# NISUS: UMA ARQUITETURA PARA GERENCIAMENTO DE DESEMPENHO DE SISTEMAS DISTRIBUÍDOS CORBA

RENATO FICHE JUNIOR

DCC-IM/UFRJ
Mestrado em Informática

Orientador:
Prof. Carlo Emmanoel Tolla de Oliveira, Ph.D.

Rio de Janeiro

2005

# NISUS: UMA ARQUITETURA PARA GERENCIAMENTO DE DESEMPENHO DE SISTEMAS DISTRIBUÍDOS CORBA

**Renato Fiche Junior**

Dissertação submetida ao corpo docente do Instituto de Matemática e do Núcleo de Computação Eletrônica da Universidade Federal do Rio de Janeiro - UFRJ, como parte dos requisitos necessários à obtenção do grau de Mestre em Informática.

Aprovada por:

Prof._____ (Orientador)
Carlo Emmanoel Tolla de Oliveira, Ph.D.

Prof._____
Paulo Figueiredo Pires, D. Sc.

Prof._____
Francisco Reverbel, Ph.D.

Rio de Janeiro

2005

FICHE JUNIOR, RENATO.

NISUS: Uma Arquitetura para Gerenciamento de Desempenho de Sistemas Distribuídos CORBA / Renato Fiche Junior. Rio de Janeiro: UFRJ / IM / DCC, 2005.

91 pp.

Dissertação (Mestrado em Informática) – Universidade Federal do Rio de Janeiro - UFRJ, IM / DCC, 2005.

Orientador: Prof. Carlo Emmanoel Tolla de Oliveira

1. Gerenciamento de Desempenho. 2. CORBA 3. Portable Interceptors.

I. IM/NCE/UFRJ. II. Título (série).

*Apenas uma página não seria suficiente para expressar os meus agradecimentos, mas vou resumir. Primeiramente, agradeço a Deus por me dar saúde, felicidade e sabedoria. Em segundo, agradeço à minha família, principalmente aos meus pais, pelo apoio e paciência nos momentos difíceis por que enfrentei ao longo do mestrado e da vida. Por último, agradeço aos grandes amigos que fiz, mesmo àqueles que infelizmente já se foram.*

# RESUMO

FICHE JUNIOR, Renato. **NISUS;** uma arquitetura para gerenciamento de desempenho de sistemas distribuídos CORBA. Orientador: Prof. Carlo Emmanoel Tolla de Oliveira. Rio de Janeiro: UFRJ/IM, 2004. Dissertação (Mestrado em Informática).

Para garantir a qualidade do serviço dos sistemas, a necessidade de identificar rapidamente problemas no seu funcionamento tem chamado a atenção das empresas. Isso torna o gerenciamento de sistemas um aspecto fundamental. Ao mesmo tempo, há uma dificuldade em aplicá-lo aos sistemas distribuídos por sua complexidade e descentralização geográfica e sem afetar o seu desempenho. Em vista do problema apresentado, esse trabalho tem o objetivo de apresentar uma arquitetura para gerenciamento de desempenho para qualquer sistema distribuído CORBA baseado em Java. Ele utiliza SNMP para monitorar os componentes de hardware e um mecanismo incluído na especificação CORBA que provê aos desenvolvedores a possibilidade de estender a funcionalidade do ORB – *Portable Interceptors*. A idéia consiste em um conjunto de componentes que cooperam entre si a fim de prover um gerenciamento de desempenho robusto. O serviço apresentado permite que o desempenho seja gerenciado de forma centralizada e transparente, sendo capaz de rastrear problemas em partes bem específicas de tais sistemas sem interferir do seu desempenho. Um experimento foi desenvolvido e o Sistema de Gestão Acadêmica da UFRJ – SIGA – foi utilizado como estudo de caso, e os resultados serão mostrados e discutidos nesse trabalho.

# ABSTRACT

FICHE JUNIOR, Renato. **NISUS;** uma arquitetura para gerenciamento de desempenho de gerenciamento de sistemas distribuídos CORBA. Orientador: Prof. Carlo Emmanoel Tolla de Oliveira. Rio de Janeiro: UFRJ/IM, 2004. Dissertação (Mestrado em Informática).

The need for service quality in enterprise applications is driving companies to profile their online performance. The application performance management is an important aspect. However, distributed applications are hard to manage due to their complexity and geographical dispersion. To cope with this problem, this work presents a Java based management architecture solution for CORBA distributed applications. The solution uses SNMP for hardware component monitoring and a CORBA feature that provides to developer the ability to extend the ORB functionality - Portable Interceptors. The solution has a set of components that cooperate to each others to provide a non-intrusive and a robust performance management service. A detailed analysis through a web console can then be performed to expose behavioral problems in specific parts of the application without performance overhead. A prototypical implementation was tested and the SIGA from UFRJ was used as a case study. The results are showed and discussed in this work.

**/MEG.**

## LISTA DE SIGLAS

| | |
|---|---|
| ARM | Application Resource Measurement |
| COM | Component Object Model |
| CORBA | Component Object Request Architecture |
| NSM | Network and Systems Management |
| ORB | Object Request Broker |
| OMG | Object Management Group |
| QoS | Quality of Service |
| PI | Portable Interceptors |
| RMI | Remote Method Invocation |
| SLA | Service Level Agreements |
| SLM | Service Level Management |
| SNMP | Simple Network Management Protocol |
| XMI | Extensible Markup Interchange |
| XML | Extensible Markup Language |
| UML | Unified Language Modeling |
| IIOP | Internet Inter-ORB Protocol |
| MIB | Management Information Base |

# LISTA DE FIGURAS

# LISTA DE TABELAS

# SUMÁRIO

# INTRODUÇÃO

Por serem complexos e atenderem a um grande número de usuários, os sistemas normalmente são distribuídos. E, entender o comportamento de tais sistemas é um dos maiores desafios que a engenharia de software enfrenta atualmente. Entretanto, diversos componentes de software e hardware compõem um sistema distribuído; podendo estar dispostos em diversos pontos da rede da empresa. Portanto, nesse cenário, o gerenciamento se torna uma tarefa difícil de ser aplicada, comparando-se a uma aplicação local monousuário.

Atualmente, falhas provocadas por erros de programação e alta latência são comuns nos sistemas distribuídos. Isso torna o gerenciamento de desempenho de sistemas essencial, principalmente quando eles estão no ambiente de produção. Isso permite que problemas no seu funcionamento (e.g. falhas provocadas por erros de programação e alta latência[1]) sejam identificados rapidamente. E, quanto menor for o intervalo entre a ocorrência e a correção desses problemas, maior será a disponibilidade e a confiabilidade de tais sistemas.

No entanto, as soluções de gerenciamento de desempenho existentes não apresentam informação detalhada sobre o funcionamento dos sistemas distribuídos. Em alguns casos, apenas dados sobre o estado da rede e das máquinas usadas para servi-los são conhecidos (e.g. tráfego de rede; e consumo de CPU e memória). Contudo, esses dados são insuficientes para determinar com precisão a origem dos problemas de desempenho. Além disso, algumas soluções não apresentam preocupação em como as informações serão apresentadas para os desenvolvedores e administradores.

Em outros casos, os dados estão espalhados e disponíveis apenas em arquivos *log* sem um formato padrão, dificultando a sua interpretação. Além disso, elas normalmente exigem esforço ao serem adotadas, necessitando que sejam realizadas customizações e

[1] Tempo que decorre entre a aplicação de um estímulo e a resposta provocada.

modificações contínuas. Portanto, elas acabam reduzindo a produtividade dos desenvolvedores e tornando o processo de análise de desempenho tedioso e demorado.

Em vista dos problemas apresentados, o presente trabalho propõe o NISUS, uma arquitetura para gerenciamento de desempenho de sistemas distribuídos que são baseados em CORBA e na linguagem Java. Ele é composto por um conjunto de componentes que trabalham juntos a fim de gerenciar o desempenho dos objetos CORBA do sistema, falhas e consumo de recursos de hardware. Um console Web foi proposto e implementado e mostra que informações bem específicas podem ser disponibilizadas aos administradores e desenvolvedores.

Através de um experimento e de um estudo de caso, o trabalho aqui apresentado mostra que é viável coletar informações detalhadas e necessárias sobre o desempenho de sistemas distribuídos CORBA sem que o seu desempenho seja comprometido.

A dissertação que relata o trabalho desenvolvido está organizada da seguinte forma: no primeiro capítulo. O primeiro apresenta uma revisão literária, mostrando as principais motivações para o gerenciamento de desempenho e as principais técnicas atualmente empregadas. No segundo, faz-se uma apresentação dos principais conceitos tecnológicos necessários para a compreensão do trabalho. No terceiro, a solução arquitetural é apresentada e detalhada. No quarto, a implementação dos componentes da arquitetura é mostrada. No quinto, faz-se uma apresentação do protótipo e do estudo de caso, onde os resultados são discutidos. No sexto e último capítulo, é feita a conclusão deste trabalho, apresentando perspectivas futuras.

# 1 REVISÃO LITERÁRIA

*Este capítulo, com base nos trabalhos realizados pela comunidade acadêmica, apresenta as principais motivações e razões por que o gerenciamento de desempenho de sistemas distribuídos é necessário. Ele apresenta, por fim, os principais trabalhos relacionados sobre monitoramento e gerenciamento de sistemas distribuídos.*

## 1.1 INTRODUÇÃO

Os sistemas distribuídos requerem um nível de gerenciamento completamente diferente do que aplicações locais monousuário. Isto por causa da sua complexidade e dispersão geográfica. SCALLAN (2000) faz uma revisão dos principais problemas que podem ser encontrados durante a execução de um sistema distribuído: gargalos de desempenho, uso excessivo de recursos, falhas de rede, *race conditions*, *deadlocks*, erros de programação etc. E, quanto menor for tempo de reação a tais problemas, maior será a disponibilidade do sistema. Portanto, o acompanhamento contínuo do funcionamento do sistema distribuído é uma tarefa importante para garantir a satisfação dos seus usuários.

Segundo DENNIS (2002), as soluções de gerenciamento de rede e sistemas (*Network and Systems Management* – NSM) são responsáveis por uma grande fatia do orçamento em tecnologia da informação (TI). Em 2002, por exemplo, empresas investiram aproximadamente $7,1 bilhões na compra de produtos desta categoria. As empresas fornecedoras de tais produtos são lideradas pela *IBM Tivoli* com 13,9% do mercado e a *Computer Associates Inc.* com 13,8 %. Portanto, as empresas têm feito um enorme investimento nessa área.

## 1.2 TESTE DE DESEMPENHO versus GERENCIAMENTO DE DESEMPENHO

Teste de desempenho refere-se à avaliação de desempenho antes de o sistema entrar no ambiente de produção. E, gerenciamento de desempenho, refere-se ao acompanhamento do desempenho dos sistemas desde a sua implantação até o fim de sua utilização.

Os testes de desempenho ajudam a antecipar alguns problemas que seriam enfrentados durante a utilização do sistema. Porém, normalmente eles exigem modificações extensas nos sistemas, criação de inúmeros cenários de teste de desempenho. Conseqüentemente, as ferramentas de teste exigem muito esforço ao serem implantadas nos sistemas. Segundo PRESSMAN (2001), manutenção é 80% desenvolvimento e 20% correção de erros. A manutenção é inevitável. Para continuar ativo, o sistema deve se adaptar. Como os sistemas estão em constante evolução, torna-se custoso aplicar os testes continuamente e manter os cenários de teste atualizados.

Além disso, os testes de desempenho exigem recurso de hardware que seja capaz de simular dezenas ou centenas de usuários utilizando o sistema simultaneamente. Como conseqüência, os testes dificilmente retratam com precisão o que seria experimentado se o sistema estivesse em produção. Com isso, as estimativas de desempenho acabam não sendo confiáveis. De acordo com pesquisas da GARTNER GROUP (2003) *apud:* MIDDLEWARE COMPANY (2003), apenas 14% das estimativas de tempo de resposta dos sistemas são corretas.

O gerenciamento de desempenho de sistemas tem o objetivo de acompanhar constantemente o desempenho enquanto os mesmos estiverem em produção. Dessa forma, as informações fornecidas através do gerenciamento são mais confiáveis, pois representam um retrato do que está realmente acontecendo com o desempenho do sistema. Portanto, o gerenciamento de sistemas, ao contrário dos testes de desempenho não lida com estimativas e sim com certezas.

Quando um sistema é colocado em produção, ele fica disponível a um número razoavelmente grande de usuários. Com isso, novos requisitos surgem, fazendo novas funcionalidades tenham quer ser implementadas e as existentes sejam constantemente modificadas. Dessa forma, o processo de manutenção pode afetar o desempenho do sistema. E, como aplicar os testes de desempenho nesse contexto de inúmeras mudanças pode ser custoso, o gerenciamento de desempenho torna-se ainda mais necessário.

## 1.3 CONTRADOS AO NÍVEL DE SERVIÇO

Contratos ao nível de serviço (ou *Service Level Agreements* – SLA) representam acordos onde requisitos mínimos aceitáveis de qualidade de serviço (QoS) são negociados entre clientes e fornecedores. Um SLA é um documento formal e normalmente anexado ao contrato. E, o não cumprimento do SLA implica em penalidades, estipuladas no contrato, para o fornecedor do serviço.

Um SLA deve conter parâmetros objetivos e mensuráveis os quais o fornecedor de serviços se compromete a atender. Esses parâmetros podem ser:

- Requisitos de desempenho;
- Disponibilidade do sistema ou serviço;
- Tempo de identificação das causas de uma falha no sistema;
- Tempo de reparo de uma falha no sistema;
- Tempo relacionado à entrega de um serviço ao cliente

Quando claramente definidos e compreendidos, os SLAs tornam-se ferramentas de gerência poderosas para os usuários (SCHWEITZER, 1999). Com a definição desses parâmetros, SLAs podem dar aos clientes garantia de desempenho, correção de falhas e disponibilidade dos serviços. Conseqüentemente, os SLAs representam, também, ferramentas estratégicas que ajudam fornecedores a diferenciar suas ofertas.

O SLA deve representar um acordo entre clientes e fornecedores onde prevaleça o bom senso. A sua ausência torna difícil, para um cliente, cobrar qualidade de serviço de um fornecedor. Por outro lado, um SLA muito rigoroso pode aumentar o custo do serviço e forçar que o fornecedor acabe fazendo promessas que não conseguirá cumprir depois. O SLA pode ser dinâmico, podendo ser alterado a qualquer momento do projeto.

Finalmente, as informações sobre a qualidade do serviço devem ser passadas aos clientes através de relatórios, permitindo, assim, um melhor controle sobre o serviço contratado. Portanto, os fornecedores devem adotar algum mecanismo de gerência de SLAs para mostrar o cumprimento dos acordos. Esse processo denomina-se *Service Level Management*. Portanto, os sistemas devem ser constantemente gerenciados para apoiar esse processo de verificação do cumprimento das SLAs.

## 1.4 REQUISITOS PARA GERENCIAMENTO DE DESEMPENHO

Como foi visto, o gerenciamento de sistemas faz-se necessário. Contudo, alguns trabalhos mostram que alguns requisitos são importantes para aplicar gerenciamento de desempenho aos sistemas distribuídos. SOUDER *et al* (2001) apresentam alguns requisitos para sistemas de gerenciamento de desempenho:

- Capacidade em lidar com grande quantidade de informações – Normalmente os sistemas distribuídos são complexos e grandes, produzindo uma quantidade expressiva de chamadas remotas que precisam ser monitoradas;
- Capacidade em filtrar informações – Segundo o autor, em muitos casos, apenas um subconjunto do sistema precisa ser monitorado, tornando necessário um filtro de informações para evitar que informação excessiva seja mostrada. VANHELSUWÉ (2003) fez uma análise em três ferramentas de teste de desempenho: *Borland's Optimizeit Suite*, *Quest Software's JProbe Suite* e *ej-technologies' JProfiler*. Segundo o autor, um dos problemas encontrados está relacionado com a forma de apresentação das informações. A Figura 1, por exemplo, mostra um grafo gerado por uma das ferramentas e retrata o problema da falta de um filtro. A falta de um filtro fez com que demasiada informação tenha sido apresentada, dificultando a compreensão dos dados e, conseqüentemente, o processo de análise.

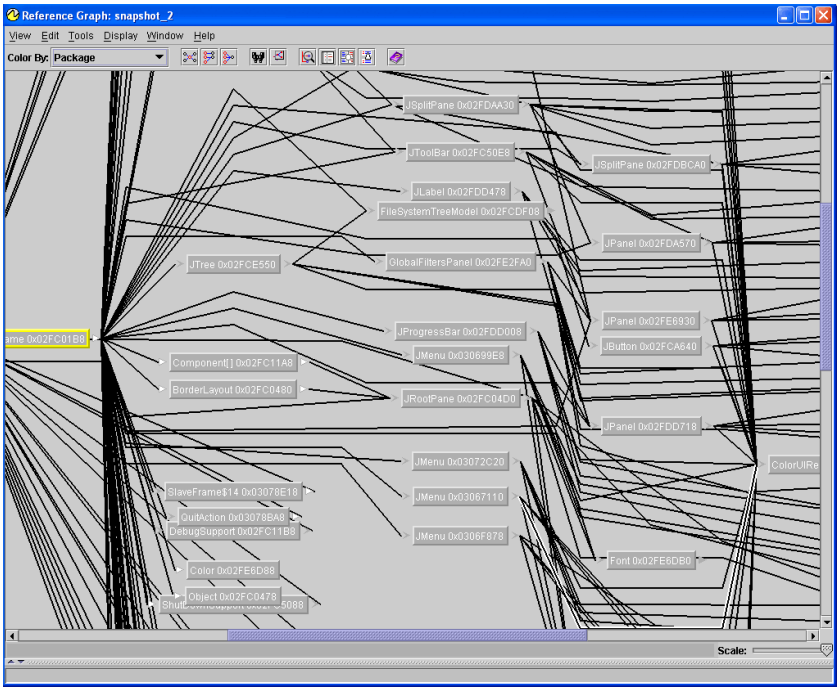


Figura 1 – Problema causado pela falta de filtro de informações.

- O gerenciamento de desempenho deve ser distribuído – Um sistema distribuído constitui-se de objetos remotos localizados em diversas máquinas da rede. Portanto, para monitorar tais objetos, o sistema de gerenciamento de desempenho deve ser também distribuído, sendo que os seus componentes também devem estar.

MOE e CARR (2001) apresentam alguns outros requisitos:

- Mínimo impacto no desempenho – Se uma quantidade excessiva e desnecessária for processada, o desempenho pode ser comprometido. Um filtro de informações pode evitar que isso aconteça, por exemplo. E, segundo o autor, o esforço dos desenvolvedores em adotar um sistema de gerenciamento deve ser mínimo;
- A visualização das informações deve ser fácil – Os desenvolvedores não devem gastar muito tempo para analisar as informações coletadas; senão o processo de detecção de gargalos se tornará demorado e ineficiente.

## 1.5 TÉCNICAS DE MONITORAMENTO TRADICIONAIS

HAUCK e RADISIC (2001) classificam as técnicas tradicionais para monitoramento de desempenho em:

- Monitoramento de tráfego de rede;
- Monitoramento no nível de sistema operacional – Essa técnica refere-se ao monitoramento de alguns parâmetros do sistema, tais como CPU, utilização de memória, número de processos em estado de execução etc. Esses parâmetros não são suficientes para verificar o cumprimento das SLAs.
- Monitoramento do lado cliente da aplicação – Essa técnica tem o objetivo de monitorar o desempenho do sistema sob a perspectiva do usuário. Ela é ideal para verificar o cumprimento das SLAs, mas torna-se impossível identificar a raiz de um mal funcionamento na aplicação;
- Monitoramento da aplicação como um todo – Como monitorar o lado cliente da aplicação não é suficiente, monitorar o lado servidor também se faz necessário. Nessa categoria, normalmente duas abordagens são utilizadas para obter dados sobre o funcionamento das aplicações distribuídas no lado servidor: instrumentação de código[2] e varredura de arquivos de *log*.

Os autores concluem que essas técnicas não são adequadas o suficiente. Segundo eles, a instrumentação de código parece ser a abordagem mais adequada. Entretanto, por causa do enorme esforço exigido para os desenvolvedores, ela não é normalmente utilizada. Além disso, o código-fonte do sistema torna-se ilegível e difícil de ser mantido quando código-fonte extra é inserido no sistema. Portanto, a instrumentação deve ser automática, ou seja, transparente tanto para os desenvolvedores quanto para os administradores do sistema.

## 1.6 TRABALHOS RELACIONADOS

[2] Instrumentação significa inserir código extra, diretamente no código do sistema, responsável pela coleta de dados, principalmente sobre o desempenho do sistema.

Há trabalhos similares para sistemas distribuídos que utilizam a tecnologia CORBA. O trabalho de DEBUSMANN *et al* (2002) também apresenta uma solução baseada em *Portable Interceptors*. Porém, de acordo com os resultados, a solução apresentou um aumento de 65% no tempo de resposta em função do uso de ARM (*Application Resource Measurement*). Além disso, este trabalho não apresenta preocupação em como os dados são coletados e visualizados.

MOE e CARR (2001) apresentam uma solução que ajudam os desenvolvedores a entender o comportamento do funcionamento de sistemas distribuídos também baseados em CORBA. Uma ferramenta de visualização foi usada para estudar as estatísticas. Porém, este trabalho não permite o monitoramento on-line de desempenho, falhas e recursos consumidos pelas aplicações.

SRIDHARAN, DASARATHY e MATHUR (2000) apresentaram um *framework* para coletar estatísticas de desempenho. Como este trabalho também foi baseado em *Portable Interceptors*, o código das aplicações não precisa ser modificado. O *framework* é capaz de medir o uso de CPU e memória para cada transação. Entretanto, informação sobre o desempenho é armazenada apenas em arquivos de *log*. Ele prove uma ferramenta de visualização, mas ela pode ser usada apenas enquanto a aplicação está executando.

O projeto *CorbaTrace* (CORBATRACE 2003) usa *Portable Interceptors* para que a comunicação entre os objetos CORBA seja armazenada em arquivos XML. Para melhor visualização, o projeto prove formas de transformá-los em arquivos XMI[3] para serem vistos como diagramas de seqüência[4] em qualquer ferramenta UML. No entanto, eles não apresentam informação sobre o desempenho da aplicação, ou seja, o projeto destina-se à geração automática de documentação.

[3] XMI é um padrão para representar modelos UML em XML.
[4] Um diagrama de seqüência mostra interações entre objetos organizadas seqüencialmente no tempo. Em particular, mostra os objetos participando de uma interação e a sucessão de mensagens trocadas entre eles.

Existem soluções especificas para sistemas construídos com a tecnologia EJB. MOS e MURPHY (2001) apresentam um *framework* genérico e não-intrusivo para monitoramento de desempenho de aplicações distribuídas baseadas nesta tecnologia. As informações de desempenho são enviadas para uma fila de mensagens JMS e um console gráfico retiram as mensagens da fila e apresentam as informações para o usuário. Entretanto, ele permite apenas análise de desempenho em tempo real e fornece poucas informações e gráficos. DEBUSMANN *et al* (2002) apresenta uma solução genérica para avaliar o desempenho de aplicações EJB usando JMX, mas eles não apresentam preocupação com a coleta e apresentação das informações.

A aplicação *Broker* foi usada por alguns anos no antigo sistema de inscrição em disciplinas da Universidade Federal do Rio de Janeiro – UFRJ. Ele serviu para mostrar o número corrente de clientes atendidos e a latência média em cada servidor Web. Além disso, o algoritmo *Round Robin* foi usado para distribuir igualmente o número de clientes entre o conjunto de servidores Web. Entretanto, a carga de cada servidor Web era baseada apenas no número de clientes e com o *Broker* não era possível obter com precisão a origem de um problema de desempenho.

