Universidade Federal Fluminense

GLAUCO HENRIQUE FREITAS

Um experimento de uso de um framework de suporte a requisitos não-funcionais de qualidade

NITERÓI

2005

GLAUCO HENRIQUE FREITAS

Um experimento de uso de um framework de suporte a requisitos não-funcionais de qualidade

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

Orientador:
ORLANDO GOMES LOQUES FILHO

Universidade Federal Fluminense

NITERÓI

2005

Às minhas pessoas queridas.

Agradecimentos

Agradeço primeiramente a Deus, por me dar a oportunidade de presenciar as aventuras e desafios terrenos, que me permitiram o crescimento moral e intelectual alcançado nessa grande escola, o planeta Terra.

Agradeço ao meu pai, minha mãe e meu irmão pelo apoio na concretização do sonho de concluir o mestrado e por nunca me deixarem desistir de correr atrás da realização do que em alguns momentos parecia tão distante.

Agradeço à minha amiga e namorada Renata, que tornou os momentos difíceis do mestrado mais amenos e por sempre acreditar na busca dessa conquista, oferecendo amor e carinho para que eu pudesse trilhar meu caminho.

Agradeço aos companheiros de república Alex, Hébio (Snypis), Kennedy (Kenny G.), Leandro (Bixão), Raphael, Renan, Renathinha e Thiago (Facada) pelos momentos de alegria, que proviam diversão, que sempre geravam aprendizado. Aos companheiros de Niterói Felipe Carone, Lalita, Marcel, Mariana, Nina, Sandro, Tiago Gobbi e Tiago (Jovem) pelos rocks, cervas e conversas.

Aos colegas da Uff, Alexandre Senna, Aline, Allyson, Ary, Átila (Taz), André (Totó), Bruno Novelli, Cris, Cristiane Targa, Cristiano, Dani, Eduardo (CC), Idalmis, Igor, Ivairton, Jacques, Jonildo, Juliana, Luciana, Luciano, Luís, Robson, Rodrigo, Rone, Sanderson, Sidney, Stênio, Vivi Thomé e Vivi Trindade, que sempre me serviram de exemplo e me deram força para lutar contra o tempo em busca do título, além dos agradáveis momentos de diversão, convivência e entretenimento. Em especial, a Alex, André (Mokado), Jonivan, Léo, Marcos e Paulo Motta pelos projetos desenvolvidos juntos, que demandaram paciência e cooperação.

Ao Prof. Orlando Loques pela orientação, paciência, compreensão, persistência, lições de vida, discussões, conselhos e por acreditar na minha capacidade até o fim do mestrado. Sem o seu acompanhamento esse título não seria meu.

Aos membros da banca pelas contribuições apresentadas.

A todos os professores do Instituto de Computação da Universidade Federal Fluminense que contribuiram para minha formação no Mestrado. Em particular, aos professores Julius Leite, Cristina Boeres, Anna Dolejsi, Michael Stanton e Maria Luiza, pelas aulas que sempre agregavam informações úteis para vários momentos dessa caminhada. Às secretárias do IC, Ângela, Izabela e Maria por toda atenção e apoio necessário nas tarefas do dia a dia.

Aos professores da Universidade Federal de Ouro Preto que deram uma base de conhecimentos suficiente para que eu pudesse dar esse passo na minha caminhada.

À CAPES por financiar parcialmente este trabalho.

Agradeço a todos outros que porventura foram omitidos por um esquecimento momentâneo, mas nunca por falta de importância. A todas estas pessoas que, com o convívio, colocaram uma "pecinha" desse quebra-cabeça, para que no final esse título pudesse ser montado.

"Comece fazendo o necessário, depois faça o possível,
e quando você se der conta estará fazendo o impossível"

São Francisco de Assis

Resumo

Neste trabalho investigamos a garantia de requisitos não-funcionais de qualidade (ou de QoS) especificados através de contratos de QoS em aplicações distribuídas. Tais requisitos englobam características de desempenho e comunicação, dentre outras, que dependem da disponibilidade dos recursos do sistema de suporte, os quais são requeridos e utilizados pelas diversas aplicações.

Neste contexto, torna-se adequado o emprego de mecanismos capazes de verificar o atendimento dos requisitos das aplicações e monitorar a utilização dos recursos disponíveis. Essa abordagem permite que situações que caracterizam o fornecimento inadequado do serviço possam ser identificadas e atividades de adaptação possam ser efetuadas, visando prover garantias de qualidade ao nível da aplicação.

A infra-estrutura necessária para garantir o provimento de serviços com qualidade diferenciada pode ser implementada de maneira *ad-hoc*, específica por aplicação, ou de forma genérica e reaproveitável através de um *framework* adequado, tal como o CR-RIO (*Contractual Reflective - Reconfigurable Interconnectable Objects*), que foi utilizado como base neste trabalho.

Em particular, investigamos a inclusão de mecanismos padronizados de localização e monitoramento de recursos no CR-RIO e desenvolvemos um conjunto de aplicações com requisitos de qualidade de serviço, utilizando os mecanismos de especificação e gerenciamento de contratos nele disponíveis. Essas atividades permitiram identificar ganhos no desempenho das aplicações e uma melhor utilização dos recursos disponíveis, devido a utilização dos mecanismos do *framework*. Além disso, pudemos investigar as vantagens e desvantagens do seu emprego em relação a uma implementação específica para cada aplicação.

Palavras-chave

Auto-configuração, adaptação em tempo de execução, CBabel, computação autônoma, contratos de QoS, CR-RIO, monitoração, NWS, requisitos não-funcionais, seleção de recursos.

Abstract

In this work we investigate the guarantee of nonfunctional requirements of quality of service (or QoS), which are specified by QoS contracts in distributed applications. Such requirements can be associated to resources or services, such as processing and communication, that are required and shared by several applications.

In this context, mechanisms to verify the attendance of the applications' requirements and to monitor the usage level of the available resources are required. This approach allows situations in which inadequate services are being delivered to be identified, and as a response to perform adaptation activities, aiming to provide guarantees of quality at the application level.

The infrastructure required to ensure the required quality of service can be implemented either in an ad-hoc manner (application specific) or in a generic and reusable way by a suitable framework, such as that used in this work, named CR-RIO (Contractual Reflective - Reconfigurable Interconnectable Objects).

In particular, we investigate the inclusion of standard mechanisms for resource monitoring and location into the CR-RIO. In addition, we developed a set of applications with QoS requirements, using CR-RIO´s specification and contract management mechanisms. Based on the experiments performed using the framework, we identified increases in application performance as well as better utilization of the available resources. Moreover, we observed the advantages and disadvantages of its deployment when compared to an application specific (ad hoc) implementation.

Keywords:

Autonomic computing, CBabel, CR-RIO, monitoring, nonfunctional requirements, NWS, QoS contracts, resource selection, run-time adaptation, self-configuration.

Glossário

ACK	:	*Acknowledgement*
API	:	*Application Programming Interface*
ASCII	:	*American Standard Code for Information Interchange*
CORBA	:	*Common Object Request Broker Architecture*
DAG	:	*Directed Acyclic Graph*
GUI	:	*Graphical User Interface*
LAN	:	*Local Area Network*
MeCo	:	*Metric Collector*
NTP	:	*Network Time Protocol*
NSSD	:	*Network-Sensitive Service Discovery*
NWS	:	*Network Weather System*
QoS	:	*Quality of Service*
Remos	:	*REsource MOnitoring System*
RRDtool	:	*Round Robin Database Tool*
SLP	:	*Service Location Protocol*
SNMP	:	*Simple Network Management Protocol*
TC	:	*Traffic Controler*
XDR	:	*External Data Representation standard*
XML	:	*Extensible Markup Language*

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

O aumento da qualidade das redes de comunicação, no atual contexto tecnológico, tem permitido aos usuários utilizarem recursos e dados computacionais dispersos geograficamente [Foster 2001], proporcionando assim um alto grau de conectividade entre estes. Essa interligação entre recursos geograficamente distribuídos é comumente denominada *grid*. Tal conectividade torna possível escolher, dentre vários recursos equivalentes, quais serão utilizados por uma determinada aplicação. Essa escolha é feita com base em fatores como desempenho, disponibilidade ou mesmo localização geográfica, permitindo assim uma melhor execução (baseada em alguma métrica) de aplicações de alto desempenho. Porém, uma execução ainda mais otimizada pode ser conseguida com a possibilidade de que as aplicações, ou sua infra-estrutura, possam se adaptar diante de variações de disponibilidade de recursos ou de necessidades do usuário. Por execução mais otimizada, neste contexto, entende-se a utilização dos recursos de acordo com as necessidades da aplicação, sem deixá-los ociosos e sem deixar a aplicação carente a eles.

Essa capacidade de adaptação geralmente está embutida no código das aplicações, na forma de construções da linguagem de programação, como por exemplo, em códigos associados ao tratamento de exceções, ou nos algoritmos, como por exemplo em protocolos tolerantes a falhas. Além disso, esses mecanismos costumam ser altamente específicos para uma determinada aplicação. Tais características tornam a adaptação custosa de ser modificada, dificultando o reúso [Cheng et al. 2004] e mais susceptível a erros.

De outra forma, pode-se tratar a adaptação de maneira externa à aplicação, separando os aspectos funcionais, que implementam a funcionalidade básica da aplicação através de atividades especializadas, dos aspectos não-funcionais, mecanismos de suporte que satisfazem os requisitos não-específicos da aplicação. Entre os aspectos não-funcionais podemos destacar disponibilidade, confiabilidade, segurança, sincronização, suporte a tempo real,

persistência, coordenação e suporte a depuração, que podem necessitar da adaptação da aplicação, ou da sua infra-estrutura, para serem garantidos. Tais mecanismos geralmente podem ser reutilizados em diversas aplicações.

Em um exemplo de aplicações no contexto de *grids*, pode ser necessário que requisitos relacionados às capacidades das máquinas, como disponibilidade de recursos de processamento ou de memória, por exemplo, atendam a restrições de qualidade de serviço (QoS - *Quality of Service*) exigidas pelos serviços da aplicação. Uma vez que esses recursos podem estar compartilhados por várias aplicações, tais disponibilidades podem variar ao longo do tempo, o que torna necessária a presença de mecanismos capazes de acompanhar os níveis de utilização dos recursos, possibilitando verificar se estes ainda satisfazem as restrições exigidas pela aplicação. Na presença de variações significativas nos níveis dos recursos monitorados, pode ser necessário efetuar alguma ação, tentando adaptar a aplicação ao novo estado do sistema ou encerrando-a, na impossibilidade de outras alternativas.

Nesse contexto, a utilização de arquitetura de *software* ganha destaque, uma vez que incentiva a separação de interesses, facilitando distinguir os requisitos funcionais do sistema dos requisitos não-funcionais [Loques et al. 1999]. O princípio da arquitetura de software considera que um sistema pode ser desenvolvido a partir de sua organização, como uma combinação de componentes, suas interações e suas propriedades de interesse [Ansaloni 2003], permitindo um entendimento mais global do sistema, com maior nível de abstração e viabilizando a reutilização de módulos (componentes e conectores).

Os componentes funcionais do sistema encapsulam a funcionalidade do sistema, possuem pontos de acesso bem definidos, e podem ser representados pelas classes de aplicação, procedimentos, funções ou programas executáveis. Interligando esses componentes funcionais aparecem os conectores, que podem encapsular os requisitos não-funcionais do sistema e flexibilizam a execução do mesmo. Os conectores podem também estar interligados diretamente entre si.

Viabilizando a ligação real entre componentes e/ou conectores estão as portas, que identificam os pontos de acesso pelos quais são oferecidos ou solicitados serviços [Sztajnberg 2002]. Existem dois tipos de porta: de saída, que representam serviços que podem ser requisitados, como por exemplo, a invocação de um método que presta um serviço, e de entrada, que representam os métodos disponíveis em um módulo servidor, sendo encontradas na definição da interface de tal módulo. Maiores detalhes sobre a utilização de arquiteturas de *software* serão apresentados no próximo capítulo.

A topologia do sistema, chamada configuração, pode ser obtida a partir da descri-

ção dos componentes e das interações existentes entre eles, representando explicitamente a arquitetura empregada em sua concepção, que pode ser definida através de alguma linguagem de descrição de arquitetura (ADL - *Architecture Description Language*). Utilizando a linguagem, devem ser definidos os componentes, os conectores, seus respectivos pontos de interação e as ligações entre estes.

Essa visão de componentes permite a reutilização de módulos e proporciona ao desenvolvedor um entendimento mais amplo do sistema, facilitando o gerenciamento de detalhes da evolução de um sistema de software e provendo um melhor entendimento das implicações dessas mudanças [Perry 1987]. Nesse contexto, os componentes da aplicação podem ser projetados independentemente, e então serem adicionadas características não-funcionais ao sistema. Isto é bastante útil, pois muitas vezes nem todos os requisitos de suporte de uma aplicação são conhecidos antes da fase de desenvolvimento e precisam ser integrados à aplicação posteriormente [Sztajnberg 2002].

Os requisitos não específicos da aplicação, que também podem ser chamados requisitos de QoS, podem ser adicionados ao sistema apenas conhecendo-se os pontos de acesso dos componentes, sem a necessidade de saber como as tarefas da aplicação são executadas. Mecanismos para especificação dos requisitos de QoS devem, então, ser fornecidos, e um ambiente capaz de implantar essas restrições nos recursos do sistema de forma automatizada torna-se necessário. A capacidade de monitorar e adaptar a aplicação conforme a disponibilidade destes recursos, para assegurar a garantia de atendimento das restrições de QoS em tempo de execução, deve ser característica de tal ambiente.

Os requisitos de QoS podem ser especificados a partir de contratos que definem as interações entre as entidades participantes [Perry 1987], e descrevem as características de QoS oferecidas e requeridas por estas entidades, visando formalizar as exigências básicas, para a instalação e operação de serviços, que devem ser atendidas local e globalmente. Os contratos definem perfis de qualidade a serem estabelecidos ou mantidos, permitindo que os aspectos não-funcionais desejados sejam monitorados durante a etapa de funcionamento do sistema.

O foco deste trabalho é o desenvolvimento de exemplos de aplicação pertencentes à classe cliente-servidor para o ambiente CR-RIO [Curty 2002]. Os exemplos visam a validação do ambiente estudado e a discussão a respeito do mapeamento de componentes do mesmo a ferramentas disponíveis. O CR-RIO baseia-se na proposta de um ambiente de suporte a contratos de qualidade de serviço diferenciado no nível de Arquitetura de Software. Foi realizado um estudo deste *framework* e da linguagem de descrição de arquiteturas com

suporte a contratos de QoS adotada por ele, para a elaboração desta dissertação. Completando o trabalho, foram estudadas outras abordagens de mesmo propósito e comparadas com o CR-RIO.

1.1 Objetivos do trabalho

Este trabalho objetiva:

- Revisar alguns conceitos referentes à arquitetura de software, contratos e sistemas autônomos.
- Apresentar o *framework* CR-RIO [Curty 2002, Ansaloni 2003], uma evolução do R-RIO proposto em [Lobosco 1999], e discutir algumas modificações.
- Apresentar detalhes relevantes da linguagem CBabel adotada pelo CR-RIO para descrever arquiteturas e contratos de QoS
- Desenvolver algumas aplicações exemplo, discutir o mapeamento dos componentes do *framework* para cada uma delas e demonstrar as melhorias conseguidas com sua utilização.
- Avaliar a capacidade de reutilização dos módulos.
- Discutir o mapeamento de alguns componentes flexíveis do *framework* cujas funcionalidades já estão disponíveis em ferramentas disponibilizadas, como por exemplo o sistema de monitoramento, que pode empregar instrumentais como o NWS (*Network Weather Service*), o Remos (*REsource MOnitoring System*), ou o Ganglia.

1.2 Estrutura da dissertação

O demais capítulos da dissertação estão dispostos da seguinte forma:

- No capítulo 2 são apresentados alguns conceitos fundamentais para o entendimento da proposta.
- No capítulo 3 é explicitada a proposta da dissertação.
- No capítulo 4 apresentamos exemplos e avaliações da nossa proposta.

- No capítulo 5 são expostos alguns trabalhos correlatos estudados
- O capítulo 6 apresenta a conclusão do trabalho e descreve alguns trabalhos futuros.

Capítulo 2

Conceitos preliminares

Este capítulo apresenta conceitos de arquitetura de *software*, apresentando alguns elementos sobre descrição arquitetural e contratos, abordando seus fundamentos e suas formas de descrição. Introduz uma discussão sobre mecanismos de monitoramento e trata também aspectos de auto-adaptação e sistemas autônomos. Tais conceitos foram escolhidos, por acreditarmos na implementação de um serviço a partir da sua descrição através de um contrato, que o define em termos de elementos arquiteturais com requisitos que devem ser cumpridos, mesmo que para isso sejam necessárias adaptações na aplicação, ou em sua infra-estrutura. Ações, que para serem tomadas com maior precisão, necessitam de instrumentações capazes de definir o estado de um determinado recurso.

2.1 Arquitetura de *software*

Sistemas grandes e complexos requerem mecanismos de decomposição de forma a facilitar seu tratamento. O particionamento de um sistema em componentes facilita a realização de avaliações sobre o comportamento do sistema, através do entendimento do comportamento de cada um de seus componentes. A concepção de um sistema de *software* pode partir da definição de uma arquitetura, que o descreve em termos de seus elementos (os componentes) e das conexões existentes entre eles (os conectores) [Shaw et al. 1995], abstraindo os detalhes de implementação. Essa descrição se preocupa com os aspectos de alto nível do sistema, como sua organização de forma geral.

Os componentes, a grosso modo, correspondem às unidades compiladas de uma linguagem de programação convencional, processos e outros objetos do nível do usuário, como arquivos. Sua implementação é feita de forma a permitir reutilização em diferentes configurações e, através de sua interface, serviços são oferecidos e/ou requeridos. Os

conectores intermedeiam a interação entre os componentes, ou seja, estabelecem as regras que guiam as interações dos componentes e especificam algum mecanismo auxiliar necessário [Shaw et al. 1995].

A estrutura do sistema e sua topologia são especificadas pela arquitetura. Os módulos funcionais do sistema (ou componentes) e os conectores formam a estrutura, e a topologia é caracterizada pela criação e interligação desses componentes e conectores formando uma configuração.

Além dos componentes e conectores, outro elemento de fundamental importância que compõe a arquitetura de um sistema de *software* é a porta. Esta é responsável por identificar o(s) ponto(s) de interação entre um componente ou conector e o ambiente em que foi criado. A interface de um componente ou de um conector é representada pelo seu conjunto de portas.

Através da modularidade oferecida pela arquitetura de *software*, a concepção de *software* por composição é encorajada, o que permite a reutilização de módulos previamente concebidos minimizando esforços de desenvolvimento.

Com o conceito de arquitetura de *software*, um entendimento amplo do sistema é oferecido, permitindo ao projetista concentrar-se nos elementos arquiteturais (componentes e conectores) e na sua estrutura de interconexões (configuração). Isso proporciona ao projetista um alto nível de abstração, com foco na estrutura global do sistema, e não em detalhes com pouca relevância para o processo de desenvolvimento do mesmo. O que para o ciclo de vida da aplicação pode ser importante, principalmente para programas grandes, favorecendo sua evolução dinâmica.

Outra conseqüência da aplicação do conceito de arquiteturas é a separação de interesses. Uma arquitetura de *software* auxilia na divisão entre o processamento feito por um sistema (realizado pelos componentes) e a interação entre suas unidades computacionais [Medvidovic 1999]. A partir da separação de interesses, as funções básicas de uma aplicação (propriedades funcionais) são distintas dos aspectos operacionais ou propriedades não-funcionais, ou de qualquer outro aspecto não coberto na descrição funcional. Assim, permite-se que os projetistas concentrem-se separadamente nos aspectos funcionais da aplicação, podendo encapsulá-los nos componentes, e nos não-funcionais, podendo encapsulá-los nos conectores [Sztajnberg 2002]. Isso reduz o custo de desenvolvimento de aplicações, pois possibilita um maior reaproveitamento de *software*, e facilita sua extensão e manutenção.

Os elementos de uma arquitetura são definidos e especificados através da utilização de uma linguagem de descrição arquitetural, ou ADL (*Architecture Description Language*). Na linguagem, são especificados tipos abstratos para componentes e conectores, é feita a instanciação destes e são definidas as ligações entre eles - o que estabelece a configuração [Lisbôa et al. 2002]. A descrição tem o intuito de servir de meta-informação sobre a aplicação.

2.1.1 Elementos básicos de arquitetura

De forma geral, existem três tipos de elementos fundamentais em uma arquitetura de *software*: componentes, conectores e portas; que podem ser especificados independentemente e posteriormente configurados em uma arquitetura, além de poderem ser reaproveitados em diferentes contextos.

2.1.1.1 Componentes

Um componente refere-se ao encapsulamento de uma porção de código, como um processo, objeto, *procedure*, ou qualquer pedaço de código ou dados identificável. Os componentes podem corresponder a unidades de compilação de linguagens de programação convencionais e objetos, tais como arquivos [Shaw et al. 1995], ou ainda, no nível de linguagens de programação, corresponder a módulos de programação, classes, objetos ou um conjunto de funções [Curty 2002].

Os componentes apresentam explicitamente uma interface, através da qual podem fornecer ou requerer serviços. No nível de linguagem de programação, por exemplo, os métodos, funções e procedimentos definidos podem representar serviços oferecidos, enquanto invocações a métodos, funções e procedimentos podem representar requerimentos de serviços.

Os componentes são as unidades que encapsulam os requisitos funcionais do sistema. Estes devem funcionar como um tipo ou classe, e possuir uma especificação que defina seus pontos lógicos de interação (as portas) com outros componentes e suas propriedades, possibilitando sua reutilização em diferentes configurações. Pela interface, representada pelo conjunto de portas, são fornecidos ou requeridos serviços pelo componente como mostrado na figura 2.1, que ilustra a representação de um componente cliente e de um componente servidor com suas respectivas portas de saída e de entrada.

Figura 2.1: *Representação de componentes. O cliente com sua porta de saída representando a invocação de métodos e o servidor com sua porta de entrada representando os métodos definidos.*

2.1.1.2 Portas

As portas são os pontos lógicos de interação definidos nas interfaces dos componentes e conectores. São responsáveis pela ligação real entre componentes e/ou conectores e podem incluir assinatura, funcionalidade e propriedades de interação. As portas podem ser de entrada e de saída. As portas de saída representam requisições de serviços nos clientes, e as portas de entrada, os métodos disponíveis em um módulo servidor. Nos conectores, as portas representam pontos de interceptação de requisições para tratamento não-funcional das mesmas e são denominadas *roles* em algumas referências como [Ivica e Larsson 2002], porém a função desempenhada é a mesma.

2.1.1.3 Conectores

Os conectores fazem a interação entre componentes. Em princípio eles encapsulam a funcionalidade necessária à interação sem a necessidade de modificar ou adaptar as interfaces dos componentes envolvidos.

Além de intermediar essa interação entre componentes, os conectores especificam quaisquer mecanismos auxiliares de implementação e podem representar dados compartilhados, chamadas remotas, fluxos de dados, protocolos de comunicação, protocolos de interação, modelos de sincronização e de concorrência.

Durante a configuração, várias instâncias de um mesmo tipo de conector podem ser criadas, para intermediar as interações entre instâncias de componentes. A configuração é realizada interligando-se as portas dos conectores às portas dos componentes envolvidos.

Algumas características dos conectores os tornam atraentes para o tratamento de aspectos relacionados à interação e a comunicação entre componentes:

- a capacidade de examinar o conteúdo de requisições e respostas antes de encaminhá-las aos seus destinos finais, podendo controlar e manipular o repasse destas ao destino;

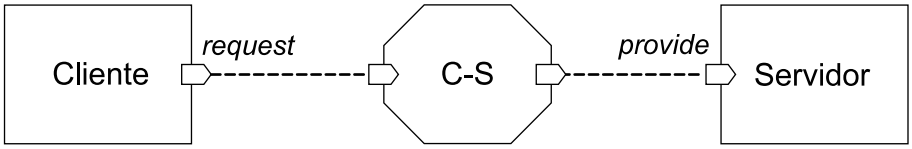

Figura 2.2: *Representação de arquitetura de um sistema cliente-servidor. A interação entre os componentes cliente e servidor é intermediada pelo conector c-s.*

- o conhecimento da interface e as referências dos componentes interligados por eles;
- a sua similaridade com componentes quanto às propriedades de configuração, facilitando sua criação através da composição de outros conectores.

A figura 2.2 mostra a representação da arquitetura de um sistema cliente-servidor, cujas interações são intermediadas por um conector. As portas dos componentes cliente e servidor estão interligadas às portas do conector c-s, o que torna possível a interação entre os componentes. As requisições de Cliente, executadas através da porta de saída (*request*), são interceptadas e encaminhadas ao componente Servidor através da porta de entrada (*provide*).

2.1.2 Descrição arquitetural

O desenvolvimento de *software* baseado em idiomas arquiteturais comuns, ou seja, em recorrência de características comuns de um domínio de aplicações, deixou de focar em linhas de código para focalizar em elementos arquiteturais (componentes de *software* e conectores) e suas estruturas de interconexão. Para dar suporte ao desenvolvimento baseado em arquitetura, análises e notações de modelagem formal e ferramentas de desenvolvimento que operam nas especificações arquiteturais se tornaram necessárias. A ADL e seus conjuntos de ferramentas associados surgem como propostas de solução [Medvidovic e Taylor, Shaw et al. 1995].

A partir da ADL é feita a descrição de uma arquitetura. Diversas ADLs e ferramentas de suporte têm sido desenvolvidas com o objetivo de modelar arquiteturas, algumas voltadas para aplicações de domínios particulares, outras de propósito geral [Medvidovic e Taylor]. As ADLs fornecem um *framework* conceitual e uma sintaxe concreta para a caracterização de arquiteturas de *software*. Através delas descreve-se a estrutura dos componentes utilizados, interconexões entre estes, e um estado inicial. Tipicamente, essas linguagens oferecem ferramentas para compilar, analisar ou simular descrições arquiteturais escritas em suas linguagens associadas.

Quando uma descrição da arquitetura é utilizada, os componentes do sistema e sua estrutura de interconexão são selecionados e especificados pelo projetista, e durante o processo de configuração esses componentes e conectores devem ser criados. A partir da descrição da arquitetura é possível instanciar sistemas através do mapeamento das entidades arquiteturais em construções do nível de implementação.

O sistema de *software* configurado pode ter seus componentes inicializados, paralisados, removidos, substituídos e reinicializados, dando suporte à seleção, especificação e organização topológica dos componentes, ou seja pode ser feita a programação da configuração.

No código 2.1, é mostrado um exemplo de descrição arquitetural utilizando a ADL CBabel (*Building Applications By Evolution With Connectors*) [Sztajnberg 2002] empregada no ambiente CR-RIO (*Contractable Reflective-Reconfigurable Interconnectable Objects*) [Curty 2002, Ansaloni 2003], que será apresentado com maiores detalhes no capítulo 3. O exemplo descreve um sistema composto por dois componentes (um cliente e um servidor) e um conector que os interliga, através de suas portas, como o mostrado na figura 2.2.

Código 2.1 *Descrição de arquitetura na ADL Babel.*

```
1:  module ClienteServidor {
2:      port request;
3:      port provide;
4:      module Servidor {
5:          in port provide;
6:      } servidor;
7:      module Cliente {
8:          out port request;
9:      } cliente;
10:     connector C-S {
11:         in port request;
12:         out port provide;
13:     } cs;
14:     instantiate servidor, cliente, cs;
15:     link cliente to servidor by cs;
16: } cliente_servidor;
17: start cliente_servidor;
```

Em algumas ADLs, é permitido que as portas sejam suprimidas na instrução de ligação, assim o conector pode adaptar-se aos módulos envolvidos. Babel [Malucelli 1996], a partir da qual CBabel foi criada, é um exemplo desse tipo de ADL. Por isso, os conectores do ambiente CR-RIO são adaptáveis ao contexto (*context-reflective*), devido ao fato de que, quando colocados entre dois módulos, podem ser dinamicamente adaptados (em tempo de configuração) para intermediar as interações. Tal característica foi também contemplada na ADL CBabel. As ligações são feitas, com base em regras simples, que verificam se as portas podem ser consideradas complementares.

2.2 Contratos

A introdução de requisitos não-funcionais, ou requisitos de QoS, no projeto de uma aplicação, apresenta desafios na medida em que podem estar envolvidos recursos de *hardware* e *software*, potencialmente distribuídos em uma rede. Além disso, a introdução de requisitos de QoS, normalmente requer a programação de mecanismos específicos dos sistemas de suporte [Sztajnberg 2002].

Uma aplicação pode ter requisitos de QoS tão diversificados quanto comunicação, segurança ou tolerância a falhas. O programador desta aplicação teria que incluir, entre as suas tarefas, a programação de ações concretas sobre o sistema de suporte ou rede de comunicação, para impor e monitorar QoS.

O conhecimento desses requisitos no momento em que inicia-se um serviço permite a realização de verificações quanto à viabilidade do seu provimento. Tal medida pode garantir que a aplicação obtenha os recursos suficientes do sistema de suporte para o seu funcionamento correto, atendendo as suas restrições de qualidade. Para tanto, é necessária a existência de um formalismo para expressar as restrições sobre os serviços do sistema de suporte (requisitos de QoS) e um mecanismo que interaja com o sistema de suporte para o provimento do serviço requerido.

A definição de restrições para o comportamento dos mecanismos específicos dos sistemas de suporte pode ser realizada através da definição de contratos. Um contrato define uma formação comportamental de um conjunto de entidades comunicantes. Cada contrato especifica os seguintes aspectos dessa formação [Helm et al. 1990]:

- Identifica os participantes envolvidos e suas obrigações contratuadas (*Contractual Obligations*), que consiste nas obrigações de tipo (*Type Obligations*), onde os participantes devem prover variáveis e interface externa; e nas obrigações causais (*Causal*

Obligations), onde os participantes devem apresentar uma seqüência de ações caso certas condições sejam verdadeiras.

- Define invariantes que os participantes se esforçam em manter e que durante a execução podem ser violadas. Também são definidas ações que devem ser tomadas para satisfazer o invariante novamente.
- Define pré-condições para o estabelecimento do contrato e métodos que os instanciam.

Segundo [Beugnard et al. 1999] os contratos podem ser divididos em quatro tipos:

- Contratos Sintáticos: também conhecidos como contratos básicos ou de associação, são requeridos simplesmente para o sistema funcionar. Envolvem definições de que operações os componentes podem executar, os parâmetros de entrada e saída de cada componente e as possíveis exceções na operação. Na prática, podem consistir de definições de interfaces e podem ser descritos através de uma IDL (*Interface Definition Language*) ou mesmo por linguagens de programação usuais.
- Contratos de Comportamento: consistem de assertivas lógicas que servem de pré- e pós-condições para modelar o comportamento de um componente. Esses contratos possuem certo nível de negociabilidade, pois permitem que componentes tenham atuação flexibilizada de acordo com o ajuste das assertivas lógicas, de responsabilidade dos clientes que solicitam um serviço.
- Contratos de Sincronização: definem a sincronização entre chamadas de métodos. Seu foco é descrever as dependências entre serviços providos por um componente, tais como seqüência, paralelismo ou desordem.
- Contratos de QoS: especificam níveis esperados de comportamento do sistema em relação a suas propriedades. Os contratos de QoS podem ser especificados estaticamente através da enumeração das características dos serviços e alternativamente, pode-se implementar uma solução mais dinâmica que administra negociações entre as partes que utilizam e fornecem estes serviços.

O interesse desse trabalho é a utilização dos contratos de QoS no nível de arquitetura de *software*, para descrever requisitos não-funcionais de uma aplicação.

2.2.1 Contratos de QoS

Na concepção de sistemas com base em componentes, estes podem se comportar de duas formas: provendo um tipo de serviço (servidores) ou requerendo serviços oferecidos (clientes). Em alguns casos, os componentes podem assumir ambos papéis, de cliente e de servidor. A solicitação de um serviço é feita pelo cliente através de uma porta específica (porta de saída) para um servidor, que a recebe também por uma porta específica (porta de entrada). Em um determinado nível, tais solicitações são feitas por mensagens, e para que a troca de mensagens seja possível, é necessária a ligação entre a porta de saída do cliente e a porta de entrada do servidor, feita diretamente ou com o intermédio de algum(s) conector(es).

Normalmente, os serviços prestados são oferecidos segundo um determinado nível de qualidade, e clientes que requisitam serviços especificam que níveis de qualidade esperam encontrar nos serviços que forem utilizar. Sendo assim, há uma correlação entre o que é oferecido e o que é requerido, isso caracteriza o contrato de qualidade de serviço. Idealmente, a qualidade dos serviços oferecidos deve ser tão boa quanto o que os clientes requerem. Porém, em algumas aplicações, os clientes podem contentar-se com menos, desde que isso esteja bem especificado no contrato, dando margem ao processo de negociação do serviço prestado.

A especificação dos contratos é feita em função de determinadas características de qualidade, ou propriedades, que podem ser monitoradas nos serviços prestados, chamadas categorias de QoS. A escolha das propriedades a serem observadas pode refletir em decisões no nível de implementação dos serviços [Frolund e Koistinen 1998].

Em um contrato são especificados os atributos para as propriedades monitoradas e os níveis possíveis de serem atingidos (para prestadores de serviço) ou os níveis satisfatórios de serem obtidos (para requisitantes de serviço). Tais atributos podem ser simples valores numéricos, faixas de valores, porcentagens ou até distribuições de probabilidade. Políticas a serem aplicadas quando os requisitos de QoS não puderem ser cumpridos, devido a restrições do ambiente de execução, também podem ser definidas.

Tais políticas incluem definições de regiões de operação para os valores das propriedades, e as transições entre os serviços caso o nível de qualidade saia de uma determinada região e entre em outra. A transição entre serviços pode guiar a adaptação dinâmica da aplicação para acomodar as restrições do ambiente, uma vez que, para o estabelecimento de serviços, podem ser incluídas ações a serem executadas antes de sua instauração.

Os requisitos não-funcionais, ou de QoS, podem ser especificados em vários formatos [Jin e Nahrstedt 2004], como através de linguagens de descrição como QuAL [Florissi 1996], QDL [Pal et al. 2000], HQML [Gu et al. 2002], SLAng [Lamanna et al. 2003], Xelha/ RCDL [Duran-Limon e Blair 2004] e CBabel [Loques et al. 2004], de regras e funções de associação, como em Agilos [Li e Nahrstedt 1999] [Li 2000] ou mesmo por meio de funções de utilidade de recursos, como em Q-RAM [Rajkumar et al. 1997].

No código 2.2 é apresentado um contrato de QoS utilizando a linguagem CBabel, que descreve a inicialização de um serviço caso a exigência imposta, no mínimo 50% (0.50) de CPU disponível *(linhas 9-11)*, seja atendido. O não atendimento de tal exigência deixa o serviço indisponível, como descrito pela cláusula de negociação *(linhas 5-7)*. Esta cláusula está de acordo com a proposta apresentada em [Curty 2002], em que o estado *out_of_service* é declarado explicitamente na regra de transição. Porém, em [Corradi 2005] foi proposta a remoção desse estado, que seria um *default*, e o seu alcance automático em casos em que é necessária a negociação, mas não há nenhum serviço a ser provido. Assim, em situações de um único serviço no contrato, a cláusula de negociação poderia até ser omitida.

No capítulo 4 será apresentado um melhor detalhamento da semântica desses contratos utilizando CBabel, através de diversos exemplos de sua utilização. No capítulo 5 serão apresentadas algumas abordagens que tratam esta especificação, bem como uma comparação entre estas.

Código 2.2 *Um contrato de instanciação de um serviço em CBabel.*

```
1: contract {
2:     service {
3:         instantiate servidor at host with pProc;
4:     } sProcessing;
5:     negotiation {
6:         sProcessing → out_of_service;
7:     };
8: } captura_imagens;
9: profile {
10:     Processing.avail >= 0.50;
11: } pProc;
```

2.3 Sistemas de monitoramento

No contexto de prestação de serviços, a atividade de monitoramento permite identificar situações que violem o funcionamento esperado de um determinado serviço, e que poderiam degradá-lo ou mesmo impedir o seu oferecimento. Além disso, viabiliza a escolha de componentes mais adequados para a instanciação do serviço (ver seção 3.4). No código 2.2, a disponibilidade de capacidade de processamento da máquina deve ser monitorada, para verificação da possibilidade de manter o serviço ativo. As propriedades monitoradas podem ser registradas em um repositório para uso futuro (e.g. redimensionamento do contrato ou estabelecimento de prêmios ou multas) e, opcionalmente, usadas para guiar ou negociar algum procedimento adaptativo visando dar continuidade à provisão do serviço. A atividade de monitoramento, embora dirigida pelas especificações dos contratos, possui requisitos básicos que podem ser implementados por serviços independentes. Estes serviços de monitoração poderiam ser então reutilizados em diferentes contextos de aplicação.

A mensuração de recursos é uma questão fundamental em aplicações sensíveis à qualidade dos mesmos, envolvendo várias questões, tais como:

- A definição de ponto ou pontos (servidor, cliente, rede) a partir do(s) qual(is) as métricas devem ser coletadas.
- A indicação do responsável por coletar as métricas.
- O uso de coletores passivos (*packet sniffing*) ou realização de testes ativos explicitamente.
- Delimitação de que informações devem ser extraídas do coletor.

Essas questões permitem classificar as técnicas de monitoração em quatro categorias, como mostrado na figura 2.3, onde o componente MeCo (*Metric Collector*) representa o coletor de métricas usado para adquirir e armazenar as medidas de desempenho, no contexto do projeto Tapas [Molina-Jimenez et al. 2004].

A coleta de métricas realizada pelo próprio consumidor (ou cliente do serviço) é mostrada na figura 2.3(a). Tal esquema frisa que as métricas devem ser coletadas pela parte interessada, usuária do serviço. A figura 2.3(b) mostra a coleta das métricas feitas pelo provedor (ou servidor) do serviço, sendo o MeCo instalado ao lado do mesmo. No esquema da figura 2.3(c), nem o provedor nem o usuário coletam as métricas, que são

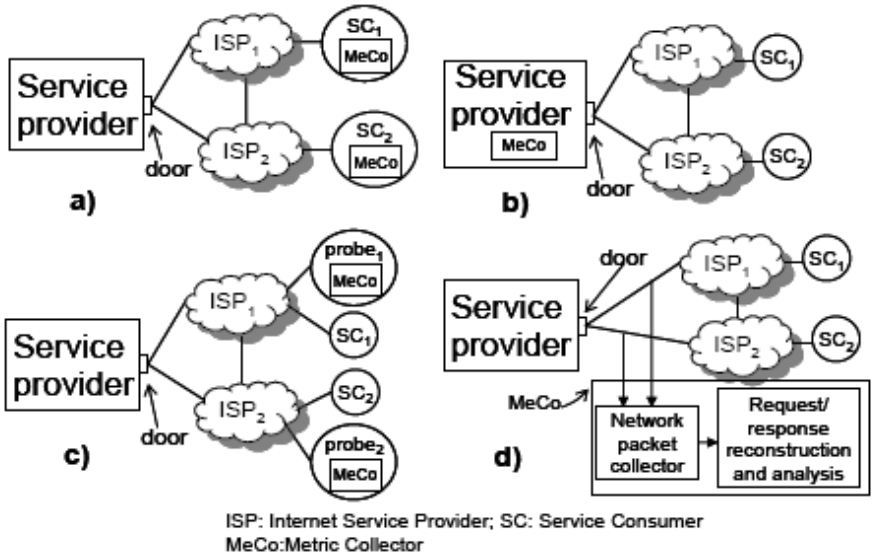

Figura 2.3: *Técnicas para coleta de métricas [Molina-Jimenez et al. 2004].*

adquiridas por uma entidade externa ($probe_1$ e $probe_2$ na figura) confiável tanto para o provedor quanto para o consumidor. Do ponto de vista de sua funcionalidade, os *probes* são clientes estrategicamente posicionados e habilitados com mecanismos para a coleta de métricas. A idéia do último esquema, mostrado na figura 2.3(d), é instalar um MeCo em algum lugar no caminho entre o provedor e o consumidor para coletar todos os pacotes que estejam entrando ou saindo do provedor. Como o MeCo não está instalado no provedor nem no consumidor, ele pode ser visto como uma terceira parte confiável, como no esquema mostrado pela figura 2.3(c). Uma discussão sobre ferramentas para monitoração de recursos da infra-estrutura de suporte, bem como suas classificações, segundo essas técnicas, será apresentada na seção 3.3.

