem mudanças significativas que comprometam os objetivos do mesmo. Esta complexidade é ainda maior se levarmos em consideração o desenvolvimento de aplicações distribuídas, compostas por diversos componentes executando em diferentes nós de uma rede.

O arcabouço descrito nesta tese tem por objetivo simplificar a construção de aplicações distribuídas adaptativas, fornecendo os mecanismos para a implementação das decisões de projeto tomadas em resposta às questões acima levantadas. Por exemplo, uma vez definida a freqüência adequada para a verificação de uma eventual necessidade de realizar adaptações em uma dada aplicação, o arcabouço fornece os meios para realizar esta tarefa dentro da freqüência indicada podendo-se, ainda, alterá-la dinamicamente. Do mesmo modo, o arcabouço disponibiliza um conjunto de objetos para coletar informações do ambiente, permitindo ainda ao desenvolvedor estender este conjunto de acordo com as necessidades específicas da aplicação.

Além disto, o arcabouço apóia a construção de aplicações adaptativas em todas as etapas do ciclo de desenvolvimento, que inclui:

- **Análise:** o arcabouço descreve os principais componentes de um sistema adaptativo, suas funcionalidades e relacionamentos e disponibiliza um vocabulário próprio para tratar as questões relativas à adaptação;

- **Projeto:** o arcabouço decompõe o problema em um conjunto de componentes, cada um responsável por uma tarefa específica no processo de adaptação que envolve a monitoração do ambiente de execução, a detecção de variações significativas no mesmo e a reconfiguração da aplicação face ao novo estado do ambiente. O arcabouço define ainda as interfaces de comunicação entre estes componentes;

- **Implementação:** o arcabouço possui diversos componentes já implementados que podem ser combinados para a construção dos mecanismos de adaptação da aplicação sendo desenvolvida. Outros componentes podem ser implementados estendendo-se classes abstratas disponibilizadas pelo arcabouço;

- **Execução:** os componentes do arcabouço foram desenvolvidos levando-se em consideração a necessidade do gerenciamento de sua execução. Isto inclui o início e término de sua execução, a suspensão temporária e reinício dos mesmos, alteração de propriedades utilizadas em sua execução e, em alguns casos, a incorporação ou remoção em tempo de execução de novas funcionalidades.

2.3 Visão Geral

Os principais requisitos de projeto levados em consideração na construção do arcabouço foram:

1. O arcabouço deve ser genérico o suficiente para poder ser utilizado no desenvolvimento de uma grande variedade de aplicações distribuídas;

2. O arcabouço deve dar apoio ao desenvolvimento de aplicações distribuídas. Isto compreende um mecanismo que permita reconfigurações coordenadas dos componentes distribuídos que compõem a aplicação;

3. Os componentes do arcabouço devem ser flexíveis, permitindo ao usuário estender suas funcionalidades, de forma a satisfazer necessidades específicas da aplicação sendo desenvolvida;

4. Os serviços disponibilizados pelo arcabouço devem permitir a execução concorrente de aplicações adaptativas. Isto implica em permitir reconfigurações em tempo de execução de componentes do arcabouço, de forma a permitir a inclusão, exclusão e alteração de funcionalidades específicas das aplicações sendo executadas;

5. Os componentes do arcabouço devem permitir o gerenciamento de sua execução, permitindo a suspensão temporária, reinício ou interrupção de sua execução além da alteração de propriedades específicas;

6. O arcabouço deve promover uma clara separação entre o código funcional da aplicação e o código não-funcional responsável pelo comportamento adaptativo.

A figura 2.2 apresenta o diagrama de pacotes UML [BRJ99] do modelo utilizado para a implementação do arcabouço. O modelo é dividido em três pacotes principais. O primeiro, **Monitoração**, inclui os componentes responsáveis pelo monitoramento do ambiente de execução. Componentes do pacote **Detecção de Eventos e Notificação** devem ser utilizados para analisar periodicamente os dados coletados a fim de verificar a ocorrência de alterações significativas no estado do ambiente de execução. O arcabouço utiliza uma abordagem baseada em eventos para notificar as aplicações sobre estas variações. Uma vez notificada a ocorrência de eventos que sinalizam mudanças no estado do ambiente, a aplicação utiliza componentes disponíveis no pacote **Reconfiguração Dinâmica** para determinar as ações de reconfiguração que devem ser executadas para adaptar a aplicação a este novo estado. Componentes deste pacote ajudam também na execução destas ações.

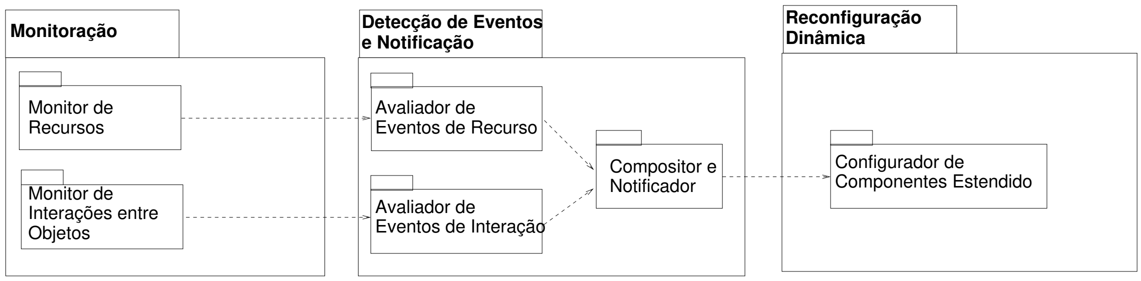

Figura 2.2: Diagrama de pacotes do modelo proposto

2.3.1 Monitoração

O objetivo do serviço de monitoração é coletar dados que permitam acompanhar o estado do ambiente de execução. A análise destes dados permite identificar mudanças no ambiente que indiquem a necessidade de reconfiguração da aplicação.

Um importante aspecto do ambiente de execução a ser monitorado é o estado dos recursos distribuídos existentes. Como exemplo destes recursos podemos citar a memória e UCP constantes de cada nó do ambiente e o estado da conexão entre eles (largura de banda disponível para comunicação, taxa de erros atual, latência de comunicação, etc.).

Um segundo aspecto a ser monitorado corresponde às interações realizadas entre os objetos que compõem a aplicação distribuída. Interceptadores, como definidos no padrão CORBA [OMG02], podem ser inseridos entre os objetos da aplicação, para coletar e armazenar informações relevantes sobre cada chamada de método. Pode-se armazenar, por exemplo, o momento da chamada, a identificação dos objetos requisitante e requisitado, os parâmetros enviados e os resultados devolvidos. A análise destes dados permite identificar alterações no padrão das requisições que exijam reconfigurações por parte da aplicação. Esta análise pode indicar, por exemplo, um aumento ou redução de demanda por um serviço ou um desbalanceamento na alocação de recursos distribuídos.

2.3.2 Detecção de Mudanças no Ambiente de Execução

O objetivo do pacote `Detecção de Eventos e Notificação` é detectar mudanças relevantes no estado do ambiente no qual a aplicação está sendo executada, tendo como base a análise dos dados coletados pelo pacote `Monitoração`. Este pacote é também responsável pela notificação da ocorrência de eventos que sinalizam estas mudanças.

O sub-pacote `Avaliador de Eventos de Recurso` recebe notificações a respeito de mudanças no estado de utilização dos recursos do ambiente que são utilizadas para a avaliação de expressões booleanas definidas pelo usuário. Estas expressões definem condições na utilização dos recursos que levam ao disparo de ações de reconfiguração por parte da aplicação. Um servidor de vídeo, por exemplo, pode diminuir a quantidade de quadros por segundo a ser apresentada ao cliente caso a largura de banda disponível para a comunicação com o mesmo fique abaixo de um dado valor.

O sub-pacote `Avaliador de Eventos de Interação` analisa os dados armazenados a respeito das interações entre os objetos que compõem a aplicação, através da execução periódica de funções definidas pelo usuário. Estas funções buscam por padrões de interação que indiquem a ocorrência de eventos relacionados com as interações entre os objetos da aplicação.

Eventos detectados por componentes destes dois sub-pacotes são enviados ao `Compositor e`

`Notificador`, que é responsável pela entrega de notificações da ocorrência de eventos aos objetos que tenham registrado interesse em recebê-las. Em particular, são notificados componentes do pacote `Reconfiguração Dinâmica` incorporados ao código da aplicação.

2.3.3 Reconfiguração da Aplicação

O objetivo do pacote `Reconfiguração Dinâmica` é executar as ações de reconfiguração da aplicação distribuída necessárias para acomodar um novo estado do ambiente de execução.

Componentes deste pacote estendem as funcionalidades de *Component Configurators* (Configuradores de Componentes) [KC00, Kon00], que definem uma arquitetura para reconfiguração de aplicações distribuídas. Nesta arquitetura, para certos componentes da aplicação são instanciados Configuradores que mantêm as dependências entre os diversos componentes que compõem a aplicação. Além disto, eles disponibilizam mecanismos através dos quais eventos de reconfiguração podem ser propagados.

Nós estendemos a arquitetura original dos Configuradores para torná-los cientes do estado do ambiente de execução e sugerimos uma estruturação para o código que define as ações de reconfiguração que devem ser executadas em resposta a mudanças no ambiente de execução.

2.4 Implementação do Modelo

Para a implementação do arcabouço, utilizamos a linguagem Java e o JacORB, uma implementação livre do padrão CORBA [OMG02, BVD01, Sof02]. Esta escolha foi feita levando-se em consideração diversas vantagens destas ferramentas como:

1. A portabilidade de Java para diferentes plataformas (hardware e sistemas operacionais), que permite a execução de componentes do arcabouço em um ambiente distribuído com máquinas heterogêneas e sem a necessidade de recompilação;

2. A independência da linguagem de programação oferecida por CORBA. Diversos componentes do modelo podem ser estendidos e novos componentes podem ser desenvolvidos, desde que respeitem as interfaces definidas no arcabouço. CORBA oferece a flexibilidade na escolha da linguagem na qual estes componentes são implementados, além de facilitar a integração

do arcabouço com outros sistemas existentes como o ReMoS, um sistema de monitoração do ambiente de rede, que será apresentado no capítulo 3;

3. Transparência na localização e ativação de objetos, que garante flexibilidade na escolha das máquinas nas quais os componentes do modelo serão executados;

4. Utilização de diversos serviços disponíveis em CORBA, como o serviço de nomes, notificação e segurança.

Nesta implementação do arcabouço utilizamos o banco de dados relacional MySQL para armazenar o estado de objetos persistentes definidos no modelo bem como os dados coletados a partir das interações entre os objetos que compõem as aplicações adaptativas escritas com o auxílio do mesmo.

Informações a respeito das ferramentas utilizadas na implementação do arcabouço, seu código fonte e documentação podem ser obtidas através do seguinte apontador:
`http://www.ime.usp.br/~sidam/software/AdaptationFramework`.

Capítulo 3

Monitoração

Este capítulo apresenta o serviço de monitoração do ambiente de execução. Fazem parte do arcabouço dois pacotes relativos a este serviço: monitoração de recursos do ambiente e monitoração das interações realizadas por objetos da aplicação. Este capítulo apresenta o projeto, implementação e exemplos de uso destes dois pacotes.

3.1 Monitoramento de Recursos

O pacote `Monitor de Recursos` tem por função disponibilizar meios para acompanhar o estado de utilização dos recursos disponíveis no ambiente. Estes recursos tipicamente compreendem componentes de hardware como memória, UCP, disco e enlaces de rede. No entanto, componentes de software podem também ser monitorados por objetos pertencentes a este pacote. Pode-se, por exemplo, monitorar o tempo médio de resposta que um servidor leva para atender requisições ou a quantidade de requisições por ele atendidas.

A monitoração de recursos é baseada no conceito que denominamos **Parâmetro de Monitoração**, seguindo a abordagem proposta em [SB97]. Cada Parâmetro de Monitoração representa um aspecto do recurso distribuído a ser monitorado. Por exemplo, dado o recurso UCP, pode-se monitorar seu percentual de uso ou a quantidade de memória real disponível. Dado um enlace da rede, pode-se medir a largura de banda disponível ou sua latência de comunicação.

Cada Parâmetro de Monitoração possui um conjunto de faixas (valores) de operação. A tabela 3.1 ilustra um possível conjunto de faixas que podem ser utilizadas para acompanhar o percentual de uso da UCP.

Faixa	**Valor Inicial**	**Valor Final**
1	0	5
2	5.01	10
3	10.01	20
4	20.01	50
5	50.01	75
6	75.01	85
7	85.01	95
8	95.01	100

Tabela 3.1: Faixas de operação: percentual de uso da UCP

O uso de diferentes faixas de operação permite uma maior flexibilidade na tomada de decisão para uma possível adaptação. Um servidor Web, por exemplo, poderia passar a enviar as figuras em preto e branco caso a largura de banda disponível para a comunicação com seu cliente se tornasse estreita e caso a largura de banda ficasse ainda menor, poderia passar a omití-las por completo. A quantidade de faixas de operação define a granularidade da monitoração a ser realizada.

A monitoração de recursos do ambiente é um componente fundamental na construção de uma grande variedade de aplicações adaptativas. Um servidor de vídeo, por exemplo, poderia ajustar a quantidade de quadros por segundo a ser apresentada ao cliente ou ajustar a resolução de cada imagem de acordo com o estado da largura de banda disponível para a conexão estabelecida entre eles. Um cliente poderia escolher dentre um conjunto de servidores equivalentes para qual deles enviar uma requisição, de acordo com a carga de trabalho monitorada em cada máquina servidora.

Estes exemplos ilustram que diferentes aplicações adaptativas podem ter interesse em monitorar diferentes parâmetros do ambiente. Como um dos objetivos do arcabouço é ser reutilizável, faz-se necessário que o conjunto de parâmetros monitoráveis possa ser estendido de acordo com as necessidades de cada aplicação.

Um componente do arcabouço, denominado **Repositório de Entidades**, armazena os parâmetros do ambiente que podem ser monitorados. Utilizando a interface deste componente, o usuário pode definir novos parâmetros. Cada parâmetro possui um identificador, uma descrição e as faixas de valores utilizadas em sua monitoração. Cada parâmetro está relacionado a um **Tipo de Entidade de Monitoração**. Por exemplo, os parâmetros percentual de uso da UCP e quantidade de memória real disponível referem-se a entidades de monitoração do tipo máquina (ou *host*). Os parâmetros largura de banda disponível e latência de comunicação referem-se a entidades do tipo enlace de rede. As instâncias de cada tipo de entidade são denominadas **Entidades de Monitoração**. Cada ambiente de execução possui seu próprio conjunto de entidades. O laboratório de Computação Paralela e Distribuída (LCPD) do IME-USP possui, por exemplo, as máquinas `Limeira, Campinas e IlhaBela`. A interface do `Repositório de Entidades` permite também definir os tipos e entidades de monitoração constantes do ambiente de execução onde as aplicações adaptativas serão executadas.

3.1.1 Objetos de Monitoração de Recursos

No arcabouço a implementação da monitoração de recursos do ambiente é baseada em componentes denominados **Objetos de Monitoração de Recursos - OMR**. Para cada Parâmetro de Monitoração, implementa-se um OMR que deve ser instanciado em cada máquina na qual se deseja acompanhar o estado do parâmetro monitorado. Periodicamente o OMR verifica, através de uma consulta local ao recurso correspondente, em qual faixa o parâmetro monitorado está operando. Sempre que houver uma variação na faixa de operação, o Objeto de Monitoração envia uma notificação ao componente responsável pela avaliação de eventos relacionados à disponibilidade de recursos. A adoção desta estratégia, baseada no modelo *push*, reduz a quantidade de mensagens de monitoração que trafega na rede pelos seguintes motivos:

1. O componente responsável pela avaliação de eventos de recurso não necessita consultar periodicamente os OMRs intanciados no ambiente (modelo *pull*);

2. A notificação de mudança no estado de um recurso é gerada somente quando houver uma variação na faixa de operação do parâmetro monitorado, indicando uma mudança significativa do ponto de vista do monitoramento. Mesmo para recursos cujo estado oscila com muita freqüência, esta forma de monitoramento causará uma quantidade reduzida de mensagens, caso as faixas de operação tenham sido definidas apropriadamente.

A figura 3.1 apresenta a interface disponibilizada pelo OMR. O método `parameter()` devolve uma referência ao parâmetro sendo monitorado, enquanto `me()` devolve uma referência para a entidade monitorada. O método `current_range()` devolve a faixa de operação atual. Portanto, apesar de operar no modelo *push*, o OMR permite que opcionalmente se utilize o modelo *pull*. `Frequency()` é utilizado para se obter a freqüência utilizada pelo OMR para avaliar o valor corrente do parâmetro monitorado.

```
interface Rmo {
  MonitoredEntities::Parameter  parameter ();
  MonitoredEntities::Entity     me ();
  unsigned long    frequency ();
  unsigned long    current_range ();

  void  suspend ();
  void  resume ();
  void  change_frequency (in unsigned long new_frequency);
  oneway void  shutdown ();
};
```

Figura 3.1: Interface: Objeto de Monitoração de Recursos

No projeto do Objeto de Monitoração de Recursos levou-se em consideração facilidades que permitem o gerenciamento de sua execução. O método `suspend()` permite que a monitoração seja temporariamente suspensa enquanto `resume()` retoma a monitoração. A freqüência com que localmente é verificado o valor de operação do parâmetro pode ser alterada através do método `change_frequency()`. O método `shutdown()` pode ser utilizado para interromper a execução do OMR.

3.1.2 Implementação

O diagrama de classes do OMR responsável pela monitoração do percentual de uso da UCP em uma máquina está representado na figura 3.2. Atualmente o arcabouço possui implementados objetos responsáveis pela monitoração da UCP e da quantidade de memória principal disponível. O arcabouço foi projetado para ser extensível, permitindo que facilmente se inclua novos OMRs. Das sete classes apresentadas na figura 3.2, cinco são comuns a qualquer Objeto de Monitoração e, portanto, não necessitam ser reescritas no desenvolvimento de um novo OMR.

As classes `CpuMonitor` e `RmoCpuImpl` são específicas do parâmetro percentual de uso da UCP. `CpuMonitor` contém o código responsável por verificar periodicamente a taxa de uso da UCP.

Praticamente todo o trabalho de desenvolvimento de um novo OMR consiste na escrita do monitor específico para o parâmetro. `RmoCpuImpl` possui um código muito simples, que cria um objeto `CpuMonitor` e inicia o processo de monitoração.

Todas as demais funcionalidades de um OMR estão implementadas nas demais classes. Elas são responsáveis pela interface de comunicação (figura 3.1), por suspender e reiniciar a monitoração, alterar sua freqüência e verificar a faixa de operação do parâmetro, notificando, sempre que esta sofrer alguma variação, o componente do pacote `Detecção de Eventos` responsável pela avaliação de eventos (descrito no capítulo 4) .

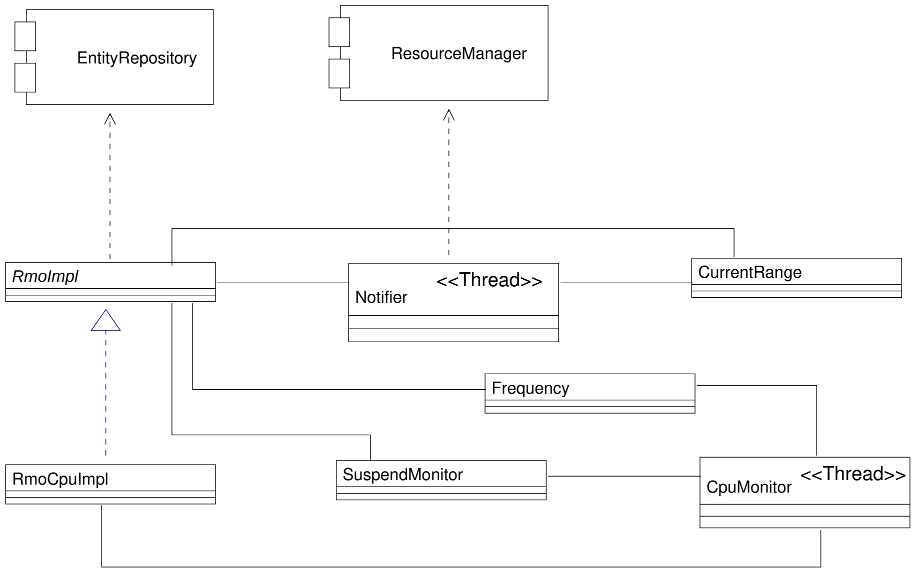

Figura 3.2: Diagrama de classes: Objeto de Monitoração de Recursos

3.2 Monitoramento de Interações Entre Objetos

O pacote `Monitor de Interações entre Objetos` fornece suporte para a monitoração das interações realizadas entre os objetos que compõem a aplicação distribuída. Esta monitoração é realizada através do uso de Interceptadores que são componentes de software inseridos entre os objetos cliente e servidor. Sempre que o cliente invocar um método em um objeto, a mensagem correspondente a esta invocação é interceptada e posteriormente reencaminhada para o objeto alvo. Através do uso de Interceptadores, pode-se extrair informações úteis de cada chamada de método.

Estas informações podem ser armazenadas em memória persistente para uma posterior análise (por exemplo, pelo pacote de detecção de eventos). Cada requisição pode ainda receber uma marca de tempo que permite identificar quando a requisição foi recebida pelo servidor. Pode-se ainda medir o tempo transcorrido entre a recepção da mensagem e o despacho da resposta ao cliente, o que corresponde à duração do atendimento à chamada do método pelo objeto servidor. Como exemplo das informações a respeito de cada interação que podem ser armazenadas temos:

- identificador do objeto para o qual a requisição é destinada;
- nome do método sendo invocado;
- endereço da máquina cliente;
- marca de tempo da realização da requisição;
- tempo de duração da execução do método no servidor.

Estas informações são de caráter geral, podendo ser úteis para diversas aplicações. No entanto, em outros casos pode ser interessante coletar informações específicas das aplicações. Por exemplo, no caso de um sistema de informação distribuído onde os clientes são móveis, pode ser útil identificar a região da cidade onde o cliente estava no momento da solicitação e qual dado foi alvo da consulta. A análise do padrão global de acesso realizado pelos clientes pode determinar uma melhor distribuição dos dados, de forma a minimizar o tempo de resposta médio das consultas realizadas.

Outro aspecto relevante sobre a monitoração das interações entre objetos é a necessidade de se determinar quais objetos da aplicação serão monitorados. Uma aplicação distribuída tipicamente é estruturada em um conjunto de componentes, cada um disponibilizando uma interface específica com um conjunto de métodos. Dependendo dos objetivos da adaptação, apenas as requisições enviadas a um subconjunto destes métodos devem ser monitoradas.

Conforme veremos no capítulo 8, que aborda vários de trabalhos relacionados à adaptação dinâmica, a utilização de dados sobre as interações entre objetos da aplicação tem sido pouco explorada e é um dos diferenciais do presente trabalho.

3.2.1 Implementação

A implementação da monitoração das interações entre os objetos da aplicação deve satisfazer os seguintes requisitos:

1. Transparência: a interceptação das chamadas de método das operações monitoradas deve ser o mais transparente possível para o código da aplicação;

2. Desempenho: o custo adicional das chamadas de método provocado pela interceptação deve ser pequeno, de forma a ser considerado desprezível para as aplicações que fizerem uso da mesma.

Considerando que a comunicação entre os componentes da aplicação se dê através de CORBA, a forma mais natural para prover o serviço de monitoração das interações entre objetos é através do uso de Interceptadores Portáteis (*Portable Interceptors*), que fazem parte da especificação CORBA [OMG02].

Interceptadores Portáteis são componentes de software que interceptam o fluxo normal de execução do ORB. Eles são inseridos no código do ORB, devendo ser registrados antes que seu método de inicialização (`init()`) retorne à aplicação a referência ao ORB a ser utilizada.

Existem dois tipos de Interceptadores: Interceptadores de Requisição e Interceptadores de IOR. Em nossa implementação, utilizamos apenas Interceptadores de Requisição para interceptar o fluxo de requisições e respostas propagadas entre objetos cliente e servidor através do ORB. No entanto, não é permitido aos Interceptadores de Requisição alterar os argumentos especificados pelo cliente na chamada de método.

Interceptadores de Requisição podem ser inseridos em diversos pontos da comunicação entre os objetos cliente e servidor, chamados de pontos de acesso. A figura 3.3 ilustra os pontos de acesso disponíveis. Existem dois tipos de Interceptadores de Requisição: aqueles inseridos no lado cliente da aplicação e aqueles inseridos no lado servidor.

Para cada ponto de acesso, é fornecido um objeto através do qual o Interceptador pode acessar informações a respeito da requisição. Informações comuns aos dois tipos de interceptadores (lado cliente e servidor) são disponibilizadas através de um objeto do tipo `RequestInfo`. Duas subclasses

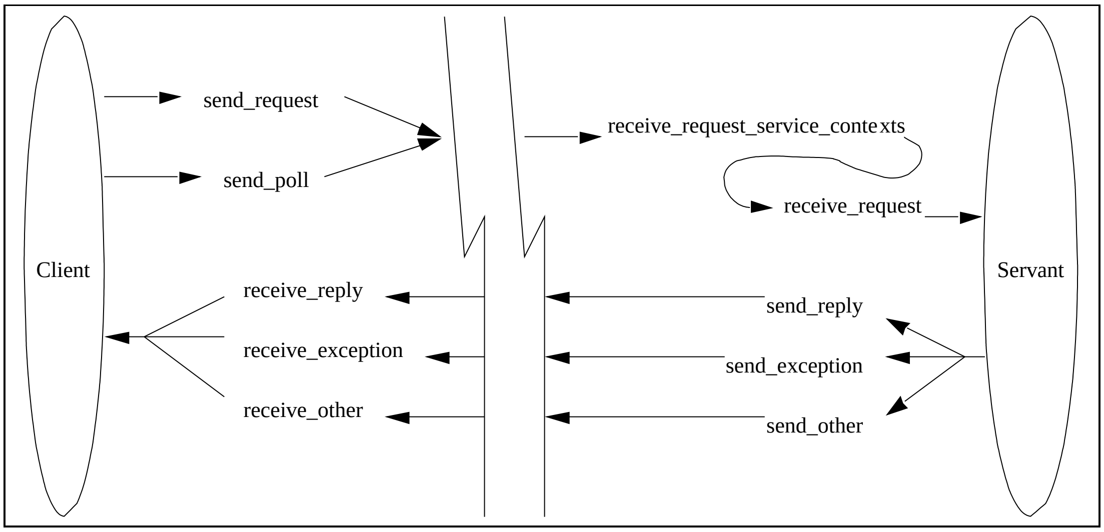

Figura 3.3: Pontos de acesso

de `RequestInfo`, `ClientRequestInfo` e `ServerRequestInfo`, fornecem informações específicas a interceptadores do lado cliente e interceptadores do lado servidor, respectivamente.

Entre os métodos disponíveis em `RequestInfo`, destacamos:

- `operation()`: retorna o nome do método sendo invocado;
- `arguments()`: retorna uma lista contendo os argumentos passados como parâmetros do método invocado;
- `result()`: devolve o resultado da invocação do método;

O método `target()`, constante da interface `ClientRequestInfo`, permite descobrir qual o objeto sendo chamado para realizar a operação.