### 1.6.1 CONCLUSÃO

Este capítulo discutiu as diferenças entre teste e gerenciamento de desempenho, apresentou as motivações para gerenciamento de desempenho e os principais requisitos para aplicá-lo aos sistemas distribuídos. Conclui-se que embora gerenciar o desempenho seja necessário, as técnicas aplicadas atualmente não são adequadas o suficiente. Elas não devem ser utilizadas individualmente e, ao mesmo tempo, os requisitos para gerenciamento de desempenho devem ser cumpridos.

# 2 REVISÃO TECNOLÓGICA

*Este capítulo apresenta os principais aspectos tecnológicos necessários para a compreensão da solução proposta. Primeiramente, ele apresenta os dois principais modelos de gerenciamento de rede, o protocolo SNMP. Em seguida, ele apresenta os conceitos de sistemas distribuídos e a tecnologia CORBA. Por fim, o capítulo apresenta Portable Interceptors e suas aplicações.*

## 2.1 SNMP

Os dois principais modelos de gerenciamento de redes são o OSI (YEMINI 1993) e o Internet. O modelo OSI utiliza o protocolo CMIP (IETF 1995) e o modelo Internet utiliza os protocolos SNMP (IETF 1990) e o RMON (IETF 1990).

O protocolo SNMP foi desenvolvido para gerenciar dispositivos e equipamentos dentro de uma rede TCP/IP. Projetado inicialmente para aplicações de gerência de redes simples, o protocolo firmou-se no mercado, sendo o protocolo de gerência mais utilizado em redes de comunicação de dados.

Conforme indica o próprio nome do protocolo, sua estrutura é simples em todos os sentidos, o que facilita muitos aspectos importantes da sua implementação. Na prática, um protocolo de gerência deve ser realizável em um grau que permita aos produtos nele baseados um fácil desenvolvimento e alto nível de interoperabilidade, de forma transparente. Por outro lado, além da despretensão, o protocolo foi projetado visando não interferir no desempenho das redes gerenciadas, procurando reduzir o tráfego de mensagens de gerência necessárias.

### 2.1.1 SURGIMENTO

Em resposta à necessidade de um protocolo de gerência para as redes baseadas em TCP/IP, o IAB (*Internet Architecture Board*) realizou o primeiro esforço para uma padronização de um protocolo simples de gerência de redes consolidado na RFC 1052

(*Recommendations for the Development of Internet Network Managements Standards*) (IETF 1998). Este documento descrevia as diretrizes recomendadas no projeto de protocolos de gerência.

O passo seguinte foi a especificação do protocolo proposto, na forma de documentos RFC's candidatos a se tornarem padrão (IETF 1988).

A partir daí, a primeira versão do SNMP (SNMPv1) foi especificada nos documentos apresentados na Tabela 1.

| RFC | Título | Ano |
|---|---|---|
| RFC 1155 | Structure and Identification of Management Information for TCP/IP-based Internets | 1990 |
| RFC 1157 | A Simple Network management Protocol | 1990 |
| RFC 1213 | Management Information Base for Network Management of TCP/IP-based Internets: MIB-II | 1991 |

Tabela 1 – Documentos de especificação do protocolo SNMPv1

Estes documentos alcançaram o status de *Internet standard* e foram o alicerce para a aceitação da arquitetura SNMP. Neles estão as descrições de como devem ser organizadas as informações de gerenciamento numa base de dados local ao elemento gerenciado e como o protocolo SNMP pode ser usado para coletá-las e alterá-las. A partir destes documentos, houve um rápido desenvolvimento de bases de dados de gerência (*MIB – Management Information Base*) para vários tipos de equipamentos e interfaces, além de especificações para adaptação do SNMP a protocolos de transporte proprietários já conhecidos.

### 2.1.2 ARQUITETURA

A arquitetura inicial SNMP deriva do SGMP (*Simple Gateway Management Protocol*) (IETF 1987), protocolo desenvolvido para gerenciar roteadores e equipamentos de interconexão de uma rede TCP/IP. A propósito, a primeira versão do SNMP

apresentava informações para gerenciamento e controle desses dispositivos de interconexão, o que determinou as escolhas dos elementos componentes da arquitetura SNMP.

Dessa forma, em uma rede sendo gerenciada segundo a arquitetura SNMP, pode-se distinguir os seguintes elementos específicos de gerência de rede:

1. Vários equipamentos, dispositivos e softwares que contêm cada um uma entidade de processamento SNMP chamada de agente;
2. Pelo menos uma estação gerenciadora (gerente) que contém a inteligência e o domínio das operações de gerência. Esta estação executa aplicações de gerência que monitoram e controlam os elementos da rede, por meio de acesso remoto à informação de gerenciamento armazenada nesses elementos;
3. Um protocolo específico usado para transferir informações de gerência entre agentes e gerentes.

Os agentes possuem um repositório de informações de gerência que é chamado MIB (*Management Information Base*). É de responsabilidade do agente manter na MIB um reflexo da realidade operacional e administrativa do recurso sendo gerenciado. Desta forma, o gerente pode saber o que está acontecendo por meio de pedidos de informações da MIB do agente.

É importanto observar que cabe à estação gerente fazer a coleta de dados de cada elemento gerenciado por intermédio do agente correspondente. O protocolo SNMP é então responsável pelo transporte das informações de gerenciamento entre o gerente e os agentes existentes nos elementos de rede.

Os agentes, normalmente componentes de software, operam no elemento de rede gerenciado e funcionam basicamente respondendo à solicitações (*requests*) de um gerente SNMP e enviando ou respostas a tais solicitações (*responses*), ou ainda notificações espontâneas (*traps*) sobre o estado do elemento.

### 2.1.3 MIB

A organização das informações na MIB de um agente SNMP é estruturada na forma de objetos gerenciados. Estes objetos, que são unidades de armazenamento de dados representados na MIB por suas instâncias únicas ou múltiplas, compõem a base de informações de gerência referente àquele determinado elemento da rede.

A primeira especificação de MIB, chamada MIB-1 (IETF 1990), continha grupos de dados para identificação dos equipamentos, descrição de interfaces de rede dos mesmos e informações operacionais dos protocolos TCP, UDP, IP, ICMP e EGP. A evolução da MIB-I foi chamada MIB-2 (IETF 1991) e permitia acesso a um conjunto de informações mais abrangente acerca dos dispositivos gerenciados.

A coleção de objetos que constitui a MIB está organizada em grupos de objetos similares. Conforme já citado, tais objetos são descritos por identificadores de forma a construir uma estrutura hierárquica em árvore, onde cada nó é um número inteiro. Um objeto nesta estrutura é referenciado por um identificador único composto de uma sequência destes números separados por pontos. A
Figura 2 ilustra esta organização, exemplificando a localização do objeto 1.3.6.1.2.1.1 do grupo *system* na árvore da MIB.

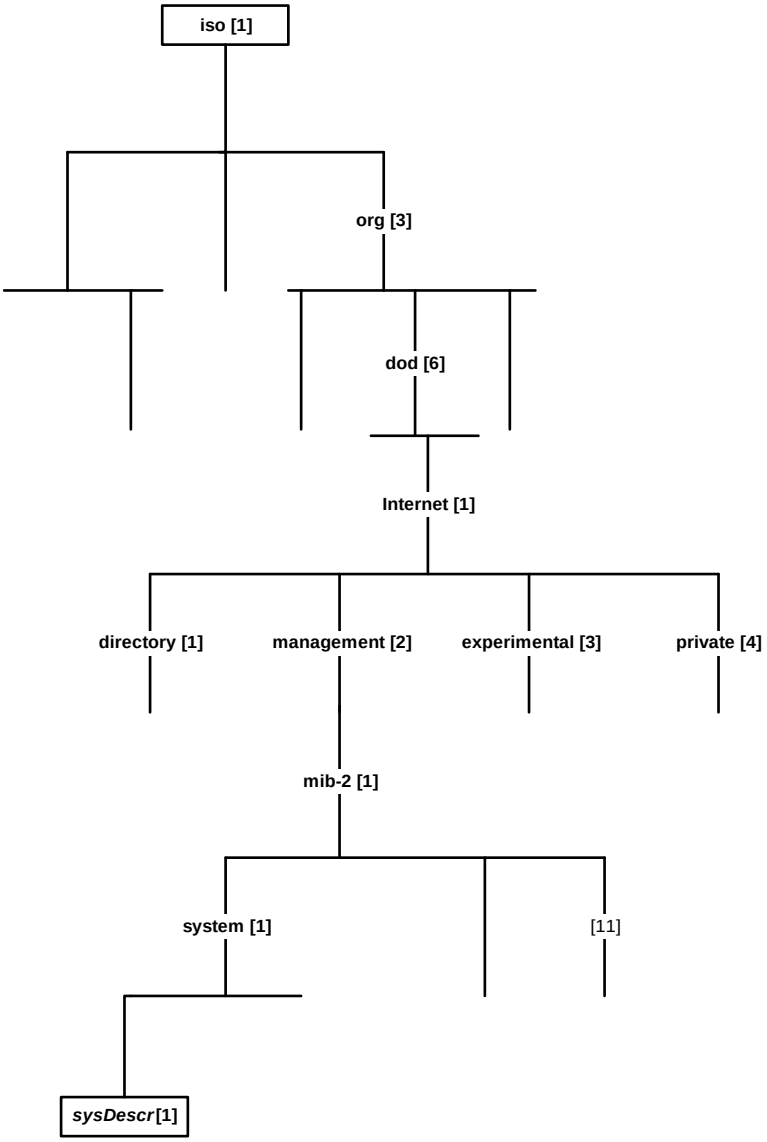


Figura 2 – Exemplo de localização de um objeto na árvore da MIB

Vale notar que uma instância de um objeto na MIB é descrita pela mesma sequência que define o objeto acrescida de um último número. Este terá valor zero se o objeto possuir somente uma instância ou um número maior que zero se existir mais de uma instância do objeto, sendo neste caso as instâncias organizadas em tabelas. No exemplo da Figura 2, o objeto *sysDescr* possui somente uma instância, logo a referenciamos como 1.3.6.1.2.1.1.0 .

Uma certa manipulação de dados na forma de tabelas é possível em SNMPv1. Uma tabela pode ser definida na MIB de um agente. O processo de criação e deleção de itens destas tabelas é feito por meio de operações denominadas *set*. Problemas podem ocorrer quando vários gerentes tentam criar ou eliminar itens numa mesma tabela. Em função disso, além de um campo *index* para indexar individualmente os itens, é necessário um campo em cada item chamado *status* que será o responsável em dizer qual o estado da linha (criada, deletada, apagada, etc...), para evitar inconsistências causadas pela manipulação dos dados por mais de um gerente.

Por outro lado, como resultado da evolução da MIB-I, a RFC 1213 (IETF 1991) define os objetos para a MIB padrão SNMP chamada de MIB-II. Esta MIB, total ou parcialmente, é implementada em produtos comerciais. Acréscimos podem ser feitos adicionando novos módulos da MIB (sejam novas padronizações definidas em RFC dos grupos de trabalho do IETF ou objetos proprietários criados por um determinado fabricante). Estes novos objetos são concatenados nas suas devidas posições na árvore, sem alterar de maneira nenhuma os objetos já existentes. Os objetos nunca são redefinidos, mesmo se um objeto for declarado obsoleto pelo IETF, sua identificação na árvore permanece.

Inúmeras novas MIB's foram padronizadas para necessidades específicas e tecnologias emergentes. Assim, uma MIB pode conter exatamente os objetos de interesse para um determinado equipamento ou componente da rede, tornando os dados um reflexo da realidade gerencial.

Deve-se notar que a informação de gerenciamento possui suas próprias regras de definição, o que torna as MIB's reservatórios de informação completamente desvinculados do protocolo de gerência. No princípio da definição do SNMP, pretendia-se facilitar a migração da infra-estrutura de informações de gerência do protocolo SNMP para um protocolo de gerência OSI. O tempo mostrou que isto não iria acontecer, mas esta separação facilitou a migração da versão 1 para novas versões.

### 2.1.4 VERSÕES

A função do protocolo de gerência é permitir a troca de informações entre as entidades que compõem o sistema de gerência da rede, no caso, gerentes e agentes. O protocolo ainda delineia o mecanismo de segurança usado pelo contexto administrativo. O protocolo SNMP possui poucas mensagens. Estas são operações básicas a serem executadas pelos gerentes e agentes, incluindo pedidos de informação, respostas a estes pedidos e notificações espontâneas de eventos nos dispositivos gerenciados.

O protocolo de transporte sugerido para transferência das mensagens SNMP é o UDP (IETF 1980). Cada mensagem SNMP deve estar totalmente contida num pacote UDP. Cada mensagem é uma ação de comunicação entre entidades SNMP. Para a versão 1 do protocolo foram padronizadas as seguintes operações (IETF 1990):

1. *GetRequest* – operação na qual o gerente pede o valor de determinados objetos na MIB do agente;
2. *GetNextRequest* – operação na qual o gerente pede o valor do próximo objeto na MIB do agente;
3. *SetRequest* – operação na qual o gerente altera o valor de um determinado objeto na MIB do agente;
4. *GetResponse* – operação que é a resposta ou confirmação do agente às mensagens anteriores;
5. *Trap* – operação que é a comunicação espontânea de um evento do agente para um gerente

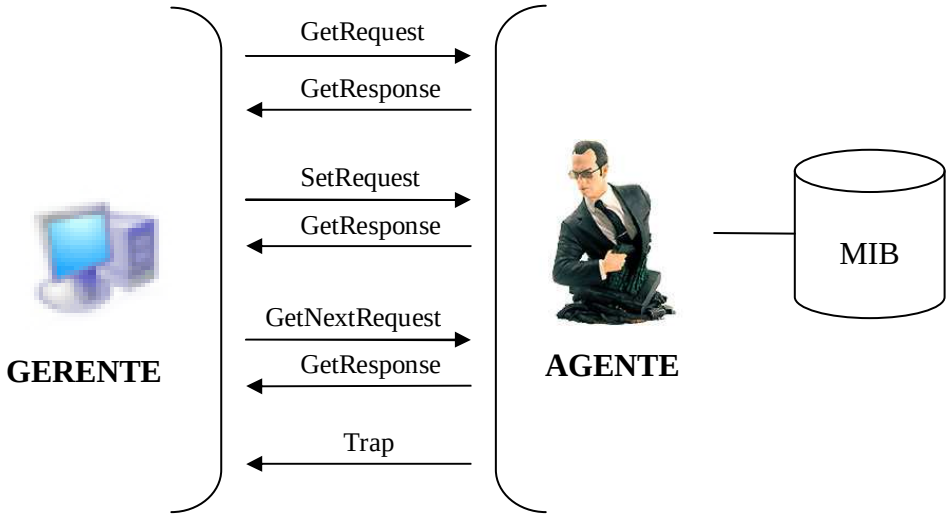


Figura 3 – Comunicação agente-gerente no SNMPv1

Para suprir algumas deficiências da primeira versão do SNMP, foi proposta pelo IETF o SNMPv2 (STALLINGS 1999). Esta versão permite a comunicação segura entre duas estações gerentes e entre os dispositivos gerenciados. O SNMPv2 faz a coleta de grandes volumes de dados gerenciados mais eficiente que a versão anterior e possui melhores mecanismos para os tratamentos de erros. As inovações da versão 2 do SNMP são descritas a seguir:

- A Estrutura da Informação Gerenciada (SMI) do SNMPv2 expande a SMI do SNMPv1 para incluir novos tipos de dados e melhorar a documentação associada a objetos. A SMI é agora dividida em quatro partes: definição de objetos, tabelas que proíbem a criação e a remoção de linhas pelo gerente e tabelas que permitem a criação e a remoção de linhas pelo gerente, e definições de notificações e módulos de informação;
- *GetBulkRequest* – Esta nova operação permite recuperar uma grande quantidade de informações do agente através de uma única mensagem SNMP;
- Comunicação gerente-gerente;
- Mecanismos de segurança que permitam a autenticação e a privacidade de uma mensagem SNMP. Esses mecanismos garantem que uma mensagem

SNMP não seja alterada, retardada ou repetida no percurso, que seja feita a identificação correta do usuário que enviou a mensagem, e que esta possa ser resguardada contra escuta clandestina;

- Utilização de mecanismos de transportes alternativos ao TCP/IP;

Um grande problema do SNMPv2 é sua incompatibilidade com os agentes da versão anterior, o que torna a transição de uma versão para outra mais custosa.

Em 1998, foi padronizada a versão 3 do protocolo SNMP que surgiu em função dos problemas de segurança das versões anteriores e com o objetivo de viabilizar a padronização de partes da arquitetura cujo consenso não tenha sido atingido nas versões anteriores. As novas características de segurança que o SNMP versão 3 trouxe à versão 2 do SNMP são:

- Novos e flexíveis métodos de segurança para as mensagens (criptografia das mensagens);
- Serviço para envio e recebimento de autenticações de mensagens.

## 2.2 SISTEMAS DISTRIBUÍDOS

Os sistemas distribuídos são caracterizados por um conjunto de serviços localizados em computadores independentes conectados por uma rede. Segundo TANENBAUM (1999), um sistema distribuído é aquele que executa processos em um conjunto de máquinas sem memória compartilhada, máquinas estas que aparecem como um único computador para seus usuários. Ele é formado por componentes de hardware e software localizados em computadores em rede que se comunicam e coordenam suas ações através de troca de mensagens.

A abordagem para comunicação em ambientes de software muito utilizada na prática ainda é a chamada remota de procedimentos (RPC - *Remote Procedure Call*), que foi concebida nos anos 70. Segundo TANENBAUM (1999), RPC é o método baseado no

paradigma cliente-servidor que permite aplicações chamarem procedimentos localizados em outras máquinas. Cada mensagem transferida inclui os parâmetros para a execução do procedimento e, se houver, os resultados da execução do mesmo. Nesse caso, os computadores que se comunicam combinam previamente quais são os procedimentos que podem ser chamados e quais são as mensagens que devem ser trocadas. A interação via RPC requer comunicação constante entre cliente e servidor. A execução do procedimento é dita síncrona, pois só acontece por estímulo da aplicação que realizou a chamada remotamente. Essa operação de troca de mensagens é invisível ao usuário do sistema.

### 2.2.1 OBJETOS DISTRIBUÍDOS

As linguagens de programação Orientada a Objetos são baseadas em três conceitos fundamentais: classes, herança e polimorfismo (CANTÙ 1998). A Análise e Projeto Orientados a Objetos introduzem diversos conceitos: classes, objetos, instância, atributos, métodos, abstração, encapsulamento, herança, persistência, relacionamento entre instâncias, acoplamento etc (AMBLER 1998).

Pela fusão de dois paradigmas (Sistemas Distribuídos e Orientação a Objetos) deu-se origem aos Objetos Distribuídos. Ele introduz um novo paradigma para integração de aplicações heterogêneas onde as aplicações são vistas como objetos que possuem interfaces bem definidas. A aplicação cliente acessa os objetos através de uma interface bem definida precisando, assim, preocupar-se apenas com a sua implementação. Como a interface de um objeto não muda, a aplicação cliente não precisa ser reconstruída quando o processamento interno de uma determinada aplicação servidora sofre alguma modificação evolutiva ou corretiva.

O paradigma intrínseco na computação de objetos distribuídos (DOC – *Distributed Objects Computing*) faz com que as aplicações sejam extremamente flexíveis. Os objetos podem ser reutilizados através dos mecanismos provenientes da OO. Como os sistemas distribuídos são baseados em componentes, menor esforço é exigido durante a sua manutenção.

## 2.3 CORBA

CORBA define uma especificação que permite aos objetos distribuídos comunicarem entre si de forma transparente, não importando onde eles estejam, em que plataforma ou sistema operacional estejam executando, em que linguagem de programação eles foram implementados e até mesmo qual protocolo de comunicação eles utilizam.

Por sua transparência de plataforma, CORBA vem sendo adotado principalmente para integração de sistemas legados. De acordo com EDWARDS *et al* (2003), atualmente quase todas as empresas cliente utilizam CORBA de alguma forma.

O CORBA 1.0 foi inicialmente especificado em outubro de 1991, sendo considerado como um produto intelectual do *Object Management Group*[5] (OMG), um consórcio de mais de 800 companhias das mais diferentes áreas (*IBM*, *Canon*, *DEC*, *Philips*, *Sun*, *Apple* etc, assim como grandes usuários como *Citicorp*, *British Telecom*, *American Airlines* etc) interessadas em prover uma estrutura padrão para o desenvolvimento independente de aplicações, usando técnicas de orientação a objeto em redes de computadores heterogêneas. Ao contrário do que muitos imaginam, a OMG produz especificações, e não implementações.

Introduzido como parte do CORBA 2.0 em agosto de 1996, o *Internet Inter-ORB Protocol* (IIOP) é um outro padrão OMG. Antes do IIOP, a especificação CORBA tratava somente da interação entre objetos distribuídos criados pelo mesmo fornecedor. Definindo o IIOP, a especificação CORBA torna-se a solução definitiva para a interoperabilidade entre objetos que não estão presos a uma plataforma, linguagem ou padrão específico.

### 2.3.1 ORB

O ORB é a estrutura mais importante da arquitetura CORBA. Ele possui a função de intermediar todas as transferências entre cliente e servidor, e fazer com que a transação

[5] http://www.omg.org

seja transparente para cada uma das partes durante todo o processo.

A Figura 4 mostra uma transação oriunda de uma aplicação (cliente) sendo enviada através do ORB a uma implementação de objeto CORBA localizada em uma outra aplicação (servidora). O ORB é responsável pela localização do objeto ao qual se destina a requisição, assim como, pelo envio dos parâmetros da requisição no formato aceito por este objeto. Também é função do ORB o retorno de parâmetros de saída da requisição para a aplicação cliente, se assim houver.

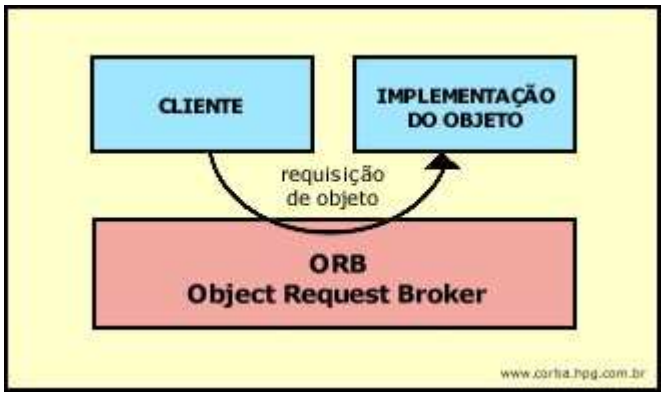


Figura 4 – Uma requisição enviada através do ORB[6]

A interface com a qual a aplicação cliente tem contato é totalmente independente de qualquer fator relacionado à heterogeneidade do ambiente distribuído no qual se encontra, como por exemplo: a localização do objeto e a linguagem de programação utilizada para sua implementação. Entretanto, ORB's diferentes podem apresentar características de implementação diferentes, resultando em serviços prestados a objetos e clientes com qualidades e propriedades diferentes.

### 2.3.2 A LINGUAGEM DE DEFINIÇÃO DE INTERFACE – IDL

CORBA fornece a IDL (*Interface Definition Language*), uma linguagem própria para definição de interfaces. A interface possui a lista de atributos e operações que um objeto deve prover. Um objeto implementa uma interface se ele pode ser alvo de qualquer uma

[6] http://www.corba.hpg.com.br

das solicitações descritas na interface. As solicitações originam-se da invocação de operações ou da obtenção ou determinação de atributos.