Como assumimos que a interação entre o provedor do serviço e seu consumidor é "regulamentada" por contratos, como o especificado no código 2.2, que estipulam as obrigações que os prestadores do serviço devem honrar, a monitoração possui papel importante na certificação do atendimento dos serviços, junto com uma outra entidade que faz a conferência dos valores mensurados com os valores desejados. Para muitas aplicações, uma degradação no nível de QoS oferecido ao consumidor pode tornar o serviço inutilizável, sendo de interesse dos participantes do serviço monitorar o nível de QoS para poder garantir a entrega do mesmo, ou para poder adaptar a aplicação de forma a possibilitar a operação sobre diferentes níveis de qualidade de serviço. Conceitos relativos à adaptação serão mostrados na próxima seção.

2.4 Sistemas autônomos

As aplicações de rede, como navegação, vídeo conferência, envio de mensagens instantâneas, compartilhamento de arquivos, jogos *online* e outros serviços fornecidos em servidores tornam-se, a cada momento, mais necessárias para um número cada vez maior de pessoas. Da perspectiva do usuário, essas aplicações são usadas para acessar serviços oferecidos pelos provedores de serviços através da Internet. Uma grande parte desses serviços são integrados estaticamente, isto é, em tempo de projeto um provedor de serviço disponibiliza uma configuração do serviço consistindo dos recursos e componentes apropriados para seu provimento. O maior problema com tais serviços é que eles não lidam bem com variações nas características do sistema e nos requisitos dos usuários [Huang 2004].

A auto-configuração é uma abordagem capaz de resolver tal limitação, pois um serviço auto-configurável é capaz de encontrar automaticamente uma configuração de serviço "otimizada" de acordo com as características do ambiente e as necessidades do usuário. Em [Curty 2002, Moura et al. 2002, Atighetchi et al. 2003, Cheng et al. 2004] são propostas infra-estruturas para o desenvolvimento de aplicações distribuídas capazes de adaptar-se em casos de mudanças em seu ambiente de execução.

Adicionando à auto-configuração, as capacidades de auto-reparação, auto-otimização e auto-proteção, que serão melhor detalhadas na próxima seção, temos o conceito de sistemas autônomos. Segundo [IBM] algumas condições definem um sistema autônomo:

- O sistema deve se conhecer em termos de: que recursos ele tem acesso, quais são suas capacidades e limitações, e como e porque ele está conectado a outros sistemas.
- O sistema deve ser capaz de automaticamente se configurar e re-configurar de acordo com as mudanças no ambiente em execução.
- O sistema deve ser capaz de monitorar suas partes de modo a ajustar seu desempenho, para manter sua atividade e alcançar objetivos pré-definidos.
- O sistema deve ser capaz de trabalhar mesmo encontrando problemas que podem causar o mau funcionamento de algumas de suas partes.
- O sistema deve detectar, identificar e proteger-se contra ataques para manter sua segurança e integridade.
- O sistema deve ser capaz de adaptar-se ao ambiente de acordo com as mudanças (e.g. falhas, sobrecarga de recursos) ocorridas, interagindo e estabelecendo protocolos de

comunicação com sistemas vizinhos.

- O sistema deve funcionar em ambientes heterogêneos e implementar padrões abertos, ou seja, um sistema autônomo não pode, por definição, ser uma solução proprietária.
- O sistema deve prever e reservar os recursos necessários de forma otimizada e transparente para o usuário.

2.4.1 Fundamentos da computação autônoma

O conceito de aplicações auto-gerenciáveis foi encorajado inicialmente por esforços da IBM em 2001 ao publicar uma iniciativa de computação autônoma [IBM 2001]. Em [Hewlett-Packard 2003, Microsoft 2004] a HP e a Microsoft, respectivamente, reconheceram a necessidade do auto-gerenciamento para o futuro das aplicações e ambientes de TI (Tecnologia da Informação).

Como citado anteriormente, de forma a incorporar o auto-gerenciamento aos sistemas, eles devem possuir quatro características fundamentais: auto-configuração, auto-reparação, auto-otimização e auto-proteção [Ganek e Corbi 2003].

2.4.1.1 Auto-configuração (*Self-configuring*)

Os sistemas devem adaptar-se automaticamente a mudanças dinâmicas no ambiente. Quando *hardware* e *software* são capazes de definirem-se *on-the-fly*, ou seja, em tempo de execução, são chamados auto-configuráveis. O aspecto de auto-gerenciamento indica que novas características, *softwares* ou servidores podem ser dinamicamente adicionados à infra-estrutura sem interromper o serviço e com o mínimo de intervenção humana.

2.4.1.2 Auto-reparação (*Self-healing*)

Para ser auto-reparável o sistema deve ser capaz de recuperar-se de falhas em um componente faltoso, detectando e isolando tal componente. Caso seja possível a restauração de tal componente, este deve ser recuperado e reintegrado, caso contrário, ele deve ser substituído sem causar interrupção no serviço. Nesses sistemas, há a necessidade de prever problemas e tomar atitudes que previnam o impacto de falhas na aplicação. O objetivo da auto-reparação deve ser minimizar a interrupção da aplicação, mantendo-a disponível ininterruptamente. Os desenvolvedores de componentes do sistema devem focar em maximizar a confiabilidade e disponibilidade de cada *hardware* e *software* objetivando manter

uma disponibilidade contínua.

2.4.1.3 Auto-otimização (*Self-optimizing*)

É a capacidade de monitorar e ajustar os recursos automaticamente. Requer sistemas de *hardware* e *software* capazes de conhecer as necessidades do usuário e maximizar a utilização de *software* eficientemente, sem intervenção humana. Os recursos, idealmente, devem formar uma coleção de recursos computacionais que devem ser administrados por um gerenciador de carga lógico. A redistribuição dinâmica da carga para o sistema deve ser permitida pelo gerenciamento da carga e pela alocação de recursos, que possuem mecanismos necessários para conhecer os requisitos de carga. De forma similar, dispositivos de armazenamento, base de dados, redes, e outros recursos devem estar em constante ajuste possibilitando operações eficientes mesmo em ambientes imprevisíveis.

2.4.1.4 Auto-proteção (*Self-protecting*)

Os sistemas devem prevenir, detectar, identificar e proteger-se de ataques. Devem ser capazes de definir e gerenciar os direitos de acesso do usuário para todos os recursos computacionais, protegendo-os contra acessos desautorizados a tais recursos. Esses sistemas também precisam detectar e relatar invasões, e prover *backups* e capacidade de restabelecimento de serviços.

2.4.2 Acréscimo da computação autônoma

Os sistemas de gerenciamento, de forma a seguir a tendência da computação autônoma, podem conter mecanismos aprimorados de determinação de problemas e coleta de dados para ajudar na prevenção de interrupções e detecção de brechas de segurança. Tais sistemas devem manter transparente para o usuário os elementos de *hardware* e *software* utilizados para tal feito.

A base para os sistemas autônomos é formada pela intercomunicação de elementos autônomos que são responsáveis pelo próprio gerenciamento, ou seja, configuração, reparo, otimização e proteção a ataques.

Os elementos autônomos monitoram o comportamento do sistema (ou do componente) utilizando sensores ou agentes e fazem os ajustes necessários a partir de configuradores. Através da monitoração feita pelos agentes, da análise desses dados, do planejamento de que atitude deve ser tomada (se alguma for necessária), e da execução desta através dos

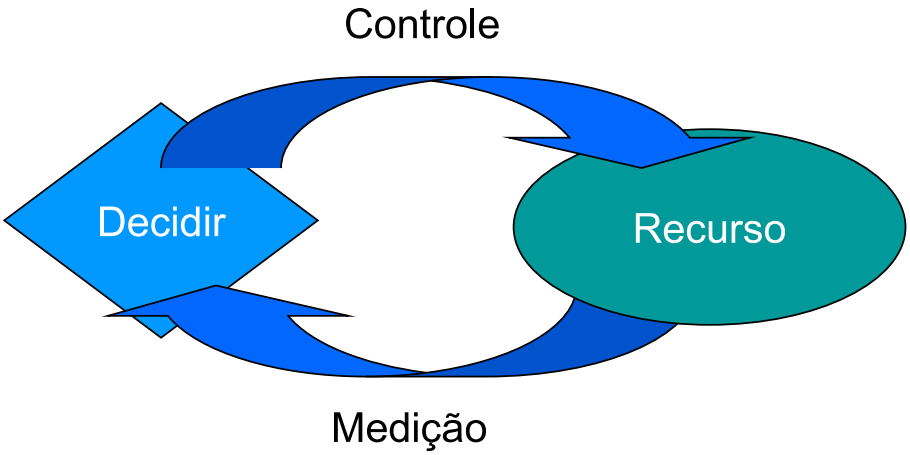

Figura 2.4: *Laço de controle formado pelas ações que compõem a computação autônoma.*

configuradores, um laço de controle é criado como pode ser visto na figura 2.4. Esse laço resume as principais atividades da computação autônoma. Há a medição dos recursos, cujos dados monitorados são passados para um mecanismo de decisão, que os analisa e decide que ações devem ser tomadas e passa essas determinações para um controlador que possui instrumentos para efetivá-las no recurso em questão.

2.5 Contexto da proposta

Diante dos conceitos apresentados neste capítulo, é contextualizada a proposta deste trabalho, que possibilita descrever, desenvolver e gerenciar aplicações baseadas em módulos da arquitetura, definidos em uma ADL. Tais aplicações, além dos requisitos funcionais, possuem requisitos não-funcionais, que podem ser descritos através de contratos textuais de alto nível, os contratos de QoS. Nesse contexto, o não cumprimento dos requisitos não-funcionais, constantemente monitorados, estabelece a necessidade da adaptação da aplicação ou de sua infra-estrutura, guiada por informações extraídas dos contratos.

A adaptação geralmente está associada ao estilo arquitetural da aplicação. Dado um estilo particular, haverá um conjunto de operações para efetuar mudanças na configuração da arquitetura [Garlan et al. 2003]. Em casos mais gerais, operações de adição e remoção de componentes e conectores devem ser providos. Entretanto, para estilos arquiteturais específicos, operações de mais alto nível podem ser definidas. Como por exemplo, em um estilo cliente-servidor, o suporte a operações de replicação de servidor pode a melhorar o desempenho, enquanto que em um estilo *pipe-filter*, pode ser suportada a adição de filtros para compactar o dado em um *pipe* objetivando o aumento de desempenho.

Apesar da necessidade de definir operações de adaptação para cada estilo, isso é feito somente uma vez por estilo. Utilizando novamente o exemplo do estilo cliente-servidor, a operação de mover um cliente poderá ser a mesma para diversas aplicações nesse mesmo estilo. Podem também ser fornecidas adaptações de mudança do comportamento do sistema, através de mudanças de suas propriedades específicas, como por exemplo alterar o protocolo de comportamento interno de um conector.

Tais operações de adaptação são descritas através dos contratos de QoS em um alto nível de abstração. Esses contratos são de fundamental importância no contexto de sistemas autônomos, pois guiam as modificações a serem feitas nas aplicações visando manter a obediência aos requisitos do usuário, também descritos nos contratos. No ambiente CR-RIO, que abrange alguns conceitos de sistemas autônomos, os contratos são fundamentais para permitir que as especificações das necessidades do usuário, as políticas de adaptação e o comportamento da aplicação sejam feitos em um nível alto.

2.6 Conclusão do capítulo

Neste capítulo foi apresentada uma breve revisão de conceitos referentes a arquitetura de *software*, contratos e sistemas autônomos. As arquiteturas de *software* possibilitam, dentre outros benefícios, a separação de interesses, essencial na busca da reutilização. Os requisitos funcionais e não-funcionais podem ser separados através do processo de configuração, no qual os componentes podem implementar requisitos funcionais e os conectores podem implementar requisitos não-funcionais. Dessa forma a evolução dos sistemas é facilitada, fazendo com que os módulos possam ser adicionados, removidos ou mesmo modificados sem que os demais elementos precisem, necessariamente, ser alterados.

Foram apresentados os quatro tipos de contratos, de acordo com a classificação descrita em [Beugnard et al. 1999] e identificado o interesse específico em contratos de QoS para este trabalho, bem como a conveniência de sua introdução no nível arquitetural.

Foram mostradas características fundamentais dos sistemas autônomos que viabilizam a manutenção de sistemas que estão cada vez mais complexos, heterogêneos e com grande complexidade de *hardwares* e *softwares*. Além disso, foi descrito como a proposta se encaixa nos conceitos apresentados. No próximo capítulo será apresentado o *framework* CR-RIO, que contempla os conceitos de arquiteturas de *software* descritos neste capítulo, possui uma linguagem de descrição arquitetural e de requisitos não-funcionais da aplicação, possui entidades monitoradas, além de atender algumas características de sistemas

autônomos.

Capítulo 3

Proposta

Este capítulo apresenta o contexto de aplicações com requisitos de QoS, e discute um possível suporte para o gerenciamento dessas aplicações para que não seja necessária a intervenção humana. Inicialmente, é apresentada uma potencial infra-estrutura geral de suporte concebida com o intuito de identificar as atividades envolvidas no provimento desse tipo de gerenciamento. Em seguida, é mostrada uma proposta que contempla essa infra-estrutura e são expostos seus componentes. Por último, são apresentadas algumas ferramentas de monitoramento, que podem auxiliar na manutenção da qualidade do serviço prestado e uma implementação de um módulo de seleção de recursos.

3.1 Gerenciando aplicações com requisitos de qualidade de serviço

Observando algumas propostas existentes, como [Loyall et al. 1998, Pal et al. 2000, Wang et al. 2001, Ansaloni 2003, Garlan et al. 2004], que estão no escopo do gerenciamento de aplicações com requisitos de QoS, percebemos alguns elementos comuns a elas, como a especificação dos níveis de qualidade desejados, a descrição de alterações a serem feitas quando esses níveis forem violados, a utilização de mecanismos de monitoração de recursos sobre os quais os níveis de qualidade são declarados e a inclusão de um suporte que permita efetivação de adaptações no sistema quando necessárias. Esses elementos, bem como suas funcionalidades são descritos a seguir:

- Especificação: permite a descrição dos requisitos não-funcionais de uma aplicação e das configurações necessárias para o estabelecimento dos serviços da aplicação com a qualidade desejada;

- Monitoramento: efetua a monitoração dos níveis dos recursos utilizados pela aplicação, ou da própria aplicação;
- Gerenciamento: gerencia os serviços requeridos pela aplicação, levando em consideração o nível de qualidade almejado e os níveis dos recursos;
- Configuração: efetua as configurações necessárias no sistema para impor e suportar os serviços oferecidos pela aplicação, assim como a alocação dos recursos a serem utilizados.

3.1.1 Especificação

Através da especificação é possível conhecer os níveis de qualidade de uma aplicação, orientar o monitoramento dos recursos por ela utilizados e determinar as configurações necessárias para o estabelecimento dos serviços oferecidos por ela.

Em [Florissi 1996, Pal et al. 2000, Gu et al. 2002, Lamanna et al. 2003] e também em [Duran-Limon e Blair 2004, Loques et al. 2004] são apresentadas algumas linguagens para descrição de serviços, com seus aspectos não-funcionais úteis para compor aplicações a partir de componentes mais primitivos. A descrição de um serviço contém as informações referentes às configurações necessárias para que ele possa ser estabelecido. O uso destas linguagens permite a elaboração de contratos, os quais visam formalizar as exigências básicas para a instalação e operação desses serviços. Os contratos definem perfis de qualidade - que especificam os níveis desejados dos requisitos não-funcionais - a serem estabelecidos ou mantidos, permitindo o conhecimento das propriedades não-funcionais que devem ser monitoradas durante a etapa de funcionamento de uma aplicação.

3.1.2 Monitoração

A monitoração é o elemento responsável pela realização da coleta de informações referentes às propriedades dos recursos do sistema que são utilizadas pelos serviços. Ou seja, deve fornecer acesso aos mecanismos de nível de sistema e gerar eventos de interesse sobre a utilização destes, permitindo explorar a disponibilidade dos recursos ou reagir à mudanças em suas utilizações. Algumas ferramentas que realizam tal tarefa serão abordadas na seção 3.3, e a partir das informações de monitoração geradas por elas é possível verificar se os níveis dos recursos satisfazem aos requisitos de QoS desejados, os quais são descritos na especificação.

3.1.3 Gerenciamento

Gerenciar aplicações com requisitos não-funcionais concerne à coordenação das atividades referentes ao processo de estabelecimento dos serviços oferecidos por ela, observando e respeitando os níveis de qualidade desejados pelos seus usuários (clientes). Essa observação aos níveis de qualidade é feita através do elemento de monitoração, que repassa as informações relativas aos recursos utilizados pela aplicação possibilitando a verificação de sua conformidade com o especificado. E o respeito aos níveis desejados é possível através da coordenação a um mecanismo capaz de ajustar a aplicação ou sua infra-estrutura de suporte. Em [Curty 2002] é proposto um *framework* que permite a especificação e suporte de contratos de QoS. Esse contratos são associados aos componentes da arquitetura de *software* de uma aplicação. Na seção 3.2 serão abordados maiores detalhes deste *framework*.

3.1.4 Configuração

Para lidar com o requisito de adaptação dinâmica de uma aplicação perante mudanças no estado dos recursos por ela utilizados, é conveniente o uso de algum mecanismo que ofereça esse suporte. Tal mecanismo deve permitir o gerenciamento da aplicação de forma que sua infra-estrutura, ou ela própria possam ser dinamicamente alteradas, sem a necessidade de parar totalmente os serviços por ela oferecidos. Uma abordagem é a adoção do modelo de Arquitetura de Software, que considera a possibilidade de uma aplicação ser manipulada a partir do nível arquitetural, onde são representados seus componentes e suas interligações [Shaw et al. 1995]. Com base nesse conceito, [Lisbôa 2003] propôs um ambiente de suporte a configuração de arquitetura de software. Sua estrutura é capaz de, a partir da interpretação de comandos presentes na descrição arquitetural, escrita em uma ADL, obter informações sobre a configuração arquitetural do sistema. A partir dessas informações, o ambiente instancia os módulos (componentes e conectores) do sistema, realizando as ligações entre eles e efetivando a execução do sistema.

Em alguns casos, para o processo de configuração de um sistema, pode ser necessário utilizar mecanismos que permitam localizar recursos que são oferecidos no ambiente distribuído de uma aplicação. Como por exemplo, buscar por uma máquina que possua uma determinada capacidade de processamento disponível, na qual poderá ser instalado e provido um serviço. Na seção 3.4 serão discutidos tais mecanismos e suas possíveis utilizações. A próxima seção descreve a proposta CR-RIO, um *framework* que permite gerenciar aplicações com requisitos de QoS, descritos através de contratos no nível arquitetural de

software.

3.2 CR-RIO

O CR-RIO (*Contractual Reflective - Reconfigurable Interconnectable Objects*) [Curty 2002, Ansaloni 2003, Loques et al. 2004, Corradi 2005] é um *framework* projetado para prover suporte à especificação e ao gerenciamento de contratos de QoS para aplicações. Com a utilização de uma ADL, o CR-RIO permite a descrição da configuração arquitetural de uma aplicação, em termos de seus módulos e a interligação entre eles, associada à descrição de um contrato de QoS que define os níveis de qualidade desejados pelos serviços da aplicação. Através do uso da meta-informação contida nessas descrições, é possível orientar as adaptações na aplicação ou em sua infra-estrutura de suporte, em intervalos em que a qualidade dos recursos utilizados violam os níveis aceitáveis declarados no contrato.

O gerenciamento do contrato é realizado automaticamente segundo um padrão arquitetural que pode ser diretamente mapeado em componentes específicos inclusos no ambiente de suporte. Isto permite ao projetista escrever um contrato e seguir uma receita padrão para inserir o código adicional requerido para a sua implantação no ambiente de suporte.

Inicialmente proposto em [Curty 2002], o *framework* foi alvo de alguns refinamentos que são explicitados em [Ansaloni 2003, Corradi 2005]. A seguir os componentes do CR-RIO (com a adoção das modificações) serão mostrados.

3.2.1 Descrição

A descrição no CR-RIO abrange o papel da especificação tratada na seção anterior. A descrição da arquitetura permite expor a configuração arquitetural e a descrição dos contratos de QoS, além de permitir especificar os níveis de qualidade desejados para os serviços de uma aplicação, juntamente com os requisitos não-funcionais necessários à garantia da qualidade desejada.

3.2.1.1 Descrição arquitetural

No capítulo anterior, foi apresentado um exemplo simples de uma descrição arquitetural, utilizando a ADL CBabel para uma aplicação cliente-servidor. Agora, mostraremos a descrição da arquitetura utilizada nos exemplos que serão desenvolvidos no capítulo 4.

Nossa arquitetura inicial é composta por três elementos básicos: o cliente (`Client`), o servidor (`Server`) e o grupo de servidores (`ServerGroup`), como apresentado na descrição arquitetural presente no código 3.1. Cada componente servidor nessa arquitetura possui uma interface externa que exemplificamos com uma interface para o provimento do serviço de cálculo de pi *(código 3.1, linhas 3-5)*, definida através da porta de entrada (`in port`), cujo papel é prover o serviço com a assinatura exposta na linha 2. O módulo cliente também utiliza dessa interface para requerer (`out port`) o serviço *(código 3.1, linhas 6-8)*. A ligação entre o cliente e o servidor é intermediada pelo conector de grupo, que possui uma porta de entrada, na qual o cliente está conectado, e outra de saída, na qual o servidor se conecta *(código 3.1, linhas 9-12)*. A configuração inicial começa com a instanciação de um grupo de servidores *(código 3.1, linha 13)* e de um conjunto de três servidores *(código 3.1, linhas 14-17)* associados a esse grupo, e logo é feita a ligação de cada um deles ao grupo. Depois, o cliente é instanciado e unido ao grupo *(código 3.1, linhas 18-19)*. Os componentes `Client` e `Server` são mapeados, respectivamente, em códigos correspondentes às funcionalidades de um cliente acessando um serviço em um servidor remoto e este prestando um serviço. O conector de grupo (`ServerGroup`) representa a ligação entre essas partes através de uma comunicação de grupos como a provida pelo *JGroups* [JGroups], usado em nossos experimentos. A linha 21 estabelece que essa arquitetura é regida pelo contrato `csr` (códigos 4.2 e 4.3).

Para facilitar a descrição da arquitetura, o tipo de porta utilizado nos componentes foi declarado separadamente *(código 3.1, linha 2)* e optou-se por criar referências particulares a estas portas dentro dos módulos, para não haver ambigüidade na descrição das ligações entre os mesmos. A verificação de compatibilidade de assinatura de interfaces dos componentes interligados pode ser feita estaticamente, analisando-se o *script* de configuração, ou dinamicamente, quando a interligação de dois módulos for realizada por um agente de configuração. Elementos cujas interfaces possuem assinaturas casadas podem ser automaticamente interconectados. Por outro lado, problemas de incompatibilidade de assinaturas podem ser superados com a utilização de um conector que implemente uma ponte [Wegner 1996] capaz de fazer a tradução adequada.

Maiores detalhes sobre os aspectos formais da descrição arquitetural através da linguagem CBabel e seu BNF podem ser encontrados nos apêndices B e A de [Sztajnberg 2002], respectivamente.

Código 3.1 - *ADL.*

```
1:  module ClientServer{
2:      port long pi (int numberOfInteractions); // tipo de porta
3:      module Server{ // classe do componente servidor
4:          in port pi provide;
5:      }
6:      module Client { // classe do componente cliente
7:          out port pi request;
8:      }
9:      connector ServerGroup { // classe do conector de grupo de servidores
10:         in port pi receive;
11:         out port pi forward;
12:     }
13:     instantiate ServerGroup serverGroup;
14:     for numServers=1 to 3{
15:         instantiate Server s[numServers];
16:         link serverGroup.forward to s[numServers].provide;
17:     }
18:     instantiate Client c;
19:     link c.request to serverGroup.receive;
20: }cliente_servidor;
21: start cliente_servidor with csr;
```

3.2.1.2 Descrição contratual

A descrição dos requisitos não-funcionais através dos contratos de QoS (*Contracts*), como dito no capítulo anterior, permite informar o uso que uma aplicação irá fazer dos recursos compartilhados e as variações aceitáveis sobre a disponibilidade desses recursos. Além de delinear ações a serem tomadas, caso essas variações de disponibilidade excedam os valores aceitáveis. No capítulo 4 diversos exemplos de contratos serão mostrados e analisados permitindo um melhor entendimento destes.

Um contrato é imposto em tempo de execução através de um conjunto de componentes de suporte (descritos a seguir) que mapeiam a semântica do contrato em elementos computacionais.

3.2.2 Infra-estrutura de gerenciamento

A infra-estrutura de gerenciamento do *framework* é composta pelos seguintes componentes: *Contract Manager* (Gerente de contratos) e *Contractor* (Contratador), que interagem com o *Configurator* (Configurador, do elemento configuração) e com *Resource Agent* (Agente de Recurso, do elemento monitoração), respectivamente. Através da figura 3.1, são ilustrados esses componentes nas principais atividades da computação autônoma, como dito na seção 2.4.

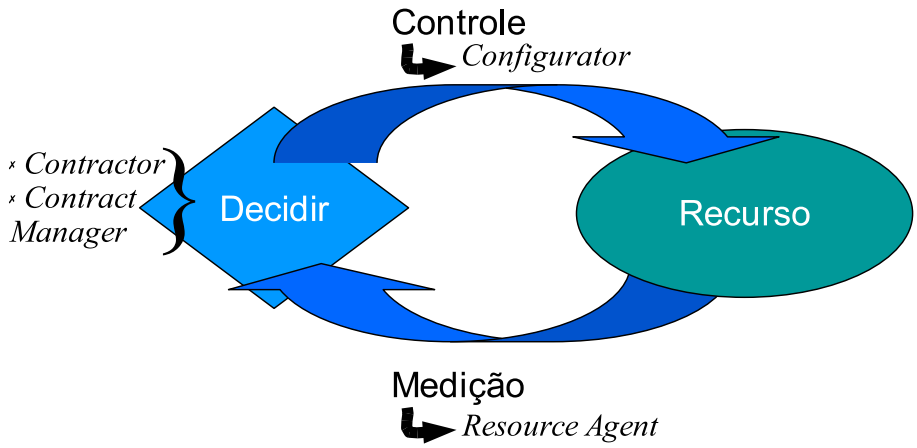

Figura 3.1: *Componentes CR-RIO e o laço de controle formado pelas ações que compõem a computação autônoma.*

3.2.2.1 *Contract Manager*

O *Contract Manager* (CM) é o responsável por interpretar a descrição do contrato, extraindo os serviços, suas restrições e a máquina de estados de negociação de serviços. Com essas informações, ele tenta estabelecer um serviço de acordo com a ordem dos serviços declarada na cláusula de negociação. Para tal, ele solicita aos *Contractors* a verificação das restrições exigidas pelo serviço. Caso sejam atendidas, as configurações do serviço (primitivas arquiteturais descritas) são passadas para o *Configurator* e o serviço é estabelecido. Após seu estabelecimento, se algum recurso utilizado pelo serviço não atender mais às especificações desejadas, o CM é avisado. Este por sua vez consulta novamente a cláusula de negociação, para identificar a existência de um próximo possível serviço.

Caso exista, ele tenta estabelecer o serviço cujas restrições sejam compatíveis com os atuais níveis de recursos, obedecendo à ordem de preferência descrita na cláusula de negociação de serviços. Na ausência de serviços a serem providos, o estado fora de serviço (*out_ of_ service*) é alcançado. O gerenciamento do contrato é encerrado e o CM notifica a aplicação sobre a escassez de recursos para os serviços requeridos, podendo o *Configurator* finalizar a execução da aplicação.

3.2.2.2 *Contractor*

O *Contractor* coordena a monitoração das propriedades de recursos especificadas nas restrições associadas aos serviços. Tal atividade consiste em enviar requisições de solicitação dos valores dessas propriedades aos *Resource Agent* e comparar as restrições do serviço corrente com os valores monitorados. Se alguma restrição do serviço corrente for violada, o *Contractor* notifica ao CM.

3.2.3 Configuração

A efetivação das configurações no ambiente CR-RIO é responsabilidade do *Configurator*, que tem acesso aos recursos no nível do sistema de suporte.

3.2.3.1 *Configurator*

O *Configurator* efetiva as configurações, sejam elas na aplicação ou na infra-estrutura de suporte, requeridas pelos serviços descritos em um contrato. Para isso, ele mapeia as primitivas arquiteturais em ações do nível de configuração (ou seja, no nível do sistema de suporte). Como por exemplo, mapear primitivas de instanciação de servidores na ação de adicionar uma nova réplica. Neste contexto, o ambiente de suporte a gerenciamento de configuração arquitetural proposto em [Lisbôa 2003] pode ser utilizado como *Configurator*. Como esse ambiente se baseia na ADL CBabel para descrever as configurações requeridas por uma aplicação (implantação e interligação de seus componentes e conectores), as configurações arquiteturais descritas nos serviços de um contrato poderiam ser repassadas diretamente para esse ambiente, sem a necessidade de mapeamentos adicionais.

3.2.4 Monitoração

A monitoração no contexto do CR-RIO é feita através da interação do componente *Resource Agent* com ferramentas que efetuam tal tarefa, como as descritas na seção 3.3.

3.2.4.1 *Resource Agent*

O *Resource Agent* encapsula o acesso aos mecanismos que executam ou fornecem suporte para a execução das atividades que compõem os serviços de uma aplicação, provendo interfaces para efetivar a alocação de recursos, iniciar serviços locais do sistema e monitorar os valores de propriedades requeridas. No processo de gerenciamento de contratos, os *Resource Agents* são responsáveis pela monitoração das propriedades de recurso sobre as quais foram definidas as restrições de qualidade. Os valores monitorados são passados para o *Contractor* quando uma mudança significativa é detectada, disparando no *Contractor* o processo de verificação de validade das restrições associadas ao serviço corrente.

A interação entre os componentes *Contract Manager*, *Contractor*, *Configurator* e *Resource Agent* é ilustrada na figura 3.2

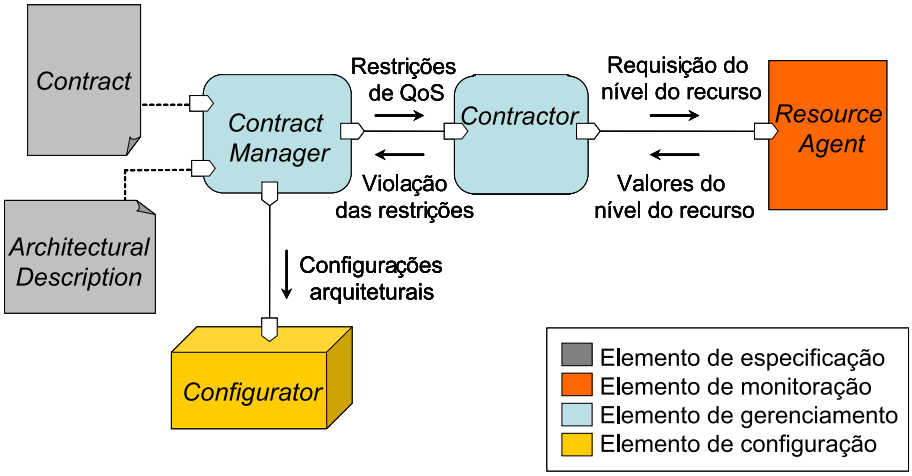

Figura 3.2: *Arquitetura de suporte do framework CR-RIO.*

3.3 Ferramentas de monitoramento

Com a adição dos requisitos de qualidade às aplicações, o monitoramento de parâmetros de elementos da infra-estrutura de suporte a serviços se torna de suma importância. Ele permite verificar se o estado do sistema satisfaz aos requisitos mínimos para que um serviço de uma aplicação possa ser iniciado com a qualidade desejada. Além disso, para

muitas aplicações, uma degradação no nível de QoS oferecido ao consumidor pode tornar o serviço inutilizável. Assim, é de interesse dos participantes do serviço monitorar o nível de QoS para poder garantir a entrega do mesmo, ou para poder adaptar a aplicação ou sua infra-estrutura de suporte de forma a ser possível operar sobre diferentes níveis de qualidade de serviço. Nesse sentido, conforme discutido na seção anterior, contratos podem ser usados pelas aplicações para informar às entidades participantes o nível de QoS requisitado e, adicionalmente, algumas estratégias de adaptação a serem usadas.

A tarefa de monitoração consiste na coleta de métricas como as de desempenho, por exemplo, tempo de resposta, latência, largura de banda, *jitter*, utilização de *buffers*, porcentagem de uso de CPU, necessárias para verificar se os níveis de QoS requeridos são atendidos, na implantação de um serviço e ao longo de seu funcionamento. Tal monitoração freqüentemente deve ser realizada com a ajuda de uma entidade externa, de forma que os resultados sejam confiáveis tanto para o provedor de recurso quanto para o usuário do mesmo.

Abordaremos a seguir algumas ferramentas de monitoração que se encaixam no propósito deste trabalho e logo escolheremos uma, de acordo com suas características, para ser utilizada em aliança ao *framework* nas simulações propostas. Incluímos um resumo deste estudo das ferramentas no apêndice A para consulta de trabalhos futuros, como [Cardoso 2005], onde há a proposta de integrar as informações obtidas através de diversas ferramentas de monitoração e acessá-las através de uma interface padronizada.

O NWS (*Network Weather System*) é um sistema distribuído implementado para coletar informações sobre os recursos e produzir estimativas de desempenho de curto prazo, com base nos dados das medições de desempenho feitas anteriormente. O objetivo do sistema é caracterizar e prever dinamicamente o desempenho que poderá ser fornecido no nível de aplicação, a partir de um conjunto de recursos computacionais e de rede. Tais estimativas de desempenho têm sido úteis, por exemplo, para implementar agentes de escalonamento dinâmico para aplicações de meta-computação [Wolski et al. 1999].

O Remos (*REsource MOnitoring System*) foi desenvolvido com o propósito de disponibilizar informações sobre os recursos para aplicações distribuídas. É capaz de capturar informações de diferentes tipos de rede e das máquinas pertencentes a elas, através do uso de vários coletores que usam diferentes tecnologias como SNMP (*Simple Network Management Protocol*) e *benchmarking* [Dinda et al. 2001]. Com a utilização de coletores apropriados para cada tipo de rede e provendo uma arquitetura para distribuir a saída desses coletores em um ambiente distribuído, o Remos coleta informações detalhadas de

cada localização e as distribui para todos os requisitantes de maneira escalável.

O Ganglia é um sistema de monitoração proposto para ambientes de alto desempenho como *clusters* e *grids*, que utiliza tecnologias como XML (*Extensible Markup Language*) para representação de dados, XDR (*External Data Representation standard*) para transporte dos dados de forma compacta e RRDtool (*Round Robin Database Tool*) para armazenamento e vizualização de dados [Massie et al. 2004]. Sua implementação se preocupa em minimizar *overheads* em cada um dos pontos de monitoração e maximizar a colaboração entre eles, e é suportada por vários sistemas operacionais e arquiteturas de processadores.

O *toolkit* NetLogger (*Networked Application Logger*) é proposto para análise de desempenho de sistemas complexos tais como aplicações cliente-servidor ou *multi-threaded* [Gunter e Tierney 2003]. Essa ferramenta combina eventos de rede, de *host* e da aplicação, provendo uma visão geral do sistema, o que facilita a identificação de pontos que influenciam o desempenho do sistema.

O GridRM (*Grid Resource Monitoring*) [Baker e Smith 2002] é um projeto de pesquisa que pretende prover uma interface padronizada para o acesso a diversos conjuntos de fontes de dados encontradas nos ambientes de *grid*, como alguns citados anteriormente (e.g. Ganglia, NetLogger, NWS, etc).

3.3.1 Considerações sobre os sistemas de monitoramento

Embora diversas ferramentas de monitoração tenham sido propostas ultimamente, como as citadas no decorrer desta seção, geralmente essas são de propósito muito geral. Em [Zanikolas e Sakellariou 2005] são apresentadas outras propostas de ferramentas para monitoração, porém a decisão sobre qual delas usar deve ser tomada para cada caso específico de forma a otimizar sua execução. Em casos onde apenas uma simples monitoração do *host* é necessária, sem precisar agregar os dados de vários agentes, uma solução mais leve é a implementação dos próprios agentes de monitoração a partir dos utilitários do sistema operacional. Dessa forma os *overheads* dos sistemas de monitoração são evitados.

Para nossas simulações, a necessidade era de um ferramenta que instrumentasse recursos de processamento e de transporte. Além dessas características, ela deveria ter conhecimento de parâmetros específicos da aplicação. Para tanto, utilizamos o NWS para nos fornecer as medições referentes aos recursos de processamento e comunicação, e implementamos a medição de uma propriedade referente a aplicação, o tempo de resposta.

Para a tomada de tal decisão, levamos em consideração a facilidade de aquisição da ferramenta, sua disseminação no mundo acadêmico (mais referenciada), sua facilidade de extensão e sua capacidade de evolução (ou seja, disponibilização de novas versões).

3.4 Ferramentas de descoberta de recursos

A caracterização dos níveis dos recursos é interessante, não apenas por informar o estado dos recursos utilizados no provimento do serviço, mas também para permitir a escolha de qual utilizar na inicialização do mesmo. Uma possível solução é obter uma lista de candidatos através de um serviço de descoberta de recursos, e depois aplicar alguma técnica de seleção, de modo a escolher os melhores candidatos na lista. Há algumas propostas que integram a descoberta de serviços e a mensuração de medidas relacionadas a eles.

3.4.1 NSSD - *Network-Sensitive Service Discovery*

O NSSD é um exemplo de ferramenta que permite a procura e a seleção de um recurso, ou mesmo um conjunto deles, que satisfaça algumas propriedades funcionais e não-funcionais desejadas. Considerando, por exemplo, uma partida de um jogo *online*, no qual os usuários necessitam de um servidor que o execute, portanto tal servidor deve satisfazer requisitos funcionais (e.g. suporte ao jogo específico) e além disso, pode ser necessário o suporte a requisitos não-funcionais (e.g. atraso máximo na rede) [Huang e Steenkiste 2003]. Para atender às necessidades desse cenário, o NSSD precisa obter a lista de servidores de jogo, obter as latências entre cada servidor e os jogadores, ordená-las e escolher a melhor.

O NSSD oferece uma API que permite ao usuário solicitar buscas locais para serviços específicos, usando métricas padronizadas de otimização. Alternativamente, o usuário pode solicitar a seleção de um conjunto de bons candidatos para cada serviço, de modo que possa aplicar algum método específico de seleção sobre os resultados, visando alguma otimização em nível global. As buscas locais envolvem a escolha de um único serviço, tal como selecionar um servidor de jogo que minimize a latência máxima para um grupo de usuários. Por outro lado, uma otimização global envolve a seleção coordenada de diferentes tipos de serviços, como por exemplo, em uma aplicação servidora de vídeo, selecionar um servidor de *streaming* e um *transcoder* de modo que o consumo de banda geral seja minimizado.

Para as buscas, o NSSD permite escolher as seguintes métricas: latência média e

máxima, largura de banda mínima e máxima e carga no servidor.

O código 3.2 ilustra uma função da API do sistema NSSD (maiores detalhes em [Huang 2004]). O argumento `service_properties` determina características do recurso (e.g. versão de protocolo utilizado), `target_list` informa para quais clientes essa consulta deve ser otimizada, `num_servers` especifica quantos servidores iguais são requeridos na solução, e `num_solutions` especifica quantas soluções locais devem ser retornadas. O demais informam restrições de latência (`latency_type` e `latency_constraint`), banda passante (`bw_type` e `load_constraint`) e carga de processamento (`load_constraint`).

As soluções são retornadas em `solution` na forma de endereços IP. Quando múltiplos servidores idênticos são requisitados, `mapping` especifica que servidor deve ser usado por cada cliente, e `fitness` define o valor de aptidão da resposta.

