BRUNO NOVAIS DE PAULA SILVA

COMUNICAÇÃO REMOTA EM UMA ARQUITETURA DE SUPORTE A SIMULAÇÃO DISTRIBUÍDA

LAVRAS - MG
2010

BRUNO NOVAIS DE PAULA SILVA

COMUNICAÇÃO REMOTA EM UMA ARQUITETURA DE SUPORTE A SIMULAÇÃO DISTRIBUÍDA

Monografia de graduação apresentada ao Departamento de Ciência da Computação da Universidade Federal de Lavras como parte das exigências do curso de Ciência da Computação para obtenção do título de Bacharel em Ciência da Computação.

Orientador:

Dra. Marluce Rodrigues Pereira

Co-orientador:

Dr. Bráulio Adriano de Mello

LAVRAS - MG
2010

BRUNO NOVAIS DE PAULA SILVA

COMUNICAÇÃO REMOTA EM UMA ARQUITETURA DE SUPORTE A SIMULAÇÃO DISTRIBUÍDA

Monografia de graduação apresentada ao Departamento de Ciência da Computação da Universidade Federal de Lavras como parte das exigências do curso de Ciência da Computação para obtenção do título de Bacharel em Ciência da Computação.

APROVADA em ____ de _____ de _____

Dr. Cláudio Fabiano Motta Toledo UFLA

Dr. Tales Heimfarth UFLA

Dra. Marluce Rodrigues Pereira

Orientadora

Dr. Bráulio Adriano de Mello

Co-orientador

LAVRAS - MG
2010

Aos meus pais Paulo e Alcione primeiramente.

A meu irmão que me deu apoio durante toda uma jornada.

A minhas tias Néia e Shirley.

A meus tios Anízio e Meire.

A meus tios Antônio e Rita.

A todos os meus familiares em especial.

DEDICO.

AGRADECIMENTOS

Agradeço a professora e orientadora Marluce, que confiou em mim e me deu o apoio necessário para concluir este trabalho.

Agradeço ao professor Bráulio, por ter me orientado durante um período de trabalho, e por posteriormente ter aceitado ser meu co-orientador me ajudando bastante.

A todos os outros professores do DCC – UFLA, que durante estes quase quatro anos conseguiram fazer com que eu aprendesse muito.

A meus amigos de curso, que estiveram presentes em todos os momentos.

Aos meus companheiros de trabalho na Tbit, que ao longo do último ano se mostraram grandes amigos e companheiros.

A minha família. Agradeço a todos vocês, os que estão próximos e os que estão distantes, vocês são os responsáveis por me fazer chegar aqui.

Em especial a minha mãe Alcione, meu pai Paulo e meu irmão Marcos, que me deram apoio em todas as decisões que eu tomei, que me aconselharam sempre que eu tive dúvidas e que me deram suporte sempre que eu tive dificuldades.

RESUMO

O Gerenciador de Repositórios do *Distributed Co-Simulation Backbone* (DCB) é um sistema desenvolvido na linguagem Java que auxilia os projetistas de modelos de simulação a criar seus ambientes de simulação. O gerenciador existente, embora funcional, necessita de algumas melhorias. Uma delas é prover a comunicação entre diferentes repositórios localizados em um sistema distribuído (em diferentes máquinas), que é a motivação deste trabalho. Existem diferentes tecnologias para realizar comunicação em problemas distribuídos, como sockets, SOAP, CORBA e RMI. Comparações de desempenho para sistemas desenvolvidos em Java que utilizam estas diferentes tecnologias indicam que RMI (*Remote Method Invocation*) é uma boa solução. RMI consiste em uma interface de programação que provê a execução de chamadas remotas em aplicações desenvolvidas em Java. Neste trabalho, a comunicação entre os diferentes repositórios do Gerenciador de Repositórios foi implementada utilizando RMI e funcionando em um modelo cliente-servidor. O modelo implementado permite aos projetistas terem acesso a diferentes elementos de simulação que são gerenciados por outros tutores, de forma transparente, sem exigir conhecimentos específicos de comunicação entre computadores.

Palavras-chave**:** DCB, Comunicação remota, RMI, Programação distribuída, Simulação Computacional.

ABSTRACT

The Repository Manager of Distributed Co-Simulation Backbone (DCB) is a system developed in Java language that helps designers of simulation models to create their simulation environments. Although functional, the manager needs some improvements. One of them is to provide communication between different repositories located in a distributed system (in different machines), that is the motivation of this work. There are different solutions for provide communication in distributed system, as sockets, SOAP, CORBA and RMI. Performance comparisons for systems developed in Java using these different technologies shows that RMI (Remote Method Invocation) is a good solution. RMI consists of a programming interface that provides the execution of remote calls in applications developed in Java. In this work, the communication between the different repositories of Repository Manager was implemented using RMI and working in a client-server model. The implemented model allows designers to have access to different elements of simulation that are managed by other tutors, in a transparent manner, and without requiring specific knowledge of communication between computers.

Keywords*:* DCB, Remote Comunication, RMI, Distributed Programming, Computer Simulation.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS

API	*Application Programming Interface*
CORBA	*Common Object Request Broker Architecture*
DCB	*Distributed Co-Simulation Backbone*
IDL	*Interface Definition Language*
J2SDK	*Java 2 Software Development Kit*
JVM	*Java Virtual Machine*
ORB	*Object Request Broker*
RMI	*Remote Method Invocation*
SOAP	*Simple Object Access Protocol*

SUMÁRIO

1 INTRODUÇÃO

1.1 Contextualização e Motivação

Nos últimos tempos, a humanidade tem passado por uma enorme evolução ocorrida principalmente devido aos avanços tecnológicos e da ciência, que trouxeram tanto incontáveis benefícios quanto preocupações. Aliado a estas tecnologias estão os sistemas computacionais que estão cada vez mais necessários e presentes no nosso cotidiano. Quanto a estes sistemas computacionais as preocupações referem-se principalmente a investimentos e tempo despendido no desenvolvimento de novos sistemas computacionais (Bruschi 2002).

Em termos computacionais essa evolução aconteceu de forma mais significativa nas últimas quatro décadas, e umas das áreas quem vem se destacando é a avaliação de desempenho. A avaliação de desempenho tem grande importância devido aos objetivos de minimizar ou eliminar algumas dessas preocupações. Assim, pode-se fazer a avaliação de desempenho tanto em sistemas existentes quanto em sistemas não existentes. Para se fazer essa avaliação em sistemas existentes pode-se utilizar técnicas de aferição, onde as informações requeridas por um estudo podem ser obtidas a partir do próprio sistema. Para sistemas não existentes há técnicas de modelagem onde se constrói um modelo representativo do sistema (Bruschi 2002).

A escolha por uma destas técnicas vai depender do objetivo da avaliação, do sistema a ser avaliado e do nível de detalhe a ser considerado. Isso porque existem diferenças entre as abordagens. Nas técnicas de aferição é necessário que o sistema a ser avaliado esteja concluído para se extrair as informações buscadas. Já as técnicas de modelagem baseiam-se na construção de um modelo que represente as características a serem analisadas no sistema real. As soluções para este último modelo podem ser obtidas através de soluções analíticas ou por simulação. Embora a primeira solução seja bastante atrativa

devido à precisão dos resultados, não possui muita flexibilidade e há uma grande dificuldade em representar as características essenciais de alguns sistemas computacionais (Brushi 2002). Neste trabalho será ressaltada a segunda forma de solução: a simulação.

A simulação abstrai o sistema a um modelo e a partir deste modelo um programa de simulação é gerado. Segundo Brushi (2002), esta técnica oferece grande flexibilidade e baixo custo, pois qualquer alteração que queira ser feita ao sistema se mostra com maior facilidade no programa de simulação. Mello (2005) apresenta argumentos que viabilizam ainda mais o uso desta técnica. Segundo ele, a validação e a fase de testes de funcionamento de um novo sistema, normalmente ocorrem apenas nas etapas finais do projeto (implementação). Este retardo na validação pode adiar a descoberta de problemas, que se detectados em fases iniciais, poderiam ser resolvidos com menor esforço/custo. Isso ressalta a importância do ambiente de simulação. Esse ambiente se mostra bastante eficaz quando se refere a validações e testes intermediários principalmente na detecção de erros precocemente, e de forma que possam ser solucionados de modo mais fácil e ágil.

Surge, com a evolução da tecnologia da informação e suas aplicações em processos, o conceito e a necessidade de integração entre as interfaces de hardware e software, que segundo Mello (2005) seria o escopo da simulação. Não apenas esta diferença, como também a diferenciação do sistema em termos de tecnologia, linguagens utilizadas e os diferentes níveis de abstração constituem em um contexto geral o que chamamos de sistemas heterogêneos. Podemos definir então em termos mais gerais, que “sistemas que combinam componentes diferentes que precisam cooperar são chamados heterogêneos” (Hubert 1998 *apud* Mello 2005).

Com o crescimento das redes de computação e da tecnologia da informação, fica então possível pensar em um sistema descentralizado que

funcione de maneira cooperativa, ou seja, uma integração de sistemas heterogêneos. Porém, segundo Junqueira (2009), esta integração de sistemas heterogêneos aumenta a complexidade na tarefa de analisar o sistema e requer o desenvolvimento de novas soluções. Vale ressaltar o uso da simulação distribuída, que trata da execução de programas computacionais em máquinas dispersas, posicionadas em diferentes locais geográficos e conectadas através de uma rede de computadores.

Existe atualmente uma arquitetura de co-simulação distribuída e heterogênea, que trata dos principais tópicos sobre simulação heterogênea, o *Distributed Co-Simulation Backbone (*DCB). Por admitir diferentes linguagens e que diferentes noções temporais sejam implementadas e trabalhem em conjunto, ele pode ser considerado heterogêneo. Baseado no *High Level Architecture* (HLA), que é um padrão IEEE para arquitetura de sistemas de simulação distribuídos Fujimoto (2001), o DCB será a arquitetura de co-simulação a qual este trabalho irá abordar.

O DCB tem como um dos objetivos facilitar o trabalho de construção de modelos heterogêneos de simulação. Para isso ele faz uso de técnicas de cooperação entre elementos heterogêneos de simulação, de forma com que seja mantida a integridade do elemento (Mello 2005). O objetivo principal do DCB é a independência dos elementos com o restante do modelo, comunicando-se apenas com seu gateway, não realizando atividades como a execução de atividades remotas, já que isso não é sua tarefa (Strassburger 1998).

Com a tarefa de localizar e gerenciar elementos foi criado um gerenciador de repositórios distribuídos de elementos. Esse gerenciador de repositórios permite agilizar a construção de modelos para a execução na arquitetura do DCB (Sá e Mello 2010).

1.2 Objetivos do Trabalho

Este trabalho apresenta a especificação e o desenvolvimento de um modelo de comunicação remota para o gerenciador de repositórios de elementos para que ele passe a trabalhar de forma descentralizada (em rede), e de forma transparente ao usuário (projetista) do modelo. Isso implica que a partir de uma máquina seja possível acessar elementos que estão armazenados em outras máquinas, e de tal forma com que o projetista do modelo de simulação não tenha a necessidade de conhecer informações da rede, passando esta competência então ao gerenciador de repositórios distribuídos de elementos. Esta tarefa será feita de forma implícita pelo gerenciador, provendo então a transparência de localização do elemento, por parte do projetista.

1.2.1 Objetivo Geral

O objetivo deste trabalho é a implementação de um modelo de comunicação remota que permita a comunicação entre diferentes Repositórios de Elementos do DCB, utilizando uma API Java chamada RMI (*Remote Method Invocation)*.

1.2.2 Objetivos Específicos

O trabalho ainda possui objetivos específicos, que são:

- Analisar alguns modelos e técnicas que possam prover a comunicação e tornar o sistema distribuído e transparente ao usuário (projetista).
- Comparar os modelos de comunicação remota, quanto às características e desempenho, de forma que seja possível aferir qual a melhor opção a ser aplicada no Gerenciador.

1.3 Estrutura do Trabalho

Este trabalho está estruturado da seguinte maneira:

- O Capítulo 2 apresenta o Referencia Teórico, mostrando os principais conceitos utilizados no trabalho e as soluções para acesso remoto a objetos.
- O Capítulo 3 mostra a Metodologia do trabalho, mostrando as características da pesquisa, e a forma como foram feitas as implementações.
- O Capítulo 4 tem os Resultados obtidos neste trabalho.
- O Capítulo 5 finaliza o trabalho apresentando as conclusões, discutindo os resultados obtidos e propondo trabalhos futuros.

2 REFERENCIAL TEÓRICO

O objetivo deste capítulo é apresentar os principais conceitos utilizados neste trabalho. São abordados os conceitos de simulação computacional, simulação distribuída, o *Distributed Co-Simulation Backbone* (DCB), o gerenciador de repositórios distribuídos e as soluções existentes na literatura para acesso remoto a objetos: *sockets*, CORBA, SOAP e RMI.

