UNIVERSIDADE FEDERAL DE SANTA CATARINA
Departamento de Informática e Estatística

Programa de Pós-Graduação em Ciência de Computação

DISSERTAÇÃO DE MESTRADO

TERCILIO STEDILE JUNIOR

UM MODELO PARA COMPARTILHAMENTO DE BASES DE DADOS DISTRIBUÍDAS E HETEROGÊNEAS

Dissertação submetida à Universidade Federal de Santa Catarina como parte dos requisitos para a obtenção do grau de Mestre em Ciência da Computação

Orientador: Prof. Dr. Luis Fernando Friedrich

Florianópolis - SC, julho de 2005

UM MODELO PARA COMPARTILHAMENTO DE BASES DE DADOS DISTRIBUÍDAS E HETEROGÊNEAS

TERCILIO STEDILE JUNIOR

Esta dissertação foi julgada adequada para a obtenção do título de Mestre em Ciência da Computação, na área de concentração Sistemas de Computação e aprovada em sua forma final pelo Programa de Pós-Graduação em Ciência da Computação.

_____
Prof. Dr. Luis Fernando Friedrich
Orientador

_____
Prof. Dr. Raul Sidnei Vazlawick
Coordenador do Curso

Banca Examinadora

_____
Prof. Dr. Luis Fernando Friedrich
Presidente da Banca

_____
Prof. Dr. Vitório Bruno Mazzola
Membro

_____
Prof. Dr. Frank Augusto Siqueira
Membro

_____
Prof. Dr. Rômulo Silva de Oliveira
Membro

Agradecimentos

Ao professor Dr. **Luiz Fernando Friedrich**, orientador, pelo apoio, ensino e incentivo.

Aos colegas professores **Diogo Wink** e **Dionei Domingos**, por suas contribuições na discussão e avaliação de minhas pesquisas e implementação do modelo.

À **Jefferson Thiago Leitholdt** e **Joederli Kiefer Barreto**, pelo auxílio da codificação de programas e simulações do modelo.

À **UNERJ** – Centro Universitário de Jaraguá do Sul, por proporcionar o apoio necessário e o acesso aos laboratórios para a construção e simulação deste trabalho.

A todos, que de forma direta ou indireta contribuíram para a realização deste trabalho.

Dedicatória

À **Simone**, minha esposa, pelo apoio incondicional e por compreender minhas ausências neste período.

Aos meus filhos, **Marco Antonio** e **Ana Luiza** pelo carinho e incentivo nas horas difíceis.

A toda a minha família e amigos, que sempre me acompanharam, mesmo à distância, pelo incentivo, carinho e amor.

Sumário

Índice de Figuras e Quadros

Lista de Siglas e Abreviaturas

API	Application Program Interface
BDD	Base de Dados Distribuída
COM	Component Object Model
CORBA	Common Object Request Broker Architecture
DBMS	Data Base Management System
DCE	Distributed Computing Environment
DCOM	Distributed Component Object Model
DLL	Dynamic Link Library
DSL	Digital Subscriber Line
ECG	Esquema Conceitual Global
ECL	Esquema Conceitual Local
EE	Esquema Externo
FIS	Federeted Information System
IEL	Esquema Interno Local
HTTP	Hiper Text Transfer Protocol
IDL	Interface Definition Language
IP	Internet Protocol
IPX	Internetwork Packet Exchange
JDBC	Java DataBase Connectivity
JVM	Java Virtual Machine
LAN	Local Area Network
MAN	Metropolitam Area Metwork
MBIS	Mediator-base Information Systems
MSL	Mediator Specification Language
ODBC	Open DataBase Connectivity
ODMG	Object Database Management Group
OMG	Object Management Group
ORB	Object Request Broker
OSF	Open Software Foundation
PMS	Peer Mediator System
RMI	Remote Method Invocation
RPC	Remote Procedure Call
SCM	Service Control Manager
SGBD	Sistema Gerenciador de Banco de Dados
SGBDD	Sistema Gerenciador de Banco de Dados Distribuído
SPX	Sequenced Packet Exchange
SQL	Structured Query Language
SVBDD	Sistema de Vários Bancos de Dados Distribuídos
TCP	Transmission Control Protocol
TCP/IP	Transmission Control Protocol/Internet Protocol
UDP	User Datagram Protocol
UML	Unified Modeling Language
WAN	Wide Area Network
WWW	World Wide Web
XML	Extensible Markup Language