Código 3.2 - *API do NSSD.*

```
1: Input
2:     service_properties     // atributos do serviço
3:     target_list            // lista para os quais a solução deve ser otimizada
4:     num_servers            // número de servidores idênticos necessários
5:     num_solutions          // número de soluções necessárias
6:     latency_type           // MAX/AVG/NONE
7:     latency_constraint     // restrição/MINIMIZE/NONE
8:     bw_type                // MIN/AVG/NONE
9:     bw_constraint          // restrição/MAXIMIZE/NONE
10:    load_constraint        // restrição/MINIMIZE/NONE
11: Output
12:    solution               // a melhor solução
13:    mapping                // mapeamento usuário-servidor
14:    fitness                // aptidão da resposta
```

3.4.1.1 Funcionamento

A implementação do NSSD é baseada em trabalhos anteriores relacionados à descoberta de serviços e mensuração de medidas da rede. Como ilustrado pela figura 3.3, um simples processador de consultas NSSD (QP - *Query Processor*) pode ser construído em cima de uma infra-estrutura de descoberta de serviços (e.g. SLP - *Service Location Protocol*) [Guttman et al. 1999], e de um serviço que seja capaz de estimar propriedades de rede (e.g., GNP ou Remos), como a latência ou largura de banda entre os nós. Quando o

módulo QP recebe uma consulta NSSD (1), ele encaminha a parte funcional da consulta ao diretório de serviços (2). Esse diretório retorna a lista de candidatos que atende as propriedades funcionais especificadas na consulta (3). A partir desse ponto o módulo QP obtém as informações de rede necessárias (por exemplo, a latência entre cada candidato e um usuário X) através da infra-estrutura de mensuração da rede (4). Finalizando o processo, o QP computa a melhor solução a partir dos candidatos e a retorna para o usuário (5).

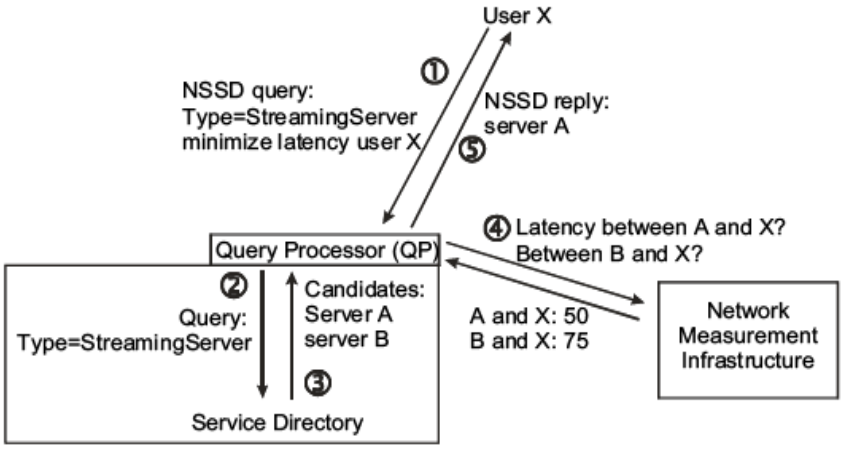

Figura 3.3: *Funcionamento do NSSD.*

3.4.1.2 Considerações

O NSSD é uma proposta para integrar a descoberta de serviços e a mensuração de medidas, mas que não está disponível livre e facilmente para uso. Em nossos experimentos, temos conhecimento que todos os nós são potenciais prestadores do serviço, então a etapa de descoberta dos candidatos se torna desnecessária. Dessa forma resta filtrar os melhores candidatos para o provimento do serviço. Portanto optamos por implementar um módulo de seleção de recursos simples que encaixava-se às nossas necessidades, explicitadas na próxima seção.

3.4.2 As reais necessidades

Para os nossos experimentos (abordados no capítulo 4), de forma a melhorar o desempenho das aplicações, necessitamos de uma ferramenta capaz de retornar o melhor recurso com base em algum parâmetro (e.g. o servidor com maior capacidade de processamento disponível). Os parâmetros para a escolha devem considerar as características de processamento e do enlace de comunicação. Ou seja, o interesse é retornar um servidor ou um

grupo deles que satisfaça algumas restrições relacionadas a tais características.

Como foi utilizado o NWS para obter informações sobre a utilização dos recursos, optamos por implementar nosso próprio módulo de seleção de recursos (que é nossa primeira contribuição nesse trabalho) utilizando as medições disponibilizadas por essa ferramenta.

3.4.3 Seleção de recursos

Visando suprir as necessidades identificadas, implementamos um módulo de escolha de recursos, que ao receber uma consulta, interage com a base de dados do NWS. Essa interação visa buscar por informações referentes às máquinas envolvidas na consulta e seu resultado é filtrado de acordo com o parâmetro recebido. As máquinas que são alvos da consulta podem ser todas registradas no servidor de nomes do NWS ou apenas um sub-conjunto informado para o módulo.

Como o NWS fornece previsões, além das medidas sobre o estado dos recurso, o seletor de recursos poderia ser implementado sobre qualquer uma dessas duas informações. Fomos desencorajados de implementá-lo com base nas previsões por três motivos. Primeiro, por percebermos que as previsões para os recursos de processamento na maioria dos casos, utilizam a política *Last Value* que consiste em adotar o valor da última medição como próximo valor previsto. Segundo, por obtermos um desempenho insatisfatório nas previsões dos recursos de comunicação. E por último, para evitar o *overhead* introduzido pelos cálculos de previsão, uma vez que eles são feitos utilizando todos os métodos disponíveis no NWS.

A API de acesso ao seletor de recursos é mostrada nos códigos 3.3 e 3.5. Com a utilização da função `getBestServers` (*código 3.3*) é possível buscar por servidores que satisfaçam as restrições presentes em `resources` e `minimalValue`. O primeiro define quais agentes de recursos serão consultados (e.g. agente de monitoração do processamento, da comunicação, etc). O segundo determina os valores mínimos desejados que estes recursos devem atender. O número de respostas desejadas (ou seja, a quantidade de servidores) é especificada em `numberOfServers`. E finalmente, `criterion` define o critério de escolha dos servidores. Ajustado como `HIGH`, por exemplo, define que o recurso desejado é o que possui a maior valoração da propriedade do recurso em questão (e.g. para o recurso de processamento, em que a propriedade é carga, `HIGH` indica que o de maior disponibilidade de CPU é preferencial). Caso `minimalValue` não seja especificado é assumido que o valor mínimo é zero, então a decisão é feita unicamente pela informação presente em `criterion`.

Código 3.3 - *API do Seletor de Recursos para busca por servidores.*

```
1: getBestServers ( resources[ ],          //tipos de recursos
2:                  minimalValues[ ],      //valores mínimos desejados
3:                  numberOfServers,       //número de servidores desejados
4:                  criterion[ ] )         //critério de escolha
```

Para exemplificar a utilização desta função, consideramos uma aplicação cliente-servidor, em que o cliente faz requisições de processamento a vários servidores. Através dos contratos poderia ser especificada a necessidade de alocação de 5 servidores, com no mínimo 70% de CPU livre e com uma latência de comunicação máxima de 10 ms. Como os servidores de maior disponibilidade de CPU e de menor latência no enlace são preferenciais, utilizamos `HIGH` e `LOW`, respectivamente. Então a consulta que deverá ser elaborada pelo *Configurator* para implantanção do serviço fica como demonstrado pelo código 3.4.

Código 3.4 - *Exemplo de uso da API de busca por melhores servidores.*

```
servers[ ] = getBestServers ( [CPU, LATENCY], [70, 10], 5, [HIGH, LOW] );
```

Como resultado dessa consulta são retornados os 5 melhores servidores, com base nos requisitos informados, sendo que o primeiro da lista será o de maior disponibilidade de processamento, pois recurso de CPU é de maior prioridade, uma vez que foi o primeiro a ser informado.

Para nossas simulações, selecionamos um servidor de cada vez, com base na sua utilização de recursos de processamento e não há necessidade de informar uma quantidade mínima de disponibilidade de processamento. Apenas desejamos, a cada consulta, que seja retornado um servidor e que este possua a maior disponibilidade de CPU dentre os demais.

Através da função `getBestGroup` (*código 3.5*) da API é possível buscar por nomes de grupos (sem clientes conectados) filtrados por informações presentes em `resources` e `criterion`. O parâmetro `groupNames` define os grupos que serão pesquisados para determinar qual é o mais adequado para o cliente adentrar. Por último, `fromHost` define o *host* de origem para as consultas de propriedades referentes ao enlace de comunicação (e.g. latência entre o *host* definido em `fromHost` e os grupos de servidores definidos em `groupNames`).

Uma possível utilização da API de escolha de grupos é mostrada em 3.6. Nesta, a seleção do grupo de servidores preferencial é baseada na latência de comunicação (`LATENCY`),

Código 3.5 - *API do Seletor de Recursos para busca de grupo de servidores.*

```
1: getBestGroup ( resources[ ],      //tipos de recursos
2:                criterion[ ],      //critério de escolha
3:                groupNames[ ],     //conjunto de grupos a serem pesquisados
4:                fromHost )         //origem da medição (para recursos de rede)
```

que, quanto menor, melhor (`LOW`). Ela é calculada entre o *host* informado em `hostName` e os grupos de servidores (variável `groupNames`). A tabela 3.1 descreve os parâmetros da API, bem como seus possíveis valores de parâmetros.

Código 3.6 - *Exemplo de uso da API de busca por melhor grupo de servidores.*

```
servers[ ] = getBestGroup ( [LATENCY], [LOW], groupNames, hostName );
```

3.5 Conclusão do capítulo

Neste capítulo foi apresentada uma infra-estrutura identificada para o gerenciamento de aplicações com requisitos não-funcionais. Foi mostrado também que o *framework* CR-RIO contempla tal modelo. Agregamos a ele, então, uma ferramenta de monitoramento existente para informar o estado de utilização dos recursos e a partir dessa ferramenta implementamos um módulo de seleção de recursos que, com base nos parâmetros informados, retorna os recursos mais viáveis a serem alocados ou desalocados.

Para a seleção de recursos considerando mais de uma variável (e.g. latência e carga de processamento), pode ser necessária a especificação de uma função utilidade como utilizada em [Huang 2004]. Tais funções permitem estipular, por exemplo, pesos que definem a importância de determinadas propriedades de cada recurso requerido, permitindo a escolha de um conjunto específico dentre os disponíveis. Essa informação poderia ser inserida nos contratos e passada para o *Configurator* quando uma adaptação tornar-se necessária. Com esse conhecimento em mãos, ele poderia consultar o seletor de recursos buscando um conjunto que melhor satisfizesse as restrições, priorizando os recursos que apresentam melhor caracterização das propriedades de maior peso.

No próximo capítulo serão apresentados alguns casos de estudo que permitirão avaliar a utilização do *framemork* e discutiremos sobre as formas de implementação de aplicações com requisitos não-funcionais, e o ganho de desempenho conseguido com a garantia destes requisitos.

Tabela 3.1: *Descrição dos argumentos para uso da API.*

Parâmetro	**Descrição**
resources	Informa quais as propriedades são interessantes para a busca dos servidores. Seus possíveis valores são: • CPU, para exigência de porcentagem de CPU disponível; • MEM, para quantidade de memória livre, expressa em Mbytes; • HD, para capacidade de armazenamento, expresso em Mbytes; • CONNECTION, para tempo, em ms, de estabelecimento de uma conexão TCP; • BANDWIDTH, para a taxa de transmissão de dados, expresso em Mbps; • LATENCY, para tempo de envio de uma mensagem TCP ao destino, expresso em ms;
minimalValues	É a quantidade mínima desejada do recurso para alocação de determinado nó. Caso seja necessária a informação sobre mais do que um recurso, a lista de valores mínimos deve estar na mesma ordem que a lista de recursos.
numberOfServers	Expressa o número de servidores desejados que atendam aos requisitos.
criterion	É a informação utilizada para escolher dentre os recursos que satisfazem as exigências mínimas. Seus valores são: • LOW, encontra o recurso que possui sua caracterização (seu valor) mais baixo. • HIGH, encontra o recurso que possui sua caracterização mais alta.
groupNames	Informa quais grupos devem ter suas propriedades verificadas. Tal informação é passada por um vetor de nomes de grupos para o seletor de recursos, que verifica as propriedades referentes a estes grupos.
fromHost	Indica a partir de qual *host* as propriedades de comunicação devem ser calculadas. Essa informação apenas é utilizada para descoberta de grupos com base em parâmetros de rede.

Capítulo 4

Exemplos

Os exemplos aqui descritos buscam validar a proposta apresentada, através da discussão de aspectos de implementação de casos de uso reais e ilustrar os benefícios proporcionados por ela. Tais benefícios serão apresentados em forma de interpretações de gráficos comparativos e de discussões de características (como manutenibilidade, confiabilidade, reusabilidade e outras) que aparecem com a aplicação dos conceitos presentes na proposta. Essa avaliação também permite demonstrar a flexibilidade de ajuste da adaptação.

Genericamente, um sistema de software pode ser formado por componentes que oferecem algum tipo de serviço (servidores) e componentes que utilizam serviços oferecidos (clientes). Então, estes sistemas podem ser descritos como aplicações cliente-servidor, com clientes invocando métodos dos servidores para a realização de alguma tarefa de preferência com garantias de qualidade. Interligando esses componentes aparecem os conectores, como por exemplo, um conector de grupo que pode ser responsável por interceptar as requisições feitas pelos clientes e de alguma forma repassá-las para os servidores. Estes conectores em geral são distribuídos, apresentando uma parte no lado do cliente e outra no lado do servidor, como pode ser visualizado na figura 4.1. Porém, nas próximas arquiteturas mostradas no decorrer desse trabalho, ilustraremos somente um conector no meio do cliente e do servidor para simplificar, como na figura 4.2, mas que simboliza a existência das duas partes.

Nossa avaliação está baseada em simulações de requisições intensas a um grupo de servidores, que disponibilizam serviços comuns a clientes que requerem certa qualidade de serviço, descrita através de um contrato de QoS. Definimos serviço como qualquer recurso que possa ser acessado remotamente e descrito através de uma interface, e servidor como a máquina responsável por hospedar o serviço e prover sua interface. Na próxima seção

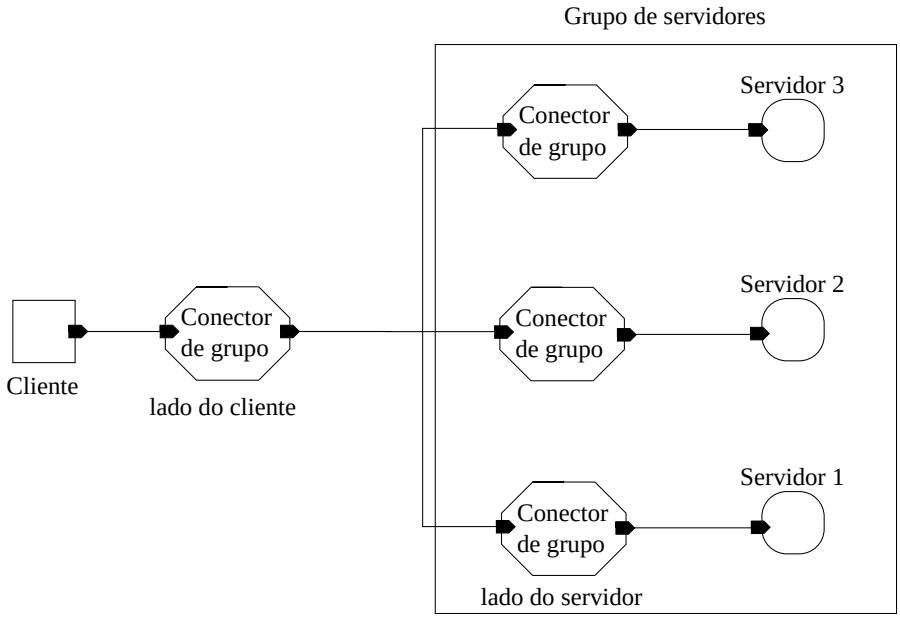

Figura 4.1: *Arquitetura de uma aplicação de configuração com 3 servidores ilustrando os conectores distribuídos, parte no lado do cliente e parte no lado do servidor.*

serão apresentados experimentos de aplicações *Bag of Tasks* em paralelo e avaliados seus desempenhos em situações como: utilizando ou não o módulo de seleção de recursos, e com ou sem o uso do *framework* para a gerência dos contratos. Além disso será feito um comparativo entre as duas últimas formas de desenvolvimento e implantação dessas aplicações. Na seção 4.2 serão mostrados experimentos envolvendo aplicações com balanceamento de carga e faremos uma comparação com e sem a presença do *framework*. Na seção 4.3.1 e na 4.3.2, serão mostradas características das aplicações tolerantes a falhas e *workflow*, respectivamente, e um possível mapeamento dos contratos para descrição dos seus requisitos não-funcionais.

4.1 Exemplo I - Aplicações *Bag of Tasks* em paralelo

Nesta seção apresentamos experimentações envolvendo aplicações *Bag of Tasks* (BoT). As aplicações BoT são compostas por tarefas que são independentes umas das outras. Apesar da sua simplicidade, tais aplicações são usadas em vários cenários como mineração de dados (*data mining*), pesquisas intensivas (como de quebra de chaves), aplicações *parameter sweeps*[1], manipulação de imagens, cálculos fractais entre outros [Cirne et al. 2003].

[1]Essas aplicações geralmente envolvem a execução de um único programa (ou um pequeno conjunto de programas) muitas vezes, variando a execução pela troca de algum argumento na linha de comando, arquivos de entrada ou ambos. A execução individual é freqüentemente independente, e pouca ou nenhuma comunicação ocorre, a não ser arquivos de saída de uma execução serem usados como entrada

Devido a sua independência entre tarefas, as aplicações BoT podem ser executadas em ambientes computacionais amplamente distribuídos. Os experimentos desta subseção são feitos com uma aplicação simples deste tipo, na qual o cliente faz sucessivas requisições a um serviço de cálculo de Pi disponível nos servidores, que as processam em paralelo.

Nosso primeiro conjunto de experimentos foi feito com um número fixo de servidores provendo um determinado serviço (seção 4.1.1). Convencionamos referenciar este conjunto como aplicações de configuração estática, por não ocorrem modificações em sua configuração inicial (figura 4.2). Nestes experimentos, os clientes do serviço provido não podiam exigir qualidade do mesmo. As alterações na utilização dos recursos ocorridas no ambiente de execução eram repassadas para os clientes em forma de degradação de desempenho do serviço acessado. Primeiramente, foram efetuados experimentos sem a utilização de ferramentas de monitoramento para a escolha dos servidores, que foram alocados aleatoriamente para a execução (seção 4.1.1.1). Numa segunda etapa, a escolha foi feita com base na utilização da capacidade de processamento dos servidores (seção 4.1.1.2).

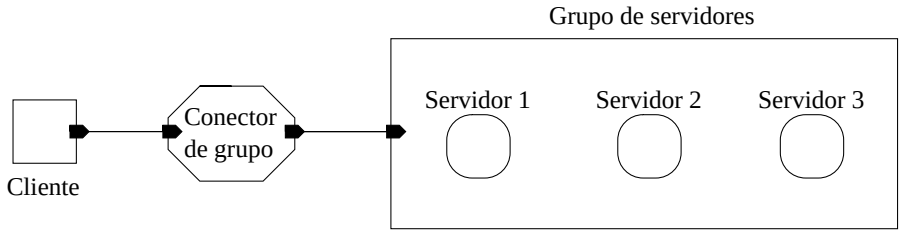

Figura 4.2: *Arquitetura de uma aplicação de configuração estática com 3 servidores.*

O segundo conjunto de experimentos foi elaborado com suporte a requisitos não-funcionais da aplicação e possibilidade de re-configuração dinâmica da infra-estrutura (seção 4.1.2). Essa re-configuração, nesse caso, trata de adicionar ou remover servidores do conjunto em execução. Esse conjunto é um agrupamento virtual de servidores, que podem estar fisicamente distribuídos, porém participam de um mesmo grupo de execução, processando as requisições que são feitas para o grupo. No momento em que uma máquina é adicionada a um determinado grupo de servidores, seus recursos são usados para prover o serviço disponibilizado ao cliente associado a este grupo. Em um primeiro experimento, a re-configuração foi provida de maneira *ad-hoc*, ou seja, de maneira específica para a aplicação, com seu código mesclado ao código da aplicação. Nos demais experimentos, os mecanismos responsáveis pela adaptação estão em um ambiente de suporte separado, neste caso no *framework* CR-RIO.

para outra.

4.1.1 Aplicação paralela com configuração estática da infra-estrutura

Para os experimentos de requisição de serviços a servidores com configuração estática da infra-estrutura, foram utilizados 3 servidores e o tempo de experimentação foi 10 minutos (600.000 milissegundos). É importante lembrar que esses hospedeiros (servidores) também estavam sendo utilizados por outras aplicações de diversos usuários. Foram feitos experimentos sem utilizar módulos para verificação de estado dos recursos (seção 4.1.1.1) e utilizando-os (seção 4.1.1.2).

4.1.1.1 Sem utilização de mecanismos de monitoração e seleção de recursos

Ao executarmos a aplicação sem nos preocuparmos em que estado de utilização de recursos estão os servidores hospedeiros do serviço, arriscamos deixar que máquinas abarrotadas de requisições tentem executar e responder aos pedidos de certos clientes, enquanto outros servidores permanecem ociosos. Através da figura 4.3 é perceptível uma grande variação no tempo de resposta percebido pelo cliente quando escolhemos o conjunto de servidores de forma aleatória para o provimento do serviço. Através da figura 4.4 (que representa o mesmo experimento da 4.3, mudando apenas a forma de exibição) percebemos que a maior concentração de tempos de resposta fica por volta de 100 ms, nos intervalos de tempo entre 0 e 2 minutos (120.000 ms) e entre 8 (480.000 ms) e 10 minutos (600.000 ms), quando a carga destinada a esse conjunto de servidores está baixa. Decorridos 120.000 ms de execução, submetemos os servidores a uma carga maior de processamento, o que acarreta um aumento abrupto do tempo de resposta. O que leva a maior concentração de tempo de resposta ficar por volta de 200 ms, ou seja, quando a carga do sistema está alta o cliente percebe uma degradação na qualidade do serviço prestado.

As sobrecargas associadas aos servidores foram geradas com a utilização de outro processo cliente realizando requisições intensas ao serviço provido pelo conjunto de servidores, aumentando o consumo de processamento destes. Esse método foi utilizado para todos experimentos em que era desejado aumentar a concorrência por recursos de CPU nos servidores.

4.1.1.2 Utilizando mecanismos de monitoração e seleção de recursos

Outro conjunto de experimentos ainda com a configuração estática, porém agora utilizando mecanismos de monitoração e seleção de recursos, se divide em dois. Primeiro

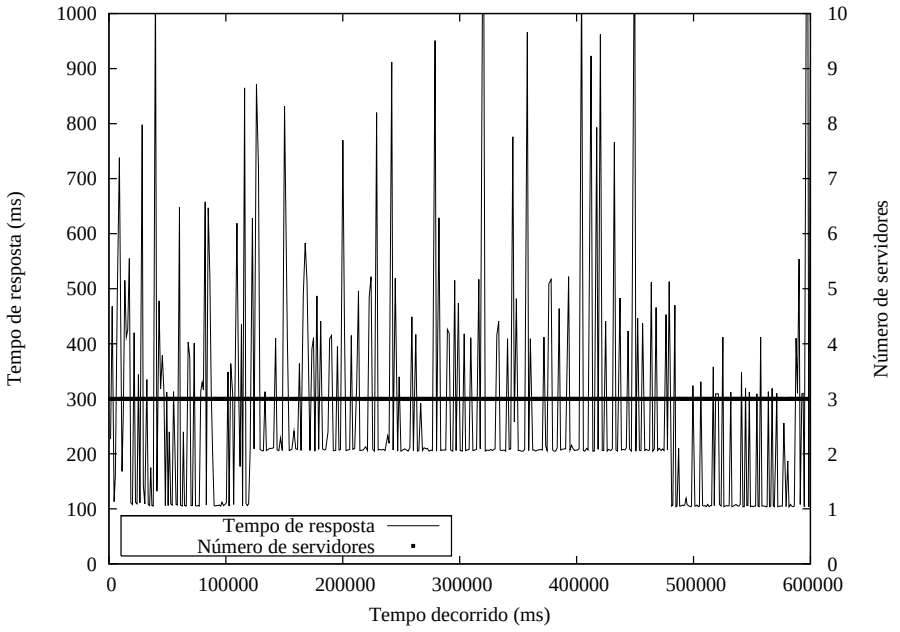

Figura 4.3: *Grande variação do tempo de resposta percebido pelo cliente em aplicação paralela com configuração estática da infra-estrutura e imposição de sobrecarga de processamento nos servidores. Essa variação pode ser explicada pela ausência de ferramentas de monitoração para a escolha dos servidores.*

simulamos sobrecarga nos servidores e depois excesso de fluxo no enlace de comunicação.

A) Gerando sobrecarga de processamento

Ao utilizarmos um módulo responsável pela seleção de recursos baseada em monitoração para escolher em quais servidores será instalado e funcionará o mesmo serviço, conseguimos uma redução da variação dos tempos de resposta percebidos pelo cliente para a mesma configuração anterior, como pode ser observado pelo gráfico 4.5. Isso ocorre devido à seleção de máquinas menos carregadas para o provimento do serviço. Entretanto o cliente ainda continua percebendo um aumento no tempo de resposta quando os servidores recebem a carga extra.

B) Gerando atraso no enlace de comunicação

Em um segundo experimento, ainda com a configuração estática da infra-estrutura, foi mantida a carga de processamento ao conjunto de servidores, porém foi inserido um atraso na comunicação de 200 ms entre o cliente e os servidores após 5 minutos (300.000 ms) do início da execução, como forma de simular um aumento repentino no tráfego da rede. Esse atraso foi imposto com a utilização do utilitário de controle de tráfego (*Traffic Controler*

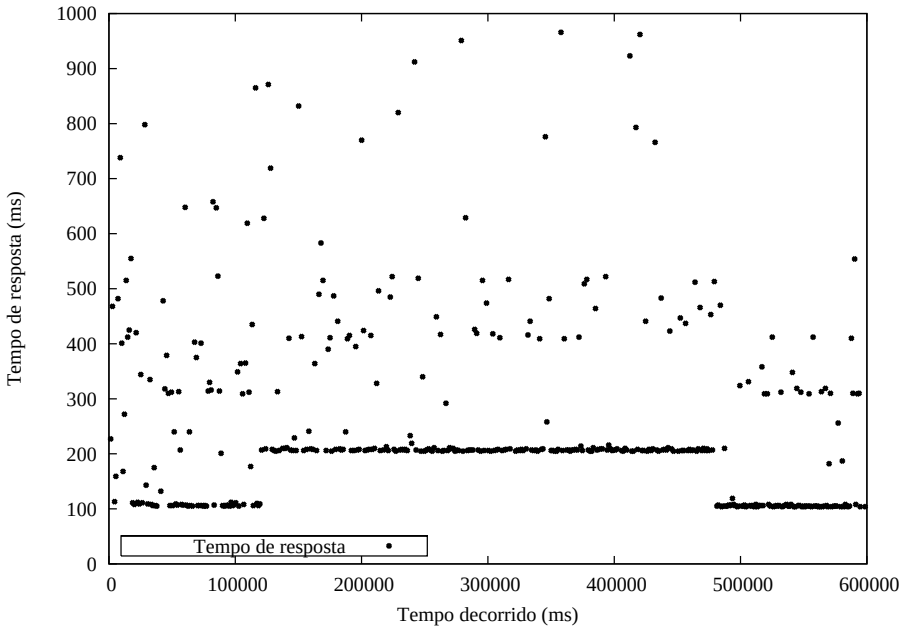

Figura 4.4: *Concentração da maior parte dos tempos de resposta percebidos pelo cliente em aplicação paralela com configuração estática da infra-estrutura e imposição de sobrecarga de processamento nos servidores.*

- TC) presente no Unix. Esse método foi utilizado em todos os experimentos nos quais era desejado impor um atraso no enlace de comunicação. O resultado, como esperado, mostra o aumento do tempo de resposta percebido pelo cliente, como pode ser visto na figura 4.6, a partir da segunda metade do tempo de execução.

4.1.2 Aplicação paralela com adaptação da infra-estrutura

Ao nos preocuparmos em manter um certo nível de qualidade para o serviço prestado pelos servidores, primeiramente surge a idéia de embutir um código extra na aplicação para verificação da qualidade oferecida (seção 4.1.2.1). Porém, dessa forma o reuso de software é dificultado uma vez que esses códigos costumam ser específicos para a aplicação em questão. Então separamos essas preocupações relativas a qualidade de serviço da funcionalidade da aplicação, deixando as verificações em uma estrutura separada (*framework*), como melhor detalhado na seção 4.1.2.2.

4.1.2.1 Sem utilização de mecanismos externos à aplicação

Ao inserirmos no código da aplicação as verificações de qualidade, devemos informá-la qual o nível de qualidade desejado. Isso pode ser feito através de linhas de código ou passado por parâmetro para a aplicação. Porém, ambas possibilidades não permitem que estes

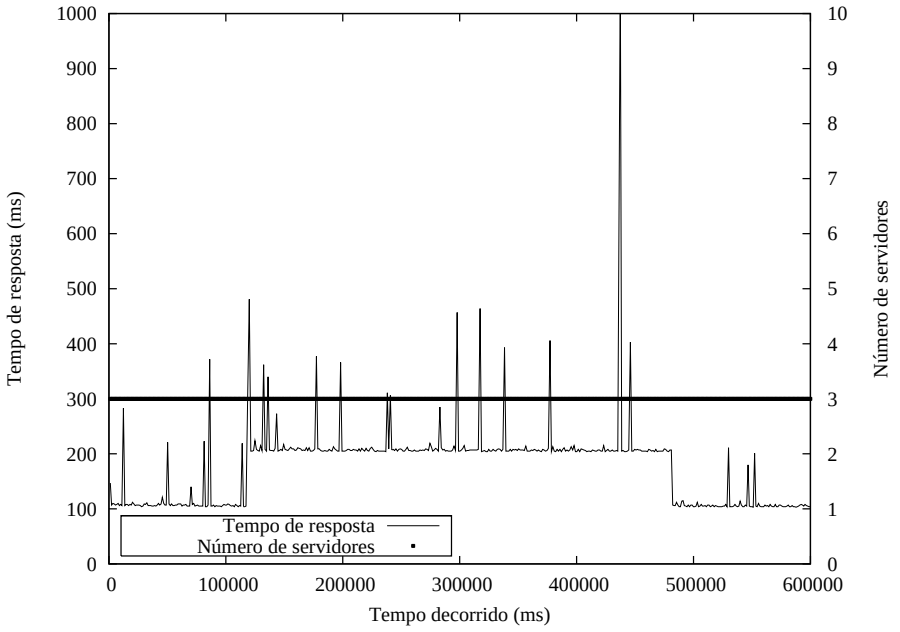

Figura 4.5: *Tempo de resposta percebido pelo cliente em aplicação paralela com configuração estática da infra-estrutura e imposição de sobrecarga de processamento nos servidores. Os servidores foram escolhidos com base em suas utilizações de processamento, por isso as variações são menos constantes.*

valores sejam alterados dinamicamente. A figura 4.7 ilustra de forma simplificada o diagrama de classes da aplicação implementada dessa forma. O `RpcDispatcherServer` provê o serviço que será requisitado, além de permitir que alguém lhe requisite entrar ou sair de um grupo específico. O cliente (`RpcDispatcherClient`) acessa o serviço disponível no servidor através do método `requestService`. Estipulados os limiares considerados satisfatórios pelo cliente (`minResponseTime` e `maxResponseTime`), o tempo de resposta passa a ser constantemente monitorado pela aplicação cliente (através da função `analyze()`) para garantir o atendimento a estes limites e caso não seja possível, a própria aplicação se encarrega de requisitar que servidores adicionais passem a compor o conjunto de servidores em execução (através da instância da classe `SenderThread`). Nessa versão, os sistemas de monitoração e de seleção de recursos não foram agregados à aplicação, portanto os servidores adicionais são escolhidos aleatoriamente.

A maior desvantagem dessa implementação é a ausência da separação de interesses (*separation of concerns*), pois a funcionalidade básica da aplicação se mistura com seus aspectos não-funcionais.

O resultado do experimento feito com essa forma de implementação é mostrado na figura 4.8. É perceptível a permanência das oscilações nos tempos de resposta obtidos pelo cliente, uma vez que a alocação dos servidores não considera a carga de processamento dos

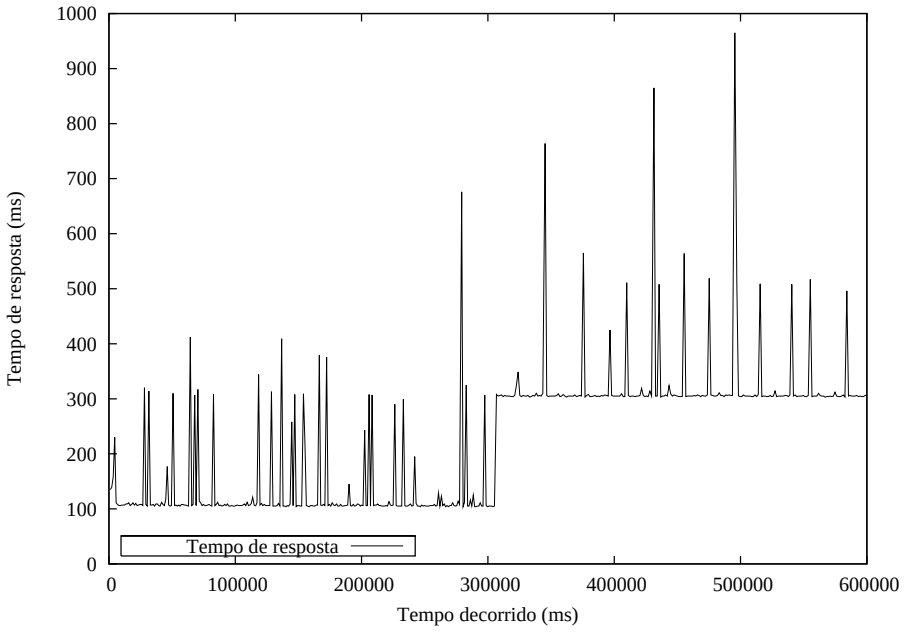

Figura 4.6: *Tempo de resposta percebido pelo cliente em aplicação paralela, com configuração estática da infra-estrutura e imposição de atraso na comunicação.*

mesmos. Podemos observar também o aumento dinâmico do número de servidores respondendo às requisições, inicialmente 3 e atingindo o máximo de 6 servidores no momento em que recebem maior carga. Quando a exigência aos servidores diminui, novamente o número de servidores é modificado, voltando a 3, como inicialmente.

Através da figura 4.9, que representa o mesmo experimento, percebemos que, na maior parte do tempo, há grande concentração de tempos de resposta nas proximidades de 100 ms, mesmo nos momentos de alta exigência aos servidores. Isso ocorre porque quando a carga aumenta, o número de servidores também é incrementado, pois a aplicação cliente percebe um aumento no tempo de resposta do serviço e requisita a inserção de mais servidores para a execução. Como o serviço é paralelizável, cada servidor pode executar parte dele e retornar a resposta do seu processamento.

Realizamos também experimentos em que a aplicação, além de conter o código responsável por verificar o atendimento aos requisitos de qualidade do serviço, possuía acesso aos mecanismos de monitoração e seleção de recurso. Dessa forma a aplicação era capaz de escolher o melhor servidor (de acordo com alguma política implementada na própria aplicação) para fazer parte do grupo em execução, ou para deixá-lo. O resultado dos experimentos com essa implementação foram semelhantes, em termos de desempenho, aos que serão apresentados no gráfico 4.12. Entretanto, a verificação da qualidade do serviço, as políticas de adaptação e o acesso às medidas coletadas sobre os recursos de suporte

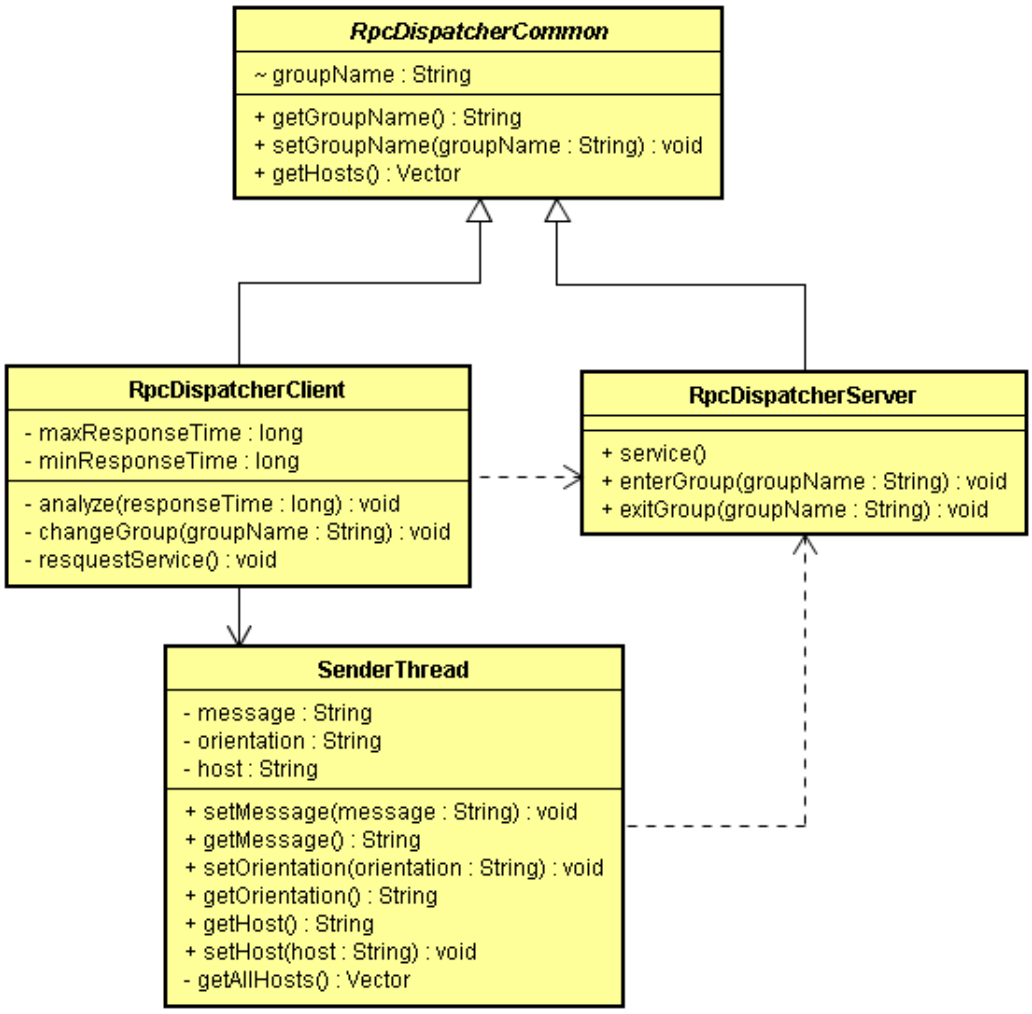

Figura 4.7: *Diagrama de classes simplificado da aplicação.*

ficaram altamente específicos para a aplicação, infringindo o conceito da reutilização tanto buscado, o que acarreta em uma manutenibilidade do sistema mais custosa.

4.1.2.2 Utilizando mecanismos de monitoração integrando ao *framework* CR-RIO

Para os experimentos de requisição de serviços a servidores com configuração dinâmica da infra-estrutura e utilizando mecanismos de monitoração para compor o *framework* CR-RIO, novamente realizamos dois experimentos: primeiramente sobrecarregamos os processadores (subseção A) e depois aumentamos a latência da comunicação dos servidores com o cliente (subseção B). O CR-RIO foi responsável por obter as informações sobre a qualidade desejada pelo cliente através dos contratos, gerenciar a monitoração dos recursos utilizados para o provimento da aplicação e sua qualidade de serviço oferecida. Caso essas medidas não estejam de acordo com os valores especificados nos contratos, ele dispara

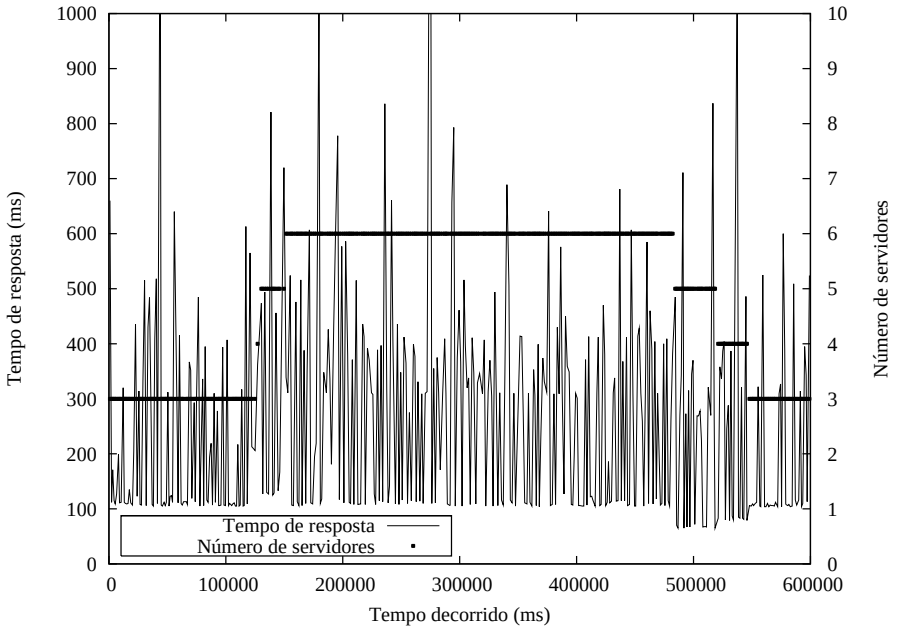

Figura 4.8: *Tempo de resposta percebido pelo cliente em aplicação paralela, com configuração estática da infra-estrutura e imposição de sobrecarga de processamento nos servidores. O número de servidores aumenta para tentar manter os tempos de resposta dentro dos limiares desejados. A constante variação pode ser explicada pela ausência de ferramentas de monitoração para a escolha dos servidores.*

medidas adaptativas visando deixar a execução da aplicação em obediência aos anseios do usuário.