2.1 Simulação Computacional

Algumas definições para simulação computacional podem ser encontradas na literatura. A simulação computacional é um processo de experimentos em sistemas ou fenômenos físicos, realizados através de modelos matematicamente computadorizados, os quais representam características observadas em sistemas reais (Chung 2004). A idéia então é de tentar representar uma situação real em forma de um modelo computacional.

Simulação é o processo de criar um modelo computacional para representar o comportamento de um sistema real (Kelton 2007). A simulação pode ser entendida como a recriação de um sistema real em um ambiente controlado, o que possibilita entender, manipular e verificar o processo em etapas, examinando suas formas de operação mais seguramente, sem a necessidade de verificar no sistema real, e de forma a reduzir custos, quando comparados aos que seriam necessários em análises com modificações no sistema produtivo real.

2.2 Simulação Distribuída

Existem duas maneiras de se fazer uma simulação: seqüencial ou distribuída. A segunda maneira vem ganhando interesse especial pelo intuito de diminuir o tempo de uma simulação computacional (Bruschi 2002 *apud* Carvalho 2009). Não apenas por este motivo, mas também por ter benefícios de potencial como o desenvolvimento descentralizado de projetos (Borrielo 2000 *apud* Mello 2005). Com isso, é possível executar um único modelo de simulação em ambientes computacionais distribuídos.

Na simulação distribuída duas abordagens se destacam: SRIP (*Single Replication in Parallel);* e MRIP *(Multiple Replication in Parallel).* No primeiro, os processos lógicos são particionados e mapeados em um processador, com a comunicação feita através de mensagens. Enquanto na segunda abordagem, as replicações de um mesmo programa são executadas independentemente em paralelo (Carvalho 2009).

2.3 Distributed Co-Simulation Backbone

Com o propósito mais geral de dar suporte à execução distribuída de modelos heterogêneos, o DCB é distribuído em módulos, que executam tarefas específicas, aumentando a coesão e diminuindo o acoplamento (Mello 2005 *apud* Carvalho 2009).

A Figura 1 apresenta a arquitetura do DCB destacando cada um dos módulos. Dentre estes módulos, estão: o Gateway que tem função bem definida de tratamento de interfaces dos componentes; o Embaixador do Federado (EF) que tem como propósito geral o gerenciamento de mensagens recebidas; o Embaixador do DCB (EDCB) que tem como propósito geral o envio de

mensagens; e o Núcleo do DCB (NDCB) que além de manter os serviços de comunicação por troca de mensagens, mantém o tempo virtual global (GVT- *Global Virtual Time*) atualizado. O EDCB em conjunto com o EF, também mantém o gerenciamento de tempo virtual local (LVT – *Local Virtual Time*) e o modo de sincronização utilizado por cada federado para cooperar com a federação (Mello 2005).

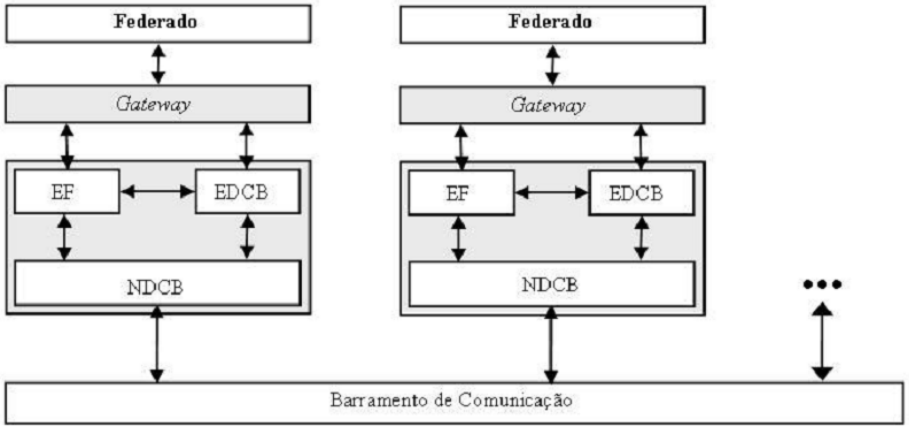

Figura 1 Arquitetura do DCB. Fonte: Mello 2005.

2.4 Gerenciador de Repositórios Distribuídos

O gerenciador de repositórios distribuídos é um ambiente de gerenciamento de repositórios de elementos que foi desenvolvido com o principal objetivo de localizar e gerenciar elementos para que possam ser reconhecidos e reutilizados em novos modelos. O gerenciador de repositórios permite agilizar a construção de modelos para execução na arquitetura do DCB (Mello e Parreiras 2009).

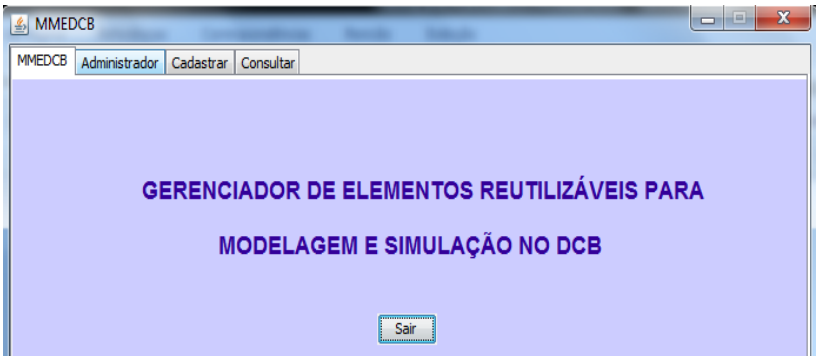

Figura 2 Gerenciador de Repositórios Distribuídos

A Figura 2 apresenta a interface inicial do Gerenciado de Repositórios Distribuídos. Essa ferramenta exerce diversas funções como: operações sobre cadastro de usuários; operações de cadastro de elementos; configuração de elementos; entre outras. As operações sobre o cadastro de usuários que podem ser feitas são: a inserção de novos usuários, remoção, definição de usuários como tradicionais ou administradores. Esta última determina as ações que podem ser realizadas por cada tipo de usuário, permitindo assim mais opções de ações aos usuários administradores, e colocando algumas restrições aos usuários tradicionais, como a remoção de dados de outros usuários, e acesso a outras informações que estão restritas aos administradores.

As operações de cadastro de elementos permitem tanto a inserção de elementos com suas informações quanto à remoção. Essas informações podem ser nome, descrição do elemento, nível de restrição, tipo (síncrono ou assíncrono) e características do elemento. A figura 3 ilustra a interface do DCB e a tela que manipula a parte de cadastro de elementos.

Para a função de configuração dos elementos, Sá e Mello (2010) propuseram um modelo de geração automática de configurações dos elementos, que antes era feito manualmente pelo usuário e projetista do modelo, e exigia

então a necessidade de conhecimento da arquitetura do DCB por parte do usuário. Com essa proposta, a realização desta tarefa passou então ser de forma automática, isentando então o usuário de tal função.

O cadastro de um elemento envolve diversas informações associadas a ele como: sincronia, atributos de entrada e saída, porta de entrada, IP, entre outras citadas anteriormente. Isso faz com que o sistema necessite de uma padronização para que seja modelado de forma completa, sendo muito recomendável que essa modelagem, como por exemplo, o cadastro de elementos seja feita no Gerenciador de Repositórios Distribuídos. Dessa forma as informações que se referem aos elementos estarão registradas em arquivos no formato XML (característica do Gerenciador), o que torna mais fácil o carregamento dessas informações para qualquer tipo de manipulação. Assim, respeitam-se os princípios de modelagem da arquitetura utilizada.

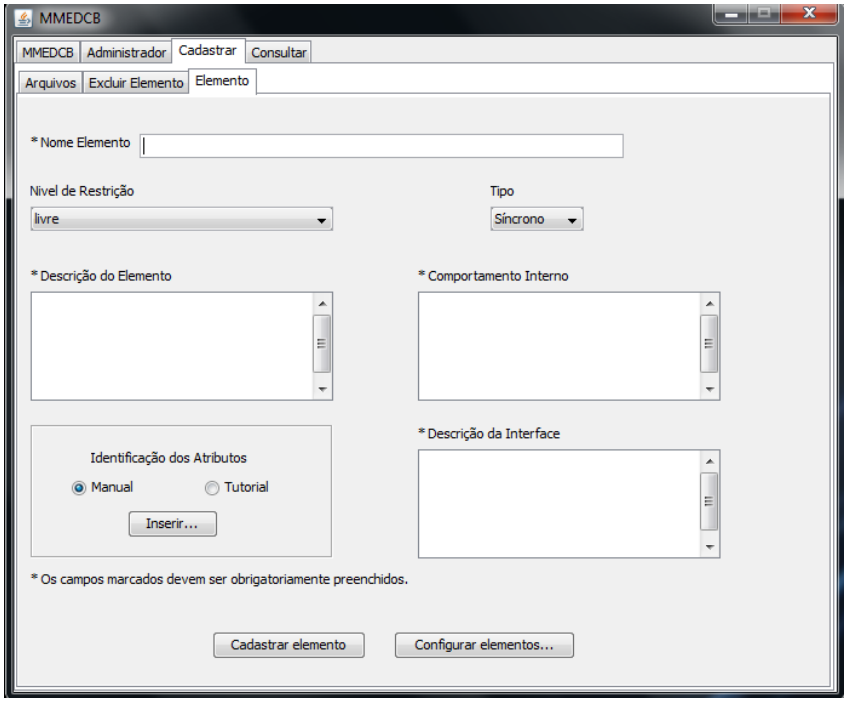

Figura 3 Tela de cadastro de elementos do Gerenciador de Repositórios Distribuídos.

2.5 Soluções Encontradas na Literatura para acesso remoto a objetos

Atualmente encontram-se disponíveis na literatura algumas alternativas para o acesso remoto a objetos, como: *Sockets; Remote Method Invocation* (RMI); *Common Object Request Broker Architecture* (CORBA); *Simple Object Access Protocol* (SOAP); entre outras. Essas alternativas serão mostradas a seguir, apresentando uma introdução e uma descrição de como cada uma delas realiza a função de comunicação entre sistemas.

2.6 *Sockets*

A comunicação de processos é algo muito importante para diversas aplicações. Para isso existe a programação com o uso de *sockets*, que são justamente os elos de ligação entre os processos de um servidor e de um cliente. Pode-se entender por modelo cliente/servidor, a comunicação onde todas as requisições são feitas pelo usuário e respondidas pelo servidor. Kurose (2003) definiu *socket* como sendo uma "interface entre a camada de aplicação e a de transporte dentro de uma máquina".

Diversas aplicações cliente/servidor surgiram então, onde os clientes e o servidor poderiam estar em máquinas diferentes, distantes umas das outras e se comunicando. Porém, estas aplicações necessitam de utilizar protocolos de transporte para se comunicarem. Quando um aplicativo interage com o software de protocolo, ele deve especificar detalhes, como por exemplo, se é um servidor ou um cliente. Além disso, os aplicativos que se comunicam devem especificar detalhes adicionais (Kurose 2003).

Os protocolos de transporte existentes via *sockets* são: o TCP que é orientado a conexão e tem um transporte de dados confiável; e o UDP que não é orientado a conexão e por isso não é um protocolo confiável de envio de dados.

O protocolo TCP além de ser confiável, tem implementado o controle de fluxo da rede, que garante que nenhum dos lados tanto cliente quanto servidor fiquem sobrecarregados com o envio demasiado de pacotes. Tem implementado também o controle de congestionamento, que evita que pacotes sejam perdidos quando a rede está sobrecarregada. Isso é feito diminuindo a velocidade com que os dados são transmitidos. Já no protocolo UDP não há implementado nenhum destes controles, que reforça ainda mais a idéia de ser um protocolo não confiável (Kurose 2003).

A maioria das aplicações utilizam então o protocolo TCP de envio, pois necessitam de que os pacotes sejam recebidos no destino de forma correta e sem perdas. Isso não significa que o protocolo UDP não tenha nenhuma vantagem, pois ele é bastante utilizado em aplicações de comunicações por voz, e vídeo, que necessitam de maior rapidez na transferência dos dados (Kurose 2003).

2.7 Common Object Request Broker Architecture (CORBA)

Common Object Request Broker Architecture (CORBA) é a arquitetura padrão criada pelo *Object Management Group* para estabelecer a troca de dados entre sistemas distribuídos heterogêneos. Com o uso de diversos tipos de software e hardware tornam-se necessários aplicativos e modelos que provêm esta comunicação em diferentes interfaces tanto de hardware e software. CORBA é uma das alternativas a ser utilizada pelo fato de atuar de modo que os objetos possam se comunicar de forma transparente ao usuário, mesmo que para isso seja necessário interoperar com outro software, em outro sistema operacional e em outra ferramenta de desenvolvimento (Pritchard 2000). CORBA é um dos modelos mais populares de objetos distribuídos.