A sintaxe de IDL para interfaces é muito similar à das classes C++ e à das interfaces Java. Também estão disponíveis as estruturas, enumerações e uniões. A Figura 5 apresenta alguns exemplos de IDL.

```
/* Comentários seguem C++/Java (sem aninhamento) */
/* Uma estrutura (como em C/C++) e seus membros */
struct Struct_1 {
    char membro_1;
    boolean membro_2;
    float membro_3;
};

/* Uma interface (como em Java) e suas operações e atributos */
interface Interface_1 {
    /* operações têm sintaxe semelhante a C++/Java;
    argumentos podem ser "in", "out" ou "inout"; */
    long operacao_2 (in long a,inout long b,out long c);
    // um atributo gera operações "get/set" associadas;
    attribute string atributo_1;
};

/* Uma interface pode derivar de outra */
interface Interface_2 : Interface_1 {
    Interface_1 operacao_3 (in Struct_1 arg1,in Struct_1 arg2);
    // gera apenas operação "get" associada ao atributo;
    readonly attribute long atributo_2;
};
```

Figura 5 – Exemplos de sentenças IDL

A IDL é independente de linguagem de programação. Ela promove uma separação entre o processo de descrição de interfaces e o de implementação de objetos. Enquanto que as interfaces de objetos são declaradas em IDL, os objetos podem ser implementados em qualquer linguagem (ou combinação de linguagens). Esta é uma importante característica, quando se pensa no desenvolvimento para ambientes heterogêneos, pois nem todas as linguagens de programação são disponíveis em qualquer plataforma. 18

Clientes e implementações de objeto CORBA podem utilizar qualquer linguagem de programação. Para isto, interfaces IDL são traduzidas para construções semelhantes da linguagem utilizada (como classes C++ e interfaces Java), e invocações de operação resultam em chamadas de métodos ou funções.

### 2.3.3 STUBS E SKELETONS

Uma interface IDL, quando compilada, resulta em um *stub* e um *skeleton* para cada interface descrita na IDL. Esses dois elementos realizam a interface do ORB com, respectivamente, a aplicação cliente e a implementação do objeto (Figura 6).

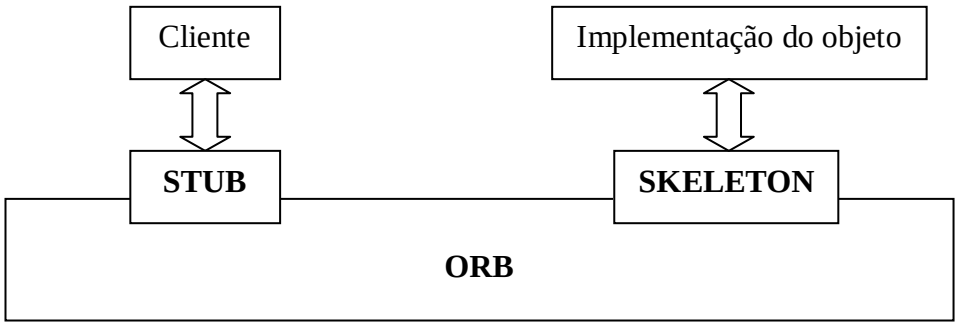


Figura 6 – As interfaces do cliente e da implementação do objeto com o ORB são feitas, respectivamente, pelos *stubs* e pelos *skeletons*.

Um *stub* é uma classe escrita na linguagem utilizada pelo cliente. O *stub* coopera com o ORB no intuito de "empacotar" uma solicitação do cliente para ser transmitida pela rede. O ORB se encarrega de obter a localização do objeto, e entrega a solicitação ao *skeleton* correspondente. O *skeleton* tem o papel complementar ao do *stub*, ou seja, coopera com o ORB para "desempacotar" uma solicitação feita pelo cliente.

Na verdade, nem sempre uma solicitação de um cliente trafega pela rede. É o caso de as aplicações cliente e servidor residirem na mesma máquina. De qualquer forma, estes detalhes ficam transparentes ao programador.

*Stubs* e *skeletons* são gerados através de tradutores de IDL para a linguagem desejada, e incorporados diretamente no código do cliente e da implementação do objeto. Por

exemplo, caso um objeto CORBA seja implementado em C++, deverá ser gerado um *skeleton* C++. Se um cliente deste objeto for desenvolvido em Smalltalk, deve ser gerado, a partir da definição da interface em IDL, um *stub* Smalltalk. Desta maneira, o cliente faz uma solicitação à implementação do objeto através de uma invocação do método correspondente no *stub*. O *skeleton* recebe a solicitação do ORB e faz a invocação do método adequado na implementação do objeto.

### 2.3.4 PROTOCOLOS DE COMUNICAÇÃO

A comunicação entre objetos cliente e servidor é realizada, de forma transparente, entre seus ORB's. Cada ORB fornece uma API de serviços, utilizada pelos *Stubs* e *Skeletons* destes objetos. Os *stubs* e *skeletons* funcionam como uma "cola" entre os clientes, servidores e os ORB's, que realizam enviam as requisições de um nó a outro.

O protocolo empregado na comunicação entre ORBs é padronizado pelo OMG. O GIOP (*General Inter-ORB Protocol*) define um conjunto de primitivas de comunicação, independentes das camadas de transporte e de rede, que permite a comunicação entre ORB's. O protocolo IIOP (*Internet Inter-ORB Protocol*) é a implementação deste protocolo para redes baseadas nos protocolos TCP/IP, sendo obrigatório para as implementações CORBA. Os tipos de dados especificados em IDL são representados no formato CDR (*Common Data Representation*), antes de serem transportados pela rede. O CDR normaliza a transferência de dados entre clientes e servidores que executam em plataformas de hardware diferentes.

O modelo de objetos CORBA é baseado no conceito de referências de objetos. Uma referência de objeto identifica unicamente um objeto local ou remoto. Clientes só podem realizar invocações de operações nestes objetos estando de posse destas referências (HENNING 1998). Instâncias de servidores CORBA são representadas por *Interoperable Object References* (IOR's), estas referências são geradas pelo adaptador de objetos e permanecem opacas para clientes e servidores. Internamente, o formato de uma IOR é extensível, possuindo uma estrutura composta por espaços reservados para informações de cada protocolo, estes dados são usados por ORB's de clientes para

localizar servidores em ORB's locais ou remotos. Em algumas implementações, a IOR inclui o endereço de rede de um objeto CORBA.

O Adaptador de Objetos (*Object Adapter*) é o componente responsável pela criação de IOR's. Ele implementa a ativação e desativação de servidores; sendo responsável em direcionar as requisições a estes servidores. O adaptador de objetos utilizado inicialmente pelo CORBA era o BOA (*Basic Object Adapter*), mas por falta de portabilidade entre ORB's de diferentes fabricantes, a OMG padronizou o POA (*Portable Object Adapter*) em 1998, na especificação CORBA 2.2. No POA, os *skeletons* gerados por diferentes compiladores IDL de diferentes fabricantes e sua interação com o ORB foram padronizados.

### 2.3.5 APLICAÇÕES DO CORBA

Atualmente, a tecnologia CORBA vem sendo utilizada em diversas áreas no Brasil, principalmente Telecomunicações e Saúde. Segue abaixo, alguns casos de utilização do CORBA no Brasil:

- PACS: O Instituto do Coração do Hospital das Clínicas da FMUSP iniciou em 1998 o projeto PACS (*Picture Archiving and Communication Systems*). O objetivo deste projeto é construir um ambiente distribuído para transmissão, arquivamento, processamento e visualização de imagens médicas, integrando-as aos sistemas de informações do Hospital das Clínicas da Faculdade de Medicina da USP e do Instituto do Coração (InCor). Tendo que integrar informações de sistemas distribuídos e heterogêneos, o Instituto do Coração escolheu a tecnologia CORBA como base para a construção deste sistema. O projeto PACS está em produção desde abril de 2000 e foi integrado ao Sistema de Informações Hospitalares (HIS) da USP em 2001.

  Durante o desenvolvimento do projeto foram criadas duas especificações: o Serviço de Identificação de Pacientes e o Serviço de Acesso a Observações

Clínicas. Elas permitirão que o Instituto do Coração troque informações entre outras instituições de saúde que utilizem CORBA.

- A empresa Disoft[7] possuía um produto desenvolvido em Visual Basic utilizando o banco de dados SQL Server. Como o sistema não estava mais atendendo às necessidades de velocidade, capacidade de processamento, escalabilidade e suporte à Internet, a empresa decidiu criar, em 1998, um novo sistema baseado em CORBA e Java. A comunicação entre as aplicações clientes (*applets* Java) e a aplicação servidora é feita através de CORBA (produto *OrbixWeb*).

  Como resultado, na mesma plataforma Windows NT e com o mesmo banco de dados SQL Server, o sistema ficou cerca de 10 vezes mais rápido do que o original. Adicionalmente, o produto ganhou independência de plataforma, de banco de dados, flexibilidade e escalabilidade.

- A OpenComm do Brasil desenvolveu dois produtos em que desempenho e uma arquitetura robusta para gerência de um sistema distribuído eram fundamentais: O primeiro deles é a Plataforma Multiserviços - *Open Telephony Server*. Este sistema é uma plataforma de atendimento automático de ligações para serviço com perfil de chamadas massivas com as seguintes características: entroncamento com a rede pública comutada usando mais de 3000 linhas (100 troncos E1), plataforma e controle de chamadas distribuídos, múltiplas camadas, processamento on-line, banco de dados centralizado, utilizando CORBA (produto *OmniORB2r*) como tecnologia para integração dos diversos componentes e dentro do servidor de telefonia. O segundo produto é o sistema *OpenAgent*, um software integrado para *CallCenter* que utiliza CORBA e uma implementação da especificação JTAPI (*Java Telephony API*). Ambos os produtos foram desenvolvidos em C++, utilizando o OmniORB.

  Como resultado, a utilização de CORBA nestes sistemas apresentou um desempenho excelente, comportamento estável, além de permitir a integração de

[7] http://www.disoft.com.br

software distribuído entre os diferentes ambientes que faziam parte do projeto (Linux, Solaris e Windows).

- SIGA: Desenvolvido no NCE, o Sistema Integrado de Gestão Acadêmica – SIGA tem por objetivo integrar todos os sistemas de registro acadêmico da UFRJ. No modelo anterior, estes sistemas eram separados, um para a Graduação e dois para a Pós-graduação. O sistema é acessível à comunidade acadêmica em geral. Diferentemente do anterior (restrito às secretarias acadêmicas), aluno e professor têm acesso às funções como inscrição em disciplinas, lançamento de notas, boletim e histórico escolar.

  Esta abrangência influencia na quantidade de usuários. São 60.000 alunos, mais de 3.000 professores e milhares de funcionários. Por isso, o novo sistema foi desenvolvido inteiramente no formato Web.

### 2.3.6 VANTAGENS

A principal vantagem do uso de CORBA é sua capacidade de permitir o desenvolvimento de sistemas distribuídos de maneira independente de sistema operacional, linguagem de programação, protocolos e topologia de rede.

Esta arquitetura provê, ainda, um conjunto de serviços abertos e padronizados (*CORBAServices* e *CORBAFacilities*) que facilitam a implementação de sistemas distribuídas. CORBA é empregado também na integração de sistemas legados. Isto permite a utilização de aplicações legadas por clientes escritos em linguagens de alto nível, sem que haja necessidade de reescrita total de tais sistemas para plataformas de software e hardware diferentes.

O CORBA permite integração com outras tecnologias de sistemas distribuídos. A especificação do RMI-IIOP, por exemplo, permite que objetos remotos RMI utilizem o protocolo IIOP sem exigir complexas modificações. Além disso, objetos CORBA escritos em Java não perdem a flexibilidade e vantagens da linguagem Java e do RMI,

como. A OMG tem publicado uma especificação de interoperabilidade entre COM e CORBA.

### 2.3.7 DESVANTAGENS

A integração de clientes CORBA com servidores implementados para ORBs de fabricantes diferentes ainda não é muito trivial. Isso se deve principalmente as diferenças relacionadas aos padrões implementados por cada ORB. Padronizações como a POA (*Portable Object Adapter*), contudo, foram realizadas no sentido de reduzir esta limitação.

CORBA exige um elevado custo de aprendizagem. A compreensão dos conceitos do ORB, IDL, adaptador de objetos com sua posterior utilização bem sucedida exige conhecimento avançado em programação orientação a objetos. A especificação do RMI-IIOP, contudo, trouxe maior facilidade na implementação de objetos CORBA para Java, não sendo necessário, neste caso em particular, uso de IDL's.

## 2.4 PORTABLE INTERCEPTORS

*Portable Interceptors* (PI) representam não apenas uma importante característica do CORBA, mas um elemento fundamental deste trabalho. PI's foram oficialmente incluídos na especificação CORBA 2.5 em setembro de 2001.

A idéia principal de *Portable Interceptors* consiste em permitir que os desenvolvedores registrem objetos no ORB. Esses objetos são *automaticamente* invocados quando determinados eventos ocorrem durante a execução das chamadas CORBA. Isso significa que a funcionalidade provida pelo ORB pode ser estendida sem a necessidade de alteração na sua implementação e no código-fonte do sistema distribuído.

### 2.4.1 CATEGORIAS

Existem duas categorias de PI: *Request Interceptors* e *IOR Interceptors*. Elas serão apresentadas a seguir:

- **Request Interceptors**

  Os *intecerptors* desta categoria recebem eventos a partir do início da invocação de uma requisição até o seu término. Existem dois tipos de *Request Interceptors*: *Client Request Interceptor* e *Server Request Iinterceptor*.

  A Figura 7 mostra os pontos chamados pelo ORB quando uma requisição é realizada.

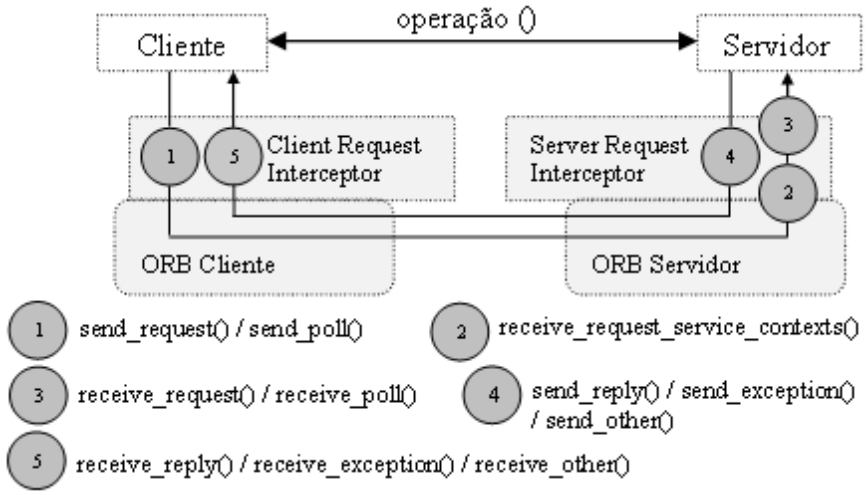


Figura 7 – Métodos dos *Request Interceptors* invocados pelo ORB.

Quando uma chamada remota é feita pela aplicação cliente, o ORB Cliente invoca o método *send_request* de cada *Client Request Interceptor* registrado nele, de forma transparente para o desenvolvedor e para a aplicação cliente. Quando a requisição chega ao servidor, o ORB os métodos *receive_request-_service_contexts* e *receive_request* antes que a requisição seja processada.

- **IOR Interceptors**

  Os *interceptors* dessa categoria têm o objetivo de controlar o ciclo de vida dos objetos CORBA.

### 2.4.2 APLICAÇÕES

Um trabalho realizado por NARASIMHAN, MOSER e MELLIAR-SMITH (1999) apresentou algumas possíveis aplicações para *Portable Interceptors*:

- Adaptação de Protocolos – As requisições podem ser enviadas através de um protocolo diferente do IIOP (e.g. SOAP e DCOM). Isso é bastante útil quando aplicações CORBA devem se comunicar com sistemas legados que utilizam algum protocolo proprietário. Portanto, a comunicação poderia ser feita de forma transparente entre os sistemas;
- Monitoramento – O desempenho dos objetos CORBA pode ser monitorado de forma transparente para os desenvolvedores, sem que código-extra seja inserido no desempenho do sistema;
- Segurança – PI's pode ser usado para autenticação;
- Confiabilidade – PI's podem ser usados para replicação automática de objetos e detecção de falhas.

De acordo com FRIEDMAN e HADAD (2001), as aplicações CORBA podem prover as seguintes funcionalidades quando *Portable Interceptors* são aplicados apenas no lado cliente: *Caching* de objetos, Controle de Fluxo e Distribuição de Carga.

Além disso, eles podem ser usados para geração automática de documentação, como foi visto através do projeto *CorbaTrace* (CORBATRACE, 2003) no capítulo 1.

### 2.4.3 VANTAGENS E LIMITAÇÕES

A transparência representa a sua principal vantagem. Porém, observar-se algumas limitações:

- Limitações com a linguagem Java – MARCHETTI *et al* (2001) comenta que determinadas informações das transações não podem ser obtidas por um *Portable Interceptor* construído em Java. Porém, uma técnica que será apresentada na seção seguinte permite resolver essa limitação.

- Interoperabilidade – FICHE e OLIVEIRA (2004) concluem que há diferenças na implementação de dois ORB's: *OpenORB* e *VisiBroker*. De acordo com os autores, o *OpenORB* não executa um *Client Request Interceptor* quando este é instalado na aplicação servidora, mas o *Visibroker* invoca. Essas diferenças podem prejudicar a interoperabilidade de sistemas construídos baseados *Portable Interceptors*.

### 2.4.4 TÉCNICAS

MARCHETTI *et al* (2001) propõe algumas técnicas que podem ser implementadas com *Portable Interceptors*:

- *Piggybacking* – Permite que *Client Request Interceptors* e *Server Request Interceptors* troquem informação entre si. Como mostra a Figura 8, a informação pode ser enviada junto com a requisição e a sua resposta.

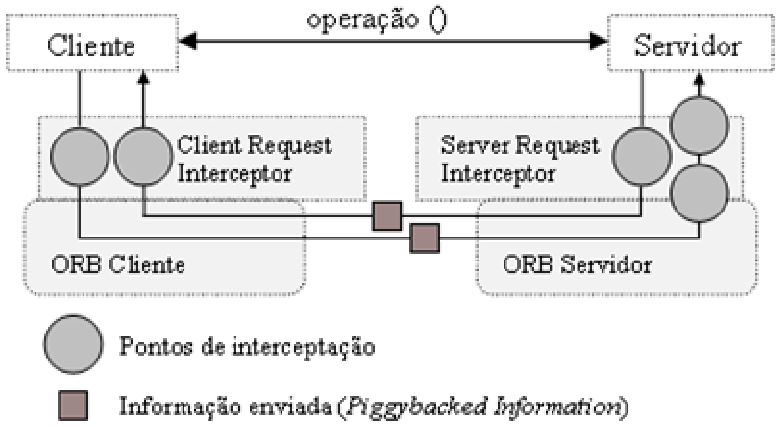


Figura 8 – Técnica *Piggybacking*.

- Redirecionamento – Essa técnica pode ser usada na implementação de mecanismos de tolerância a falha e distribuição de carga. Se um servidor estiver com problemas (e.g. consumo excessivo de memória), as requisições atendidas por ele podem gerar falhas. Para evitar isso, elas podem ser redirecionadas a outro servidor que tenha condições de atendê-las. Além disso, as requisições

podem ser distribuídas igualmente entre um conjunto de servidores a fim de otimizar a utilização dos seus recursos. Enfim, essas técnicas podem ser usadas para prover confiabilidade, desempenho e escalabilidade às aplicações CORBA.

- *Proxy* – Pode ser usada para superar as limitações de *Portable Interceptors* em Java descritas na seção anterior. Essa idéia consiste em redirecionar as transações para um *object proxy* s*erver* (GAMMA 1998 *apud*: MARCHETTI *et al* (2001)).

### 2.4.5 CUSTO

MARCHETTI *et al* (2001) realizou, também, uma análise de desempenho de *Request Interceptors*. Os experimentos foram baseados nas implementações *JacORB*, *ORBacus* e *Orbix*. Em cada experimento, um conjunto de 10.000 requisições foi processado 20 vezes. A latência foi calculada para cada requisição.

No primeiro experimento, nenhum *interceptor* foi instalado, ou seja, ele serviu como base para o cálculo dos custos. Em todos os outros experimentos, um *Request Interceptor* foi instalado apenas no ORB da aplicação cliente (i.e. *Client Request Interceptor*). Porém, no segundo experimento, ele não tinha nenhuma implementação. A tabela 1 mostra os resultados para os dois primeiros experimentos. Por exemplo, o tempo de resposta de uma requisição teve, em média, um acréscimo de 0.1 ms com o uso de um *Client Request Interceptor* (sem implementação) para o JacORB.

| Experimento / ORB | JacORB | ORBacus | Orbix |
|---|---|---|---|
| Experimento base (sem o uso de *interceptors*) | 1 | 1.1 | 1.44 |
| *Client Request Interceptor* (sem implementação) | 1.1 | 1.17 | 1.46 |
| Custo | 0.1 | 0.07 | 0.02 |

Tabela 3 – Latência média (em ms) para os dois primeiros experimentos.

O terceiro experimento teve o objetivo de medir o custo da técnica de redirecionamento permanente (i.e. por cliente). Os resultados mostraram que os custos foram bastante semelhantes aos do segundo experimento. O quarto experimento teve o objetivo de medir o custo da técnica de redirecionamento não-permanente (i.e. por requisição). A tabela 2 mostra a média das latências e o custo desta técnica.

| Experimento / ORB | JacORB | ORBacus | Orbix |
|---|---|---|---|
| Redirecionamento não-permanente | 8.8 | 2.04 | 2.45 |
| Custo | 7.8[8] | 0.94 | 1.41 |

Tabela 4 – Latência média (em ms) para o quarto experimento.

### 2.4.6 MECANISMOS RELACIONADOS

- *Smart Stubs* (ou *Smart Proxies*) – Esse é mecanismo simples e consiste em criar um outro *stub* (*smart stub*) que estende a funcionalidade do *stub* original. Dessa forma, o *smart stub* pode acrescentar mecanismos de monitoramento de desempenho, *caching* etc. A Figura 9 mostra como comunicação remota é feita com o uso desse mecanismo. Na verdade, a aplicação cliente interage com o *smart stub* que repassa a chamada ao *stub* original. O trabalho de WANG *et al* (2000) conclui que *Portable Interceptors* representam um mecanismo mais eficiente e robusto do que *smart stub*, que não faz parte da especificação CORBA e pode ser aplicado apenas no início da chamada remota, em especificas interfaces cliente.

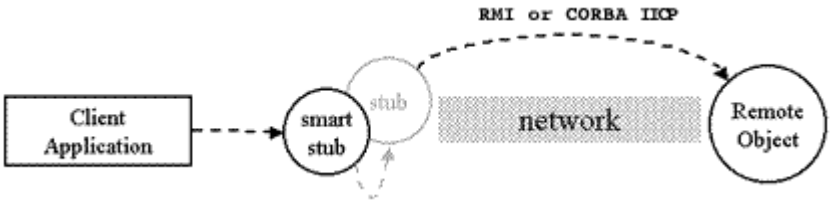


Figura 9 – Comunicação remota com o uso de *Smart Stubs*.

[8] O custo foi relativamente alto em função de uma falha encontrada no JacORB.