A) Gerando sobrecarga de processamento

Para os experimentos de requisição de serviços a servidores com configuração dinâmica da infra-estrutura e utilizando mecanismos de monitoração para compor o *framework* CR-RIO, inicialmente foram utilizados 3 servidores, alocados de acordo com sua utilização de capacidade de processamento e, conforme necessário, foram adicionadas mais réplicas (novamente levando em consideração a carga de CPU) de forma a manter a qualidade de serviço especificada no contrato mostrado nos códigos 4.2 e 4.3, explicados mais a frente.

A figura 4.10 mostra a arquitetura dessa aplicação, onde servidores estão associados a grupos que podem estar geograficamente distribuídos. O cliente requisita um serviço ao grupo de servidores, porém sem saber quantos, nem quais servidores irão processar sua requisição. A parte cliente do conector se encarrega de escolher qual grupo fará o processamento e a parte do conector no lado dos servidores implementa a política de distribuição das requisições entre os servidores dentro do grupo. Do lado do servidor, cada um processa apenas 1/n do serviço (onde `n` é o número de servidores no grupo) e responde

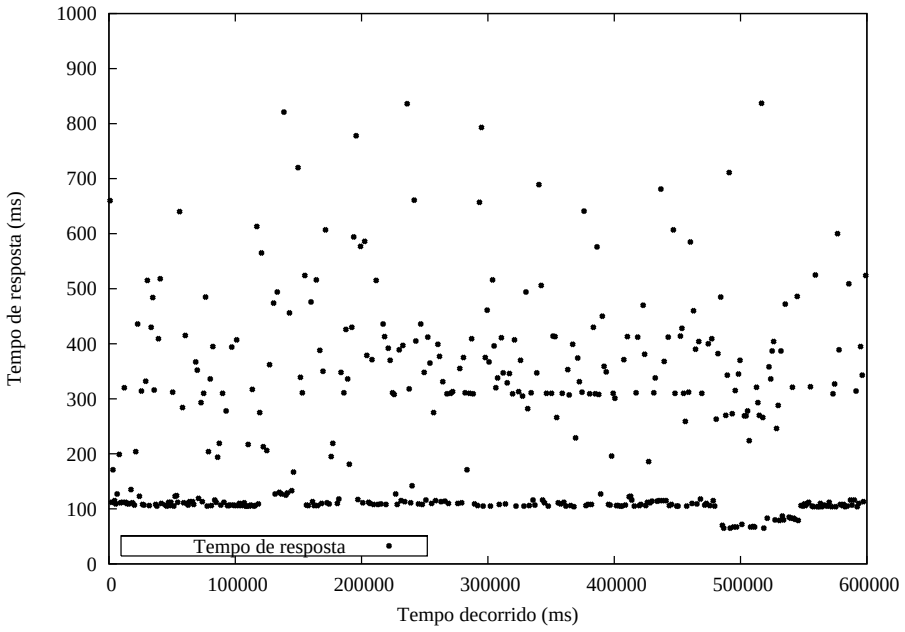

Figura 4.9: *Concentração da maior parte dos tempos de resposta percebidos pelo cliente em aplicação paralela, com configuração estática da infra-estrutura e imposição de sobrecarga de processamento nos servidores.*

o resultado do seu processamento, que é agregado no conector e repassado para o cliente. A seguir serão apresentados uma descrição de arquitetura, que descreve a configuração inicial de uma aplicação de cálculo de pi, processada em paralelo; as categorias e o contrato de QoS com os requisitos não-funcionais para tal aplicação. Também serão discutidos o funcionamento das partes do *framework* que tentam manter a qualidade do serviço dentro do que foi delineado pelos contratos, e a forma como foram implementados.

- Categorias de QoS

A categoria `Processing` *(código 4.1, linhas 01-04)* define propriedades relativas ao processamento de cada servidor. A propriedade `utilization` representa a carga de processamento de um determinado *host*, seu tipo é numérico (`numeric`) e sua medida é em porcentagem de uso (`%`). O termo `decreasing` informa que quanto maior o valor desta propriedade, pior é a qualidade do serviço, ou seja, quanto mais utilizado estiver o processador de uma máquina, pior será a qualidade oferecida por ele. Em momentos de seleção de recursos, tal classificação se torna um importante critério de desempate entre recursos com caracterização semelhante. Outra propriedade definida é `clockfrequency`, que se refere à freqüência de operação do processador da máquina, seu tipo é numérico e sua medida é em `MHz`. Diferentemente de `utilization`, esta propriedade é `increasing`, ou seja, quanto maior o valor desta propriedade, melhor é a qualidade do serviço prestado pelo recurso. Ambas categorias são de validação, identificado pelo termo `in`, ou seja, ela

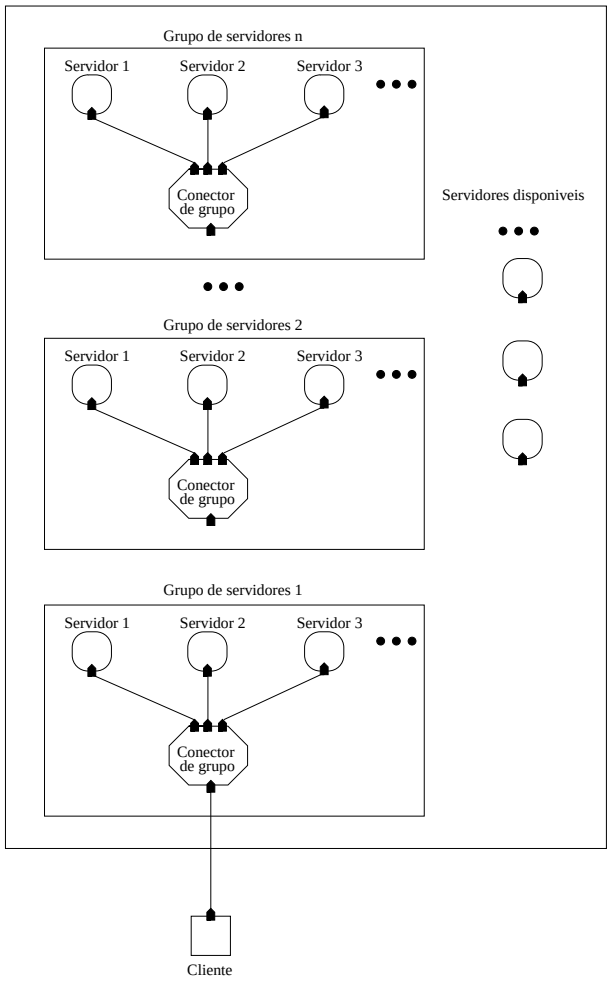

Figura 4.10: *Arquitetura da aplicação Bag-of-Tasks.*

será utilizada para definir uma restrição que deverá ser válida para satisfazer o perfil ao qual está relacionada.

A categoria `ServerGroup` *(código 4.1, linhas 05-08)* define as propriedades de um grupo de servidores. A carga de processamento de um determinado grupo é representada por `load`, uma propriedade de verificação `(in)`, e a política de distribuição de carga utilizada pelo grupo de servidores é orientada pela propriedade `distribPolicy`. As políticas são enumeradas na definição da propriedade, a primeira (`bestCpu`) indica que as requisições provindas dos clientes serão enviadas para o servidor que possuir maior disponibilidade de recurso de processamento; a segunda, que serão distribuídas via `Round Robin`; `random`, como o próprio nome diz, de forma randômica; e a última que será por *multicast*, enviando todas as requisições para todos os servidores do grupo.

`Client` *(código 4.1, linhas 09-12)* representa o cliente da aplicação e possui as propriedades: `responseTime`, o tempo de resposta percebido pelo cliente e `linkPolicy`, as políticas que orientarão o mecanismo de seleção de recursos na busca a um novo enlace.

Código 4.1 - *Descrição das categorias de QoS (QoSCategory) que serão utilizadas nos contratos adiante.*

```
1:  QoSCategory Processing {
2:      utilization: decreasing numeric % in;
3:      clockfrequency: increasing numeric MHz in;
4:  }
5:  QoSCategory ServerGroup {
6:      load: decreasing numeric % in;
7:      distribPolicy: enum (bestCpu, roundRobin, random, multicast) out;
8:  }
9:  QoSCategory Client {
10:     responseTime: decreasing numeric ms in;
11:     linkPolicy: enum (lowLatency, highLatency, lowBand, highBand, optim) out;
12: }
13: QoSCategory Replication {
14:     numberOfReplicas: increasing numeric in;
15:     replicaMaint: enum (add, release, maintain) out;
16:     allocationPolicy: enum (bestMem, highLoad, lowLoad, optim) out;
17: }
18: QoSCategory Transport {
19:     latency: decreasing numeric ms in;
20:     bandwidth: increasing numeric Mbps in;
21: }
```

As opções de políticas são enumeradas `(enum)`: `lowLatency`, que orienta a escolha de um enlace com a menor latência disponível; `highLatency`, que define a busca pelo enlace com maior latência; `lowBand`, que guia a procura por enlace de menor banda passante disponível (as duas últimas são úteis para desalocações de componentes); `highBand`, que orienta a procura pelo enlace de maior banda passante disponível e `optim`, uma política "ótima" para a definição do melhor enlace. Um exemplo de uma política ótima seria uma que orientasse a busca por um novo enlace levando em consideração as propriedades de latência e de processamento. `linkPolicy` é uma propriedade de orientação `(out)`, portanto não é necessária para a validação do perfil em que está presente, caso o serviço que possui esse perfil seja ativado, tal propriedade servirá como guia no nível de implementação para escolha de recursos.

A categoria referente ao mecanismo de replicação de servidores, `Replication`, pos-

sui as propriedades `numberOfReplicas`, `replicaMaint` e `allocationPolicy` *(código 4.1, linhas 13-17)*. `numberOfReplicas` refere-se ao número de réplicas de servidores em um determinado grupo, `replicaMaint` enumera ações que podem ser efetuadas na manutenção da réplica, podendo ser adicionada uma nova `(add)`, removida `(release)` ou mantida `(maintain)`. A propriedade `allocationPollicy` define as políticas que orientam a alocação de réplicas por um grupo. A opção `bestMem` define que será alocada a réplica que possuir mais recursos de memória disponíveis. Com a política `highLoad` é definido que será alocado o servidor que contiver maior disponibilidade nos recursos de processamento e com `lowLoad`, o servidor selecionado será o de menor disponibilidade de capacidade de processamento. Outras políticas, como por exemplo, `bestHd` ou `nearProcessor` poderiam ser incluídas, visando buscar por máquinas com maior disponibilidade de CPU e máquinas mais próximas dos clientes, respectivamente.

`Transport` *(código 4.1, linhas 18-21)* refere-se às propriedades do enlace de comunicação. Conta com as propriedades `latency` e `bandwidth`, expressas respectivamente através das unidades ms e Mbps.

Essas categorias são específicas por recurso, porém podem ser reaproveitadas por aplicações que também utilizam esses mesmos tipos de recursos. Portanto são altamente reutilizáveis e podem compor um repositório de forma a viabilizar seu reúso.

- Contrato de QoS

O contrato apresentado nos códigos 4.2 e 4.3 descreve serviços associados com restrições de qualidade definidas para o tempo de resposta observado pelo cliente. Tais restrições são uma condição, dentre outras, para o estabelecimento de cada serviço. A seguir descrevemos o funcionamento do contrato, considerando os diferentes serviços:

- **sMaintain:** O serviço `sMaintain` *(código 4.2, linhas 02-04)* liga um cliente a um grupo de servidores, caso as restrição sobre o tempo de resposta e latência de comunicação entre o cliente e o grupo de servidores definidas no perfil `pMaintainServer` *(código 4.3, linhas 01-05)* sejam satisfeitas. Tais condições são especificadas através dos valores da propriedade `responseTime` *(código 4.3, linhas 02-03)*. Uma outra restrição do serviço `sMaintain` *(código 4.3, linha 4)* é apenas uma orientação para manter a conexão entre o cliente e o atual grupo de servidores. E o perfil `pCom` *(código 4.3, linhas 24-26)* orienta o mecanismo de distribuição de carga a utilizar o *multicast* para difundir as requisições do cliente. Esse serviço é, na verdade, um estado estacionário onde nenhuma medida adaptativa é necessária.

Código 4.2 - *Um contrato de QoS especificando requisitos desejados pelo cliente para aplicação em paralelo.*

```
1: contract {
2:   service {
3:     link client to serverGroup with pMaintainServer, pCom;
4:   } sMaintain;
5:   service {
6:     instantiate server with pAddServer;
7:     link server to serverGroup with pCom;
8:   } sAddServer;
9:   service {
10:    remove server at serverGroup with pRemoveServer;
11:  } sRemoveServer;
12:  service {
13:    link client to serverGroup with pMoveClient;
14:  } sMoveClient;
15:  service {
16:    instantiate server with pNoMoreServersToAdd;
17:  } sBadQuality;
18:  service {
19:    instantiate server with pNoMoreServersToRemove;
20:  } sExcellentQuality;
21:  negotiation {
22:    not sMaintain → (sAddServer || sRemoveServer || sMoveClient || sBadQuality ||
                        sExcellentQuality);
23:    sAddServer → (sMaintain || sRemoveServer || sMoveClient || sBadQuality);
24:    sRemoveServer → (sMaintain || sAddServer || sMoveClient || sExcellentQuality);
25:    sMoveClient → (sMaintain || sAddServer || sRemoveServer || sBadQuality ||
                      sExcellentQuality);
26:    sBadQuality → (sMaintain || sRemoveServer || sMoveClient || sAddServer);
27:    sExcellentQuality → (sMaintain || sAddServer || sMoveClient);
28:  }
29: } Csr;
```

Código 4.3 - *Perfis do contrato apresentado em código 4.2 para aplicação em paralelo.*

```
1: profile {
2:     Client.responseTime <= 150;
3:     Client.responseTime >= 80;
4:     Replication.replicaMaint := maintain;
5: } pMaintainServer;
6: profile {
7:     Client.responseTime > 150;
8:     Transport.latency <= 15;
9:     Replication.numberOfReplicas < 30;
10:    Replication.replicaMaint := add;
11:    Replication.allocationPolicy := lowLoad;
12: } pAddServer;
13: profile {
14:    Client.responseTime < 80;
15:    Replication.numberOfReplicas > 1;
16:    Replication.replicaMaint := release;
17:    Replication.allocationPolicy := highLoad;
18: } pRemoveServer;
19: profile {
20:    Transport.latency > 15;
21:    Client.responseTime > 150;
22:    Client.linkPolicy := lowLatency;
23: } pMoveClient;
24: profile {
25:    ServerGroup.distribPolicy := multicast;
26: } pCom;
27: profile {
28:    Client.responseTime < 80;
29:    Replication.numberOfReplicas = 1;
30:    Replication.replicaMaint := maint;
31: } pNoMoreServersToRemove;
32: profile {
33:    Client.responseTime > 150;
34:    Replication.numberOfReplicas = 30;
35:    Replication.replicaMaint := maint;
36: } pNoMoreServersToAdd;
```

- **sAddServer:** Em caso de violação do perfil `pMaintServer` com o serviço `sMaintain` em funcionamento, uma tentativa de estabelecer o próximo serviço será efetuada, de acordo com a ordem estabelecida na cláusula de negociação de serviços do contrato *(código 4.2, linhas 21-28)*. Neste caso, o próximo serviço é `sAddServer` *(código 4.2, linhas 05-08)*, que adiciona mais réplicas ao grupo de servidores em execução. Este serviço apenas entrará em vigor se suas restrições forem satisfeitas *(código 4.3, linhas 06-12)*, ou seja, o tempo de resposta percebido pelo cliente deve estar acima de 150 ms, a latência do enlace deve estar abaixo de 15 ms e o número de réplicas em execução deve ser menor que o total disponível, isto é, inferior a 30. Caso sejam atendidas suas restrições, este serviço tentará adicionar réplicas *(código 4.3, linha 10)* com a menor utilização de capacidade de processamento possível, seguindo a orientação da política de alocação *(código 4.3, linha 11)*. É interessante notar que as réplicas acrescentadas também receberão requisições através de *multicast (código 4.3, linhas 24-26)*, pois o perfil `pCom` também está associado a esse serviço. Uma vez ativo, esse serviço permanece adicionando servidores até que outro serviço se torne válido e o substitua, uma vez que este não é um serviço preferencial, pois não possui a palavra reservada `not`. Caso o tempo de resposta fique abaixo de (ou igual a) 150 ms ou o número de réplicas atinja o limite máximo ou a latência ultrapasse o limite de 15 ms, surgirá a necessidade de instauração de outro serviço, de acordo com a regra de negociação. A exigência da latência inferior ou igual a 15 ms para ativação do serviço `sAddServer`, impede que servidores sejam adicionados desnecessariamente em casos em que o causador do atraso no tempo de resposta é a má qualidade no enlace de comunicação e não sobrecarga dos servidores.

- **sRemoveServer:** Em uma situação de não atendimento às restrições do serviço `sAddServer`, o próximo serviço, `sRemoveServer` *(código 4.2, linhas 09-11)*, tentará ser negociado. O perfil associado a esse serviço é `pRemoveServer` *(código 4.3, linhas 13-18)*, que define que o tempo de resposta percebido pelo cliente deve ser inferior a 80 ms e que o número de réplicas não deve ser inferior a 1, como suas condições de entrada. As demais restrições *(código 4.3, linhas 16-17)* informam que a política de manutenção é a remoção de réplicas, e que a mais carregada deve ser a escolhida. Estas informações orientam o mecanismo de gerenciamento de réplicas na remoção de servidores do grupo, até que outro serviço possa ser estabelecido.

- **sMoveClient:** Outro serviço disponível é o `sMoveClient` *(código 4.2, linhas 12-14)*, responsável por desconectar o cliente do grupo atual e conectá-lo a outro, que é estabelecido se o tempo de resposta estiver acima do limite estipulado e a latência

no canal de comunicação com o atual grupo for maior que o máximo estabelecido, como descrito no perfil `pMoveClient` *(código 4.3, linhas 19-23)*. Através de suas restrições é possível verificar se o aumento no tempo de resposta no cliente ocorreu por insuficiência de qualidade na conexão entre o cliente e os servidores, pois a latência é verificada *(código 4.3, linha 20)*. Constatado que o problema é na comunicação, o cliente será movido para outro grupo com menor latência, conforme orientado pela política de ligação *(código 4.3, linha 22)*. Esse grupo de destino para o qual o cliente será migrado, é encontrado através de uma consulta ao módulo de seleção de recursos (ver seção 3.4.3), com os parâmetros desejados.

- **sBadQuality, sExcellentQuality:** Os demais serviços `sBadQuality` e `sExcellentQuality` se estabelecem, respectivamente, quando o tempo de resposta é muito baixo porém a quantidade de servidores já alcançou o limite mínimo, e quando o tempo de resposta é muito alto mas não há mais servidores para serem acrescentados. São serviços estacionários, que não envolvem migração de cliente para outros grupos nem adição ou remoção de servidores, servindo apenas para evitar que a aplicação atinja um estado "fora de serviço", podendo a aplicação ser provida, mesmo com baixa qualidade, como é o caso do `sBadQuality`, ao invés de se tornar indisponível. Quando esse serviço está corrente, ou seja não há mais servidores para serem adicionados, pode ocorre a transição para o serviço `sAddServer` em uma situação peculiar, em que uma(s) das réplicas falha(m). Com isso o contador do número de réplicas é decrementado e novas réplicas podem ser acrescentadas.

- Funcionamento

A aplicação passará, então, a tentar respeitar as especificações dos contratos, e para verificar as adaptações infra-estruturais, o conjunto de processadores foi novamente submetido à carga descrita na seção anterior. Entretanto, neste experimento, foram introduzidas a especificação da qualidade desejada feita através do contrato da aplicação, a monitoração e o mecanismo de seleção de recursos, provendo caracterização e a escolha dos recursos, além da infra-estrutura do *framework* provendo mecanismos (verificações de atendimento às necessidades e capacidade de disparar adaptações) para que os requisitos desejados pelo cliente sejam atendidos. Ao ser submetido a um aumento de carga súbito, o conjunto de servidores inicial, formado por 3 réplicas, não consegue atender os pedidos do cliente dentro dos limites temporais. O diagrama da figura 4.11 ilustra o comportamento dos elementos de suporte do CR-RIO nessa situação.

É identificado pelo `Contractor`, o não atendimento do requisito de tempo de res-

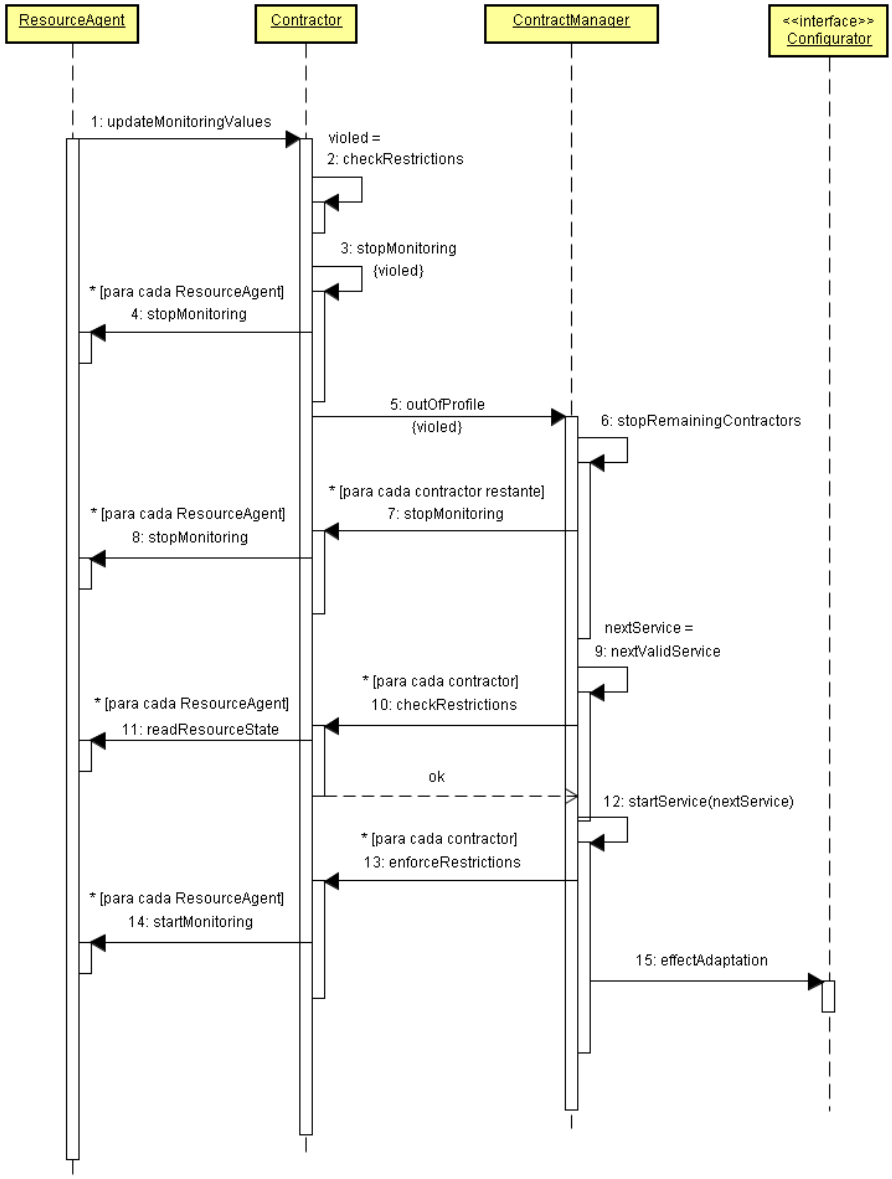

Figura 4.11: *Diagrama de interação do processo de violação de contrato no CR-RIO.*

posta, através dos valores adquiridos pelos `ResourceAgents`, que atualizam os valores das propriedades monitoradas (a uma periodicidade determinada no próprio `ResourceAgent`) sempre que uma mudança significativa é detectada. A definição de mudança significativa nos valores pode ser feita no `ResourceAgent`, através de uma constante de fator de tolerância (*Tuning*), com a qual é comparada a variação dos valores, e caso seja significativa em relação a *Tuning*, o novo valor é propagado ao `Contractor` *(figura 4.11, 1:updateMonitoringValues)*. Portanto é permitido que o recurso sofra múltiplas alterações de estado dentro da faixa admitida pelo valor de *Tuning* sem que os `Contractors` sejam informados.

A definição do valor de *tuning* pode ser feita na definição da categoria a qual está associada. Por exemplo, na categoria `client`, onde está presente a propriedade `responseTime`,

com a utilização de um *tuning* valorado em 5, os valores passam a ser atualizados no `resourseState` somente se o valor monitorado diferenciar em mais de 5 ms do valor anteriormente monitorado. Em nossos exemplos, simplificamos a definição das categorias ocultando a definição do valor de *tuning*, por desejarmos que todas as medidas do tempo de resposta fossem conhecidas pelo `Contractor`.

O `Contractor` é capaz de perceber que os valores monitorados estão fora das especificações almejadas *(figura 4.11, 2:checkRestrictions)*, uma vez que possui as restrições associadas ao serviço em execução, passadas previamente pelo gerente antes do estabelecimento do serviço. Foi empregada uma estratégia de contagem das ocorrências sucessivas de medidas (no nosso caso 5 ocorrências) que ultrapassavam as especificações, de forma a evitar trocas de serviço desnecessárias, assim, o serviço só deixa de vigorar quando seus requisitos não são obedecidos por cinco vezes consecutivas. Isso foi necessário porque a alta utilização das máquinas por diversas aplicações de usuários acarretava em alguns picos no tempo de resposta percebido pelo cliente, que não representavam sobrecarga efetiva no servidor, nem no enlace. Eram apenas colisões na concorrência de acesso aos servidores e que, em geral, não se repetiam por mais de 5 vezes, por isso não era necessário adaptar o sistema. Conseguimos perceber que quando o sistema estava realmente sobrecarregado, os altos tempos de resposta se prolongavam por mais de 5 vezes seguidas, e essa observação também é válida para os baixos tempos de reposta. Embora essa medida de filtro diminua as oscilações dispensáveis entre serviços, o desenvolvimento de uma estratégia mais robusta para evitar instabilidade na percepção de mudanças no tempo de resposta é necessário.

Comprovada a violação de alguma das restrições referente ao serviço corrente, o processo de monitoramento é interrompido nos agentes de recurso gerenciados pelo `Contractor` *(figura 4.11, 3:stopMonitoring e 4:stopMonitoring)* e a informação de que os perfis do serviço corrente `(sMaintain)` não estão sendo atendidos é passada para o gerente de contratos *(figura 4.11, 5:outOfProfile)*, que ao receber essa notificação, requisita a cada `Contractor` restante, caso exista algum, o interrompimento do processo de monitoração por ele gerenciado *(figura 4.11, 6:stopRemainingContractors e 7:stopMonitoring)*, já que o serviço corrente será encerrado. É importante notar que esse encerramento consiste apenas de um ajuste de uma variável (e não o descarregamento das classes referentes à monitoração), para que o sistema não permaneça identificando a deficiência na prestação do serviço. Para cada `Contractor` remanescente, o monitoramento efetuado pelos agentes a ele associados é interrompido *(figura 4.11, 8:stopMonitoring)*. Nesse exemplo, os `ResourceAgents` no cliente são gerenciados por um único `Contractor`, pois há apenas

um contrato por *host*. A partir daí, o Gerente tenta estabelecer o próximo serviço *(figura 4.11, 9:nextValidService)* de acordo com a seqüência definida na regra de negociação *(código 4.2, linhas 21-28)*.

Para saber se um serviço pode ser estabelecido, o `ContractManager` solicita a cada `Contractor` responsável pelo gerenciamento da monitoração das propriedades associadas ao serviço candidato, que verifique se os valores dessas propriedades satisfazem às restrições associadas a esse serviço *(figura 4.11, 10:checkRestrictions)*. Para efetuar isso, o `Contractor` requisita a cada `ResourceAgent` que efetue a leitura dos valores correntes das propriedades de recurso envolvidas nas restrições associadas ao serviço *(figura 4.11, 11:readResourceState)*. Recebidos esses valores, é verificado se satisfazem às restrições do serviço `sAddServer` (que é o próximo serviço segundo a regra de negociação), e caso possam ser atendidas, o `ContractManager` é notificado.

Para o estabelecimento do serviço *(figura 4.11, 12:startService(nextService))*, os `Contractors` são requisitados para iniciar o processo de monitoração das propriedades dos recursos a serem utilizados, verificando as restrições associadas ao serviço a ser estabelecido *(figura 4.11, 13:enforceRestrictions)* e para isso, cada `Contractor` solicita aos `ResourceAgents` por ele gerenciados, o início do monitoramento dessas propriedades *(figura 4.11, 14:startMonitoring)*. Finalmente, as configurações arquiteturais são enviadas ao configurador *(figura 4.11, 15:effectAdaptation)*, que se encarregará de estabelecer a configuração exigida para o provimento do serviço `sAddServer`.

Como `sMaint` é um serviço preferencial (`not`), o próximo serviço, isto é, `sAddServer` só será estabelecido caso as restrições do primeiro não possam ser atendidas. Na ausência do `not` antes do serviço, mesmo se este estiver vigente com suas restrições atendidas e acontecer de um outro com prioridade maior puder ser provido, haverá a troca de serviços, e o de maior prioridade entrará em vigência.

Uma vez que as configurações arquiteturais do `sAddServer` impõem a necessidade de adição de novas replicas, o configurador utiliza um mecanismo de seleção de recursos para identificar quais servidores seriam mais adequados para compor o conjunto de servidores em execução. A escolha é feita orientada por informações presentes nos perfis. Para esse serviço, o perfil `pAddServer` exige que o novo servidor a compor o grupo seja o que possuir menor utilização de capacidade de processamento *(código 4.3, linha 11)*. Uma busca ao mecanismo de seleção informando os parâmetros da escolha, retornará o servidor mais adequado.

O novo serviço, então, entra em vigor, e novas réplicas são adicionadas ao conjunto

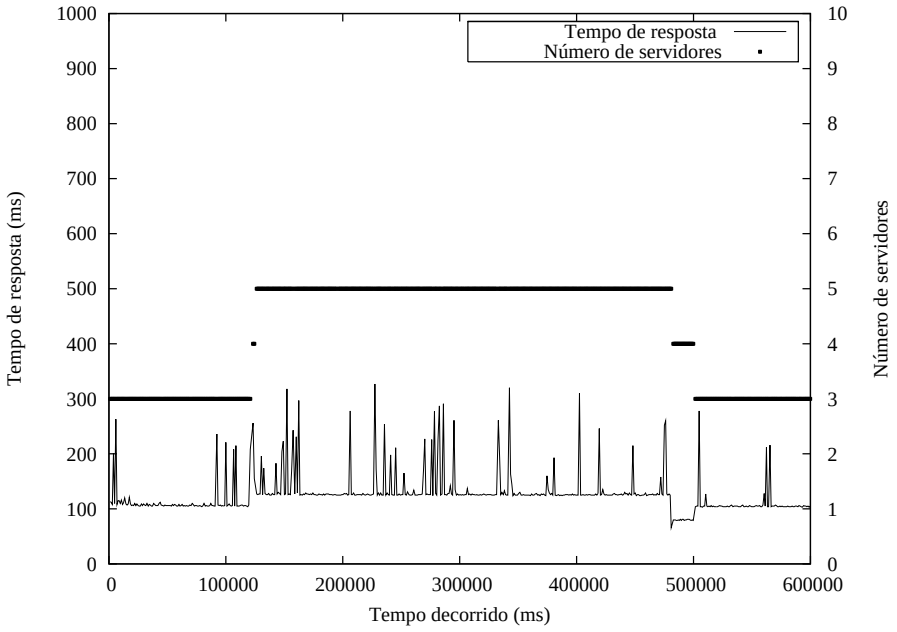

Figura 4.12: *Tempo de resposta percebido pelo cliente em aplicação paralela com configuração dinâmica da infra-estrutura através do framework CR-RIO e imposição de sobrecarga de processamento nos servidores. O número de servidores aumenta para tentar manter os tempos de resposta dentro dos limiares desejados.*

de servidores em execução até que o serviço de maior prioridade (`sMaintain`) possa ser novamente estabelecido. O comportamento do serviço `sAddServer` é similar ao operador de adaptação `addServer()` presente em [Garlan et al. 2003], que é chave de uma das táticas (no caso `fixServerLoad`) que compõem a estratégia de adaptação.

Através da figura 4.12, é perceptível o esforço no sentido de manter o tempo de resposta, que é um dos requisitos apetecidos pelo cliente, entre 80 e 150 ms. Percebemos que com 2 servidores adicionais, totalizando 5, o cliente volta a receber suas respostas dentro de limites temporais considerados satisfatórios e o serviço `sMaintain` pode ser restabelecido.

Passados 8 minutos (480.000 ms) do início da execução, quando a carga submetida aos servidores retorna aos patamares iniciais, o `Contractor` percebe que o valor da propriedade tempo de resposta está abaixo do limite estipulado no contrato para o serviço `sMaintain`. O gerente de contrato é novamente avisado sobre o não atendimento das restrições presentes nos perfis do serviço corrente e procura estabelecer o próximo serviço válido. Seguindo a regra de negociação, o próximo serviço é o `sAddServer`, portanto suas restrições são enviadas para o `Contractor`, que verifica se estas podem ser atendidas. Logo na primeira propriedade do perfil `pAddServer` *(código 4.3, linhas 6-12)*, que é pré-requisito para o estabelecimento do serviço `sAddServer`, a restrição de tempo de resposta

não é atendida, pois está acima do valor informado no perfil, e o gerente é informado. Então, ele procura pelo próximo serviço: `sRemoveServer`. Suas restrições são enviadas para o `Contractor`, que confirma o atendimento destas após verificar o valor atual das propriedades dos recursos monitorados junto aos `ResourceAgents`.

O serviço `sRemoveServer` procede de forma semelhante ao operador de adaptação `remove()` [Garlan et al. 2003], que embora não seja explicitado nos exemplos dentro de estratégias de reparo nos trabalhos pesquisados, é factível uma tática de adaptação que contemple seu uso, em casos onde é possível remover servidores buscando o uso eficiente dos recursos computacionais. O configurador recebe as primitivas arquiteturais referentes esse serviço, que determina a remoção de réplicas do conjunto de servidores em execução. O mecanismo de seleção de recursos é consultado novamente para identificar os servidores mais adequados para serem removidos, guiado por informações expressas nos perfis vinculados ao serviço. Para o serviço `sRemoveServer`, o perfil `pRemoveServer` exige que o servidor a deixar o grupo seja o que possuir maior utilização de capacidade de processamento *(código 4.3, linha 17)*. Quando ele sai do grupo apenas o serviço associado ao grupo em questão não é mais prestado, porém os demais continuam a ser executados.

Escolhemos remover o servidor mais carregado do conjunto, por acreditarmos que em um grupo de máquinas sobrecarregadas, a que apresenta maior utilização de processamento é a mais concorrida pelas aplicações em geral, e disponibilizando-na para execução de outros serviços, um melhor aproveitamento dos recursos compartilhados é obtido. Porém uma simples mudança nas políticas de alocação/desalocação feita textualmente nos contratos, permite que o comportamento seja completamente diferente, destacando certa flexibilidade presente na proposta. Por exemplo, se quisermos que a máquina a deixar o grupo seja a menos carregada, basta alterarmos a restrição `allocationPolicy` presente no perfil `pRemoveServer` para `lowLoad` ao invés de `highLoad`.

O serviço `sRemoveServer` é estabelecido, e réplicas são removidas do conjunto de servidores em execução até que o serviço de maior prioridade (`sMaintain`) possa ser novamente instituído. Com a remoção de 2 servidores, retornando à quantidade inicial 3, o cliente volta a receber suas respostas dentro de limites temporais considerados satisfatórios e o serviço `sMaintain` pode ser firmado novamente.

A tabela 4.1 resume o mapeamento entre os serviços do nível arquitetural e sua codificação no nível de implementação feita no `Configurator`. São rotinas de baixo nível para adição e remoção de servidores a grupos e para transferência de clientes entre grupos.

- Implementação

Tabela 4.1: *Mapeamento entre o nível arquitetural e o nível de implementação.*

Nível Arquitetural	**Nível de Implementação**
Serviço sAddServer	EncontrarMelhorServidor ConectarAoGrupo IncrementarContadorDeMembrosDoGrupo
Serviço sRemoveServer	EncontrarMelhorServidor DesconectarDoGrupo DecrementarContadorDeMembrosDoGrupo
Serviço sMoveClient	EncontrarMelhorGrupoDeServidores DesconectarDoGrupoAtual ConectarAoNovoGrupo

Para implementação deste exemplo, além da aplicação em si com seus requisitos funcionais, é necessária a codificação de alguns componentes do CR-RIO, os *hotspots*. Especializações de `ResourceAgent`, para monitorar, e de `ResourceState`, para armazenar os valores das propriedades de verificação das categorias de interesse `Replication`, `Client` e `Transport` foram desenvolvidos. Para `Replication`, foi criada a classe `ReplicationAgent`, que utiliza funções disponíveis (`getView()`) na biblioteca *JGroups* para descobrir o número de servidores ativos em determinado grupo e `ReplicationState` que armazena tais informações. Para a categoria `client`, a classe `ClientAgent`, que acessa métodos na própria aplicação para valorar o tempo de resposta, foi criada. Tais métodos podem ser embutidos na aplicação através do uso de tecnologias como Javassist (*Java Programming Assistant*), um *toolkit* baseado em reflexão, que utiliza manipulação de *bytecodes* para permitir que classes de uma aplicação Java sejam modificadas antes do seu carregamento na máquina virtual [Chiba e Nishizawa 2003], provendo transparência para a aplicação cliente. `ClientState` é responsável por guardar as informações adquiridas pelo `ClientAgent`. As propriedades da categoria `Transport` são mapeadas para a classe `TransportAgent`, que interage com o NWS para adquirir informações sobre as condições do enlace de comunicação entre o cliente e o grupo de servidores, e armazenam suas informações em `TransportState`.

As classes `ReplicationAgent`, `ClientAgent` e `TransportAgent` herdam a superclasse `ResourceAgent` e implementam seu método `getResourceValues` como pode ser observado pelo diagrama de classes da figura 4.13. Através da implementação desse método é efetu-

ada a leitura das propriedades do recurso ao qual a classe especializada se refere. Além disso, o período de monitoração (`probePeriod`) pode ser redefinido. `ReplicationState`, `ClientState` e `TransportState` herdam a classe `ResourceState` e definem, respectivamente, os atributos `numberOfReplicas`, `responseTime` e `latency`/`bandwidth`. Atributos que serão preenchidos no monitoramento e utilizados pelos `Contractors` para validar as restrições associadas a um determinado serviço. Apenas o atributo `change` é definido na superclasse, que indica se os valores dos atributos definidos em uma classe especializada foram alterados. Essa informação é utilizada pelo `ResourceAgent`, que quando detecta que os valores desses atributos foram alterados, dispara a atividade de verificação de restrições, implementada no `Contractor`. Através do uso do atributo `change`, na criação de uma classe especializada de `ResourceAgent`, pode ser definida uma função de variação (como por exemplo utilizando o `tuning`) que identifique quando uma mudança significativa ocorre.