Implementamos no arcabouço um mecanismo de monitoração das interações entre os objetos de uma aplicação distribuída utilizando Interceptadores de Requisição que executam do lado cliente. A figura 3.4 apresenta o diagrama de classes desta implementação.

A classe `ClientInitializer` implementa a interface `ORBInitializer`, responsável por registrar os Interceptadores durante a inicialização do ORB. Seu método `post_init()` cria uma instância da classe `ClientInterceptor`, que implementa o código do Interceptador. Em nossa implementação, o Interceptador utiliza um banco de dados relacional para armazenar os dados co-

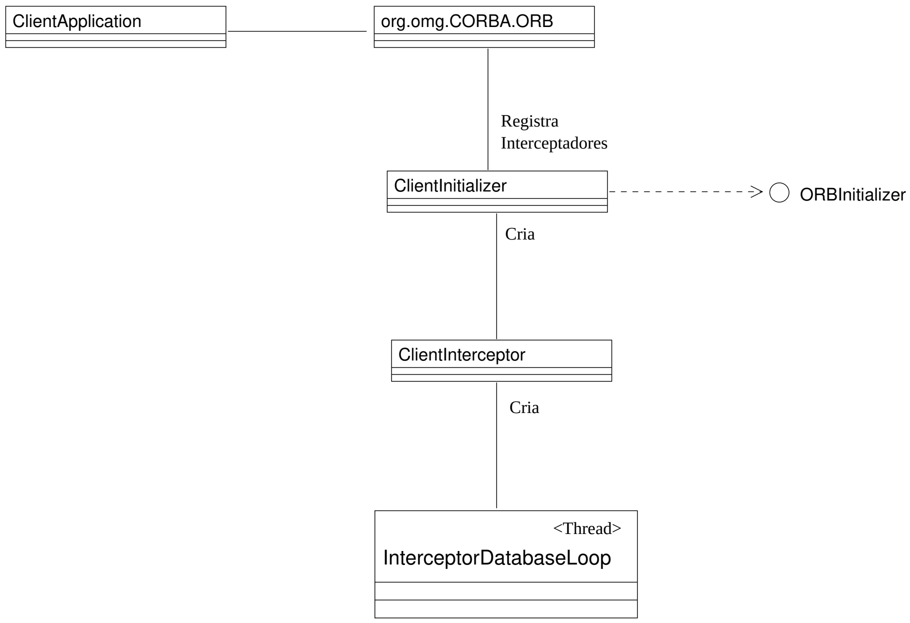

Figura 3.4: Diagrama de classes: monitoração de interações

letados durante a monitoração. Os argumentos de seu construtor incluem a máquina responsável por executar o servidor de banco de dados, o nome do banco de dados e a tabela onde os dados serão armazenados, bem como um usuário e sua senha com direitos de escrita nesta tabela. Além disto, deve-se fornecer também uma lista contendo os nomes dos métodos a serem monitorados. Somente as invocações realizadas a estes métodos terão seus dados armazenados pelo Interceptador. Ao final da execução do método `post_init()`, o Interceptador fará parte do ORB.

Durante sua inicialização, o Interceptador (classe `ClientInterceptor`) instancia um objeto da classe `InterceptorDatabaseLoop`, com quem compartilha uma estrutura de dados que armazena os dados relativos às chamadas de método realizadas pela aplicação. `ClientInterceptor` insere dados nesta estrutura sempre que a aplicação realizar uma chamada a um dos métodos monitorados. `InterceptorDatabaseLoop` é executado em uma *thread* independente que periodicamente remove os dados armazenados nesta estrutura e os salva no banco de dados. Desta forma, minimiza-se o custo da monitoração, evitando que a chamada a um método monitorado sofra um atraso devido ao tempo de acesso ao SGBD. A tabela 3.2 descreve quais são os dados armazenados.

Para métodos assíncronos (definidos em CORBA IDL como `oneway`), os campos indicados com um asterisco conterão valor nulo.

Dado	**Tipo**
Nome do método chamado	`String`
Marca de tempo da requisição	`Timestamp`
Marca de tempo do recebimento da resposta*	`Timestamp`
Nome da máquina que realizou a requisição	`String`
Endereço IP da máquina que realizou a requisição	`String`
IOR do objeto servidor	`String`
Tempo transcorrido entre a requisição e a resposta em milisegundos*	`Long`

Tabela 3.2: Dados monitorados pela interceptação de interações

Limitações e Solução Alternativa

A implementação dos mecanismos para a monitoração das interações entre os objetos da aplicação conforme descrito acima possui a vantagem de ser transparente ao código da aplicação. As únicas alterações necessárias são na inicialização do ORB, quando deve ser fornecida uma cadeia de caracteres contendo o nome da classe responsável por registrar o Interceptador, e ao término da aplicação, quando deve-se realizar uma chamada ao método `destroy()` do ORB para que a *thread* criada pelo Interceptador seja destruída.

No entanto, nos deparamos com uma limitação da especificação de Interceptadores Portáteis. No seu mapeamento para Java, o acesso aos argumentos passados na invocação do método (que poderiam ser obtidos através do método `arguments()`) não está disponível para aplicações que não utilizam invocação dinâmica (DSI/DII) [OMG02]. Ou seja, para as aplicações que utilizam *stubs* e *skeletons*, uma chamada ao método `arguments()` gerará uma exceção do tipo `NO_RESOURCES`. Esta limitação é decorrente do fato do ORB realizar o *marshaling* dos argumentos antes de executar os Interceptadores no lado cliente e realizar o *unmarshaling* somente após a execução dos Interceptadores no lado servidor. A operação de *unmarshaling* somente poderia ser realizada através da utilização de classes auxiliares, que não estão disponíveis no contexto no qual os Interceptadores são executados [Sun02a, Jac02].

Esta limitação restringe a flexibilidade do mecanismo de interceptação. Conforme descrito na seção 3.2, para algumas aplicações é importante armazenar informações específicas das interações entre seus objetos, que são enviadas como argumentos de entrada e saída na chamada de métodos. Para estes casos, implementamos uma solução alternativa cujo diagrama de classes é ilustrado na figura 3.5. Esta solução, no entanto, é menos transparente ao código da aplicação.

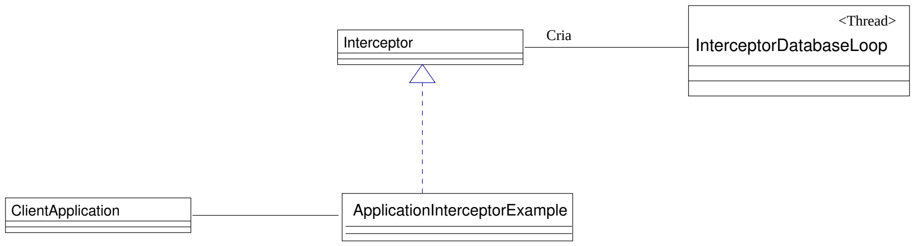

Figura 3.5: Diagrama de classes: solução alternativa para monitoração de interações

As classes `Interceptor` e `InterceptorDatabaseLoop` possuem um funcionamento semelhante às classes correspondentes na implementação que utiliza Interceptadores Portáteis. `Interceptor` insere os dados coletados pela monitoração em uma estrutura, enquanto `InterceptorDatabaseLoop` periodicamente os remove, armazenando-os em um banco de dados.

Os dados a serem armazenados variam para cada método, de acordo com o número e tipo dos argumentos. A implementação fornecida sugere ao usuário criar uma tabela para cada método monitorado. Os atributos desta tabela devem corresponder aos argumentos do método que se deseja salvar, bem a como outras informações gerais como as descritas na tabela 3.2.

O método `add_invocation()`, constante da interface de `Interceptor`, recebe como entrada um vetor de objetos Java e o nome da tabela na qual estes dados devem ser armazenados. O `Interceptor` os insere na estrutura de dados compartilhada com `InterceptorDatabaseLoop` que, por sua vez, os armazena na tabela fornecida tendo o cuidado de verificar se a quantidade e tipo dos objetos fornecidos correspondem à descrição da tabela.

Para minimizar as alterações necessárias ao código da aplicação, sugere-se que o usuário implemente uma sub-classe de `Interceptor` que contenha métodos com a mesma definição dos métodos a serem monitorados. Estes métodos devem armazenar os argumentos recebidos em um vetor de objetos e realizar a chamada ao método `add_invocation()`, fornecendo também o nome da tabela no qual eles devem ser armazenados. Outras informações podem ser inseridas neste vetor como, por exemplo, a marca de tempo no qual a invocação foi realizada. No código da aplicação é necessário incluir a cada chamada de um método monitorado uma nova chamada ao método correspondente na sub-classe de `Interceptor` implementada. A título de exemplo de uso, o arcabouço fornece a classe `ApplicationInterceptorExample`.

Custo da Monitoração de Interações entre Objetos

Com o objetivo de medir o custo que a monitoração das interações entre objetos impõe à aplicação, implementamos uma aplicação bastante simples composta por um servidor que disponibiliza apenas dois métodos: `returnInfo()`, que recebe como parâmetros de entrada duas cadeias de caracteres, devolvendo uma outra ao objeto chamador e `noReturn()`, definida como `oneway` e que, portanto, não devolve valor algum. Implementamos também um cliente deste servidor que realiza 500 chamadas a `returnInfo()` e 500 a `noReturn()`. Realizamos, então, as medições utilizando duas versões do cliente: (1) a que utiliza o Interceptador implementado utilizando-se Interceptadores Portáteis de CORBA e (2) a que utiliza o Interceptador alternativo baseado apenas em Java.

Utilizando a ferramenta *Optimizeit* [Bor], medimos o consumo de CPU gerado pela interceptação das chamadas realizadas pelos clientes aos métodos `returnInfo()` e `noReturn()`. Para o cliente que utilizou a versão do Interceptador baseado em CORBA, a monitoração foi responsável por um acréscimo de 12,90% no consumo de CPU necessário para executar as 1000 chamadas de método realizadas. Já a monitoração baseada apenas em Java implicou em um acréscimo de 4,53% no consumo de CPU. O impacto dos custos medidos é ainda menor se levarmos em consideração que o tempo total para execução de uma requisição inclui também o tempo gasto na comunicação, correspondente ao tempo de envio da requisição e o recebimento da resposta.

Os testes foram realizadas no LCPD, Laboratório de Computação Paralela e Distribuída do IME-USP. A máquina utilizada foi a Orlândia, cuja descrição de hardware e software básico encontra-se no apêndice A.

Podemos concluir, pelos resultados dos testes realizados, que o custo da monitoração de interações entre objetos da aplicação é pequeno, sendo praticamente desprezível para a grande maioria das aplicações.

Comunicação através de Java RMI

O arcabouço pode ser utilizado para implementar os mecanismos de adaptação de uma aplicação distribuída mesmo que os componentes responsáveis pelo seu comportamento funcional se comuniquem através de Java RMI [Sun02b, FL98]. Isto é possível já que boa parte dos componentes do arcabouço são executados fora do espaço de endereçamento dos componentes da aplicação, conforme descrito na seção 2.3. Além disto, o código dos componentes do arcabouço que executam no mesmo espaço de endereçamento dos componentes da aplicação é isolado do código responsável pelo comportamento funcional da mesma.

Como RMI não disponibiliza um serviço de Interceptadores como parte de sua arquitetura, o usuário do arcabouço pode optar por utilizar a solução alternativa descrita anteriormente. Uma outra possibilidade é a utilização de implementações independentes de Interceptadores para a arquitetura RMI, como o desenvolvido para o projeto Aroma [NMMS00] da Universidade da Califórnia em Santa Barbara. Esta possibilidade é identificada como um possível trabalho futuro, já que é necessário a modificação da solução proposta pelo projeto Aroma, que é voltada para a utilização de replicação como forma de obter alta disponibilidade e tolerância a falhas por parte da aplicação.

3.3 Monitoração da Infra-estrutura de Rede

Um recurso importante a ser monitorado em ambientes distribuídos é o estado da comunicação entre seus nós. Tipicamente, aplicações adaptativas distribuídas possuem interesse no estado de parâmetros do ambiente de rede como a largura de banda disponível para comunicação entre seus componentes, a taxa de erros que o meio apresenta e a latência de comunicação.

O desenvolvimento de um serviço de monitoramento da infra-estrutura de rede que atenda às necessidades de aplicações adaptativas apresenta diversos desafios:

- O tráfego gerado por diferentes aplicações pode possuir características bastante distintas. Assim, diferentes aplicações podem precisar acessar diferentes tipos de informação. Para aplicações que manipulam esporadicamente grandes quantidades de dados, a largura de banda disponível para a transmissão de uma grande rajada de dados pode ser mais importante do que a largura de banda média disponível. Já aplicações que transmitem dados de forma contínua, como aplicações de vídeo e áudio, possuem mais interesse nesta última;

- Para a maioria das aplicações é mais interessante ter uma previsão a respeito do tráfego da rede no futuro. Uma boa estimativa é difícil de ser obtida. Para que se possa obter uma estimativa razoavelmente precisa é necessário levar em consideração o conhecimento das aplicações em execução e o seu uso de recursos;

- Várias informações de interesse das aplicações são geradas por diferentes entidades. Algumas informações somente podem ser obtidas através de um banco de dados estático mantido por administradores, outras podem ser acessadas através de protocolos como SNMP e outras, como a latência de comunicação, devem ser medidas pelo serviço de monitoramento através da utilização de testes específicos;

- A forma pela qual as informações devem ser obtidas e quais informações estão disponíveis varia de acordo com a tecnologia de rede adotada. Algumas tecnologias de rede, como ATM, disponibilizam informações a respeito da largura de banda disponível como parte de seu gerenciamento de tráfego (controle de fluxo). Em outras tecnologias, como Ethernet, esta informação não está disponível de forma direta;

- O nível adequado de abstração das informações de rede apresentadas varia de acordo com a aplicação. Uma aplicação pode ter interesse em conhecer a topologia de toda a rede, juntamente com informações sobre roteamento e o estado de cada enlace. Outra aplicação pode apenas estar interessada no estado da comunicação entre dois pontos da rede, independentemente da quantidade de enlaces intermediários e das características de cada um destes.

Tantos desafios tornam o serviço de monitoramento de rede complexo o suficiente para ser objeto de estudo de pesquisas específicas. Dentro desta linha, destacamos o projeto ReMoS (*Resource Monitoring System for network-aware applications*) [DGK+01, DGL+97, LMG+99], desenvolvido na Universidade Carnegie Mellon. ReMoS foi projetado com o objetivo de disponibilizar informações de rede para aplicações através de uma interface uniforme. Esta interface poderia ser periodicamente acessada por Objetos de Monitoração de Recursos para obtenção do estado de parâmetros do ambiente de rede que se deseja monitorar. A integração entre o ReMos e o arcabouço descrito nesta tese é um dos trabalhos futuros identificados.

Capítulo 4

Detecção e Notificação de Eventos

Este capítulo descreve a infra-estrutura para detecção e notificação de eventos relativos a mudanças no ambiente de execução.

O arcabouço permite a definição de dois tipos de eventos. **Eventos de Recurso** são aqueles gerados a partir da análise dos dados enviados pelos Objetos de Monitoração de Recursos instanciados no ambiente. **Eventos de Interação** são resultantes da análise das interações realizadas pelos objetos da aplicação.

Este capítulo inicia-se com uma descrição dos requisitos levados em consideração no projeto do arcabouço. Em seguida, apresenta-se a descrição dos dois tipos de eventos permitidos e o projeto e implementação da infra-estrutura constantes no arcabouço para sua detecção. Comenta-se a respeito de eventos compostos, definidos a partir da composição de outros eventos, e como o arcabouço pode dar apoio para a utilização dos mesmos. Finalmente, é apresentado o mecanismo de notificação de eventos para as aplicações adaptativas.

4.1 Requisitos de Projeto

Um importante requisito de projeto é a necessidade do arcabouço levar em consideração as necessidades específicas das aplicações. Uma aplicação adaptativa pode ter interesse em um subconjunto próprio de eventos. Por exemplo, enquanto uma aplicação pode ter interesse em ser notificada quando uma máquina servidora estiver com um percentual de uso da UCP acima de 90%, outra aplicação pode possuir interesse em notificações a respeito de variações significativas na largura de banda disponível para a comunicação entre seus componentes instanciados em nós distintos. Portanto, o arcabouço deve permitir que o usuário estenda o conjunto de eventos que podem ser detectados, de forma a não restringir o conjunto de aplicações adaptativas que podem ser escritas com apoio do mesmo. Em nossa abordagem, o usuário pode fornecer definições de novos eventos. Da definição constam um identificador único e uma descrição do evento, além de uma série de outros parâmetros que variam de acordo com o tipo de evento sendo definido (Evento de Recurso ou de Interação), conforme veremos nas seções seguintes.

Um segundo aspecto levado em consideração no projeto do arcabouço é que o ambiente de execução distribuído pode executar várias aplicações adaptativas concorrentemente. Neste caso, o código necessário para a detecção de um evento deve poder ser carregado e removido dinamicamente do componente do arcabouço responsável pela detecção de eventos sem interferir na detecção de outros eventos. Desta forma, novas aplicações adaptativas podem ser adicionadas e removidas do ambiente. O arcabouço permite ainda a substituição em tempo de execução do código responsável pela a detecção de um dado evento. Esta flexibilidade é necessária quando o administrador da aplicação sentir a necessidade de alterar as condições que levam ao disparo do mesmo.

Finalmente, o suporte à detecção de eventos deve possuir uma interface para gerenciamento, através da qual seja possível carregar, remover e alterar o código relativo à detecção de um dado evento, iniciar sua detecção além de suspendê-la temporariamente e posteriormente reiniciá-la. Esta interface de gerenciamento deve ainda permitir a alteração de parâmetros específicos para os dois tipos de evento (de Recurso e eventos de Interação). Estes parâmetros são explicados nas seções seguintes.

4.2 Tipos de Eventos

Esta seção descreve os dois tipos de evento disponibilizados pelo arcabouço.

4.2.1 Eventos de Recurso

Eventos de Recurso estão relacionados a mudanças no estado de utilização dos recursos distribuídos. Conforme descrito na seção 3.1, estes recursos podem ser tanto componentes de hardware quanto de software e possuem seu estado monitorado por Objetos de Monitoração de Recursos - OMRs (seção

3.1.1). Objetos de Monitoração notificam um componente do pacote `Avaliador de Eventos de Recurso`, denominado Gerente de Recursos, sempre que houver uma variação na faixa de operação do parâmetro monitorado no nó do ambiente no qual o Objeto de Monitoração está instanciado. O Gerente de Recursos mantém o último valor fornecido por todos os OMRs. Desta forma, ele adquire uma visão global do estado de utilização dos recursos do ambiente sendo monitorado por OMRs.

O instante no qual um OMR detecta uma variação na faixa de operação (vide seção 3.2) de um parâmetro em uma entidade do ambiente difere do instante no qual o Gerente de Recursos recebe esta notificação, dado que a comunicação entre estes processos em um ambiente distribuído leva um determinado tempo. Além disto, mensagens de notificação podem chegar fora de ordem, ou seja, o Gerente de Recursos pode receber notificações em uma ordem diferente da qual elas foram geradas pelos OMRs, não havendo garantia de entrega FIFO, muito menos de ordem causal. Por esse motivo, toda mensagem gerada por um OMR recebe uma marca de tempo. Ao receber uma mensagem, o Gerente de Recursos verifica se a marca de tempo da mensagem recebida é menor do que a marca de tempo armazenada juntamente com o último valor recebido do OMR. Se for, a mensagem é antiga, sendo descartada sem ser processada.

A avaliação da ocorrência de um Evento de Recurso é baseada em uma expressão booleana fornecida pelo usuário como parte de sua definição. Esta expressão contém um ou mais Parâmetros de Monitoração, como por exemplo a expressão:

```
uso_ucp > 4 AND mem_disponível < 2
```

Esta expressão é avaliada como verdadeira quando, em uma máquina, o percentual de uso da UCP (`uso_ucp`) estiver acima da faixa 4 de operação e a quantidade de memória real disponível (`mem_disponivel`) estiver abaixo da faixa 2. A expressão pode conter uma quantidade arbitrária de parâmetros, desde que eles se refiram ao mesmo Tipo de Entidade (vide seção 3.1).

A figura 4.1 apresenta a interface IDL dos dois tipos de eventos definidos no arcabouço. Independentemente de seu tipo, a definição do evento deve conter um identificador (`eid`) e uma descrição (`description`). No caso de Eventos de Recurso, ela deve conter também a expressão booleana, o Tipo de Entidade a que seus parâmetros se referem e um tempo de duração (`duration_time`). Este tempo é utilizado para evitar que eventos sejam disparados devido a variações temporárias na utilização de um recurso, como por exemplo um pico na utilização da UCP. Tal situação ocorre, por exemplo, quando uma aplicação "pesada", como o Mozilla ou Open Office, for iniciada em uma máquina.

```
interface Event {

  string  eid();
  string  description();
};

interface ResourceEvent : Event {
  string                  expression ();
  unsigned long           duration_time ();
  MonitoredEntities::EntityType metype();
};

interface InteractionEvent : Event {
  unsigned long frequency();
  string        class_name();
};
```

Figura 4.1: Interface: Eventos de Recurso e Interação

4.2.2 Eventos de Interação

Eventos de Interação estão relacionados com a análise das interações entre os componentes que compõem a aplicação distribuída. Dados relativos a cada chamada de método entre os objetos monitorados são armazenadas em memória persistente, conforme descrito na seção 3.2. Estes dados são analisados periodicamente, procurando-se por padrões que indiquem situações anormais ou indesejadas.

A detecção de Eventos de Interação é feita através de **Avaliadores de Interações**. Cada avaliador é responsável por detectar um Evento de Interação específico. O arcabouço disponibiliza uma classe abstrata, denominada `Evaluator`, que implementa as funcionalidades básicas de um avaliador.

Para implementar o avaliador responsável por detectar um Evento de Interação específico de sua aplicação, o usuário deve implementar uma sub-classe de `Evaluator`. Esta classe deve sobrescrever dois métodos. O primeiro, `user_evaluation_code()`, deve conter o código responsável pela leitura e análise dos dados armazenados em memória persistente. Caso a análise dos dados identifique a ocorrência do evento, um objeto do tipo `EventNotification` deve ser devolvido. Este objeto, como seu nome sugere, deve conter os dados a respeito do evento ocorrido a serem transmitidos para os componentes da aplicação que tenham registrado interesse em receber notificações a respeito de ocorrências do evento. O segundo método a ser sobrescrito é denominado `cleanDatabase()`. Este método é responsável pela limpeza (exclusão) dos dados armazenados na memória persistente de forma a evitar que o espaço em disco utilizado cresça indefinidamente.

Cada avaliador é executado em uma *thread* independente. O código constante do método `user_evaluation_code()` é executado periodicamente, em uma freqüência definida pelo usuário na definição do Evento de Interação. O valor atribuído a esta freqüência é fundamental para maximizar os ganhos com o processo de adaptação, conforme ficará evidenciado nos exemplos apresentados no capítulo 7. Ao término de cada execução do método `user_evaluation_code()`, o método `cleanDatabase()` é chamado.

A figura 4.1 apresenta também a interface IDL utilizada para Eventos de Interação. Para definição destes eventos, o usuário deve fornecer além da identificação e descrição dos mesmos, a freqüência (`frequency`) na qual o método `user_evaluation_code()` deve ser executado e o caminho (`class_name`) no qual o arquivo contendo o código compilado da classe escrita pelo usuário pode ser encontrado.

4.2.3 Repositórios de Eventos

Eventos de Recurso são criados através do **Repositório de Eventos de Recurso**. Este componente faz parte do pacote `Avaliador de Eventos de Recurso` (figura 2.2) e disponibiliza um método para criação (definição) de novos eventos. Conforme visto na seção 4.2.1, o usuário define um evento de recurso informando uma identificação, descrição, expressão booleana, tempo de duração e o tipo de entidade de monitoração associada ao mesmo. De posse destes parâmetros, o repositório cria um objeto CORBA, devolvendo ao usuário sua referência (IOR). Objetos que definem eventos são persistentes. A implementação atual do arcabouço utiliza um SGBD para esta finalidade.

O repositório de eventos de recurso disponibiliza também um método que permite a exclusão de um evento e outro que fornece sua IOR dada a sua identificação.

De forma semelhante, consta do arcabouço o **Repositório de Eventos de Interação**, que faz parte do pacote `Avaliador de Eventos de Interação`. Os métodos disponibilizados são similares, diferindo apenas nos parâmetros utilizados para criação de novos eventos.

4.3 Detectando Eventos de Recurso

A detecção de Eventos de Recurso é realizada por um componente do arcabouço denominado **Gerente de Recursos**, cujo diagrama de classes é apresentado na figura 4.2.

O Gerente de Recursos recebe dos diversos OMRs instanciados no ambiente notificações a respeito de mudanças no estado de utilização dos recursos. Cada OMR, ao ser intanciado, se registra junto ao gerente de recursos, informando qual entidade do ambiente e parâmetro por ele monitorados. O Gerente de Recursos organiza uma tabela (implementada atualmente como uma tabela de espalhamento) que contém todas as entidades do ambiente que estão sofrendo alguma monitoração (`TBMonitoredEntities`) e qual o último valor informado para cada parâmetro sendo monitorado (`ParameterValue`).