Resumo

Este trabalho faz uma avaliação dos recursos disponíveis para viabilizar a interoperabilidade de bases de dados distribuídas e heterogêneas e propõe um modelo alternativo para a solução do problema. O modelo proposto se apresenta como definição metodológica, possível de ser implementado em qualquer ambiente operacional e aplicável em qualquer base de dados. O objetivo deste trabalho é mostrar que as alternativas disponíveis para solucionar esta problemática, quando já implementadas, apresentam um alto grau de complexidade e não atendem adequadamente todas as demandas relacionadas com a interoperabilidade de bases de dados distribuídas e heterogêneas. O trabalho apresenta uma proposta de mediador, modelado em UML (Unified Modeling Language), que permite a interoperabilidade entre bases de dados distribuídas e heterogêneas. Também é apresentada a implementação do modelo, onde são exploradas as características, comportamento e avaliada sua aplicabilidade em diferentes ambientes operacionais e linguagens de programação.

Palavras-chave: Mediadores, Bases de Dados, Interoperabilidade em Ambientes Heterogêneos.

Abstract

This research does an assessment of the available resources to perform the interoperability of the distributed and heterogeneous databases and suggests an alternative model for the solution of this problem. The suggested model is shown as a methodological definition that is possible to be implemented in any of the existing operating systems and suitable to any database. The goal of this study is to show that the available alternatives to solve this problem, when already implemented, show a high complexity level and do not meet all the demands related to the interoperability of distributed and heterogeneous databases. The paper presents a proposal of a *middleware*, modeled using UML (Unified Modeling Language), that allows the interoperability among distributed and heterogeneous databases. There is also presents the implementation of the model where the characteristics are exploited, as well as its behavior and its application is evaluated in different operating systems and programming languages.

Key-words: *Middleware*, Database, Interoperability in Heterogeneous Environments.

Capítulo 1 Introdução

1.1 Motivação

O advento da Internet, o aumento da disponibilidade de infra-estrutura de comunicação, o avanço nas tecnologias de fabricação de hardware e desenvolvimento de software criam um ambiente propício ao desenvolvimento de aplicações distribuídas nas mais diversas áreas. A diversidade de aplicações possíveis neste novo ambiente exige a integração de vários recursos nas áreas de redes, sistemas operacionais, ambientes de programação, sistemas de armazenamento de dados e metodologias de projeto e desenvolvimento de sistemas.

Nas organizações, o sistema de informações pode ser composto por apenas um sistema, onde todas as informações são armazenadas em um único repositório de dados, ou distribuídos por vários subsistemas com seus repositórios de dados específicos. Quanto maior o porte das organizações, maior é a ocorrência da segunda alternativa que se caracteriza pela perda de autonomia dos subsistemas, necessidade de uma administração de dados centralizada e pela dificuldade de manipulação das informações dispersas pelos vários repositórios de dados.

A integração de bases de dados de diferentes sistemas tem sido um tópico de constantes pesquisas. As incompatibilidades de tipos de rede desapareceram com o sucesso da Internet e a distribuição física passou a ser gerenciada pela utilização de *frameworks* e ferramentas, como CORBA e Java.

Para integrar bases de dados heterogêneas, se faz necessária a aplicação de *esquemas de integração* onde são mapeadas as informações das diversas bases de dados. A questão mais crítica desta tecnologia é a autonomia. Quando um recurso de dados muda seu conteúdo (ou formato) não pode notificar o sistema de integração sobre estas mudanças ocorridas, exigindo a pesquisa e o desenvolvimento de novas técnicas. Da mesma forma, muitos paradigmas do desenvolvimento de software estão mudando, em função das possibilidades oferecidas pelo desenvolvimento de novas tecnologias de comunicação de dados, permitindo uma maior interoperabilidade entre bases de dados e processos.