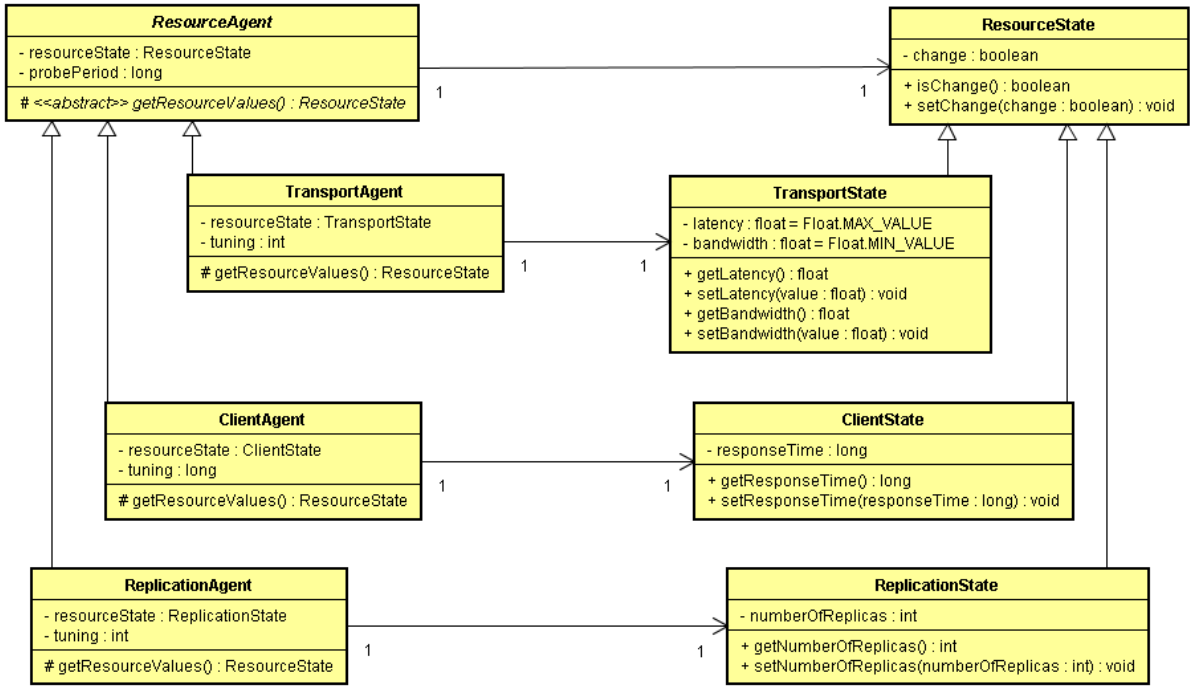

Figura 4.13: *Especializações dos `ResourcesAgents` e `ResourceStates` para as categorias `Replication`, `Client` e `Transport`.*

Um `Contractor` específico para as restrições de qualidade descritas no contrato também deve ser criado. Uma recorrência de geração código para os `Contractors` foi identificada, assim como para os `ResourceState`, ou seja, sua criação segue uma "receita padrão", então seu código pode vir a ser gerado automaticamente a partir dos contratos.

Como o `Contractor` é o responsável por gerenciar o monitoramento feito pelos `Re-`

`sourceAgents`, cada *host* que possuir alguma propriedade de recurso a ser monitorada e verificada deve ter o `Contractor` implantado. O último *hotspot* é o `Configurator`, específico para cada tipo de aplicação e cujo comportamento é moldado pelo contrato. Neste caso, é de responsabilidade do `Configurator` consultar o mecanismo de seleção de recursos, buscando pelo servidor mais apropriado para entrar e sair do grupo, nos serviços `sAddServer` e `sRemoveServer`, respectivamente; e pelo grupo mais adequado para a migração do cliente no serviço `sMoveClient`. Também cabe a ele efetivar as ações necessárias para entrada, saída e migração. O ambiente de configuração *Architecture Configurator* [Lisbôa 2003] pode ser utilizado como responsável por prover tais tarefas, porém seu alto custo das chamadas e a carência de suporte a grupos em sua versão disponível no momento, nos desencorajou de empregá-lo.

A figura 4.14 mostra a especialização do `Contractor` e do `Configurator` através da criação de uma classes especializada de `Contractor` (`ContractorCSR`) e da implementação da interface `Configurator` (`ConfiguratorCSR`) para uma aplicação Cliente-Servidor Replicado (CSR).

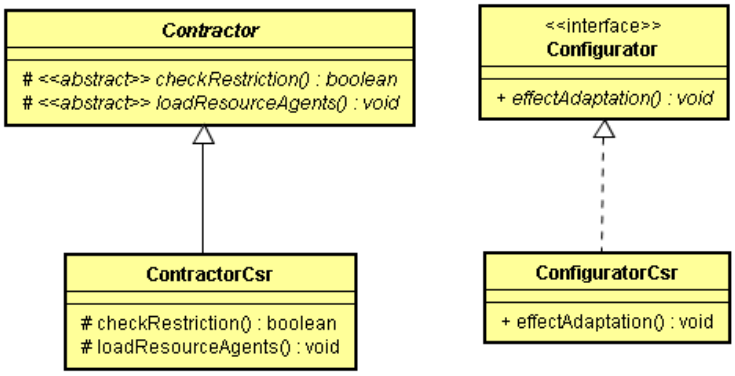

Figura 4.14: *Especializações do `Contractor` e do `Configurator` para aplicação CSR.*

A classe `ContractorCSR`, ao herdar `Contractor`, é obrigada a implementar os métodos abstratos `checkRestriction` e `loadResourceAgents` definidos na superclasse. A função do primeiro método é verificar se os valores monitorados satisfazem às restrições de qualidade associadas aos serviços declarados no contrato da aplicação. No segundo, é implementado o carregamento das classes especializadas de `ResourceAgent`, que são responsáveis pelo monitoramento das propriedades de recurso utilizadas nas restrições desse contrato. `ConfiguratorCSR` implementa o método `effectAdaptation` definido na interface, cuja função é traduzir as configurações arquiteturais descritas em um serviço para ações do nível de sistema que garantam o estabelecimento do serviço.

Para o gerenciamento do contrato para um cliente da aplicação, é necessário implan-

tar um `Contractor` (por contrato) específico da aplicação nas máquinas onde estão os `ClientAgent`, `ReplicationAgent` e `TransportAgent`. Na implementação atual do *framework*, os `Contractors` e os `ResourceAgents` devem se comunicar localmente, objetivando reduzir as freqüentes transferências de dados de monitoração na rede. O `Contract Manager`, responsável pelo gerenciamento do contrato, pode ser único para vários clientes acessando um serviço caso este se restrinja a um ambiente de rede local. Isso é possível porque o *framework* é capaz de carregar e gerenciar vários contratos simultaneamente [Corradi 2005], então com a definição de um contrato para cada cliente, expressando sua qualidade desejada, o gerenciamento independente é factível. O fato de termos definido um grupo para cada cliente, possibilita a entrada/saída de servidores em um determinado grupo sem interferir no desempenho dos outros clientes vinculados a seus respectivos grupos.

Se o serviço exceder os limites de uma LAN - *Local Area Network*, a implantação de um `Contract Manager` em cada máquina cliente se torna mais viável, evitando a comunicação remota entre o `Contractor` e o `Contract Manager`. Essa solução distribuída é mais adequada, pois a falha de um *Contract Manager* não interfere no gerenciamento dos contratos dos outros usuários da aplicação.

A configuração experimental não mantém constante a carga total dos servidores nem controla a variação na latência entre os caminhos de rede. Portanto o tempo de resposta de uma requisição ocorrido em um dado momento não pode ser comparado diretamente com o de outra que ocorreu no mesmo momento em configurações diferentes (por exemplo com e sem adaptação). Estamos interessados apenas nos valores relativos do conjunto de requisições. Foram utilizadas máquinas pertencentes a um *grid*, com diversas aplicações de usuários em execução, o que explica a variação dos tempos de resposta medidos.

B) Gerando atraso no enlace de comunicação

Em um segundo experimento com a configuração dinâmica da infra-estrutura da aplicação foi mantida a carga de processamento ao conjunto de servidores, porém foi inserido um atraso de 200 ms na comunicação entre o cliente e os servidores após 5 minutos (300.000 ms) do início da execução, como forma de simular um aumento repentino no tráfego da rede. Como conseqüência, o tempo de resposta percebido pelo cliente sofre um aumento considerável (como pode ser visto pela figura 4.15 no início da área marcada aos 300.000 ms de execução). Esse aumento é identificado pelo `ResourceAgent` que repassa o valor monitorado para o `Contractor`, que nota a não conformidade com os valores das restrições

presentes no serviço inicial (`sMaintain`).

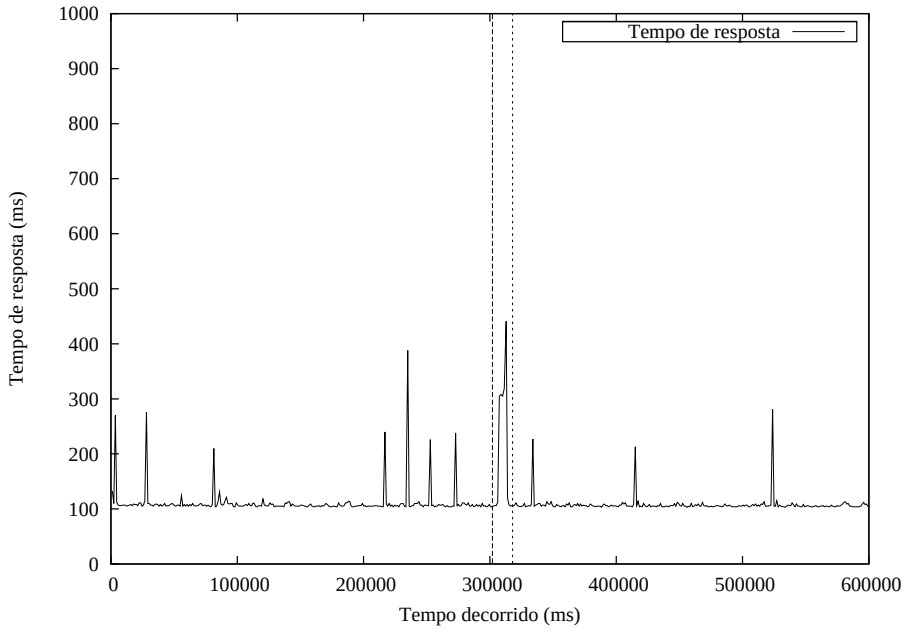

Figura 4.15: *Tempo de resposta percebido pelo cliente em aplicação paralela com configuração dinâmica da infra-estrutura através do framework CR-RIO e imposição de latência na comunicação. No período de tempo compreendido entre as áreas tracejadas ocorre a mudança de grupo do cliente.*

O gerente de contratos é informado que as restrições referentes ao serviço corrente não podem mais ser atendidas e busca pelo próximo serviço na cláusula de negociação. As restrições dos serviços `sAddServer` e `sRemoveServer` são passadas para verificação feita pelo `Contractor`, mas falham na checagem da latência de comunicação e do tempo de resposta, respectivamente. O gerente transmite, então, as restrições do serviço `sMoveClient`, e recebe uma confirmação de atendimento de suas restrições.

Para o estabelecimento do serviço, o configurador necessita ser comunicado para que possa estabelecer a configuração exigida para o provimento do serviço `sMoveClient`. As configurações arquiteturais para esse serviço instituem a necessidade de mover o cliente para um grupo de servidores, que apresente a menor latência de comunicação, conforme descrito no perfil `pMoveClient` *(código 4.3, linha 19-23)* associado a esse serviço. A descoberta do grupo mais apropriado (de acordo com o que está descrito no contrato) para a transferência do cliente é feita pelo módulo de seleção de recursos, acessado pelo `Configurator`, que de posse do nome do grupo, fornece os mecanismos necessários para deslocar o cliente para o novo grupo escolhido.

Para esse experimento, foram criados alguns grupos de servidores pré-definidos, sem clientes associados a eles. A escolha do grupo para o qual o cliente será conectado no

momento da re-configuração é feita com base na qualidade do seu canal de comunicação.

A conduta do serviço `sMoveClient` se assemelha à do operador de adaptação `move(to: ServerGroupT)` [Garlan et al. 2003] presente na tática `fixBandwidth` que compõe a estratégia de reparo para latência em níveis elevados. O serviço `sMoveClient` entra em vigência, o cliente é transferido para o novo grupo, e o tempo de resposta retoma aos patamares iniciais, como pode ser visto pela figura 4.15 (no fim da área marcada, logo depois do aumento visto aos 300.000 ms de execução) e os `Contractors` identificam que o serviço de maior prioridade (`sMaintain`) pode ser novamente fixado.

4.2 Exemplo II - Aplicações com distribuição de carga

Nesta seção apresentamos experimentos envolvendo aplicações com balanceamento de carga, que é uma técnica que pode ser usada para reduzir a contenção por um determinado recurso, distribuindo os acessos dentre as instâncias redundantes que implementam um determinado serviço. As aplicações distribuídas podem empregar o balanceamento de carga em diversos caminhos e em vários níveis para aumentar a escalabilidade do sistema [Othman et al. 2004].

Sites de Internet com grandes quantidades de acessos, por exemplo, geralmente utilizam balanceamento de carga no nível de rede e no nível operacional para melhorar o desempenho e acessibilidade de certos recursos, tais como computadores na rede ou processos, respectivamente. Nosso trabalho focará na distribuição de carga no nível de rede, distribuindo acessos entre instâncias de servidores.

Para essa classe de aplicações, dividimos nossos experimentos novamente em dois conjuntos. O primeiro conjunto (seção 4.2.1) com aplicações de configuração estática e distribuição de carga através da política *Round Robin*, considerada uma estratégia não adaptativa segundo [Othman et al. 2004] por não levar em consideração condições de carga dinâmicas. O segundo experimento (seção 4.2.2), elaborado com suporte a configuração dinâmica da infra-estrutura e conta com uma política de distribuição de carga adaptativa, que escolhe o membro menos carregado como destino. E para cada conjunto foram elaboradas execuções com a simulação de uma sobrecarga nos processadores e depois de um fluxo excessivo no enlace de comunicação.

4.2.1 Aplicação monolítica com configuração estática da infra-estrutura

Nesta seção testamos o comportamento de aplicações com distribuição de carga na ausência de mecanismos para adaptação da aplicação ou da sua infra-estrutura em situações de alta disputa por servidores e de alta latência na comunicação de rede. O serviço utilizado é semelhante ao utilizado na seção 4.1, porém ao invés de o paralelizarmos, distribuímos as requisições a ele pelos servidores que o disponibilizam.

A) Gerando sobrecarga de processamento

Nas simulações com a configuração estática da infra-estrutura, ao ser inserida uma sobrecarga de processamento nas máquinas responsáveis pela execução do serviço, suas filas de recebimento de requisições ficaram abarrotadas, o que acarretou no aumento do tempo de resposta percebido pelo cliente, como pode ser visto pela figura 4.16, a partir de 2 minutos (120.000 ms) do início da execução. Apenas quando a carga extra é retirada, após 8 minutos (480.000 ms), o tempo de resposta volta a uma faixa considerada satisfatória.

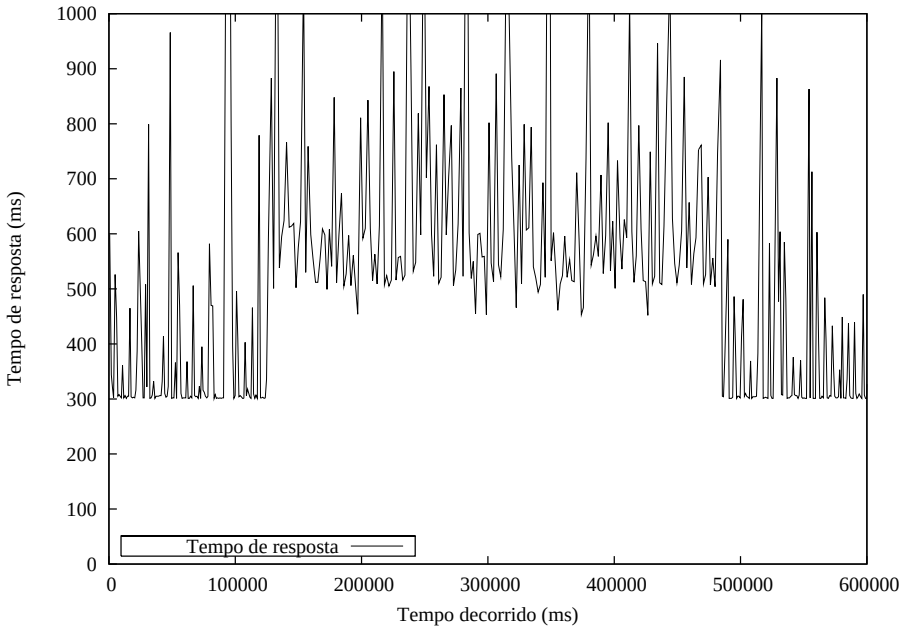

Figura 4.16: *Tempo de resposta percebido pelo cliente em aplicação monolítica com balanceamento de carga (Round Robin), configuração estática da infra-estrutura e com sobrecarga no processamento.*

B) Gerando atraso no enlace de comunicação

Um segundo teste para esta configuração foi feito interferindo, de maneira gradativa, na latência de comunicação entre o cliente e o grupo de servidores. O atraso imposto foi de 10 ms aos 200.000 ms, 100 ms aos 300.000 ms e 200 ms aos 400.000 ms. À medida em que o enlace foi submetido a uma maior latência, os clientes experimentavam um maior atraso na execução dos serviços. Tal comportamento pode ser averiguado pela figura 4.17 a partir de 200.000 ms.

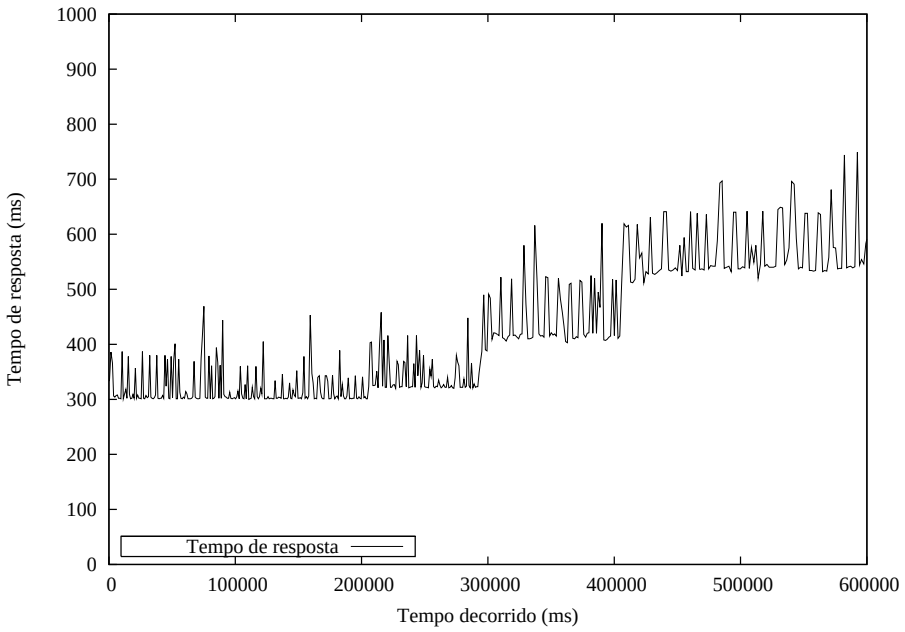

Figura 4.17: *Tempo de resposta percebido pelo cliente em aplicação monolítica com balanceamento de carga (Round Robin), configuração estática da infra-estrutura e imposição de atraso na comunicação.*

4.2.2 Aplicação monolítica com adaptação da infra-estrutura

Os experimentos de requisição de serviços balanceadas a servidores com configuração da infra-estrutura dinâmica, inicialmente foram utilizados 3 servidores, e conforme necessário, foram adicionadas mais réplicas, de forma a manter a qualidade de serviço especificada no contrato mostrado nos códigos 4.2 (os serviços e a cláusula de negociação) e 4.4 (os perfis). Os perfis deste contrato diferem do anterior apenas em relação aos limites de tempo de resposta tolerados e na orientação da forma como o conector difundirá as requisições. A arquitetura dessa aplicação e sua descrição arquitetural são as mesmas do exemplo descrito na seção anterior. A novidade é que o conector que intermedeia as requisições dos clientes

para os servidores agora as distribui para o servidor que apresenta a menor carga de processamento (ao invés de utilizar *multicast*), como orientado no contrato (*código 4.4, linhas 24-26*).

A) Gerando sobrecarga de processamento

No primeiro experimento, ao ser inserida uma sobrecarga nos processadores ativos (aqueles que estão em algum grupo de execução), o `ResourceAgent`, responsável pela monitoração dos recursos, junto com o `Contractor`, reponsável pela verificação da conformidade dos valores lidos com os desejados, previamente estipulados pelos contratos, percebem o aumento do tempo de resposta e o gerente é alertado. O gerente, de posse do próximo serviço a ser ativado (informação conseguida através da regra de negociação), isto é o `sAddServer`, verifica a possibilidade de sua implantação através dos `Contrators`, caso seja possível (ou seja, há recursos suficientes para a implantação), a necessidade de inclusão de novas réplicas é enviada para o `Configurator`.

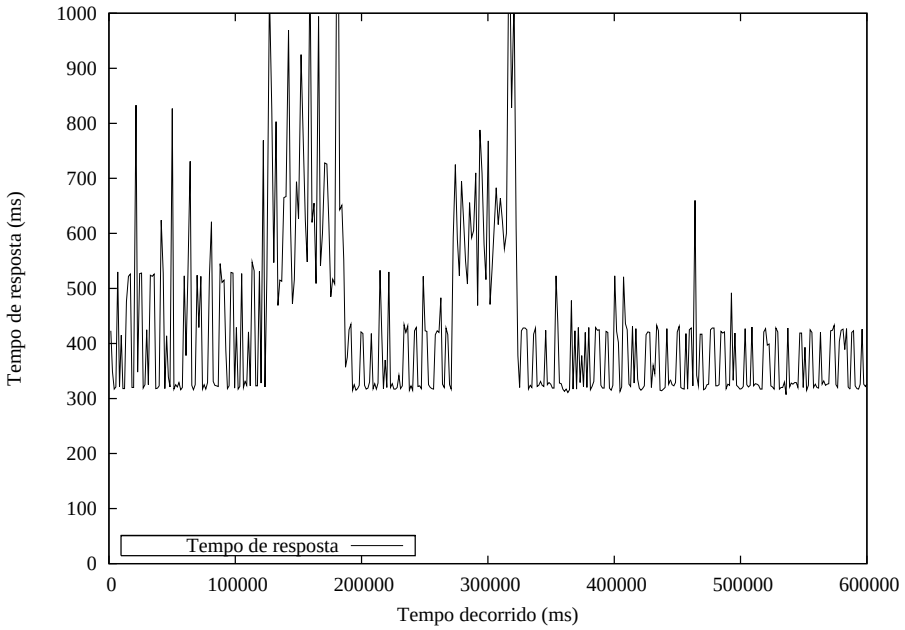

Figura 4.18: *Tempo de resposta percebido pelo cliente em aplicação monolítica com balanceamento de carga, configuração dinâmica da infra-estrutura através do framework CR-RIO e imposição de sobrecarga no processamento.*

Com o auxílio do módulo de seleção de recursos, o servidor mais adequado (seguindo orientação presente no *código 4.4, linha 11*) para compor o grupo de execução é encontrado. As requisições feitas pelos clientes, com a adição de mais réplicas ao conjunto em execução, são melhor distribuídas entre estas, ou seja, a taxa de chegada em cada servidor diminui, encurtando a fila de requisição. Com o intuito de tentar melhorar ainda mais

Código 4.4 - *Perfis do contrato apresentado em código 4.2 para aplicação com distribuição de carga.*

```
1: profile {
2:       Client.responseTime <= 400;
3:       Client.responseTime >= 300;
4:       Replication.replicaMaint := maintain;
5: } pMaintainServer;
6: profile {
7:       Client.responseTime > 400;
8:       Transport.latency <= 15;
9:       Replication.numberOfReplicas < 30;
10:      Replication.replicaMaint := add;
11:      Replication.allocationPolicy := lowLoad;
12: } pAddServer;
13: profile {
14:      Client.responseTime < 300;
15:      Replication.numberOfReplicas > 1;
16:      Replication.replicaMaint := release;
17:      Replication.allocationPolicy := highLoad;
18: } pRemoveServer;
19: profile {
20:      Transport.latency > 15;
21:      Client.responseTime > 400;
22:      Client.linkPolicy := lowLatency;
23: } pMoveClient;
24: profile {
25:      ServerGroup.distribPolicy := bestLoad;
26: } pCom;
27: profile {
28:      Client.responseTime < 300;
29:      Replication.numberOfReplicas = 1;
30:      Replication.replicaMaint := maint;
31: } pNoMoreServersToRemove;
32: profile {
33:      Client.responseTime > 400;
34:      Replication.numberOfReplicas = 30;
35:      Replication.replicaMaint := maint;
36: } pNoMoreServersToAdd;
```

a qualidade de serviço oferecida, foi implementada uma política de distribuição de carga adaptativa, que escolhe o membro menos carregado, baseado nas leituras efetuadas pelo NWS, como destino para as requisições.

O resultado deste experimento pode ser examinado na figura 4.18. após o aumento do tempo de resposta percebido com a sobrecarga dos servidores, réplicas são adicionadas pelo configurador com o intuito de reduzir o tempo de resposta. As requisições dos clientes passam a ser distribuídas para os servidores pertencentes ao conjunto, com base em sua carga de processamento, resultando em tempos de atendimento do serviço mais baixos na maioria do casos. Porém, como o ambiente de execução é compartilhado com aplicações de diversos usuários, há momentos em que mesmo o servidor menos carregado do grupo responde às requisições com um atraso longo, isso ocorre devido à grande concorrência de acesso às máquinas (figura 4.18 por volta dos 5 minutos de execução, ou seja, 300.000 ms).

B) Gerando atraso no enlace de comunicação

Ao afetar o enlace de comunicação, no segundo experimento, com o aumento da latência em 10 ms (figura 4.19, logo após 200.000 ms), o tempo de resposta notado pelo cliente permanece respeitando os limiares estipulados no contrato para o serviço preferencial (`sMaint`) e portanto nenhuma medida adaptativa é tomada. Com o aumento brusco do atraso na comunicação de 100 ms (aos 5 minutos, ou seja 300.000 ms), o tempo de resposta no cliente passa a extrapolar os limites desejados.

O `Contractor` ao perceber o não atendimento aos requisitos do contrato, comunica tal fato ao gerente. Este por sua vez tenta estabelecer o próximo possível serviço. As restrições de cada próximo serviço (`sAddServer`, `sRemoveServer` e `sMoveClient`) são passadas para o `Contractor` que verifica a possibilidade de atendimento. Os dois primeiros serviços são impossibilitados de serem oferecidos por não atenderem às restrições de latência e tempo de resposta, respectivamente. O último (`sMoveClient`) possui suas restrições atendidas e pode ser provido. O gerente, ao receber essa informação, comunica ao `Configurator` para que este estabeleça a nova configuração: o cliente conectado ao grupo de menor latência no enlace. O módulo de seleção é consultado para encontrar o grupo com melhor enlace e o cliente é conectado a ele, e conseqüentemente, o cliente consegue experimentar tempos de resposta satisfatórios novamente.

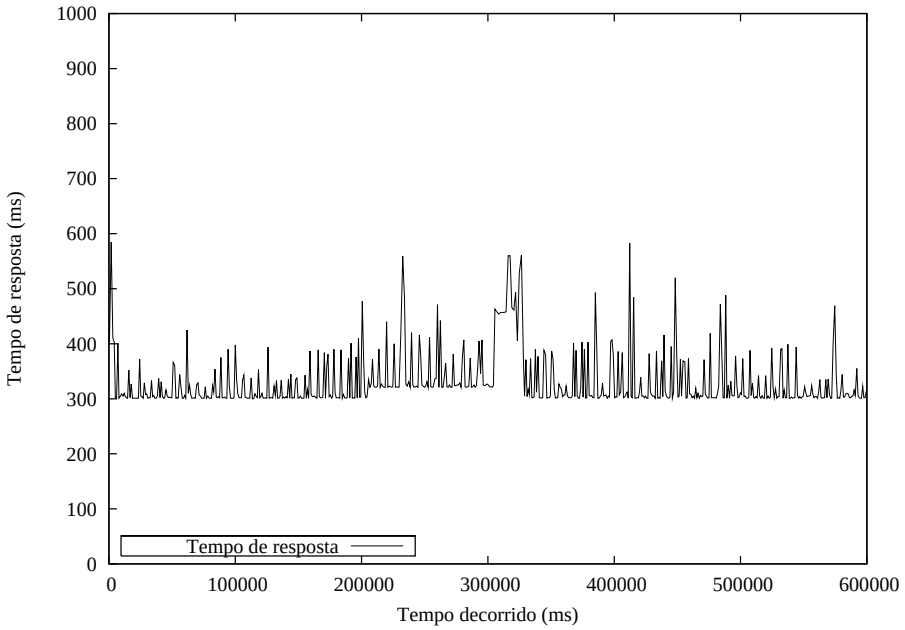

Figura 4.19: *Tempo de resposta percebido pelo cliente em aplicação monolítica com balanceamento de carga, configuração dinâmica da infra-estrutura através do framework CR-RIO e imposição de atraso na comunicação.*

4.3 Exemplo III - Outras aplicações

O propósito dessa seção é descrever aplicações potenciais que poderiam se beneficiar com o suporte do *framework* CR-RIO. Inicialmente apresentamos aplicações com tolerância a falhas e na subseção 4.3.2 mostramos as aplicações *workflow*.

4.3.1 Aplicações tolerantes a falhas

Diferentemente do exemplo Aplicações *Bag of Tasks* em paralelo, apresentado na seção 4.1, no qual a replicação foi empregada no intuito de aumentar o poder de processamento para a realização de uma tarefa, as réplicas do componente servidor neste exemplo serão utilizadas para garantir a disponibilidade do serviço mesmo em caso de falhas de alguns desses componentes.

É importante ressaltar que esse conjunto de réplicas é transparente para o cliente, que tem a visão abstrata de um único componente servidor, que é formado por um conjunto de componentes replicados. Assim, a falha de uma réplica individual passa a ser considerada isoladamente da falha do grupo como um todo. Protocolos de coordenação se fazem, então, necessários de forma a assegurar a consistência de estado e a transparência do conjunto, além de serem responsáveis pelo controle da concorrência e pela recuperação em situações

de falhas parciais ou totais das réplicas. A partir das funcionalidades do protocolo *State Transfer* [Ban] e da possibilidade de substituí-lo de acordo com as necessidades do usuário, providas pelo *JGroups*, tais necessidades podem ser supridas.

State Transfer é responsável por tratar as requisições de estado do grupo e responder para o membro. Quando um novo membro entra no grupo e requisita pelo estado (`GET_STATE`) do grupo, o processo de transferência é iniciado e o novo membro do grupo receberá o estado. Mesmo durante o processo em que o estado é transferido, novas mensagens podem chegar. Elas serão gerenciadas pelo coordenador de estado (`state coordinator`), que é o coordenador do grupo. Com o uso desse protocolo, qualquer novo membro de um grupo pode ter seu objeto local com estado idêntico aos objetos dos membros mais antigos. Isso é feito através de uma tabela *hash* distribuída, que mantém a referência a todos objetos (registrados) dos membros do grupo. Maiores detalhes podem ser encontrados em [Ban].

Consideremos uma aplicação cliente-servidor onde o servidor recebe requisições para a atualização de entradas em uma base de dados. O principal requisito não-funcional desta aplicação é que este servidor tenha um alto grau de disponibilidade, ou seja, alguma estratégia de tolerância a falhas deve ser fornecida. O código 4.5 mostra uma forma de replicação que pode ser provida para a aplicação. Estes mecanismos de tolerância a falhas podem ser encapsulados em módulos não-funcionais da aplicação, ou seja, em conectores [Loques et al. 1997], como apresentado na figura 4.20. Nesta, o conector aparece distribuído, no lado dos servidores ele representa a parte servidora da estratégia de replicação com tolerância a falhas e disponibiliza os algoritmos de eleição e consistência de estado, entre outros mecanismos necessários para o gerenciamento das réplicas. No lado do cliente, ele é responsável por disponibilizar algoritmos de comparação de resultados, tais como a escolha do primeiro resultado a chegar, a composição de resultados, ou o resultado mais freqüente (maioria) para ser passado para o cliente.

As diversas técnicas de replicação existentes que provêem tolerância a falhas são classificadas em passivas e ativas [Lung 1996]. Na primeira abordagem, somente uma das réplicas (primária) recebe, executa e responde as invocações dos clientes. As demais réplicas do conjunto (*backups*) têm a função de substituir a réplica primária em situações de falha. Os estados dos membros se mantêm mutuamente consistentes através de mensagens de *checkpoint* enviadas pela réplica primária informando seu estado para as réplicas de *backup*. Caso a réplica primária falhe, um algoritmo de eleição seleciona a nova réplica primária dentre as de *backups*, que assume a execução da operação a partir do *checkpoint*

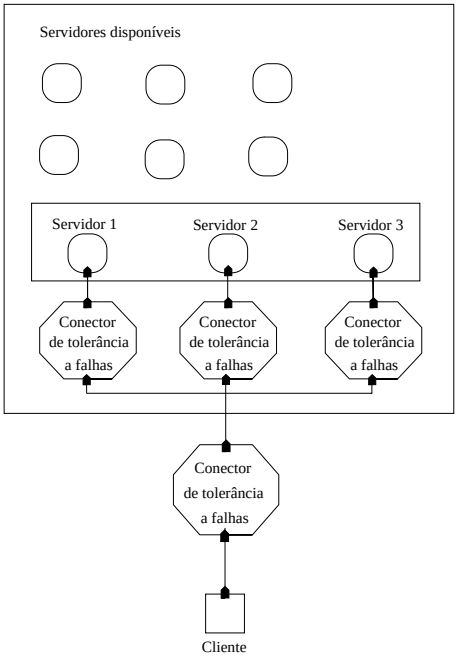

Figura 4.20: *Arquitetura da aplicação tolerante a falhas.*

mais recente.

Na abordagem de replicação ativa, ou também conhecida como *State machine* (descrita em [Schneider 1990]), o estado das máquinas é replicado e cada réplica não faltosa do grupo é ativa, ou seja, executa as mesmas requisições na mesma ordem e produz o mesmo resultado. O custo computacional desta abordagem, porém, é maior em relação à primeira, pois são alocados mais recursos do sistema (memória, processador, etc.), o fluxo de mensagens no meio de comunicação é mais elevado e alguma forma de comparação dos resultados passa a ser necessária (retornar o primeiro resultado, retornar o resultado mais freqüente ou retornar uma concatenação de resultados).

Qualquer que seja a técnica de replicação com tolerância a falhas utilizada, é necessária a instanciação de módulos extras nos conectores que implementem as funcionalidades necessárias, tais como os algoritmos de eleição de réplica primária, algoritmos para consistência de estado (*checkpoints*) e algoritmos para comparação de resultados. A utilização de bibliotecas de comunicação de grupo pode tornar a implementação destes módulos mais simples e mais rápidas. O *JGroups*, por exemplo, elege a réplica primária, ou coordenador, baseado no tempo de execução das réplicas, ou seja, a que está há mais tempo executando é eleita caso a réplica primária falhe.

No código 4.5, a estratégia de replicação com tolerância a falhas *leader_follower* [Lung 1996] presente no contrato, é identificada pelo conector responsável pela ligação do cliente com o servidor *(código 4.5, linha 4)*. Nessa estratégia, todas as réplicas recebem

e executam todas as requisições, ou seja, são ativas, mas somente uma réplica responde as requisições do cliente, a líder. Outras estratégias poderiam ser escolhidas, como por exemplo, exigir que um número desejado réplicas responda, ou a maioria, ou até mesmo todas. Essas estratégias seriam mapeadas para o nível de implementação, através da instanciação de conectores mais adequados que, no caso de utilização de mecanismos de comunicação de grupo como o *JGroups*, se diferenciariam no ajuste da propriedade `GroupRequest`, mostrando mais uma vez a maleabilidade da proposta.

Código 4.5 - *Contrato para aplicação tolerante a falhas.*

```
1:  contract {
2:    service {
3:        instantiate server with pToleranteFalhas;
4:        link client to server by leader_follower;
5:    } sReplicacao;
6:    service {
7:        instantiate server with pAddReplicas;
8:        link client to server by leader_follower;
9:    } sAddReplica;
10:   negotiation {
11:     not sReplicacao → sAddReplica;
12:     sAddReplica → sReplicacao;
13:   }
14:  } CSTF;
15: profile {
16:       Replication.numberOfReplicas = 3;
17:       Storage.consistency := true;
18:       Replication.replicaMaint := maint;
19: } pToleranteFalhas;
20: profile {
21:       Replication.numberOfReplicas < 3;
22:       Storage.consistency := true;
23:       Replication.replicaMaint := add;
24: } pAddReplicas;
```

O serviço `sReplicação` é composto pela instanciação de três réplicas do componente servidor, determinado pelo grau de replicação exigido, presente no perfil `pToleranteFalhas` *(código 4.5, linhas 15-19)*, e pela ligação destas instâncias ao componente cliente através

do conector *leader_follower (código 4.5, linha 4)*. Além do grau de replicação, o perfil presente no serviço principal, `pToleranteFalhas`, também expõe a necessidade de consistência de estado entre as réplicas *(código 4.5, linha 17)*. Caso réplicas falhem, novas são inseridas no grupo, de forma a manter o número desejado presente no contrato, e recebem o estado do grupo, visando manter a consistência de estado, antes de iniciar o recebimento de requisições dos clientes.

Utilizando o *framework* CR-RIO para dar suporte a tal aplicação, percebemos que os *hotspots* `ResourceAgents` e `Contractor`, estão anexados a um conector de tolerância a falhas. O `ResourceAgent` é responsável por verificar o número de membros no grupo e o `Contractor` por detectar se o número de réplicas não faltosas satisfaz a restrição imposta pelo contrato. Caso não seja respeitada, o gerente é informado sobre o desrespeito às exigências do serviço, e consulta o próximo serviço de acordo com a regra de negociação *(código 4.5, linhas 10-13)*. Neste caso o próximo serviço é o `sAddReplica`, o gerente pede a verificação de suas restrições junto ao `ResourceAgent` e ao `Contractor`, e caso possam ser satisfeitas esse serviço entrará em vigor. Suas primitivas arquiteturais são, então, passadas para o `Configurator`, que tomará a providência de instanciar mais uma réplica segundo orientação presente no perfil desse serviço *(código 4.5, linha 23)*. Ao se juntar ao grupo, a nova réplica busca por atualizações de estado, através de primitivas como `getState`, para manter seu estado consistente com o restante do grupo. Após esta fase, o `ResourceAgent` volta a se preocupar com as possíveis falhas das réplicas, objetivando manter a qualidade do serviço como relatado no contrato da aplicação.

A existência de bibliotecas específicas para a comunicação de grupo com tolerância a falhas, que implementam serviços de *membership* e de comunicação de grupo oferecendo as funcionalidades necessárias para a detecção de falhas, manutenção de estado das réplicas e gerenciamento do grupo formado pelas réplicas de forma bem definida facilitam a implementação desse exemplo. Assim os *hotspots* `ResourceAgents` e `Contractor` apenas mapeiam chamadas a essas funções definidas nessas bibliotecas. É interessante ressaltar que o próposito dessa seção não é propor mecanismos de tolerância a falhas, apenas mostrar a possibilidade de mapeamento de diversas aplicações para nossa proposta.

4.3.2 Aplicações *workflow*

Muitas aplicações para *grid*, como por exemplo, de cálculos astronômicos ou de bioinformática, requerem processamento *workflow* [Yu et al. 2005], em que as tarefas são executadas baseadas em seu controle ou na dependência dos dados. O *workflow* é composto

pela conexão de múltiplas tarefas de acordo com sua dependência. A estrutura do *workflow* indica o relacionamento temporal entre as tarefas. Em geral, pode ser representado com um DAG (*Directed Acyclic Graph*), em outros casos com um *non*-DAG.