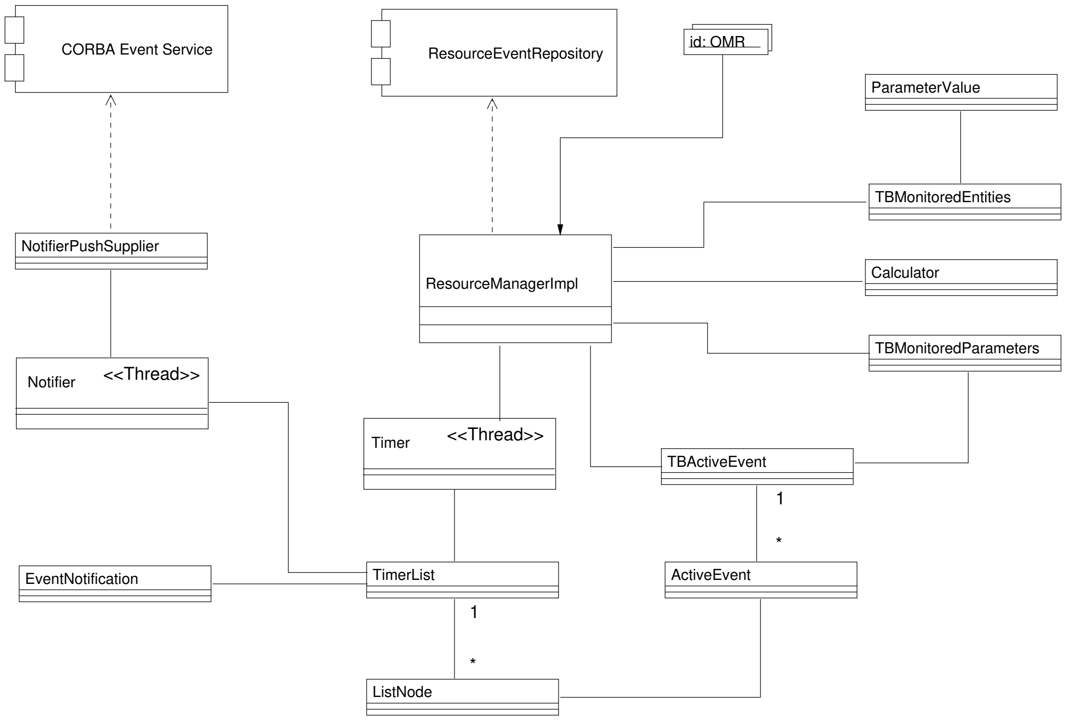

Figura 4.2: Diagrama de classes: Gerente de Recursos

A figura 4.3 apresenta a interface IDL do Gerente de Recursos. O método `rmo_register()` é utilizado pelo OMR ao se registrar junto ao Gerente de Recursos. Quando sua execução for finalizada, o OMR informa o Gerente de Recursos através do método `rmo_unregister()`, que remove de suas estruturas internas os dados relativos ao mesmo. Ao ter sua execução interrompida temporariamente (seção 3.1.1), o OMR realiza uma chamada ao método `rmo_suspend()`. Isto faz com que o Gerente de Recursos passe a não mais considerar como válido o último valor fornecido pelo OMR. `Rmo_resume()` é chamado pelo OMR quando seu monitoramento tiver sido retomado.

```
interface ResourceManager{
  void register    (in ResourceEvent re);
  void unregister  (in string eid) raises(NoSuchEvent, EventNotBeeingEvaluated);
  void suspend     (in string eid) raises(NoSuchEvent, EventNotBeeingEvaluated);
  void resume      (in string eid) raises(NoSuchEvent, EventNotBeeingEvaluated);

  void change_parameter (in string meid, in string pid,
                         in unsigned long new_range);

  void rmo_register   (in string strRmo, in string meid, in string pid);
  void rmo_unregister (in string meid, in string pid);
  void rmo_suspend    (in string meid, in string pid);
  void rmo_resume     (in string meid, in string pid);
  range_list list_me_parameter_range(in string meid);
};
```

Figura 4.3: Interface: Gerente de Recursos

A figura 4.4 apresenta o diagrama de sequência da avaliação de Eventos de Recurso. Vimos que o usuário define novos eventos através da interface do Repositório de Eventos de Recurso. Uma vez definido o evento, o usuário pode requisitar que ele passe a ser detectado realizando uma chamada ao método `register()`, constante da interface do Gerente de Recursos. Ao receber uma chamada para este método, o Gerente de Recursos obtém junto ao repositório a expressão booleana e o tempo de duração (seção 4.2.1) associados ao evento. Os eventos que estão sendo avaliados são chamados de **eventos ativos** (`ActiveEvent`) e são organizados em uma tabela (`TBActiveEvents`).

O OMR notifica mudanças na faixa de operação do parâmetro monitorado realizando uma chamada ao método `change_parameter()`. Ao receber esta chamada, o Gerente de Recursos identifica quais eventos ativos possuem em sua expressão booleana o parâmetro que sofreu alteração em sua faixa de operação. Cada evento ativo identificado terá sua expressão reavaliada para a entidade do ambiente monitorada pelo OMR, desde que ela possua valores para todos os parâmetros constantes da expressão.

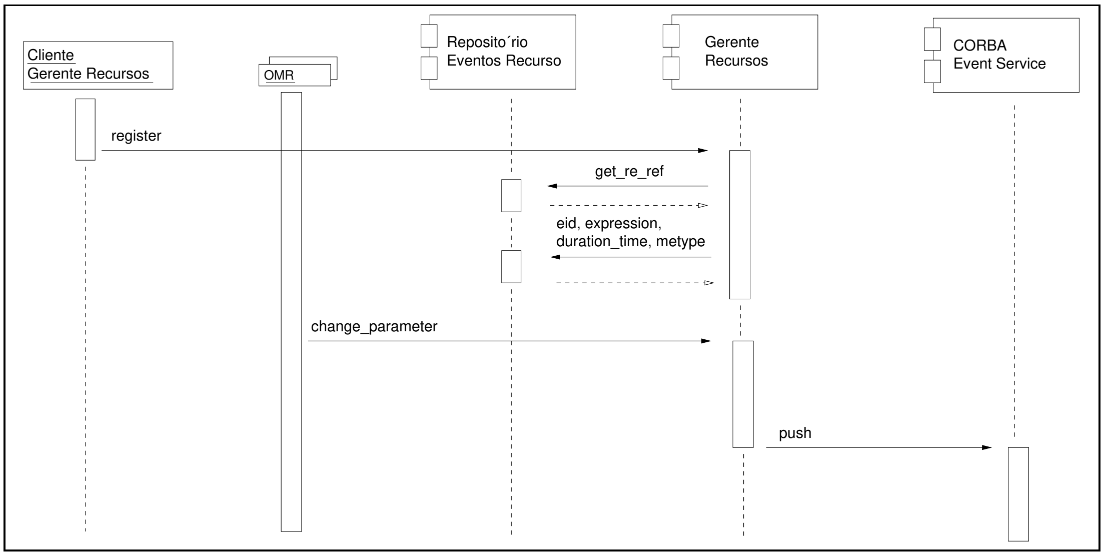

Figura 4.4: Diagrama de seqüência: avaliação de Eventos de Recurso

Sempre que uma expressão for avaliada como verdadeira, gera-se uma entrada em uma lista denominada `TimerList`. A posição deste novo elemento na lista é determinada pelo tempo de duração associado ao evento. Esta lista contém todas as ocorrências de Eventos de Recurso cujas expressões foram avaliadas como verdadeiro e é ordenada pelo instante de tempo no qual a notificação do evento deve ser disparada. Uma *thread*, denominada `Timer`, verifica a cada segundo se existem entradas nesta lista cujo tempo de notificação é menor ou igual ao tempo corrente. Para cada uma destas entradas, gera-se um objeto de notificação da ocorrência do evento (`EventNotification`) e, através de uma outra *thread*, `Notifier`, as notificações são enviadas ao canal de eventos responsável por entregá-las às aplicações. Da notificação do evento constam a identificação do evento, em que entidade do ambiente ele ocorreu e os valores de cada um dos parâmetros constantes de sua expressão.

Após uma expressão ter sido avaliada como verdadeira para uma dada entidade e antes da notificação ter sido gerada (ou seja, dentro do intervalo de tempo de duração definido pelo usuário), pode acontecer de o Gerente de Recursos receber uma notificação de mudança de um dos parâmetros da expressão que a tornem falsa. Neste caso, a entrada na lista de notificação relativa a este evento será removida e nenhuma notificação será gerada.

A interface do Gerente de Recursos possui ainda métodos para interromper temporariamente a detecção de um dado evento (`suspend()`) e reiniciá-la posteriormente (`resume()`). Através do método `unregister()`, pode-se remover da memória do Gerente de Recursos todos os dados relativos a um evento ativo. Para que este evento volte a ser detectado, deve-se chamar novamente o método `register()` quando, então, o Gerente de Recursos voltará a consultar o repositório para obter a definição do evento e reconstruirá as estruturas de dados necessárias à sua detecção. Isto permite ao usuário alterar a definição do evento e voltar a detectá-lo sem a necessidade de reiniciar o Gerente de Recursos.

4.4 Detectando Eventos de Interação

O **Gerente de Interações** é o componente do arcabouço responsável pela detecção de Eventos de Interação. Seu diagrama de classes é apresentado na figura 4.5 e sua interface na figura 4.6.

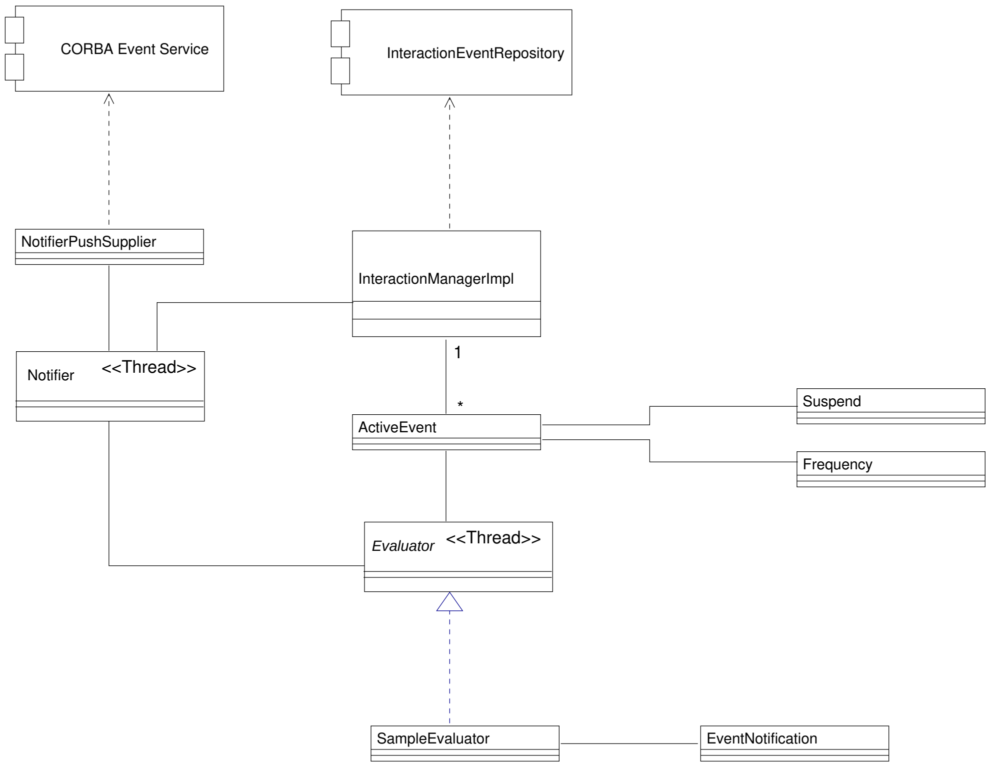

Figura 4.5: Diagrama de classes: Gerente de Interações

Através do método `register()`, o usuário solicita que se inicie a detecção de um evento. Assim como ocorre com o Gerente de Recursos, o Gerente de Interações consulta o Repositório de Eventos de Interação e obtém a definição do evento a ser detectado. Também de forma semelhante, um objeto `ActiveEvent` é instanciado, representando este evento. No entanto, este objeto é bastante diferente do seu correspondente na detecção de Eventos de Recurso.

Uma das variáveis de instância da classe `ActiveEvent` corresponde ao Avaliador de Interações, implementado pelo usuário e responsável pela detecção do mesmo. Da definição do Evento de Interação consta o caminho no qual pode ser encontrado o arquivo contendo o código compilado do avaliador. Este código é dinamicamente incorporado ao código do Gerente de Interações, utilizando-se para isto classes pertencentes ao pacote de reflexão computacional de Java [Gre02]. Todo avaliador herda da classe abstrata `Evaluator` sendo, portanto, executado em uma *thread* independente.

Cada evento ativo contém também um objeto da classe `Frequency`, responsável por armazenar a freqüência na qual o código do avaliador deve ser executado. Esta freqüência pode ser alterada dinamicamente através do método `change_frequency()`.

Cada evento ativo possui ainda um objeto da classe `Suspend`. Esta classe implementa um monitor [AE91], responsável por suspender temporariamente a execução do avaliador sempre que o usuário assim solicitar através de chamada ao método `suspend()`. Para retomar a execução do avaliador, o usuário deve utilizar o método `resume()`.

```
interface InteractionManager {
  void register    (in InteractionEvent ie) raises(EventClassNotFound);
  void unregister  (in string eid) raises(NoSuchEvent, EventNotBeeingEvaluated);
  void suspend     (in string eid) raises(NoSuchEvent, EventNotBeeingEvaluated);
  void resume      (in string eid) raises(NoSuchEvent, EventNotBeeingEvaluated);
  void change_frequency (in string eid, in unsigned long new_frequency)
    raises(NoSuchEvent, EventNotBeeingEvaluated);
};
```

Figura 4.6: Interface: Gerente de Interações

O método `unregister()` provoca a remoção de todos os objetos responsáveis pela detecção de um dado evento da memória do Gerente de Interações. Isto inclui a remoção de seu avaliador. De forma semelhante ao que ocorre no Gerente de Recursos, o método `register()` pode ser utilizado para iniciar novamente a detecção deste evento, permitindo assim ao usuário alterar a definição do evento e voltar a detectá-lo sem a necessidade de reiniciar o Gerente de Interações.

Um aspecto interessante da substituição do código do avaliador é que até a sua versão 1.3, reflexão em Java estava praticamente restrita à introspecção, isto é, à habilidade de verificar as estruturas de dados utilizadas no programa, como as classes [Chi00]. A alteração do comportamento da aplicação restringia-se à possibilidade de instanciação de uma classe, alteração do valor de um campo e à chamada de um método sem o conhecimento prévio destes componentes. Uma vez carregada na JVM uma classe, não era possível a substituição em tempo de execução de seu código. Isto passou a ser possível a partir da versão 1.4 através da tecnologia HotSwap [Sun02c, Dmi01, Dmi02].

4.5 Eventos Compostos

Eventos compostos são aqueles definidos como uma combinação de outros eventos. O suporte à definição de eventos compostos no arcabouço aumenta a capacidade de expressão de acontecimentos complexos que devam ser monitorados. Como exemplo, considere o caso no qual um cliente pode escolher um dentre vários servidores para processar sua requisição. A definição de um evento pode determinar o estado da conexão entre o cliente e cada um dos servidores. Um outro evento poderia determinar o estado de utilização das máquinas servidoras. Finalmente, um evento composto poderia ser definido como uma combinação de ambos, levando-se em consideração tanto o custo da comunicação quanto o custo da computação no processamento da requisição.

Atualmente o arcabouço não possui suporte à definição de eventos compostos. Ao definir um evento de recurso, o usuário pode fornecer uma expressão booleana composta de diversos Parâmetros de Monitoração. No entanto, não é possível a definição de eventos compostos que sejam disparados a partir da ocorrência de uma combinação de outros eventos.

Como trabalho futuro, pode-se incorporar o suporte a eventos compostos no arcabouço integrando-o ao EPS [Mor98]. O EPS (*Event Processing Service*) é um monitor de eventos capaz de processar eventos compostos genéricos. Ele utiliza uma estrutura de árvores agregadas [ME01], que aumenta a eficiência no processamento de eventos. Eventos compostos podem ser construídos utilizando-se os seguintes operadores de composição de eventos:

- **&**: este operador de eventos representa a conjunção ($E_1 \& E_2$) de dois tipos de evento E_1 e E_2. Uma instância deste tipo de evento será gerada quando tiverem sido geradas instâncias dos tipos de evento E_1 e E_2, independentemente da ordem de geração.

- $|$: este operador de eventos representa a disjunção $(E_1|E_2)$ de dois tipos de evento E_1 e E_2. Uma instância deste tipo de evento será gerada quando uma instância do tipo de evento E_1 ou uma instância do tipo de evento E_2 tiver sido gerada.

- ;: este operador de eventos representa a seqüência $(E_1; E_2)$ de dois tipos de evento E_1 e E_2. Uma instância deste tipo de evento será gerada quando tiverem sido geradas instâncias dos tipos de evento E_1 e E_2, nesta ordem. Além disso, essas instâncias não podem ser concorrentes.

- **TimesEq**: este operador de eventos representa n repetições de um mesmo tipo de evento E (notação: $TimesEq(E, n, atrib)$). Uma instância deste tipo de evento será gerada quando n instâncias do tipo de evento E tiverem sido geradas e todas elas tiverem o mesmo valor para o atributo $atrib$.

- **TimesDf**: este operador de eventos representa n repetições de um mesmo tipo de evento E, mas com valores diferentes para o atributo (notação: $TimesDf(E, n, atrib)$). Uma instância deste tipo de evento será gerada quando n instâncias do tipo de evento E tiverem sido geradas e não existir, entre as n instâncias de E , duas instâncias com o mesmo valor para o atributo $atrib$.

4.6 Notificação de Eventos

Para notificar as aplicações da ocorrência de eventos de mudança no ambiente de execução, o arcabouço utiliza o **serviço de eventos** definido pela especificação CORBA [HV99, BVD01, OMG02].

O serviço de eventos de CORBA provê um mecanismo através do qual notificações a respeito da ocorrência de eventos podem ser distribuídas a uma quantidade ilimitada de interessados, sem que o fornecedor da notificação do evento tenha que conhecer seus consumidores e sem a necessidade que ele realize diversas chamadas de notificação, uma para cada consumidor.

Um componente do serviço de eventos, o **canal de eventos**, recebe os dados relativos à ocorrência de eventos de um ou mais fornecedores e os entrega a um ou mais consumidores. O canal de eventos pode funcionar como um cliente do fornecedor, consultando periodicamente a ocorrência de eventos (modelo *pull*) ou pode disponibilizar uma interface através da qual o fornecedor informa ao mesmo a ocorrência de eventos (modelo *push*).

O arcabouço define um canal de eventos padrão utilizado para notificação da ocorrência de eventos de mudança do ambiente de execução. As aplicações adaptativas, interessadas em receber estas notificações, registram-se junto a este canal de eventos utilizando para isto uma classe abstrata constante do arcabouço, conforme será detalhado no capítulo 5. Este canal de eventos utiliza o modelo *push*, conforme ilustrado na figura 4.7.

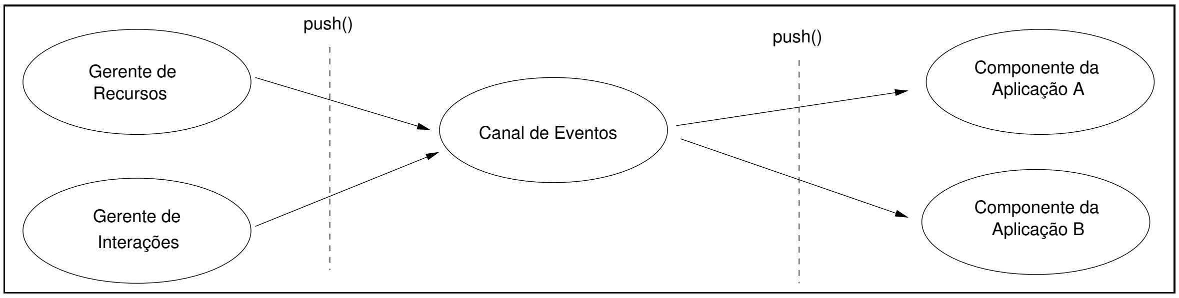

Figura 4.7: Notificação de eventos de mudança do ambiente de execução

Cada notificação de evento contém a identificação do evento ocorrido e um conjunto de atributos com seus respectivos valores. No caso de Eventos de Recurso, estes atributos correspondem aos parâmetros constantes da expressão booleana fornecida na sua definição e os valores correspondem às faixas de operação que tornaram a expressão verdadeira. No caso de Eventos de Interação, estes atributos variam de acordo com o código do avaliador fornecido pelo usuário. Estas informações constantes da notificação de eventos são encapsuladas em um objeto da classe `EventNotification`, fornecida pelo arcabouço.

Todas as notificações de eventos de mudança no ambiente de execução são enviadas através do canal de eventos e são encaminhadas a todos os consumidores. Isto pode gerar uma quantidade desnecessária de tráfego na rede, caso as aplicações não possuam interesse em receber notificações de todos os eventos. Para evitar este problema, pode-se opcionalmente utilizar o **serviço de notificação** de CORBA, uma extensão do serviço de eventos. Através dele é possível realizar uma filtragem na entrega de notificações de eventos, permitindo que as aplicações descrevam em quais eventos possuem interesse.

Capítulo 5

Reconfiguração Dinâmica

Mudanças no ambiente de execução geram notificações de ocorrência de eventos, enviadas através de canais de eventos CORBA. Estas notificações devem ser recebidas pela aplicação adaptativa, que deve prover um tratamento adequado para cada evento. Tipicamente isto requer uma reconfiguração dos objetos que compõem a aplicação.

Este capítulo apresenta o suporte fornecido pelo arcabouço para o recebimento das notificações de ocorrência de eventos e para a estruturação do código responsável pelo tratamento dos eventos de interesse da aplicação. Além disto, o arcabouço fornece suporte à reconfiguração segura dos componentes da aplicação através do uso de Configuradores de Componentes (*Component Configurators*) [KC00, Kon00].

5.1 Requisitos de Projeto

Os componentes do arcabouço apresentados até este ponto são executados em processos independentes dos processos que compõem a aplicação adaptativa. Estes componentes são responsáveis pela monitoração do ambiente de execução e pela detecção de eventos que sinalizam mudanças no mesmo. Ao contrário destes, os componentes do arcabouço descritos neste capítulo devem ser introduzidos no código da aplicação, sendo responsáveis pelo recebimento de notificações de eventos, pelo tratamento a ser dado para cada tipo de evento (ou seja, as ações de reconfiguração da aplicação necessárias para adaptá-la ao novo estado do ambiente) e, finalmente, por disponibilizar os mecanismos necessários para a execução das ações de reconfiguração identificadas.

Este código, que fornece o comportamento adaptativo da aplicação, deve estar separado do código funcional da mesma. Ou seja, a arquitetura da aplicação é dividida em duas camadas: uma sendo responsável pelo comportamento relativo ao domínio específico da aplicação (nível base) e a outra responsável por tratar mudanças no estado do ambiente de execução, que corresponde ao processamento reflexivo da aplicação (meta-nível)[Mae87].

5.2 Configuradores de Componentes

Para dar suporte à reconfiguração da aplicação, utilizamos no arcabouço o conceito de Configuradores de Componentes [KC00, Kon00] que definem um padrão arquitetural para reconfiguração de aplicações distribuídas.

Para cada componente da aplicação que possa estar envolvido ou ser afetado por uma reconfiguração, instancia-se um objeto denominado Configurador de Componente (ou simplesmente Configurador), conforme ilustrado na figura 5.1.

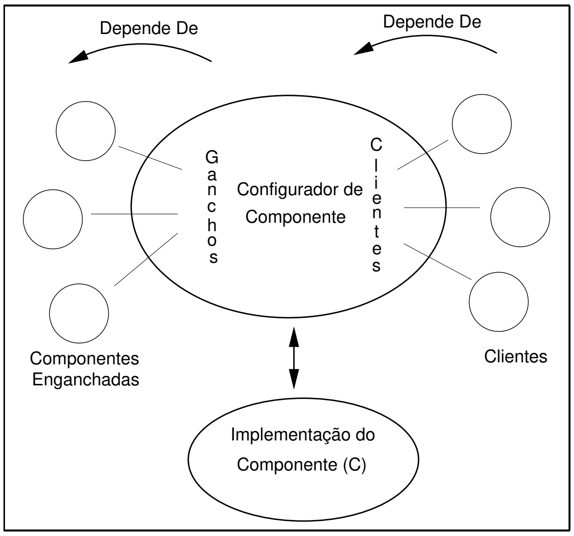

Figura 5.1: Configurador de Componente

O Configurador mantém referências que representam as dependências existentes entre o componente da aplicação que o instanciou e os demais componentes da aplicação. O Configurador de cada componente C possui uma lista de Configuradores relativos aos componentes dos quais C depende. Referências a estes componentes são armazenadas em **ganchos**. Outros componentes podem, por sua vez, depender de C. Estes são denominados clientes de C e também são mantidas referências aos seus respectivos Configuradores. Em geral, para definir que um componente C_1 depende de um componente C_2 deve-se realizar duas ações:

1. Associar uma referência ao Configurador de C_2 a um gancho no Configurador de C_1;

2. Associar uma referência ao Configurador de C_1 como cliente no Configurador de C_2.

Configuradores são também responsáveis por distribuir eventos entre componentes inter-dependentes. Sua implementação de referência define um conjunto básico de eventos que inclui a falha de um componente ou sua destruição, reconfiguração interna e substitiução de sua implementação. A tabela 5.1 contém a lista deste conjunto de eventos.

Evento	**Descrição**
STARTED	Componente inicializado
FINISHED	Componente teve execução concluída
SHUTDOWN	Componente teve execução interrompida
RECONFIGURED	Componente reconfigurado
REPLACED	Componente substituído
MIGRATED	Migração de componente
DELETED	Componente excluído
FAILED	Falha de componente
APP_SPECIFIC	Evento específico da aplicação
UNKNOWN	Evento desconhecido

Tabela 5.1: Eventos básicos pré-definidos no Configurador de Componente

Através da manutenção explicíta das dependências entre os componentes da aplicação, os Configuradores mantêm a consitência da configuração do sistema em tempo de execução e permitem que reconfigurações sejam realizadas mantendo-se esta consistência. Por exemplo, caso um componente falhe, os componentes que dele dependem são notificados e podem realizar uma finalização suave.

O Configurador é o local onde o programador deve inserir o código responsável pela reconfiguração da aplicação. Ele pode também implementar versões especializadas do Configurador, levando em conta características específicas da aplicação

Como exemplo de uso, considere uma aplicação multimídia para visualização de vídeos. Objetos de monitoração podem verificar o estado da largura de banda disponível para a comunicação com o servidor. Caso a largura de banda caia para uma determinada faixa de operação que impeça que a aplicação continue recebendo o vídeo com a qualidade corrente, o Configurador da aplicação pode ser notificado. Ele, então, gera uma mensagem ao servidor solicitando que a quantidade de quadros apresentada por segundo seja diminuída. Quando o estado de utilização da rede voltar a uma situação confortável, um novo evento pode ser gerado e o Configurador pode solicitar ao servidor que volte a apresentar os quadros na freqüência original.

5.2.1 Implementação

O pacote `Reconfiguração Dinâmica` possui uma versão do Configurador que denominamos `CorbaComponentConfigurator`. Nesta implementação, utilizamos CORBA para realizar a comunicação entre os Configuradores. Portanto, os componentes da aplicação podem estar distribuídos em diversos nós do ambiente de execução. As figuras 5.2 e 5.3 apresentam a interface IDL do `CorbaComponentConfigurator`.

```
enum EventType {STARTED, FINISHED, SHUTDOWN, RECONFIGURED, REPLACED, MIGRATED,
                DELETED, FAILED, APP_SPECIFIC, UNKNOWN};

struct DistributedEvent
{
  EventType type;
  string description;
};

interface CorbaComponentConfigurator; // forward declaration

struct DistributedDependencyAttributes
{
  string name;
  any value;
};

typedef sequence<DistributedDependencyAttributes> DistributedAttributeList;

struct DistributedDependencySpecification
{
  string name;
  CorbaComponentConfigurator componentConfig;
  DistributedAttributeList attributes;
};

typedef sequence<DistributedDependencySpecification> DistributedDependencyList;
```

Figura 5.2: Interface: Corba Component Configurator (A)

Os métodos `addHook()` e `deleteHook()` são utilizados para criar um gancho e para excluí-lo, respectivamente. Cada dependência gancho é identificada por uma cadeia de caracteres denominada `hookName`.

`hook()` é utilizado para especificar o componente da aplicação do qual este componente depende, enquanto `unHook()` quebra esta dependência.

Os métodos `registerClient()` e `unregisterClient()` são similares a `hook()` e `unHook()`, especificando, no entanto, componentes que dependem deste componente (clientes).

Pode-se especificar tanto para dependências gancho quanto para dependências cliente uma lista de atributos (`DistributedDependencyAttributes`). Esta lista é composta por pares contendo o nome do atributo e seu valor.

O método `eventOnHookedComponent()` anuncia a ocorrência de um evento em um componente da aplicação do qual este componente depende. `eventOnClient()` é similar, tendo o evento sido gerado por um componente cliente.