A arquitetura CORBA define o *Object Request Broker* (ORB) como um módulo intermediário entre cliente e objeto, que fica encarregado de aceitar a

requisição do cliente, enviá-la para o objeto servidor que assim que disponível retorna a resposta da requisição ao cliente (Pritchard 2000).

CORBA é independente da linguagem de programação pelo fato de ter uma *Interface Definition Language* (IDL) desenvolvida em C++, que provê uma comunicação com outras linguagens como Java, COBOL, Python, entre outras. Esse é um dos grandes motivos para se utilizar este modelo em diversas aplicações.

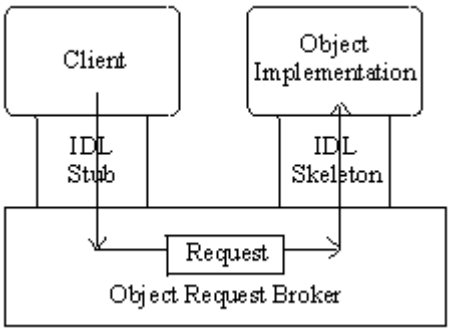

Figura 4 Requisição sendo passada do cliente para o objeto de implementação. Fonte: http://www.omg.org

A Figura 4 mostra o modelo de comunicação do CORBA. O cliente faz sua requisição armazenando os dados no stub. O stub carrega as informações de requisição do cliente, inclusive a interface de comunicação (IDL), que informa ao cliente quais são os métodos que podem ser invocados remotamente. As informações são repassadas ao destino através do objeto de requisição (Object Request Broker). Quando o objeto chega ao destino, as requisições são entendidas e processadas no Object Implementation, que é onde a implementação dos métodos se encontram. O skeleton tem a função similar a do stub, que é onde se define qual a interface de comunicação (OMG 2010).

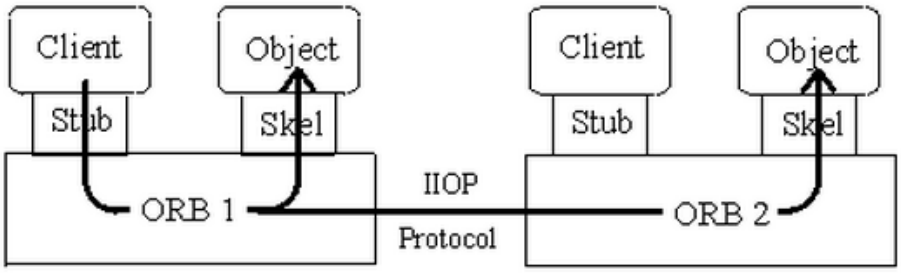

Figura 5 Modelo de comunicação do CORBA através do Módulo ORB intermediário. Fonte: http://www.omg.org

A Figura 5 representa o modelo de interoperabilidade do CORBA via *Object Request Broker* (ORB). Nela tem-se o modelo de comunicação do CORBA, que define o ORB como um módulo intermediário entre o cliente e o objeto. É ele que fica responsável por aceitar a requisição do cliente, enviá-lo para o objeto competente e, assim que disponível o resultado da execução, retorná-la pra o cliente (OMG 2010). O *Internet Inter-Orb Protocol* (IIOP) é um protocolo abstrato pelo qual os módulos ORB se comunicam. Este protocolo nada mais é que uma implementação para o protocolo de rede TCP/IP.

2.8 Simple Object Access Protocol (SOAP)

Simple Object Access Protocol (SOAP) é também um protocolo de troca de dados de sistemas distribuídos. Sua diferença é que utiliza tecnologias baseadas em XML. Ele provê formas de se construir mensagens que trafeguem na rede através de vários protocolos, e é também um protocolo independente de modelo de linguagem de programação assim como os modelos RMI e CORBA.

Ele é um protocolo que geralmente é utilizado em servidores HTTP, embora este não seja uma restrição do modelo (Box 2000).

Administrado pelo órgão W3C, o SOAP é um protocolo que transfere mensagens na forma de documentos XML, que seguem as especificações definidas pelo órgão.

SOAP fornece um mecanismo simples e leve para a troca de informação estruturada e digitada entre pares em um ambiente descentralizado distribuído usando XML. O SOAP é composto por três partes: o envelope SOAP que define um quadro global para expressar o que está em uma mensagem, quem deve lidar com isso, e se é opcional ou obrigatório; as regras de codificação SOAP que definem um mecanismo de serialização que podem ser usados para casos de troca de tipos de dados definidos pelo aplicativo; e a representação SOAP RPC que define uma convenção que pode ser usada para representar as chamadas de procedimento remoto e respostas (Box 2000).

O SOAP é bem seguro já que tem mecanismos de tratamento de erros e, além disso, ainda apresenta como concepção a simplicidade, e a independência de linguagem de modelos de objetos.

2.9 Remote Method Invocation (RMI)

O RMI trata-se de uma API Java de suporte a comunicação remota de objetos. Os métodos em objetos remotos podem ser chamados por objetos locais, se estes estiverem presentes localmente, ou podem ser chamados passando- se argumentos para servidores remotos que retornam valores ao cliente que fez a requisição (Gehlsen 2001). Neste último caso, o modelo cliente/servidor é facilmente percebido.

Segundo Jandl (2003) o RMI foi criado com o intuito de permitir que um objeto escrito na linguagem Java invoque outros métodos que estão executando em outra *Java Virtual Machine* (JVM) como se eles estivessem na mesma JVM. A forma com que isso é feito, faz com que a chamada a um método remoto ou local tenha uma única sintaxe, e seja feita de forma transparente.

As aplicações que utilizam RMI são constituídas de um programa servidor, que instancia objetos e os registra como remotos no serviço de nomes, e clientes, que pesquisam no serviço de nomes do RMI por objetos remotos e obtêm referências a estes objetos através de suas interfaces (Sun Microsystems 2010). Diferentemente como foi descrito o modelo CORBA, que utiliza uma linguagem de definição de interfaces (IDL), no Java RMI não há uma linguagem de definição de interfaces, já que as interfaces são escritas na própria linguagem Java, usando-se a mesma sintaxe utilizada na definição de interfaces locais.

Quanto às interfaces dos objetos são criados os *stubs* do lado do cliente, que são classes que atuarão como representantes do objeto servidor (*proxies*). São a partir deles que será possível referenciar os objetos remotos. Quanto à integridade dos objetos, o Java RMI utiliza a serialização de objetos, que é a capacidade de armazenar e recuperar objetos em Java. Essa serialização é feita pelo RMI tanto para se transferir parâmetros que o cliente passa ao servidor quando invoca um método remoto, quanto para o servidor retornar os resultados ao cliente (Sun Microsystems 2010).

O kit J2SDK de desenvolvimento Java traz todas as classes necessárias para a invocação de métodos remotos, e também um conjunto de ferramentas para executar o serviço de nomes e a geração dos objetos *proxies*.

Para que se possa localizar os objetos remotos é necessário um servidor de nomes, chamado *Registry*, que é um serviço que registra objetos que poderão ser invocados remotamente por um cliente. A função deste servidor de nomes é

então associar cada objeto a um nome e a um endereço. A partir deste nome e endereço, um programa cliente pode obter referências a objetos e invocar remotamente seus métodos. Cada máquina deve ter um serviço de nomes ativo. Esse serviço é executado por um programa chama *rmiregistry*, que vem no kit Java citado anteriormente. A figura 6 ilustra o funcionamento do Java RMI de forma resumida.

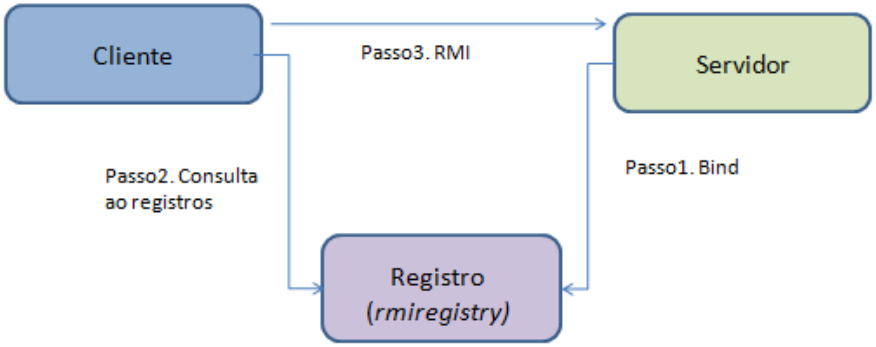

Figura 6 Modelo básico de funcionamento do RMI

O funcionamento básico do RMI é constituído então em três passos:

1. Passo1. O servidor cria objetos remotos e os registra no *rmiregistry* (Bind).
2. Passo2. O cliente faz a busca no serviço de nomes do Java RMI a procura de objetos e obtêm do servidor um *stub.*
3. Passo3. O cliente acessa a interface do objeto remoto através do *stub* e invoca os métodos deste objeto (RMI).

Além de muito seguras e robustas, as aplicações distribuídas que utilizam o Java RMI oferece um desempenho muito bom se comparadas a aplicações que utilizam outras tecnologias (Calabrez 2004).

2.9.1 API do Java RMI

A API (*Application Programming Interface*) do Java RMI, contém o conjunto de classes e interfaces necessárias para a invocação remota a objetos.

No pacote *java.rmi*, tem-se a interface *Remote*, da qual devem derivar todas as outras interfaces. Essa interface serve apenas para indicar que uma determinada interface é remota (Jandl 2003).

A classe *RemoteObject*, presente no pacote *java.rmi.server*, é a superclasse dos objetos remotos, do qual derivam as principais classes do Java RMI (Jandl 2003).

A classe *RemoteServer*, presente no pacote *java.rmi.server*, é uma classe abstrata que serve de base para as classes dos objetos que serão servidos remotamente. A classe *Naming* também do pacote *java.rmi* é a classe que serve de acesso para consultas de métodos, inclusão e também remoção de objetos no registro de nomes (Jandl 2003).

2.9.2 Criando uma Aplicação que utiliza o Java RMI

Esta seção explica como se faz uma pequena aplicação que utiliza o Java RMI. De acordo com Carvalho (2009), uma aplicação Java RMI é formada por:

- Uma interface remota;
- Um ou mais objetos remotos que implementam esta interface;
- O *rmiregistry*, ou registro;
- Um ou mais clientes;

- Os stubs.

De acordo com a documentação do Java RMI da Sun Microsystem (2010), os passos para a criação de uma aplicação com Java RMI são:

- Definir uma interface remota;
- Implementar o servidor;
- Registrar o objeto remoto no servidor de nomes;
- Criar um objeto cliente.

Com o intuito de detalhar os passos acima, um exemplo de aplicação retirado do guia de implementação da Sun Microsystem (2010) será apresentado. A aplicação é um simples código para fazer uma chamada remota a um método presente num servidor local (*host*). O cliente faz a invocação do método e recebe uma mensagem "Hello, World!" do servidor.

2.9.2.1 Definição da interface remota

Um objeto remoto é uma instância de uma classe que implementa a *remote interface.* A interface remota estende a interface *java.rmi.remote* e declara um conjunto de métodos remotos (Sun Microsystems 2010). A figura 7 apresenta a definição de uma interface remota usada no exemplo, contendo apenas um método (*sayHello*) que irá apenas retornar a *string* quando chamada:

```
package example.hello;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Hello extends Remote {
    String sayHello() throws RemoteException;
}
```

Figura 7 Interface Remota. Fonte: Sun Microsystems (2010).

2.9.2.2 Implementando o servidor

A classe “Server” é a classe que contém o método *main* e que cria uma instância do objeto remoto da aplicação, exporta o objeto remoto, e faz o *bind* da instância para um nome no Java RMI *registry*. O código ilustrado na Figura 8 traz a implementação do servidor, e é ele que contém o método que irá retornar a *string* “Hello, world!”:

```
package example.hello;

import java.rmi.registry.Registry;
import java.rmi.registry.LocateRegistry;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class Server implements Hello {

    public Server() {}

    public String sayHello() {
        return "Hello, world!";
    }

    public static void main(String args[]) {

        try {
            Server obj = new Server();
            Hello stub = (Hello) UnicastRemoteObject.exportObject(obj, 0);

            // Bind the remote object's stub in the registry
            Registry registry = LocateRegistry.getRegistry();
            registry.bind("Hello", stub);

            System.err.println("Server ready");
        } catch (Exception e) {
            System.err.println("Server exception: " + e.toString());
            e.printStackTrace();
        }
    }
}
```

Figura 8 Implementação do Servidor. Fonte: Sun Microsystems (2010).