Esse tipo de aplicação levou ao desenvolvimento de alguns sistemas de gerenciamento de *grid workflow* com algoritmos de escalonamento como GraADS [Berman et al. 2005] e Condor [Tannenbaum et al. 2002] entre outros. Esses sistemas facilitam o fluxo de trabalho da execução da aplicação em *grids*, entretanto a maioria não se preocupa com requisitos de qualidade de serviço do usuário, como *deadlines* ou preços a pagar.

Normalmente, a tarifação dos serviços é feita de acordo com nível de qualidade de serviço oferecido, e os provedores do serviço exigem preços mais altos, quanto maior a QoS oferecida. Entretanto, os usuários nem sempre necessitam que a execução da sua aplicação seja completada tão rapidamente quanto possível, desde que cumpra seus requisitos desejados. Nestes casos, pode ser preferível utilizar serviços com menor nível de qualidade, diminuindo o custo do serviço e possibilitando o atendimento de um número maior de clientes simultâneos. Através dos contratos, os usuários podem especificar o que desejam em termos de qualidade e quanto estão dispostos a pagar.

Na figura 4.22 é mostrado o escalonamento gerado para o DAG representado na figura 4.21, que mostra a relação de precedência entre as tarefas `A`, `B`, `C`, `D` e `E`. As tarefas `A` e `B` não necessitam da execução de nenhuma tarefa para que possa ser iniciada. A tarefa `C` possui `A` e `B` como pais, portanto necessita que estas sejam finalizadas antes do seu início. A tarefa `B` também é pai da tarefa `D`, que por sua vez possui como filha a tarefa `E`, que também depende da execução de `C`. Cada tarefa possui portas de saída e de entrada que representam as primitivas *send* e *receive*, respectivamente para a troca de informações entre elas.

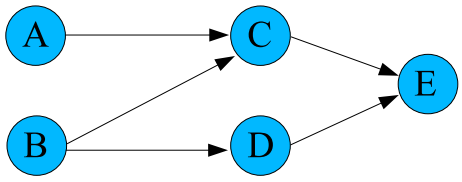

Figura 4.21: Grafo acíclico direcionado.

Um escalonador, de posse do grafo, poderia ser configurado para gerar como saída um contrato de QoS como o representado pelos códigos 4.6 e 4.7, indicando os tempos de início de cada tarefa. Então, a partir de um ambiente como o CR-RIO em conjunto com um de gerenciamento de componentes como [Santos 2005], que é capaz de implantar uma aplicação com base em sua descrição arquitetural, as tarefas podem ser carregadas para

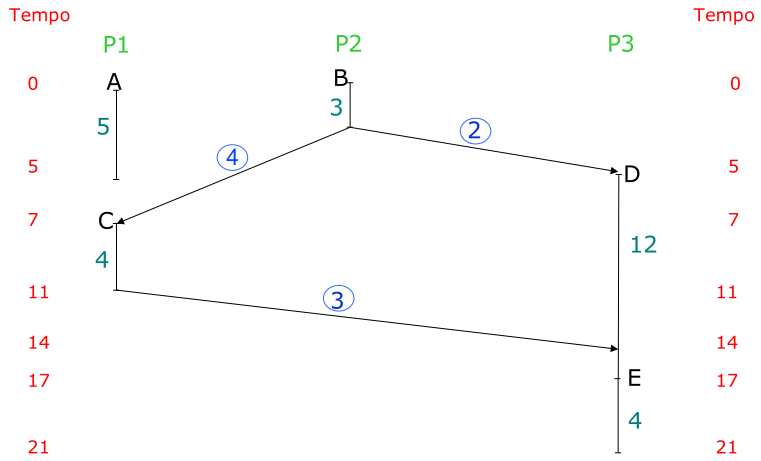

Figura 4.22: Escalonamento para o grafo da figura 4.21.

seu respectivo processador.

Para o gerenciamento dessa aplicação o `ResourceAgent` deve implementar um cronômetro que será consultado pelo `Contractor`. Se o tempo de início de uma tarefa coincidir com o tempo marcado pelo cronômetro, essa tarefa será automaticamente instanciada e iniciada. Caso a tarefa em questão tenha alguma predecessora, é feita a ligação entre elas antes do início da sua atividade. O serviço `sTarA_Bhost1_2` *(código 4.6, linhas 2-5)* inicia as tarefas `tarefaA` e `tarefaB`, caso o perfil `pTA_TB` seja atendido, ou seja, se o cronômetro estiver marcando o tempo zero. Logo após esse instante, um estágio estacionário (serviço `sWaiting`) é alcançado (de acordo com a regra de negociação), onde nenhuma tarefa deve ser iniciada. Esse serviço é ativado quando o tempo marcado é diferente dos tempos de inicialização das outras tarefas conforme descrito em seu perfil *(código 4.7, linhas 13-18)* e permanece ativo até que o perfil de outro serviço se torne válido. Aos 7 s do início da execução, a tarefa `tarefaC` deve ser iniciada, e como seu perfil torna-se válido, o serviço de instanciação dessa tarefa entra em vigor. Após sua ligação com as tarefas `tarefaA` e `tarefaB`, ela inicia sua execução. Após esse instante o serviço `sWaiting` novamente se torna o corrente até que o perfil do serviço de instanciação de outra tarefa se torne válido.

Com esse exemplo percebemos que mesmo em aplicações que exigem paralelismo entre as diversas tarefas, o *framework* CR-RIO é capaz de oferecer suporte a sua implantação. Entretanto estudos podem ser feitos no sentido de mapear também adaptações em aplicações dessa classe. Um exemplo de adaptação é a migração de uma tarefa para outro processador, assim que for identificado que este não será capaz de cumprir a execução da tarefa dentro de um determinado tempo pré-definido. Para isso são necessários me-

Código 4.6 - *Um contrato de QoS especificando requisitos desejados pelo cliente para uma aplicação workflow.*

```
1: contract {
2:   service {
3:       instantiate tarefaA at host1 with pTA_TB;
4:       instantiate tarefaB at host2;
5:       start tarefaA;
6:       start tarefaB;
7:   } sTarA_Bhost1_2;
8:   service {
9:       instantiate tarefaC at host1 with pTC;
10:      link tarefaA, tarefaB to tarefaC;
11:      start tarefaC;
12:  } sTarChost1;
13:  service {
14:      instantiate tarefaD at host3 with pTD;
15:      link tarefaB to tarefaD;
16:      start tarefaD;
17:  } sTarDhost3;
18:  service {
19:      instantiate tarefaE at host3 with pTE;
20:      link tarefaC, tarefaD to tarefaE;
21:      start tarefaE;
22:  } sTarEhost3;
23:  service {
24:      with pWaiting
25:  } sWaiting;
26:  negotiation {
27:      sTarA_Bhost1_2 → sWaiting;
28:      sTarChost1 → sWaiting;
29:      sTarDhost3 → sWaiting;
30:      sTarEhost3 → sWaiting;
31:      sWaiting → sTarA_Bhost1_2 || sTarChost1 || sTarDhost3 || sTarEhost3;
32:  }
33: } Workflow;
```

Código 4.7 - *Perfis do contrato apresentado em código 4.6 para uma aplicação workflow.*

```
1: profile {
2:         time.start = 0;
3: } pTA_TB;
4: profile {
5:         time.start = 7;
6: } pTC;
7: profile {
8:         time.start = 5;
9: } pTD;
10: profile {
11:         time.start = 17;
12: } pTE;
13: profile {
14:         time.start != 0;
15:         time.start != 5;
16:         time.start != 7;
17:         time.start != 17;
18: } sWaiting;
```

canismos para salvar o estado de execução da tarefa (e.g. através de *log*) antes da sua transferência para outro processador, para auxiliar a manter a consistência da aplicação como um todo.

4.4 Reutilização de componentes do *framework*

Como um dos atrativos da proposta CR-RIO é a reusabilidade de alguns componentes para diversas aplicações, nessa seção identificamos que partes do *Framework* são reutilizáveis e sob quais circunstâncias. Para tal, dividimos essa análise em termos de componentes de especificação, de monitoração, de gerenciamento e de configuração. A tabela 4.2 resume as unidades que podem ser reutilizadas.

Tabela 4.2: *Entidades reútilizáveis no framework CR-RIO.*

	Específico	**Reutilizável**
Especificação	Perfil, Serviço, Regra de negociação	Categorias de QoS
Monitoração		*Resource Agent*
Gerenciamento	*Contractor*[1]	*Contract Manager*
Configuração	*Configurator*[2]	

[1]Apesar do componente *Contractor* não ser reutilizável, seu código (ou seu *bytecode*) pode ser gerado de forma automatizada baseado no contrato de QoS.

[2]Esse elemento pode ser reaproveitado entre aplicações da mesma classe, com medidas adaptativas comuns a elas.

4.4.1 Especificação

Através da especificação dos contratos são descritos os requisitos não-funcionais de uma aplicação e as configurações necessárias para o estabelecimento dos serviços da aplicação com a qualidade desejada. Podemos dividir a especificação em seções de definição de categorias, de serviços, de perfis e da máquina de negociação. Dentre estas, a definição das categorias é a que apresenta maior potencial de reaproveitamento (e.g. a categoria de processamento, uma vez definida, pode ser utilizada por diferentes tipos de aplicação sem necessidade de mudanças). Mesmo que propriedades de um categoria sejam necessárias para uma aplicação e não sejam para outra, o fato de serem definidas para a primeira e reutilizadas para a segunda não interfere no seu funcionamento. As demais seções do contrato são específicas para cada tipo de aplicação e devem ser redefinidas.

4.4.2 Monitoração

Os mecanismos de monitoração são os responsáveis pela caracterização dos recursos do sistema e da sua infra-estrutura. Nossa abordagem é geral o suficiente para usar as tecnologias de sistemas de monitoramento existentes. Para conectá-las ao *framework* uma implementação do `ResourceAgent` deve ser desenvolvida para cada uma dessas tecnologias. Para cada categoria de QoS deve ser desenvolvido um `ResourceAgent` que é responsável por interagir com o mecanismo de monitoração utilizado e requisitar informações sobre o estado do recurso de interesse. Uma vez que as categorias de QoS são reutilizáveis, seus

`ResourceAgents` associados também podem ser reutilizados desde que seja utilizado o mesmo mecanismo de monitoração.

Em nossas simulações elaboramos `ResourceAgents`, com base em informações coletadas pelo NWS, capazes de colher informações sobre o enlace de comunicação, sobre o processamento e sobre o armazenamento (primário e secundário), que podem ser utilizadas por qualquer aplicação interessada em monitorar tais recursos utilizando o NWS. Uma interface padronizada de acesso a plataformas de monitoração está sendo desenvolvida [Cardoso 2005], visando permitir a reutilização independentemente do sistema de monitoração desejado.

4.4.3 Gerenciamento

A gerência é responsável por coordenar os serviços oferecidos por uma aplicação, considerando os níveis dos recursos e o nível de qualidade desejado. Para que o gerenciamento possa ser provido há a necessidade de implementar o `Contractor` que gerenciará a monitoração efetuada pelos `ResourceAgents`. Essa implementação é específica para as restrições de qualidade descritas no contrato. Porém, como dito anteriormente, foi identificada a possibilidade de geração de seu código de maneira automatizada a partir do contrato, de forma a diminuir a quantidade de codificação a ser feita pelo desenvolvedor da aplicação.

O `Contract Manager`, responsável por interpretar os contratos, iniciar tentativas de estabelecimento de serviços e renegociá-los caso necessário, é reutilizável por ser independente do serviço que está gerenciando. Sua função é coordenar os componentes, de forma a possibilitar a prestação do serviço com a qualidade desejada pelo cliente, funcionando como um "motor" para o processo de gerenciamento.

4.4.4 Configuração

O mecanismo de configuração, encarregado de efetuar as configurações necessárias no sistema para suportar os serviços oferecidos pela aplicação e de alocar recursos a serem utilizados, é específico para cada aplicação e para cada contrato. Porém ambientes de suporte a gerenciamento de configuração arquitetural como apresentados em [Lisbôa 2003, Santos 2005] podem ser utilizados. Dessa forma, configurações arquiteturais descritas nos serviços de um contrato podem ser repassadas diretamente para tais ambientes. Remanescendo apenas a necessidade da implementação ou acesso a algum mecanismo de seleção de recursos, caso a medida adaptativa exija uma escolha dentre

possíveis recursos a serem considerados na configuração.

4.5 Conclusão do capítulo

Neste capítulo foram apresentadas algumas aplicações e sua possível interação com o *framework* de forma a garantir requisitos não-funcionais exigidos pelo usuário. Através dos exemplos das aplicações *Bag of Tasks* (capítulo 4.1) e das monolíticas com distribuição de carga (4.2), ficou claro o ganho de desempenho das aplicações quando possuem seus requisitos não-funcionais sob vigilância constante. Os exemplos em geral mostraram a aplicabilidade do *framework* no gerenciamento de aplicações com requisitos de QoS. Percebemos também, que uma vez implementado o contrato para um estilo arquitetural, toda a infra-estrutura pode ser reusada com poucas alterações. Entre as aplicações *Bag of Tasks* e as monolíticas com distribuição de carga, por exemplo, a principal diferença está na forma como as requisições são distribuídas. Isso é orientado por seus respectivos contratos, que além disso, especificam as margens de tolerância para os tempos de resposta dos serviços providos e as regras para a adaptação, caso esses tempos não sejam respeitados.

Um fator de forte influência no desempenho e na precisão da adaptação das aplicações é a freqüência de captura do mecanismo de monitoração. O intervalo entre uma medição e outra geralmente é configurável, mas não pode ser muito curto, pois interfere na própria medição, mas também não pode ser muito longo, senão as consultas podem encontrar dados defasados. Para o NWS, a ferramenta de monitoração utilizada em nossos experimentos, seu consumo de recursos está disponível em [NWS]. Outro elemento que produz efeito no desempenho da aplicação é a filtragem das medidas efetuadas, pois a partir dela podemos tomar decisões mais consistentes relacionadas aos disparos de adaptações. Também relativo às medidas, deve ser considerado o efeito de margens de tolerância (*tuning*), que definiriam a granularidade das alterações que devem ser explicitadas nas medições dos recursos e repassadas ao nível de gerenciamento do contrato.

Embora os experimentos utilizem cargas de processamento simulados e introduzam atrasos na comunicação de rede forçados e não baseados em *logs* de servidores reais, elas são capazes de mostrar, em termos de desempenho, vantagens em relação a aplicações não gerenciadas, como visto pelos gráficos apresentados.

Com o gerenciamento automatizado dos recursos (alocação/desalocação dos servidores) é possível perceber e resolver problemas de alocação ineficiente dos recursos às

aplicações de forma mais rápida que em sistemas controlados com intervenção humana. E com a especificação das políticas de alocação feita nos contratos é possível a seleção dos recursos conforme a necessidade específica da aplicação.

A utilização do *framework* se mostra eficiente, não somente por viabilizar o gerenciamento de requisitos não-funcionais, mas também por permitir a evolução dinâmica das aplicações. Elas precisam evoluir dinamicamente em seus aspectos funcionais e não-funcionais, ou mesmo em sua estrutura, e com esta abordagem junto com ambiente de configuração *Architecture Configurator*, a gerência da evolução destas aplicações é facilitada, mesmo durante sua operação, com o dinamismo e consistência esperados. Isso favorece, por exemplo, a inclusão de novas políticas de distribuição e até mesmo de adaptação, e possibilita também a seleção da mais adequada.

Além disso, o modelo de contratos utilizado permite verificar em que situações um determinado serviço X pode ser provido, em que condições a aplicação transita para outro serviço e para quais serviços a aplicação pode transitar dado o serviço X. É possível também determinar em que momentos uma aplicação deixa um serviço de prioridade mais alta para prover um de menor prioridade e vice-versa.

Nossos experimentos foram elaborados em máquinas Intel® Pentium® 4 CPU 2.60 GHz, com 512 Kb de *cache* e 512 Mb de memória RAM, utilizando o sistema operacional Fedora Core 2, *kernel* 2.6 e NWS versão 2.12.2 para as máquinas servidoras. A máquina cliente utilizada foi Intel® Pentium® 4 CPU 3.20 GHz, com 1 Mb de *cache* e 512 Mb de memória RAM, utilizando o sistema operacional Fedora Core 3, *kernel* 2.6 SMP, Iproute2 versão 2.6.11 (utilizado pelo *Traffic Controller*) e NWS versão 2.12.2.

Capítulo 5

Trabalhos Correlatos

Várias pesquisas vêm sendo realizadas ultimamente na área de adaptação dinâmica no nível de implementação. Grande parte destas propõem o auto-reparo específico para um sistema, mas não provêem mecanismos externos reutilizáveis de forma a serem adicionados a outros sistemas de uma forma disciplinada como na abordagem de nível arquitetural.

Este capítulo apresenta alguns trabalhos correlacionados com a abordagem apresentada nesta dissertação. Serão mostrados trabalhos que se preocupam com os aspectos de QoS nos níveis de descrição arquitetural e de suporte em aplicações semelhantes às apresentadas no capítulo 4.

5.1 Soluções gerais

Nessa subseção serão apresentadas as propostas *Rainbow*, QuO e LuaCORBA, que assim como o CR-RIO, se preocupam em propor um *framework* completo para especificação e gerência de requisitos não funcionais para aplicações, expondo em pontos bem definidos (os *hotspots*) as questões de implementação.

5.1.1 *Rainbow*

O *Rainbow* [Cheng et al. 2002, Garlan et al. 2003, Garlan et al. 2004] permite a especificação dos aspectos a serem monitorados e em que situações uma adaptação é necessária afim de garantir os requisitos de qualidade da aplicação informados pelo projetista. Em tempo de execução da aplicação, uma infra-estrutura de suporte monitora as propriedades definidas no modelo arquitetural, avalia a ocorrência de violações nas restrições especificadas no mesmo e, se necessário, efetua adaptações no sistema.

Através da figura 5.1, podem ser vistas as etapas do processo de adaptação que não está amarrado a nenhum sistema específico, podendo assim, ser reutilizada nos diversos sistemas desenvolvidos. O sistema é monitorado (1) para observar seu comportamento; (2) os valores monitorados são abstraídos e relacionados com as propriedades do modelo arquitetural; (3) as propriedades modificadas do modelo arquitetural são analisadas para determinar se estão dentro de faixas aceitáveis; (4) Caso não estejam, reparos são disparados, o que adapta a arquitetura (5); e as mudanças arquiteturais são propagadas para o sistema em execução (6).

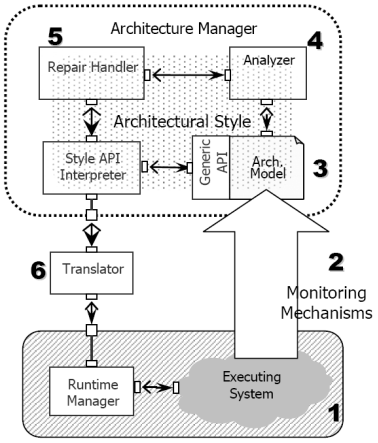

Figura 5.1: *Framework de adaptação.*

Um cenário semelhante ao proposto na seção 4.2 é apresentado em [Garlan et al. 2004], que consiste num conjunto de clientes requisitando conteúdo *web* a um grupo de servidores. Cada grupo consiste de um conjunto de servidores replicados e mantém uma fila de requisições. O cliente conecta-se ao grupo e suas requisições são enviadas à fila compartilhada pelo grupo. Os servidores do grupo retiram as requisições do grupo, processam-as e retornam seus resultados diretamente para o cliente.

Devido às orientações presentes no serviço `sAddServer` da nossa proposta *(código 4.2, linhas 05-08)*, seu comportamento é similar ao operador de adaptação `addServer()` presente em [Garlan et al. 2003], que é chave de uma das táticas (no caso `fixServerLoad`) que compõem a estratégia de adaptação como dito anteriormente. Além disso os serviços `sRemoveServer` *(código 4.2, linhas 09-11)* e `sMoveClient` *(código 4.2, linhas 12-14)* procedem de forma semelhante aos operadores de adaptação `remove()` (ausente nas táticas dos exemplos apresentados) e `move(to:ServerGroupT)` (presente na tática `fixBandwidth`) que compõe a estratégia de reparo para latência em níveis elevados. A tabela 5.1 resume as semelhanças entre os serviços definidos em nossos exemplos e os operadores de adaptação do *Rainbow*.

Tabela 5.1: *Semelhanças entres os operadores de adaptação [Garlan et al. 2003] e os serviços de adaptação da nossa proposta.*

Operadores de adaptação do *Rainbow*	**Serviços do CR-RIO**
addServer(): é aplicado ao grupo de servidores e o seu objetivo é adicionar um novo servidor ao grupo.	**sAddServer:** o *configurator* consulta o mecanismo de escolha de recursos (`getBestServers`), escolhe qual servidor será adicionado ao grupo do cliente em questão e efetiva a nova configuração. A escolha é baseada nas propriedades ajustadas da categoria *Replication*.
remove(): é aplicado a um servidor específico e o seu objetivo é removê-lo do grupo ao qual pertence.	**sRemoveServer:** o *configurator* consulta o mecanismo de escolha de recursos (`getBestServers`), escolhe qual servidor será removido do grupo do cliente em questão e efetiva a nova configuração. A escolha é baseada nas propriedades ajustadas da categoria *Replication*.
move(to:ServerGroupT): é aplicado ao cliente e utiliza o operador de consulta `findGoodSGroup(cl:ClientT, bw:float)` para encontrar o grupo de servidores com a melhor banda passante com o cliente.	**sMove:** o *configurator* consulta o mecanismo de escolha de recursos (`getBestGroup`), escolhe qual o melhor grupo de servidores, com base nas propriedades ajustadas da categoria *Client*, e efetiva a nova configuração. A possibilidade de ajustar tais propriedades dá maior flexibilidade na escolha de outros grupos (e.g. escolher outro grupo com base na latência).

O *Rainbow* utiliza o conceito de invariante para definição do nível de qualidade desejado, e através das táticas de adaptação, exprime as ações a serem tomadas se as invariantes não forem respeitadas. Essas táticas incluem os operadores de adaptação que são usados para auxiliar o processo adaptativo. Em nossa proposta temos a definição de serviços, que possuem faixas de atuação. De acordo com o nível dos recursos, os serviços são ativados e neles estão embutidas as medidas adaptativas. Tais medidas são orientadas por propriedades definidas na representação do recurso através da *QoSCategory*. Um exemplo é a utilização da propriedade `allocationPolicy` (definida no código 4.1), que orienta a escolha de novos servidores.

5.1.2 QuO

O sistema QuO (*Quality Objects*) da BBN [Loyall et al. 1998, Pal et al. 2000] é um *framework* que estende o CORBA para dar suporte a aplicações que se adaptam de acordo com a disponibilidade de recursos. Neste sistema o usuário pode definir regiões de operação, enquanto os contratos especificam o desempenho esperado da aplicação. Tais definições são feitas utilizando o conjunto de linguagens (QDL - *Quality Description Languages*) para descrição de contratos de QoS. O sistema monitora o ambiente de execução e a aplicação, e quando a aplicação muda de região de operação, *handlers* específicos da aplicação são invocados.

Em uma aplicação usando o *framework* QuO, quando um cliente faz uma chamada a um método remoto, ela é interceptada pelo Delegate (figura 5.2, passo 1). Este dispara a avaliação do contrato (2), onde são coletados os valores das condições do sistema medidos pelos objetos *SysCond* (3 e 4). O contrato é composto por um conjunto de regiões aninhadas que descrevem os possíveis estados de QoS do sistema. Cada região é definida por um predicado composto por valores dos objetos de condição do sistema (*SysCond*). O *Contract* avalia esses predicados para definir quais regiões estão ativas (5). Se uma transição de estado ocorre (quando há mudança nos valores dos predicados das regiões, de forma que uma região que estava inativa torna-se ativa), *Callbacks* – chamadas a métodos da aplicação cliente – ou chamadas a métodos dos objetos *SysCond* podem ser disparadas (6). O *Contract* passa a lista das regiões ativas para o *Delegate* (7), que de acordo com essa informação escolhe, uma alternativa para processar a chamada ao método remoto. Ele pode, por exemplo, bloquear a chamada ao método e efetuar alguma adaptação no sistema, caso o nível de QoS tenha degradado, ou simplesmente passá-la ao objeto remoto (8), caso contrário. O objeto remoto recebe e processa a chamada ao seu método e retorna o valor resultante (9). O *Contract* é então novamente avaliado, da mesma forma descrita anteriormente, e, de acordo com a informação de quais regiões estão ativas, o *Delegate* escolhe um comportamento a ser tomado, por exemplo, podendo repassar o valor de retorno para o cliente (10).

Essa proposta possui alto grau de relação com nossa proposta, os requisitos não funcionais são descritos através do conceito de regiões, que misturaram um pouco de contrato de QoS com implementação de sistema, dificultando sua legibilidade. Tal fato também compromete a separação de interesses da proposta, uma vez que embute aspectos de implementação nas descrições de alto nível dos requisitos não funcionais.

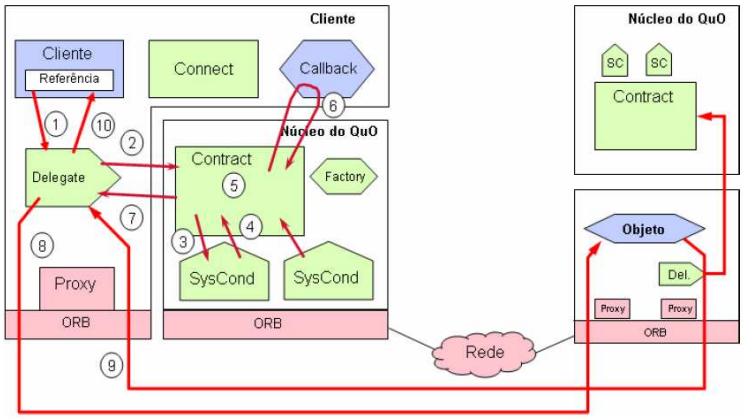

Figura 5.2: *Chamada a um método remoto em uma aplicação utilizando o Framework QuO.*

5.1.3 LuaCORBA

Outra proposta que permite às aplicações selecionar dinamicamente os componentes que melhor se adequam aos seus requisitos, verificar se seus requisitos estão sendo atendidos e reagir, quando necessário, para mantimento desses requisistos está em [Moura et al. 2002]. Para isso há uma infra-estrutura baseada na linguagem Lua e na plataforma CORBA. Há também um mecanismo (LuaMonitor) que permite o monitoramento de requisitos de qualidade, que podem ser especificados dinamicamente, e quando mudanças relevantes nas propriedades monitoradas são detectadas, estratégias de adaptação são ativadas com intuito de adequar a aplicação ao novo estado do sistema.

Nesse trabalho é proposto um sistema de compartilhamento de carga por servidores que oferecem uma mesma interface funcional. Tal compartilhamento é de responsabilidade dos clientes, que localizam dinamicamente os servidores menos carregados e passam a direcionar suas requisições para eles. Para avaliar a carga em cada servidor, duas medidas associadas a propriedades dinâmicas são utilizadas: o tempo médio de resposta do servidor (a propriedade `ResponseTime`) e a carga total da máquina hospedeira do serviço (a propriedade `LoadAvg`). Há um monitor para cada uma dessas propriedades. O primeiro monitor registra o tempo médio de resposta de um servidor às requisições feitas por seus clientes. O segundo monitor executa na máquina servidora e computa, a cada minuto, o número médio de processos na fila de prontos observados nos últimos 1, 5 e 15 minutos. Neste segundo monitor, foi definido um aspecto que representa, através de um `string`, se foi observado um aumento na carga submetida ao sistema.

Os agentes de serviço foram implementados por um *script* Lua executado nas máquinas servidoras. Esse *script* é responsável pela criação dos monitores das propriedades

`ResponseTime` e `LoadAvg`, pela criação do componente servidor (implementado por um objeto Lua) e pela exportação da oferta do serviço correspondente. Essa oferta de serviço conterá as duas propriedades dinâmicas descritas acima e duas propriedades do tipo `Object Reference`, através das quais são fornecidas as referências para os dois monitores aos *proxies* inteligentes. O *proxy* inteligente utiliza essa infra-estrutura para implementar o compartilhamento de carga. Inicialmente, ele seleciona o componente servidor que apresenta melhor desempenho e disponibilidade no momento. Essa seleção é realizada através de uma consulta ao *trader*, que utiliza como critério de ordenação das ofertas de serviço, o tempo médio de resposta dos componentes servidores e elimina os componentes hospedados onde há uma tendência de aumento de carga. Caso nenhuma oferta atenda a restrição imposta, uma segunda consulta será realizada, onde apenas a ordenação das ofertas é especificada.

Através do registro de um evento junto ao monitor de tempo de resposta associado ao componente selecionado, o *proxy* inteligente é informado de uma queda de desempenho relevante nesse componente. Esse evento foi definido como um incremento na média do tempo de resposta maior que a carga esperada para dois clientes. O evento notificado será tratado pelo *proxy* inteligente imediatamente antes da próxima invocação de serviço. Para adaptar-se à queda de desempenho no serviço representado, o *proxy* inteligente tenta localizar um componente servidor mais adequado, realizando para isso, uma nova consulta ao *trader*. Se obtiver uma oferta de serviço associada a um tempo médio de resposta considerado melhor que o apresentado pelo componente atualmente selecionado, esse componente é substituído.

Assim como em nossa proposta, esse projeto conta com uma linguagem (Lua) utilizada para descrever as estratégias de adaptação. Por ela ser uma linguagem interpretada, essas estratégias podem ser atualizadas dinamicamente. Embora a linguagem Lua contribua para a capacidade de definição dinâmica de estratégias de adaptação, o seu nível de abstração é mais baixo (nível de implementação) que a linguagem CBabel utilizada em CR-RIO. Nesta, as preocupações relativas à adaptação são expostas explicitamente no nível da arquitetura, o que facilita a descrição de técnicas genéricas de adaptação.

5.2 Aplicações específicas

Essa seção explicita algumas propostas que, de alguma forma, embutem as garantias dos seu requisitos não-funcionais na aplicação, e como elas poderiam se beneficar com a

utilização do *framework* para especificar e garantir tais requisitos.

5.2.1 *OnCall*

Em [Norris et al. 2004] é apresentado um sistema para gerenciamento dos recursos de um *cluster* de maneira econômica e eficiente. Isso é feito permitindo às aplicações negociar capacidade computacionais entre si. As decisões de alocação (e de troca de servidores) são feitas de maneira automatizada, sem a intervenção humana. Inicialmente é feito um provisionamento dos recursos para as aplicações, no qual cada nó é ligado estaticamente a uma aplicação. Depois elas podem aumentar sua capacidade de processamento adquirindo mais recursos computacionais (servidores) com outras aplicações no *cluster* ou reduzir, liberando-os.

Nesse projeto, cada aplicação desenvolve sua política de negociação dos recursos, e tem acesso às informações sobre o desempenho (na forma de uso de CPU e de disco rígido) de cada máquina onde a aplicação esta executando. As informações sobre o enlace de comunicação são irrelevantes nesse projeto e o relato da qualidade desejada aparece embutida no código da aplicação. A descrição das necessidades de cada aplicação e a partir de qual nível de utilização dos recursos ela pode "emprestar" capacidade computacional poderiam ser feitas através dos contratos de QoS. Com isso o CR-RIO poderia tratar essa negociação entre as aplicações, deixando a elas, apenas a preocupação com a funcionalidade principal de cada aplicação.

5.2.2 *AnyCast*

Na proposta *Anycast* [Zegura et al. 2000] é testado o desempenho de um sistema com servidores *web* replicados, no qual os clientes escolhem qual servidor tratará sua requisição. Isso é feito a partir de uma consulta contendo o identificador do grupo, o *Anycast Domainname* - ADN, e um critério de seleção para ser utilizado na escolha. O *Anycast Resolver* - AR, que recebe tais consultas, conhece os servidores do grupo e adquire diversas métricas de cada um deles. Tais métricas podem ser coletadas de diversas formas, como através de requisições aos servidores (*Server Probing*), essa maneira é capaz de informar sobre tempo de processamento no servidor e sobre atraso da rede; através de informações enviadas pelo servidor sobre seu estado (*Server Push*), que não mede atraso de rede, apenas fornece medições sobre o processamento; através do acesso a essas medidas (de processamento) em um arquivo remoto (*Probing for Locally Maintained Server*

Performance); ou pela coleta de informações sobre experiências anteriores diretamente no cliente (*User experience*), que possui um baixo custo de coleta. Para evitar que o AR retorne os mesmos servidores para todos os clientes, o que degradaria o desempenho, é utilizado um algoritmo que seleciona um conjunto de servidores equivalentes.

O usuário especifica os critérios de seleção no nível da aplicação através de filtros, o que provê certa flexibilidade, mas que não possui um alto nível de abstração como a especificação através de contratos. Tais filtros operam em um conjunto de membros para escolher um sub-conjunto. Eles podem ser independente de conteúdo (e.g. escolher um membro randomicamente), ou baseado em alguma medida de desempenho (e.g. o membro com maior disponibilidade de processamento). Com o suporte do CR-RIO, a escolha do melhor servidor com base em alguma métrica (especificada através do contrato) ocorreria de forma semelhante ao exemplo apresentado na seção 4.2. Dessa forma, uma maior flexibilidade é oferecida, pois uma mudança textual no contrato permitiria a adoção de outra métrica para selecionar o servidor destino.

A utilização do conceito de servidores equivalentes, empregado nessa proposta, é uma abordagem interessante para a diminuição da oscilação dos tempos de resposta, uma vez que evita que vários clientes direcionem suas requisições para um mesmo "melhor" servidor. Sua utilização em nossa proposta poderia amenizar as variações nos tempos de resposta percebidos pelos clientes nos experimentos de distribuição de carga.

5.2.3 *Cygnus*

Em [Othman et al. 2004] é proposto um sistema de monitoração e balanceamento de carga, e experimentos são realizados utilizando o suporte desse sistema e da plataforma CORBA. Para o balanceamento foram utilizadas as políticas *Round Robin*, Aleatória e *Less Loaded*, que escolhe um membro do grupo com a menor carga. Para esta última, as métricas de carga de CPU e de número de requisições por segundo são suportadas. Os limiares toleráveis para um servidor e os desejados para alocação de um novo, são definidos de antemão. Com ajuste de um valor de amortecimento é possível definir quanto o novo valor medido interfere nas decisões de balanceamento.

Assim como a proposta *Anycast*, não há a preocupação com a especificação do nível de qualidade desejado em alto nível. Isso é feito através do estabelecimento de um limite no número de requisições por segundo, que acima desse valor, o servidor é informado para redirecionar suas requisições para o Cygnus, para que estas sejam enviadas para outro servidor com menor carga. Para beneficiar tal proposta com o uso do nosso *framework*,

poderia ser definido um contrato especificando o limite de requisições por segundo que cada servidor pode atender, e tal propriedade (requisições/segundo) permanece sob constante monitoração em todos os servidores. Caso atinja o máximo desejado, as requisições são direcionadas para outro servidor com menor taxa de requisições recebidas. A definição do valor de amortecimento é uma abordagem interessante, que poderia ser incluída na nossa proposta, tornando a tomada de decisão sobre a adaptação mais precisa e ajustável. Essa técnica poderia ser implementada em conectores e ser selecionada no nível da configuração.

5.2.4 WebSeAl

Outra proposta que aborda a distribuição de carga em servidores *Web* é [Karaul et al. 1998], que responsabiliza um agente no cliente por direcionar as requisições para os servidores. Esse agente intercepta as requisições, coleta informações dinâmicas sobre os servidores e sobre a rede, e toma decisões com base nessas informações. Decidido qual servidor será utilizado, ele repassa as requisições para esse servidor e coleta suas respostas. Se o servidor selecionado falhar, ele redireciona as requisições para outro servidor. No provedor do serviço há o agente servidor, que intercepta as requisições, as repassa para o servidor *Web* local, aceita suas respostas e as encaminha para o agente cliente adicionando informações de endereçamento se necessário.

Assim como o Cygnus e o *AnyCast*, essa proposta não se preocupa em especificar o nível de qualidade de serviço desejado. Apenas distribui as requisições com base na média dos tempos de resposta, o que poderia ser feito de maneira menos específica à aplicação com a utilização do *framework* CR-RIO e os contratos de QoS.

Capítulo 6

Conclusão

Inicialmente neste trabalho, foi apresentada uma breve revisão de conceitos referentes a arquiteturas de software, contratos de QoS, atividades de monitoração e de auto-adaptação. A partir desses conceitos foi situado o contexto em que nosso trabalho se adequava, onde é possível descrever, desenvolver e gerenciar aplicações baseadas em módulos da arquitetura definidos em uma ADL. Tais aplicações, além dos requisitos funcionais, possuem requisitos não-funcionais, que podem ser descritos através de contratos textuais de nível alto, também chamados contratos de QoS. Os requisitos não-funcionais podem ser constantemente monitorados, de modo que o não cumprimento dos mesmos, pode disparar atividades de adaptação da aplicação ou de sua infra-estrutura, guiada por informações extraídas dos contratos.

Algumas ferramentas de monitoração e descoberta de recursos foram investigadas, visando escolher a que melhor se encaixava às nossas necessidades. Incluímos o resultado da investigação neste documento para uso em trabalhos futuros, tal como o descrito em [Cardoso 2005], que visa desenvolver uma interface padronizada para o acesso às informações obtidas através de diversas ferramentas de monitoração. A fácil aquisição, sua constante atualização e sua consolidação no mundo acadêmico nos levou a optar pela utilização do NWS para efetivar a aquisição de informações sobre os recursos.

A partir do NWS, o sistema de monitoração que mais se adequou às nossas necessidades, implementamos um módulo de seleção de recursos com base em suas medições. Tal módulo facilita a escolha de recursos, tais como um servidor ou um grupo de servidores, que obedeçam a requisitos informados. Ele é de fundamental importância para a escolha de servidores mais adequados para o provimento de um serviço de acordo com os anseios do cliente definidos em seu contrato. Tal sistema também é útil para selecionar quais servidores devem deixar de prover um determinado serviço, visando fazer um uso mais

eficiente dos recursos computacionais disponíveis. Outra utilidade é a escolha do grupo de servidores mais adequado para que o cliente se conecte, de forma a ser servido com a qualidade de serviço desejada.

Visando avaliar a proposta, selecionamos quatro conjuntos de aplicações: *bag of tasks* em paralelo, com distribuição de carga, tolerante a falhas e aplicações *workflow*. Para os dois primeiros conjuntos, foram realizados experimentos avaliando o ganho de desempenho de uma aplicação quando esta possui seus requisitos não-funcionais garantidos. Tal garantia pode ser embutida na aplicação, feita de forma amarrada à implementação específica, ou pode ser realizada por meio de um *framework* concebido especialmente para esse fim, possibilitando o reaproveitamento de código. Para as aplicações tolerantes a falhas e as *workflow* foram mostradas as descrições dos seus requisitos não-funcionais através dos contratos. Além disso, foram esboçados os mapeamentos dos respectivos suportes no *framework*

Nossa contribuição está em validar, através de implementações de exemplos, os conceitos apresentados em [Curty 2002, Ansaloni 2003] a partir do *framework* desenvolvido em [Corradi 2005]. Tais exemplos permitiram verificar as vantagens potenciais do desenvolvimento de sistemas com garantias de qualidade, além de mostrar formas de assegurá-las. A experiência de desenvolver sistemas com base no *framework* nos permitiu obter uma separação de interesses satisfatória, decompondo as preocupações em unidades menores, claramente distintas, em que cada uma delas representa um interesse restrito. Assim, podemos tratar as funcionalidades da aplicação e numa fase posterior, considerar os requisitos não-funcionais. A adição do suporte a esses requisitos pode ser feita de forma incremental sem a necessidade de modificar a aplicação em si, concentrando-se na implementação de *hotspots* específicos. Em suma, tal experiência permitiu avaliar as dificuldades e as facilidades de implementar aplicações com requisitos de QoS utilizando o *framework*.

A análise dos resultados dos experimentos nos mostraram que a garantia dos requisitos de qualidade de serviço provoca uma melhora significativa no desempenho das aplicações selecionadas. É de se esperar que o desenvolvimento de aplicações com o suporte do *framework* CR-RIO facilite a implementação de outras aplicações da mesma classe, possibilitando que elas tenham seu comportamento alterado, como por exemplo mudar o algoritmo de distribuição utilizado, com o ajuste de um ou outro parâmetro no contrato. Isto poderá beneficiar usuários leigos, pois os permite, com um ajuste textual nos contratos, configurar suas aplicações de acordo com suas necessidades específicas.