1.2 Objetivos

O objetivo geral deste trabalho é de propor um modelo de mediador, que dê ênfase à utilização dos recursos e infra-estrutura de comunicação existentes nos ambientes operacionais e na aplicação de metodologias e notações de desenvolvimento de sistemas. Consideramos que a combinação destes dois fatores permite atender todas as demandas relacionadas à interoperabilidade de bases de dados distribuídas e heterogêneas.

Os objetivos específicos são:

- Estudar a infra-estrutura e recursos disponíveis para comunicação em sistemas distribuídos.
- Estudar as características e recursos de interoperabilidade das bases de dados.
- Estudar as principais e modelos de mediadores que se aplicam à interoperabilidade entre bases de dados distribuídas e heterogêneas.
- Projetar e implementar um modelo de mediador que viabilize a interoperabilidade utilizando recursos já disponíveis nos ambientes operacionais.

- Validar a aplicabilidade do modelo com sua implementação em um estudo de caso.

Pretendemos mostrar com este trabalho, que podemos ter bons resultados na implementação de soluções de interoperabilidade aplicando recursos nativos dos ambientes operacionais, como *sockets*, e padrões de modelagem de sistemas, como a UML.

1.3 Conteúdo do Documento

Este documento descreve as características de ambientes distribuídos e os recursos disponíveis para resolver a problemática da interoperabilidade de bases distribuídas heterogêneas, propondo um modelo alternativo em UML, que foca metodologia de desenvolvimento e não produto. No Capítulo 2 é apresentada a conceituação da comunicação em sistemas distribuídos, onde são descritos os componentes da infra-estrutura de software, que viabilizam a implementação de aplicativos distribuídos em ambientes operacionais heterogêneos. No Capítulo 3 discorremos sobre os sistemas aplicativos, bases de dados e as características de distribuição e heterogeneidade apresentadas pelos SGBDDs (Sistemas Gerenciadores de Bancos de Dados Distribuídos). No Capítulo 4 são apresentados conceitos, propostas e soluções que se propõe a atender as demandas da interoperabilidade de bases de dados em ambientes distribuídos e heterogêneos. O Capítulo 5 apresenta a proposta de Mediador e sua implementação em JAVA, utilizando *sockets* como mecanismo de comunicação. O Capítulo 6 traz um estudo de caso onde é utilizado o Mediador proposto. No estudo de caso a mediação é simulada nos ambientes operacionais Windows e Linux, sendo seus componentes implementados nos ambientes de desenvolvimento JAVA e Delphi. O Capítulo 7 apresenta as conclusões deste trabalho e as perspectivas de trabalhos futuros.

Capítulo 2

Comunicação em Sistemas Distribuídos

No início de sua utilização, os computadores operavam independentemente uns dos outros, não havendo qualquer comunicação entre eles. O hardware e software eram proprietários e normalmente, fornecidos pelo mesmo fabricante. Em função das limitações de hardware, as aplicações eram desenvolvidas para propósitos específicos e só permitiam atender funções repetitivas e com baixo nível de inteligência. O compartilhamento de dados entre ambientes operacionais distintos, quando possível, era mínimo e feito de maneira rudimentar através de meios de armazenamento físico, como cartões e fitas de papel, fitas magnéticas.

Em um segundo momento, foram desenvolvidos dispositivos que permitiram interconectar computadores em redes [TAN2002]. Em seguida, surgiram os protocolos padronizados que, juntamente com a evolução das tecnologias de construção de hardware, desenvolvimento de software e comunicação de dados, permitiu a implementação de aplicações em sistemas de computação distribuída.

Um **sistema de computação distribuída,** consiste de uma coleção de elementos de processamento, não necessariamente homogêneos, que são interconectados por uma rede de computadores, que cooperam para executar determinadas tarefas [ELM2000]. O objeto geral dos sistemas de computação distribuída é dividir um grande problema em pequenas partes e resolvê-las de maneira eficiente e de forma coordenada. A viabilidade econômica desta abordagem se baseia em duas razões: i) maior capacidade de computação é aproveitada para

resolver tarefas complexas, e ii) cada elemento de processamento autônomo pode ser gerenciado independentemente e desenvolver suas próprias aplicações.