`name()` devolve uma cadeia de caracteres contendo o nome atribuído a este Configurador no momento de sua inicialização enquanto `info()` devolve sua descrição.

O método `listHooks()` devolve uma lista das dependências gancho registradas enquanto `list-Clients()` devolve uma lista de dependências cliente. `printHooks()` e `printClients()` devolvem uma cadeia de caracteres contendo os nomes dos componentes aos quais as dependências gancho e clientes referem-se, respectivamente.

Os atributos de uma dependência, quer seja gancho, quer seja cliente, podem ser obtidos através do método `getAttributes()`.

Ao ser inicializado, o Configurador recebe um nome e referências Java e CORBA (IOR) relativas ao componente da aplicação ao qual ele se refere. Ele utiliza a referência Java caso necessite chamar algum método deste componente. O método `implementation()` devolve a IOR ao qual o Configurador se refere.

`NotifyHooks()` e `notifyClients()` podem ser utilizados para gerar notificações de eventos aos componentes enganchados e clientes, respectivamente.

```
interface CorbaComponentConfigurator
{
  exception invalidArgument{};
  exception ElementExists{};
  exception NotFound{};
  exception HookBusy{};
  exception HookVacant{};
  exception WrongKey{};

  void destroyCorbaComponentConfigurator ();

  void addHook (in DistributedDependencySpecification hookSpecification)
    raises (ElementExists);

  void deleteHook (in string hookName) raises (NotFound);


  void hook (in DistributedDependencySpecification hookSpecification)
    raises (NotFound, HookBusy);

  void unhook (in string hookName) raises (HookVacant, NotFound);

  void registerClient (in DistributedDependencySpecification clientSpecification)
    raises (ElementExists);

  void unregisterClient (in CorbaComponentConfigurator client) raises (NotFound);

  void eventOnHookedComponent (in CorbaComponentConfigurator hookedComponent,
                               in DistributedEvent e, in unsigned long timeToLive);

  void eventOnClient (in CorbaComponentConfigurator client,
                      in DistributedEvent e, in unsigned long timeToLive);

  string name ();

  void setInfo (in string s);

  string info ();

  DistributedDependencyList listHooks ();

  CorbaComponentConfigurator getHookedComponent(in string hookName)
    raises (NotFound);

  unsigned long numberOfClients ();

  DistributedDependencyList listClients ();

  Object implementation ();

  string printHooks ();

  string printClients ();

  DistributedAttributeList getAttributes(in DistributedDependencySpecification
                                         hookSpecification)
    raises (NotFound);

  void notifyClients (in DistributedEvent e);

  void notifyHooks (in DistributedEvent e);
};
```

Figura 5.3: Interface: Corba Component Configurator (B)

5.3 Recebendo Notificações de Mudanças no Ambiente

Conforme descrito no capítulo 4, o arcabouço fornece suporte para a detecção de mudanças no ambiente de execução decorrentes de variações na disponibilidade dos recursos (seção 4.3) e variações no padrão das interações ocorridas entre os componentes da aplicação (seção 4.4). Mudanças no ambiente são notificadas através de canais de eventos CORBA. Para receber estas notificações, a aplicação deve se registrar junto ao canal de eventos pré-definido pelo arcabouço (seção 4.6).

O pacote `DynamicConfiguration` disponibiliza a classe abstrata `EnvironmentAware` que se registra junto ao canal de eventos através do qual notificações de eventos de mudança do ambiente são recebidas. O programador deve implementar uma classe concreta que herda de `EnvironmentAware`, sobrescrevendo seu método `push()`. Este método recebe do canal de eventos objetos da classe `EventNotification` (4.6). O método escrito pelo usuário deve verificar qual o evento ocorrido, através da leitura do atributo `eid` constante da notificação do evento e que informa a identificação do mesmo. De acordo com o tipo de evento sendo notificado, o código do usuário deve realizar uma chamada ao método que implementa a estratégia responsável por adaptar a aplicação ao novo estado do ambiente. As estratégias de adaptação são implementadas em classes distintas, estruturadas de acordo com o padrão *Strategy* [GHVJ94], conforme veremos a seguir.

A figura 5.4 ilustra a relação entre as classes `EnvironmentAware`, `CorbaComponentConfigurator` e o componente da aplicação escrito pelo programador. `EnvironmentAware` e `CorbaComponentConfigurator` são executadas em *threads* independentes.

5.4 Tratando Eventos de Mudança no Ambiente

A classe concreta derivada de `EnvironmentAware` é responsável pelo tratamento dos eventos recebidos. O arcabouço disponibiliza uma classe exemplo, denominada `EnvironmentAwareExample`, que ilustra a estruturação sugerida para este código. Esta estruturação segue o padrão *Strategy*, definido em [GHVJ94]. Para cada evento a ser tratado pela aplicação, define-se uma ou mais estratégias de adaptação.

A estratégia de adaptação contém o código que decide quais ações de reconfiguração devem ser executadas para adaptar a aplicação à mudança no ambiente de execução notificada. No caso

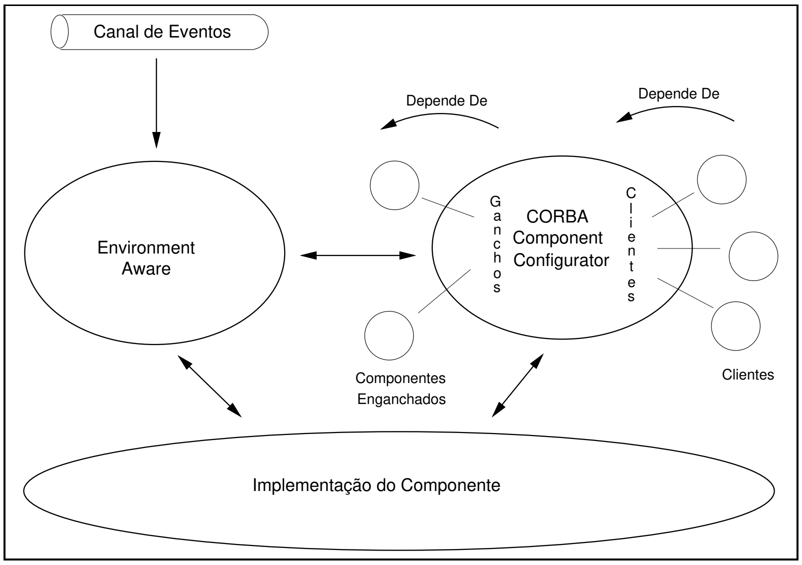

Figura 5.4: Configurador Estendido

do exemplo da aplicação de vídeo apresentada na seção 5.2, este código verificaria em qual faixa de operação a largura de banda para comunicação com o servidor está operando. Se a largura de banda disponível estiver baixa, o método `notifyHooks()` do Configurador é chamado para solicitar ao servidor que diminua a freqüência de apresentação dos quadros que compõem o vídeo. Neste caso, o servidor deve ter sido registrado no componente responsável pela apresentação do vídeo através de uma dependência gancho.

Eventualmente o programador pode querer implementar mais de uma estratégia para tratar um dado evento. Considere uma aplicação que possui uma lista de servidores equivalentes capazes de atender suas requisições. Caso a aplicação receba um evento notificando que o servidor em uso está sobrecarregado, ela pode utilizar diferentes estratégias para a escolha do novo servidor a ser utilizado como, por exemplo:

1. Consultar todos os servidores candidatos e escolher aquele que tenha menor carga de trabalho;

2. Escolher o primeiro servidor consultado cuja carga de trabalho esteja abaixo de um determinado valor.

Além disto, em alguns casos pode ser útil alterar dinamicamente a estratégia de adaptação a ser utilizada no tratamento de um evento. No exemplo da aplicação de vídeo, por exemplo, dependendo do vídeo sendo apresentado, o usuário poderia preferir visualizá-lo com uma menor resolução a diminuir a quantidade de quadros apresentada por segundo, em resposta a uma diminuição na largura de banda disponível para comunicação com o servidor.

5.5 Implementação

A figura 5.5 apresenta o diagrama de classes do pacote `Reconfiguração Dinâmica`.

Para instanciar o Configurador de um componente de sua aplicação, o usuário deve utilizar uma instância da classe `CorbaComponentConfiguratorFactory`.

Para receber notificações de eventos de mudança do ambiente de execução, deve-se instanciar uma classe que herde de `EnvironmentAware`. Conforme dito anteriormente, esta classe é também responsável por executar o código responsável pela identificação das ações de reconfiguração que devem ser tomadas face à ocorrência de um evento e utiliza o padrão *Strategy* para a estruturação deste código. O arcabouço fornece a classe `EnvironmentAwareSample` para exemplificar o recebimento de eventos. Esta classe, ao receber uma notificação, verifica se ela é uma instância de um evento cujo identificador é `eventA`. O arbouço possui duas estratégias para o tratamento deste evento, implementadas nas classes `EventAStrategy1` e `EventAStrategy2`. Para exemplificar a alteração em tempo de execução da estratégia utilizada, alterna-se entre as duas estratégias implementadas sempre que uma notificação de `eventA` é recebida.

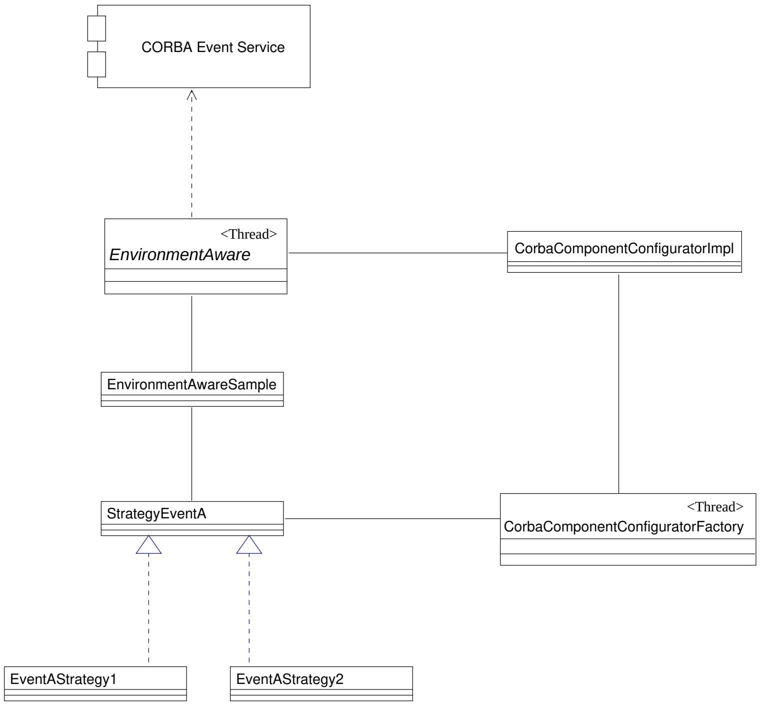

Figura 5.5: Diagrama de classes: reconfiguração dinâmica

Capítulo 6

Validação do Modelo

Como forma de validação da nossa abordagem para o desenvolvimento de aplicações adaptativas, utilizamos o arcabouço para adicionar adaptação dinâmica a uma aplicação distribuída. A aplicação utilizada é o protótipo de um sistema de disseminação de informações de trânsito para uma metrópole. Este protótipo foi desenvolvido dentro do contexto do projeto SIDAM (`http://www.ime.usp.br/~sidam`). Nele, a geração, armazenamento e consulta de dados sobre o tráfego em várias regiões são distribuídos e as aplicações cliente executam em máquinas móveis. Em um ambiente tão dinâmico, a utilização de técnicas adaptativas é um fator essencial para a manutenção dos objetivos de prover uma alta disponibilidade do serviço, balanceamento adequado de carga entre os servidores e minimizar o tempo de resposta aos clientes.

Este capítulo descreve a aplicação utilizada e como empregamos o arcabouço para promover reconfigurações em tempo de execução de seus componentes em resposta a variações detectadas no ambiente de execução. Ele inicia com uma descrição do projeto SIDAM e do sistema de disseminação de informações de trânsito utilizado como aplicação de referência. Descrevem-se as necessidades de adaptação identificadas para esta aplicação e como elas foram incorporadas utilizando-se o arcabouço.

6.1 O Projeto SIDAM

O objetivo do projeto SIDAM [EdSeS$^+$00, dSGE98] foi o de estudar os problemas relacionados ao projeto e implementação de sistemas de informação descentralizados utilizados tanto por usuários móveis quanto fixos. Objetivou-se o desenvolvimento de novos algoritmos, teorias e métodos relativos ao tema. A pesquisa abrangeu diversas áreas como sistemas distribuídos, análise de desempenho, tolerância a falhas, arquitetura de software, sistemas de banco de dados e representação de conhecimento. Como aplicação de referência, o SIDAM utilizou um sistema *on-line* para disseminação de informações de trânsito em uma área metropolitana. No entanto, o modelo desenvolvido é genérico o suficiente para ser utilizado em um grande espectro de aplicações. A abordagem seguida no SIDAM é baseada nos seguintes requisitos de projeto:

- as aplicações devem ser capazes de se adaptar a variações de carga, padrões de requisição e a diversos cenários de disponibilidade de recursos;
- as aplicações devem prover uma alta disponibilidade de seus dados e serviços e, sempre que possível, garantir uma baixa latência de acesso;
- apesar da base de dados global estar distribuída em várias máquinas, esta distribuição e localização física de qualquer dado específico deve permanecer transparente ao cliente.

6.2 Modelo de Computação Adotado no SIDAM

O projeto SIDAM adota o **modelo indireto** [BBIM93] para as interações cliente-servidor. Neste modelo, ilustrado na figura 6.1, todas as comunicações de e para um dispositivo móvel são intermediadas por estações base que atuam como representantes de todos os dispositivos móveis dentro de uma dada região geográfica.

O modelo do sistema consiste de dois tipos de máquinas: os computadores fixos e os dispositivos móveis, onde os computadores fixos estão conectados entre si através de uma rede fixa e confiável.

Alguns computadores fixos atuam como estações base. Cada estação base define uma região geográfica, denominada célula, na qual é capaz de se comunicar com o conjunto de dispositivos móveis que se encontram dentro da mesma. A informação sobre quais dispositivos estão presentes a cada momento dentro da célula é mantida pela respectiva estação base.

Os demais servidores fixos atuam como servidores de dados e podem disponibilizar serviços tanto para máquinas fixas quanto móveis. Na aplicação de referência SIDAM, os servidores de dados armazenam informações de trânsito.

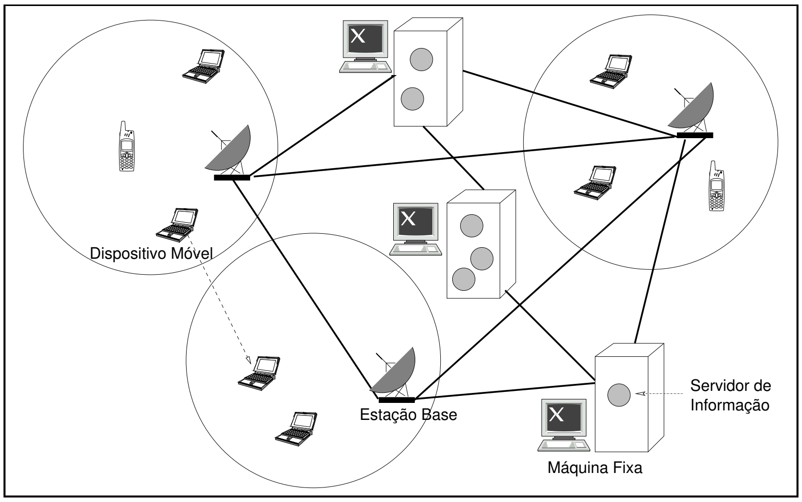

Figura 6.1: Modelo de computação utilizado no SIDAM

6.2.1 Aplicação de Disseminação de Informações de Trânsito

A aplicação de referência utilizada no SIDAM é um serviço distribuído de disseminação de informações de trânsito para a cidade de São Paulo. A aplicação pode receber consultas e atualizações de seus dados por usuários móveis. Estes podem ser tanto usuários não privilegiados, como cidadãos comuns consultando informações sobre a condição do trânsito quanto usuários privilegiados, como funcionários da Companhia de Engenharia de Tráfego (CET), que podem consultar e atualizar as informações constantes no sistema.

As principais vias da cidade de São Paulo possuem sensores que medem o fluxo do tráfego que por elas passa. Estes sensores são utilizados em conjunto com semáforos inteligentes, que adaptam a duração das luzes de sinalização de acordo com a leitura indicada pelos sensores. Os dados obtidos através destes sensores são também utilizados para a atualização das informações mantidas pela aplicação SIDAM.

Para manter gerenciáveis o tamanho da base de dados e o volume de consultas, a aplicação SIDAM considera a partição da cidade em microrregiões. Cada microrregião corresponde a algumas quadras e representa uma unidade atômica de armazenamento de informações de trânsito. Os dados das microrregiões são armazenados em **Servidores de Informação**. Cada Servidor de Informação pode conter dados de uma ou mais microrregiões. Servidores de Informação são executados por computadores fixos. As seguintes operações estão disponíveis aos usuários:

1. **query**: o usuário informa o nome de uma rua, um segmento de rua ou microrregião e a aplicação retorna atributos como velocidade média do tráfego neste trecho, taxa de congestionamento e ocorrências especiais tais como, por exemplo, acidentes, manifestações públicas ou inundações;

2. **update**: o usuário informa a sua identificação, o nome de uma rua, um segmento de rua ou microrregião, e o novo valor para qualquer um dos atributos anteriores;

3. **subscribe**: o usuário informa o nome de uma rua, um segmento de rua ou microrregião e um valor limite para algum dos atributos. Sempre que o valor corrente da microrregião para o atributo passar do limite indicado, o usuário é notificado;

4. **broadcast**: o usuário informa sua identificação, a identificação de um grupo de usuários (que é pré-definido) e a mensagem a ser enviada ao grupo. Esta operação é de uso exclusivo dos usuários privilegiados (funcionários da CET) e é utilizada para facilitar o planejamento e execução de atividades em conjunto.

6.3 Necessidade de Adaptação Dinâmica

Em um ambiente dinâmico como o da aplicação de referência do SIDAM, onde os clientes são móveis e os dados encontram-se distribuídos, técnicas adaptativas são fundamentais para se atingir uma alta disponibilidade do serviço, balancear adequadamente a carga de trabalho entre os servidores e minimizar o tempo de resposta aos clientes.

Esta seção descreve algumas possibilidades de uso de adaptação dinâmica como forma de tratar variações no ambiente de execução da aplicação de referência do SIDAM.

6.3.1 Suporte a Variações no Padrão de Acesso a Servidores de Informação

Devido à mobilidade dos usuários, a quantidade de acessos gerados por célula para um dado Servidor de Informação pode variar significativamente no tempo. Um evento em uma região da cidade que desperte grande interesse como um jogo de futebol, por exemplo, pode gerar uma grande quantidade de requisições para certos Servidores de Informação. Cada requisição de um dispositivo móvel deve ser roteada da estação base responsável pela célula na qual o dispositivo se encontra até a máquina na qual é executado o Servidor de Informação que contém os dados da microrregião consultada. Este roteamento pode ser custoso, caso a mensagem tenha que passar por diversos nós da rede.

Uma adaptação para lidar com esta variação no padrão de acesso é migrar o Servidor de Informação para uma máquina disponível mais próxima[1] das estações base cujas células estiverem originando um maior número de acessos.

6.3.2 Suporte a Variações no Estado de Utilização de Recursos

A carga de trabalho nas diversas máquinas servidoras pode variar significativamente, quer pela quantidade de acessos destinados aos Servidores de Informação que nelas executam, quer pela execução de outras aplicações, caso as máquinas não sejam usadas exclusivamente para a aplicação de disseminação de informações de tráfego.

A migração de Servidores de Informação pode também ajudar no balanceamento da carga de trabalho entre as máquinas, evitando situações de sobrecarga. A utilização do mesmo mecanismo de adaptação (migração de Servidores de Informação) para tratar duas situações distintas exige cuidados para evitar ações de reconfiguração conflitantes. Ao migrarmos um Servidor de Informação para deixá-lo mais próximo das células que geram uma maior quantidade de requisições, devemos ter o cuidado de verificar a carga de trabalho na máquina destino. De forma semelhante, ao migrarmos um Servidor de Informação para aliviarmos a carga de trabalho em uma máquina, devemos dar preferência às máquinas próximas das regiões que geram uma maior quantidade de acessos.

6.3.3 Suporte a Heterogeneidade de Dispositivos Móveis

Dispositivos móveis apresentam grande diversidade nas suas propriedades tanto de hardware quanto de software. Dependendo do dispositivo, a resolução máxima da tela varia. Alguns dispositivos possuem tela colorida, enquanto outros comportam apenas diversos tons de cinza. Alguns dispositivos podem apresentar imagens e interpretam diversos formatos como JPEG e PNG, outros apenas texto. O poder de processamento, quantidade de memória, dispositivos de entrada e saída

[1] Proximidade em termos de custo de roteamento (por exemplo, número de *hops*.)

e diversas outras propriedades podem variar de acordo com o dispositivo. Esta diversidade dificulta o desenvolvimento de sistemas de informação para ambientes de computação móvel já que os dispositivos clientes podem não estar aptos a apresentar os dados obtidos, caso sejam utilizados diversos formatos como imagens, documentos HTML, áudio e vídeo.

Uma possível abordagem para este problema é variar o formato do resultado obtido em resposta a uma consulta de acordo com as propriedades do dispositivo cliente. No contexto da aplicação de referência do SIDAM, a situação do trânsito em uma microrregião pode ser apresentada de diversas formas:

- Através de uma imagem colorida, na qual operações de ampliação e redução são permitidas e as ruas são coloridas de acordo com o volume de tráfego;
- Através de uma imagem em tons de cinza;
- Através de um áudio que enfatize situações de congestionamento;
- Simplesmente de forma textual.

Analogamente, pode-se lidar com variações na largura de banda disponível para comunicação com um dispositivo móvel. De acordo com a largura de banda disponível e as propriedades de hardware e software do dispositivo, escolhe-se o formato da resposta a ser apresentada ao cliente.

6.4 Projeto do Protótipo SIDAM

Um dos resultados do projeto SIDAM foi o desenvolvimento de um protótipo denominado PSIT (Protótipo de um Sistema de disseminação de Informações de Trânsito). Os principais componentes do PSIT (`http://www.ime.usp.br/~sidam/software/psit/`) são:

1. **IS - Information Server:** servidor responsável por manter as informações de trânsito de um subconjunto de microrregiões. Tipicamente haverá diversos Servidores de Informação distribuídos por máquinas interligadas pela rede fixa;
2. **LS - Locator Server:** o Servidor de Localização mantém a informação sobre quais Servidores de Informação são responsáveis por quais microrregiões, ou seja, um mapeamento entre

Servidores de Informação e microrregiões. O fato de os dados a respeito das microrregiões estarem distribuídos em diversos Servidores de Informação é transparente aos clientes. Um cliente pode se comunicar com qualquer Servidor de Informação e realizar uma consulta a respeito de qualquer microrregião. Caso o servidor contactado não possua a informação solicitada, este acessa um Servidor de Localização para obter o endereço de um servidor que possui o dado e reencaminha a consulta do cliente para este servidor. Podem existir diversos Servidores de Localização instanciados no ambiente. Eles utilizam uma árvore de propagação para manter o mapeamento consistente entre si;

3. **Sidam:** um servidor especial denominado Sidam mantém referências a todos os servidores de informação e localização instanciados no ambiente distribuído. Ele é utilizado para obtenção de referências iniciais no momento da inicialização destes componentes. Há apenas um servidor Sidam no sistema.

6.5 Adicionando Adaptação Dinâmica ao Protótipo SIDAM

Utilizamos o arcabouço para instrumentar a aplicação PSIT de forma a promover reconfigurações em tempo de execução em resposta a variações no ambiente de execução.

Nosso principal objetivo foi validar a aplicabilidade do arcabouço e sua aderência aos requisitos de projeto descritos na seção 2.3. Do ponto de vista da aplicação PSIT, a incorporação de mecanismos adaptativos teve por objetivo melhorar seu desempenho, tendo por base os seguintes aspectos: (1) minimizar o tempo de resposta aos clientes e (2) manter a carga de trabalho das máquinas servidoras em um patamar adequado de utilização.

O mecanismo de reconfiguração adotado em nossa implementação foi a reinstanciação de Servidores de Informação [2]. Adicionamos um novo componente ao PSIT, denominado Starter, a ser

[2]Utilizamos o termo reinstanciação já que em nossa implementação não há migração do código do Servidor de Informação de uma máquina para outra. Um novo servidor é instanciado na máquina destino e o servidor antigo terá sua execução interrompida após o término da propagação da referência do novo servidor aos clientes da aplicação. Também não foi necessário realizar uma replicação dos dados mantidos pelo servidor antigo já que na implementação corrente as máquinas nos quais Servidores de Informação são executados compartilham um espaço em disco através do uso de NFS onde os dados são armazenados.

instanciado em toda máquina capaz de executar Servidores de Informação. O Starter é responsável por iniciar a execução de Servidores de Informação.

6.5.1 Adaptação Baseada na Interação entre Componentes da Aplicação

Utilizando o arcabouço, incorporamos ao PSIT o suporte a variações no padrão de acesso a Servidores de Informação, conforme descrito na seção 6.3.1. Servidores de Informação são reinstanciados em máquinas próximas às estações base nas quais são originadas o maior número de requisições. Isto faz com que os gastos de comunicação na rede sejam minimizados, obtendo-se um menor tempo médio de resposta aos clientes. Para determinar a quantidade de requisições proveniente de cada célula, utilizou-se o serviço de monitoração das interações entre os objetos da aplicação disponibilizado pelo arcabouço (seção 3.2.1).

O primeiro passo na implementação deste mecanismo de adaptação no PSIT foi a incorporação da monitoração das interações entre objetos da classe `Query`, responsáveis por solicitar informações a respeito da situação do trânsito em uma dada microrregião, e objetos da classe `Server`, que implementam Servidores de Informação. Dos vários métodos disponibilizados pelos Servidores de Informação, apenas as chamadas a `getInfo()` necessitam ser monitoradas. Este método fornece a condição de tráfego em uma dada microrregião como, por exemplo, densidade de veículos, tamanho global do congestionamento, etc.

Como a comunicação entre objetos do PSIT é realizada através de Java RMI, utilizamos a solução alternativa disponibilizada pelo arcabouço para a monitoração de interações. Utilizando os procedimentos descritos na seção 3.2.1, implementamos uma sub-classe de `Interceptor` e a instanciamos no lado do cliente da aplicação(`Query`). Para armazenar em memória persistente os dados relativos às chamadas realizadas ao método `getInfo()`, definimos uma tabela em um banco de dados relacional, cuja descrição é apresentada na tabela 6.1.