- *Custom Marshaling* – Esse é um mecanismo do COM (*Component Object Model*) similar ao *Portable Interceptors*.

# 3 ARQUITETURA

*Neste capítulo, o autor apresenta a arquitetura da solução, descrevendo os seus componentes. Logo em seguida, ele explica como eles funcionam juntos para monitorar o desempenho das aplicações distribuídas baseadas em CORBA.*

## 3.1 INTRODUÇÃO

O termo "Arquitetura" refere-se à estrutura organizacional e comportamental de um determinado software. Uma arquitetura de software engloba:

- As decisões significativas sobre a organização do software.
- A seleção dos elementos estruturais e suas interfaces pelas quais o software é composto juntamente com os comportamentos especificados na colaboração entre estes elementos.

A arquitetura do NISUS baseia-se nos requisitos necessários para o gerenciamento de desempenho de sistemas distribuídos discutidos no capítulo anterior. Ao mesmo tempo, a arquitetura combina as principais técnicas de monitoramento de desempenho tradicionais: monitoramento do tráfego de rede, monitoramento no nível de sistema operacional, monitoramento do lado cliente da aplicação e monitoramento amplo do sistema.

Porém, a instrumentação de código não é utilizada devido aos problemas já mencionados. A arquitetura é baseada em *Portable Interceptors*. Com isso, os seus componentes são automaticamente instalados assim que as aplicações (clientes e servidoras) iniciam. Além disso, o NISUS pode ser acoplado a qualquer sistema distribuído CORBA já existente sem que o seu código-fonte tenha que ser modificado. Portanto, o NISUS pode ser aplicado facilmente a qualquer sistema distribuído CORBA escrito em Java sem exigir extensas customizações.

Como mostrado na Figura 10, a arquitetura do NISUS inclui um console Web e um conjunto de componentes. Os componentes atuam como uma extensão da funcionalidade do ORB, tanto no lado cliente quanto no servidor. Sendo assim, eles localizem-se no mesmo contexto de memória das aplicações clientes e servidoras.

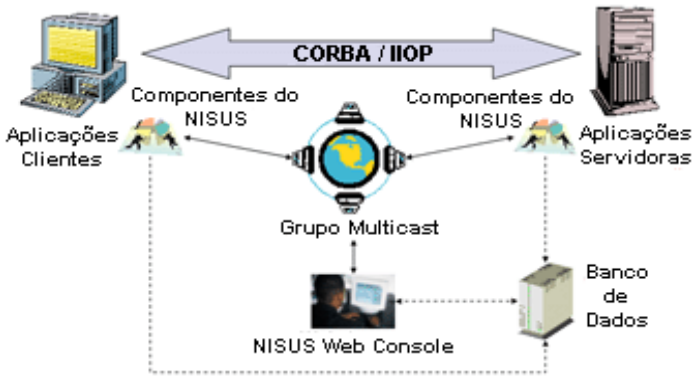


Figura 10 – Arquitetura do NISUS.

Com o NISUS, o processo de gerenciamento de sistemas distribuídos torna-se centralizado e transparente. A transparência, no contexto aqui empregado, significa que tanto os desenvolvedores quanto os administradores não precisam conhecer os componentes da arquitetura. Portanto, durante o processo de desenvolvimento de um novo sistema, os desenvolvedores podem concentrar-se mais com a implementação das funcionalidades oferecidas aos usuários.

Além disso, não se deve permitir que a complexidade da dispersão geográfica inerente aos sistemas distribuídos dificulte o processo de gerenciamento. Por esta razão, a arquitetura inclui, ainda, um mecanismo de comunicação *multicast* entre o console Web e o restante dos componentes instalados nas aplicações clientes e servidoras. O canal de comunicação *multicast* esconde dos administradores e desenvolvedores o fato de o sistema distribuído possuir na verdade diversas aplicações cliente e servidora. Portanto, eles podem gerenciar os sistemas distribuídos como se estes fossem sistemas bem simples, monousuário.

O canal de comunicação é capaz de prover ainda funções como:

- Envio de instruções de gerenciamento – Se uma das máquinas servidoras, por exemplo, estiver com algum problema grave, o administrador pode enviar um comando pedindo que a máquina seja reiniciada.
- Pedido de informação sobre desempenho e utilização de recursos – O NISUS possui componentes responsáveis em prover dados sobre o uso de determinados recursos. Portanto, o administrador pode requisitar dados sobre o consumo de CPU, memória etc.
- Controle e configuração dos componentes – O administrador pode modificar os parâmetros de configuração dos componentes facilmente, sem requerer que as aplicações (clientes e servidoras) sejam reiniciadas.

## 3.2 ARMAZENAMENTO DE DADOS CENTRALIZADO versus DISTRIBUÍDO

Para não causar um aumento no tempo de resposta do sistema, o mecanismo de gravação de dados do NISUS é assíncrono, ou seja, os dados não são gravados imediatamente após serem coletados.

Quanto maior for o número de usuários que estiverem usando o sistema simultaneamente, maior será a quantidade de chamadas feitas ao sistema e, conseqüentemente, maior será a quantidade de dados coletados. Ou seja, se o sistema estiver experimentando uma carga elevada de usuários, o NISUS poderá consumir muita memória para armazenar os dados. Por esta razão, o mecanismo de coleta de dados combina duas estratégias: centralização e distribuição.

O NISUS evita que quantidade excessiva de dados esteja em memória. Para isso, o mecanismo é capaz de guardá-los localmente em disco e, quando o sistema estiver ocioso, ele os enviará para o banco de dados.

## 3.3 COMPONENTES

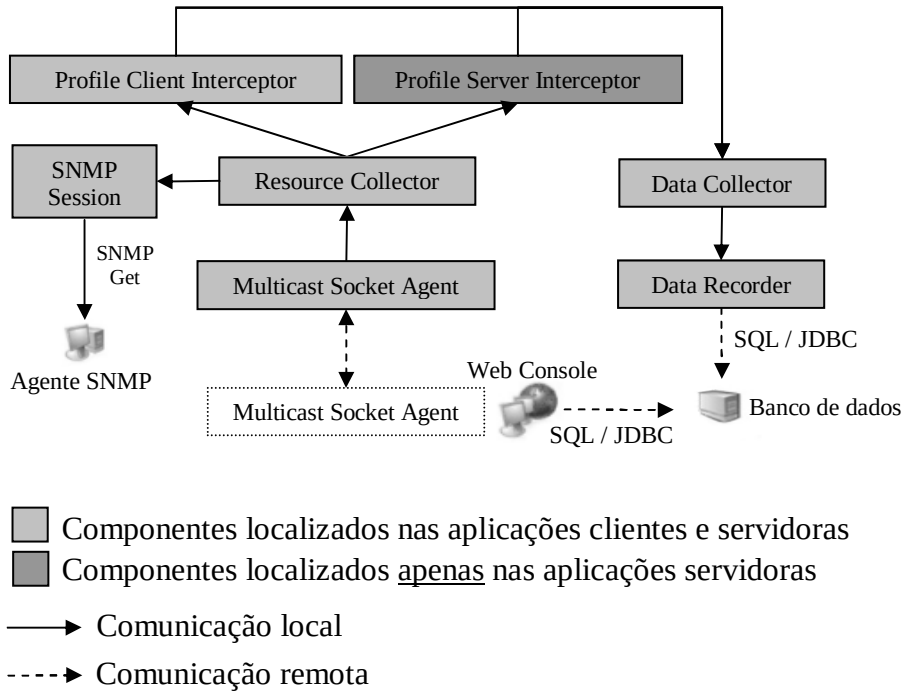


Figura 11 – Componentes do NISUS.

Os componentes estão presentes no mesmo contexto de memória das aplicações (clientes e servidoras), ou seja, um componente está no mesmo contexto de memória que o outro. Por isso, a comunicação entre eles ocorre sempre localmente. Além disso, como a Figura 11 mostra, o componente *Profile Server Interceptor* deve ser instalado apenas na aplicação servidora.

As setas mostradas na Figura 11 indicam que um componente invoca o outro. No NISUS, assim como em outras infra-estruturas baseadas em componentes, um componente tem interação com um outro para obter ou repassar alguma informação necessária; ou apenas para delegar responsabilidades. Por exemplo, o componente *Data Collector* invoca o componente *Data Recorder*. Isso pode sugerir que o primeiro guarda os dados coletados e transfere, em um determinado momento, a responsabilidade de gravá-los no banco de dados para o segundo. O componente *SNMP Session*, por sua vez, invoca o agente SNMP instalado na máquina que hospeda a aplicação (cliente ou servidora).

### 3.3.1 PROFILE CLIENT INERCEPTOR E PROFILE SERVER INTERCEPTOR

Os componentes *Profile Client Interceptor* e *Profile Server Interceptor* são *CORBA Request Interceptors* (Capítulo 2). O primeiro represente um *Client Request Interceptor* e o segundo um *Server Request Interceptor*. Portanto, o componente *Profile Server Interceptor* pode ser instalado apenas nas aplicações servidoras.

Ambos têm a responsabilidade de gerenciar o desempenho dos objetos CORBA do sistema acompanhando os passos executados durante as chamadas realizadas. Por esse motivo, eles são considerados os principais componentes da arquitetura do NISUS.

Enquanto o componente *Profile Server Interceptor* pode ser instalado apenas nas aplicações servidoras, o componente *Profile Client Interceptor* pode ser instalado tanto nas aplicações clientes quanto nas servidoras. Na verdade, existem algumas combinações de configuração desses componentes que devem ser compreendidas:

- *Profile Client Interceptor* instalado apenas nas aplicações cliente – Essa configuração permite apenas monitorar o tempo de resposta da aplicação, pois nenhum *interceptor* é instalado nas aplicações servidoras.
- *Profile Client Interceptor* instalado apenas nas aplicações cliente e o *Profile Server Interceptor* instalado nas aplicações servidoras – Essa configuração tem o mesmo objetivo da primeira e permite, também, monitorar o tempo de rede das chamadas.
- *Profile Client Interceptor* instalado tanto nas aplicações cliente quanto nas servidoras e o *Profile Server Interceptor* instalado nas aplicações servidoras – Essa configuração tem o mesmo objetivo das outras e permite, também, monitorar as chamadas realizadas dentro das transações, ou seja, com essa configuração, o NISUS torna-se capaz de monitorar as sub-transações, obtendo a seqüência de invocações CORBA realizadas desde o início da transação até o seu fim.

Vale observar que a seguinte configuração não é possível: apenas o componente *Profile Client Interceptor* instalado tanto nas aplicações cliente quanto nas servidoras. Como será visto mais adiante, um identificador da transação é transmitido até o componente *Profile Server Interceptor*. Portanto, com essa configuração, não seria possível identificar a transação, ou seja, não teria a qual transação associar as sub-transações monitoradas.

Pela Figura 11, pode-se observar que eles invocam o componente *Data Collector*. Isto significa que os dados são, primeiramente, armazenados em um coletor para serem, posteriormente, gravados em um banco de dados.

### 3.3.2 DATA COLLECTOR E DATA RECORDER

O componente *Data Collector* tem como objetivo acumular os dados coletados pelos dois *Request Interceptors*. Ele mantém uma estrutura interna capaz de armazenar, em memória, os dados coletados. O componente *Data Recorder*, por sua vez, é responsável pela gravação dos mesmos no banco de dados.

O componente *Data Collector* não grava os dados imediatamente. Ele acumula as informações e, de tempos em tempos (*record interval time*), executa o seu procedimento de gravação. Quando isso acontece, as informações são passadas ao componente *Data Recorder* que as envia de uma só vez (em lote) ao banco de dados para evitar *overhead* de rede. Porém, ele recebe um número máximo (*threshold*) de informações de cada vez. Isso é feito para evitar que, quando uma quantidade excessiva de informações for armazenada, o procedimento de gravação consuma muito processamento da máquina (cliente ou servidora) transmitindo ao banco de dados uma quantidade muito grande de informações.

Se o parâmetro *record interval time* for pequeno demais, muitas chamadas ao banco de dados serão feitas para efetuar a gravação de todos os dados coletados. Isto poderá afetar o desempenho da rede e das máquinas. Por outro lado, se o *record interval time*

for alto demais ou se a quantidade de usuários no sistema for grande, o componente *Data Collector* irá armazenar dados demais na memória. Para evitar esta situação, o componente *Data Collector* é capaz de gravar os dados na máquina local quando a quantidade for maior do que um determinado limite (superior a 2**threshold*, por exemplo). E, quando a aplicação estiver mais ociosa, o componente *Data Collector* recupera os dados armazenados localmente e os envia ao banco de dados. Portanto, o NISUS é capaz de proteger o tráfego de rede e o consumo de memória e processamento das máquinas (clientes e servidoras).

O intervalo de tempo (*record interval time*) em que o procedimento de gravação é executado, o limite de informações a serem gravadas (*threshold*) e o limite de informações em memória em cada execução do procedimento são parâmetros configuráveis como será visto mais adiante.

### 3.3.3 SNMP SESSION

Esse componente tem, como objetivo principal, fornecer dados sobre a utilização de recursos de hardware das máquinas. Ele implementa, na verdade, a técnica "monitoramento ao nível de sistema operacional". Dentre outras, as seguintes informações podem ser fornecidas:

- Consumo de memória das máquinas;
- Utilização de CPU;
- Lista de processos em execução;
- Capacidade total de memória.

### 3.3.4 RESOURCE COLLECTOR

O componente *Resource Collector* tem o objetivo de fornecer aos demais componentes informações sobre o consumo de recursos da aplicação (cliente e servidora). Ele chama o componente *SNMP Session* de tempos em tempos (a cada minuto, por exemplo) para obter dados sobre a utilização dos recursos mencionados na seção anterior e os repassa

aos componentes *Profile Client Interceptor* e *Profile Server Interceptor*. Dessa forma, essas informações são gravadas juntamente com os dados coletados sobre as transações, sendo possível, por exemplo, saber qual o consumo de memória e CPU no momento que uma determinada transação foi executada.

### 3.3.5 MULTICAST SOCKET AGENT

Este componente é responsável pelo envio e recebimento das mensagens ao canal de comunicação *multicast*. Além dos clientes e servidores, o NISUS Web Console também possui uma instância desse componente. Portanto, é através dele que os administradores enviam mensagens às aplicações clientes e servidores.

## 3.4 COMO OS COMPONENTES TRABALHAM JUNTOS?

Para compreender detalhadamente como os componentes do NISUS trabalham em conjunto, iremos considerar o mais simples cenário cliente-servidor: uma única aplicação cliente realizando chamadas (transações) a uma única aplicação servidora.

Quando a aplicação cliente realiza uma chamada à aplicação servidora, o componente *Profile Client Interceptor* é automaticamente invocado pelo ORB. Ele obtém o tempo (*timestamp*) de chamada e cria um identificador global único (GUID) para a transação. O ORB envia ao servidor, então, o identificador criado juntamente com os dados (parâmetros) da requisição através do mecanismo *piggybacking* (Capítulo 2).

Quando a requisição chega à aplicação servidora, o ORB invoca o componente *Profile Server Interceptor*. Este, por sua vez, obtém o tempo de chegada e o identificador da requisição enviado pela aplicação cliente. Desta forma, o NISUS torna-se capaz de medir o tempo levado para a transação ir da aplicação cliente até a servidora. A chamada é passada, então, para o objeto CORBA. Quando o processamento da transação termina, o ORB invoca novamente o componente *Profile Server Interceptor* que obtém o tempo de término da execução. Nesse momento, o componente *Data Collector* recebe os dados coletados para serem gravados em um banco de dados. A seguir, ORB envia a resposta à aplicação cliente.

Quando a resposta chega à aplicação cliente, o ORB invoca novamente o componente *Profile Client Interceptor*. Este, por sua vez, obtém o tempo de chegada da resposta. Dessa forma, o NISUS torna-se capaz de medir o tempo levado para a resposta ir da aplicação servidora até a cliente. Por fim, o componente *Data Collector* recebe os dados coletados para serem gravados em um banco de dados.

Pode-se observar que os dados coletados são gravados no banco de dados em dois momentos:

- Quando a transação termina de ser processada pela aplicação servidora;
- Quando a resposta chega à aplicação cliente

Pode-se observar, ainda, que o NSUS é capaz de obter as seguintes informações sobre as transações:

- Tempo de resposta das transações (latência);
- Tempo de execução (ou processamento) da requisição;
- Tempo de rede levado para a transação ir da aplicação cliente para a servidora (ida);
- Tempo de rede levado para a resposta ir da aplicação servidora para a cliente (volta)

## 3.5 NISUS WEB CONSOLE

O *Web Console* é o terminal gráfico com o qual desenvolvedores e administradores podem gerenciar o desempenho além de falhas ocorridas e recursos utilizados pelo sistema. Ele permite, ainda, a configuração dos componentes, definição de filtros etc.

DE PAUW *et al* (1998) *apud:* MOE e CARR (2001) apresentam duas abordagens interessantes para gerenciamento de desempenho de sistemas orientados a objeto: microscópica e macroscópica. A abordagem macroscópica refere-se à análise das

informações em um nível mais alto, sem muito detalhamento. Ao contrário, a abordagem microscópica concentra-se na seqüência de mensagens, ou seja, em um nível mais baixo. Segundo o autor, nenhuma abordagem é melhor do que a outra, mas as duas são importantes para o gerenciamento de desempenho de sistemas.

O *Web Console* se baseiam nas duas abordagens propostas. Ele possui três módulos descritos a seguir.

### 3.5.1 MONITORAMENTO DE DESEMPENHO DE TRANSAÇÕES

Este módulo permite que sejam feitas análises de desempenho no nível de detalhe mais alto, ou seja, ele baseia-se na abordagem microscópica. Para isso, FICHE e OLIVEIRA (2004) propõem os Diagramas de Seqüência de Desempenho.

A Figura 12 mostra um Diagrama de Seqüência de Desempenho. Neste exemplo, a aplicação cliente localizada na máquina “fiche” faz uma chamada ao objeto CORBA *br.ufrj.labase.nisus.demo.corba.BusinessRule* localizado na aplicação servidora da máquina “servidor” **(a)**. A chamada levou 0,38 segundo **(b)** para ir da aplicação cliente até a aplicação servidora. O diagrama mostra, ainda, que a operação chamada foi a “doIt” e teve 0,901 segundo **(c)** como tempo de resposta. No contexto da execução da transação “doIt”, o objeto CORBA em questão invocou a operação “open” **(d)** de um outro objeto CORBA (*br.ufrj.labase.nisus.demo.corba.AccountManager*). Para evitar o problema da sobrecarga de informação, o diagrama apresenta alguns laços (*loops*) executados durante o processamento da transação. Como indicado no diagrama, a operação “open” foi executada 35 (trinta e cinco) vezes seguidas. Se todas as 35 chamadas estivessem explicitadas no diagrama, a sua interpretação seria mais trabalhosa.

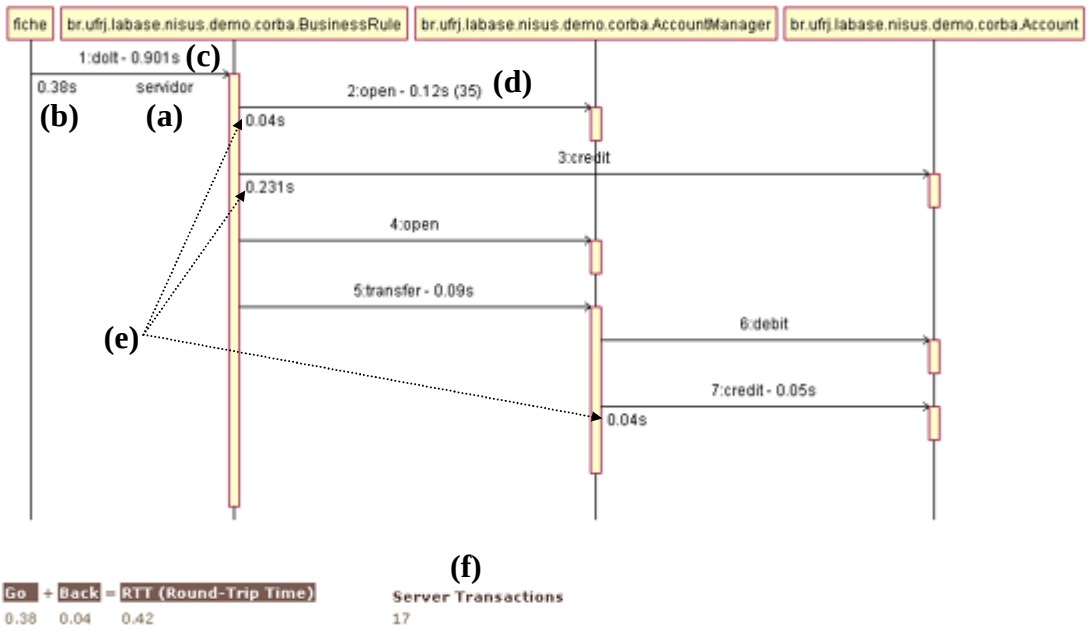


Figura 12 – Diagrama de Seqüência de Desemepenho.

Os pontos indicados com **(e)** mostram o tempo entre uma chamada e outra. Por exemplo, 0,231 segundo foi o tempo entre o término da operação “open” e o início da operação “credit”. Em **(f)**, o número de transações que o servidor estava executando além da transação analisada é mostrada. Portanto, neste caso, outras 17 transações estavam sendo atendidas concorrentemente com a transação “doIt”.

### 3.5.2 GERENCIAMENTO ON-LINE DE CARGA E RECURSOS

Este módulo permite que recursos de hardware como CPU e memória sejam gerenciados. Ele foi idealizado na aplicação *Broker* mostrada no Capítulo 2. Como foi visto, a carga de um servidor Web é baseada apenas no número de clientes atendidos por ele. Informações mais detalhadas como consumo de memória e CPU, por exemplo, não eram considerados. E, com o NISUS, é possível monitorar o consumo de memória e CPU, além de identificar os processos que estão executando nas aplicações clientes e servidoras.

A metáfora da temperatura possui vantagens como simplicidade e objetividade. Assim, a carga corrente experimentada pelas aplicações clientes e servidoras pode ser obtida através da seguinte fórmula:

- **Carga** = **Temperatura** = (Uso de memória % + Uso de CPU % + (*Carga de Transações* %)) / 3.
  *Carga de Transações* = (n / t)*100, onde *n* representa o número de transações que ainda não foram processadas, e *t* o número máximo estimado que a aplicação servidora do sistema pode atender de uma só vez.

### 3.5.3 RELATÓRIOS

Ele permite que sejam feitas análises de desempenho no nível de detalhe mais baixo (estratégia macroscópica).

Com as informações coletadas pelo NISUS, é possível criar diversos relatórios como, por exemplo:

- Porcentagem de transações recebidas por servidor;
- Tempo de resposta médio das transações (de um determinado objeto ou de todos);
- Maiores e Menores tempos de resposta (de um determinado objeto ou de todos);
- Relação de objetos mais utilizados pelo sistema, com os seus respectivos métodos;
- Consumo de memória e CPU de acordo com o tempo;
- Número de transações atendidas por um servidor de acordo com o tempo.

## 3.6 CONCLUSÃO

Neste capítulo, as principais características da arquitetura foram apresentadas. A arquitetura é capaz de atender aos principais de requisitos de gerenciamento de desempenho. Ela baseia-se ainda nas técnicas tradicionais de monitoramento de

desempenho sem requerer o uso de instrumentação de código. Além disso, o mecanismo de coleta de dados combina centralização e distribuição. Dessa forma, esse capítulo apresentou uma abordagem mais adequada para gerenciar o desempenho de sistemas distribuídos sem causar danos ao desempenho e sem afetar a legibilidade do código.