Este capítulo apresenta os principais componentes de hardware e software que caracterizam os sistemas de computação distribuída.

2.1 Redes

As redes utilizadas na construção de sistemas distribuídos são compostas por mídias de transmissão (cabos, fibras e canais wireless) dispositivos de hardware (hub, switch, bridges e repetidores) e componentes de software (protocolos de comunicação e drivers). Os computadores e os dispositivos destinados à comunicação são chamados de *host*. Cada um dos computadores e componentes de conexão de uma rede são chamados de *nós.*

Mais recentemente, com a crescente utilização da Internet, surgiram muitos novos nós, aumentando as exigências de disponibilidade, escalabilidade, mobilidade, segurança e qualidade de serviço nos ambientes de rede.

As redes podem ser classificadas de acordo com a sua distribuição geográfica, conforme segue:

• LAN - Local Area Network: São as redes locais. Permitem uma comunicação em alta velocidade, chegando atualmente a *gigabit* por segundo. Os *nós* podem ser interconectados por cabos ou por rede sem fio(*wireless).* Estão fisicamente distribuídas dentro de um prédio ou campus.

• WAN - Wide Area Network: São formadas pela interconexão de várias redes locais (LANs) que podem estar geograficamente distribuídas em prédios, cidades ou continentes. A velocidade de comunicação é normalmente menor que nas redes locais, em função da disponibilidade e custo do suporte de comunicação.

• MAN - Metropolitan Area Network: São redes baseadas em suporte de comunicação de alta velocidade, como fibra ótica, instaladas em cidades para transmissão de dados, voz e imagem a uma distancia de até 50 quilometros. A tecnologia DSL (Digital Subscriber Line) e Cable Modem são as mais utilizadas.

• Redes sem fio: A comunicação sem cabo pode ser utilizada em LANs e WANs. São utilizados equipamentos de comunicação por rádio frequência, infra-vermelho ou satélite. São utilizadas em ambientes onde existe dificuldade de instalação de cabos e necessidade de mobilidade. Estão enquadrados nesta categoria os sistemas de comunicação por telefonia celular.

• Internetwork: A Internetwork é um sistema de comunicação na qual várias redes são conectadas, provendo um ambiente comum de comunicação de dados, através da conciliação das várias tecnologias e protocolos de seus componentes [COU2001]. As Internterworks são compostas de vários componentes de rede. São interconectadas por *roteadadores* e *gateways* e uma camada de software que suporta o endereçamento e transmissão de dados pelos diversos *nós*. A Internet é o melhor exemplo de Internetwork.

2.2 Sistemas Operacionais

Um sistema operacional provê dois serviços fundamentais ao usuário. O primeiro permite utilizar o hardware de um computador criando uma máquina virtual que difere da máquina real, facilitando o uso desta máquina pelo usuário. O segundo, gerencia o compartilhamento do hardware com os vários usuários.

Segundo Tanenbaum [TAN2002], existem dois tipos de sistemas operacionais: sistemas operacionais multi-processados, que gerenciam recursos de um multi-processador e sistemas operacionais multi-computador, desenvolvidos para multi-computadores homogêneos. Os

sistemas operacionais são ainda classificados em fracamente acoplados e fortemente acoplados. Os sistemas operacionais fracamente acoplados são os com estações independentes ligadas por uma rede local e seu principal objetivo é o gerenciamento dos recursos de hardware. Em caso de perda de conexão entre as estações da rede, as estações continuam funcionando localmente, ou seja, os nós da rede tem um baixo grau de interação. Já nos sistemas operacionais fortemente acoplados, o software integra fortemente cada nó da rede e com a interrupção da conexão de um dos nós da rede, a aplicação poderá ser descontinuada. O principal objetivo dos sistemas operacionais fortemente acoplados é oferecer serviços locais para clientes remotos. Existem ainda os *middleware*, que são uma camada adicional nos sistemas operacionais que provêem transparência de distribuição.