Uma vez definidas quais interações devem ser monitoradas e implementado o suporte a esta monitoração, implementamos o avaliador para um evento que denominamos `AccessLocation`. Seguindo o procedimento descrito na seção 4.2.2, escrevemos a classe `AccessLocationEvaluator` que herda da classe `Evaluator`, disponibilizada pelo arcabouço. Sobrescrevemos seu método `user_evaluation_code()` que lê as tuplas do banco de dados, contabilizando a quantidade de acessos

Campo	Tipo	Descrição
id	mediumint(9)	Chave da tabela utilizando-se *auto-increment*[a]
ts	timestamp(14)	Marca de tempo na qual a tupla foi salva no BD
clientLocation	varchar(125)	Região onde o cliente se encontrava ao enviar a requisição para o servidor
clientHostName	varchar(125)	Nome da máquina na qual o cliente se encontrava ao enviar a requisição para o servidor
serverLocation	varchar(125)	Região onde o servidor se encontrava ao atender a requisição
serverHostName	varchar(125)	Nome da máquina na qual o servidor se encontrava ao atender a requisição
serverName	varchar(125)	Nome do servidor que atendeu a requisição
groupID	varchar(125)	Identificação da microrregião sendo consultada
status	int(11)	Identifica se o solicitante é um objeto `Query` ou um outro Servidor de Informação

Tabela 6.1: Descrição da tabela de interações do método `getInfo()`

[a]O próprio SGBD gera o valor a ser armazenado em atributos definidos como *auto-increment* como uma seqüência de números inteiros incrementados de 1 em 1 a medida que novas tuplas são armazenadas.

gerados para cada Servidor de Informação, agrupando-as pelas "regiões geradoras" das requisições. Uma notificação da ocorrência deste evento é gerada caso existam servidores que possuam uma quantidade de acessos realizados por clientes "distantes" pelo menos 20% superior aos acessos realizados por clientes localizados em sua região. Entre os atributos da notificação de eventos do tipo `AccessLocation`, constam os servidores que se encontram nesta condição e a região que gerou o maior número de acessos para cada um deles.

A última etapa de implementação consiste em preparar a aplicação para receber notificações da ocorrência do evento `AccessLocation` e tratá-las. Dado que o componente Sidam mantém referências a todos os servidores de informação e localização instanciados no ambiente distribuído, optamos por adicionar a *thread* `EnvironmentAware` a este componente. Conforme descrito na seção 5.3, esta *thread* recebe notificações sobre a ocorrência de eventos de interação no ambiente. A estratégia implementada para tratar a ocorrência de eventos `AccessLocation` localiza, para cada Servidor de Informação constante da notificação do evento, uma máquina na região que tenha gerado o maior número de acessos para este servidor e que esteja com uma carga de trabalho abaixo de um determinado patamar. São, então, chamados os métodos nos componentes da aplicação que, com o auxílio dos Configuradores de Componentes, promovem a reinstanciação do servidor. A determinação da carga de trabalho de uma máquina é obtida através de consultas a Objetos de Monitoração de Recursos (seção 3.1.1) nela instanciados. Estes objetos monitoram o percentual de uso do processador e a quantidade de memória física disponível.

Estratégias bem mais complexas para o tratamento do evento `AccessLocation` poderiam também ser implementadas. A estratégia poderia levar em consideração o custo de comunicação entre as diversas regiões da cidade. Este custo poderia ser obtido através de Objetos de Monitoração de Recursos que mediriam a latência de comunicação entre os nós da rede. O evento `AcessLocation` poderia informar a quantidade de acessos gerada pelos clientes de todas as regiões da cidade e não somente da região que gerou o maior número de acessos. Usando esta informação, um algoritmo de otimização poderia ser executado para escolher a região que deveria abrigar o Servidor de Informação de forma a minimizar o custo global de comunicação, caso o padrão de acesso ao mesmo fosse mantido. Poderíamos também levar em consideração o tempo necessário para migrar o servidor. O que queremos destacar, no entanto, é que o modelo definido nesta tese possui os mecanismos necessários para auxiliar o usuário no desenvolvimento de aplicações adaptativas e que estes podem ser empregados em uma grande variedade de estratégias.

O capítulo 7 apresenta os resultados obtidos com a incorporação do mecanismo de adaptação descrito nesta seção para uma série de cenários simulados. Estes cenários descrevem diversos padrões de acesso aos Servidores de Informação. A análise dos resultados permite verificar que, em algumas situações, o uso de adaptação gera resultados bastante significativos, enquanto que em outros o resultado pode até ser contra-producente. Será também discutida a importância de se determinar a freqüência com que o ambiente de execução deve ser analisado.

6.5.2 Adaptação Baseada na Variação da Disponibilidade de Recursos

Para manter a carga de trabalho das máquinas servidoras dentro de um determinado intervalo, utilizou-se a monitoração de recursos do ambiente. Caso uma máquina ficasse sobrecarregada, parte de seus Servidores de Informação seriam reinstanciados para máquinas servidoras menos utilizadas, como descrito na seção 6.3.2.

A monitoração do ambiente é realizada através da instanciação de Objetos de Monitoração de Recursos em máquinas nas quais Servidores de Informação podem ser executados. Em cada máquina foram instanciados dois OMRs, um responsável por monitorar o percentual de uso da UCP e outro que monitora a quantidade de memória física disponível.

Definimos um evento, denominado `ServerOverloaded`, que permite detectar quando uma destas máquinas torna-se sobrecarregada. Esta sobrecarga foi definida como a situação na qual há um grande consumo de UCP e a quantidade de memória física disponível está abaixo de um determinado valor.

Para o recebimento de notificações deste evento, utilizamos a mesma *thread* `EnvironmentAware` instanciada no componente Sidam.

Implementamos duas estratégias para tratar ocorrências de eventos `ServerOverloaded`:

1. **FirstFit**: realiza uma consulta seqüencial aos OMRs instanciados nas demais máquinas pertencentes à mesma região da máquina sobrecarregada, a fim de verificar a disponibilidade de UCP e memória principal. A primeira máquina que possuir valores abaixo de um determinado patamar é escolhida para receber a reinstanciação de um dos Servidores de Informação da máquina sobrecarregada, interrompendo-se o procedimento de consulta;

2. **BestFit**: consulta todas as máquinas pertencentes à mesma região da máquina sobrecarregada, escolhendo-se aquela que possuir a maior disponibilidade para receber a reinstanciação de um dos Servidores de Informação da máquina sobrecarregada.

Caso não haja uma máquina que satisfaça os critérios de disponibilidade definidos, nenhuma reconfiguração da aplicação é realizada.

6.5.3 Conclusões a Respeito da Utilização do Arcabouço

A experiência da utilização do arcabouço na instrumentação do PSIT nos permite avaliar sua aplicabilidade no desenvolvimento de aplicações distribuídas adaptativas. Os principais aspectos observados foram:

1. o arcabouço facilitou a análise da solução por separar de maneira clara os aspectos relativos à monitoração do ambiente, à verificação de mudanças em seu estado e à reconfiguração da aplicação;

2. o projeto da solução também foi bastante simplificado já que o arcabouço propõe os componentes necessários para realizar cada uma das tarefas acima, definindo-se de forma clara a colaboração entre eles que é realizada através de interfaces bem definidas;

3. o uso do arcabouço permite uma boa reutilização de código, agilizando a implementação da solução;

4. o arcabouço mostrou-se bastante flexível, permitindo ao usuário particularizar os aspectos relativos a monitoração, detecção de eventos e reconfiguração às necessidades específicas de sua aplicação;

5. o arcabouço permite uma clara separação do código contendo a funcionalidade da aplicação daquele responsável por sua reconfiguração;

6. o arcabouço auxilia a reconfiguração de aplicações distribuídas compostas por diversos componentes instanciados em vários nós do ambiente distribuído.

Apesar de o arcabouço facilitar consideravelmente a construção de aplicações adaptativas distribuídas, seu uso não exime o desenvolvedor de refletir sobre questões fundamentais ao desenvolvimento de software adaptativo que, no nosso entendimento, só podem ser resolvidas levando-se em consideração aspectos específicos da aplicação. Entre elas, destacam-se:

1. Quais os objetivos da adaptação?

2. Em que circunstâncias a adaptação representa um ganho de custo/benefício?

3. Que informações do ambiente devem ser coletadas para que se possa tomar decisões sobre a adaptação?

4. Com que freqüência o sistema deve verificar se há necessidade de realizar adaptações?

Uma discussão mais completa sobre estas questões encontra-se na seção 2.2.

Capítulo 7

Resultados Experimentais

Este capítulo apresenta os resultados obtidos com a incorporação de mecanismos de adaptação dinâmica ao protótipo do sistema de disseminação de informações de trânsito (PSIT), descrito no capítulo 6.

Os testes descritos a seguir foram realizados no Laboratório de Computação Paralela e Distribuída (LCPD) pertencente ao Instituto de Matemática e Estatística da Universidade de São Paulo.

Ao todo, foram utilizados 6 microcomputadores, cujas especificações de hardware e software básico encontram-se no apêndice A. Estas máquinas encontram-se interligadas através de uma rede Ethernet [Tan02] de 10 Mbps. Utilizamos também uma máquina RISC/Solaris, responsável pela execução do Sistema Gerenciador de Banco de Dados MySQL.

Os testes foram realizados no período noturno, no qual poucos usuários utilizam os equipamentos do laboratório, de forma a minimizar as interferências decorrentes da execução de outras aplicações no ambiente.

7.1 Variações no Padrão das Interações entre Componentes

Conforme descrito na seção 6.5.1, utilizamos o arcabouço para incorporar mecanismos de adaptação dinâmica ao protótipo da aplicação de referência do projeto SIDAM (PSIT), utilizando a análise do padrão das interações realizadas entre os componentes da aplicação para a determinação da necessidade de realizar reconfigurações na mesma.

Avaliamos o desempenho da versão adaptativa do PSIT através da simulação de diversos cenários. Em cada cenário, variamos a quantidade de requisições oriunda de cada região de uma cidade hipotética bem como o período de estabilidade de cada padrão de acesso. A cidade foi dividida em quatro macrorregiões (norte, sul, leste e oeste) e definimos um custo de comunicação para computadores pertencentes à rede fixa (seção 6.2) entre cada uma das regiões, segundo a tabela 7.1. Sem perda de generalidade, considerou-se que o custo de comunicação para dois computadores em uma mesma região seria igual.

	Norte	**Sul**	**Oeste**	**Leste**
Norte	-	20	8	10
Sul	20	-	10	8
Oeste	8	10	-	15
Leste	10	8	15	-

Tabela 7.1: Custo de comunicação entre regiões da cidade em milisegundos

Utilizamos o serviço de monitoração descrito na seção 3.2.1, para a interceptação das chamadas ao método `getInfo()`, disponibilizado por Servidores de Informação. Este método informa a suposta situação do trânsito em uma dada microrregião. A análise da quantidade de requisições oriunda de cada região da cidade foi utilizada para determinar uma melhor distribuição dos Servidores de Informação entre os computadores que compõem a rede fixa. O mecanismo de adaptação implementado foi a reinstanciação de Servidores de Informação em uma das máquinas pertencentes à região de onde foi gerada a maior quantidade de acessos aos mesmos, objetivando-se reduzir o custo global de comunicação (para a maioria dos supostos usuários).

Para simplificar a análise dos resultados obtidos através das simulações, utilizamos apenas um Servidor de Informação em nossos testes que, portanto, seria reinstanciado entre as macrorregiões de acordo com a demanda. No entanto, nossa implementação não impõe qualquer limite à quantidade de Servidores de Informação que podem ser instanciados no ambiente.

7.1.1 Primeiro Cenário

O primeiro cenário simulado inicia com a maioria dos acessos (50%) sendo realizada por clientes localizados em uma determinada região (no caso, sul), enquanto que as regiões norte, leste e oeste contribuem com 10%, 20% e 20% dos acessos, respectivamente. Em seguida, este padrão de acesso é alterado, tornando-se a região oeste responsável por 70% dos acessos. Cada uma das demais regiões passa a contribuir com 10% dos acessos. O Servidor de Informação é inicialmente instanciado em uma máquina pertencente à região norte.

A versão adaptativa do PSIT reage à primeira fase da simulação reinstanciando o Servidor de Informação em uma máquina pertencente à região sul. Como esperado, a mudança do padrão de acesso correspondente à segunda fase da simulação faz com que a aplicação seja novamente reconfigurada, sendo o Servidor de Informação reinstanciado na região oeste.

A figura 7.1 apresenta o diagrama de implantação UML relativo ao início da execução deste cenário.

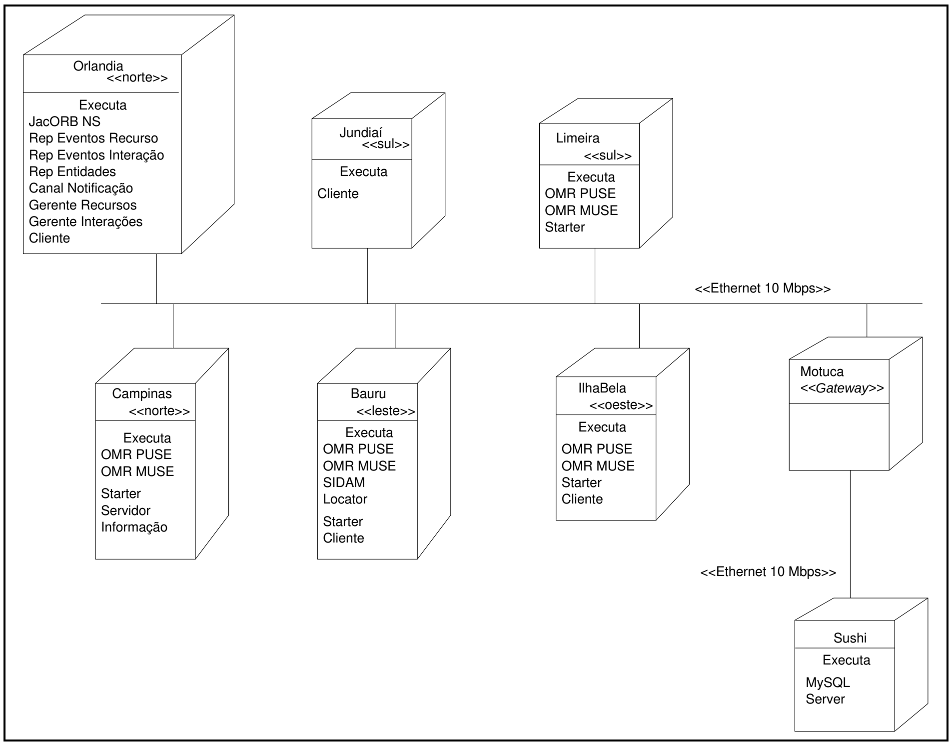

Figura 7.1: Primeiro cenário: diagrama de implantação

Realizamos diversos testes para este cenário, alterando a periodicidade na qual o Avaliador (seção 4.2.2) responsável pela detecção de mudanças no padrão de acesso ao método `getInfo()` é executado. Os resultados obtidos são ilustrados no gráfico correspondente à figura 7.2. O eixo x indica a periodicidade na qual o Avaliador é executado, enquanto o eixo y representa o tempo médio de acesso gasto pelo Servidor de Informação para atender as requisições.

O tempo médio de acesso foi calculado dividindo-se o tempo total gasto pelo Servidor de Informação para atender a todas as requisições do teste pela quantidade de requisições realizadas. Para calcular o tempo gasto pelo Servidor de Informação para atender a cada requisição, utilizamos o serviço de monitoração das interações entre os objetos da aplicação conforme descrito na seção 6.5.1. Para cada chamada ao método `getInfo()`, coletamos a marca de tempo correspondente ao momento no qual a chamada foi realizada pelo cliente e a marca de tempo correspondente ao recebimento pelo cliente da resposta enviada pelo Servidor de Informação. A diferença em milisegundos entre estes dois tempos corresponde ao tempo gasto no atendimento da requisição. Este tempo, portanto, compreende o tempo necessário ao Servidor de Informação para processar a requisição somado aos custos de comunição relativos ao envio das mensagens de requisição e de resposta.

Repetimos cada teste 30 vezes e utilizamos a média (do tempo médio de acesso, calculado conforme descrito no parágrafo anterior) obtida através destas execuções para traçar os pontos correspondentes às curvas apresentadas na figura 7.2. As tabelas 7.2 e 7.3 apresentam os seguintes dados relativos ao tempo médio de acesso obtido através destas execuções: média, mínimo, máximo, desvio padrão e intervalo de 95% de confiança [WW90].

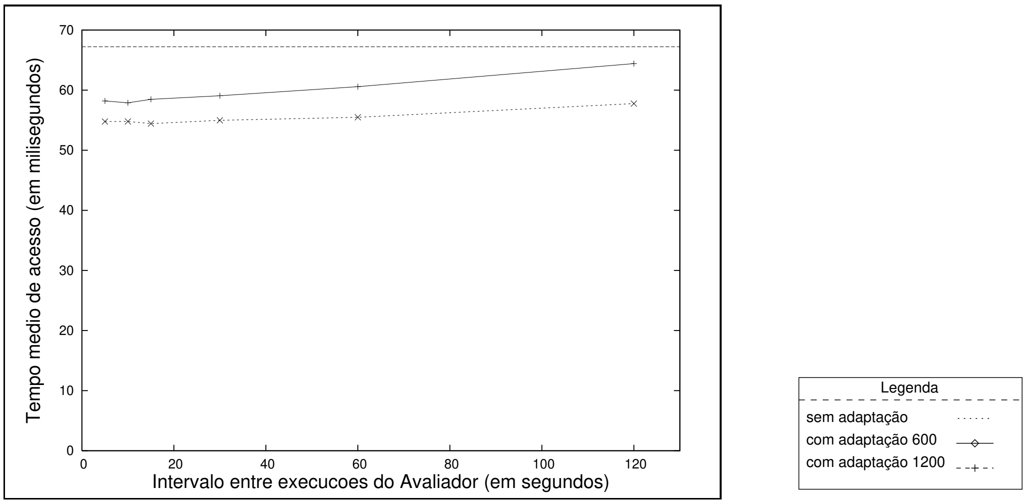

Figura 7.2: Resultados do primeiro cenário

No gráfico correspondente à figura 7.2, a curva identificada pela legenda "com adaptação 600" ilustra o resultado obtido quando são realizadas 600 consultas ao Servidor de Informação (300 em cada fase da simulação), enquanto a curva identificada pela legenda "com adaptação 1200" ilustra o resultado obtido ao se realizar 1200 consultas. A linha pontilhada horizontal indica o tempo médio de acesso obtido através da execução da versão não adaptativa do PSIT. Estes resultados indicam claramente um considerável ganho de desempenho obtido pela incorporação de mecanismos de adaptação dinâmica na aplicação de referência do SIDAM, que foi de 17,33% no teste realizado com 1200 consultas quando o Avaliador foi executado a cada 15 segundos.

	SA[a]	5	10	15	30	60	120
média	67,21	58,17	57,90	58,47	59,08	60,58	64,42
mínimo	58,60	51,55	48,45	48,23	49,54	48,63	56,93
máximo	73,87	64,56	66,64	64,11	68,54	71,58	71,64
desvio padrão	3,92	3,42	4,26	3,92	4,84	5,92	4,20
int confiança 95%	65,81	56,94	56,38	57,07	57,34	58,46	62,91
	68,61	59,39	59,42	59,87	60,81	62,70	65,92

Tabela 7.2: Dados relativos às execuções do primeiro cenário utilizando-se 600 requisições

[a] Sem Adaptação

	SA	5	10	15	30	60	120
média	65,85	54,78	54,79	54,44	54,98	55,48	57,77
mínimo	59,52	49,99	48,66	49,12	49,43	47,74	47,57
máximo	71,67	60,56	58,06	58,38	64,28	60,46	64,35
desvio padrão	2,42	2,41	2,24	2,36	3,36	2,86	3,66
int confiança 95%	64,98	53,92	53,98	53,60	53,77	54,46	56,46
	66,71	55,65	55,59	55,28	56,18	56,51	59,08

Tabela 7.3: Dados relativos às execuções do primeiro cenário utilizando-se 1200 requisições

O gráfico correspondente à figura 7.3 apresenta o tempo de acesso ao Servidor de Informação para cada uma das 600 consultas realizadas em um dos testes relativos à este primeiro cenário ao utilizarmos a versão não adaptativa do PSIT. Percebe-se que, para a maioria das consultas realizadas, o tempo de resposta ficou entre 50 e 100 milisegundos, sendo de aproximadamente 75.

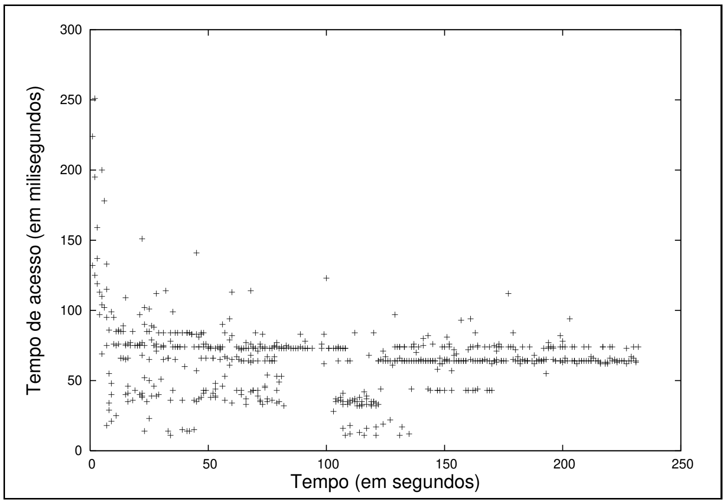

Figura 7.3: Tempos de acesso obtidos quando não utilizamos adaptação

A figura 7.4 apresenta um gráfico semelhante, obtido através da execução da versão adaptativa do PSIT. Percebe-se que o tempo de acesso foi reduzido, obtendo-se um valor de aproximadamente 50 milisegundos para a maioria das consultas realizadas.

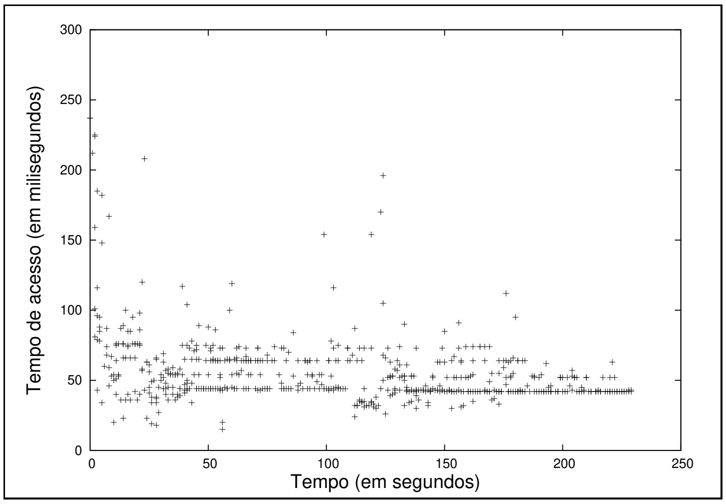

Figura 7.4: Tempos de acesso obtidos ao utilizarmos adaptação

7.1.2 Segundo Cenário

O objetivo do segundo cenário foi testar uma situação pouco favorável aos mecanismos de adaptação incorporados ao PSIT, fazendo com que a diferença entre os volumes de acesso realizados por cada região fosse pequena.

Na primeira etapa deste cenário, cada região gera a mesma quantidade de acessos, de um total de 300 requisições. Este padrão de acesso é então alterado, tornando-se a região sul responsável por 40% dos acessos, enquanto as demais regiões geram apenas 20% dos mesmos, quando são realizadas mais 300 requisições. Durante a segunda etapa, o Servidor de Informação é reinstanciado na região sul mas, como esperado, o ganho de desempenho obtido é pequeno, conforme ilustrado na figura 7.5. O melhor resultado obtido foi um ganho de 8,45%, ao executarmos o Avaliador a cada 15 segundos. Foram realizadas 300 requisições, sendo igualmente divididas entre as fases.

Assim como no primeiro cenário, realizamos 30 testes e utilizamos a média de seus resultados para a construção do gráfico correspondente à figura 7.5. A tabela 7.4 apresenta dados relativos a estas 30 execuções.

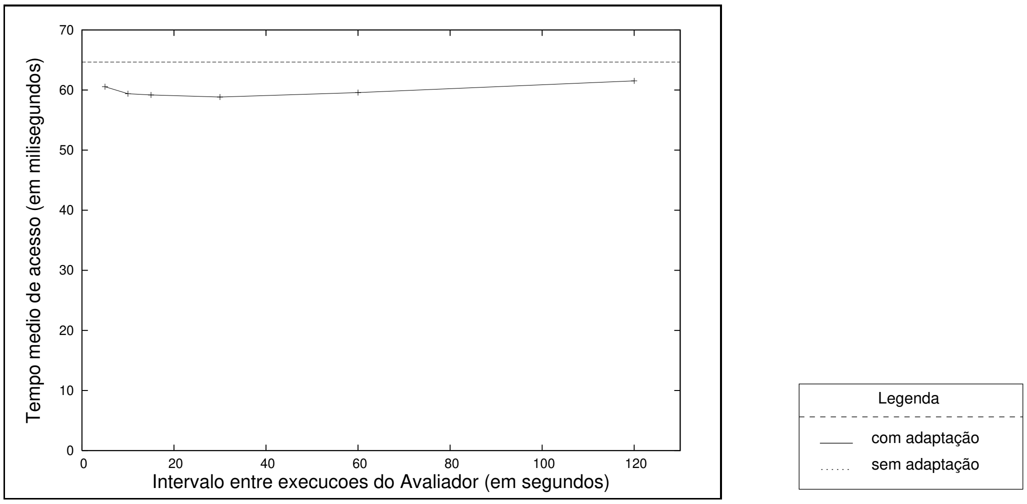

Figura 7.5: Resultados do segundo cenário

	SA	5	10	15	30	60	120
média	64,65	60,55	59,38	59,19	58,83	59,57	61,52
mínimo	56,41	46,26	49,31	44,47	46,04	43,74	52,49
máximo	70,90	70,85	67,50	69,03	69,06	69,34	69,98
desvio padrão	3,35	5,74	5,31	5,91	6,04	6,16	4,63
int confiança 95%	63,05	58,50	57,48	57,08	56,67	57,36	59,86
	65,45	62,60	61,28	61,31	60,99	61,78	63,18

Tabela 7.4: Dados relativos às execuções do segundo cenário

7.1.3 Terceiro Cenário

O terceiro cenário ilustra uma situação na qual o padrão de acesso é alterado com alta freqüência: a cada minuto, sugerindo uma situação onde os usuários apresentam alta taxa de mobilidade inter-regiões. Em cada fase da simulação, 100% dos acessos são realizados a partir de uma única região, mas esta é chaveada entre duas regiões (norte e sul). O total de acessos nesta simulação foi de 360, sendo realizados 60 acessos a cada fase. A figura 7.6 apresenta o gráfico com os resultados obtidos na execução deste cenário.