# 4 IMPLEMENTAÇÃO

*Este capítulo mostra como cada elemento da arquitetura do NISUS foi implementado. Ele mostra, ainda, o esforço exigido e as dificuldades encontradas ao longo da implementação.*

## 4.1 INTRODUÇÃO

A fase de implementação tem o objetivo de construir os componentes descritos no capítulo anterior para serem usados no experimento e no estudo de caso. O *NISUS Web Console* também foi desenvolvido para mostrar quais informações podem ser disponibilizadas e como elas podem ser visualizadas. Para isso, foi utilizado o seguinte ambiente de desenvolvimento:

- *IntelliJ* – Ambiente de desenvolvimento para a linguagem Java
- SEQUENCE[9] – É uma aplicação Java capaz de apresentar diagramas de seqüência a partir de uma descrição textual.
- *JFreeChart*: API para geração de gráficos.
- JSP e JSTL (*Java Standard Tag Libs*) – O *Web Console* foi implementado com essas duas tecnologias para desenvolvimento de aplicações Web escritas em Java.
- API SNMP – Foi utilizada uma API SNMP para a linguagem Java baseada no padrão SNMPv1, da empresa *AdventNet*[10], em sua versão 4.0.
- Rational ROSE (Modelagem de objetos) e DBDesigner[11] (Modelagem de dados)
- JUnit[12] – API para desenvolvimento de testes de unidade para a linguagem Java.

## 4.2 WEB CONSOLE

[9] http://www.zanthan.com/itymbi/archives/cat_sequence.html
[10] http://www.adventnet.com
[11] http://www.fabforce.net/dbdesigner4
[12] http://www.junit.org

Essa seção descreve a implementação de cada parte do *Web Console* foi construído: Diagrama de Seqüência de Desempenho, Gerenciamento On-line de Carga e Recursos, Relatórios e Configuração.

### 4.2.1 DIAGRAMA DE SEQUÊNCIA DE DESEMPENHO

Como mostrado na Figura 13, o SEQUENCE é uma aplicação *desktop* Java capaz de apresentar Diagramas de Seqüência da UML a partir de uma entrada textual.

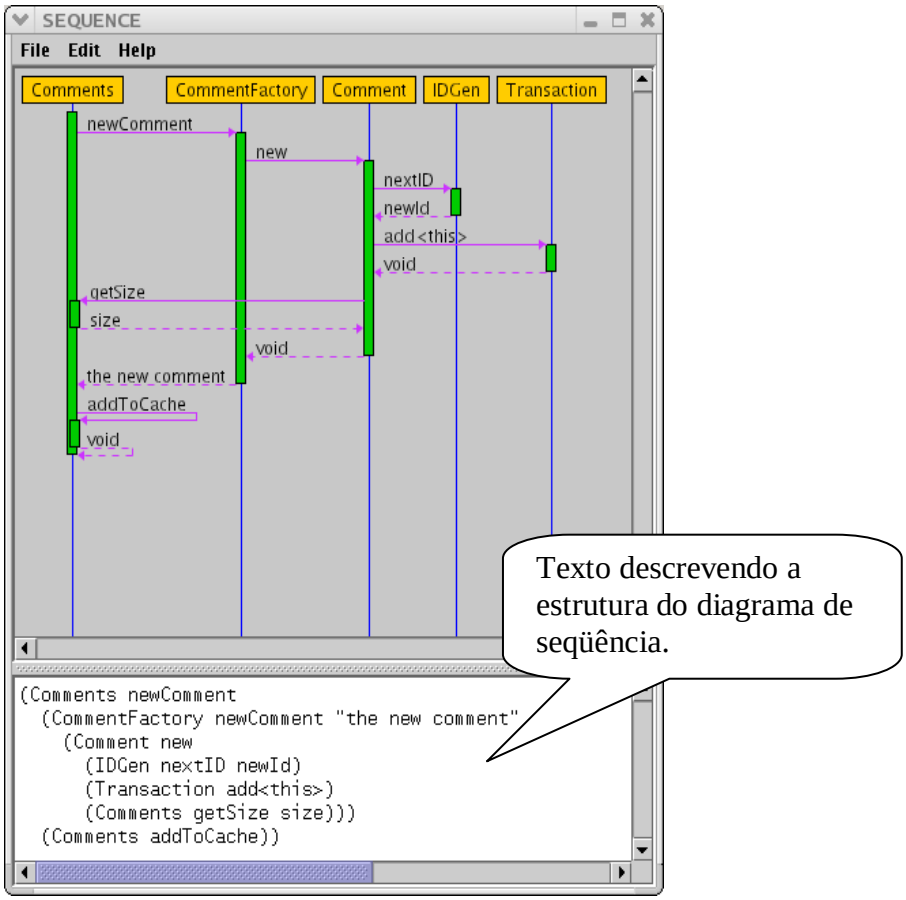


Figura 13 – Versão original do SEQUENCE.

Para ser utilizado no NISUS, foram realizadas algumas modificações no SEQUENCE como seguem descritas abaixo:

- Modificações na gramática que descreve o diagrama de seqüência. Agora, a gramática considera as informações propostas pelos Diagramas de Seqüência de Desempenho descritas no capítulo anterior.
- Foi criada uma versão *Applet* para que os diagramas de seqüência de desempenho sejam visualizados em qualquer *browser* e não apenas no *desktop*.
- Originalmente, o SEQUENCE funciona apenas com o JDK 1.4. Então, foi criada uma versão compatível com a máquina virtual Java que acompanha a maioria dos *browsers*.

### 4.2.2 GERENCIAMENTO ON-LINE DE CARGA E RECURSOS

A tela representada pela Figura 14 mostra a lista de aplicações clientes e servidoras. A lista mostra o número de transações ainda não processadas, o consumo de CPU, memória, o número de dados coletados que estão em memória e que foram gravados temporariamente em disco e a temperatura.

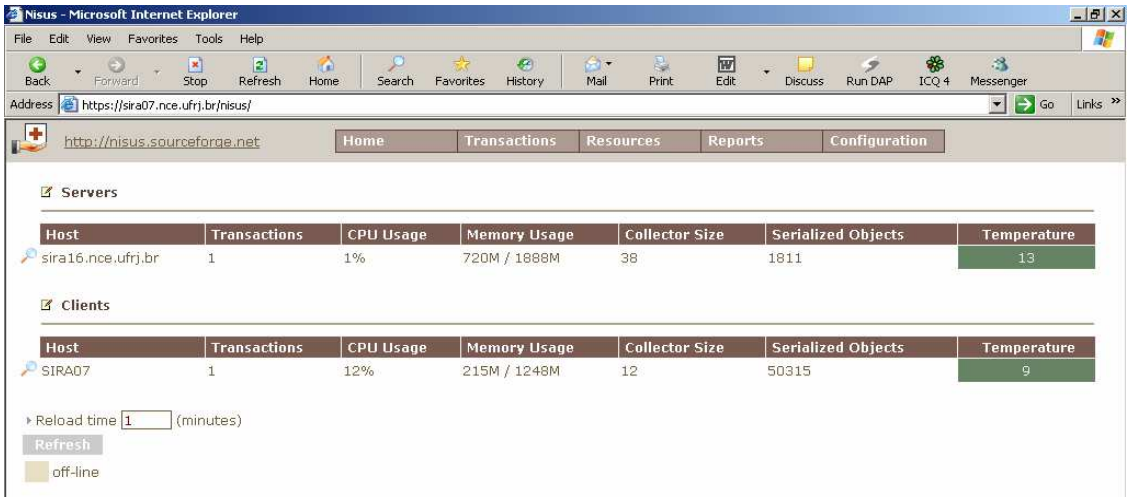


Figura 14 – Painel de Gerência On-line de Temperatura e Recursos

### 4.2.3 RELATÓRIOS

Para mostrar que tipo de informação sobre o desempenho dos sistemas distribuídos pode ser disponibilizado, alguns relatórios foram desenvolvidos:

- **Tempo médio de transações**

A Figura 15 mostra os parâmetros do relatório. Esse relatório é capaz de mostrar o tempo médio de processamento e de resposta de um determinado objeto, operação, máquina (cliente e servidor) e data. O tempo de resposta refere-se ao tempo de execução sob o ponto de vista da aplicação cliente, onde o tempo de rede é, então, considerado. O tempo de processamento não leva em consideração o tempo de rede, ou seja, ele representa o tempo de resposta sob o ponto de vista da aplicação servidora. Vale observar que o relatório pode ser agrupado por servidor.


Figura 15 – Parâmetros do relatório “Tempo médio de transações”.

A Figura 16 mostra o resultado do relatório, onde a barra vermelha representa o tempo médio de processamento e a barra azul representa o tempo médio de resposta do objeto *br.ufrj.labase.nisus.demo.corba.BusinessRule* considerando todos os servidores.

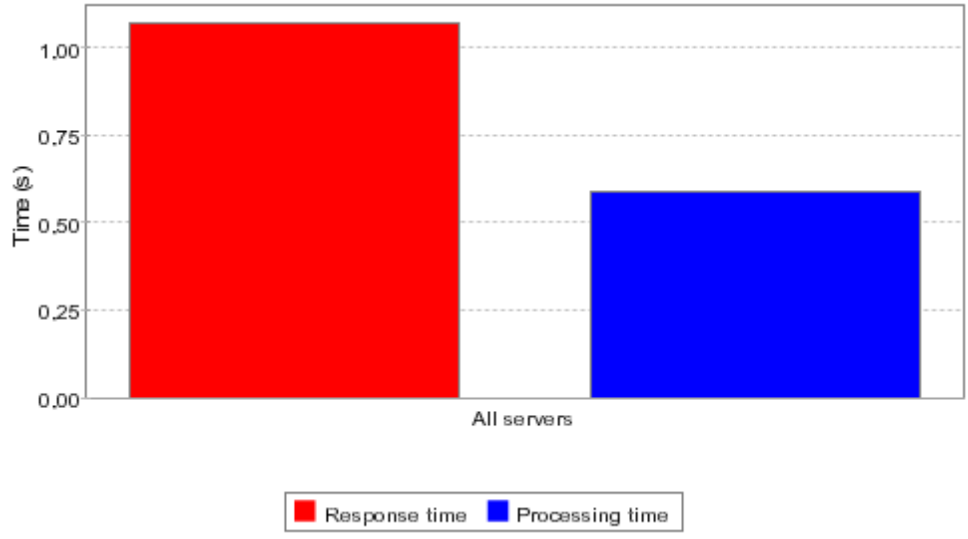

Figura 16 – Gráfico “Tempo médio de transações”.

- **Porcentagem de transações recebidas por servidor**

A Figura 17 mostra os parâmetros do relatório, onde apenas uma data deve ser fornecida.

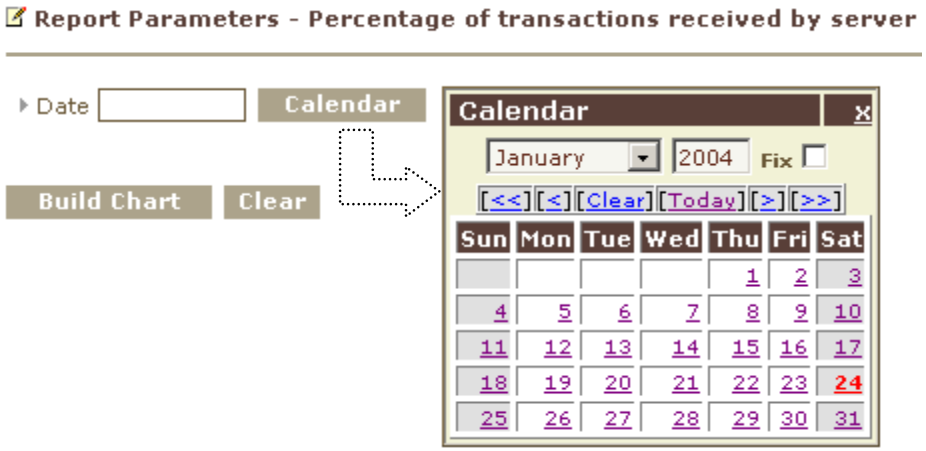


Figura 17 – Parâmetros do relatório “Porcentagem de transações recebidas por servidor”.

A Figura 18 mostra o resultado do relatório, onde a parte vermelha indica a porcentagem de transações processadas pelo servidor “fiche” (76,7%) e a parte azul indica a porcentagem de transações que foram processadas pelo servidor “Outro Servidor” (23,3%). Esse relatório indica, então, que por alguma razão não houve distribuição do número de transações entre os servidores disponíveis.

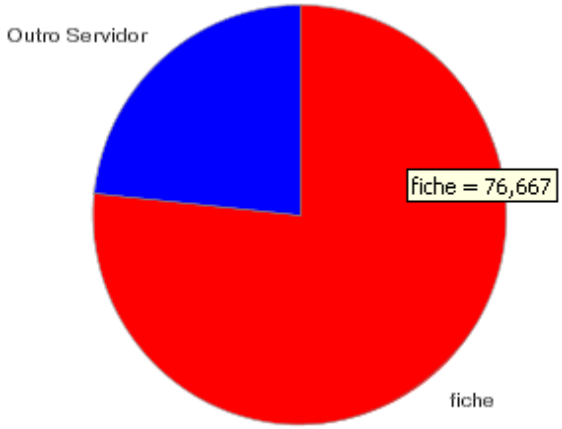


Figura 18 – Gráfico “Porcentagem de transações recebidas por servidor”.

### 4.2.4 CONFIGURAÇÃO

O primeiro passo para configurar o NISUS é informar o banco de dados a ser utilizado para armazenar as informações coletadas. Isso é feito através de um arquivo de propriedades (Figura 19) que deve estar junto com as classes do NISUS.

```
br.ufrj.labase.nisus.configuration.datasource.url=jdbc:mysql://maquina/nisus
br.ufrj.labase.nisus.configuration.datasource.driverClass=com.mysql.jdbc.Driver
br.ufrj.labase.nisus.configuration.datasource.database=MySQL
br.ufrj.labase.nisus.configuration.datasource.username=nisus
br.ufrj.labase.nisus.configuration.datasource.password=nisus
```

Figura 19 – Arquivo de configuração de banco de dados.

A partir daí, as configurações restantes podem ser feitas a partir do próprio *Web Console*.

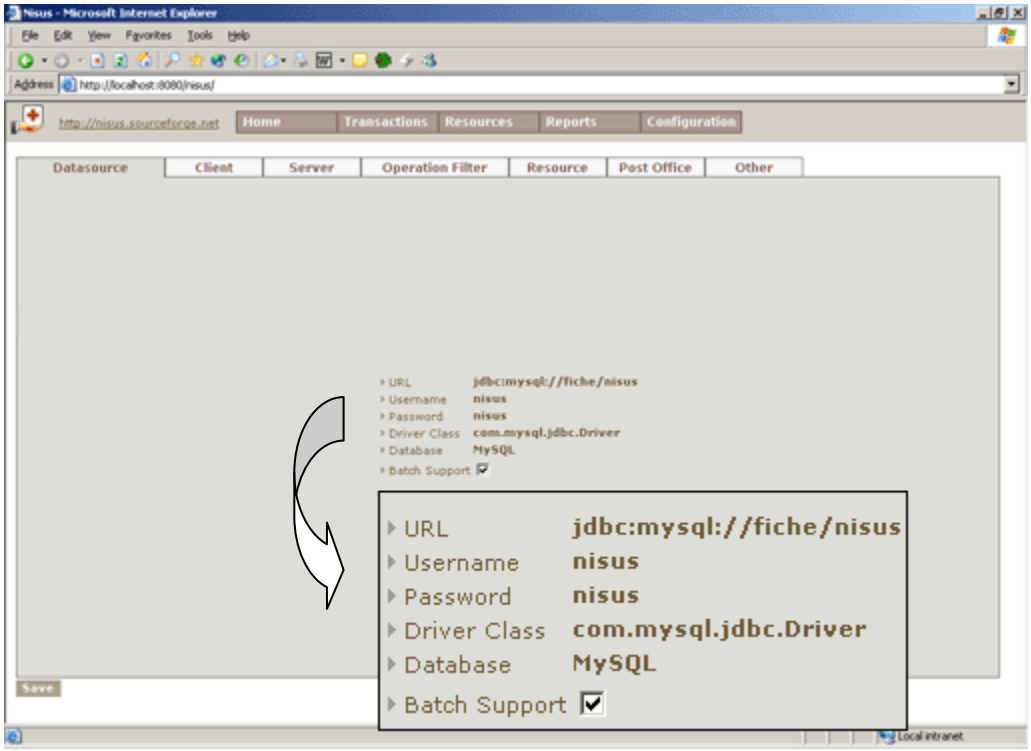


Figura 20 – Configuração de banco de dados.

A Figura 20 mostra as configurações de banco de dados. Através dessa interface, o administrador pode configurar o parâmetro (*Batch Support*) que indica se as informações coletadas serão gravadas em lote ou não. O administrador pode modificar apenas esse parâmetro.

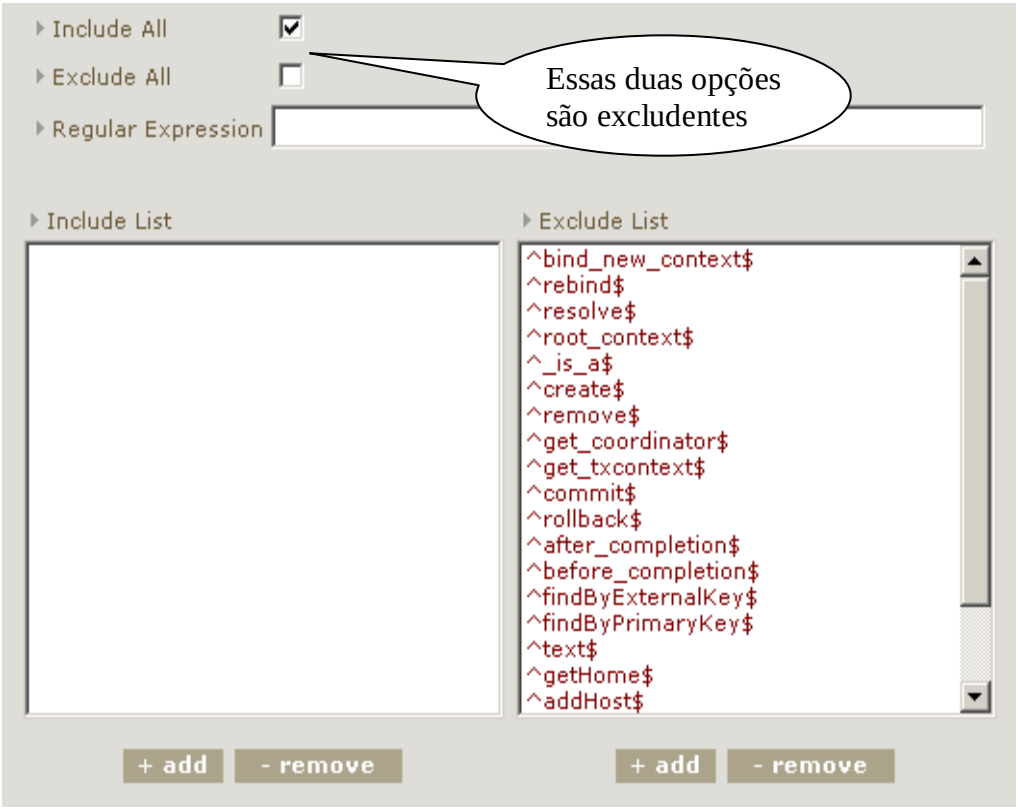


Figura 21 – Configuração do filtro de transações.

A Figura 21 mostra a tela com a qual determinadas transações podem ser filtradas/habilitadas pelo NISUS. Isso é feito através de expressões regulares. Se a opção “Include All” estiver marcada, todas as transações serão monitoradas pelo NISUS. O contrário ocorrerá se a opção “Exclude All” estiver marcada. Se uma delas estiver marcada, as expressões presentes nas listas serão desconsideradas.

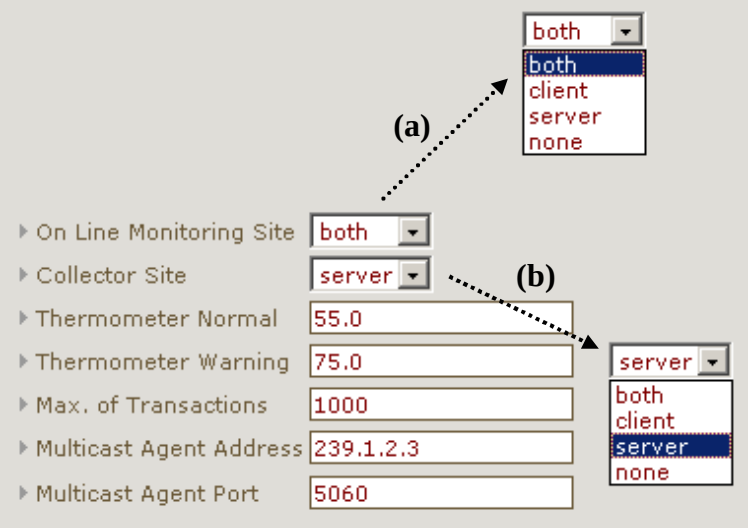


Figura 22 – Configuração do módulo de monitoramento de recursos.

A Figura 22 mostra a tela de configuração do módulo de monitoramento de recursos. Os campos *Multicast Agent Address* e *MulticastAgentPort* indicam o endereço *multicast* a ser usado no canal de comunicação e a porta, respectivamente. O campo *On Line Monitoring Site* (Figura 22.a) indica se os clientes e servidores serão membros ou não do canal de comunicação. O valor *both* indica que tanto os clientes quanto os servidores serão monitorados, ou seja, todos os clientes e servidores irão aparecer na tela de gerenciamento on-line de carga e recursos. O campo *Collector Site* determina onde informações como consumo de CPU e memória serão coletadas a cada minuto. Através dele, é possível desabilitar e habilitar essa funcionalidade quando ela não for necessária.

O campo *Max. of Transactions* representa um valor estimado do número de transações máximo que uma aplicação cliente / servidora pode requisitar / atender ao mesmo tempo. Esse valor é usado para o cálculo da temperatura. Os campos *Thermometer Normal* e *Thermometer Warning* indicam os limites para os estados *Normal*, *Aviso* e *Crítico*. Pela Figura 22, os limites seriam: *Normal* $\leq 55 <$ *Aviso* $\leq 75 <$ *Crítico* $\leq 100$.