2.9.2.3 Implementando o Cliente

O programa cliente obtém um *stub* para um registro no servidor *host*, procura pelo objeto remoto pelo nome no registro, e invoca o método *sayHello* no objeto remoto usando o *stub*. A figura 9 contém o código cliente:

```
package example.hello;

import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class Client {

    private Client() {}

    public static void main(String[] args) {

        String host = (args.length < 1) ? null : args[0];
        try {
            Registry registry = LocateRegistry.getRegistry(host);
            Hello stub = (Hello) registry.lookup("Hello");
            String response = stub.sayHello();
            System.out.println("response: " + response);
        } catch (Exception e) {
            System.err.println("Client exception: " + e.toString());
            e.printStackTrace();
        }
    }
}
```

Figura 9 Implementação do Cliente. Fonte:Sun Microsystems (2010).

Quando o cliente invoca o método *sayHello*, o que vai acontecer será:

- o cliente abre uma conexão com o servidor usando o host e a informação da porta de onde se encontra o objeto remoto e codifica os dados de chamada.
- o servidor aceita o pedido de requisição de dados, despacha a chamada para o objeto remoto, e codifica o resultado (a string de resposta “Hello, world!”) para o cliente.
- o cliente recebe, descodifica a mensagem e retorna o resultado para o método que fez a requisição.

2.10 Comparação das tecnologias de comunicação remota (RMI x SOAP x CORBA)

Calabrez (2004) considerando o grande crescimento de opções para se construir sistemas distribuídos, fez um estudo comparando diversos métodos que provê comunicação remota aos sistemas. Segundo ele, os principais fatores que contribuem para esse crescimento são a evolução da capacidade de processamento dos computadores assim como as crescentes taxas de transmissão das redes de computadores.

Para o compartilhamento de recursos existem basicamente dois modelos, o orientado a objetos, que é o caso do RMI e o CORBA, e o orientado a serviços, que é o caso do SOAP, todos explicados neste trabalho.

No modelo orientado a objetos, cada recurso é representado por um objeto e a requisição de um serviço é feita pela chamada a um método do objeto em questão. Já no orientado a serviços, um objeto representa os serviços providos por um servidor, ao invés de recursos específicos (CALABREZ 2004).

Em seu trabalho, Calabrez (2004) justifica o uso de tecnologias de programação distribuída baseadas na linguagem Java, afirmando que esta é a linguagem que mais tem crescido em utilização e importância dos últimos anos. Pelo fato do Gerenciador de Repositórios Distribuídos do DCB estar implementado na linguagem Java, permitiu-se que esta comparação de desempenho entre os modelos fosse inserida neste trabalho.

A comparação feita por Calabrez (2004) foi entre as tecnologias RMI, CORBA e SOAP basicamente. Embora de acordo com ele estas tecnologias sejam muito diferentes entre si em questão de arquitetura e características, são essas diferenças que são os fatores determinantes de desempenho e interoperabilidade das tecnologias. Seu trabalho traz a comparação de desempenho entre estas tecnologias em três diferentes cenários: Cenário A, que

é o cliente e o servidor executando na mesma máquina; Cenário B, que é o cliente e o servidor executando em máquinas distintas de uma mesma sub-rede; e o Cenário C, que são o cliente e o servidor executando em máquinas distintas em diferentes redes.

Os resultados da comparação de desempenho dos três diferentes cenários serão mostrados nas figuras 10,11e 12. Vale ressaltar que nos gráficos aparecem RMI e RMI-IIOP, que são tecnologias que utilizam RMI, mudando apenas em alguns aspectos de arquitetura. JavaIDL e JacORB são baseados no CORBA, e Jax-RPC que é o SOAP.

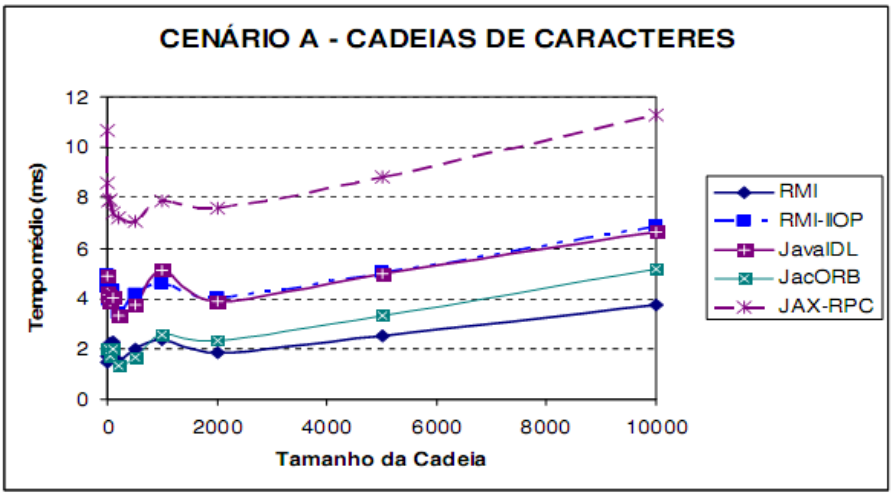

Figura 10 Cenário A: cadeias de até 10000 caracteres. Fonte Calabrez (2004)

No cenário A, apresentado na Figura 10, Calabrez (2004) processou cadeias de até 10000 caracteres em uma mesma máquina utilizando os cinco diferentes métodos. O que obteve melhor resultado de desempenho foi o RMI, seguido do CORBA. O que apresentou pior resultado foi o SOAP, com um tempo em média quatro vezes maior que o RMI.

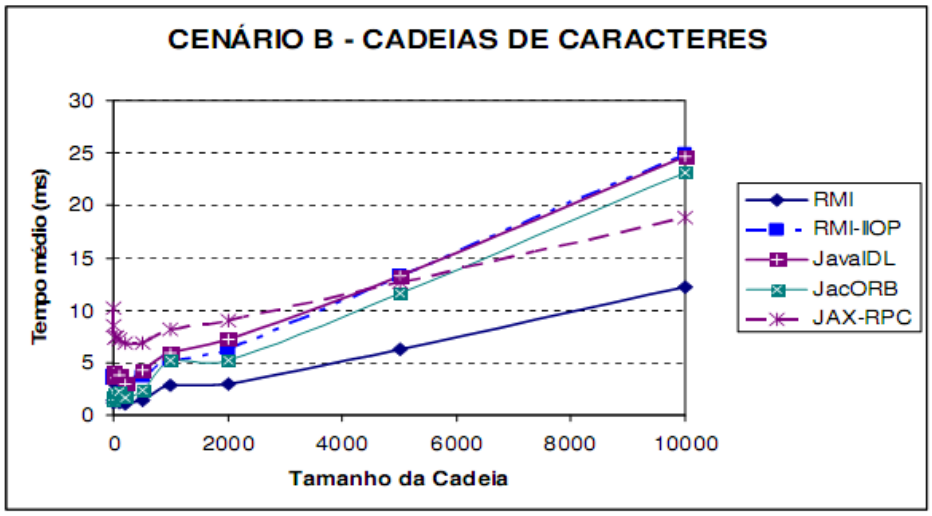

Figura 11 Cenário B. Cadeias de até 10000 caracteres. Fonte: Calabrez (2004)

A Figura 11 ilustra o Cenário B. Neste cenário foram processadas também cadeias de até 10000 caracteres, mas diferentemente do Cenário A, com o cliente e o servidor rodando em máquinas distintas, em uma mesma sub-rede. O RMI obteve o melhor desempenho, com o tempo em média duas vezes menor do que as outras tecnologias.

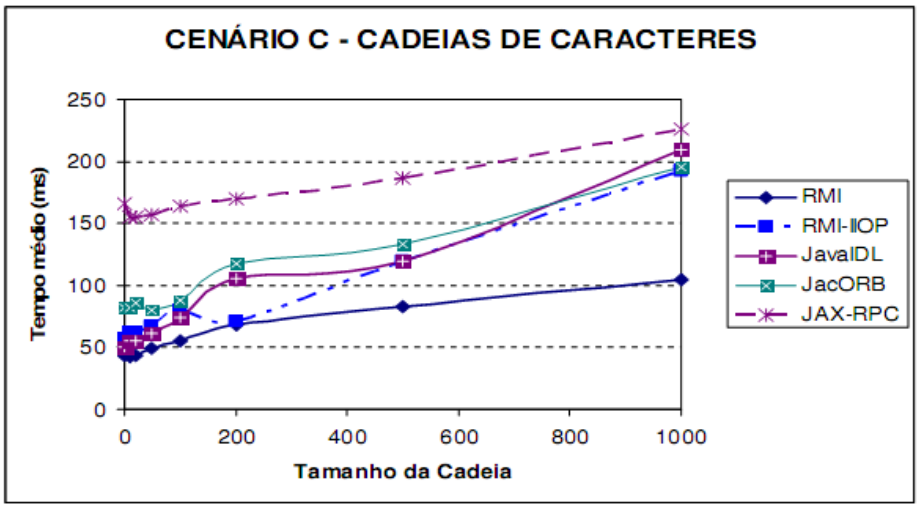

Figura 12 Cenário C. Cadeias de até 10000 caracteres. Fonte: Calabrez (2004)

A Figura 12 ilustra o terceiro e último cenário apresentado neste trabalho que é o cliente e o servidor executando em máquinas distintas e em diferentes redes. O SOAP apresentou o pior resultado, seguido do CORBA que embora melhor que o SOAP em desempenho em cadeias de até 10000 caracteres, apresenta um aumento linear maior que o SOAP à medida que o tamanho da cadeia cresce. Novamente o RMI foi o que obteve melhores resultados.

No seu trabalho, Calabrez (2004) apresentou diferentes experimentos, não apenas com cadeias de caracteres, mas também com lista de objetos, e variando entre esses experimentos quanto ao tamanho da cadeia e também o número de objetos. Como conteúdo deste trabalho, foi ilustrado apenas com cadeias até 10000 caracteres porque todos os resultados foram bastante similares e no Gerenciador de Repositórios Distribuídos os elementos que são comunicados entre cliente e servidor são cadeias de caracteres. Calabrez (2004) como uma de suas conclusões afirma que nos três cenários, as tecnologias que

utilizam RMI apresentaram resultados muito bons se comparados às outras, e que quando se trata de utilizar a linguagem Java, o RMI para prover comunicação remota de sistemas, é a melhor opção dentre estas apresentadas.

Existem outras maneiras de se fazer a comunicação, como por exemplo, utilizando *sockets*, que embora não seja pior, não foi apresentado na análise comparativa de desempenho de Calabrez (2004) junto ao RMI, CORBA e SOAP.

Jagannadham (2007) fez uma análise de desempenho semelhante ao apresentado até agora, mas incluindo o uso de *sockets* e outra tecnologia que nesse trabalho não foi abordada, o *Servlet*.

A Figura 13 ilustra um cenário de comparação de Jagannadham (2007). Nesse cenário ele trafegou diferentes tipos de dados, como inteiros, *doubles* e *strings* numa rede local. Ele de forma semelhante à Calabrez (2004) apresentou outros diferentes cenários, como para invocação local e invocação em diferentes sub-redes. Como os resultados foram bastante semelhantes nos diferentes cenários, e também aos do trabalho de Calabrez (2004), não são apresentados todos os gráficos de resultados.

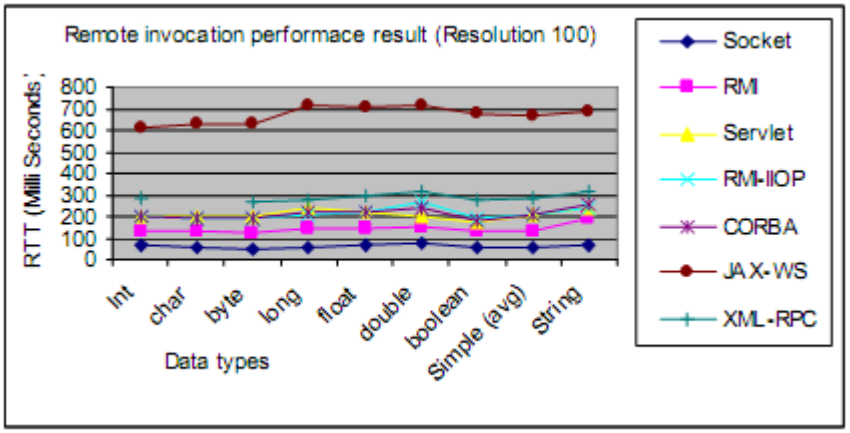

Figura 13 Cenário: Invocação remota. Fonte: Jagannadham (2007)

O modelo JAX-WS (SOAP), foi o que apresentou piores resultados de desempenho nos dois estudos apresentados. Jagannadham (2007) justifica esse baixo desempenho afirmando que o carregamento de diferentes classes, e a manipulação de arquivos XML, que é o tipo de dado trafegado por essa tecnologia, são os que mais causam atrasos de comunicação.