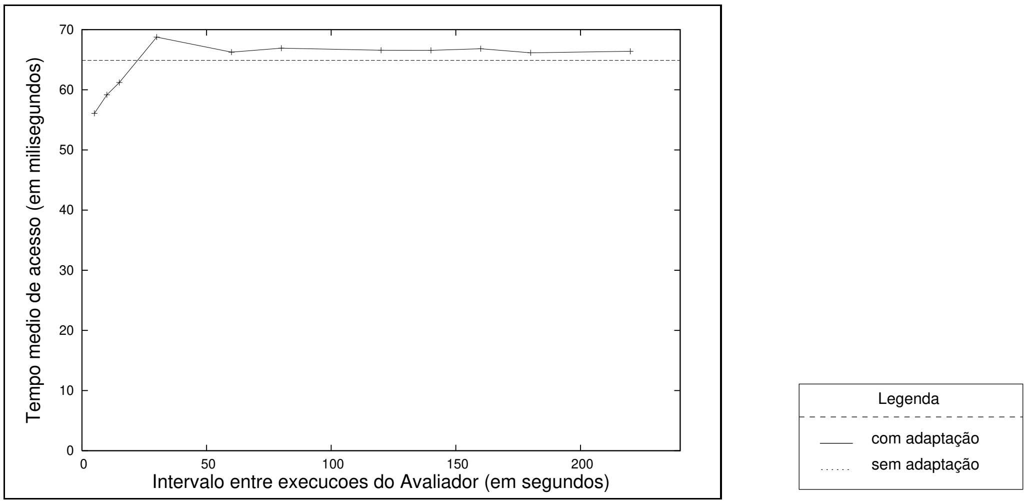

Figura 7.6: Resultados do terceiro cenário

Um aspecto relevante a ser observado a partir deste experimento é a importância de se ajustar o parâmetro que determina a freqüência de execução do Avaliador responsável pela análise dos

	SA	5	10	15	30	60	80	120	140	160	180	220
média	64,89	56,09	59,17	61,21	68,76	66,26	66,92	66,58	66,55	66,83	64,14	66,40
mínimo	56,81	40,06	47,39	49,32	58,57	54,01	58,39	59,26	54,86	59,88	58,03	59,62
máximo	73,21	66,16	72,86	68,51	75,54	77,17	76,06	76,34	73,07	71,99	73,34	74,72
desvio padrão	3,62	5,14	5,71	4,68	4,95	4,95	4,50	3,85	4,75	3,42	3,87	3,65
int confiança 95%	63,55	54,15	56,89	59,41	66,86	64,35	65,15	65,10	64,73	65,51	64,65	65,00
	66,23	58,03	61,45	63,01	70,67	68,16	68,68	68,06	68,38	68,15	67,63	67,81

Tabela 7.5: Dados relativos às execuções do terceiro cenário

dados coletados pela interceptação das chamadas realizadas ao método `getInfo()`. Se o intervalo entre execuções deste Avaliador estiver muito próximo do intervalo no qual o padrão de acesso varia, a aplicação adaptativa pode apresentar um desempenho inferior ao obtido pela versão não adaptativa, conforme ilustrado na figura 7.6. O motivo para este comportamento é o fato do gasto com a reconfiguração da aplicação não ser compensado por um melhor desempenho da mesma, já que o padrão de acesso sofre nova variação logo após cada reconfiguração . Por outro lado, se o intervalo entre execuções do Avaliador for muito grande em comparação ao intervalo no qual o padrão de acesso varia, o desempenho da aplicação adaptativa se aproximará do resultado obtido com sua versão não adaptativa, pois as reconfigurações tornar-se-ão cada vez mais raras.

7.1.4 Quarto Cenário

Este cenário demonstra uma situação na qual os mecanismos de adaptação incorporados ao PSIT resultam em um forte ganho de desempenho para a aplicação. Em uma primeira etapa, 600 acessos são realizados ao Servidor de Informação, sendo 90% dos mesmos gerados por clientes localizados na região norte. Clientes localizados nas regiões sul, leste e oeste contribuem com 5, 3 e 2% do total dos acessos realizados, respectivamente. Em uma segunda etapa, mais 600 requisições são realizadas, sendo 90% das mesmas gerados por clientes localizados na região sul. Clientes presentes nas regiões norte, leste e oeste realizam 5, 3 e 2% do total de requisições, respectivamente.

Para as simulações realizadas neste quarto cenário, alteramos a tabela que determina o custo de comunicação entre máquinas localizadas em diferentes regiões da cidade, conforme a tabela 7.6.

O resultado obtido é apresentado no gráfico correspondente à figura 7.7 onde observamos que

	Norte	**Sul**	**Oeste**	**Leste**
Norte	-	100	100	100
Sul	100	-	100	100
Oeste	100	100	-	100
Leste	100	100	100	-

Tabela 7.6: Custo de comunicação entre regiões da cidade em milisegundos

a utilização da versão não adaptativa do PSIT resultou em um tempo médio de acesso 132,92% superior ao obtido com a versão adaptativa ao executarmos o Avaliador a cada 5 segundos. A tabela 7.7 apresenta dados relativos às execuções realizadas.

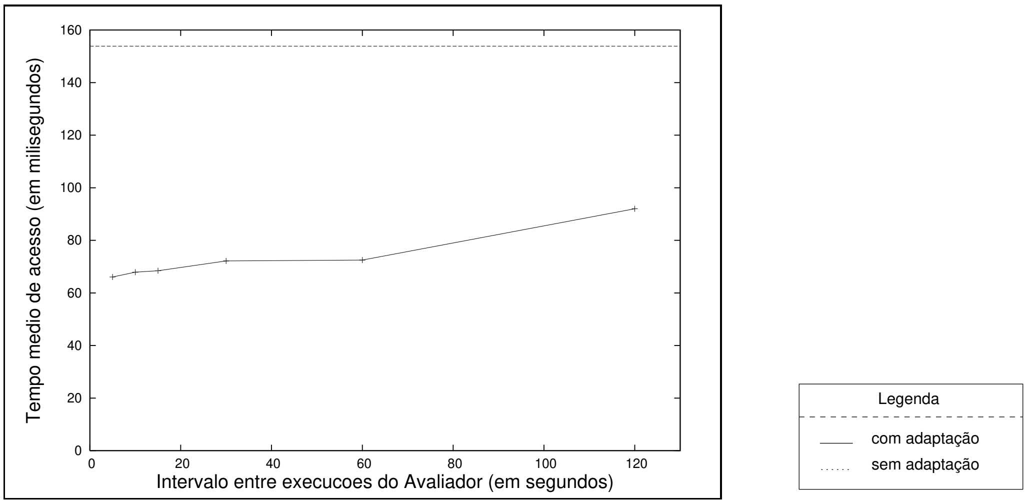

Figura 7.7: Resultados do quarto cenário

	SA	5	10	15	30	60	120
média	153,87	66,06	67,88	68,43	72,18	72,50	92,02
mínimo	151,47	56,61	55,59	58,31	60,36	63,38	79,56
máximo	158,49	81,08	92,62	80,06	86,05	85,99	99,92
desvio padrão	1,62	5,61	7,03	5,71	5,92	5,10	5,09
int confiança 95%	153,24	64,05	65,37	66,39	70,02	70,61	90,13
	154,51	68,07	70,40	70,48	74,33	74,39	93,90

Tabela 7.7: Dados relativos às execuções do quarto cenário

7.2 Variações da Disponibilidade de Recursos

Utilizamos o arcabouço para incorporar mecanismos de adaptação dinâmica baseados na disponibilidade de recursos ao PSIT. Definimos um evento denominado `ServerOverloaded` que permite identificar a sobrecarga de máquinas pertencentes à rede fixa de acordo com o consumo de UCP e da disponibilidade de memória principal nas mesmas, conforme descrito na seção 6.5.2.

Nos testes realizados, utilizamos 4 Servidores de Informação. Através de um utilitário que aumenta o consumo de UCP em uma máquina, forçamos a ocorrência de eventos `ServerOverloaded`. Em resposta a ocorrência deste evento, a versão adaptativa do PSIT reinstanciou, em um outro computador pertencente à mesma região, um dos Servidores de Informação da máquina sobrecarregada, aliviando, assim, sua carga de trabalho. Testamos também a alternância da estratégia de adaptção utilizada (*First Fit* e *Best Fit*).

O objetivo dos testes realizados não foi o de medir ganhos de desempenho obtidos com o uso da adaptação, dado que Servidores de Informação consomem pouca UCP. Nosso objetivo central foi avaliar a aplicabilidade dos mecanismos disponibilizados pelo arcabouço para a monitoração, detecção e reconfiguração baseadas na análise da disponibilidade de recursos. Como objetivo secundário, quantificamos o tempo necessário para realizar as seguintes atividades:

1. tempo transcorrido entre o início do consumo de UCP em uma máquina, a detecção da alteração da faixa de operação do parâmetro que mede o percentual de uso da UCP e sua notificação ao Gerente de Recursos;

2. tempo transcorrido entre o recebimento da notificação de alteração da faixa de uso da UCP em uma máquina pelo Gerente de Recurso até a notificação do evento `ServerOverloaded`;

3. tempo transcorrido entre o recebimento da notificação do evento `ServerOverloaded` até a escolha da máquina candidata à reinstanciação de um Servidor de Informação;

4. tempo necessário para realizar a reinstanciação do Servidor de Informação.

A tabela 7.8 apresenta os resultados obtidos em segundos.

Nos testes realizados, o intervalo entre as verificações do percentual de uso da UCP utilizado pelos OMRs instanciados era de 2 segundos, o que explica o valor de 2,2174 segundos apresentado

1. Detecção e notificação da mudança de faixa de operação	2,2174
2. Detecção e notificação do evento `ServerOverloaded`	15,551
3. Escolha da máquina candidata	
First Fit	0,057
Best Fit	0,126
4. Reconfiguração da aplicação	0,251

Tabela 7.8: Resultados dos experimentos com disponibilidade de recursos

no primeiro item da tabela 7.8.

Ao definirmos o evento `ServerOverloaded`, atribuimos o valor de 15 segundos ao seu tempo de duração. A expressão booleana do evento deve permanecer verdadeira durante este período de tempo para que o Gerente de Recursos realize a notificação de sua ocorrência, conforme descrito na seção 4.2.1. Este é o motivo pelo qual o tempo para detecção e notificação do evento foi de 15,551 segundos.

7.3 Considerações Sobre os Experimentos Realizados

O nosso principal objetivo ao utilizarmos o arcabouço na incorporação de mecanismos para adaptação dinâmica ao protótipo do projeto SIDAM foi validar sua aplicabilidade e aderência aos requisitos de projeto descritos na seção 2.3.

Nossa avaliação final foi positiva já que o arcabouço efetivamente simplificou o processo de desenvolvimento dos mecanismos para adaptação dinâmica da aplicação de referência do SIDAM, tanto por simplificar a definição da arquitetura da solução, dado que o mesmo define seus principais componentes e o modelo de colaboração, quanto por promover a reutilização dos componentes disponibilizados pelo arcabouço, reduzindo consideravelmente o esforço em programação. Além disto, o arcabouço mostrou-se flexível, permitindo a especialização de seus componentes às necessidades específicas da aplicação.

Os resultados dos testes apresentados neste capítulo reforçam a aplicabilidade do arcabouço, além de evidenciar a importância de levarmos em consideração a análise das interações entre os componentes da aplicação como forma de identificar a necessidade de reconfiguração da aplicação.

Um aspecto observado ao realizarmos os testes é que o uso de uma ferramenta gráfica para o gerenciamento da execução dos componentes do arcabouço facilitaria sua utilização. Através desta ferramenta, o usuário poderia observar quais componentes estão sendo executados nos nós do ambiente e utilizaria as interfaces de gerenciamento dos mesmos para proceder atualizações em tempo de execução que se fizerem necessárias, além de poder instanciar e interromper a execução destes componentes. Nos testes utilizamos *shell scripts* do Unix para realizar estas tarefas.

Capítulo 8

Trabalhos Relacionados

Pesquisas relacionadas ao desenvolvimento de software adaptativo ganharam impulso a partir de meados da década de 90 [ACM95]. Com o desenvolvimento da computação móvel e mais recentemente das pesquisas em curso visando o desenvolvimento de espaços ativos [MRN02, KHR+00], o ambiente de computação tornou-se altamente dinâmico. Estes ambientes têm exigido um grande investimento em pesquisas que visam desenvolver mecanismos que permitam ao software adaptar-se dinamicamente a variações no ambiente de execução.

Mecanismos para adaptação dinâmica têm sido empregados em diferentes contextos: na área de sistemas operacionais, para o desenvolvimento de sistemas de arquivos que apresentem bom desempenho mesmo sofrendo variações no padrão da carga de trabalho [MRC+97] e para substituição em tempo de execução de seus componentes [CAK+96]; para a construção de plataformas de *middleware* flexíveis, capazes de acomodar os requisitos impostos tanto pelo ambiente de execução quanto por uma grande variedade de aplicações como aplicações multimídia, de tempo real, de trabalho cooperativo e de ambientes móveis, especializáveis para execução em dispositivos que variam de PDAs a computadores de mesa e multicomputadores [KCCB02, BCA+01, BBI+00, CDP+00, KRL+00]; em ambientes de computação móvel, para permitir a inclusão ou remoção de componentes de hardware e para a detecção e utilização ótima de diferentes tecnologias de rede à medida que o usuário se desloca com seu dispositivo [IBW97], bem como para detecção e interação com serviços e recursos descobertos dinamicamente [SAW94, BGI99] e suporte à variações na disponibilidade de recursos (como, por exemplo, a largura de banda disponível para comunicação) bem como em aplicações de tempo real de domínio complexo, onde é necessário que o sistema se reconfigure dinamicamente com relação ao conjunto de reações possíveis para a situação corrente do ambiente [MGPK99, OGT+99].

Este capítulo apresenta alguns trabalhos que possuem uma relação direta com o objetivo de nossa pesquisa, que é o desenvolvimento de um modelo e da infra-estrutura de suporte para construção de aplicações adaptativas distribuídas. Após um breve resumo de cada trabalho, realiza-se uma análise comparativa com o modelo proposto nesta tese.

8.1 Sudame e Badrinath

Sudame e Badrinath [SB97] descrevem um mecanismo baseado em mensagens ICMP [Com00] para propagação de informações a respeito da condição de parâmetros do ambiente de rede como, por exemplo, o custo financeiro da comunicação, latência e largura de banda. Este mecanismo pode ser estendido para acomodar parâmetros de recurso específicos de ambientes de computação móvel como a intensidade do sinal, localização do dispositivo e a reserva de energia. De acordo com os valores destes parâmetros, propagados através de mensagens ICMP, ações podem ser disparadas automaticamente para ajustar parâmetros do protocolo de transporte sendo utilizado (TCP ou UDP) de forma a permitir uma adaptação à condição atual do ambiente de rede. Aplicações também podem ser notificadas das variações detectadas no ambiente. Cada parâmetro é associado a um conjunto de níveis (valores). Cada nível corresponde a uma marca alta ou baixa, conforme ilustrado na figura 8.1. Um nível com uma marca alta é violado quando o novo valor para o parâmetro o cruza de baixo para cima. Um nível com uma marca baixa é violado quando este cruzamento se dá de cima para baixo.

Na figura 8.1 quando o parâmetro muda de valor de $v1$ para $v2$, os níveis C e A são violados. Quando a variação se dá de $v3$ para $v4$, o nível B é violado. O mecanismo utiliza uma nova mensagem ICMP, denominada ICMP STATUS, para propagar informação sobre o ambiente toda vez que um nível é violado. O mecanismo provê um tratador (*handler*) para esta mensagem. Este realiza uma chamada a uma função que dispara ações que ajustam parâmetros específicos do protocolo de transporte sendo utilizado (TCP ou UDP). O tratador também notifica as aplicações que tenham registrado interesse em receber mensagens relativas ao parâmetro em questão. O conjunto de ações

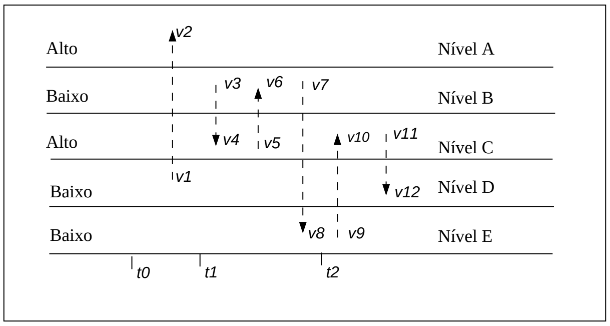

Figura 8.1: Adaptação de protocolo de rede em dispositivos móveis

disponíveis é pré-definido. As ações específicas a serem tomadas no caso de violações dos níveis dos parâmetros sendo monitorados são definidas pelo usuário e ficam armazenadas em uma estrutura denominada tabela de ações. Um exemplo de ação pré-definida é a `UDP_BUFFER`, que passa a armazenar em um *buffer* os datagramas UDP transmitidos pela aplicação quando a rede tornar-se indisponível.

Este trabalho apresenta um mecanismo que permite a adaptação da camada de transporte a variações detectadas no ambiente de rede, através de ações que ajustam parâmetros dos protocolos TCP e UDP. Além disto, as aplicações interessadas podem receber notificações das variações detectadas. No entanto, não há nenhum suporte à reconfiguração dos componentes da aplicação. Além disto, o trabalho é focado na adaptação do protocolo de transporte, sendo o conjunto de ações de adaptação pré-definido e baseado apenas em ajuste de parâmetros. Tanto a monitoração quanto notificação e a adaptação são executadas localmente em uma máquina, não contemplando um ambiente distribuído. É prevista apenas a monitoração de recursos do ambiente, não sendo levada em consideração a monitoração das interações entre objetos da aplicação.

8.2 Proxy Ativo

Fox et al. [FGBA96, FGCB98] apresentam uma abordagem para o problema de como tolerar variações na largura de banda e na arquitetura de software e hardware dos clientes. O enfoque deste trabalho está em aplicações de acesso a dados na Internet. O modelo de adaptação é baseado nos seguintes princípios:

- a adaptação é baseada na compressão dos dados transmitidos do servidor para o cliente; Esta compressão é realizada por componentes denominados destiladores (*distillers*), que são específicos para cada tipo de dado a ser transformado. Ao realizar a transformação, os destiladores preservam o conteúdo semântico dos dados. Como exemplo, pode-se ter um destilador que transforma um arquivo *PostScript* para formato texto (ASCII);

- a adaptação é realizada somente no momento da solicitação do dado pelo cliente. Assim, o servidor não armazena versões diferentes de um mesmo dado;

- a complexidade inerente à adaptação não deve ser incluída nem no programa cliente nem no servidor. Isto é possível realizando-se a adaptação em um componente intermediário denominado *proxy* que é compartilhado tanto pelos clientes quanto pelo servidor.

Conforme ilustrado na figura 8.2, um *proxy* é inserido entre o cliente e o servidor. Ele fica localizado próximo ao servidor e é conectado a este através de uma conexão que possui grande largura de banda e baixa latência de comunicação. Sua função é recuperar conteúdo de servidores Internet a partir de solicitações realizadas pelos clientes, determinar os tipos de dados que compõem a resposta à solicitação (texto, figuras, etc.) e determinar quais destiladores devem ser empregados de forma a adaptar este conteúdo à largura de banda disponível para a comunicação com o cliente bem como à arquitetura de hardware e software do mesmo.

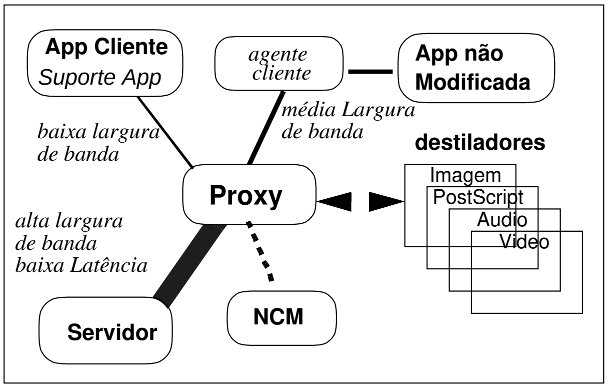

Figura 8.2: Arquitetura de *Proxy* Ativo

Dois parâmetros são utilizados para disparar a adaptação: a largura de banda disponível para a comunicação com o cliente e a arquitetura de hardware e software que ele possui. O *proxy* escolhe quais destiladores devem ser empregados de acordo com o valor destes parâmetros. A arquitetura de hardware e software pode ser armazenada em um banco de dados caracterizando, assim, o perfil de cada dispositivo usuário do sistema. Esta informação é recuperada pelo *proxy* quando necessário, como sugerido em [FGCB98]. O segundo parâmetro, largura de banda disponível para a comunicação com o cliente, varia dinamicamente. Sua monitoração é realizada pelo *proxy*.

Este trabalho considera que a aplicação é estruturada em dois componentes, o cliente e o servidor, sendo um *proxy* inserido entre eles. O modelo proposto nesta tese é mais geral, considerando o caso onde a aplicação é composta por diversos componentes distribuídos em vários nós do ambiente. Em Proxy Ativo a adaptação consiste unicamente na transformação do dado sendo transmitido ao cliente, realizado pelos destiladores. Consideramos que a flexibilidade com relação aos mecanismos de adaptação que a aplicação pode adotar, como a escolha de algoritmos alternativos, alteração de parâmetros utilizados na computação, migração e replicação de componentes, entre outros, são fundamentais para lidar com a variedade de ambientes e aplicações existentes.

8.3 Odyssey

Odyssey [NPS95, NSN$^+$97, NS99] é um conjunto de extensões ao sistema operacional NetBSD com o objetivo de prover suporte à adaptação de aplicações de acesso a informações requisitadas por dispositivos móveis. Estas aplicações no dispositivo móvel lêem e atualizam dados armazenados em servidores remotos.

Apesar de possuir uma abordagem diferente da utilizada em *proxy ativo*, Odyssey utiliza o mesmo princípio de transformar o fluxo de dados sendo transmitido ao cliente de forma a tolerar variações no ambiente. Adaptação é definida como uma troca da qualidade do dado pelo consumo de recursos. Para quantificar esta noção de qualidade, Odyssey define uma propriedade denominada **fidelidade**. Para cada item de dado, existe uma versão mais atual, completa e detalhada do mesmo denominada **cópia de referência**. Em uma situação ideal, a cópia de referência seria apresentada ao cliente em resposta a uma requisição. No entanto, à medida que recursos (por exemplo, largura de banda) vão se tornando escassos, o item de dado é degradado de alguma forma. Fidelidade é definida como o grau de similaridade entre o item de dado sendo apresentado e a cópia de referência.

Na visão de Odyssey, o sistema operacional é responsável por monitorar a disponibilidade de recursos, notificar as aplicações de mudanças significativas na condição dos mesmos e tomar decisões de forma a otimizar a utilização dos mesmos. Cada aplicação, por sua vez, é responsável por decidir a melhor forma de explorar os recursos disponíveis. Considera-se que o dispositivo móvel pode estar executando diversas aplicações concorrentemente. A figura 8.3 ilustra a arquitetura de Odyssey.

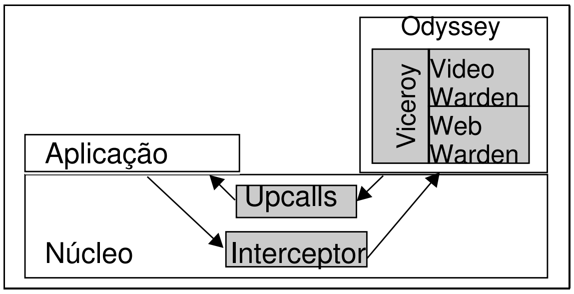

Figura 8.3: Arquitetura de Odyssey

Objetos Odyssey são disponibilizados às aplicações como elementos do sistema de arquivos. O *interceptor* redireciona requisições ao sistema de arquivos para um componente denominado *viceroy*. A tarefa mais importante do *viceroy* é a monitoração da disponibilidade de recursos e o gerenciamento do seu uso. Os *wardens* formam um conjunto de componentes específicos para cada tipo de dado disponibilizado pelo servidor. Eles disponibilizam um menu de fidelidades que as aplicações podem escolher. *Wardens* utilizam seu conhecimento sobre o tipo de dado sendo solicitado para realizar otimizações na utilização de recursos, prover consistências, etc.

Odyssey adiciona duas chamadas à API padrão do sistema operacional. A primeira, `resource request`, é utilizada pela aplicação para informar uma solicitação de recurso. Exemplos de recursos são a largura de banda para a comunicação com o servidor, latência da comunicação, poder computacional, espaço em disco, energia restante no dispositivo e custo. A aplicação informa nos parâmetros desta chamada a identificação do recurso, os limites inferior e superior de disponibilidade do mesmo necessários à execução da aplicação e um tratador (*handler*) para uma função a ser chamada quando a disponibilidade do recurso ultrapassar os limites solicitados. Esta solicitação é encaminhada ao *viceroy* que monitora todos os recursos do sistema. Se a disponibilidade de um recurso sair de algum dos limites solicitados, o *viceroy* notifica as aplicações afetadas através de uma chamada (*upcall*) à função informada pelas mesmas. A segunda chamada, `type-specific`, é utilizada pela aplicação para modificar a fidelidade na qual o dado solicitado é acessado. Por exemplo, quando uma aplicação para a apresentação de um vídeo for notificada que a largura de banda disponível caiu significativamente, ela pode reduzir a quantidade de quadros sendo apresentados por segundo.

Assim como ocorre em Proxy Ativo (8.2), a aplicação adaptativa é dividida em dois componentes: o cliente e o servidor. A adaptação é baseada na escolha entre diversas versões dos dados apresentados (níveis de fidelidade). Reiteramos a importância da flexibilidade com relação aos mecanismos de adaptação que podem ser adotados.

8.4 Chang e Karamcheti

Chang e Karamcheti [CK00] descrevem um arcabouço cujo objetivo central é eliminar a necessidade de se programar explicitamente decisões relativas à adaptação no código das aplicações. O arcabouço é baseado em dois componentes: uma interface de adaptação e um ambiente de execução virtual. A interface de adaptação disponibiliza o suporte na linguagem de programação para expressar a disponibilidade de múltiplos caminhos de execução (algoritmos diferentes para realizar uma mesma tarefa) bem como definir parâmetros relativos à monitoração do progresso da execução da aplicação. A partir desta monitoração, pode-se escolher qual caminho de execução (algoritmo) deve ser seguido. As informações sobre a interface de adaptação são inseridas pelo desenvolvedor como anotações no código fonte que são processadas por um pré-processador de macros. Este pré-processador gera agentes de monitoração e controle, que monitoram a disponibilidade de recursos e controlam a execução da aplicação, respectivamente. O código destes agentes é inserido no código da aplicação.

O segundo componente do arcabouço é um ambiente de execução virtual que pode ser configurado para emular diversos cenários de disponibilidade de recursos. Um *script* executa repetidamente a aplicação nestes ambientes virtuais, obtendo o perfil do comportamento da aplicação para cada um destes cenários. O sistema de suporte à execução utiliza estes perfis para tomar decisões sobre qual caminho seguir de forma a satisfazer as preferências definidas pelo usuário como, por exemplo, a otimização do tempo de transmissão dos dados entre o cliente e o servidor.