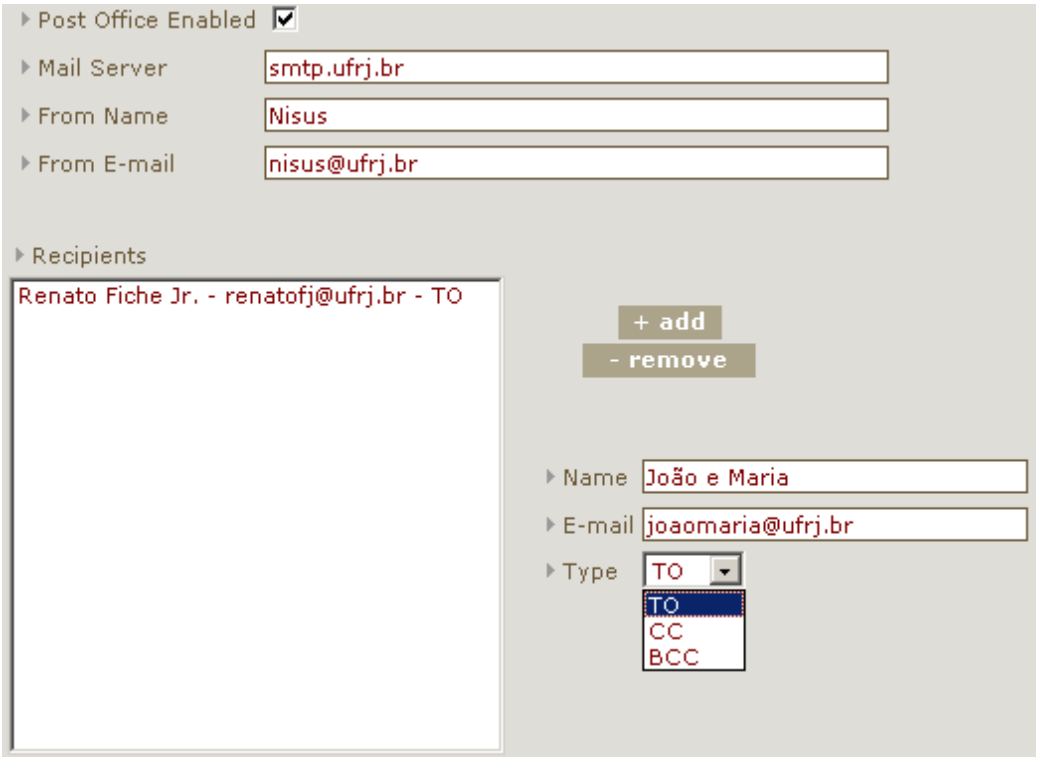


Figura 23 – Configuração do módulo de notificação.

Através da tela mostrada pela Figura 23, o administrador configura os parâmetros de notificação: Servidor de e-mail, nome e e-mail do remetente e a lista de destinatários. Essas configurações devem ser feitas porque o NISUS provê mecanismos de notificação. Por exemplo, se alguma falha ocorrer na aplicação ou durante o funcionamento do NISUS, um e-mail será enviado.

## 4.3 CLASSES

A seção presente mostra a modelagem das classes que representam os componentes da arquitetura NISUS. As classes foram agrupadas de acordo com a seguinte estrutura de pacotes mostrada pelo Figura 24.

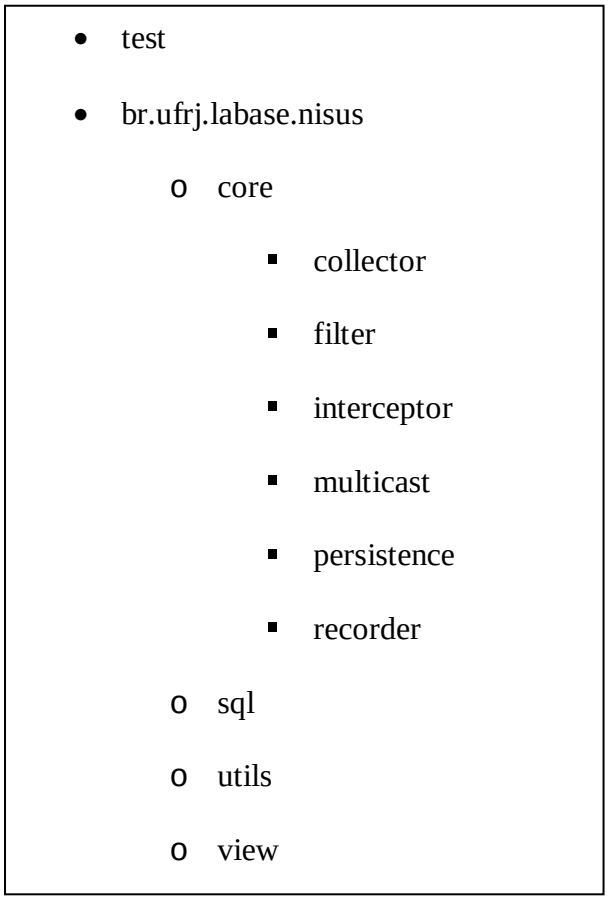


Figura 24 – Estrutura de pacotes das classes do NISUS.

No pacote *test*, encontram-se as classes de testes. No pacote *core*, concentram-se as principais classes do NISUS, aquelas responsáveis pela coleta de dados de desempenho, filtro ect. No pacote *sql*, encontram-se classes responsáveis pela execução de comandos SQL e pela abertura e fechamento de conexões com o banco de dados. No pacote *view*, encontram-se classes de controle da vista utilizadas na construção do *Web Console*.

### 4.3.1 INTERCEPTORS

As classes *ProfileClientInterceptor* e *ProfileServerInterceptor* (Figura 25) representam *CORBA Request Interceptors*. Para construí-los as interfaces *org.omg.PortableIntercetor.ClientRequestInterceptor* e *org.omg.PortableIntercetor.ServerRequestInterceptor*) foram implementadas. Essas interfaces possuem basicamente os métodos que correspondem aos pontos invocados

pelo ORB como já foi visto anteriormente. As duas classes herdam da classe *ProfileInterceptor*.

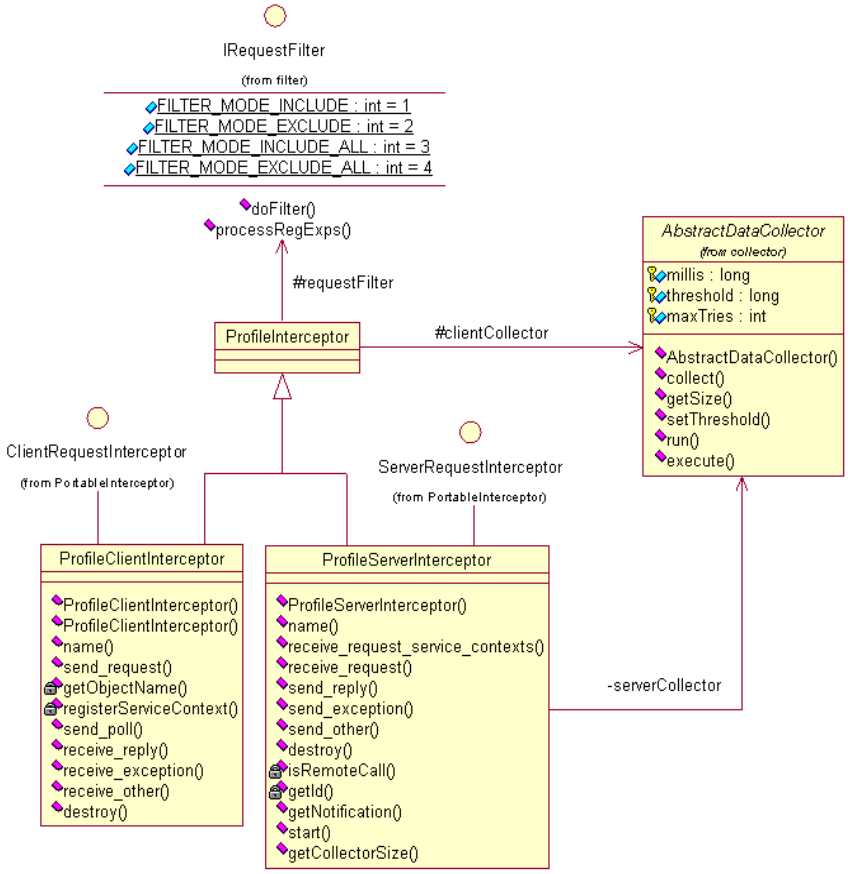


Figura 25 – Classes do pacote “interceptor”.

### 4.3.2 HIERARQUIA DE TRANSAÇÕES

Essa seção mostra como o NISUS consegue obter a seqüência exata das chamadas CORBA em um ambiente complexo, distribuído e *multithreaded*.

Como foi visto anteriormente, o objeto *ProfileClientInterceptor* é invocado pelo ORB quando a aplicação cliente realiza uma chamada à aplicação servidora. Quando isso ocorre, ele cria uma instância do objeto *ClienteTransactionData*, que armazena algumas

informações tais como o tempo da chamada (*timestamp*), nome da máquina cliente e identificador único da transação. Por fim, este objeto é associado a *thread* corrente através da classe *java.lang.ThreadLocal*.

Quando a requisição chega à aplicação servidora, o ORB invoca o objeto *ProfileServerInterceptor*. Este, por sua vez, cria um objeto *ServerTransactionData* (Figura 26), preenche nele algumas informações sobre a transação e o encapsula no objeto *TransactionComposite* (Figura 26). Este representa a raiz da hierarquia da transação e, por fim, é associado a *thread* corrente também através da classe *java.lang.ThreadLocal*.

Quando uma chamada (sub-transação) é feita dentro do contexto da transação principal, o objeto *ProfileClientInterceptor* é invocado pelo ORB servidor. Neste momento, o objeto *TransactionComposite* que representa a raiz da hierarquia é recuperado. Através do método *getRequestNotFinished*, o objeto *TransactionComposite* que representa a transação (ou sub-transação) ainda pendente é obtida. Logo em seguida, o interceptor cria um objeto *ClientTransactionData* (Figura 26), preenche algumas informações sobre a requisição (nome do objeto, operação invocada etc). Esse novo objeto *TransactionComposite* representa um novo filho do objeto *TransactionComposite* recuperado anteriormente. Esse processo, então, se repete até que a transação termine de ser executada. O código do método *getRequestNotFinished* é mostrado abaixo:

```
public ITransactionComposite getRequestNotFinished() {
 if (children != null) {
  int size = children.size();
  if (size > 0) {
   ITransactionComposite lastData = (ITransactionComposite) children.get(size - 1);
   if (!lastData.getFinished()) {
    return lastData.getRequestNotFinished();
   }
  }
 }
 return this;
```

}

Portanto, no final do processamento da transação, a hierarquia da transação estará montada através dessa composição de objetos. E, quando o processamento da transação chega ao fim, o ORB servidor invoca novamente o objeto *ProfileServerInterceptor*. Neste momento, a composição de objetos é percorrida e o objeto *AbstractDataCollector* recebe cada elemento da hierarquia para serem gravados no banco de dados. A seguir, o ORB envia a resposta à aplicação cliente.

Quando a resposta chega à aplicação cliente, o ORB cliente invoca novamente o objeto *ProfileClientInterceptor*. Por fim, o objeto *AbstractDataCollector*, no cliente, recebe o objeto *ClientTransactionData* para ser gravado no banco de dados.

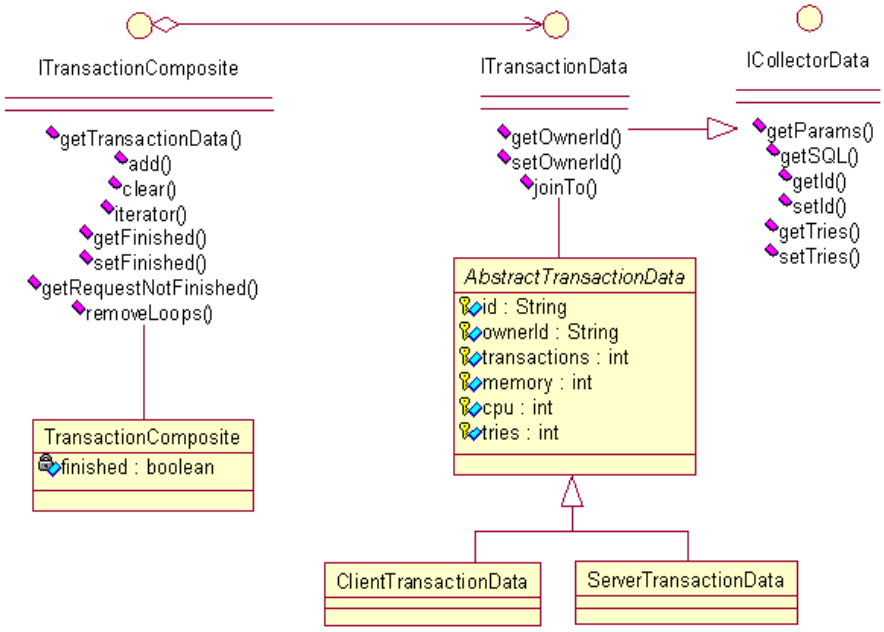


Figura 26 – Classes do pacote *collector/data*.

### 4.3.3 MECANISMO DE COLETA DE DADOS

O atributo *millis* da classe *AbstractDataCollector* (Figura 27) determina o intervalo de tempo, em milisegundos, que o objeto *DataRecorder* (Figura 27) deve ser invocado. A classe *AbstractDataCollector* tem uma estrutura interna (atributo *buffer*) para

armazenar os objetos que contêm dados sobre as transações e sub-transações (*ClientTransactionData* e *ServerTransactionData*). Cada objeto desses representa, na verdade, um comando SQL que é executado pelo objeto *DataRecorder*.

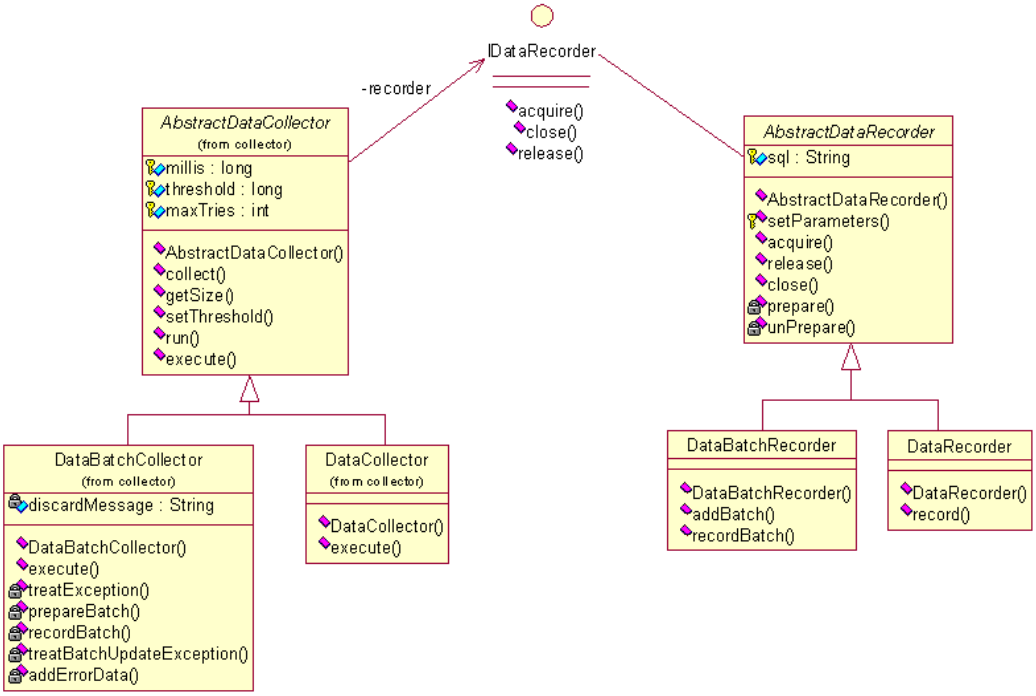


Figura 27 – Classes dos pacotes “collector” e “recorder”.

O acesso ao banco de dados é feito através de JDBC. Os dados podem ser gravados em lote (*batch*) ou não. Isso depende da implementação JDBC utilizada pelo NISUS. A possibilidade de enviar comandos SQL em lote para o banco de dados é uma característica incluída apenas na especificação JBDC 2.0. Por isso, existem duas classes concretas e que herdam de *AbstractDataCollector*: *DataBatchCollector* e *DataCollector*. A primeira com suporte à gravação em lote e a segunda não. A configuração do NISUS indicará qual implementação será usada. É importante destacar que o uso de orientação a objetos, principalmente o conceito de abstração, dá maior flexibilidade para trocar entre uma implementação e outra. Isso ocorre porque as classes *ProfileClientInterceptor* e *ProfileServerInterceptor* conhecem apenas a classe *AbstractDataCollector*, não importando a elas qual das duas implementações concretas

está sendo usada. O atributo *threshold* (da classe *AbstractDataCollector*) representa o número de dados que é enviado de uma só vez ao banco de dados.

A classe *AbstractDataCollector* contem o comportamento responsável por armazenar e recuperar os dados do disco. Esse procedimento é executado quando o número de objetos no *buffer* é maior do que duas vezes o valor do *threshold*. Portanto, quando esse limite for ultrapassado, o método *serializeCollectorData* será chamado. Esse método grava, em disco, cada objeto do *buffer* no formato de arquivo. O código desse método segue abaixo:

```
 private void serializeCollectorData() throws Exception {
  int currentSize = buffer.size();
  for (int i = 0; i < currentSize; i++) {
   ICollectorData data = (ICollectorData) buffer.remove(0);
   try {
    FileOutputStream out = new FileOutputStream(buildFileName());
    try {
     SerializationUtils.serialize(data, out);
     serial++;
    } finally {
     out.close();
    }
   } catch (Exception e) {
    buffer.add(data);
    throw e;
   }
  }
}
```

Quando o *buffer* estiver vazio, o objeto *AbstractDataCollector* verifica se há objetos em disco para serem gravados no banco de dados. Quando isso ocorre, o método *deserializeCollectorData* é invocado e os arquivos são transformados (no máximo o valor do *threshold*) em objetos e armazenados novamente no *buffer*.

```
private void deserializeCollectorData() throws Exception {
   int i = 0;
   while (serial > 0 && (i < threshold)) {
    serial--;
    try {
     File file = new File(buildFileName());
     FileInputStream in = new FileInputStream(file);
     try {
      Object data = SerializationUtils.deserialize(in);
      buffer.add(data);
      file.delete();
      i++;
     } finally {
      in.close();
     }
    } catch (Exception e) {
     serial++;
     throw e;
    }
   }
  }
```

Vale observar que o atributo *serial* representa o índice do último arquivo gravado em disco. Portanto, esse atributo é incrementado conforme os dados são gravados em disco, e decrementado quando os mesmos são recuperados do disco.

### 4.3.4 MONITORAMENTO AO NÍVEL DO SISTEMA OPERACIONAL

A classe *ResourceCollector* (Figura 28) é responsável em prover informações sobre o consumo de recursos das máquinas e informações sobre o funcionamento do próprio NISUS como, por exemplo, quantidade de dados que foram armazenados em disco, quantidade de dados que estão armazenados em memória principal etc. A cada minuto, ela fornece alguns desses dados para os objetos *ProfileClientInterceptor* e *ProfileServerInterceptor*. O método *calculateTemperature* transforma todas essas informações em um número que traduz a situação atual das máquinas. O objeto

*ResourceCollector* possui uma referência para o objeto *SNMPSession* para obter informações através de SNMP: consumo de memória, CPU, lista de processos, capacidade de memória física e virtual etc. A classe *SNMPSession* representa uma implementação do Design Pattern *Wrapper* (GAMMA *et al* 1999).

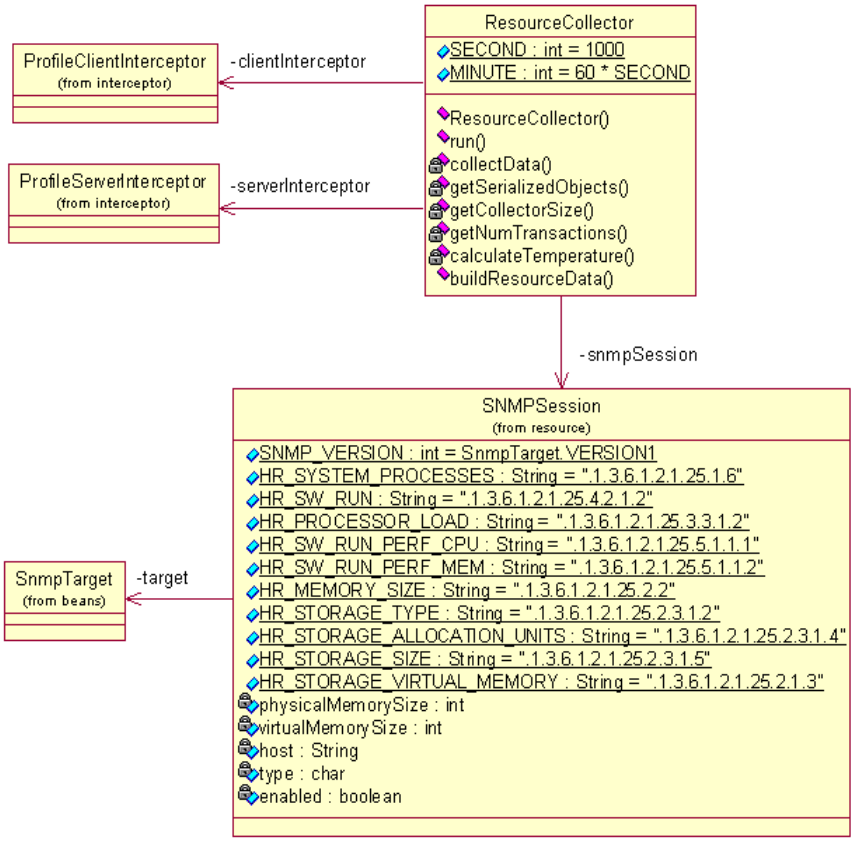


Figura 28 – Classes do pacote *collector*.

### 4.3.5 CANAL DE COMUNICAÇÃO MULTICAST

As classes *MulticastAgent* e *PresentationMulticastAgent* (Figura 29) implementam a mesma interface: *IMulticastAgent*. Um objeto da classe *MulticastAgent* é instalado nas aplicações clientes e servidoras e um objeto da classe *PresentationMulticastAgent* é instalado no console Web. O objeto *MulticastAgent* possui uma referência para o objeto *ResourceCollector* através da qual dados sobre utilização de recursos são solicitados.

A classe *MulticastSocketAgent* representa o canal de comunicação em si, no nível de rede. Ela herda da classe *java.net.MulticastSocket*. Quando esse objeto recebe uma mensagem, ele a repassa para a implementação da interface *IMulticastAgent* ao qual ele está associado, através da chamada ao método *receive*.

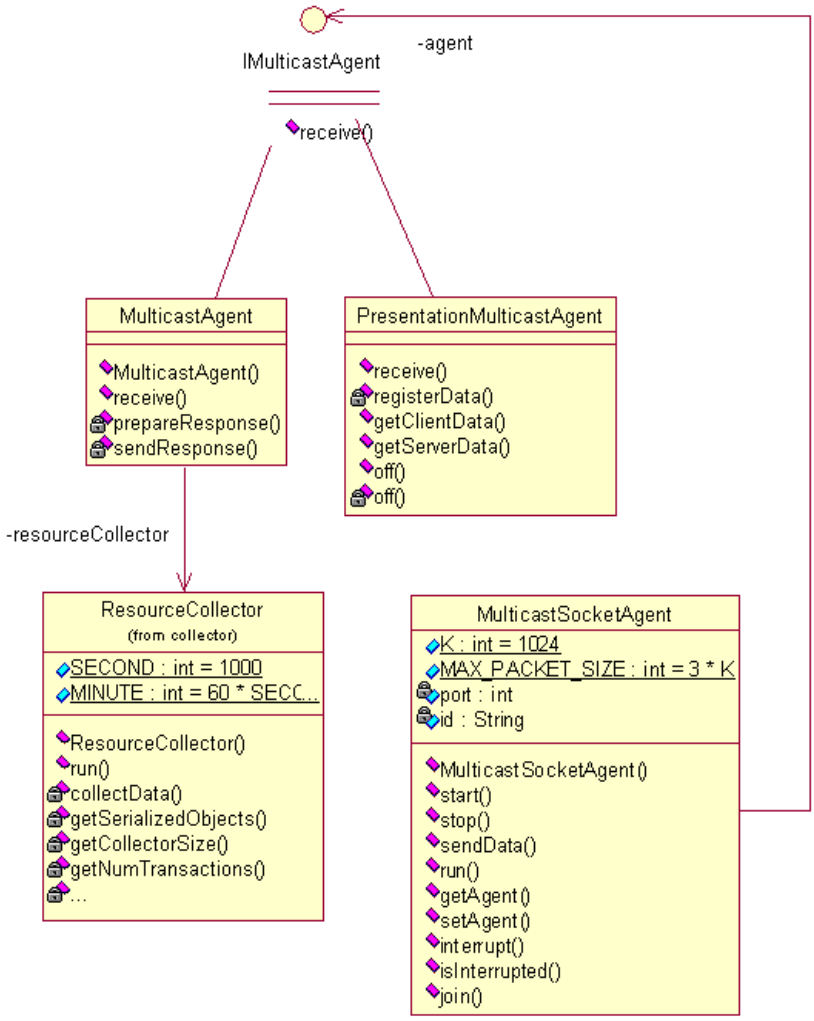


Figura 29 – Classes do pacote *multicast*.