Jagannadham (2007) concluiu seu trabalho afirmando que o uso de *sockets* foi o que teve melhor desempenho dentre todos os outros, mas que em termos de segurança, representação de dados e interoperabilidade ele se mostra muito fraco. A partir disso ele afirma que sockets não é uma boa escolha para aplicações distribuídas de média e larga escala.

Em seu trabalho, Jagannadham (2007) concluiu ainda que o uso de RMI foi o que obteve o segundo melhor resultado de desempenho, e é uma boa escolha para aplicações distribuídas. Essa conclusão é semelhante à de Calabrez (2004), embora ele não tenha incluído o uso de *Sockets* em sua análise. Esses estudos justificam o interesse maior por se utilizar o RMI neste trabalho.

Na literatura existem trabalhos que utilizam RMI para resolver problemas em simulação distribuída.

Gehlsen (2001) propôs em seu trabalho o uso de um *framework* para otimização de simulação distribuída, o *Distributed Simulation Optimization* (DISMO). Por considerar um bom aplicativo não comercial de otimização em modelos distribuídos, ele o utilizou para a otimização de um *framework* para simulação, o *Discrete Event Simulation and Modeling* (DESMO), que está implementado para várias linguagens, como: Smalltalk, C++ e Java. Esta última é representada pela versão DESMO-J proposto por Lechler and Page (1999), é baseado em uma arquitetura totalmente orientada a objetos, e toda implementada na linguagem Java.

No *framework* DISMO para otimização, é utilizado o RMI para o acesso remoto aos objetos. Gehlsen (2001) justifica o uso do DISMO (em particular o RMI) em seu modelo de simulação, por considerar que experimentos de simulação são muito complexos, e que requerem recursos consideráveis até mesmo em sistemas informáticos modernos. Ele ainda afirma que é essencial que se realize várias simulações para um projeto, e que isso se torna possível quando se delega as atividades de simulação a múltiplas máquinas remotas.

Desta forma, como este trabalho se trata de prover contribuições para uma arquitetura de suporte à simulação (o Gerenciador de Repositórios Distribuídos) que foi desenvolvido em sua totalidade na linguagem Java, foi utilizado o RMI para chamada a métodos remotos. Este modelo não foi escolhido apenas pelos bons desempenhos apresentados, mas também pela facilidade que a API já desenvolvida fornece ao desenvolvedor, pela maior segurança dos dados que trafegam na rede, e por questões de operabilidade do sistema.

3 METODOLOGIA

Este trabalho, quanto à pesquisa pode ser classificado como: de natureza aplicada ou tecnológica, devido ao fato de utilizar-se de conhecimentos adquiridos na literatura para uma aplicação prática, e implementação de um modelo de comunicação para o repositório distribuído de elementos do DCB. Quanto aos objetivos, esta pesquisa tem caráter exploratório, pelo fato da necessidade de se familiarizar com o problema e aprimorar idéias para uma contribuição, resolução de um problema e melhoria na arquitetura de simulação já existente. Quanto aos procedimentos é uma pesquisa experimental, pois busca a descoberta de novos métodos através de ensaios e estudos em laboratório, visando o controle de algumas variáveis que podem intervir no experimento. Além disso, é uma pesquisa com embasamento bibliográfico quanto ao meio de aquisição de referências**,** como artigos, teses, e outras fontes de caráter científico e transversal quanto ao tempo de aplicação do estudo, onde entende-se como transversal a pesquisa que tem tempo de realização definido.

O estudo do problema foi o primeiro passo para a realização da pesquisa, seguido da aquisição de referenciais bibliográficos para a familiarização, embasamento científico do trabalho, e identificação de problemas que são relacionados ao escopo deste trabalho. Métodos de comunicação de sistemas computacionais foram estudados, e a decisão foi de aprofundar em especial na interface de programação *Remote Method Invocation* (RMI).

O modelo de comunicação escolhido foi o Cliente-Servidor, isso por alguns fatores como: necessidade de disponibilidade de apenas uma máquina (no caso o servidor); vantagem de se ter um controle dos elementos que se concentram no servidor; a API Java chamada RMI que provê esse tipo de comunicação de forma interessante e sucinta; entre outras. A Figura 14 ilustra

como ficou definida a estrutura de comunicação entre os repositórios de elementos.

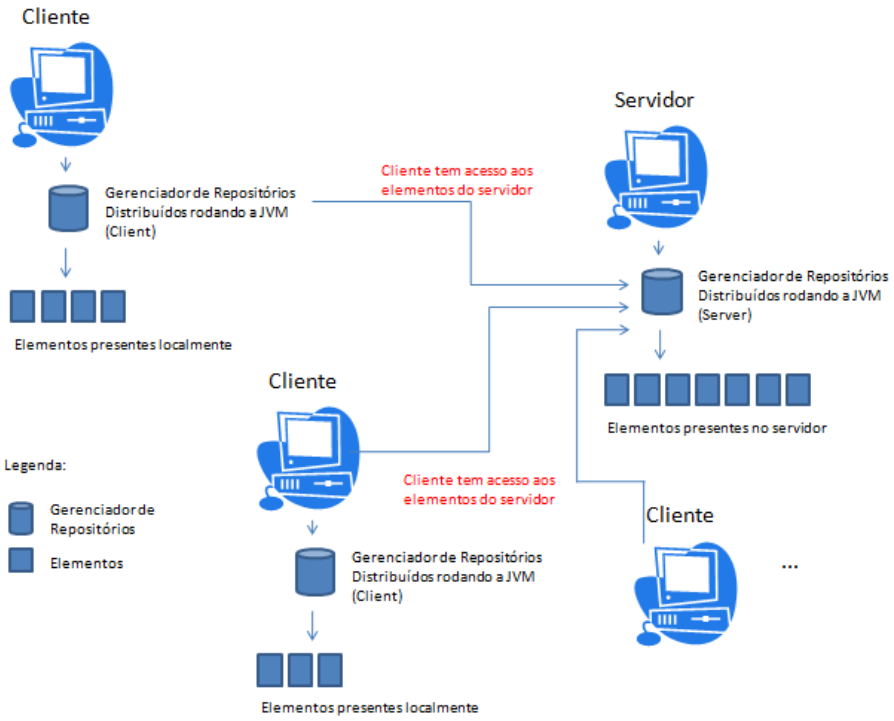

Figura 14 Estrutura comunicação (Modelo Cliente-Servidor)

Com essa estrutura, o servidor tem acesso a todos os elementos que estão armazenados para ele localmente, e os clientes têm acesso tanto a seus elementos de acesso local, quanto aos elementos cadastrados no servidor. Isso se dá de forma com que, caso o cliente faça uma solicitação (busca) por algum elemento que não esteja acessível localmente, o gerenciador de repositórios de forma transparente ao usuário, faça uma requisição ao servidor para verificar se ele contém ou não o elemento que está sendo buscado. Caso o servidor contenha

este elemento, ele será retornado ao usuário que poderá utilizá-lo no seu modelo de simulação.

Para possibilitar que o servidor tenha acesso a todos os elementos e disponibilize essas informações para outros clientes, toda vez que um usuário estiver executando o sistema como cliente e fizer a inserção de um novo elemento no sistema, suas informações serão enviadas ao servidor que guardará uma cópia idêntica em seu banco de dados. Toda essa operação ocorre de forma implícita e transparente ao usuário.

A solução proposta neste trabalho não trata do problema de gargalho no servidor e tolerância a falhas.

A Comunicação foi feita via invocação de métodos remotos, que é a forma que a API Java RMI provê a troca de informações entre diferentes máquinas. O cliente executa o gerenciador de repositórios com uma *Java Virtual Machine* (JVM) configurada para buscar acesso a outras JVMs que estão executando como servidor. Então quando o usuário faz uma busca a algum elemento, o sistema verifica primeiramente se existe aquele elemento localmente, senão o método de busca do servidor será invocado pelo cliente. A Figura 14 ilustra a forma de comunicação utilizando o RMI.

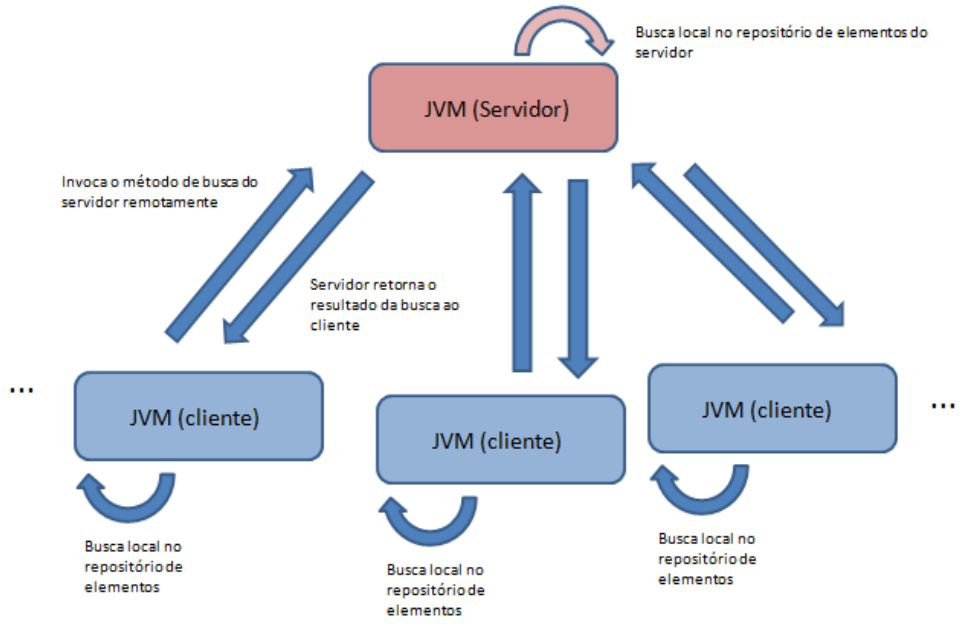

Figura 15 Chamada remota a métodos, provendo a comunicação do cliente com o servidor

A Figura 15 mostra a busca tanto local de elementos quanto à busca remota. A busca local continua sendo da mesma maneira, sendo chamados métodos locais. O que muda é que o cliente tem agora a possibilidade de acessar os dados através da JVM do servidor.

Tanto a JVM do servidor quanto a do cliente ao executar, carregam as informações dos métodos que podem ser invocados remotamente através de uma interface que foi definida no sistema. Além dessa interface, o JVM servidor ao ser iniciado gera uma tabela de nomes contendo as informações desses métodos, que podem ser acessadas por outras máquinas virtuais através de uma porta de comunicação definida. A porta 1099 foi a que ficou estabelecida para a comunicação do gerenciador. Esta porta é bastante utilizada como padrão para

comunicação RMI, mas pode ser alterada a qualquer momento caso seja necessário.

Como função de servidor, tanto o sistema quanto a tabela de informações ficam executando no sistema até que uma requisição seja feita.

Quando um cliente faz a chamada remota, o sistema verifica através de um IP (no caso o do servidor) e uma porta de comunicação, se o servidor está ou não disponível. Se por um acaso o servidor não estiver disponível, é exibida uma mensagem de falha na conexão, e o cliente fica então apenas limitado a acesso ao servidor e continua utilizando as funcionalidades locais normalmente. Se por outro lado a conexão for estabelecida, o cliente então abrirá uma conexão com o servidor, e buscará em sua tabela de nomes se existe algum método com as características passadas por ele e que possa ser invocado remotamente.

O cliente ciente então sobre quais métodos estão disponíveis no servidor, passa a poder fazer chamadas a qualquer um deles passando os parâmetros necessários. O servidor recebe os parâmetros, executa o método localmente e retorna o resultado ao cliente.

Os métodos principais que foram implementados foram: o *SearchElement (String palavra)*; *InsertElement (String[] infoElementos)*; e o *InternalComportament (String palavra).* O código fonte de cada um destes métodos é mostrado no Apêndice A.

O método *SearchElement* recebe uma variável *String* palavra, contendo o nome do elemento a ser procurado, faz a busca no servidor e retorna ao cliente uma lista com as informações do elemento encontrado. Se o elemento não for encontrado o método retorna a estrutura vazia, que é interpretado como elemento não encontrado.

O método *InsertElement* recebe uma lista de informações do elemento a ser inserido no banco de dados (nome do elemento, nível de restrição, tipo, descrição do elemento, comportamento do elemento, identificação dos atributos,

descrição da interface) e utilizando métodos locais de armazenamento no servidor armazena o elemento no banco de dados do servidor.