Um estudo comparativo de outras propostas relacionadas permitiu a consolidação dos conceitos apresentados, contribuiu para o entendimento do contexto e ajudou a identificar questões relevantes que devem ser resolvidas para a consolidação da tecnologia de sistemas adaptáveis.

6.1 Trabalhos futuros

Nesta seção, descrevemos alguns pontos relevantes para a evolução deste trabalho. Em particular, identificamos a necessidade de desenvolver técnicas confiáveis para a tomada de decisões relacionadas aos disparos das adaptações. Em nossos experimentos, usamos uma técnica simples de filtragem das medidas efetuadas. Contudo, ficou evidenciado que em outras situações, soluções mais sofisticadas são requeridas, visando amortecer grandes oscilações nas medidas adquiridas que poderiam interferir nas decisões, tais como as relacionadas ao balanceamento. Também relativo às medidas, deve ser considerado o efeito de margens de tolerância (*tuning*), que definiriam a granularidade das alterações que devem ser explicitadas nas medições dos recursos e repassadas ao nível de gerenciamento do contrato.

Outra evolução é em relação ao acesso às ferramentas de monitoração utilizadas. Uma forma padronizada de buscar as informações coletadas por diversas ferramentas é interessante, pois dará liberdade na escolha de qual ferramenta será utilizada em cada sítio. Isso vem sendo desenvolvido pelo nosso grupo de estudo [Cardoso 2005], como dito anteriormente, e a idéia é integrar esse novo ambiente de suporte ao CR-RIO, permitindo realizar testes mais complexos no gerenciamento de aplicações distribuídas com requisitos não-funcionais por vários sítios.

Uma sugestão no intuito de ampliar as opções de adaptação das arquiteturas, é o uso do ambiente de configuração *Architecture Configurator* - AC [Lisbôa 2003] que poderia prover maior flexibilidade nas medidas adaptativas para a degradação da disponibilidade dos recursos utilizados pela aplicação. Dessa forma é explorada a capacidade de configurar dinamicamente as arquiteturas com a instanciação de conectores adicionando/removendo funcionalidades extras na aplicação, como a adição de um mecanismo de criptografia entre os módulos para aumentar a segurança na comunicação, ou de um esquema de *log* para depuração. Na fase inicial dos nossos experimentos, a versão disponível do ambiente de configuração apresentava alto custo nas chamadas feitas com sua utilização, portanto não foi agregado ao *framework* em nossos testes. Em paralelo ao andamento deste trabalho,

ele foi sendo aperfeiçoado [Santos 2005], o que contribuiu para a redução dos custos das chamadas, porém ainda permaneceu sem o devido suporte a grupos. Com essa nova versão, o integramos parcialmente ao *framework*, e efetuamos testes da viabilidade do seu uso. Um aperfeiçoamento sugerido para o AC é a inclusão de um parâmetro com o nome de grupo, para abranger aplicações em que é necessária a instanciação de conectores responsáveis pela comunicação de grupo.

Outro ponto de sugestão diz respeito aos testes realizados com a atual implementação do *framework* CR-RIO. Embora as aplicações expostas nesse trabalho tenham permitido avaliar os ganhos de desempenho, reaproveitabilidade de código e separação de interesses, seria interessante efetuar testes com aplicações em cenários mais diversos, como aplicações de videoconferência [Corradi 2005], e até mesmo aplicações no contexto da computação pervasiva. Para contemplar a alta dinamicidade das aplicações pervasivas, estudos vêm sendo realizados por nosso grupo de pesquisa visando propor extensões do modelo do *framework* para agregar capacidades como a representação de grupos dinâmicos de recursos e estabelecimento de métricas para seleção dos melhores segundo o gosto do usuário. Além disso, o *framework* deve ir além, permitindo a especificação de contexto, ou seja, do conjunto de recursos em um determinado ambiente (por exemplo, onde o usuário está). Para possibilitar a seleção do recurso mais viável para ser utilizado nesse meio com constantes mudanças, torna-se necessário o desenvolvimento de módulos de descoberta de recursos que se baseam em funções utilidade e em predições (com confiabilidade) do nível do estado dos recursos.

APÊNDICE A – Monitoração

A.1 Introdução

Esse apêndice tem como objetivo mostrar algumas características de algumas ferramentas que têm como objetivo prover instrumentação para sistemas. Para simplificar a comparação entre algumas propostas de ferramentas de monitoração, que em geral utilizam nomenclaturas diferentes para referir a conceitos semelhantes (quando não iguais), que serão apresentadas mais a frente nessa seção, convencionamos alguns termos descritos em [Tierney et al. 2001, Plale et al. 2002].

- **Entidade**: é qualquer recurso que pode ser considerado útil, é único, possui um tempo de vida e um uso. Em geral as entidades são processadores, memórias, formas de armazenamento, enlaces de rede, aplicações e processos.
- **Um evento**: é uma coleção de *timestamps* e dados, está associado a uma entidade e é representado em uma estrutura específica.
- **Um tipo de evento**: é um identificador que de forma única mapeia para uma estrutura de um evento.
- **Um esquema de um evento**: define a estrutura e a semântica de todos os eventos, de forma que, dado um tipo de evento, pode ser encontrada a estrutura e interpretada a semântica desse evento.
- **Um sensor**: é um processo que monitora uma entidade e gera eventos. Eles são divididos em passivos, que utilizam medições disponíveis normalmente pelo sistema operacional (costumam prover medidas específicas do SO), e ativos, que fazem estimativas usando *benchmarks* (são mais intrusivos).

O monitoramento em sistemas distribuídos, em geral, inclui quatro estágios: (i) a geração dos eventos, que envolve aquisição e codificação das medidas feita pelos sensores,

de acordo com um esquema. O resultado deste estágio são *traces* que retratam o histórico da atividade do sistema; (ii) o processamento dos eventos gerados, que pode acontecer durante qualquer estágio do processo de monitoração, geralmente incluem filtros, de acordo com algum critério, e agrupamento de eventos (e.g. cálculo de média); (iii) a distribuição, que se refere à transmissão dos eventos de sua origem para as entidades interessadas; (iv) a apresentação, que envolve a exibição da informação, coletada e processada, para o usuário, para que ele possa concluir algo sobre o sistema monitorado [Mansouri-Samani 1992]. Normalmente, a apresentação é fornecida por uma aplicação GUI (*Graphical User Interface*) utilizando de técnicas de visualização, podendo também, ser utilizado um *stream* de eventos ou simplesmente um *trace*. Contudo, no contexto de aplicações com requisitos não-funcionais a serem cumpridos, analisamos o estágio da apresentação como um consumo de informação, uma vez que os usuários das informações monitoradas não são humanos (são aplicações), não envolvendo, portanto, visualizações.

Em [Tierney et al. 2001] são mostrados (ver fig. A.1) os componentes essenciais para o provimento da monitoração (*Grid Monitoring Architecture - GMA*) em *grids*, apresentados pelo *Global Grid Forum* - GGF

- **Um produtor:** que é um processo que implementa pelo menos uma API (*Application Programming Interface*) para o provimento dos eventos (*performance event source*).
- **Um consumidor:** que é qualquer processo que recebe os eventos usando uma implementação de alguma API consumidora (*performance event sink*).
- **Um serviço de diretório ou registro:** é um serviço de busca que permite aos produtores publicar os tipos de eventos gerados, e aos consumidores, encontrar os eventos nos quais possuem interesse. Esse registro, também possue os detalhes necessários para o estabelecimento da comunicação com as entidades registradas (e.g. endereçamento, protocolos, requisitos de segurança). Mesmo em sistemas sem a noção de eventos, os registro podem ser úteis para que produtores e consumidores se descubram.
- **Um republicador:** além dos 3 componentes centrais supracitados, a GMA define um componente intermediário, que seria uma espécie de republicador, que implementa as interfaces do produtor e do consumidor, porém este é considerado uma entidade opcional.

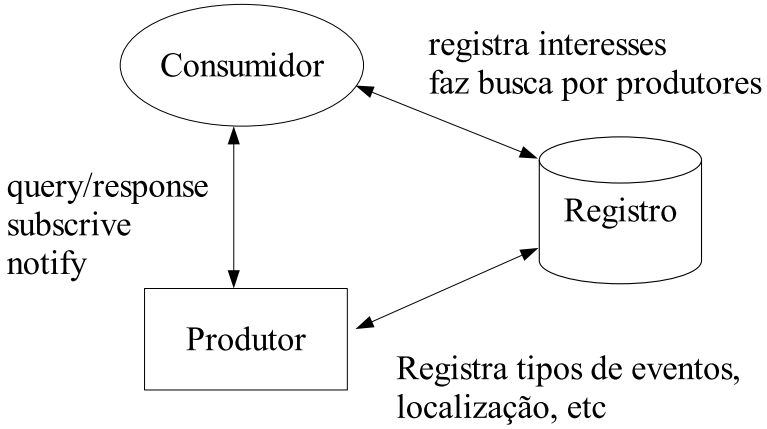

Figura A.1: *Componentes da Arquitetura de Monitoração em Grid - GMA.*

Após se descobrirem através da consulta ao registro, produtores e consumidores se comunicam diretamente. A GMA define três tipos de interação entre eles: o *publish/subscribe*, que se refere à interação em 3 fases: a demonstração do interesse por determinados tipos de evento, o envio destes eventos pelo produtor, e a finalização do interesse; o *query/response*, em que o consumidor requisita e o produtor responde com um ou mais eventos; e a *notification*, em que o servidor pode enviar para o consumidor, sem receber suas requisições.

Abordaremos a seguir algumas ferramentas de monitoração que se encaixam no propósito deste trabalho. Incluímos um resumo deste estudo das ferramentas neste trabalho para consulta de trabalhos futuros, como [Cardoso 2005], onde há a proposta de integrar as informações obtidas através de diversas ferramentas de monitoração e acessá-las através de uma interface padronizada.

A.2 Ferramentas de monitoração

A.2.1 NWS - *Network Weather System*

O NWS é um sistema distribuído implementado para coletar informações sobre os recursos e produzir estimativas de desempenho de curto prazo, com base nos dados das medições de desempenho feitas anteriormente. O objetivo do sistema é caracterizar e prever dinamicamente o desempenho que poderá ser fornecido no nível de aplicação, a partir de um conjunto de recursos computacionais e de rede. Tais estimativas de desempenho têm sido úteis, por exemplo, para implementar agentes de escalonamento dinâmico para aplicações de meta-computação [Wolski et al. 1999].

Segundo seus projetistas, o NWS é projetado para maximizar quatro características possivelmente conflitantes, que devem ser atingidas apesar de o ambiente de execução e a infra-estrutura provida pelos sistemas de meta-computação compartilhados serem altamente dinâmicos:

- **Rapidez e precisão**: O NWS deve ser capaz de prover rapidamente estimativas precisas de desempenho futuro de um recurso.
- **Execução leve**: O sistema deve sobrecarregar o mínimo possível os recursos que monitora. A execução das previsões deverá ser o mais leve possível sem interferências perceptíveis.
- **Persistência da execução**: Para ser efetivo, o NWS deve estar disponível a todo tempo, como um serviço geral do sistema.
- **Ubiqüidade**: Como um serviço do sistema, o NWS deve estar disponível para todos os lugares possíveis para execução da aplicação, dado um conjunto de recursos. Similarmente, ele deve ser capaz de monitorar e prever o desempenho de todos os recursos disponíveis.

A.2.1.1 Arquitetura

O NWS foi construído utilizando-se quatro tipos diferentes de processos componentes. Cada um desses processos pode comunicar-se com outros processos através da troca de mensagens estritamente tipadas. São eles (figura A.2):

- **Mecanismo de persistência (*Persistent State*)**: armazena e recupera medições utilizando meios persistentes.
- **Servidor de nomes (*Name Server*)**: implementa um serviço de nomes utilizado para relacionar nomes de processos e dados com a informação de contato de baixo nível (e.g. endereços, portas, etc.).
- **Sensor (*Sensor*)**: realiza medições de desempenho em um determinado recurso.
- **Previsor de estado (*Forecaster*)**: produz um valor previsto para o desempenho disponível durante um determinado espaço de tempo para um determinado recurso.

O servidor de nomes é executado em somente uma máquina do sistema. Os sensores monitoram os recursos e enviam as medidas para os gerenciadores de persistência de

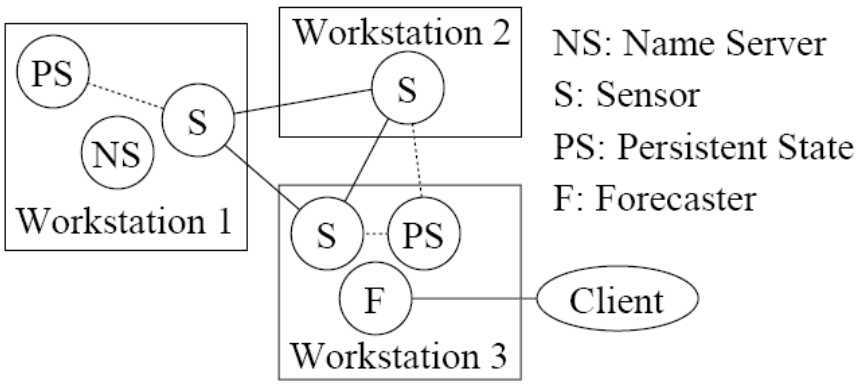

Figura A.2: *Arquitetura geral do NWS [Wolski et al. 1999].*

estado. O previsor atua como um *proxy* para os usuários do sistema agindo como um intermediário no acesso às medidas coletadas e às previsões.

A.2.1.2 Implementação

Os detalhes dos componentes que compõe a arquitetura do NWS são descritos a seguir:

- Gerenciamento de persistência de estado

Os processos de persistência de estado provêem um serviço de armazenamento e recuperação de *strings* de texto que representam os dados de cada processo de monitoração associados a um *timestamp*. Cada dado enviado para um processo de persistência de estado é imediatamente gravado em disco, e depois uma confirmação (ACK) é devolvida. Cada arquivo de dados é tratado como uma fila circular (de tamanho configurável), visto que os dados das previsões feitas vão perdendo sua utilidade com o passar do tempo, portanto os dados não são guardados por tempo indefinido.

- Gerenciamento de nomes

Um serviço de nomes, primitivo mas funcional, também é mantido pelo NWS, capaz de gerenciar ligações entre nomes de recursos e sua localização física. Seu endereço é único e deve ser conhecido por todo o sistema, permitindo a distribuição dos dados e serviços. Todos os outros processos do NWS registram seu nome e sua localização junto a esse servidor de nomes. Essa ligação expira de acordo com uma especificação de tempo de vida (*time-to-live*) que deve acompanhar o registro; sendo assim, processos ativos devem registrar-se periodicamente com mensagens do tipo "*I'm alive*", denominadas *heartbeats*.

- Monitoramento

As medições de desempenho, feitas pelos processos sensores em cada recurso, geralmente apresentam uma tensão entre o quão precisa será a medida realizada e o quão intrusivo será o processo de medição em relação ao desempenho do recurso. O NWS tenta utilizar de maneira equilibrada utilitários de monitoração e a ocupação ativa de recursos para realizar a medição de desempenho.

A implementação atual do NWS suporta medições de desempenho relacionadas a utilização de processador, a características de rede (tempo de conexão TCP, latência e largura de banda fim-a-fim) e a armazenamento (primário e secundário). Cada Sensor faz a coleta do par *timestamp* e medida de desempenho para a característica monitorada, sem se preocupar em correlacioná-la com outras.

O sensor é responsável por coletar as informações dos recursos e enviá-las ao processo de persistência de estado, que é consultado pelo previsor para a aplicação de técnicas numéricas sobre essas informações, com a finalidade de realizar as previsões. Os tipos de sensores disponíveis e o elemento de predição são descritos a seguir:

Sensor de CPU: Esse sensor combina informações geradas a partir de dois utilitários do UNIX (*uptime* e *vmstat*) fazendo medidas periódicas relativas à carga da CPU. Geralmente, o *uptime* reporta a média de uso de processamento nos últimos um, cinco e quinze minutos, enquanto que o *vmstat* reporta a porcentagem de uso de CPU por processos de usuário ou sistema, e o uso de memória. O sensor usará a medida do último minuto para calcular a fração de tempo de CPU que um processo teria caso ele fosse executado no momento em que o *uptime* foi iniciado. A partir da saída do *vmstat* o sensor usa uma combinação das medidas de tempo ocioso, tempo de usuário e tempo de sistema para gerar a estimativa da fração de uso da CPU.

Sensor de rede: Os sensores de rede no NWS realizam consultas ativas na rede. Em intervalos regulares (configuráveis), cada sensor de rede se conecta a outros sensores, executados em máquinas de interesse, e conduz uma ou mais consultas a fim de obter os valores de suas medidas. Os sensores de rede do NWS são capazes de fazer três tipos de medidas de desempenho: RTT (*Round Time Trip*) para mensagens pequenas, *throughtput* de mensagens grandes e tempo de conexão-desconexão de um *socket* TCP. A consulta com mensagens pequenas consiste na transferência de um pacote TCP de 4 bytes onde é marcado o tempo de ida e volta dessa mensagem. A consulta de *throughtput* usando mensagens grandes é usada para medir a largura de banda disponível no nível da aplicação e utiliza a transferência de uma mensagem usando TCP e a confirmação do seu recebimento.

De forma a não introduzir muita carga na rede e nos processadores que executarão as provas, os sensores são organizados de forma hierárquica, de modo a permitir que uma medição de fim-a-fim possa ser feita sobre um subconjunto representativo da totalidade de sensores.

É possível que, em um grupo de sensores, as requisições de coletas diferentes colidam entre si, o que afetaria as medições realizadas. Para evitar isso e fornecer uma maneira escalável de gerar medições de-todos-para-todos, a rede de sensores é organizada em estruturas denominadas *cliques*. Cada sensor participante de um *clique* conduz experimentações entre ele e cada outro membro do seu *clique*, mas nunca com sensores externos a ele. Sensores podem participar de vários *cliques* em níveis diferentes, de modo a tornar possível uma organização hierárquica, definindo-se diferentes *cliques* para cada nível de hierarquia e promovendo um sensor representante de cada *clique* para participar de um *clique* do próximo nível mais alto.

As medições dentro de um *clique* são gerenciadas por um único sensor (o líder), que conduz um protocolo de passagem de *token* (*token passing*) para realizar medidas entre cada par de sensores presentes. Com os devidos cuidados tomados para a conclusão de uma rodada de medições dentro de cada *clique* (por exemplo, checagem de falhas em sensores, verificação de perda de *token*, etc), cada líder comunica aos outros líderes, que formam um *clique* de nível superior, os resultados de medição referentes ao seu *clique*, que são agregados e repassados para o nível acima, até que o *clique* de último nível seja atingido, que passará as informações coletadas para os processos de persistência de estado para o armazenamento das medidas

Em [Gaidioz et al. 2000] é apresentado um protocolo escalável para medições ativas de desempenho na rede, de forma a minimizar as colisões entre as requisições do monitoramento.

- Previsores

Esse componente fornece previsões sobre o desempenho dos recursos. Para isso ele consulta o histórico de medições mantido pelo componente de persistência. Ordenado a partir de sua data de criação, esse histórico contém um conjunto de medidas com marcas de tempo (*timestamps*). O previsor aplica um conjunto de técnicas numéricas de previsão sobre as medidas e, dinamicamente, escolhe a técnica que provê maior precisão sobre o conjunto mais recente de medidas. A escolha é baseada no erro acumulativo associado a cada técnica de previsão.

Para cada método (ou modelo) de estimativa é mantido um histórico de sua atividade prévia e informação sobre a precisão das estimativas feitas anteriormente. O NWS utiliza os vários modelos de estimativa configurados no sistema de maneira concorrente. A cada previsão solicitada, cada método trabalha sobre o seu histórico de medições e erros, gerando assim uma estimativa para aquele instante de tempo. O sistema então coleta, dentre os resultados obtidos, aquele que oferece o menor erro residual, o que reflete numa maior precisão para aquele instante de tempo. A cada previsão, o NWS recalcula o *ranking* de precisão entre métodos, mantendo o histórico de desempenho sempre atualizado.

Essa metodologia de escolha entre métodos que competem entre si torna possível, por exemplo, que um determinado método escolhido em um dado instante de tempo seja preterido por outro método em outro instante, desde que este último gere um resultado mais preciso, ou com menos erro residual acumulado para aquele instante. Assim, o sistema de previsão é bem dinâmico, e torna-se não-paramétrico, visto que cada modelo de estimativa possui sua parametrização bem específica.

Existem algumas classes principais de métodos de estimativa de desempenho baseado em dados históricos que são usados pelo NWS. Cada classe se basea em cálculos de valores estatísticos sobre a série representada por esses dados. As principais são:

- Métodos baseados na média aritmética:

  -*RUN_AVG* - média de todos os valores do histórico;

  -*SW_AVG* - média de valores tomados do histórico em subconjuntos distintos, formando um esquema de "janela deslizante";

  -*ADAPT_AVG* - uma adaptação do SW_AVG com ajustes feitos no intervalo tomado na direção do menor erro residual obtido;

  -*GRAD* - gradiente estocástico, um método recursivo que tem sua precisão controlada por um fator de ganho.

- Métodos baseados na mediana:

  -*MEDIAN* - utiliza a mediana dos valores contidos no histórico;

  -*ADAPT_MED* - uma adaptação do *MEDIAN* para escolha do intervalo que oferece o menor erro residual;

  -*TRIM_MEAN* - utiliza um filtro que elimina valores distantes de um determinado valor médio, evitando-se ruídos nos valores obtidos.

- Modelos auto-regressivos baseados em soluções de sistemas não-lineares que utilizam recursividade.

A partir da análise dos dados obtidos em cada modelo de estimativa, é possível chegar a um modelo que se adapta melhor à previsão de desempenho para um determinado recurso monitorado.

A interação do NWS com aplicações desenvolvidas para plataformas de meta-computação (e.g. Legion, Globus, Condor, MPI e PVM) é feita através de comunicação via *sockets* TCP para um processo previsor, que é o representante (*proxy*) do sistema NWS para a aplicação. Também é oferecido um acesso interativo através de aplicações CGI, que possibilitam a construção de vários modelos gráficos para análise e possibilidade de escalonamento.

A.2.1.3 Considerações

No NWS, os sensores se comportam como produtores (medem e disseminam os eventos) na visão interna de um *clique*. Já para o conjunto de *cliques*, os sensores líderes agem como republicadores, assim como o mecanismo de persistência de estado e o de previsão. Para provimento de replicação, os serviços de armazenamento podem ser arranjados de forma a contemplá-la.

Em [den Burger et al. 2002] há uma extensão do NWS para provimento de informação sobre a topologia da rede, que pode ser usada para cálculo da árvore geradora mínima[1] (*minimum spanning tree*) entre os *hosts* em termos de latência ou banda passante. Também como parte dessa extensão, há um republicador operando em cima dos processos de armazenamento persistente e do servidor de nomes, que emprega o protocolo XML produtor-consumidor definido pelo GGF.

A.2.2 Remos - *REsource MOnitoring System*

O Remos foi desenvolvido com o propósito de disponibilizar informações sobre os recursos para aplicações distribuídas. É capaz de capturar informações de diferentes tipos de rede e das máquinas pertencentes a elas, através do uso de vários coletores que usam diferentes tecnologias como SNMP (*Simple Network Management Protocol*) e *benchmarking* [Dinda et al. 2001]. Com a utilização de coletores apropriados para cada tipo de rede e

[1]Dado um grafo, a árvore geradora mínima será aquela que possui ligação para todos os nós, cujos pesos são mínimos. Isso é feito retirando as ligações redundantes de maior peso (e.g. latência).

provendo uma arquitetura para distribuir a saída desses coletores em um ambiente distribuído, o Remos coleta informações detalhadas de cada localização e as distribui para todos os requisitantes de maneira escalável.

A.2.2.1 Arquitetura

A arquitetura do Remos divide os serviços necessários entre os coletores (*collectors*), os modeladores (*modelers*), e os previsores (*predictors*), como pode ser visto na figura A.3. A API disponibilizada para o usuário é implementada no modelador. Detalhes desses componentes são mostrados a seguir.

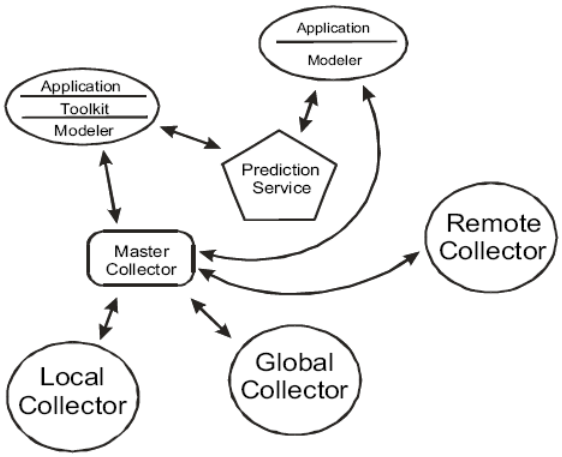

Figura A.3: *Arquitetura do Remos [Dinda et al. 2001].*

- Coletores (*Collectors*)

Os coletores são responsáveis pela aquisição e consolidação das informações necessárias da aplicação. Eles possuem uma variedade de métodos para coletar informações, e podem, por exemplo, agregar ou controlar sensores que fazem as medidas de fato, mas do ponto de vista arquitetural sua única função é coletar informações e fornecê-las para o modelador. Os coletores podem ser organizados de forma hierárquica para se obter uma alta escalabilidade. Os coletores locais, que ficam em um nível mais baixo, são responsáveis por obter informações de desempenho da sua rede específica, enquanto o global coleta de redes que estejam conectando outras LANs. Tanto os locais quanto os globais, em redes remotas, podem ser contatados por outros coletores disponibilizando informações sobre sua rede.

O coletor mestre (*master collector*) é responsável por recolher informações dos diferentes coletores e aglutiná-las em uma resposta para uma consulta feita pelo modelador. Ele possui uma base de dados com informações da localização dos outros coletores e a

porção da rede pelo qual são reponsáveis. Quando o modelador faz uma requisição, o coletor mestre consulta o coletor apropriado e reponde sem revelar que a resposta foi obtida de vários coletores, permitindo a construção de várias camadas de coletores. Como por exemplo na figura A.3, o coletor remoto poderia ser um coletor mestre que entraria em contato com vários coletores locais quando consultado.

Uma vantagem importante dessa arquitetura é a trasparência deixada para o modelador e para a aplicação sobre a origem das medidas, porque o coletor assume a responsabilidade de contatar lugares remotos e reunir toda informação disponível em uma única resposta. O modelador e a aplicação não precisam definir se a consulta interessa a nós locais ou remotos nem qual é a técnica de mensuração mais apropriada.

- Modeladores (*Modelers*)

O modelador provê a API Remos para as aplicações e se comunica com o coletor para obter as informações necessárias para responder às consultas entregues a ele. Uma vez que os coletores foram desenvolvidos somente para coletar informações sobre os recursos da rede, os modeladores são responsáveis pelo processamento necessário para converter essas informações em um formato apropriado para as aplicações. Como por exemplo, quando a aplicação requisita a largura de banda disponível entre dois pontos, os coletores enviam várias medidas separadas, mas o modelador reportará somente a largura de banda do gargalo desse caminho. Caso alguma predição seja necessária, o modelador também agirá como intermediário entre os coletores e o serviço de predição.

- Previsores (*Predictors*)

O previsor, de maneira similar ao *Forecaster* do NWS, tenta prever comportamentos futuros através do uso de medidas históricas armazenadas em sua base de dados. A figura A.4 mostra como o serviço de predição (RPS), baseado em uma arquitetura cliente-servidor, se encaixa na arquitetura do Remos. A opção por um serviço de previsão de alto nível foi adotada por várias razões. A primeira delas é que o Remos coleta as informações de coletores básicos, e depois as reúne em uma única resposta, dificultando a realização de predições em um nível mais baixo. Outro motivo é a possibilidade de haver relação entre os comportamentos de diferentes recursos. Logo, tentar predizer o comportamento independentemente, antes de agregar os resultados, poderia resultar em perda de informação. O último motivo é a existência da possibilidade de se especificar diferentes parâmetros para as predições, como a quantidade de dados históricos a ser utilizada, a granularidade de tempo ou o tempo no futuro em que a previsão é necessária; isso impediria fazer previsões em camadas mais baixas, sem se ter prévio conhecimento

da requisição da aplicação.

A.2.2.2 Implementação

A figura A.4 mostra vários tipos de coletores usados pelo Remos interagindo sobre diferentes redes, com domínios administrativos distintos. Para o usuário a complexidade do sistema é escondida pela divisão do Remos em duas partes. O lado dos componentes da aplicação, composto por modeladores e previsores; e o lado dos componentes da rede, composto pelos coletores. De acordo com a forma como coletam os dados, os coletores podem ser classificados em: Coletores SNMP, *Bridge* e *Benchmark*, mostrados a seguir.

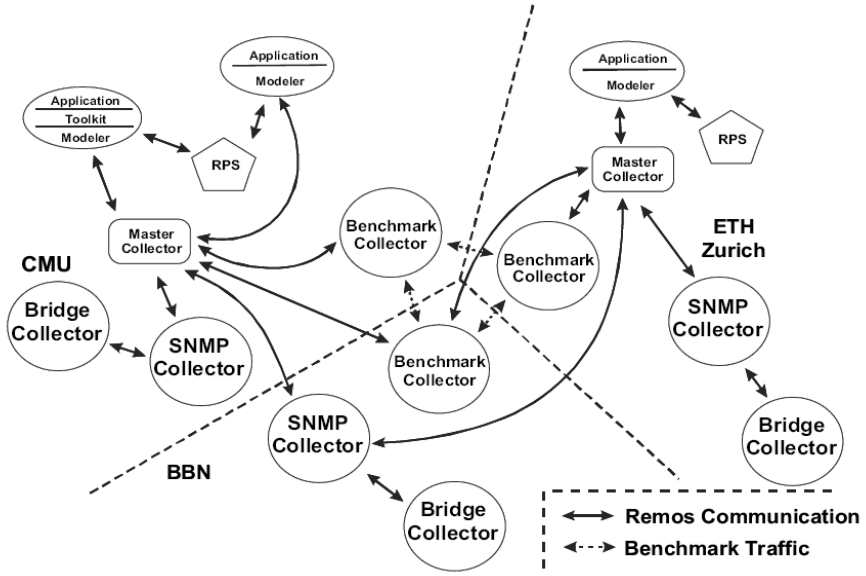

Figura A.4: *Interação entre os componentes da arquitetura Remos. As aplicações executam nos dois domínios superiores, utilizando recursos disponíveis dos três domínios administrativos. RPS é o toolkit [Dinda e O'Hallaron 1999] que engloba as medições nos hosts e gera previsões.*

- Coletor SNMP

A maioria das informações de rede disponibilizadas pelo Remos está baseada no coletor SNMP (*Simple Network Management Protocol*). O SNMP é um protocolo de base de dados para prover acesso direto e controle sobre a situação dos dispositivos de rede. O coletor usa esse protocolo para fazer consultas passivas que obtêm informações sobre topologia e desempenho de rede diretamente dos roteadores e *switches*. Sua operação ocorre em redes roteadas (camada 3), e está associado a uma rede particular, geralmente é responsável pela coleta de informações de um domínio IP de uma universidade ou departamento. Como os agentes SNMP são normalmente acessíveis apenas de endereços IP locais, os coletores SNMP possuem localização e áreas de responsabilidade bem definidos.

Inicialmente, a monitoração do coletor SNMP é feita sob-demanda, ele aguarda a consulta da aplicação e então começa a monitorar os componentes da rede para responder à consulta. Depois que a monitoração de partes da rede é iniciada, ele continuará monitorando periodicamente a fim de manter dados históricos da rede para serem usados nas previsões. É possível também a configuração para que ele comece a monitorar recursos específicos a partir de sua inicialização.

O primeiro e mais complexo passo a ser realizado pelo coletor SNMP ao receber uma consulta é a descoberta de topologia. Usando os endereços IP dos nós da consulta e os roteadores que eles estão configurados a usar, o coletor segue a rota salto por salto entre cada par de nós da consulta. Apesar de simples, o algoritmo é custoso, seu tempo de execução é de D x N^2, onde D é o diâmetro da rede e N é o número de *endpoints* envolvidos na requisição. Entretanto, o algoritmo finaliza a pesquisa pelo caminho entre o par de nós, quando um caminho previamente descoberto para o mesmo destino é alcançado.

Depois de descobertas as rotas entre os nós, o coletor consulta a banda livre entre os pares de roteadores ao longo do caminho. Sua utilização é monitorada a cada 5 segundos por padrão, entretanto esse parâmetro pode ser configurado. Finalmente, o coletor SNMP representa a rede como um grafo da sua topologia virtual. Virtual porque o grafo pode não mapear perfeitamente a rede. A implementação do coletor SNMP é feita utilizando *threads* Java sendo capaz de monitorar vários roteadores e responder a várias consultas simultaneamente.

- Coletor *Bridge*

O coletor *Bridge* apenas monitora redes roteadas no nível 3. Apesar de muitas redes de pesquisa serem conectadas usando apenas roteadores, a maioria das redes locais estão implementadas usando *switches* ethernet que operam no nível 2, e estes não provêem informações claras da topologia como as fornecidas pelas tabelas de roteamento IP. Para resolver esse problema, o coletor *Bridge* faz uma consulta ao banco de dados de encaminhamento da Bridge-MIB de cada *Bridge* ou *switches* e determina a topologia da rede.

Quando o coletor SNMP recebe uma consulta do modelador por *hosts* desconhecidos, ele a encaminha para o coletor *Bridge*. Este por sua vez, responde com as *Bridge* e conexões utilizadas na topologia pelos *hosts* da consulta. O coletor SNMP armazena estas informações de *hosts* e *Bridges* e as adiciona a uma lista de conexões monitoradas.

Após a fase de descoberta, o coletor *Bridge* passa a monitorar os *hosts* conhecidos

na rede local. É definido um intervalo de monitoração para cada *host* com base em sua informação histórica indicando a probabilidade deste se mover ou deixar a rede. Caso haja alguma alteração na localização dos *hosts*, uma mensagem de invalidação é enviada para o coletor SNMP. Se este possuir informações armazenadas sobre o referido *host*, elas são removidas e uma nova consulta pode ser feita, caso contrário, a mensagem é ignorada.

- Coletor *benchmark*

Ferramentas como o Remos normalmente não podem coletar dados SNMP através de redes metropolitanas ou redes com domínios administrativos diferentes. Nesse caso, é usado o coletor *Benchmark* que realiza testes ativos para determinar características de desempenho da rede. Um coletor *Benchmark* é executado em cada rede que tenha um coletor SNMP. Quando é requisitada uma medida de desempenho entre múltiplas redes, os coletores *Benchmarks* das redes de interesse trocam dados entre si. Ao medir a taxa com que os dados atravessam a rede ele será capaz de determinar o desempenho dos enlaces e informar ao coletor mestre.

Para determinar a banda disponível e atraso são utilizados o *Netteste* e o *traceroute*, respectivamente, e a freqüência de captação de informações é ajustável.

- Coletor mestre

As aplicações distribuídas não são capazes de obter todas as informações necessárias diretamente dos coletores *Benchmark* e SNMP, pois estes apenas monitoram uma porção particular da rede. Para solucionar esse problema foi desenvolvido o coletor mestre.

Os coletores se registram em uma base de dados, que é usada pelo coletor mestre, passando informações como o tipo de coletor e o domínio de sua responsabilidade, representado por endereços de rede e máscara. Quando o modelador consulta ao seu coletor mestre, este identifica, através do endereçamento IP, as redes e sub-redes necessárias para responder sua consulta, assim como os coletores SNMP e *Benchmarks* dessas redes. Após este passo, o coletor mestre divide a consulta e repassa essas sub-consultas aos coletores responsáveis por essas redes. Quando as respostas são recebidas desses coletores, o coletor mestre as combina em uma única resposta e retorna essa resposta ao modelador que a requisitou.

- Modelador

O modelador é a entidade que fornece a API que pode ser ligada à aplicação para que o usuário possa fazer suas requisições. Ele mantém uma conexão TCP com o coletor mestre, se comunicando com este via *sockets* usando um protocolo simples baseado em ASCII

(*American Standard Code for Information Interchange*). A finalidade desta conexão é obter as informações necessárias para responder às consultas. Ao receber uma resposta do coletor, ele consolida as informações retornadas gerando uma topologia lógica, com informações estáticas e dinâmicas associadas a cada elemento da rede. Para responder às consultas sobre fluxos de dados, o modelador também realiza cálculos a partir da topologia virtual gerada.

- Previsor

Para realizar previsões o modelador utiliza o *ToolKit* RPS [Dinda e O'Hallaron 1999]. A relação entre o RPS e o Remos é relativamente complexa por serem sistemas independentes. No contexto do Remos o RPS provê os serviços de previsão e medição dos *hosts*, enquanto o Remos provê serviços de medição dos recursos de rede para o RPS. Na implementação atual o próprio RPS coleta os dados necessários para montar o histórico usado pelas previsões. Ele faz isso através de um sensor de carga na máquina e um de banda (que na realidade é uma aplicação Remos).

A.2.2.3 Considerações

O Remos cria uma interface uniforme para as aplicações acessarem informações sobre diferentes tipos de rede. Ele permite que as aplicações descubram em tempo de execução, informações relativas a seu ambiente de execução. As aplicações podem requisitar informações sobre a topologia de rede e descobrir informações como largura de banda disponível entre dois pontos.

O coletor mestre coordena os coletores para a execução de uma determinada requisição, junta os resultados e os envia para o requisitante, ou seja, age como republicador, assim como os coletores *bridge* e *SNMP* e os previsores. Já os modeladores, presente em todas as aplicações funcionam como um consumidor, enquanto os *benchmarks* e os produtores SNMP correspondem aos produtores da GMA.

A.2.3 Ganglia

O Ganglia é um sistema de monitoração proposto para ambientes de alto desempenho como *clusters* e *grids*, que utiliza tecnologias como XML (*Extensible Markup Language*) para representação de dados, XDR (*External Data Representation standard*) para transporte dos dados de forma compacta e RRDtool (*Round Robin Database Tool*) para armazenamento e vizualização de dados [Massie et al. 2004]. Sua implementação se preocupa em

minimizar *overheads* em cada um dos pontos de monitoração e maximizar a colaboração entre eles, e é suportada por vários sistemas operacionais e arquiteturas de processadores.

A.2.3.1 Arquitetura

O Ganglia é organizado de forma hierárquica como uma federação de *clusters*, como mostrado na figura A.5. Um protocolo com base em *multicast* (*listen/announce*) é usado, em que os nós divulgam suas informações e ficam escutando pelas informações dos outros. Uma árvore de conexões ponto-a-ponto é montada dentro de cada *cluster* para controlar os *clusters* filhos e agregar seus estados. Internamente a cada *cluster*, o Ganglia usa mensagens do tipo *"i'm alive"*, nomeadas *heartbeat* (como no NWS), para endereços *multicast* bem conhecidos, servindo de base para o serviço de *membership*. O recebimento dessas mensagens indica que o nó está disponível, enquanto que o não recebimento dentro de um intervalo de tempo é um sinal de indisponibilidade do nó.

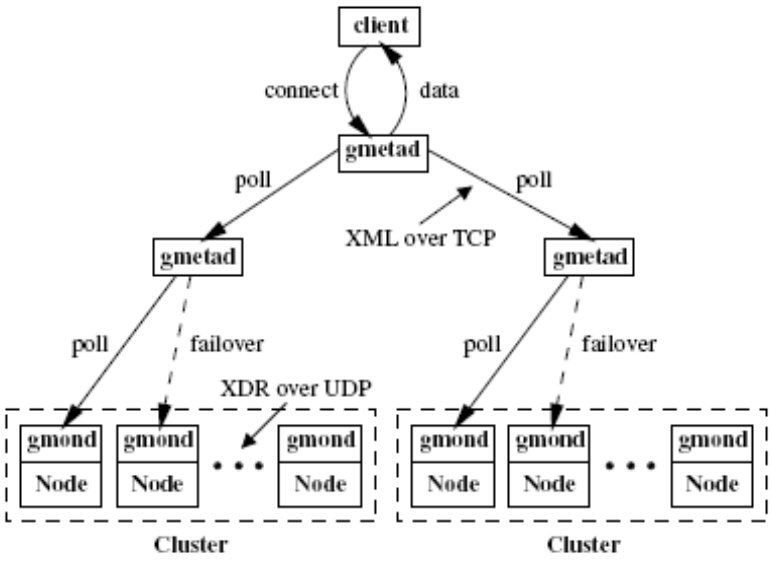

Figura A.5: *Arquitetura do Ganglia [Massie et al. 2004].*

Cada nó monitora seus recursos locais e envia pacotes *multicast* contendo o dado monitorado sempre que uma mudança significativa dos valores ocorre. Métricas específicas da aplicação também podem ser enviadas para o mesmo endereço *multicast* como forma de monitorá-la. O ganglia faz distinção entre as métricas pré-definidas (*built-in*) e as definidas pelo usuário através de um campo nos pacotes de monitoração. O primeiro tipo é formado pelas métricas coletadas pelo próprio sistema, como por exemplo porcentagem de utilização de CPU, carga média, uso de memória, rede, entre outros. Já as métricas definidas pelo usuário são informações arbitrárias coletadas por programas externos

e incorporadas ao Ganglia através de uma interface específica. Esta característica faz do Ganglia uma ferramenta extensível, que pode ser utilizada para monitorar qualquer informação coletável.