A figura 8.4 ilustra os componentes utilizados durante a execução de uma aplicação. O agente

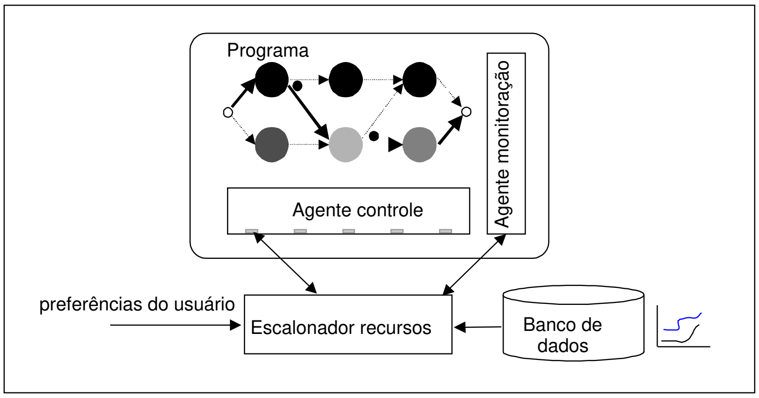

Figura 8.4: Chang e Karamcheti: componentes de execução

de monitoração observa o progresso da aplicação e estima a fração dos recursos requisitados que estão disponíveis para a aplicação. Um escalonador de recursos correlaciona as preferências do usuário com a fração dos recursos de interesse disponíveis e o banco de dados de perfis de execução para escolher o melhor caminho (algoritmo) a ser utilizado nesta situação. Finalmente, o agente de controle (*steering agent*) realiza a reconfiguração, que é a troca do algoritmo sendo utilizado. Como exemplo de uma aplicação que pode utilizar algoritmos diferentes para realizar adaptação, pode-se citar uma aplicação para distribuição de vídeo. A aplicação pode escolher entre diversos algoritmos de compressão, como LZW e Bzip2. O primeiro comprime de maneira mais eficiente os quadros que compõem o vídeo comparativamente ao segundo. No entanto, seu consumo de processamento é maior. A aplicação pode alternar entre estes algoritmos de acordo com a condição corrente dos parâmetros disponibilidade de processador e largura de banda.

A abordagem seguida por Chang e Karamcheti difere da nossa em diversos aspectos. Em [CK00], todos os diferentes caminhos (algoritmos) estão codificados em um processo monolítico, juntamente com os agentes de monitoração e controle. Em nossa abordagem, os componentes responsáveis pela monitoração do ambiente e a análise dos dados coletados residem em processos separados do código da aplicação. Em [CK00], a aplicação é composta por um único processo enquanto nosso trabalho é voltado para aplicações que podem ser compostas por diversos componentes distribuídos em uma rede. Chang e Karamcheti também não levam em consideração a interação entre os componentes da aplicação na tomada de decisão sobre a adaptação.

8.5 Conductor

Conductor [YRP99, YRP00], é um arcabouço que visa prover adaptação com relação à heterogeneidade dos enlaces e variações na condição de operação das redes de computadores. De forma semelhante ao que ocorre em Proxy Ativo e no Odyssey, a adaptação se dá pela transformação dos dados sendo transmitidos. No entanto, ao contrário das abordagens vistas anteriormente, Conductor permite que a adaptação se dê de forma coordenada em diversos pontos da rede.

As redes de computadores estão se tornando cada vez mais complexas. O fluxo de dados resultante da comunicação entre componentes de uma aplicação executados em máquinas distintas pode ter que passar por diversos enlaces heterogêneos como, por exemplo, rede de telefonia celular, rede metropolitana de alta velocidade, rede local IEEE 802.11 [Sta02], etc. Este cenário tornou-se ainda mais acentuado com o surgimento da computação móvel. Conductor distribui a adaptação por diversos nós da rede, de forma a levar em consideração as características de cada enlace e, assim, obter um melhor desempenho global.

O serviço de adaptação de Conductor deve estar disponível em vários nós da rede, preferencialmente naqueles próximos a *gateways* entre redes de diferentes tipos (Ethernet, FDDI, IEEE 802.11, etc.). Conductor consiste de dois componentes principais: os adaptadores e o arcabouço para distribuição dos mesmos.

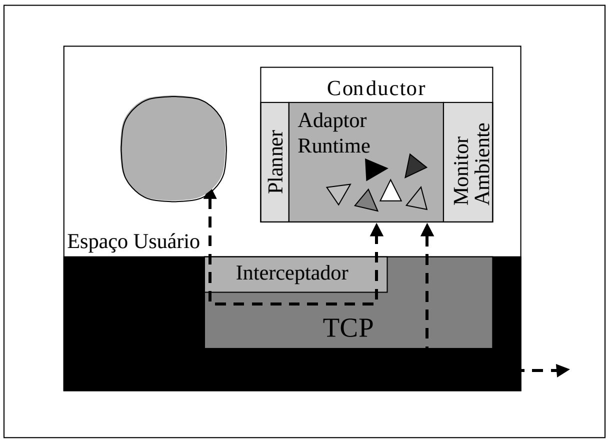

Figura 8.5: Arquitetura Conductor em um nó

Adaptadores são componentes de software que realizam alguma adaptação, normalmente específica para um tipo particular de fluxo de dados. O conjunto de adaptadores pode ser expandido. Adaptadores freqüentemente (mas não necessariamente) são disponibilizados aos pares, onde um deles converte o fluxo de dados de um dado protocolo para outro que utilize melhor o meio de transmissão e outro que reconverte o fluxo de dados para o protocolo original. Exemplos de adaptadores são um compressor e decompressor MPEG para vídeo (um par) ou um conversor que transforme uma imagem colorida para preto-e-branco (adaptador isolado).

Conductor provê um arcabouço para realizar operações sobre adaptadores. A figura 8.5 ilustra a arquitetura de Conductor em um nó. Esta arquitetura consiste de um módulo que executa no espaço usuário e uma pequena modificação do núcleo do sistema operacional. O módulo Conductor monitora o fluxo de dados, efetiva a entrega destes dados a adaptadores locais, realiza a transmissão de dados entre nós Conductor, faz o planejamento para novos fluxos de dados e possui mecanismos para garantir confiabilidade da transmissão e recuperação em caso de falhas. O núcleo incorpora uma pequena modificação, realizada com o objetivo de interceptar o fluxo de dados para que este possa ser examinado por Conductor.

Quando um novo fluxo de dados é iniciado por uma aplicação, Conductor intercepta a abertura de seu *socket* TCP e examina as informações a respeito do mesmo (e, eventualmente, os primeiros bytes transmitidos) a fim de determinar qual é o formato de dados sendo transmitido. Conductor, então, percorre o caminho de roteamento, consultando os nós Conductor existentes no mesmo, obtendo informações a cerca das condições locais da rede e de capacidade dos nós. Finalmente, é criado um plano que determina quais adaptadores devem ser utilizados e onde eles devem ser localizados de forma a otimizar o fluxo dos dados pela rede. A figura 8.6 ilustra um caminho com diversos adaptadores.

Conductor possibilita que a adaptação se dê em diversos nós de um ambiente distribuído, não estando restrito a um único ponto de adaptação, um aspecto que também levamos em consideração no nosso modelo. No entanto, a adaptação é restrita a transformações no fluxo de dados trocados por componentes da aplicação. Conductor foi projetado com o objetivo específico de compensar diferenças decorrentes da heterogeneidade dos enlaces da rede e no estado de operação da mesma enquanto o modelo proposto é mais geral. Em Conductor, toda adaptação é relacionada à comunicação entre os componentes da aplicação, não sendo oferecido suporte para lidar com a

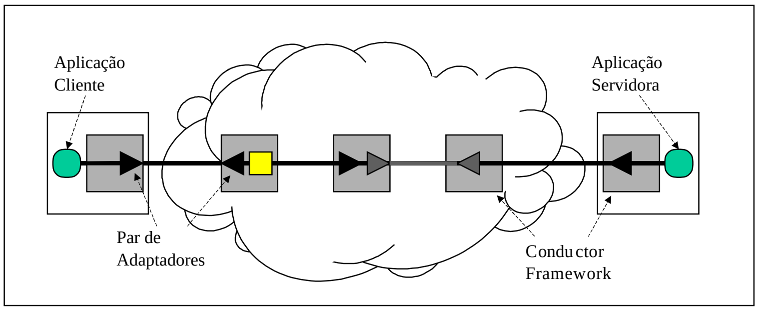

Figura 8.6: Adaptadores distribuídos de Conductor

reconfiguração dinâmica de componentes da aplicação. A análise das interações entre os mesmos também não é considerada na tomada de decisão para adaptação.

8.6 Moura et al.

Moura et al. [dMUCR02] apresentam um ambiente que tem por objetivo oferecer um conjunto de mecanismos que dêem apoio a adaptações dinâmicas de aplicações distribuídas através da seleção de componentes de acordo com a qualidade do serviço por eles oferecido. O ambiente é baseado em LuaCorba [Cer00], responsável pela associação (*binding*) entre CORBA [OMG02] e a linguagem Lua [IdFC96], desenvolvida na PUC-Rio.

O ambiente proposto provê às aplicações as seguintes facilidades e mecanismos:

- Seleção dinâmica dos componentes que melhor atendam aos requisitos da aplicação. Estes requisitos são expressos como um conjunto de propriedades não funcionais que caracterizam a qualidade dos serviços que estes componentes provêem;
- Monitoração destas propriedades, de forma a verificar se os requisitos da aplicação estão sendo satisfeitos pelo sistema ao longo de sua execução;
- Reação a mudanças relevantes nas propriedades, através da ativação de estratégias de adaptação apropriadas.

Para prover estas facilidades de uma forma transparente ao comportamento funcional da aplicação, utiliza-se a abordagem de *smart proxies* [ZBS97, BG97]. Nesta abordagem, a aplicação cliente não está vinculada a servidores específicos, acessando serviços através do *smart proxy*, conforme ilustrado na figura 8.7. O *smart proxy* encapsula as estratégias utilizadas na seleção dinâmica de componentes, monitoração e adaptação, simplificando o código da aplicação que se limita aos aspectos funcionais da mesma.

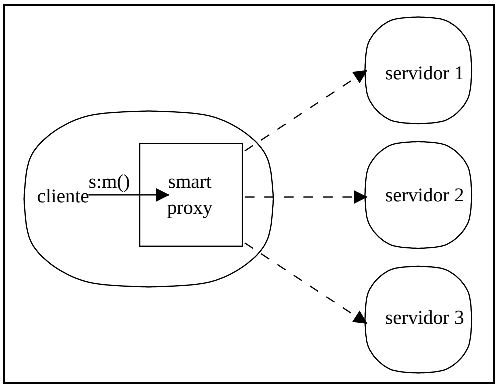

Figura 8.7: *Smart Proxy*

A seleção em tempo de execução de qual servidor atenderá a requisição do cliente é realizada pelo *smart proxy* e é baseada em consultas ao serviço de negociação (*Trading*, definido na especificação CORBA) que armazena propriedades não funcionais, como o tamanho da fila de processos prontos para executar de um processador ou o tempo médio de resposta de um dado servidor. Os valores correntes destas propriedades são obtidos através de objetos do tipo *LuaMonitor*, que observam propriedades específicas. A figura 8.8 ilustra os componentes do ambiente proposto. Os agentes de serviço são elementos responsáveis por anunciar ofertas de serviço ao *Trader*.

LuaMonitor define um tipo de objeto denominado *monitor* que representa uma propriedade específica sendo observada, como, por exemplo, o tempo de resposta associado a uma operação definida por um servidor. Um objeto pré-definido, `BasicMonitor`, implementa funcionalidades básicas provendo dois métodos, `getValue` e `setValue`, através dos quais pode-se obter e modificar o valor corrente da propriedade monitorada. Em alguns casos, uma aplicação pode estar interessada não somente no valor específico de uma propriedade, mas também em estatísticas ou no perfil de evolução da mesma. Para acomodar estes casos, foram incorporadas facilidades para a definição e obtenção de aspectos e qualificadores de uma propriedade [FK98, RCS99].

Para facilitar a implementação de estratégias de monitoração baseadas em eventos, definiu-se

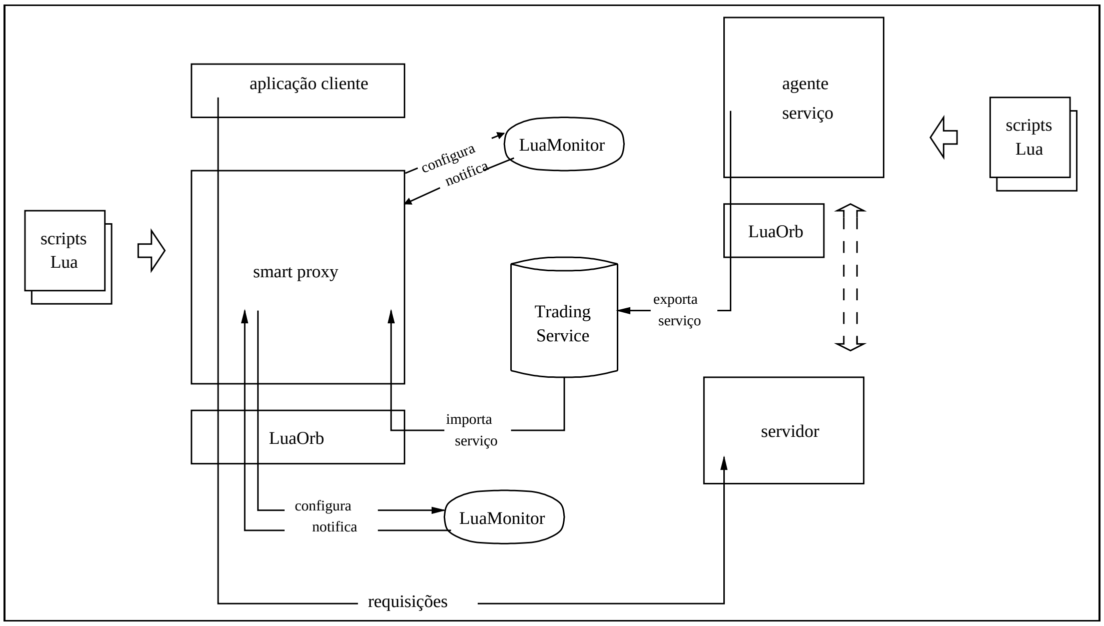

Figura 8.8: Ambiente para adaptação dinâmica baseado em LuaCorba

uma extensão do objeto de monitoração que implementa a interface `EventMonitor`. Esta interface permite que objetos interessados na propriedade sendo monitorada se registrem como observadores, definindo eventos específicos nos quais possuem interesse. Desta forma, aplicações podem ser notificadas quando houver modificações específicas no estado da propriedade monitorada. Tipicamente serão objetos *smart proxies* que se registrarão como observadores destas propriedades.

LuaMonitor define um modelo de monitoração de recursos que possui muitas semelhanças com os OMRs (Objetos de Monitoração de Recursos), definidos no nosso modelo. Ambos são extensíveis e permitem tanto o modelo *pull* quanto o modelo *push*, este último utilizado na monitoração baseada em eventos. Uma diferença significativa é que em *LuaMonitor* a definição de eventos e o registro de observadores são realizados juntos ao próprio objeto de monitoração. No nosso modelo, um componente denominado Gerente de Recursos (4.3) é responsável pela detecção e notificação da ocorrência de eventos. A estratégia seguida por *LuaMonitor* faz com que os objetos de monitoração sejam mais complexos e consumam mais recursos na sua execução. Além disto, eventos baseados no estado de mais de um tipo de recurso são mais difíceis de implementar. A solução consiste em fazer o *smart proxy* se registrar junto aos objetos de monitoração requeridos, devendo também conter a lógica responsável pela detecção do evento composto. Como o *smart proxy* deve ser escrito pelo usuário, isto implica em trabalho adicional e maior complexidade para implementar os mecanismos de adaptação da aplicação. No nosso modelo, eventos dependentes da condição de diversos recursos do ambiente podem facilmente ser definidos sem a necessidade de programação por parte do usuário.

O ambiente proposto em [dMUCR02] não possui mecanismos para monitorar as interações realizadas pelos componentes da aplicação nem para detectar eventos baseados na análise destes dados. Também não é oferecido suporte à reconfiguração de componentes da aplicação localizados em diferentes nós do ambiente.

8.7 QuO

QuO [RJJ+98, BBN02b, BBN02a] é um arcabouço cujo objetivo é permitir o desenvolvimento de aplicações baseadas em objetos distribuídos que apresentam requisitos de QoS. QuO disponibiliza um ambiente através do qual o programador pode especificar possíveis níveis de qualidade de serviço, os elementos do sistema que necessitam ser monitorados e controlados para que se possa medir e prover QoS e o comportamento necessário para adaptar a aplicação face a mudanças no estado dos recursos disponíveis.

Além do objeto cliente, ORB e objeto servidor, uma aplicação QuO é composta pelos seguintes elementos ilustrados na figura 8.9:

- Delegado (*delegate*) do objeto servidor: o delegado disponibiliza uma interface funcional idêntica ao objeto remoto mas pode disparar avaliações do estado dos recursos disponíveis a cada chamada de método ou em seu retorno. Esta avaliação é realizada através de um contrato, definido a seguir. Os desenvolvedores de aplicações podem implementar comportamentos alternativos e uma estrutura de controle através da qual escolhe-se dentre estas alternativas qual deve ser utilizada face ao estado atual do ambiente. Como exemplo de comportamentos alternativos, temos o armazenamento em um *buffer* das chamadas realizadas ou a escolha de métodos alternativos disponibilizados pelo servidor;

- Contrato (*contract*) entre o cliente e o servidor: especifica o nível de serviço desejado pelo cliente, o nível de serviço que o servidor pretende prover, regiões aninhadas que descrevem os

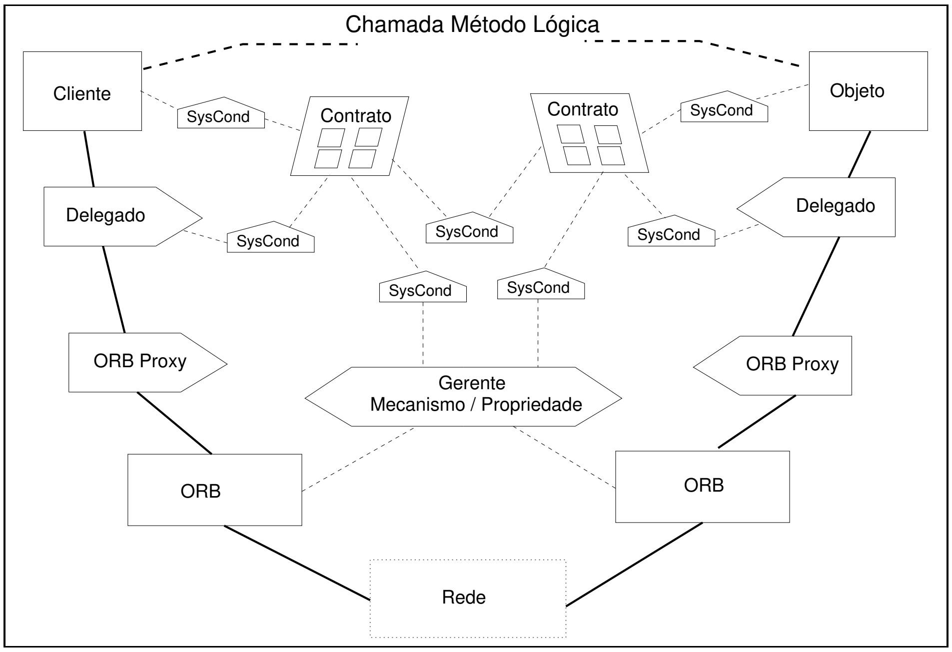

Figura 8.9: Invocação de método em uma aplicação QuO

possíveis estados de QoS do sistema e ações a serem realizadas no caso de mudança no nível de QoS;

- Objetos de Condição do Sistema (*system condition objects*), utilizados para medir e controlar a QoS. Cada Objeto de Condição monitora uma única propriedade do sistema, como o tempo transcorrido desde a última chamada de método realizada ou o número de processos ativos em um nó remoto. Objetos de Condição compartilham uma interface comum que disponibiliza o método `getValue`, usado para retornar o estado atual da propriedade monitorada.

Quando o objeto cliente realiza uma chamada de método remota, esta chamada é repassada ao Delegado local. O Delegado pode, então, realizar uma avaliação do Contrato. Esta avaliação verifica os valores atuais do estado do sistema medidos pelos Objetos de Condição.

O Contrato consiste de um conjunto de regiões aninhadas que descrevem os possíveis estados de QoS do sistema. Cada uma destas regiões é definida por um predicado contendo valores monitorados pelos Objetos de Condição. O Contrato avalia estes predicados para determinar que regiões encontram-se ativas (ou seja, que possuem predicados avaliados como verdadeiro) e passa uma lista das mesmas para o Delegado.

O Delegado escolhe como processar a chamada de método baseado nas regiões ativas e pode, por exemplo, escolher entre métodos alternativos, bloquear-se, optar por armazenar em um *buffer* a chamada quando a QoS estiver degradada ou simplesmente repassar a chamada ao objeto servidor. O Delegado realiza um processamento similar ao receber o retorno da chamada. Ou seja, ele avalia o Contrato para obter as regiões ativas de QoS e seleciona o comportamento a ser utilizado baseado nestas regiões. A figura 8.10 ilustra os passos realizados em uma chamada de método realizada por uma aplicação QuO.

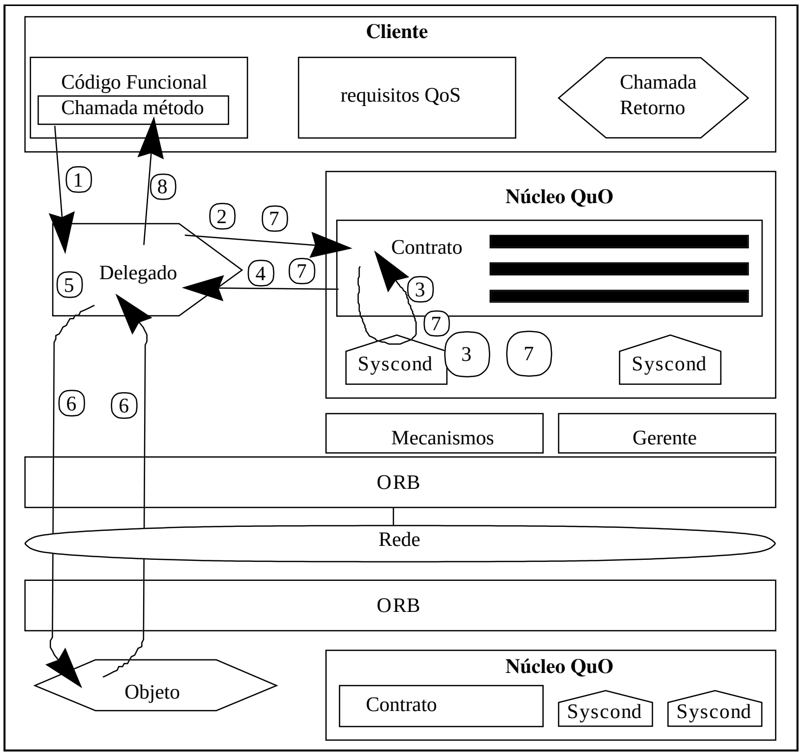

Figura 8.10: Passos de uma chamada de método em uma aplicação QuO

1. Cliente realiza uma chamada de método que é entregue ao Delegado;
2. O Delegado avalia o Contrato;
3. O Contrato obtém os valores correntes das condições do sistema e determina as regiões ativas;
4. O Contrato retorna as regiões ativas;

5. O Delegado escolhe o comportamento a ser adotado baseando-se nas regiões devolvidas. O comportamento padrão é passar a chamada de método ao objeto remoto;

6. O objeto remoto é chamado e devolve o valor da chamada de método realizada;

7. O Delegado reavalia o Contrato que obtém os valores atuais das condições do sistema e devolve as regiões ativas;

8. O Delegado escolhe o comportamento baseado nas regiões devolvidas. O comportamento padrão é devolver o valor ao cliente.

O Contrato é também avaliado sempre que houver uma variação no estado dos recursos monitorados por Objetos de Condição do Sistema que tenham sido definidos como observadores. Estes são utilizados na monitoração de recursos cujo estado varia com baixa freqüência. Objetos de Condição que monitoram recursos cujo estado varia com muita freqüência são definidos no contrato como não-observadores e, por isso, não disparam avaliações automáticas de Contrato.

A avaliação do Contrato pode determinar a ocorrência de transição de uma região ativa para outra. Esta transição pode disparar uma chamada de método ao cliente (*callback*) ou chamadas de método a um ou mais Objetos de Condição definidos no Contrato.

O arcabouço QuO consiste dos seguintes componentes:

1. Um conjunto de linguagens denominado QDL - *Quality Description Languages* que permite descrever Contratos, Objetos de Condição do sistema e o comportamento de Delegados e transições;

2. O núcleo de QuO, que coordena a avaliação de Contratos e a monitoração realizada pelos Objetos de Condição;

3. Geradores de código, que encapsulam o código descrito através de QDL, o núcleo e o código da aplicação em um único programa executável [1].

O mecanismo de adaptação adotado em QuO é fortemente baseado na interceptação das interações entre os componentes cliente e servidor da aplicação. No entanto, assim como ocorre nos

[1] a partir da versão 3.0, pode-se, opcionalmente, executar o núcleo em um processo independente [BBN02b].

demais trabalhos descritos, os dados (parâmetros e resultados devolvidos) relativos às interações não são levados em consideração na tomada de decisão de quando e como adaptar a aplicação.

Tipicamente, os Objetos de Condição definidos no contrato de uma aplicação QuO realizarão a monitoração de recursos presentes no caminho entre o objeto cliente e o servidor. Em nosso trabalho, o modelo de monitoração permite uma visão global dos recursos do ambiente distribuído. Isto permite a tomada de decisão sobre reconfigurações na aplicação (como migração ou replicação de componentes) baseada na análise de recursos não pertencentes ao caminho de invocação entre componentes.

QuO permite que a aplicação se torne ciente do estado do ambiente de execução através de chamadas realizadas pelo código definido para as transições constantes do Contrato. Além disto, mudanças de comportamento da aplicação podem ser inseridas no código do Delegado, ciente do estado do ambiente através da lista de regiões ativas devolvidas pela avaliação do Contrato. No entanto, a adaptação é realizada localmente, em cada componente da aplicação. Não há uma representação explícita das dependências entre os componentes da aplicação, nem mecanismos que permitam a comunicação de eventos de reconfiguração entre os mesmos.

8.8 Resumo

Esta seção apresenta uma análise comparativa entre os trabalhos apresentados neste capítulo, levando-se em consideração os recursos disponíveis em cada um deles para apoiar diversos aspectos do desenvolvimento de aplicações distribuídas adaptativas que julgamos importantes.

Permite o monitoramento de recursos do ambiente

Todos os trabalhos apresentados monitoram recursos do ambiente de execução, como a largura de banda disponível e latência de comunicação entre componentes, taxa de utilização de processadores e memória disponível em máquinas, reserva de energia de dispositivos móveis, etc.

Permite o monitoramento de interações entre objetos

Vários trabalhos apresentados (QuO, Moura et al., Proxy Ativo e Conductor) interceptam a comunicação entre os componentes da aplicação com a finalidade de promover adaptações de acordo com o estado do ambiente de execução. No entanto, nenhum deles coleta e analisa as interações entre os componentes da aplicação como forma de identificar padrões de acesso que identifiquem a necessidade de reconfigurar a aplicação, como proposto nesta tese. Conforme vimos no capítulo 7, adaptações disparadas a partir da análise destes dados podem promover significativo ganho de desempenho para certas classes de aplicações.

A solução apresentada é genérica, não estando vinculada a uma classe específica de aplicações

O trabalho de Proxy Ativo é voltado para aplicações de acesso a dados na Internet, enquanto Odyssey é voltado para aplicações de acesso a dados por dispositivos móveis. Os demais trabalhos não foram voltados a uma classe específica de aplicações.