O método *InternalComportament* recebe uma variável *String* palavra contendo a palavra-chave que será buscada nos campos “Comportamento interno do elemento” no elementos presentes no servidor. A busca não é feita apenas pelo nome do elemento, as palavras são buscadas também nas informações de comportamento dos elementos. Isto permite que usuários que não saibam o nome do elemento que foi cadastrado consigam ter como retorno os elementos pelo nome do elemento e pela descrição do comportamento.

Com o intuito apenas de medição de desempenho de comunicação cliente/servidor outros dois métodos foram desenvolvidos: *ThreadSearch* (*String palavra*) e *ThreadInsert* (*String infoElementos[]*). O primeiro método é para a busca de elementos no servidor, onde o dado trafegado consiste da palavra a ser buscada no servidor. O segundo método é pra a inserção do elemento no servidor, onde a troca de dados que é feita é um vetor com informações do elemento a ser inserido.

Todo esse modelo de comunicação entre os repositórios foi implementado visando não exigir do projetista do modelo de simulação conhecimentos sobre a rede, a localização e a configuração das máquinas. Isso torna a comunicação entre os gerenciadores de repositórios tarefa somente do gerenciador, reduzindo as falhas de manipulação de elementos e reaproveitamento de informações de elementos que já estejam cadastrados no sistema.

4 RESULTADOS

Após o estudo detalhado de modelos e a implantação do modelo RMI de comunicação remota no Gerenciador de Repositórios Distribuídos, tem-se a comunicação deste sendo feita de forma transparente para o usuário. O Gerenciador passa a ter duas versões, a versão cliente e a servidor. O servidor ao ser executado fica esperando por requisições vindas de clientes remotamente, e os clientes ao serem executados iniciam a conexão com o servidor. A figura 16 ilustra o servidor executando.

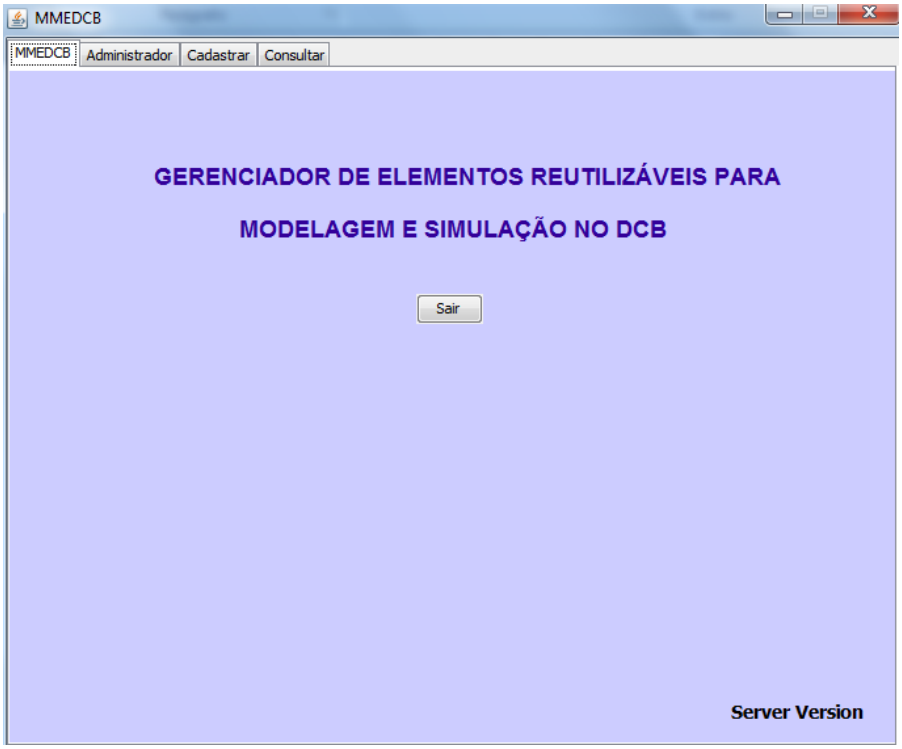

Figura 16 Execução da versão servidor do Gerenciador.

Toda vez que um cliente é iniciado, ele tentará uma conexão com o servidor. Se este estiver disponível, essa conexão será estabelecida e o usuário passa a ter então acesso aos dados tanto localmente quanto remotamente. A Figura 17 ilustra a versão cliente sendo iniciada e a mensagem de "conexão estabelecida" sendo exibida.

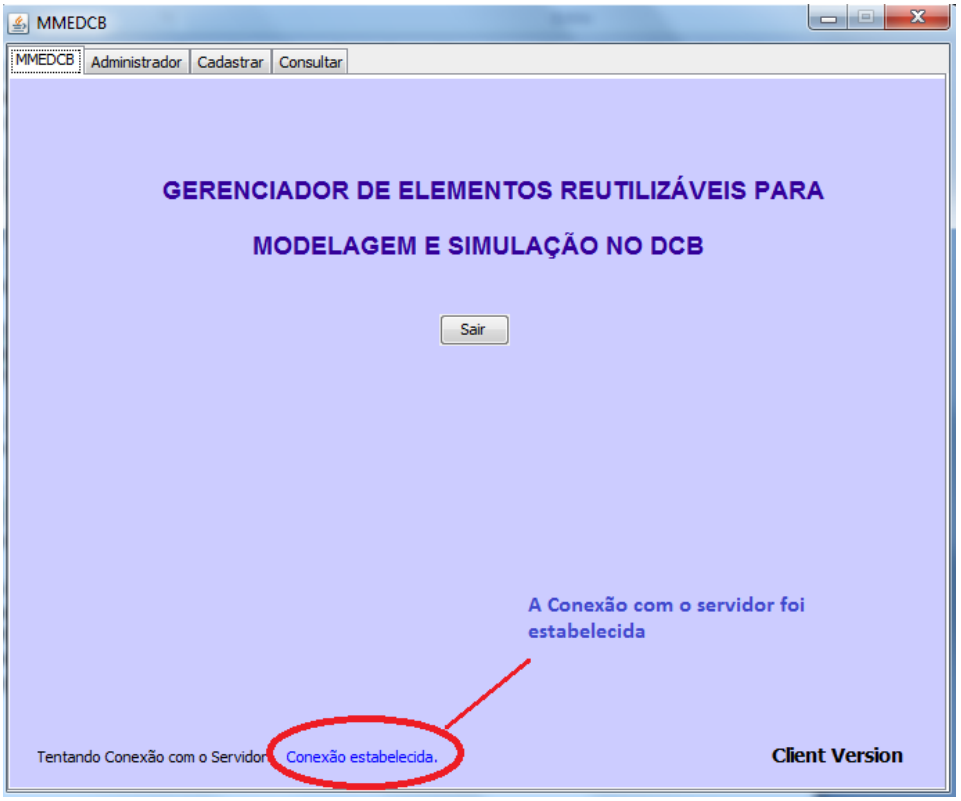

Figura 17 Versão cliente sendo executada.

Caso o servidor não esteja disponível e a conexão não seja estabelecida, uma mensagem de falha de conexão será exibida. Nesta ocasião o usuário continuará a ter acesso a todas as funções do gerenciador que são locais, ou seja, apenas os serviços de acesso a dados no servidor estarão indisponíveis. O usuário poderá então continuar suas tarefas normalmente mesmo com a indisponibilidade do servidor.

Quando o servidor volta a estar disponível, o usuário automaticamente passa a ter comunicação e as funções realizadas remotamente se retomam normalmente.

Com o Gerenciador funcionando de forma distribuída, o cliente (também projetista do modelo de simulação) pode então buscar elementos que estão disponíveis no servidor. Elementos estes de origens diferentes, já que a cada cadastro de novos elementos por qualquer gerenciador armazenará também uma cópia no servidor para reaproveitamento de informações. A figura 17 mostra uma consulta sendo feita, e elementos tanto locais quanto remotos sendo exibidos.

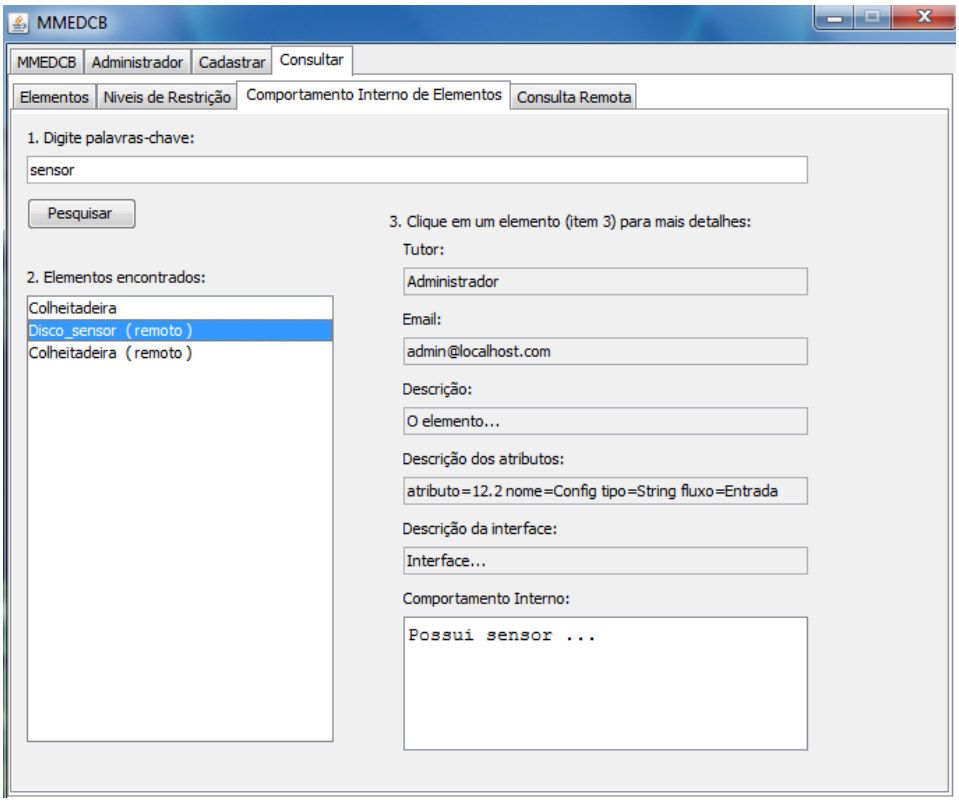

Figura 18 Cliente efetuando uma consulta por uma palavra chave "sensor".

Na Figura 18 uma consulta é feita passando a palavra “sensor” como palavra chave. São exibidos três elementos, um local, e dois que foram encontrados remotamente no servidor. Os elementos remotos quando requisitados pelo cliente, serão retornados e armazenados localmente. Isso significa que o usuário agora poderá utilizar esse elemento em seu projeto de simulação, e que foi inserido por outro tutor no gerenciador.

A Figura 19 mostra a tela de cadastro de novos elementos. Nessa tela o usuário, ao inserir um elemento além de inserido localmente, terá uma cópia armazenada no servidor, pelo motivo da disponibilidade do elemento para outros tutores que estiverem na rede. Outro motivo é a segurança dos dados, já que o

elemento ficará armazenado em dois, ou mais locais diferentes (considerando a possibilidade de outro tutor já ter utilizado o elemento).

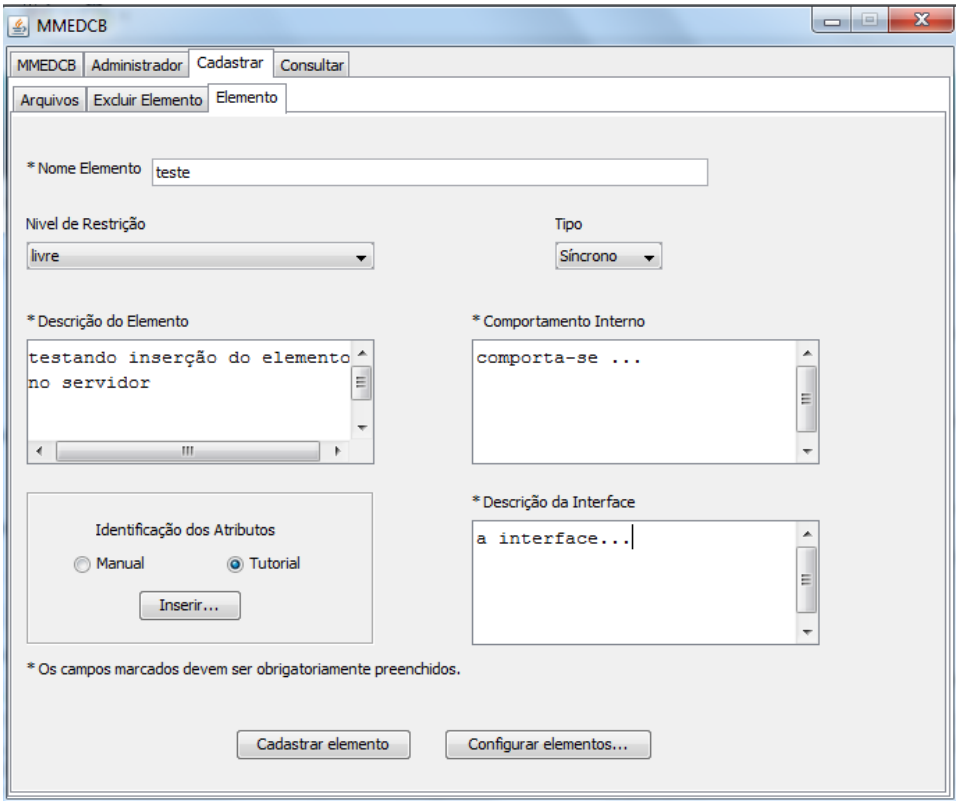

Figura 19 Tela de cadastro de novo elemento.