Todos os nós escutam os dois tipos de métricas em endereços *multicast* bem conhecidos, e coletam e mantêm dados de monitoração de todos os outros nós. Então, todos os nós possuem sempre uma visão aproximada do estado completo do *cluster* e seu estado pode ser facilmente reconstruído após a falha de algum nó.

O Ganglia é capaz de agregar informações de múltiplos *clusters* montando uma árvore de conexões ponto-a-ponto entre eles. Cada nó não-folha da árvore irá periodicamente requisitar informações de monitoração de seus filhos para depois agregar e repassar esses dados para o nível superior de sua árvore. Cada nó filho retorna as informações de monitoração de todos os outros nós pertencentes ao seu *cluster* local, agindo como um representante local. Esse representante do *cluster* pode ser qualquer nó, pois como dito anteriormente, todos eles possuem as informações de monitoração de todos os outros. Cada pai escolhe um representante de um determinado *cluster* filho, e em caso de falha há outro reserva (figura A.7), dessa forma ele é capaz de continuar obtendo informações de monitoração do seu *cluster* filho.

A.2.3.2 Implementação

A implementação consiste em dois *daemons* (o *gmond* e o *gmetad*), um programa de linha de comando (*gmetric*) e uma biblioteca no lado do cliente. O *gmond* (que executa em todos os nós de um *cluster*) provê a monitoração em um único *cluster* implementando o protocolo *listen/announce* e respondendo às requisições dos clientes com o retorno de um arquivo XML representando o dado da sua monitoração. O *Ganglia Meta Daemon* (*gmetad*), por sua vez, faz o controle de múltiplos *clusters*. As ligações de conexões TCP entre vários *daemons gmetad* permitem agregar informações de monitoração de múltiplos *clusters*. E por fim, o *gmetric* é o programa que as aplicações podem utilizar para publicar métricas específicas do usuário, enquanto a biblioteca no lado do cliente fornece acesso às caracterizações do Ganglia.

- Monitoração em um único *cluster*

A monitoração em um *cluster* é implementada pelo *gmond*, que é organizado como uma coleção de *threads*, cada uma associada a uma tarefa específica. Há a *thread* responsável por coletar a informação no nó local, publicá-la no canal *multicast* e enviar mensagens

heartbeats periodicamente, são as chamadas *collect and publish threads*. Já as *listening threads* são responsáveis por escutar no canal *multicast*, os dados monitorados de outros nós e atualizar o sistema de armazenamento do *gmond*, uma tabela *hash* hierárquica de métricas de monitoração. E por último, há as *XML export threads*, dedicadas para aceitar e processar requisições dos clientes por dados de monitoração. Todos os dados armazenados pelo *gmond* são *soft state*, ou seja, os membros devem mandar mensagens periodicamente, caso contrário suas monitorações expiram. E através do *multicast* feito por todos os nós informando o seu estado, um novo *gmond* pode passar a existir simplesmente ouvindo e anunciando mensagens de monitoração.

Os dados monitorados pelo *gmond* são armazenados em uma tabela *hash* desenvolvida de forma a prover uma alta concorrência em seu acesso. Isso permite que as *listening threads* possam armazenar simultaneamente dados de múltiplos *hosts* e ajuda a resolver a alta competitividade de acesso aos registros de métricas (*inMemory Storage*) entre as *listening threads* e os *XML export threads* como visto pela figura A.6.

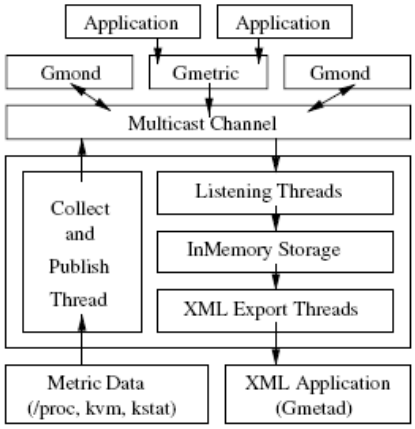

Figura A.6: *Implementação do Ganglia [Massie et al. 2004].*

Dependendo do sistema operacional e da arquitetura do processador utilizados a quantidade de métricas passíveis de serem coletadas e publicadas pelo *gmond* variam entre 28 e 37, que incluem: características do processador (como quantidade, freqüência de operação e etc), da memória física (quantidade total, livre, compartilhada e etc) e *swap*, de processos, do sistema operacional (nome, versão, arquitetura), da MTU (*Maximum Transmission Unit*) e outras. A lista completa de todas as métricas pré-definidas coletáveis pelo Ganglia e em quais plataformas são suportadas são descritas em [Ganglia].

As métricas são publicadas no canal *multicast* no formato XDR por questões de portabilidade e eficiência. As métricas pré-definidas são coletadas através de interfaces bem conhecidas (e.g. /proc, kvm, and kstat) e suas mensagens possuem 4 *bytes* para o nome

da propriedade e 4 ou 8 *bytes* para seu valor, ou seja, seu tamanho máximo é 12 *bytes*.

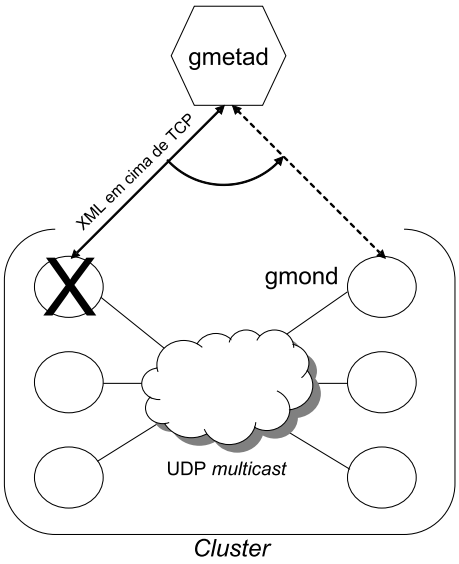

Figura A.7: *Interação entre os monitores do Ganglia. O gmetad processa e apresenta informações coletadas de um ou mais clusters executando o gmond para monitoração local. Caso um gmond falhe outro é escolhido.*

- Federação de *clusters*

Em cada nó na árvore, o *gmetad* coleta periodicamente os dados de monitoração dos seus nós filhos, extrai as informações do XML e o repassa usando *sockets* TCP para o nível acima. Seus nós filhos podem ser *daemons gmond* representando um *cluster* específico, ou *daemons gmetad*, representando conjunto de *clusters*. O nó filho que será consultado pelo *gmetad* é especificado em um arquivo de configuração, no qual é especificada a estrutura da árvore.

A.2.3.3 Considerações

Todos os componentes *gmonds* no Ganglia agem como produtores, por coletarem e divulgarem os eventos referentes ao seu nó. Os *gmetad* funcionam como produtores, do ponto de vista do seu *cluster* e como republicadores do ponto de vista do conjunto de *clusters*.

Uma vantagem da sua arquitetura é a replicação do estado do *cluster* por todos os nós, o que resulta em distribuição da carga e em tolerância a falhas, porém também ocasiona alta intrusão nos *hosts* e na rede.

O *overhead* introduzido, embora seja linear, é considerável tanto nos *hosts* quanto na rede, devido à codificação para o formato XML e à atualização por *multicast*, respectivamente. Outras desvantagens do Ganglia são o fato de contar com a disponibilidade do

multicast e a ausência de um registro, desamparando as características dinâmicas inerentes aos *clusters*, diferentemente dos *grids* (para os quais o Ganglia foi inicialmente criado) que possuem nós mais estáticos.

A.2.4 NetLogger - *Networked Application Logger*

O *toolkit* NetLogger é proposto para análise de desempenho de sistemas complexos tais como aplicações cliente-servidor ou *multi-threaded* [Gunter e Tierney 2003]. Essa ferramenta combina eventos de rede, de *host* e da aplicação, provendo uma visão geral do sistema, o que facilita a identificação de pontos que influenciam o desempenho do sistema.

O NetLogger consiste de quatro componentes: uma API e suas bibliotecas (disponíveis para C, C++, Java, Perl e Python), ferramentas para coleta e manipulação de *logs* (ou seja, dos eventos), sensores de *host* e de rede (que geralmente acessam programas de monitoramento Unix) e uma ferramenta para visualização e análise dos arquivos de *logs*.

Para que uma aplicação produza os *logs*, seu desenvolvedor deve inserir chamadas para a API do NetLogger nos pontos críticos, geralmente antes e depois de requisições de entrada/saída e de computações, e então ligar a aplicação com a biblioteca NetLogger. Os eventos gerados são armazenados localmente (*daemon syslog*) ou em um *host* remoto, neste caso, antes da transmissão, os eventos são armazenados antes em *cache* localmente, de forma a minimizar o *overhead* das altas taxas de geração.

Para a codificação dos dados, o NetLogger provê suporte ao formato de texto ULM (*Universal Logger Message*), à codificação binária e a XML, permitindo ao desenvolvedor escolher o *overhead* desejado. Para permitir a ativação/desativação dinâmica da atividade de *log*, periodicamente um arquivo de configuração é consultado.

Sua metodologia consiste em:

- Todos componentes (incluindo aplicação, middleware, sistema operacional e redes) devem ser instrumentalizados para produzir monitoração.
- Todos os eventos de monitoração devem usar um formato comum e um conjunto comum de atributos. Estes eventos devem possuir um *timestamp* preciso, pertencente a um mesmo fuso horário (GMT) e sincronizado por um método de sincronização de *clock* como o NTP (*Network Time Protocol*).

- Eventos como: entrada e saída de qualquer programa ou componente de *software*, e início e fim de operações de entrada/saída (de disco e de rede) devem ser registrados.
- A coleta dos dados deve ser feita em uma localização central. Porém uma extensão da API com suporte a um registro secundário é proposta para aumentar a tolerância a falhas, caso a central primária se torne indisponível.
- Ferramentas de correlação de eventos e visualização devem ser utilizadas para analisar os registros dos eventos monitorados.

A.2.4.1 Arquitetura

Em [Gunter et al. 2002] são mostrados os componentes principais da monitoração como pode ser visto pela figura A.8, onde estão presentes: a instrumentação da aplicação (*Application instrumentation*), responsável por produzir o dado da monitoração; o serviço de ativação do monitoramento (*Monitoring activation service*), que ativa a instrumentação, coleta os eventos, e os envia para o destino requisitante; o receptor de eventos de monitoração (*Monitoring event receiver*), que consome os dados monitorados; e o alimentador de arquivos (*Archive feeder*), que converte os eventos em registros SQL e os carrega em um repositório de eventos.

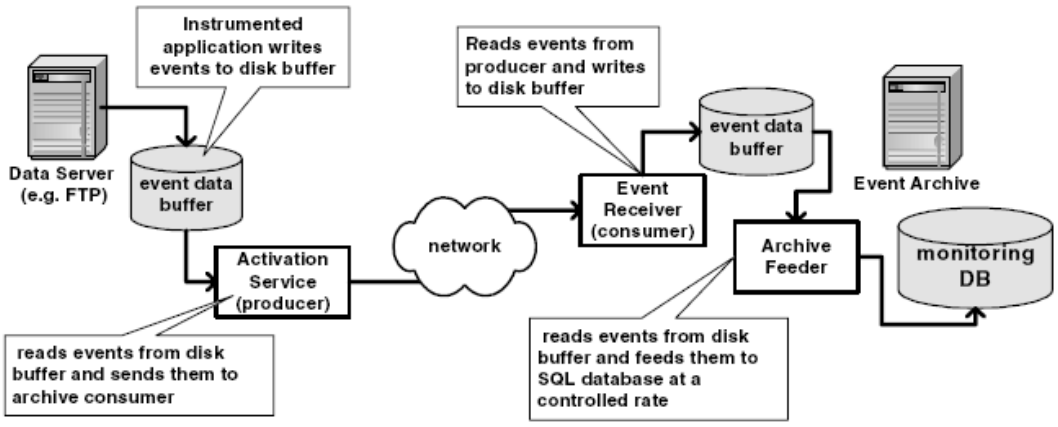

Figura A.8: *Arquitetura de funcionamento do NetLogger [Gunter et al. 2002].*

Com a finalidade de deixar o NetLogger nos moldes dos conceitos propostos na GMA, Gunter et al [Gunter et al. 2002] o extenderam adicionando o serviço de ativação de monitoramento, composto por um gerente de ativação (*Activation manager*), um produtor de ativação (*Producer activation*) e um nó de ativação (*Activation node*)

O gerente de ativação detém os detalhes da atividade de *log* (necessários para as

aplicações instrumentadas com o NetLogger - incluindo a opção de desativação do *log*) e é fornecido um cliente que ajusta esses valores remotamente (e.g. ajuste do arquivo de configuração para desativar a atividade de log). Cada nó de ativação, periodicamente, pergunta ao gerente de ativação o nível desejado de *log* e passa essa informação para as aplicações locais. Essa informação é passada através de arquivos de configuração que são verificados pelas aplicações a uma certa periodicidade. As aplicações arquivam os eventos de monitoração em um arquivo local, que é acessado pelos nós de ativação, responsáveis por repassar esses eventos para o produtor de ativação de forma assíncrona. Este, por sua vez, aceita esses eventos e os envia para os consumidores, que demonstram seus interesses através de filtros.

A.2.4.2 Considerações

No NetLogger, o gerente e o produtor de ativação funcionam como republicadores, enquanto o nó de ativação é responsável pelo papel do produtor.

Uma intrusão considerável é introduzida por causa da consulta periódica ao gerente de ativação, feita pelos nós de ativação; e aos arquivos de configuração efetuada pelas aplicações. Ao invés da configuração manual, um registro poderia ser utilizado para prover a descoberta dinâmica dos componentes do serviço de ativação. Com um registro onde todos componentes pudessem registrar seus detalhes, a intrusão seria menor, e o gerente de ativação poderia informar mudanças no nível de *log* aos nós de ativação ao invés dos nós ficarem perguntando aos gerentes a cada 5 segundos, como feito na implementação atual.

A.2.5 GridRM - *Grid Resource Monitoring*

O GridRM [Baker e Smith 2002] é um projeto de pesquisa que pretende prover uma interface padronizada para o acesso a diversos conjuntos de fontes de dados encontradas nos ambientes de *grid*, como alguns citados anteriormente (e.g. Ganglia, NetLogger, NWS, etc).

No GridRM, toda organização possui um *gateway* (baseado em java) que coleta e normaliza os eventos de um sistema de monitoração local. Os *gateways* funcionam como um republicador dos eventos gerados pelos produtores externos, ou seja, as ferramentas de monitoração utilizadas. Um registro global é utilizado para permitir que consumidores descubram os *gateways* que fornecem a informação de interesse.

O GridRM consiste de duas camadas, uma global e uma local. A primeira, que fornece a interação entre os *grids* formando uma organização virtual (*Virtual Organisation*), é baseada na GMA [Tierney et al. 2001]. Os *gateways* nessa camada colaboram entre si de forma a prover uma visão eficiente e consistente dos dados disponíveis dos recursos. Na camada local, os *gateways* provêem um ponto de acesso para os dados dos recursos locais. Os clientes podem se conectar a qualquer *gateway* e suas requisições por dados remotos são roteadas através da camada global para ser processada pelo *gateway* que possue a informação desejada.

A.2.5.1 Arquitetura

A monitoração dos recurso no GridRM é bastante expansível por possuir *plugins* para a camada de abstração do cliente (e.g. Java, *Web/Grid Services*, etc.) e para as fontes de dados (e.g. NWS, Ganglia, NetLogger, etc.)

Como pode ser visto pela figura A.9, na camada global onde é feita a interface com o cliente, através da camada ACIL (*Abstract Client Interface Layer*), há a separação entre APIs específicas dos clientes e o modelo de dados usado no GridRM.

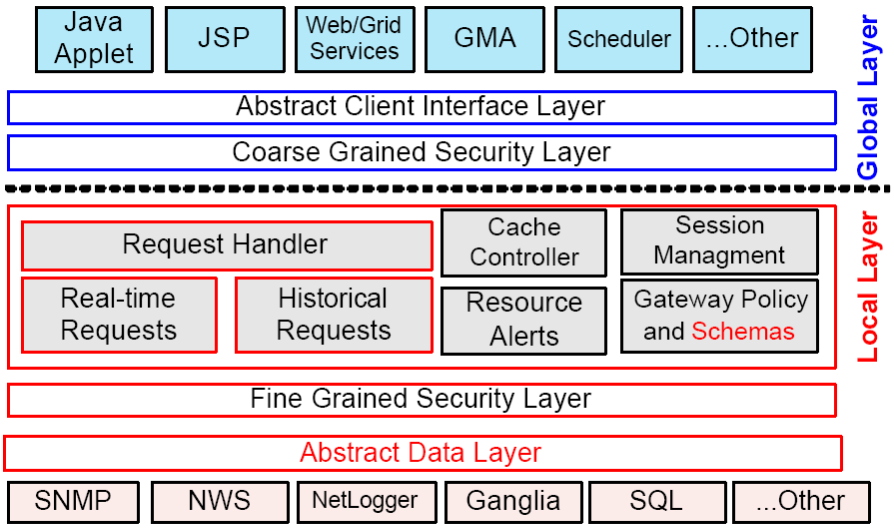

Figura A.9: *As duas camadas do GridRM [Baker e Smith 2003].*

A camada de segurança de granularidade grossa (*Coarse Grained Security Layer* - CGSL), ainda na camada global, provê o acesso controlado para o dado gerenciado, cada *gateway* é responsável pela segurança dos recursos que ele controla e interage. Na hierarquia dos *gateways*, as decisões de segurança podem ser deferidas para o *gateway* local responsável pelo recurso.

A camada local é composta por módulos para consulta de fontes de dados, para tratamento de eventos e para armazenamento/resgate de dados. Os módulos de tratamento de requisições (*Request Handler*) coordenam a busca por dados dos recursos locais e dos controlados por *gateways* GridRM. O manipulador de requisições recebe consultas dos consumidores da camada global e coleta os dados (atuais ou antigos, dependendo do tipo da consulta).

A camada de dados abstratos (*Abstract Data Layer* - ADL) fornece uma abstração para os mecanismos de base e é usada para adquirir e normalizar os dados heterogêneos dos recursos. Os *plugins* para *drivers* das fontes de dados são utilizados para estender a capacidade de coleta de dados da arquitetura de uma forma genérica, permitindo assim a incorporação de qualquer tipo de fonte de dados. Os *plugins* são dinâmicos e os *drivers* podem ser adicionados e removidos em tempo de execução, sem afetar o funcionamento do *gateway*.

A.2.5.2 Considerações

Nessa proposta os produtores são as entidades externas que geram os eventos como o NWS, o Ganglia, o Remos e etc, enquanto os *gateways* funcionam como republicadores dos dados de monitoração.

Referências

[Ansaloni 2003] Ansaloni, S. **Proposta de Padrão Arquitetural para Descrição e Implementação de Contratos de QoS**. Dissertação (Mestrado) — Universidade Federal Fluminense, 2003.

[Atighetchi et al. 2003] Atighetchi, M.; Pal, P. P.; Jones, C. C.; Rubel, P.; Schantz, R. E.; Loyall, J. P. e Zinky, J. A. **Building Auto-Adaptive Distributed Applications: The QuO-APOD Experience**. In: *ICDCSW '03: Proceedings of the 23rd International Conference on Distributed Computing Systems*, Washington, DC, USA. IEEE Computer Society, p. 104, 2003.

[Baker e Smith 2002] Baker, M. e Smith, G. **GridRM: A Resource Monitoring Architecture for the Grid**. In: *GRID '02: Proceedings of the Third International Workshop on Grid Computing*, London, UK. Springer-Verlag, p. 268–273, 2002.

[Baker e Smith 2003] Baker, M. e Smith, G. **GridRM: An Extensible Resource Monitoring System**. In: *Proceedings of the IEEE Cluster Computing Conference*, Hong Kong. , 2003.

[Ban] Ban, B. **A Flexible API for State Transfer in the JavaGroups toolkit**. Acessado em 14 de agosto de 2005. Disponível em: <`http://www.jgroups.org/javagroupsnew/docs/papers/state.ps.gz`>.

[Berman et al. 2005] Berman, F.; Casanova, H.; Chien, A.; Cooper, K.; Dail, H.; Dasgupta, A.; Deng, W.; Dongarra, J.; Johnsson, L.; Kennedy, K.; Koelbel, C.; Liu, B.; Liu, X.; Mandal, A.; Marin, G.; Mazina, M.; Mellor-Crummey, J.; Mendes, C.; Olugbile, A.; Patel, M.; Reed, D.; Shi, Z.; Sievert, O.; Xia, H. e YarKhan, A. **New Grid Scheduling and Rescheduling Methods in the GrADS Project**. *International Journal of Parallel Programming (IJPP)*, Volume 33, n. 2-3, p. 209–229, 2005.

[Beugnard et al. 1999] Beugnard, A.; Jézéquel, J.-M.; Plouzeau, N. e Watkins, D. **Making Components Contract Aware**. *Computer*, v. 32, n. 7, 1999.

[den Burger et al. 2002]den Burger, M.; Kielmann, T. e Bal, H. E. **TOPOMON: A Monitoring Tool for Grid Network Topology**. In: *ICCS '02: Proceedings of the International Conference on Computational Science-Part II*, London, UK. Springer-Verlag, p. 558–567, 2002.

[Cardoso 2005] Cardoso, L. **Integração de Plataformas de Monitoração no Contexto de Arquiteturas Adaptáveis de Software**. Dissertação (Mestrado) — Universidade Federal Fluminense, 2005. Em andamento.

[Cheng et al. 2002] Cheng, S.-W.; Garlan, D.; Schmerl, B.; Steenkiste, P. e Hu, N. **Software Architecture-Based Adaptation for Grid Computing**. In: *HPDC '02: Proceedings of the 11 th IEEE International Symposium on High Performance Distributed Computing HPDC-11 20002 (HPDC'02)*, Washington, DC, USA. IEEE Computer Society, p. 389, 2002.

[Cheng et al. 2004] Cheng, S.-W.; Huang, A.-C.; Garlan, D.; Schmerl, B. R. e Steenkiste, P. **Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure.** In: *ICAC*, p. 276–277, 2004.

[Chiba e Nishizawa 2003] Chiba, S. e Nishizawa, M. **An easy-to-use toolkit for efficient Java bytecode translators**. In: *GPCE '03: Proceedings of the second international conference on Generative programming and component engineering*, New York, NY, USA. Springer-Verlag New York, Inc., p. 364–376, 2003.

[Cirne et al. 2003] Cirne, W.; Silva, D. P.; Costa, L.; Santos-Neto, E.; Brasileiro, F. V.; Sauvé, J. P.; Silva, F. A. B.; Barros, C. O. e Silveira, C. **Running Bag-of-Tasks Applications on Computational Grids: The MyGrid Approach.** In: *ICPP*, p. 407–431, 2003.

[Corradi 2005] Corradi, A. **Um Framework de Suporte a Requisitos Não-Funcionais para Serviços de Nível Alto**. Dissertação (Mestrado) — Universidade Federal Fluminense, 2005.

[Curty 2002] Curty, R. **Uma Proposta para Descrição e Implementação de Contratos para Serviços com Qualidade Diferenciada**. Dissertação (Mestrado) — Universidade Federal Fluminense, 2002.

[Dinda e O'Hallaron 1999] Dinda, P. e O'Hallaron, D. **An extensible toolkit for resource prediction in distributed systems**. Carnegie Mellon University, 1999. Acessado em 28 de agosto de 2005. Disponível em: <citeseer.ist.psu.edu/dinda99extensible.html>.

[Dinda et al. 2001] Dinda, P. A.; Gross, T.; Karrer, R.; Lowekamp, B.; Miller, N.; Steenkiste, P. e Sutherland, D. **The Architecture of the Remos System**. In: *HPDC '01: Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing (HPDC-10'01)*, Washington, DC, USA. IEEE Computer Society, p. 252–266, 2001.

[Duran-Limon e Blair 2004] Duran-Limon, H. A. e Blair, G. S. **QoS Management specification support for multimedia middleware**. *J. Syst. Softw.*, Elsevier Science Inc., New York, NY, USA, v. 72, n. 1, p. 1–23, 2004.

[Florissi 1996] Florissi, P. G. S. **QoSME: QoS Management Environment**. Tese (Doutorado) — Columbia University, 1996.

[Foster 2001] Foster, I. **The Anatomy of the Grid: Enabling Scalable Virtual Organizations**. In: *CCGRID '01: Proceedings of the 1st International Symposium on Cluster Computing and the Grid*, Washington, DC, USA. IEEE Computer Society, p. 6, 2001.

[Frolund e Koistinen 1998] Frolund, S. e Koistinen, J. **Quality of Service Specification in Distributed Object Systems Design.** In: *COOTS*, p. 1–18, 1998.

[Gaidioz et al. 2000] Gaidioz, B.; Tourancheau, B. e Wolski, R. **Synchronizing Network Probes to avoid Measurement Intrusiveness with the Network Weather Service**. In: *HPDC '00: Proceedings of the Ninth IEEE International Symposium on High Performance Distributed Computing (HPDC'00)*, Washington, DC, USA. IEEE Computer Society, p. 147, 2000.

[Ganek e Corbi 2003] Ganek, A. G. e Corbi, T. A. **The dawning of the autonomic computing era**. *IBM Syst. J.*, IBM Corp., Riverton, NJ, USA, v. 42, n. 1, p. 5–18, 2003.

[Ganglia] Ganglia. **Ganglia Documentation Readme**. World Wide Web. Acessado em 2 de setembro de 2005. Disponível em: <`http://ganglia.sourceforge.net/docs/ganglia.html`>.

[Garlan et al. 2004] Garlan, D.; Cheng, S.-W.; Huang, A.-C.; Schmerl, B. e Steenkiste, P. **Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure**. *Computer*, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 37, n. 10, p. 46–54, 2004.

[Garlan et al. 2003] Garlan, D.; Cheng, S.-W. e Schmerl, B. **Increasing System Dependability through Architecture-based Self-repair**. In: *Architecting Dependable Systems, R. de Lemos, C. Gacek, A. Romanovsky (Eds)*, : Springer-Verlag, 2003.

[Gu et al. 2002] Gu, X.; Nahrstedt, K.; Yuan, W.; Wichadakul, D. e Xu, D. **An XML-based Quality of Service Enabling Language for the Web.** *J. Vis. Lang. Comput.*, v. 13, n. 1, p. 61–95, 2002.

[Gunter e Tierney 2003] Gunter, D. e Tierney, B. **NetLogger: A Toolkit for Distributed System Performance Tuning and Debugging**. In: *Integrated Network Management*, p. 97–100, 2003.

[Gunter et al. 2002] Gunter, D.; Tierney, B.; Jackson, K.; Lee, J. e Stoufer, M. **Dynamic Monitoring of High-Performance Distributed Applications**. In: *HPDC '02: Proceedings of the 11 th IEEE International Symposium on High Performance Distributed Computing HPDC-11 20002 (HPDC'02)*, Washington, DC, USA. IEEE Computer Society, p. 163, 2002.

[Guttman et al. 1999] Guttman, E.; Perkins, C.; Veizades, J. e Day, M. **RFC 2608: Service Location Protocol, Version 2.** jun 1999.

[Helm et al. 1990] Helm, R.; Holland, I. M. e Gangopadhyay, D. **Contracts: specifying behavioral compositions in object-oriented systems**. *SIGPLAN Not.*, v. 25, n. 10, 1990.

[Hewlett-Packard 2003] Hewlett-Packard. **Autonomic Computing**. 2003. World Wide Web. Acessado em 15 de julho de 2005. Disponível em: <`http://h71028.www7.hp.com/enterprise/downloads/7611_IS_journey_final.pdf`>.

[Huang 2004] Huang, A.-C. **Building Self-Configuring Services Using Service-Specific Knowledge**. Tese (Doutorado) — Carnegie Mellon University, 2004.

[Huang e Steenkiste 2003] Huang, A.-C. e Steenkiste, P. **Network-Sensitive Service Discovery.** In: *USENIX Symposium on Internet Technologies and Systems*, 2003.

[IBM] IBM. **Autonomic Computing**. World Wide Web. Acessado em 12 de julho de 2005. Disponível em: <`http://www.research.ibm.com/autonomic/overview/`>.

[IBM 2001] IBM. **Autonomic computing: IBM's perspective on the state of information technology**. 2001. World Wide Web. Acessado em 15 de julho de 2005. Disponível em: <`http://www.research.ibm.com/autonomic/manifesto/autonomic_computing.pdf`>.

[Ivica e Larsson 2002] Ivica e Larsson, M. **Building Reliable Component-Based Software Systems**. Norwood, MA, USA. Artech House publisher, 2002.

[JGroups] JGroups. **JGroups - A Toolkit for Reliable Multicast Communication**. World Wide Web. Acessado em 1 de novembro de 2005. Disponível em: <`http://www.jgroups.org/javagroupsnew/docs/index.html`>.

[Jin e Nahrstedt 2004] Jin, J. e Nahrstedt, K. **QoS Specification Languages for Distributed Multimedia Applications: A Survey and Taxonomy**. *IEEE MultiMedia*, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 11, n. 3, p. 74–87, 2004.

[Karaul et al. 1998] Karaul, M.; Korilis, Y. A. e Orda, A. **A market-based architecture for management of geographically dispersed, replicated Web servers**. In: *ICE '98: Proceedings of the first international conference on Information and computation economies*, New York, NY, USA. ACM Press, p. 158–165, 1998.

[Lamanna et al. 2003] Lamanna, D. D.; Skene, J. e Emmerich, W. **SLAng: A Language for Defining Service Level Agreements.** In: *FTDCS*, p. 100–106, 2003.

[Li 2000] Li, B. **Agilos: A Middleware Control Architecture for Application-Aware Quality of Service Adaptations**. Tese (Doutorado) — University of Illinois, 2000. Disponível em: <citeseer.ist.psu.edu/li00agilos.html>.

[Li e Nahrstedt 1999] Li, B. e Nahrstedt, K. **A Control-Based Middleware Framework for Quality of Service Adaptations**. *IEEE Journal of Selected Areas in Communications*, v. 17, n. 9, 1999.

[Lisbôa 2003] Lisbôa, J. C. **Utilização do Design Pattern Architecture Configurator em Um Ambiente de Suporte à Configuração de Arquiteturas**. Dissertação (Mestrado) — Universidade Federal Fluminense, 2003.

[Lisbôa et al. 2002] Lisbôa, J. C.;de Carvalho, S. T. e Filho, O. G. L. **Um Design Pattern para Configuração de Arquiteturas de Software**. In: *SugarLoafPLoP 2002*, Itaipava, RJ, Brasil. , p. 37–53, 2002.

[Lobosco 1999] Lobosco, M. **R-RIO: Um Ambiente para Suporte à Construção e à Evolução de Aplicações**. Dissertação (Mestrado) — Universidade Federal Fluminense, 1999.

[Loques et al. 1997] Loques, O.; Botafogo, R. A. e Leite, J. **A Configuration Approach for Distributed Object-Oriented System Customization**. In: *WORDS '97: Proceedings of the 3rd Workshop on Object-Oriented Real-Time Dependable Systems*, Washington, DC, USA. IEEE Computer Society, p. 185–189, 1997.

[Loques et al. 1999] Loques, O.; Leite, J.; Lobosco, M. e Sztajnberg, A. **On the Integration of Configuration and Meta-level Programming Approaches.** In: *Reflection and Software Engineering*, p. 189–208, 1999.

[Loques et al. 2004] Loques, O.; Sztajnberg, A.; Cerqueira, R. C. e Ansaloni, S. **A contract-based approach to describe and deploy non-functional adaptations in software architectures**. *Journal of the Brazilian Computer Society*, v. 10, n. 1, p. 5–18, July 2004.

[Loyall et al. 1998] Loyall, J. P.; Schantz, R. E.; Zinky, J. A. e Bakken, D. E. **Specifying and Measuring Quality of Service in Distributed Object Systems**. In: *ISORC '98: Proceedings of the The 1st IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, Washington, DC, USA. IEEE Computer Society, p. 43, 1998.

[Lung 1996] Lung, L. C. **Implementação de Técnicas de Replicação de Componentes de Software sobre a Plataforma Aberta CORBA**. Dissertação (Mestrado) — Universidade Federal de Santa Catarina, 1996.

[Malucelli 1996] Malucelli, V. **Babel - Construindo Aplicações por Evolução**. Dissertação (Mestrado) — Pontifícia Universidade Católica do Rio de Janeiro, 1996.

[Mansouri-Samani 1992] Mansouri-Samani, M. **Monitoring Distributed Systems (A Survey)**. Imperial College, London, UK, 1992. Acessado em 15 de setembro de 2005. Disponível em: <citeseer.csail.mit.edu/article/tierney01white.html>.

[Massie et al. 2004] Massie, M. L.; Chun, B. N. e Culler, D. E. **The Ganglia Distributed Monitoring System: Design, Implementation, and Experience**. *Parallel Computing*, v. 30, n. 7, 2004.

[Medvidovic 1999] Medvidovic, N. **Architecture-based specification-time software evolution**. Tese (Doutorado), 1999. Chair-Richard N. Taylor.

[Medvidovic e Taylor] Medvidovic, N. e Taylor, R. N. **A framework for classifying and comparing architecture description languages**. In: *5th ACM SIGSOFT international symposium on Foundations of software engineering.*

[Microsoft 2004] Microsoft. **Dynamic Systems Initiative Overview**. 2004. World Wide Web. Acessado em 15 de julho de 2005. Disponível em: <`http://download.microsoft.com/download/8/7/8/8783b65e-d619-46d7-aa8d-b4f13a97eeb0/DSIoverview.doc`>.

[Molina-Jimenez et al. 2004] Molina-Jimenez, C.; Shrivastava, S.; Crowcroft, J. e Gevros, P. **QoS Monitoring of Service Level Agreements**. March 2004. 30 p. Acessado em 08 de agosto de 2005. Disponível em: <`http://www.newcastle.research.ec.org/tapas/deliverables/D10.pdf`>.

[Moura et al. 2002] Moura, A. L.; Ururahy, C. D.; Cerqueira, R. ede La Rocque Rodriguez, N. **Dynamic Support for Distributed Auto-Adaptive Applications**. In: *ICDCSW '02: Proceedings of the 22nd International Conference on Distributed Computing Systems*, Washington, DC, USA. IEEE Computer Society, p. 451–458, 2002.

[Norris et al. 2004] Norris, J.; Coleman, K.; Fox, A. e Candea, G. **OnCall: Defeating Spikes with a Free-Market Application Cluster**. In: *IEEE Proceedings of the 1st International Conference on Autonomic Computing*, 2004.

[NWS] NWS. **Network Weather Service FAQ**. World Wide Web. Acessado em 26 de setembro de 2005. Disponível em: <`http://nws.cs.ucsb.edu/faq.html`>.

[Othman et al. 2004] Othman, O.; Balasubramanian, J. e Schmidt, D. C. **Performance Evaluation of an Adaptive Middleware Load Balancing and Monitoring Service**. In: *ICDCS '04: Proceedings of the 24th IEEE International Conference on Distributed Computing Systems*, Tokyo, Japan. IEEE Computer Society, 2004.

[Pal et al. 2000] Pal, P.; Loyall, J.; Schantz, R.; Zinky, J.; Shapiro, R. e Megquier, J. **Using QDL to Specify QoS Aware Distributed (QuO) Application Configuration**. In: *ISORC '00: Proceedings of the Third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, Washington, DC, USA. IEEE Computer Society, p. 310–319, 2000.

[Perry 1987] Perry, D. E. **Software interconnection models**. In: *9th international conference on Software Engineering*, Los Alamitos, CA, USA. IEEE Computer Society Press, p. 61–69, 1987.

[Plale et al. 2002] Plale, B.; Dinda, P.; Helm, M.;von Laszewski, G. e McGee, J. **Key Concepts and Services of a Grid Information Service**. In: *15th International Conference on Parallel and Distributed Computing Systems (PDCS 2002)*, Louisville, KY. , p. 437–442, 2002. Disponível em: <http://www.mcs.anl.gov/ gregor/papers/plale–pdcs2002.pdf>.

[Rajkumar et al. 1997] Rajkumar, R.; Lee, C.; Lehoczky, J. e Siewiorek, D. **A resource allocation model for QoS management**. In: *RTSS '97: Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS '97)*, Washington, DC, USA. IEEE Computer Society, p. 298, 1997.

[Santos 2005] Santos, A. **Proposta de Padrão Arquitetural para Descrição e Implementação de Contratos de QoS**. Dissertação (Mestrado) — Universidade Federal Fluminense, 2005. Em andamento.

[Schneider 1990] Schneider, F. B. **Implementing fault-tolerant services using the state machine approach: a tutorial**. *ACM Comput. Surv.*, ACM Press, New York, NY, USA, v. 22, n. 4, p. 299–319, 1990.

[Shaw et al. 1995] Shaw, M.; DeLine, R.; Klein, D. V.; Ross, T. L.; Young, D. M. e Zelesnik, G. **Abstractions for Software Architecture and Tools to Support Them**. *IEEE Trans. Softw. Eng.*, v. 21, n. 4, 1995.

[Sztajnberg 2002] Sztajnberg, A. **Flexibilidade e Separação de Interesses para a Concepção e Evolução de Aplicações Distribuídas**. Tese (Doutorado) — Universidade Federal do Rio de Janeiro, 2002.

[Tannenbaum et al. 2002] Tannenbaum, T.; Wright, D.; Miller, K. e Livny, M. **Condor: a distributed job scheduler**. MIT Press, Cambridge, MA, USA, p. 307–350, 2002.

[Tierney et al. 2001] Tierney, B.; Aydt, R.; Gunter, D.; Smith, W.; Taylor, V. e Wolski, R. **White Paper: A Grid Monitoring Service Architecture (DRAFT)**. Global Grid Forum, 2001. Acessado em 05 de setembro de 2005. Disponível em: <citeseer.csail.mit.edu/article/tierney01white.html>.

[Wang et al. 2001] Wang, N.; Schmidt, D. C.; Kircher, M. e Parameswaran, K. **Adaptive and Reflective Middleware for QoS-Enabled CCM Applications.** *IEEE Distributed Systems Online*, v. 2, n. 5, 2001.

[Wegner 1996] Wegner, P. **Interoperability**. *ACM Comput. Surv.*, ACM Press, New York, NY, USA, v. 28, n. 1, p. 285–287, 1996.

[Wolski et al. 1999] Wolski, R.; Spring, N. T. e Hayes, J. **The network weather service: a distributed resource performance forecasting service for metacomputing**. *Future Gener. Comput. Syst.*, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, v. 15, n. 5-6, p. 757–768, 1999.

[Yu et al. 2005] Yu, J.; Buyya, R. e Tham, C.-K. **QoS-based Scheduling of Workflow Applications on Service Grids**. University of Melbourne, Australia, March 2005. 8 p. Acessado em 11 de agosto de 2005. Disponível em: <`http://www.gridbus.org/reports/QoSWorkflow.pdf`>.

[Zanikolas e Sakellariou 2005] Zanikolas, S. e Sakellariou, R. **A taxonomy of grid monitoring systems**. *Future Gener. Comput. Syst.*, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, v. 21, n. 1, p. 163–188, 2005.

[Zegura et al. 2000] Zegura, E. W.; Ammar, M. H.; Fei, Z. e Bhattacharjee, S. **Application-layer anycasting: a server selection architecture and use in a replicated Web service**. *IEEE/ACM Trans. Netw.*, IEEE Press, Piscataway, NJ, USA, v. 8, n. 4, p. 455–466, 2000.