Possui flexibilidade quanto aos mecanismos de adaptação permitidos

O mecanismo de adaptação adotado em Sudame e Badrinath é baseado em ajustes de parâmetros dos protocolos TCP e UDP como forma de dar suporte a variações na rede, típicas de ambientes de computação móvel. Proxy Ativo, Odyssey e Conductor utilizam a transformação dos dados transmitidos entre os componentes da aplicação como mecanismo de adaptação. Chang e Karamcheti adotam a escolha de algoritmos alternativos para adaptar a aplicação a mudanças no ambiente. O arcabouço descrito nesta tese e os trabalhos Moura et al. e QuO não restringem o mecanismo de adaptação a ser adotado.

Possui mecanismos para auxiliar a reconfiguração de aplicações distribuídas

Sudame e Badrinath oferecem apenas uma forma de notificar as aplicações sobre variações no ambiente de execução, não havendo nenhum suporte para a reconfiguração das mesmas. Proxy Ativo disponibiliza uma API que as aplicações podem utilizar para indicar os formatos de dados por elas comportados e para definir que destilações devem ser realizadas no caso de variações do ambiente de execução. De forma semelhante, Odyssey disponibiliza um método que permite à aplicação modificar a fidelidade dos dados acessados. Em Chang e Karamcheti a aplicação deve possuir um conjunto de algoritmos alternativos e deve-se definir parâmetros do ambiente de execução a serem monitorados. De acordo com valores destes parâmetros e com dados armazenados em um banco de dados, obtidos através de simulações realizadas da execução da aplicação, o suporte em tempo de execução escolhe automaticamente qual algoritmo deve ser utilizado. Conductor realiza adaptações transparentes à aplicação, que nem são notificadas a respeito do estado do ambiente de execução.

Maior flexibilidade com relação à reconfiguração das aplicações pode ser obtida através das propostas descritas em Moura et al. e QuO. No primeiro, o código do *smart proxy*, escrito pelo usuário, pode receber eventos que indiquem mudanças no ambiente de execução e consultar o estado de diversas propriedades do ambiente através do serviço de negociação. No segundo, o código do Delegado e do Contrato, também escritos pelo usuário, podem definir quais ações devem ser tomadas face a variações no ambiente de execução.

No entanto, os trabalhos apresentados não implementam mecanismos que auxiliem na reconfiguração de aplicações compostas por componentes distribuídos em diversos nós da rede. Esta foi uma preocupação central do trabalho apresentado nesta tese, que disponibiliza mecanismos para controlar as dependências existentes entre os componentes da aplicação e notificar a ocorrência de eventos de reconfiguração entre eles.

Separa claramente a parte funcional da aplicação da parte reflexiva

Moura et al. isolam o código responsável pelo comportamento adaptativo da aplicação colocando-o no *smart proxy*. Esta separação também ocorre em QuO, através da codificação do Delegado e do Contrato, separados do código funcional da aplicação. O arcabouço descrito nesta tese estrutura o código relativo à adaptação nas classes `EnvironmentAware` e `CorbaComponentConfigurator`.

Permite que aplicações existentes se beneficiem dos mecanismos de adaptação sem a necessidade de modificar seu código

Conductor e Proxy Ativo permitem que as aplicações existentes se beneficiem dos mecanismos de adaptação oferecidos sem a necessidade de modificação de seu código. Todos os demais trabalhos, incluindo o arcabouço descrito nesta tese, exigem modificações no código das aplicações.

8.9 Reflexão Computacional

Segundo [Mae87], um sistema reflexivo é aquele que incorpora estruturas que representam aspectos dele próprio. Através destas estruturas, é possível ao sistema responder a questões a respeito de si próprio bem como dar apoio a ações que influenciam seu comportamento.

Apesar de não termos utilizado para o desenvolvimento do arcabouço uma linguagem que possua suporte à reflexão computacional, como Guaraná [OB99], nem uma plataforma de middleware reflexiva [KCCB02], como DynamicTAO [KRL+00] ou OpenORB [BCA+01], este trabalho possui uma estreita relação com esta área.

O arcabouço desenvolvido pode ser entendido como uma ferramenta de apoio ao desenvolvimento de aplicações reflexivas distribuídas, onde o objetivo da camada reflexiva é promover ações de reconfiguração dos componentes da aplicação em resposta a mudanças detectadas em seu ambiente de execução.

A arquitetura do Configurador de Componente estendido (seção 5.3) compreende a camamada correspondente ao meta-nível, presente nas linguagens e plataformas de middleware reflexivas. Esta arquitetura disponibiliza a introspecção da arquitetura do sistema, através da utilização da interface do Configurador que disponibiliza métodos de consulta às relações de dependência entre componentes inter-dependentes (seção 5.2). A reconfiguração da estrutura do sistema é obtida através da manipulação das relações de dependência mantidas pelo Configurador, através da inclusão ou remoção destas referências.

Capítulo 9

Conclusões

O desenvolvimento de aplicações distribuídas adaptativas é uma tarefa complexa já que requer de seu desenvolvedor o cuidado com diversas questões como as apresentadas na seção 2.2. O desenvolvedor deve ainda estruturar o código da aplicação de forma a separar a parte responsável pelo comportamento funcional da aplicação do código responsável pelo comportamento adaptativo, de forma a simplificar a depuração bem como as manutenções evolutivas e corretivas da mesma.

Nosso trabalho aborda a complexidade inerente ao processo de desenvolvimento de aplicações distribuídas adaptativas através da proposição de um modelo para construção desta classe de aplicações. Tendo por base este modelo, foi implementado um arcabouço que simplifica o processo de desenvolvimento, disponibilizando ferramentas para a monitoração do ambiente de execução, detecção de mudanças no mesmo e reconfiguração dos componentes da aplicação.

Um aspecto relevante do nosso trabalho é o fato de termos explorado a detecção de mudanças no ambiente de execução da aplicação não somente em termos da disponibilidade dos recursos mas também baseado na análise das interações realizadas entre os componentes que compõem a aplicação. A validação do arcabouço foi realizada através de um estudo de caso: a incorporação de mecanismos adaptativos ao protótipo de um sistema de disseminação de informações de trânsito. Resultados experimentais obtidos através da execução de cenários simulados indicaram os benefícios e as limitações do uso da análise das interações entre os componentes da aplicação como forma de detectar a necessidade de executar ações de reconfiguração da aplicação.

Um segundo aspecto a ser destacado em nossa abordagem é o mecanismo disponibilizado no arcabouço para dar apoio à reconfiguração de aplicações compostas por diversos componentes distribuídos em uma rede. A abordagem, baseada em uma extensão da arquitetura do Configurador de Componente, organiza o código que trata os eventos de mudança do ambiente em um conjunto de estratégias e disponibiliza um mecanismo de notificação de eventos de reconfiguração através do qual pode-se coordenar as ações de reconfiguração de forma segura.

Ao definirmos a arquitetura para o arcabouço, procuramos manter os mecanismos para adaptação externos aos processos que compõem a aplicação. A monitoração de recursos e a detecção de mudanças no ambiente de execução (tanto a baseada na disponibilidade de recursos, quanto a baseada na análise de interações) são executadas em processos independentes. Esta abordagem possui como benefício a possibilidade de compartilhamento desta infra-estrutura com outras aplicações adaptativas [1], além de simplificar o código da aplicação.

Finalmente, a arquitetura do arcabouço procurou disponibilizar funcionalidades comuns para a incorporação de mecanismos adaptativos a uma grande variedade de aplicações distribuídas, levando-se em consideração, no entanto, a necessidade de atendermos necessidades específicas. Esta flexibilidade foi obtida através de um modelo que permite estender os componentes do arcabouço.

9.1 Trabalhos Futuros

Durante o desenvolvimento desta pesquisa, identificamos diversos trabalhos que poderiam vir a ser desenvolvidos a partir do trabalho realizado:

1. Conforme descrito na seção 3.3, o suporte à monitoração da infra-estrutura de rede pode ser incorporado através da integração do serviço de monitoração de recursos disponibilizado pelo arcabouço com o serviço de monitoração ReMoS [DGK$^+$01], desenvolvido na Universidade Carnegie Mellon;

2. A seção 4.5 descreve a possibilidade de utilizarmos um monitor de eventos, como o EPS [Mor98], para permitir a definição de eventos compostos, disparados a partir de uma combinação de outros eventos;

[1] Conforme visto, estes componentes foram inclusive projetados para permitir execuções concorrentes de aplicações adaptativas.

3. Aplicação do arcabouço no desenvolvimento de novas aplicações adaptativas e utilização da experiência adquirida para a construção de bibliotecas de componentes do arcabouço (Objetos de Monitoração, eventos e estratégias de reconfiguração) especializados para necessidades específicas de certos domínios de aplicação;

4. O modelo apresentado nesta tese define para a monitoração de recursos do ambiente um componente denominado Gerente de Recursos (seção 4.3), que adquire uma visão global da disponibilidade dos recursos através do recebimento de notificações dos Objetos de Monitoração instanciados no ambiente. Esta abordagem possui limitações com relação à sua escalabilidade que podem ser contornadas através da divisão do ambiente de execução em aglomerados (*cluster*) de computadores, cada um contendo seu próprio Gerente de Recursos. Estes componentes poderiam ser organizados em uma estrutura hierárquica, tornando o modelo escalável;

5. Aplicações adaptativas podem basear suas estratégias de reconfiguração não apenas no valor pontual do estado de disponibilidade de recursos do ambiente mas também em dados estatísticos desta disponibilidade ou projeções futuras para a mesma. A interface do OMR 3.1.1 poderia ser estendida para contemplar estas necessidades;

6. Os serviços de monitoração de recursos e deteção de eventos do ambiente poderiam ser integrados a um serviço de alocação de recursos de forma a constituir uma infra-estrutura que poderia vir a ser utilizada em sistemas de computação em grade (*Grid Computing*), onde o objetivo é o aproveitamento de recursos ociosos em ambientes distribuídos.

Apêndice A

Especificação das Máquinas do LCPD

Máquina	Processador	Memória RAM	Sistema
Orlandia	Athlon 1.4 Ghz	256 MB	Linux 2.4.18
Jundiaí	Pentium II 400 Mhz	128 MB	Linux 2.4.18
Limeira	Pentium II 400 Mhz	128 MB	Linux 2.4.18
Campinas	Pentium II 400 Mhz	128 MB	Linux 2.4.18
Bauru	Pentium II 400 Mhz	128 MB	Linux 2.2.19-pre2
IlhaBela	Pentium II 400 Mhz	128 MB	Linux 2.4.18

Referências Bibliográficas

[ACM95] ACM OOPSLA'95 - Object-Oriented Programming Systems, Languages and Applications. *Workshop on Adaptable and Adaptive Software*, Austin, Texas, December 1995. http://www.cs.neu.edu/research/demeter/adaptable-systems.

[AE91] Gregory R. Andrews and Sally Elliott. *Concurrent Programming: Principles and Practice*. Benjamin-Cummings Publishing, 1991.

[BBI+00] Gordon S. Blair, Lynne Blair, Valerie Issarny, Petr Tuma, and Apostolos Zarras. The role of software architecture in constraining adaptation in component-based middleware platforms. In *Middleware 2000 – IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 164–184, 2000.

[BBIM93] B. R. Badrinath, A. Bakre, T. Imielinski, and R. Marantz. Handling mobile clients: A case for indirect interaction. In *1993. In Proceedings of the Fourth Workshop on Workstation Operating Systems*, Napa, California, October 1993.

[BBN02a] BBN Technologies. *QuO ToolKit Reference Guide, release 3.0.10*, April 2002. http://quo.bbn.com/.

[BBN02b] BBN Technologies. *QuO ToolKit User's Guide, release 3.0.10*, April 2002. http://quo.bbn.com/.

[BCA+01] Gordon S. Blair, Geoff Coulson, Anders Andersen, Lynne Blair, Michael Clarke, Fabio Costa, Hector Duran-Limon, Tom Fitzpatrick, Lee Johnston, Rui Moreira, Nikos Parlavantzas, and Katia B. Saikoski. The design and implementation of Open ORB 2. *IEEE Distributed Systems Online Journal*, 2(6), 2001.

[BG97] Christian R. Becker and Kurt Geihs. MAQS: management for adaptive qoS-enabled services. In *Proceedings of the IEEE Workshop on Middleware for Distributed Real-Time Systems and Services*, 1997.

[BGI99] James Beck, Alain Gefflaut, and Nayeem Islam. Moca: A service framework for mobile computing devices. In *MobiCom*, pages 62–68. ACM, 1999.

[Bor] Borland. Optimizeit suite. http://www.borland.com/optimizeit/index.html.

[BRJ99] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison Wesley, 1999.

[BVD01] Gerald Brose, Andreas Vogel, and Keith Duddy. *Java Programmming with CORBA*. John Wiley and Sons, 3rd edition, 2001.

[CAK$^+$96] C. Cowan, T. Autrey, C. Krasic, C. Pu, and J. Walpole. Fast concurrent dynamic linking for an adaptive operating system. In *Third International Conference on Configurable Distributed Systems*, pages 108–15, Annapolis, MD, May 1996.

[CDP$^+$00] F.M. Costa, H. Duran, N. Parlavantzas, K. Saikoski, G. Blair, and G. Coulson. The role of reflective middleware in supporting the engineering of dynamic applications. *Reflection and Software Engineering, LNCS 1826*, pages 79–98, 2000. Springer.

[Cer00] Renato Cerqueira. *Um Modelo de Composição Dinâmica entre Sistemas de Componentes de Software*. PhD thesis, Computer Science Department, PUC-Rio, Rio de Janeiro, Brazil, 2000.

[Chi00] Shigeru Chiba. Load-time structural reflection in Java. In *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP'2000)*, volume LNCS 1850 of *Lecture Notes in Computer Science*, pages 313–336, Sophia Antipolis and Cannes, France, June 2000. Springer.

[CK00] Fangzhe Chang and Vijay Karamcheti. Automatic configuration and run-time adaptation of distributed applications. In IEEE Computer Society, editor, *Ninth IEEE*

International Symposium on High Performance Distributed Computing, pages 11–20, Pittsburg, Pennsylvania, August 2000.

[Com00] Douglas E. Comer. *Internetworking with TCP/IP: Principles, Protocols, and Architectures*, volume 1. Prentice Hall, 4th edition, 2000. Chapter 9. Internet Protocol: Error and Control Messages (ICMP).

[DAR98] DARPA. Self adaptative software, 1998. BAA 98-12 Proposer Information Pamphlet.

[DGK+01] Peter Dinda, Thomas Gross, Roger Karrer, Bruce Lowekamp, Nancy Miller, Peter Steenkiste, and Dean Sutherland. The architecture of the Remos system. In *10th IEEE Symposium on High-Performance Distributed Computing (HPDC'10)*, San Francisco, August 2001. IEEE.

[DGL+97] T. Dewitt, T. Gross, B. Lowekamp, N. Miller, P. Steenkiste, J. Subhlok, and D. Sutherland. Remos: A resource monitoring system for network aware applications. Technical report, School of Computer Science, Carnegie Mellon University, 1997.

[Dmi01] M. Dmitriev. *Safe Class and Data Evolution in Large and Long-Lived Java Applications.* PhD thesis, University of Glasgow, March 2001.

[Dmi02] M. Dmitriev. Application of the hotswap technology to advanced profiling. In *First International Workshop on Unanticipated Software Evolution*, Málaga, Spain, June 2002.

[dMUCR02] Ana Lúcia de Moura, Cristina Ururahy, Renato Cerqueira, and Noemi Rodriguez. Dynamic support for distributed auto-adaptive applications. In *Proceedings of AOPDCS - Workshop on Aspect Oriented Programming for Distributed Computing Systems (held in conjunction with IEEE ICDCS 2002)*, pages 451–456, Vienna, Austria, July 2002.

[dSeSE00] Francisco José da Silva e Silva and Markus Endler. Requisitos e arquiteturas de software para computação móvel. In *Anais do Workshop sobre Métodos e Serviços para Computação Móvel*, São Paulo, Outubro 2000.

[dSGE98] Dilma Menezes da Silva, Marco Dimas Gubitoso, and Markus Endler. Sistemas de informação distribuídos para agentes móveis. In *Proceedings of the XXV Brazilian Software and Hardware Seminars (SEMISH'98)*, pages 125–140, Belo Horizonte, Brazil, August 1998. SBC. Available in http://www.ime.usp.br/ dilma/papers/semish98.ps.

[EdSeS+00] M. Endler, D.M. da Silva, F. Silva e Silva, R.A. da Rocha, and M.A. de Moura. Project SIDAM: Overview and Preliminary Results. In *Anais do 2o. Workshop de Comunicação sem Fio (WCSF), Belo Horizonte*, May 2000.

[FGBA96] Armando Fox, Steven D. Gribble, Eric A. Brewer, and Elan Amir. Adapting to network and client variability via on-demand dynamic distillation. *ACM SIGPLAN Notices*, 31(9):160–170, September 1996. Co-published as SIGOPS Operating Systems Review **30**(5), December 1996, and as SIGARCH Computer Architecture News, **24**(special issue), October 1996.

[FGCB98] A. Fox, S. Gribble, Y. Chawathe, and E. Brewer. Adapting to network and client variation using active proxies: Lessons and perspectives. *special issue of IEEE Personal Communications on Adaptation*, 5(4), August 1998.

[FJS99] Mohamed E. Fayad, Ralph E. Johnson, and Douglas C. Schmidt. *Building Application Frameworks: Object-Oriented Foundations of Framework Design.* John Wiley and Sons, 1999.

[FK98] S. Frolund and J. Koistinen. Quality-of-service specification in distributed object systems. In *Proceedings of the 4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS98)*, April 1998.

[FL98] Jim Farley and Mike Loukides. *Java Distributed Computing.* O'Reilley and Associates, 1998.

[FZ94] G. Forman and J. Zahorjan. The challenges of mobile computing. *IEEE Computer*, April 1994.

[GHVJ94] Erich Gamma, Richard Helm, John Vlissides, and Ralph Johnson. *Design Patterns: Elements of Reusable Object Oriented Software.* Addison-Wesley, 1994.

[Gre02] Dale Green. *The Reflection API.* Sun Microsystems, available at http://java.sun.com/docs/books/tutorial/reflect, 2002.

[HHC+99] Abdelsalam Helal, Bert Haskell, Jeffery L. Carter, Richard Brice, Darrell Woelk, and Marek Rusinkiewicz. *Any Time, Anywhere Computing.* Kluwer Academic Publishers, 1999.

[HV99] Michi Henning and Steve Vinoski. *Advanced CORBA Programming with C++.* Addison-Wesley, 1999.

[IBW97] J. Inouye, J. Binkley, and J. Walpole. Dynamic network reconfiguration support for mobile computers. In *Third Annual ACM/IEEE International Conference on Mobile Computing and Networking*, pages 13–22, Budapest, Hungary, September 1997.

[IdFC96] R. Ierusalimschy, L. H. de Figueiredo, and W. Celes. Lua—an extensible extension language. *Software: Practice & Experience*, 26(6):635–652, 1996.

[Jac02] Jacorb developers archives. available at http://lists.spline.inf.fu-berlin.de/mailman/listinfo/jacorb-developer, 2002.

[JHE99] Jin Jing, Abdelsalam Helal, and Ahmed Elmagarmid. Client-server computing in mobile enviroments. *ACM Computing Surveys*, 31(2), June 1999.

[Joh97] Ralph E. Johnson. Components, frameworks, patterns. In *ACM SIGSOFT Symposium on Software Reusability*, pages 10–17, 1997.

[KC00] Fabio Kon and Roy H. Campbell. Dependence management in component-based distributed systems. *IEEE Concurrency*, 8(1):26–36, January/March 2000.

[KCCB02] Fabio Kon, Fábio Costa, Roy Campbell, and Gordon Blair. The case for reflective middleware. *Communications of the ACM*, 45(6):33–38, June 2002.

[KHR+00] Fabio Kon, Christopher Hess, Manuel Roman, Roy H. Campbell, and M. Dennis Mickunas. A flexible, interoperable framework for active spaces. In *OOPSLA'2000 Workshop on Pervasive Computing*, Minneapolis, October 2000.

[Kon00] Fabio Kon. *Automatic Configuration of Component-Based Distributed Systems.* PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, May 2000.

[KRL+00] Fabio Kon, Manuel Roman, Ping Liu, Jina Mao, Tomonori Yamane, Luiz Claudio Magalhães, and Roy H. Campbell. Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000)*, number 1795 in LNCS, pages 121–143, New York, April 2000. Springer-Verlag.

[LMG+99] Bruce Lowekamp, Nancy Miller, Thomas Gross, Peter Steenkiste, Jaspal Subhlok, and Dean Sutherland. A resource query interface for network-aware applications. *Cluster Computing*, 2(2):139–151, 1999.

[Mae87] P. Maes. Concepts and experiments in computational reflection. In ACM Press, editor, *Proceedings of Object-Oriented Programming Systems, Languages, and Applications Conference '87*, volume 22 of *Special Issue of Sigplan Notices*, pages 147–155. ACM, December 1987.

[ME01] Douglas Moreto and Markus Endler. Evaluating composite events using shared trees. *IEEE Proceedings Software*, 2001. ISSN 1462-5970, 148(1).

[Men00] Alex C. Meng. On evaluating self-adaptive software. In Paul Robertson, Howard E. Shrobe, and Robert Laddaga, editors, *IWSAS 2000: International Workshop on Self-Adaptive Software*, volume 1936 of *Lecture Notes in Computer Science*, pages 65–74, Oxford, England, April 2000. Springer.

[MGPK99] David J. Musliner, Robert P. Goldman, Michael J. Pelican, and Kurt D. Krebsbach. Self-adaptive software for hard real-time environments. *IEEE Intelligent Systems*, 14(4):23–29, July / August 1999.

[ML98] Geraldo Robson Mateus and Antônio Alfredo Ferreira Loureiro. *Introdução à Computação Móvel*. DCC/IM, COPPE/Sistemas, NCE/UFRJ, 1998. 11^a Escola de Computação.

[Mor98] Douglas Moreto. Monitoramento de eventos compostos em sistemas distribuidos. Master's thesis, Instituto de Matematica e Estatistica, Universidade de Sao Paulo, September 1998.

[MRC+97] J. N. Matthews, D. Roselli, A. M. Costello, R. Y. Wang, and T. E. Anderson. Improving the performance of log-structured file systems with adaptive methods. In *Proc. Sixteenth Symposium on Operating Systems Principles*, pages 238–251, October 1997.

[MRN02] Renato Cerqueira Anand Ranganathan Roy H. Campbell Manuel Román, Christopher K. Hess and Klara Nahrstedt. Gaia: A middleware infrastructure to enable active spaces. *In IEEE Pervasive Computing*, pages 74–83, October-December 2002.

[NMMS00] N. Narasimhan, L. E. Moser, and P. M. Melliar-Smit. Interception in the aroma system. In *Proceedings of the ACM 2000 Java Grande Conference*, pages 107–115, San Francisco, June 2000. ACM.

[NPS95] Brian D. Noble, Morgan Price, and Mahadev Satyanarayanan. A programming interface for application-aware adaptation in mobile computing. Technical Report CS-95-119, Carnegie Mellon University, School of Computer Science, February 1995.

[NS99] B. D. Noble and M. Satyanarayanan. Experience with adaptive mobile applications in odyssey. *Mobile Networks and Applications*, 4(4):245–254, 1999. Kluwer.

[NSN+97] Brian D. Noble, M. Satyanarayanan, Dushyanth Narayanan, James Eric Tilton, Jason Flinn, and Kevin R. Walker. Agile application-aware adaptation for mobility. In *Sixteen ACM Symposium on Operating Systems Principles*, pages 276–287, Saint Malo, France, October 1997.

[OB99] Alexandre Oliva and Luiz Eduardo Buzato. The design and implementation of guaraná. In *5th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'99)*, pages 203–216, San Diego, CA, May 1999.

[OGT+99] P. Oreizy, M. Gorlick, R. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. Rosenblum, and A. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14(3):54–62, May/June 1999.

[OMG02] OMG - Object Management Group. *The Common Object Request Broker: Architecture and Specification*, November 2002. version 3.0.1.

[PS98] Evaggelia Pitoura and George Samaras. *Data Management for Mobile Computing.* Kluwer Academic Publisher, 1998.

[RC00] Manuel Roman and Roy H. Campbell. Gaia: Enabling Active Spaces. In *Proceedings of the 9th ACM SIGOPS European Workshop*, Kolding, Denmark, September 2000.

[RCS99] M. A. Rodrigues, S. Colcher, and L. F. Soares. Um framework para a provisão de QoS em ambientes genéricos de processamento e comunicação. In *Anais do XVII Simpósio Brasileiro de Redes de Computadores (SBRC99)*, Salvador, Bahia, May 1999.

[RJJ+98] Vanegas R, Zinky JA, Loyall JP, Karr DA, Schantz RE, and Bakken DE. Quo's runtime support for quality of service in distributed objects. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*, The Lake District, England, September 1998.

[Sat96] M. Satyanarayanan. Fundamental challenges in mobile computing. In *In Fifteenth ACM Symposium on Principles of Distributed Computing*, Philadelphia, PA, May 1996.

[SAW94] Bill Schilit, Norman Adams, and Roy Want. Context-aware computing applications. In *IEEE Workshop on Mobile Computing Systems and Applications*, pages 85–90, Sata Cruz, CA, December 1994. Palo Alto Research Center.

[SB97] P. Sudame and B. Badrinath. On providing support for protocol adaptation in mobile wireless networks. Technical report, Department of Computer Science, Rutgers Universit, June 1997. http://www.cs.rutgers.edu/pub/technical-reports/dcstr -333.ps.Z.

[Sof02] Software Engineering and Systems Software Group at Freie Universität Berlin and Xtradyne Technologies AG. *JacORB Programming Guide*, August 2002. http://jacorb.inf.fu-berlin.de.

[Sta02] William Stallings. *Wireless Applications and Networks*. Prentice-Hall, 2002.

[Sun02a] Sun Microsystem, available at http://java.sun.com/j2se/1.4.1/docs/api. *Java2 Platform, Standard Edition, Version 1.4.1, API Specification*, 2002.

[Sun02b] Sun Microsystem, available at http://java.sun.com/j2se/1.4/docs/guide/rmi/spec/rmiTOC.html. *RMI Architecture and Functional Specification*, 2002.

[Sun02c] Sun Microsystems. *Java Standard Edition 1.4, Java Platform Debugger Architecture*, March 2002.

[Tan02] Andrew S. Tanenbaum. *Computer Networks*. Prentice Hall, 2002.

[WW90] Thomas H. Wonnacott and Ronald J. Wonnacott. *Introductory Statistics*. John Wiley and Sons, 5th edition, 1990.

[YRP99] Mark Yarvis, Peter Reiher, and Gerald J. Popek. Conductor: A framework for distributed adaptation. In *Seventh Workshop on Hot Topics in Operating Systems*. IEEE Computer Society Press, March 1999.

[YRP00] Mark Yarvis, Peter Reiher, and Gerald J. Popek. A reliability model for distributed adaptation. In *3 rd IEEE Conference on Open Architectures and Network Programming*, Tel-Aviv, Israel, March 2000.

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