Casos e exceções foram tratados como, por exemplo, a já existência do elemento localmente quando requisitado. Quando isso ocorrer esse elemento não será armazenado de forma redundante no sistema. Outro caso seria a tentativa de armazenar elementos que já existem no servidor. Se isso ocorrer, uma mensagem informativa será exibida ao projetista que terá conhecimento da já existência desse elemento.

Como resultado final tem-se o Gerenciador de Repositórios funcionando de forma distribuída, e utilizando o RMI para chamadas remotas de sistema.

4.1 Testes

Alguns testes foram realizados para validar o funcionamento do Gerenciador de Repositórios de forma distribuída. Primeiramente, foi realizado teste de portabilidade do sistema e em seguida de desempenho. No teste de portabilidade foram realizadas manipulações de elementos, inserção, remoção, cópia, perda e retomada de comunicação.

Para testar a portabilidade do sistema foram utilizados dois ambientes de teste com diferentes cenários, onde houve variação do número de máquinas e o tipo de sistema operacional. O primeiro ambiente de teste é composto por três máquinas com sistemas operacionais Windows 7, Windows XP e Ubuntu. O segundo ambiente de teste é um laboratório de Ciência de Computação da UFLA, onde foram utilizadas 12 máquinas, com sistema operacional Windows 2000. Em todos os cenários, as máquinas tinham o kit Java SDK 6 instalado. Para executar esta aplicação é necessário apenas que as máquinas sejam capazes de executar a Máquina Virtual Java e realizar comunicação entre si.

A Tabela 1 mostra estes cenários e seus resultados obtidos para validação do novo modelo do sistema. Em todos os cenários a conexão foi estabelecida e não houve falhas.

Tabela 1 Diferentes cenários de testes.

Cenário	N° Máquinas	Servidor	Cliente (s)	Falhas	Comunicação
1	3	Windows XP	Windows 7	Não apresentou	Estabelecida
2	12	Windows 2000	Windows 2000	Não apresentou	Estabelecida
3	2	Windows XP	Ubuntu	Não apresentou	Estabelecida
4	2	Ubuntu	Windows XP	Não apresentou	Estabelecida

No cenário 1, foram executados 2 clientes com o sistema operacional Windows 7, e um servidor utilizando Windows XP.

O cenário 2 consistiu em um laboratório com 12 máquinas executando o sistema operacional Windows 2000. Foram executados 11 clientes e um servidor. Testes também foram realizados entre os sistemas operacionais Windows, e Linux. Os cenários 3 e 4 ilustram essa situação utilizando o Windows XP como servidor e cliente respectivamente, e a distribuição Linux Ubuntu 10.04 como cliente no cenário 3 e servidor no cenário 4.

A Figura 19 ilustra o modelo cliente do gerenciador sendo executado no Ubuntu.

Figura 20 Gerenciador sendo executado no ambiente Linux (Ubuntu).

No caso ilustrado pela Figura 19, o modelo cliente do Gerenciador de Repositórios sendo executado no Ubuntu estabelece a comunicação com o servidor hospedado em uma máquina com o sistema operacional Windows XP.

4.1.1 Testes de desempenho

Com o intuito de testar o desempenho do sistema, foram realizados testes para verificar os tempos necessários para se realizarem as operações básicas no sistema, a busca de elementos e a inserção. O cenário criado para a busca foi intercalar 100 consultas ao servidor através de uma rede local, por palavras contidas ou não no servidor. A figura 21 ilustra o tempo em milissegundos gasto para realizar cada busca.

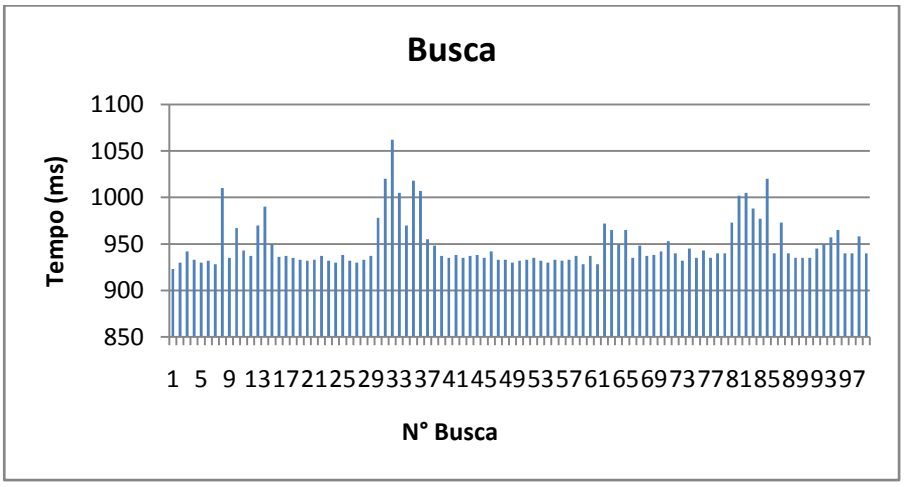

Figura 21 Cenário de busca por elementos no servidor.

A partir dos dados mostrados na Figura 21 foi constatado que o tempo médio para a busca por elementos foi de 949 milisegundos. Esse tempo envolve tanto o tempo de comunicação quanto o tempo de execução da requisição por parte do servidor.

A Figura 22 ilustra o tempo em milissegundos gasto para realizar cada inserção (cadastro) de um novo elemento.

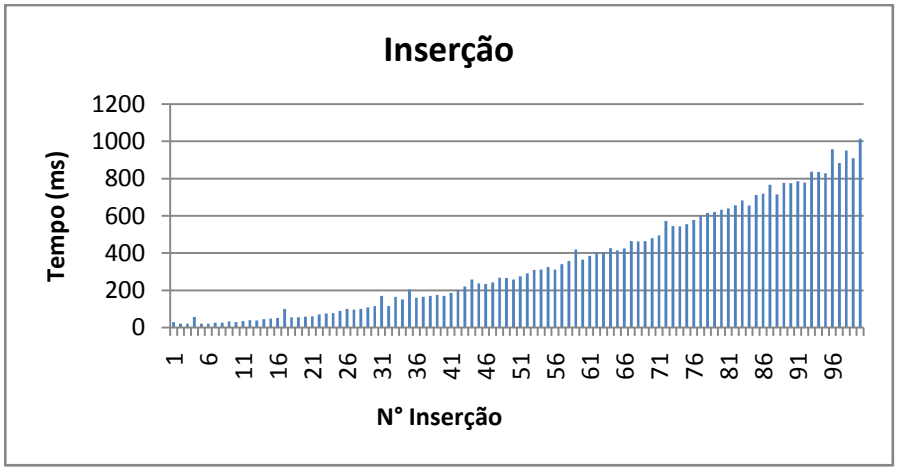

Figura 22 Cenário de inserção de elementos no servidor.

A figura 22 mostra que à medida que o número de inserções aumentou (o banco de dados foi sendo preenchido) o tempo de inserção também aumentou. O tempo médio obtido para as inserções foi de 349 milisegundos. Tendo constatado esse problema do aumento do tempo quanto à quantidade de dados no servidor, foram realizados outros testes em diferentes cenários para verificar se a queda de desempenho estava no processamento dos dados por conta do servidor, ou na comunicação implementada utilizando o RMI.

Como os testes de comunicação visam apenas identificar onde ocorre a queda de desempenho, foi definido que o tempo gasto com as operações sobre os dados no servidor não seria contabilizado. Desta forma, a comunicação foi realizada trafegando as mesmas estruturas de dados pela rede, mas a busca local e a inserção não foram feitas no servidor.

O cenário definido para esses testes foi utilizar Threads para simular a concorrência de diversas máquinas fazendo requisições ao servidor, e com diferentes números de buscas e inserções feitas por cada uma delas, ou seja, o servidor por alguns segundos respondeu diferentes quantidades de requisições.

A Figura 23 ilustra o cenário de testes com um número de inserções em aproximadamente 1400 elementos sendo feitas de forma concorrente. O tempo de cada inserção é expresso em milissegundos.

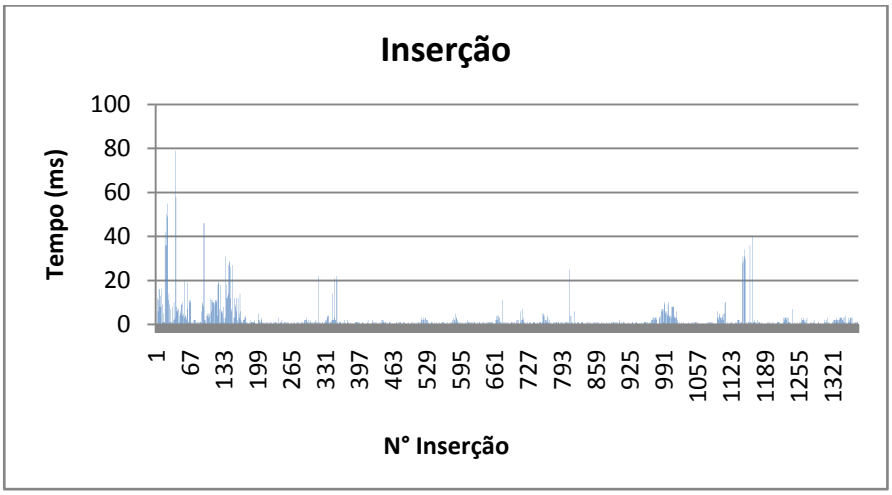

Figura 23 Inserção de elementos no servidor.

A partir dos dados obtidos e exibidos na Figura 23 foi tirada uma média de tempo e o desvio padrão para comunicação e transferência de dados na rede. A média de tempo obtida foi de 2,21 milisegundos, com um desvio padrão de 5,93.

A Figura 24 ilustra o cenário de testes com um número de buscas de aproximadamente 1600 elementos, realizadas de forma concorrente. O tempo de cada busca é expresso em milissegundos.

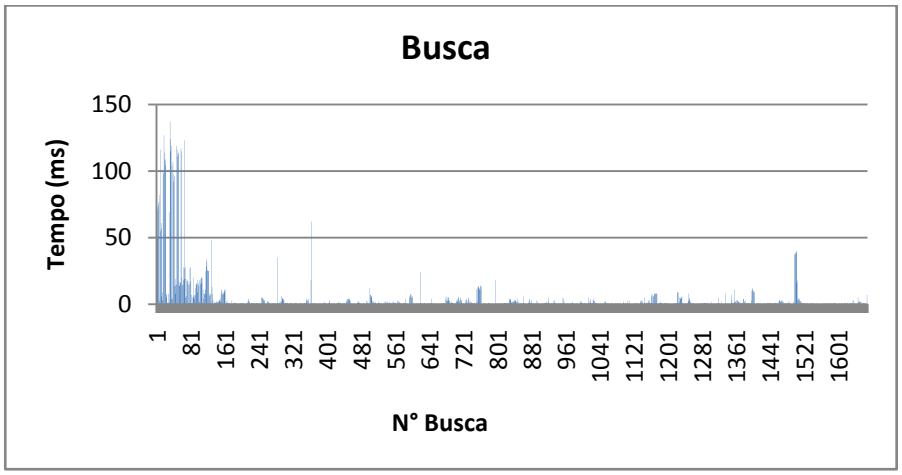

Figura 24 Busca de elementos no servidor.

Analisando os resultados obtidos na Figura 24 foi possível constatar que o tempo médio das buscas ficou em 4,17 milisegundos com um desvio padrão de 15,14.

Com os resultados apresentados nas Figuras 23 e 24 para testes de desempenho na comunicação, foi constatado que o gargalo e a maior parte do tempo do sistema ficavam no processamento feito pelo servidor e não na comunicação. Isso porque o tempo de comunicação e transferência de dados na busca representa menos de 0,5% (4,17 / 949 = 0,00439) do total do tempo, e porque o tempo da inserção ficou em torno de 0,23% (2,21 / 349 = 0, 0023) do tempo total.