## 4.4 MODELO DE DADOS

O NISUS possui algumas tabelas de configuração e uma tabela onde os dados coletados são armazenados.

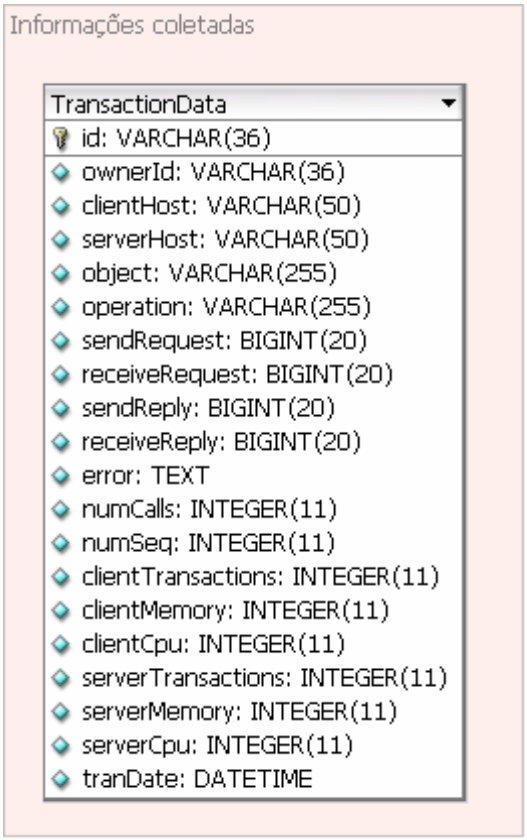


Figura 30 – Tabela *TransactionData*.

A Figura 30 mostra a estrutura da tabela *TransactionData*. Essa tabela armazena os dados coletados sobre as transações (e sub-transações). Através dela, todas as informações sobre o desempenho do sistema podem ser extraídas e analisadas. Ela possui os seguintes campos:

- *id*: Identificador único da transação (ou sub-transação). Esse campo é a chave-primária da tabela;
- *ownerid*: Identificador da transação "pai", no caso de uma sub-transação;
- *clientHost*: Nome da máquina cliente;
- *serverHost*: Nome da máquina servidora;
- *object*: Nome do objeto CORBA invocado;
- *operation*: Operação ou método do objeto CORBA;
- *sendRequest*: *Timestamp* correspondente ao início da transação;

- *receiveRequest*: *Timestamp* correspondente ao momento em que a chamada chega ao servidor;
- *sendReply*: *Timestamp* correspondente ao momento em que a transação termina de ser processada;
- *receiveReply*: *Timestamp* correspondente ao momento em que o cliente recebe a resposta do servidor;
- *error*: Mensagem de erro, se houver;
- *numCalls*: Número de chamadas seguidas feitas a uma mesma transação;
- *numSeq*: Número que determina a ordem em que as transações foram chamadas;
- *clientTransactions*: Número de transações que a aplicação cliente está esperado para serem processadas;
- *serverTransactions*: Número de transações que a aplicação servidora está processando;
- *clientMemory, clientCPU, serverMemory, serverCPU*: consumo de memória e CPU da máquina cliente e servidora;
- *tranDate*: Data e hora de ocorrência da transação.

As tabelas *ClientConfiguration* e *ServerConfiguration* (Figura 31) guardam as configurações do NISUS para as aplicações clientes e servidoras. Através delas, pode-se configurar parâmetros que indicam se os *interceptors* estarão habilitados ou não, o intervalo de tempo que os dados serão enviados ao banco de dados.

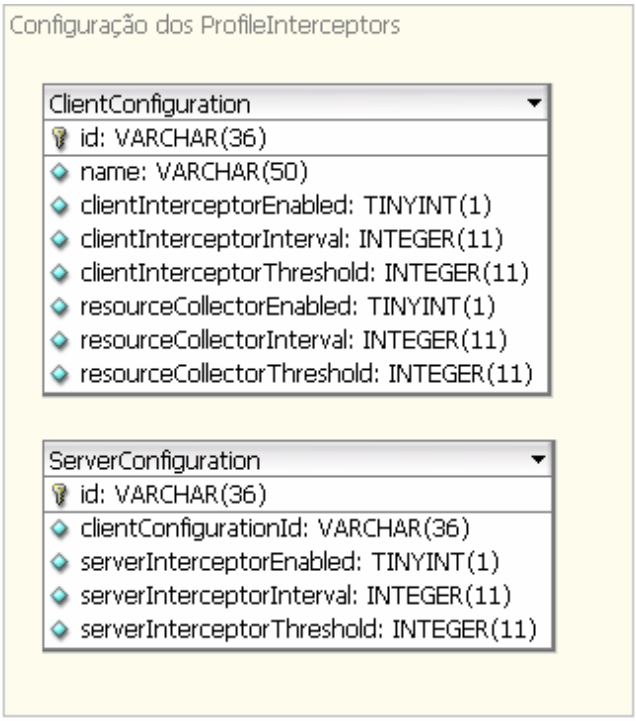


Figura 31 – Tabelas de configuração dos componentes do NISUS nas aplicações clientes e servidoras.

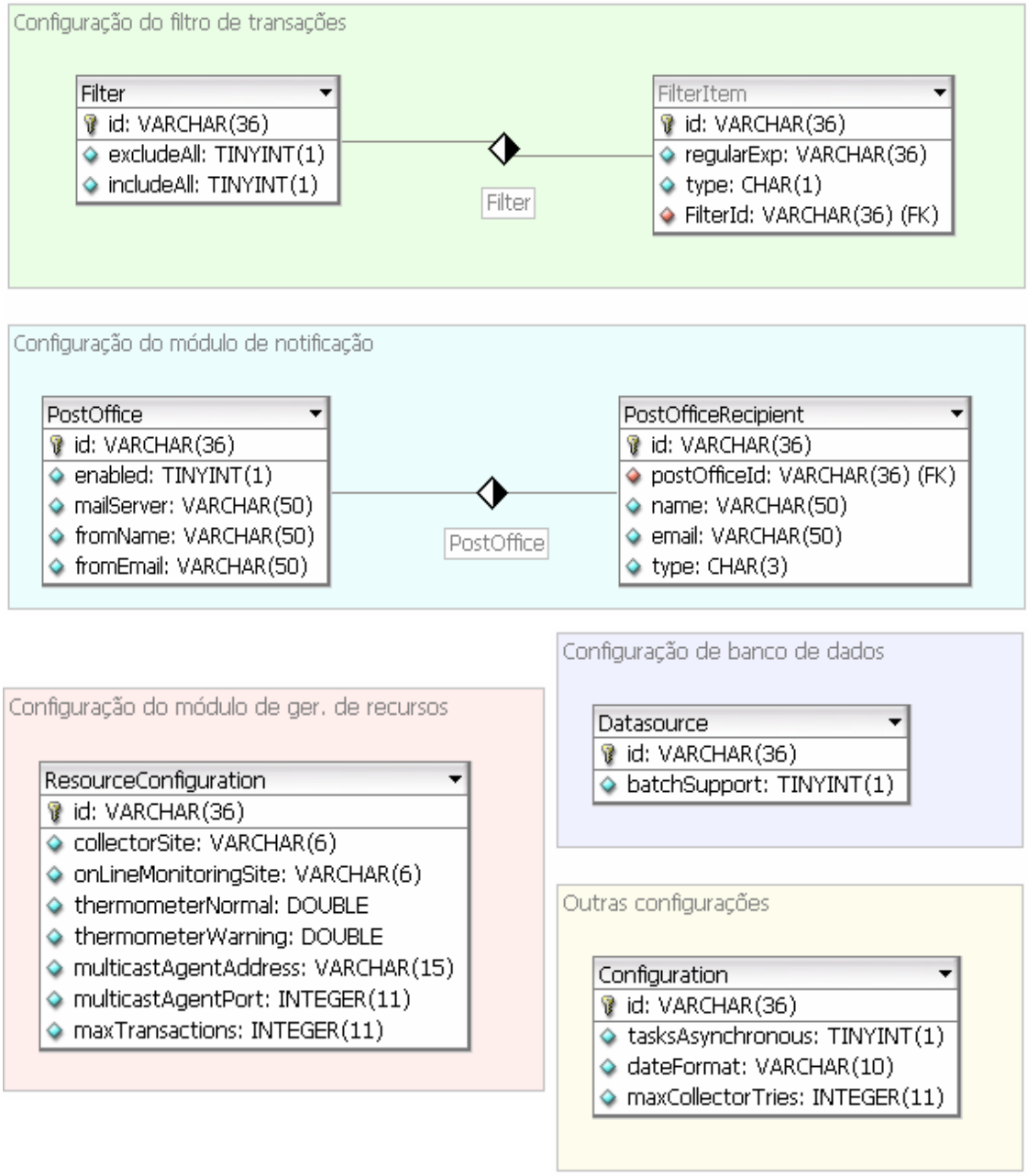


Figura 32 – Demais tabelas de configuração.

A Figura 32 mostra as demais tabelas de configuração. As tabelas *PostOffice* e *PostOfficeRecipient* são usadas para configuração de e-mail. Através da tabela *ResourceConfiguration*, pode-se configurar o endereço *Multicast* a ser usado na comunicação entre o Web Console e as aplicações cliente e servidora. Além disso, ela armazena os valores correspondentes aos limites para os estados da temperatura *Normal* e *Warning*.

## 4.5 TESTES DE UNIDADE

A implementação da arquitetura descrita neste trabalho exigiu esforço e experiência em um ambiente que dificulta a depuração de programas, visto que aplicações cliente servidora executam em processos independentes, distribuídos. Por essa razão, detectar eventuais erros no funcionamento do NISUS torna-se uma tarefa bastante difícil. Em vista desse cenário, foram desenvolvidos alguns testes de unidade (ou testes unitários) (BECK 2000) para verificar o comportamento das principais classes.

Os testes de unidade concentram-se na verificação da menor unidade de um projeto de software orientado a objeto – a classe. Em cada teste de unidade, caminhos de controle importantes são testados para descobrirem erros dentro das fronteiras da classe. A estrutura de dados local também é examinada para ter a garantia de que os dados armazenados temporariamente mantêm sua integridade durante todos os passos de execução de um algoritmo.

A Figura 33 mostra o teste de unidade criado para a classe responsável pelo filtro de operações do NISUS – classe *RequestFilter*. Por padrão, cada método da classe de teste tem o objetivo de testar um único método da classe a ser testada. Sendo assim, o método *testDoFilter* tem o objetivo de testar o método *doFilter*.

```
package test.br.ufrj.labase.nisus.core.filter;

import junit.framework.TestCase;
import test.br.ufrj.labase.nisus.mock.RequestFilterMock;

public class RequestFilterTest extends TestCase {

  public void testDoFilter() {
    RequestFilterMock requestFilter = new RequestFilterMock();

    boolean doFilter = requestFilter.doFilter("to_name");
    assertEquals(true, doFilter);

    doFilter = requestFilter.doFilter("lookup_id");
    assertEquals(true, doFilter);

    doFilter = requestFilter.doFilter("get_blahblah");
    assertEquals(true, doFilter);

    doFilter = requestFilter.doFilter("blahblah_get_blahblah");
    assertEquals(false, doFilter);
  }
}
```

Figura 33 – Teste de unidade da classe *RequestFilter*.

# 5 AVALIAÇÃO

*O capítulo a seguir tem o objetivo de apresentar o experimento ao qual a solução implementada foi aplicada. Além disso, o Sistema de Gestão Acadêmica da UFRJ foi utilizado como estudo de caso.*

## 5.1 INTRODUÇÃO

Para avaliar o impacto no desempenho causado pela implementação do NISUS bem como pela implementação *Portable Interceptors* do Visibroker, um experimento foi realizado. O experimento foi executado de acordo com alguns cenários.

Além disso, um sistema em produção foi utilizado como estudo de caso. Desenvolvido no NCE, o Sistema Integrado de Gestão Acadêmica (SIGA) tem por objetivo integrar todos os sistemas de registro acadêmico da UFRJ. Além da secretarias acadêmicas, aluno e professor têm acesso a funções como inscrição em disciplinas, lançamento de notas, boletim e histórico escolar. Esta abrangência influencia na quantidade de usuários. São 60.000 alunos, mais de 3.000 professores e milhares de funcionários.

## 5.2 EXPERIMENTO

A Figura 34 mostra a configuração do ambiente usado para experimento.

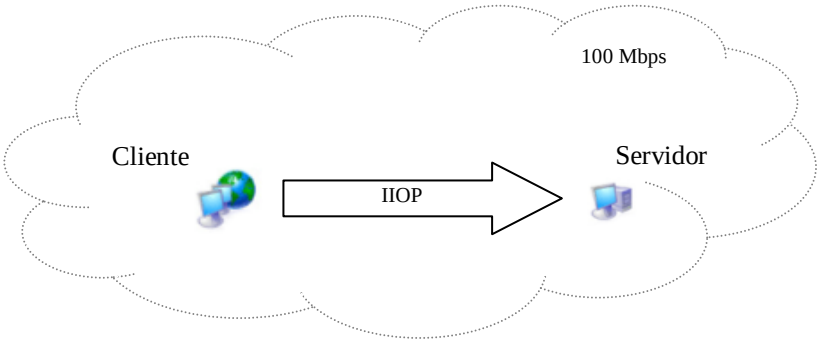


Figura 34 – Ambiente usado no experimento.

- Configuração de hardware: Athlon XP 1,7 MHz, 512 Mb de RAM.
- Configuração de software: Linux Red Hat 8.

Como o experimento tem o objetivo apenas de medir o impacto provocado no desempenho por cada *Portable Interceptor* do NISUS, o mecanismo de coleta de dados foi desabilitado. Portanto, os dados coleados foram armazenados apenas em memória.

### 5.2.1 CENÁRIOS

O experimento tem o objetivo de medir o impacto provocado no desempenho por cada *Portable Interceptor*: *ProfileClientInterceptor* (instalado no lado cliente e servidor) e *ProfileServerInterceptor*. Os seguintes cenários foram considerados:

- Cenário 1 – Sem a presença de qualquer um dos componentes do NISUS (Experimento Base);
- Cenário 2 – Todos os *interceptors* instalados e habilitados (*Client Profile Interceptor* instalado tanto no lado cliente quanto no servidor, e o *Server Profile Interceptor* instalado no lado servidor);
- Cenário 3 – O componente *Client Profile Interceptor* instalado no lado cliente e o componente *Server Profile Interceptor* instalado no lado servidor;
- Cenário 4 – Apenas o componente *Client Profile Interceptor* instalado no lado cliente;
- Cenário 5 – Mesma configuração do Cenário 2, mas, neste caso, todos os componentes estão desabilitados.

O Cenário 1 serve como referência para o cálculo do custo de desempenho. O Cenário 2 é o mais completo porque todos os componentes do NISUS foram instalados. A configuração do cenário 3 serve apenas quando se deseja obter o tempo de resposta e o tempo de rede, ou seja, as chamadas realizadas . No Cenário 4, nenhum componente é instalado no servidor, ou seja, ele serve apenas para obter o tempo de resposta da aplicação. O Cenário 5 serve apenas para medir o custo de desempenho provocado pela implementação *CORBA Portable Interceptors* do Visibroker.

Cada cenário foi executado 10 vezes com 30 clientes invocando a aplicação servidora 50 vezes. Portanto, cada cenário produziu um total de 1500 transações.

### 5.2.2 RESULTADOS

| Cenários | Média (s) | Mediana (s) | Mínimo (s) | Máximo (s) | D. Padrão (s) | Custo |
|---|---|---|---|---|---|---|
| **1** | 7,33 | 7,26 | 6,25 | 10,78 | 0,58 | 0% |
| **2** | 8,07 | 7,82 | 6,16 | 11,55 | 0,99 | 10,1% |
| **3** | 7,82 | 7,66 | 6,18 | 11,19 | 0,83 | 6,7% |
| **4** | 7,58 | 7,51 | 6,15 | 11,25 | 0,69 | 3,4% |
| **5** | 7,54 | 7,48 | 6,19 | 11,97 | 0,73 | 2,9% |

Tabela 5 – Resultados para a implementação CORBA Visibroker.

De acordo com a Tabela 5, chegamos às seguintes equações:

1. Cc + Cs + S = 10,1%
2. Cc + S = 6,7%
3. Cc = 3,4%
4. Cc + Cs + S = 2,9%

Onde:

- **Cc**: *Profile Client Interceptor* instalado no cliente
- **Cs**: *Profile Client Interceptor* instalado no servidor
- **S**: *Profile Server Interceptor* instalado
- **Cc**: *Profile Client Interceptor* instalado no cliente e desabilitado
- **Cs**: *Profile Client Interceptor* instalado no servidor e desabilitado
- **S**: *Profile Server Interceptor* instalado e desabilitado

Usando as equações 2 e 3, concluímos que S é igual a 3,3% (3,4% + S = 6,7%), ou seja, o componente *Profile Server Interceptor* causou 3,3% de impacto no desempenho. Com esse resultado e usando as equações 1 e 3, concluímos que Cs é igual a 3,4% (3,4% + Cs + 3,3% = 10,1%), ou seja, o componente *Profile Client Interceptor* instalado na aplicação servidora causou 3,4% de impacto no desempenho. Vale observar que esse mesmo componente causou o mesmo impacto quando foi instalado na aplicação cliente (dado por Cc). Na verdade, tanto o componente *Profile Client Interceptor* (instalado no

cliente e servidor) quanto o componente *Profile Server Interceptor* apresentaram o mesmo impacto no desempenho.

Observamos, ainda, que o impacto no desempenho quando todos os componentes são instalados (Cenário 2) foi igual a 10,1%, sendo que a implementação *Portable Interceptors* do *Visibroker* contribuiu com 2,9% (Cenário 5), ou seja, 29% do custo total.

## 5.3 ESTUDO DE CASO

A figura abaixo mostra a configuração do ambiente de produção usado para servir o SIGA com o NISUS instalado. Como mostra a Figura 35, ele é formado por 5 servidores Web (SIRA07, 08, 10 e 13), 5 servidores de aplicação (SIRA16, 17, 18, 19 e 20) e 1 servidor de banco de dados (UFRJ02).

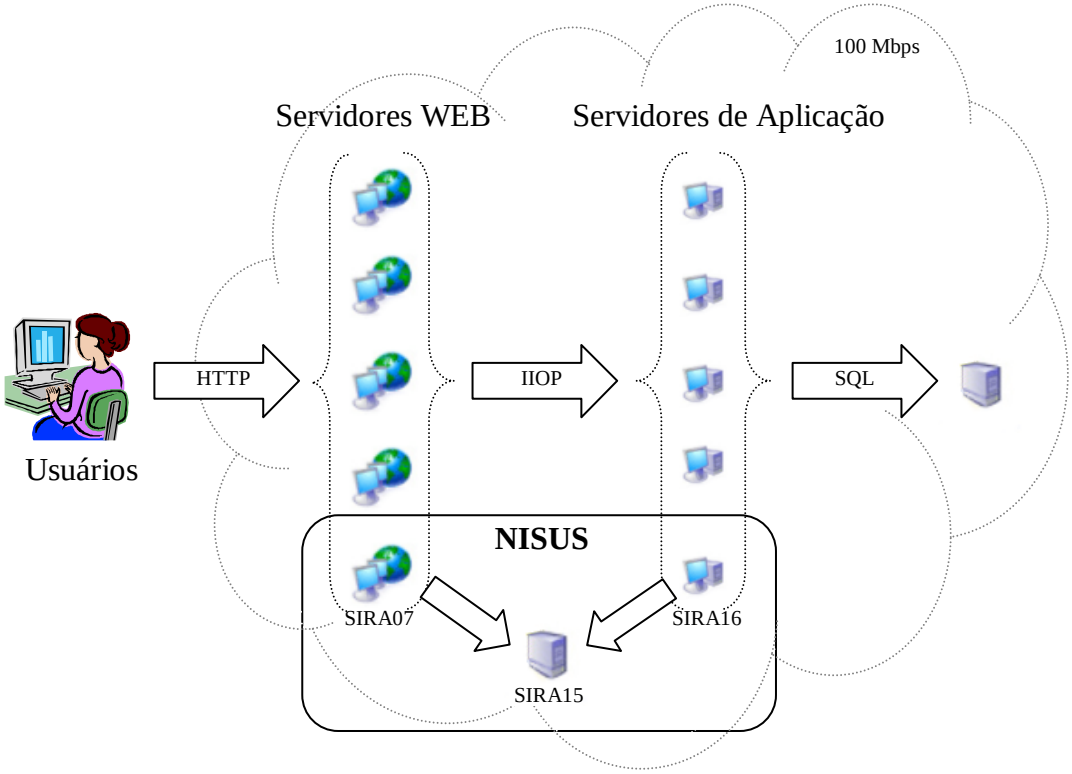


Figura 35 – Ambiente de Produção do SIGA com o NISUS

No SIGA, os servidores Web atuam como “clientes” e os servidores de aplicação atuam como “servidores”. Portanto, o NISUS tem o objetivo de gerenciar o desempenho do das transações realizadas entre os servidores Web e os servidores de aplicação. O NISUS foi instalado em apenas duas máquinas (Figura 35): SIRA07 (cliente) e a SIRA16 (servidor). O banco de dados para a coleta de dados foi instalado na SIRA15, uma máquina diferente do servidor de banco de dados do SIGA. Isso foi feito para evitar que o NISUS interfira no desempenho do SIGA.

A configuração das máquinas que o SIGA utiliza é:

- Hardware: Os servidores Web são máquinas Pentium III 800 Mhz com 512 Mb de RAM. Os servidores de aplicação Pentium III 800 Mhz com 1 Gb de RAM. O servidor de banco de dados é uma máquina Xeon 800 Mhz com 2 Gb de RAM.

- Software: Os servidores de aplicação são configurados com a implementação EJB BES 5.1 (*Borland Enterprise Server*). Os servidores Web são configurados com o RESIN 2.1.5[13]. O servidor de banco de dados é configurado com o SGBD Microsoft SQL Server 2000. Vale observar que o BES utiliza o *Visibroker* como implementação CORBA.