É importante ressaltar que as simulações foram randômicas, ou seja, os cenários de busca e inserção são independentes, variando o número de operações por segundo, o que pode causar uma diferença do tempo médio das duas funções, seguido também do tamanho dos dados trafegados e das condições da rede no momento da realização da simulação.

Se comparados os resultados de tempo comunicação e transferência de dados obtidos nesse trabalho com os resultados obtidos por Calabrez (2004) e

Jagannadham (2007) nas Figuras 10 e 13 respectivamente, é possível verificar que os tempos de comunicação foram próximos, embora os cenários não sejam idênticos e sejam independentes. Estes resultados atendem as necessidades da proposta dessas análises, que era verificar se o gargalo e a queda de desempenho do sistema estavam na comunicação ou no processamento dos dados pelo servidor. Com estes resultados pode-se observar que o gargalo encontra-se no servidor ao acessar o banco de dados.

5 CONCLUSÕES

Com o objetivo inicial de prover comunicação remota ao gerenciador de repositórios distribuídos de elementos, de forma que os projetistas de modelos de simulação do DCB pudessem reutilizar elementos, foram apresentadas diferentes tecnologias de distribuição de sistemas. Todas elas são relevantes e apresentam suas vantagens para cada tipo de aplicação.

O uso da tecnologia RMI é uma boa alternativa para tornar sistemas desenvolvidos em Java distribuído. Além de ser bem documentada, ela é uma tecnologia que tem um grande reconhecimento, sendo citada e utilizada em diversos trabalhos acadêmicos.

O gerenciador de repositórios de elementos se tornando distribuído ajuda os projetistas de modelos de simulação a ganhar tempo, e aumenta a confiabilidade do sistema quanto à perda de dados.

Os resultados experimentais mostraram que o tempo de comunicação e transferência de dados representam uma pequena parcela do tempo gasto para realizar inserções e buscas no gerenciador de elementos distribuídos. O maior tempo é gasto em acessos ao banco de dados.

Algumas funcionalidades ainda devem ser acrescidas ao sistema, com intuito de melhorá-lo, como por exemplo, quanto à segurança e tolerância a falhas.

5.1 Trabalhos Futuros

Este trabalho pode gerar novos frutos, novos modelos podem ser estudados e funcionalidades implementadas, como:

- Disponibilizar ao cliente a opção de alteração de dados, considerando os níveis de restrição e de acesso às informações do gerenciador.
- Criação de um sistema de backup com controle de versões para os arquivos, gerando maior confiabilidade ao sistema e permitindo uma recuperação de dados alterados.
- Permitir ao cliente acesso aos dados do servidor através de uma rede não local.
- Aumento da segurança do sistema quanto à perda de dados e tolerância a falhas. Com o espelhamento dos dados em dois servidores, diminuirá as chances de perda de dados, e de indisponibilidade dos serviços caso um dos servidores apresentar falhas.

6 REFERÊNCIAS BIBLIOGRÁFICAS

BORRIELO, G; WANT, R. Embended computation meets the World Wide Web. **Communications of the ACM**, New York, v. 43, p 58-66, 2000.

BRUSCHI, S. M. **ASDA - Um Ambiente de Simulação Distribuída Automático**, Tese de doutorado – Instituto de Ciências Matemáticas e de Computação, Universidade São Carlos, 2002.

CALABREZ, C. E. **Uma Comparação entre Diversas Tecnologias de Comunicação de Objetos Distribuídos em Java.** Dissertação de mestrado – Instituto de Computação, UNICAMP, Campinas,2004.

CARVALHO, F. M. M. **Controle de Checkpoints para execução otimista de modelos heterogêneos no DCB.** Monografia – Departamento de Ciência da Computação, Universidade Federal de Lavras, 2009.

CHUNG C.A.. **Simulation Modeling Handbook: A Practical Approach**. CRC Press, 2004.

FUJIMOTO, R. M. **Parallel and Distributed Simulation Systems**. In: WINTER SIMULATION CONFERENCE, 33., 2001, Arlington, Virginia. Proceedings... [S.l.:s.n.], 2001. p.147-157.

GELHSEN, B.; PAGE B. **A Framework for distributed simulation optimization.** WINTER SIMULATION CONFERENCE, 33., 2001, Arlington, Virginia. Proceedings... [S.l.:s.n.], 2001. p.508-514.

GONÇALVES, J. A. B. .**Desenvolvimento de uma agenda distribuída utilizando Java RMI.** Monografia – Faculdade de Jaguariúna, 2005.

HUBERT, H. **A survey of HW/SW Cosimulation Techniques and Tools.** 1998. 48 f. Thesis Work, Electronic Systems Design Laboratory, Royal Institute of Technology, Kista, Sweden.

JAGANAADHAM, D. ; RAMACHANDRAN, V. **Java2 Distributed Application development (Socket, RMI, Servlet, CORBA) approaches, XML-RPC and Web Services Functional analysis and Performance Comparison.** International Symposium on Communications and Information Technologies, 2007.

JANDL, P. J. . **Mais Java.** São Paulo: Futura 2003, 635 p.

JUNQUEIRA, F. **Modelagem e Simulação distribuída de sistema produtivo.** Revista Controle & Automação, Vol.20, no. 1, 2009.

KELTON, W. D.; SADOWSKI, R. P.; STURROCK, D. T. **Simulation with Arena**. 4. Ed. New York: McGraw-Hill, 2007.

KUROSE, J. F. **Computer networking: a top-down approach featuring the internet.** Third Edition. Editora Perason, 2003.

LECHLER, T. and B. Page. 1999. DESMO-J: **An Object Oriented Discrete Simulation Framework in Java.** In Proceedings of the 11th European Simulation Symposium, ed. G. Horton, D. Möller, and U. Rüde, 46-50. Erlangen: SCS Publishing House.

MELLO, B. A. **Co-Simulação Distribuída de Sistemas Heterogêneos**. Tese de doutorado – Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre, 2005.

NASCIMENTO, J. M. A. **Simulador Computacional para Poços de Petróleo com Método de Elevação Artificial por Bombeio Mecânico** . Dissertação de mestrado – Centro de Tecnologia, Universidade Federal do Rio Grande do Norte, Natal, 2005.

OMG. **CORBA Basics.** Disponível em http://omg.org. Acessado em: 30/09/2010.

PRITCHARD, J. **COM and CORBA Side by Side: Architectures, Strategies, and Implementations.** Addison Wesley, 2000.

SÁ, A. G. C. ; MELLO B. A.. **Geração automática de um modelo para simulação do SEP.** 9° Simpósio de Mecânica Computacional. São João Del Rei, MG, Brasil. 2010.

STRASSBURGER, S.; Schulze, T. ;Klein, U. ; Henriksen, J. O. . **Internet-based simulation using off-the-shelf simulation tools and HLA**. WSC, vol. 2, pp.1669-1676, 1998 Winter Simulation Conference (WSC'98), 1998.

SUN MICROSYSTEMS. **The java tutorials.** Dísponível em: http://java.sun.com/docs/books/tutorial/rmi/index.html. Acessado em: 14 de junho 2010.

SUN MICROSYSTEMS. **Java RMI reference documentation.** Dísponível em: http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp. Acessado em: 14 de junho 2010.

7 APÊNDICE A

```
/* Método Invocado remotamente para busca de novos elementos no Servidor. O método recebe por parâmetro o nome do elemento pelo cliente, é criado então uma nova consulta a banco de dados no servidor, e se encontrado o elemento, será retornada as informações deste, senão, retornará null e uma mensagem de não encontrado será exibido para o usuário. */
  @Override
  public String[] SearchElement( String palavra)
    throws RemoteException {

    banco = new Banco();

    Elemento elementos = banco.procurarElemento(palavra);
    String[] listaInfo = new String[8];

    // Se encontrou o elemento, será armazenado as informações do elemento no vetor
    if (elementos != null) {
      listaInfo[0] = elementos.getNome();
      listaInfo[1] = elementos.getTipo();
      listaInfo[2] = elementos.getDescricao();
      listaInfo[3] = elementos.getComportamentoInterno();
      listaInfo[4] = elementos.getDescricaoAtributos();
      listaInfo[5] = elementos.getDescricaoInterface();
      listaInfo[6] = ((String)((banco.procurarTutor(elementos.getCod_tutor())).getNome()));
```

```
            listaInfo[7]                                                    =
((String)((banco.procurarTutor(elementos.getCod_tutor())).getEmail()));

        }
        else {
            // Caso o elemento não seja encontrado, uma mensagem será exibida
            System.out.println("Elemento não encontrado");
        }

        return listaInfo;
    }
```

/* Método Invocado remotamente para cadastro de novos elementos no Servidor. O método recebe por parâmetro as informações do elemento cadastrado pelo cliente armazenado em um vetor de strings. é criado então um novo elemento, setado as informações e logo é cadastrado em banco. */

```
    @Override
    public void InsertElement(String infoElementos[]){

        // Instancializando novo tutor
        tutor = new Tutor();
        // Instancializando comunicação com o banco
        banco = new Banco();

        // Instancializando novo elemento
        Elemento elemento = new Elemento ();
```

```
        // Seta as Informações do Elemento
        elemento.setNome(infoElementos[0]);
        elemento.setTipo(infoElementos[1]);
        elemento.setDescricao(infoElementos[2]);
        elemento.setComportamentoInterno(infoElementos[3]);
        elemento.setDescricaoInterface(infoElementos[4]);
        elemento.setDescricaoAtributos(infoElementos[5]);

        // Seta os códigos dos elementos e do tutor no elemento
        elemento.setCod_elem(banco.ultimo_codigo_elemento() + 1);
        elemento.setCod_tutor(tutor.getCod_tutor());

        // Nível de restrição do Elemento
        NiveisRestricaoElemento n = banco.procurarNivelDeRestricao("livre");
        elemento.setCod_nivel(n.getCod_nivel());

        // Cadastra o elemento no banco
        banco.inserir_elemento(elemento);
    }
```

/* Método que retorna uma arrayList com o nome dos elementos encontradosa partir de uma busca feita na descrrição de comportamento interno de elementos. As palavras chaves são passadas concatenadas através de uma string, e logo será dividido a busca de palavra a palavra */

```
    @Override
    public ArrayList<String> InternalComportament(String palavra){

        banco = new Banco();
```

```
ArrayList<String> nomes = new ArrayList<String>();

String palavrasChave2 [] = palavra.split(" ");
ArrayList <Elemento> elementos = banco.getElementos();

/*Consulta por nome e por palavras contidas na descrição do comportamento
interno do Elemento localmente.
 */
for (int i = 0; i < palavrasChave2.length; i++) {
   for (int j = 0; j < elementos.size(); j++) {
      if (elementos.get(j).getComportamentoInterno().indexOf(palavrasChave2[i]) != -1){
         if(!(nomes.contains(elementos.get(j).getNome()))){
            nomes.add(elementos.get(j).getNome());
         }
      }
      if (elementos.get(j).getNome().indexOf(palavrasChave2[i]) != -1) {
         if(!(nomes.contains(elementos.get(j).getNome()))){
            nomes.add(elementos.get(j).getNome());
         }
      }
   }
}
return nomes;
}
```

```
        /* Este método apenas faz a busca pelo nome do elemento, e se o
encontrou informa que já tem esse elemento cadastrado no banco do servidor,
senão, retorna um false (não há elementos com esse nome). */
    @Override
    public boolean Find(String nomeElemento){

        boolean achou = false;

        banco = new Banco();

        // Retorna os elementos encontrados com o nome nomeElemento passado
        Elemento elementos = banco.procurarElemento(nomeElemento);

        // Se elementos não estiver vazio é porque encontrou
        if(elementos != null){
            achou = true;
        }

        return achou;
    }

        /* Este método apenas simula os dados trafegados na rede na busca de
elementos para realizar os testes de desempenho do sistema. Este método recebe
um nome através de uma String e retorna um vetor com informações do
elemento encontrado*/
    @Override
    public String[] ThreadSearch(String palavra){
```

```
        // Instancia de um novo vetor de Strings
        String[] listaInfo = new String[8];

        // Informações do elemento.
        listaInfo[0] = "Disco_Sensor";
        listaInfo[1] = "sincrono";
        listaInfo[2] = "...";
        listaInfo[3] = "...";
        listaInfo[4] = "...";
        listaInfo[5] = "...";
        listaInfo[6] = "...";
        listaInfo[7] = "...";

        // Retorna vetor com informações do elemento
        return listaInfo;
    }
```

/* Este método apenas simula os dados trafegados na rede na inserção de elementos para realizar os testes de desempenho do sistema. Este método recebe um vetor de Strings com informações do elemento. Como foi isentado asoperações em cima do servidor nos testes de desempenho o método não contém código. */

```
@Override
    public void ThreadInsert(String infoElementos[]){
        // Nesse método as operações são todas em cima do banco de dados.
    }
}
```