### 5.3.1 IMPLANTAÇÃO

Algumas etapas foram seguidas até a conclusão da instalação do NISUS no SIGA. Primeiramente, foi realizado um trabalho para definir o filtro de transações e os parâmetros de configuração. Em seguida, pelo número elevado de transações, concluiu-se que seria melhor armazenar os dados apenas de um único dia. Para não perder as informações, a cada dia era gerado um arquivo de *dump* da tabela *TransactionData* e o seu conteúdo era excluído. Para isso, foi criado um script no LINUX que era executado no início de cada dia:

[13] Resin é um servidor de páginas JSP e Servlets - http://www.caucho.com.

```
# backup das transações
/usr/local/bin/pg_dump --host=localhost --username=nisus --format p --data-only --
column-inserts --compress=9 --table=transactiondata "--file=/var/spool/nisus/nisus-
`date +%Y%m%d`.sql.gz" nisus

# remove transações
/usr/local/bin/psql -c \
        "DELETE FROM transactiondata" \
        nisus nisus
```

Para permitir a análise detalhada dos dados de todo o período e para não provocar interferência no banco de dados utilizado para a coleta de das informações, os arquivos de *dump* da tabela *TransactionData* foram importados para um outro banco de dados.

Vale observar que o NISUS foi instalado com todos os componentes habilitados. A configuração usada corresponde ao Cenário 2 do experimento.

### 5.3.2 RESULTADOS

O NISUS foi aplicado ao SIGA durante um período de 11 dias. Foram coletadas 107.158 transações (Média de 9.742 transações por dia). Como a SIRA07 foi o único servidor Web onde o NISUS foi instalado, todas essas transações foram executadas a partir dela. Portanto, para se ter uma idéia do número médio de transações do sistema como um todo, basta multiplicar o número médio de transações realizadas a partir de um único servidor Web por 5 (número total de servidores Web). Portanto, o número médio de transações por dia do sistema como um todo é igual a 48.710 (5 x 9.742).

- **Fluxo de Consultas**

Todas as consultas ao banco de dados do SIGA são feitas através de objetos CORBA:

1. Como o sistema baseia-se em EJB, grande parte das consultas é feita através de métodos (com o prefixo “findBy”) de objetos remotos (“Home”).

2. O sistema possui um objeto CORBA (“QueryLookup”) responsável pela execução de consultas.

Portanto, usando o NISUS pode-se conhecer o fluxo de consultas. Levando em consideração as regras acima, concluiu-se que foram executadas aproximadamente 901.565 consultas (81.960 consultas por dia), assumindo todos os servidores. Esse resultado mostra que cada transação realizou, em média, 19 consultas ao banco de dados.

- **Objetos mais utilizados**

A Tabela 6 mostra os objetos CORBA mais utilizados do sistema durante o período analisado.

| Objeto | Transação | Total |
|---|---|---|
| QueryLookupSession | query (String, Map) | 40813 |
| QueryLookupSession | query (String) | 15581 |
| CadastroDeFotoSession | leFoto | 11540 |
| MatriculaOnLineSession | Execute | 6750 |
| InscricaoDisciplinaManagerSession | execute | 1877 |
| MontandoHistoricoSession | montarBoletimXML | 1371 |
| AGFSession | execute | 1095 |
| MontandoHistoricoSession | montarHistoricoXML | 1049 |
| MontandoCridSession | montarCridXML | 911 |
| LookupSession | execute | 888 |
| VaidSession | verificaInscrDisciplinaTrancada | 387 |
| TurmaManagerSession | execute (DadosLogin, String, Document) | 290 |
| MontandoDiarioSession | montarDiarioXML | 220 |

Tabela 6 – Tempo médio de resposta de cada transação[14]

- **Tempo médio de resposta dos objetos CORBA**

A Tabela 7 mostra os objetos CORBA e o tempo médio de resposta de cada transação.

[14] Esta tabela mostra apenas as transações invocadas mais de 200 vezes.

| Objeto | Transação | Total | Média (s) |
|---|---|---|---|
| AutorizarInscricaoSession | autorizarInscricaoCursoData | 1 | 198,967 |
| MontandoBoaSession | montarBoaXML | 63 | 66,891 |
| CadastroSession | execute | 84 | 14,557 |
| InclusaoGrauSession | consultar | 74 | 6,477 |
| AGFAutorizadoSession | execute | 13 | 6,188 |
| AutorizarInscricaoSession | autorizarInscricaoAluno | 24 | 5,757 |
| InscricaoDisciplinaManagerSession | execute (DadosLogin, String, Document, long, String, String) | 1874 | 3,969 |
| InclusaoGrauSession | incluirNotasTurma | 33 | 3,701 |
| MatriculaOnLineSession | execute | 6745 | 3,634 |
| TurmaManagerSession | execute (DadosLogin, String, Document) | 290 | 3,578 |
| CadastroDistribuicaoSession | execute | 3 | 3,010 |
| MontandoDiarioSession | montarDiarioXML | 217 | 2,945 |
| MontandoHistoricoSession | montarBoletimXML | 1369 | 2,353 |
| AGFSession | execute | 1095 | 2,346 |
| MontandoHistoricoSession | montarHistoricoXML | 1047 | 2,322 |
| CadastroCursoSession | execute | 46 | 2,265 |

Tabela 7 – Tempo médio de resposta de cada transação [15].

O método *autorizarInscricaoCursoData* do objeto *AutorizarInscricaoSession* obteve o maior tempo médio de resposta (198 segundos), mas por ser um processo *batch* e trabalhar com uma quantidade bastante alta de dados, esse tempo acaba sendo normal. Portanto, essa transação não foi analisada detalhadamente.

- **Problemas de desempenho encontrados**

O método *montarBoaXML* do objeto *MontarBoaSession* também obteve um tempo médio alto (66 segundos). A Tabela X mostra a participação de cada sub-transação no tempo médio de execução do método *montarBoaXML*, sendo que as colunas 1 e 2 representam: (1) Número médio de chamadas realizadas por execução e (2) Tempo total médio de processamento por execução.

| Método | Média (s) | 1 | 2 (s) | Custo |
|---|---|---|---|---|
| findByAtividadeAcademicaCurriculoLado | 0,017 | 114 | 1.980 | 2,99% |
| findByAtividadeAcademicaCursoLado | 0,016 | 113 | 1.772 | 2,68% |

[15] Esta tabela mostra apenas as transações com duração média acima de 2 segundos.

| | | | | |
|---|---|---|---|---|
| findByAtividadeAcademicaGeralLado | 0,028 | 113 | 3.112 | 4,72% |
| findByAtividadeAcademicaLado | 0,016 | 113 | 1.822 | 2,76% |
| findByCurriculo_oid | 0,016 | 1 | 0.016 | 0,02% |
| findByEquacaoEquivalenciaLado | 0,355 | 47 | 16.694 | 25,29% |
| query (String, java.util.Map) | 0,281 | 78 | 21.796 | 33,02% |
| Outros procedimentos | - | - | 18.813 | 28,5% |

Tabela 8 – Custo de processamento de cada sub-transação invocada no contexto do método *montarBoaXML*.

Por exemplo, a sub-transação *findByEquacaoEquivalenciaLado* obteve 0.355 segundos como tempo médio de execução. Ele é invocado 47 vezes, em média, para cada execução do método *montarBoaXML*, contribuindo, assim, com 16.694 segundos (25.29%) do tempo médio de execução do método *montarBoaXML*. Observa-se que foram realizadas, em média, 578 (soma da coluna 2) consultas ao banco de dados por chamada a essa operação.

Analisando os dados coletados pelo NISUS, verificou-se que o método *findByEquacaoEquivalenciaLado*, em alguns casos, levou em torno de 12 s para ser executado, ou seja, um valor bastante alto em relação à sua média. Esse método executa uma consulta no banco de dados e foi verificado que estava faltando um índice em uma das tabelas no banco de dados.

A maior parte dos diagramas de seqüência de desempenho dessa transação apontou a sub-transação *create (BeanDocument)* como o gargalo. A Figura 36 mostra um dos Diagramas de Seqüência de Desempenho da transação *execute* do objeto *CadastroSession*. Como pode ser visto, a transação obteve tempo de resposta (11,37 segundos) próximo da média (14 segundos), sendo que a chamada ao método *create (BeanDocument)* consumiu 11 segundos, ou seja, praticamente todo o tempo de resposta da transação.

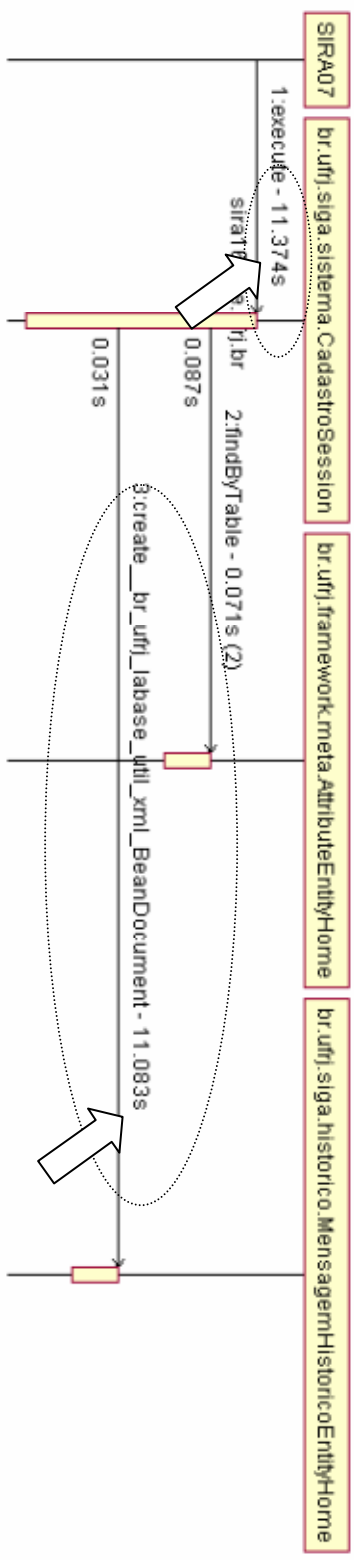


Figura 36 – Diagrama de Seqüência de Desempenho da transação *execute* do objeto *CadastroSession*.

A transação *execute* do objeto *AGFAutorizadoSession* teve média de execução razoavelmente alta (igual a 6,188 segundos) em função do tempo de execução excessivo (48 segundos) de uma das chamadas. Analisando os Diagramas de Seqüência de Desempenho, percebe-se que duas sub-transações contribuíram bastante para esse tempo, como mostra a Tabela 9.

| **Objeto** | **Sub-Transação** | **Tempo (s)** |
|---|---|---|
| DisciplinaHistoricoEntityHome | findBySegmentoHistorico | 28 |
| VaidManagerSession | addAluno | 11 |

Tabela 9 – Sub-transações mais custosas da transação *execute* do objeto *AGFAutorizadoSession*.

De acordo com as informações coletadas pelo NISUS, observou-se que esta chamada foi executada juntamente com outras 9 transações.

O método *addAluno* do objeto *VaidManagerSession* também apresentou um tempo elevado (249 segundos) quando foi executado dentro de uma das chamadas ao método *execute* do objeto *AGFSession*. Esta transação foi executada juntamente com outras 8 transações e também apresentou um tempo de execução bastante expressivo: 254 segundos.

Analisando o código-fonte do método *addAluno*, percebeu-se que ele faz uma consulta ao banco de dados, onde foi detectado a falta de um índice em uma das tabelas do banco de dados do SIGA.

## 5.4 DISCUSSÃO

Com o experimento, pôde-se ter maior controle sobre o NISUS. Através dele, foi possível identificar o impacto causado no desempenho por cada um de seus componentes.

O NISUS ajudou a identificar problemas de desempenho em algumas de suas funcionalidades do SIGA, extraindo informações bem detalhadas sobre ele e apontando com precisão a origem dos problemas. Além disso, não foi percebida nenhuma variação de desempenho entre as máquinas onde os componentes foram instalados em relação às outras, mostrando a viabilidade de coletar e analisar informações detalhadas sem causar danos ao desempenho do sistema.

Na verdade, poucos relatórios foram implementados. As análises sobre o estudo de caso foram feitas através de consultas à tabela *TransactionData* e através dos Diagramas de Seqüência de Desempenho. Os relatórios implementados serviram apenas como exemplo e não foram importantes para a análise do estudo de caso.

Durante o estudo de caso, o mecanismo de coleta de dados mostrou-se bastante eficiente. Em momentos de pico, o mecanismo chegou a gravar em disco mais de 50.000 objetos em apenas um servidor. Se todos esses dados estivessem em memória, o desempenho do servidor poderia ter sido prejudicado. Nesse sentido, o módulo de gerenciamento on-line de carga e recursos foi importante para ter uma visão da situação real das máquinas usadas no estudo de caso.

Durante a fase de implementação do NISUS, os testes de unidade reduziram bastante o tempo na detecção de erros no funcionamento dos seus componentes. O uso de *Design Patterns* trouxe maior flexibilidade ao projeto. Várias API's SNMP, por exemplo, puderam ser testadas sem a necessidade de mudanças na implementação do NISUS. Além disso, o uso do *Design Pattern Composite* (GAMMA *et al* 1999) ajudou a projetar uma solução simples e intuitiva para um problema complexo: a criação da hierarquia das transações em tempo real.

# 6 CONCLUSÃO

O NISUS representa uma séria infra-estrutura de gerenciamento de desempenho de sistemas distribuídos pela simplicidade, flexibilidade e por não exigir extensas customizações. Além disso, a forma como os componentes do NISUS coordenam as suas atividades e coletam os dados sobre o desempenho dos sistemas distribuídos CORBA representam um aspecto original no trabalho apresentado.

O NISUS utiliza-se de *CORBA Portable Interceptors* para prover transparência no gerenciamento de desempenho. Dessa forma, ele pode ser aplicado a qualquer sistema distribuído baseado em CORBA e Java sem a necessidade de modificações no código-fonte do sistema.

Foi criada uma implementação da arquitetura e esta se encontra disponível como um projeto de código-aberto[16]. Um dos grandes desafios da implementação foi construir os componentes de tal forma que eles não prejudicassem o desempenho dos sistemas distribuídos. Para isso, o NISUS utiliza um mecanismo assíncrono que combina distribuição e centralização dos dados coletados.

Com um experimento, este trabalho mostrou que o impacto no desempenho provocado pelo NISUS é baixo. Através de um estudo de caso, a implementação criada mostrou-se capaz de apontar com precisão a origem de problemas de desempenho.

Por fim, conclui-se que o NISUS atende aos principais requisitos de gerenciamento de desempenho, tais como transparência e filtro de informações. Ele permite ainda o gerenciamento microscópico, com os Diagramas de Seqüência de Desempenho, e gerenciamento macroscópico, com gráficos e relatórios consolidados.

## 6.1 PERSPECTIVAS FUTURAS

[16] http://nisus.sourceforge.net

O NISUS não representa uma solução completa para a questão de gerenciamento de desempenho. Outras aplicações de *Portable Interceptors* mencionados nesse trabalho Por exemplo, o CORBA não padroniza nenhum mecanismo de distribuição de carga (OTHMAN *et al*, 2001, p. 2). Como conseqüência, muitas das suas implementações não provêm este importante mecanismo. Além disso, o NISUS poderia ser usado para descobrir operações que não estão sendo mais utilizados pelo sistema e para gerar documentação automática do sistema. Portanto, diversas outras aplicações podem ser futuramente agregadas ao NISUS.

Poderia ser desenvolvido um serviço genérico de consultas no *Web Console*. Isso permitiria maior flexibilidade e diminuiria a necessidade de desenvolvimento de novos relatórios.

Por fim, mecanismos sofisticados de gerenciamento de sistemas distribuídos poderiam ser agregados; não apenas referentes ao desempenho. A única função pró-ativa do NISUS é o envio de e-mail caso a temperatura ultrapasse os limites especificados. Outras funções como, reiniciar a máquina quando necessário poderia ser implementado. Além disso, os componentes poderiam descobrir, por conta própria, problemas de desempenho e reportá-los aos desenvolvedores e administradores. Por estas razões, a necessidade de intervenção humana seria reduzida.

# REFERÊNCIAS

AMBLER, S. W., "Análise e Projeto Orientados a Objeto", Editora IBPI Press, RJ, 1998.

BECK, K., “Extreme Programming Explained: embrace change”. Editora Addison Wesley, 2000.

CANTÙ, M., “Dominando o Delphi 4”, Editora Makron Books, SP, 1998.

CORBATRACE, 2003, http://corbatrace.tuxfamily.org.

DEBUSMANN, M. *et al*. F.: Measuring End-to-End Performance of CORBA Applications using a Generic Instrumentation Approach. In: IEEE INTERNATIONAL SYMPOSIUM ON CONPUTERS AND COMMUNICATIONS, 7., 2002, Taormina-Giardini Naxos, Italy, p.181-187.

_____;_____;_____. Generic performance instrumentation of EJB applications for service-level management. In: IEEE/IFIP NETWORK OPERATION AND MANAGEMENT SYMPOSIUM, 2002, pp. 19-32.

DENNIS, R. Monitoring vs. Managing: Few network management tools actually manage anything. **Server World Magazine**. Jan. 2003. Disponível em: http://www.serverworldmagazine.com/monthly/2003/01/monitor.shtml. Acesso em: jan. 2004.

DE PAUW, W., LORENZ, D., VLISSIDES, J., WEGMAN, M. Execution patterns in object-oriented visualization. In: PROCEEDINGS OF THE FOURTH USENIX CONFERENCE ON OBJECT-ORIENTED TECHNOLOGIES AND SYSTEMS, Abr. 1998, pp. 219-234.

EDWARDS, W.; DESHPANDE. **The Server Side**. Jan. 2001. Disponível em: http://www2.theserverside.com/resources/pdf/CorbaEJB.pdf. Acesso em: nov. 2003.

FICHE, R.; OLIVEIRA, C. E. T. A Transparent and Centralized Performance Management Service for CORBA based Applications. In: IEEE/IFIP NETWORK OPERATION AND MANAGEMENT SYMPOSIUM, 2004, Korea, p 439-452.

FRIEDMAN, R.; HADAD, E. Client-side Enhancements using Portable Interceptors. In: INTERNATIONAL WORKSHOP ON OBJECT-ORIETED REAL-TIME DEPENDABLE SYSTEMS, 6., 2001, Rome, p. 179.

GAMMA, E., HELM, R., JOHNSON, R., VLISSIDES, J.: Design Patterns - Elements of Reusable Object-Oriented Software, 1999.

GARTNER GROUP, 2003, http://www.gartner.com.

HENNING , M.: Binding, migration, and scalability in CORBA. **Communications of the ACM**, v. 41, pp. 34-36, Out. 1998.

MIDDLEWARE COMPANY. Optimizing Java Applications. **The Server Side**. Mar. 2003. Disponível em: http://www.theserverside.com/resources/article.jsp?l=J2EEOptimize. Acesso em: nov. 2003.

HAUCK, R, RADISIC, I. Service Oriented Application Management – Do current techniques meet the requirements? In: IFIP INTERNATIONAL WORKING CONFERENCE OF DISTRIBUTES APPLICATIONS AND INTEORPERABLE SYSTEMS, 3., 2001.

YEMINI, Y. The OSI Network Management Model, **IEEE Communications Magazine**, v.31, pp. 20-29, Mai. 1993.

IETF, User Datagram Protocol, RFC 768. 1980.

_____. Simple Gateway Monitoring Protocol, RFC 1028,1987

_____. A Simple Network Management Protocol (SNMP), RFC 1157. Mai 1990. Disponível em: http://www.ietf.org/rfc/rfc1157.txt?number=1157. Acesso em: dez 2003.

_____. IAB recommendations for the development of Internet network management standards, RFC 1052. 1988.

_____. Structure and identification of management information for TCP/IP-based internets, RFC 1065. 1988.

_____. Management Information Base for network management of TCP/IP-based internets, RFC 1066. 1988.

_____. Simple Network Management Protocol, RFC 1067. 1988.

_____. Management Information Base for network management of TCP/IP-based internets, RFC 1156, 1990.

_____. A Simple Network Management Protocol (SNMP), RFC 1157. Mai 1990. Disponível em: http://www.ietf.org/rfc/rfc1157.txt?number=1157. Acesso em: dez 2003.

_____. Management Information Base for Network Management of TCP/IP-based internets: MIB-II, RFC 1213. 1991.

MARCHETTI, C. *et al.* CORBA Request Portable Interceptors: A performance Analysis. In: INTERNATIONAL SYMPOSIUM OF DISTRIBUTED OBJECTS AND APPLICATIONS, 3., 2001, Rome, p. 208.

MOE, J., CARR, D. A.: Understanding Distributed Systems via Execution Trace Data. In: IEEE INTERNATIONAL WORKSHOP ON PROGRAM COMPREHENSION, 9., 2001, Toronto, Canada, p. 60-67.

MOS, A., MURPHY, J. Performance Monitoring of Java Component-Oriented Distributed Applications. In: IEEE INTERNATIONAL CONFERENCE ON SOFTWARE, TELECOMMUNICATION AND NETWORKS, 2001, Croatia, Italy.

NARASIMHAN, Priya; MOSER, Louise E.; MELLIAR-SMITH, P. M. Using Interceptors to Enhance CORBA. **IEEE Computer**. v. 32, n. 7, pp. 62-68. jul. 1999.

OTHMAN, Ossama; O'RYAN, Carlos; SCHMIDT, Douglas C. Strategies for CORBA Middleware-Based Load Balancing. **IEEE Distributed Systems Online**. v. 2. n. 3, Mar. 2001. Disponível em: http://dsonline.computer.org/0103/features/oth0103_print.htm. Acesso em: nov. 2002.

_____;_____;_____. Designing an Adaptive CORBA Load Balancing Service Using TAO. **IEEE Distributed Systems Online**. v. 2. n. 4. Abr. 2001. Disponível em: http://dsonline.computer.org/0104/features/oth0104_print.htm. Acesso em: nov. 2002.

POA (Portable Object Adapter) Specification, Object Management Group, 1997.

SCALLAN, T. Monitoring and Diagnostics of CORBA Systems. **Java Developers's Journal**, v. 5, pp. 138-144, 2000.

SCHWEITZER, C. M. Informações de Desempenho e Acordos de Nível de Serviço para Redes de Transporte PDH e SDH. 1999. 109 p. Dissertação (Mestrado em Ciências) – Curso de Pós-Graduação em Engenharia Elétrica e Informática Industrial, Centro Federal de Educação Tecnológica, Paraná.

SOUDER, T., MANCORIDIS, S., SALAH, M.: Form: A Framework for Creating Views of Program Executions. In: IEEE Proceedings of the 2001 International Conference on Software Maintenance (ICSM'01), Florença, Itália, pp. 612-620, Nov. 2001.

SRIDHARAN, B., DASARATHY, B., MATHUR, A. P. On building non-intrusive performance instrumentation blocks for CORBA-based distributed systems. In: IEEE COMPUTER PERFORMANCE AND DEPENDABILITY SYMPOSIUM, pp. 139 - 143, 2000.

STALLINGS, W. SNMP and SNMPv2: The Infrastructure for Network Management, **IEEE Communications Magazine**, v.36, n.3, pp. 37-43, Mar.1998.

_____. “SNMP, SNMPv2,SNMPv3, and RMON 1 and 2”, Editora Addison-Wesley, 1999.

TANENBAUM, A. S., "Sistemas Operacionais Modernos". Editora LTC, 1999.

VANHELSUWÉ, L. Profiling the Profilers. **Java World**. Ago. 2003. Disponível em: http://www.javaworld.com/javaworld/jw-08-2003/jw-0822-profiler-p1.html. Acesso em: Out. 2003.

WANG, N., PARAMESWARAN, K., SCHMIDT, D. C.: The Design and Performance of Meta-Programming Mechanisms for Object Request Broker Middleware. In: Proceedings of the 6 th Conference on Object Oriented Technologies and Systems. San Antonio, Texas, USA, Jan 2000.