Os sistemas operacionais tradicionalmente são construídos para gerenciar computadores com um único processador, sendo que a maioria dos sistemas operacionais modernos possuem extensões para rodar em computadores com multi-processadores, tendo acesso a memória de forma compartilhada. Um importante objetivo dos sistemas operacionais é fazer com que o número de processadores seja transparente para as aplicações. A diferença nos dois casos é que os dados da memória principal são protegidos contra a concorrência de acesso através de primitivas de sincronização, que evitam o acesso de mais de um processador ao mesmo endereço de memória.

Os sistemas operacionais que gerenciam mais de um processo simultaneamente são chamados *multitarefa*, que são implementadas através de *máquinas virtuais.* Temos como exemplo o Unix/Linux e o Windows, que dividem o trabalho a ser executado pelo processador em *threads*, dando a cada uma delas memória, uma fração de tempo do processador e demais recursos do sistema. Os sistemas operacionais *multitarefa* podem executar as *threads* em um único processador ou distribuí-las pelos vários processadores de um mesmo computador. Um importante aspecto do compartilhamento de recursos em uma máquina virtual, é que as aplicações são protegidas uma da outra, sendo que duas

aplicações que são executadas no mesmo tempo, tem seus dados protegidos pelo sistema operacional. Do ponto de vista do usuário, vários processos executam simultaneamente em um computador com um único processador.

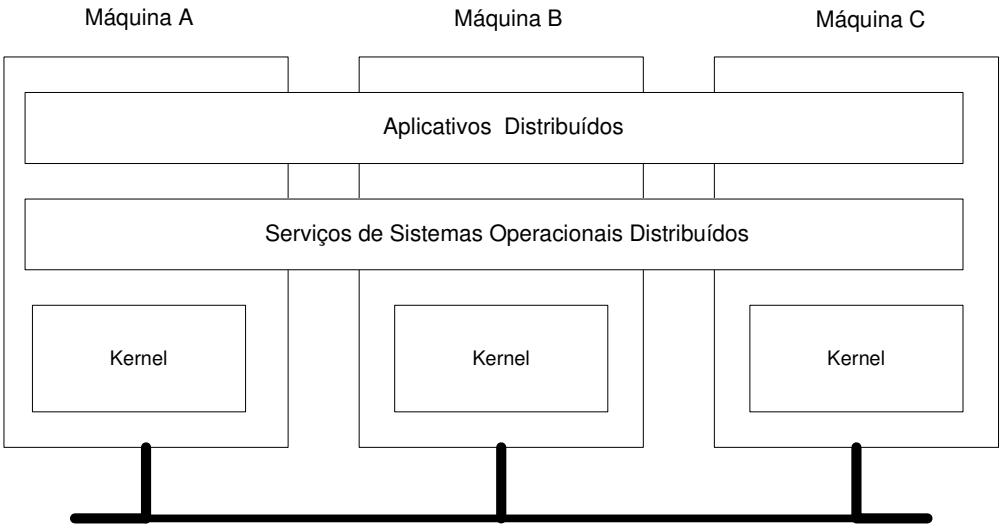

Figura 1 - Sistema Operacional Multi-computador
Fonte Tanenbaum et al [TAN2002]

Sistemas operacionais para multi-computadores possuem estruturas e complexidade diferentes de sistemas operacionais para multi-processadores. A diferença é que cada nó da rede tem seu próprio kernel contendo módulos de gerenciamento local de recursos, como memória, processador e disco. Cada nó é separado fisicamente e comunica-se com os outros nós do ambiente através do envio e recebimento de pacotes de mensagens.

Acima de cada kernel temos uma camada comum de software que implementa um sistema operacional similar a uma máquina virtual, suportando execuções paralelas e concorrentes de várias tarefas (*tasks*).

Ao contrário dos sistemas operacionais distribuídos, os sistemas operacionais de rede não exigem um suporte de hardware homogêneo que

pode ser gerenciado como se fosse um único sistema [TAN2002]. Geralmente são construídos por vários computadores mono-processados, com diferentes sistemas operacionais, conectados umas às outras através de uma rede de computadores.

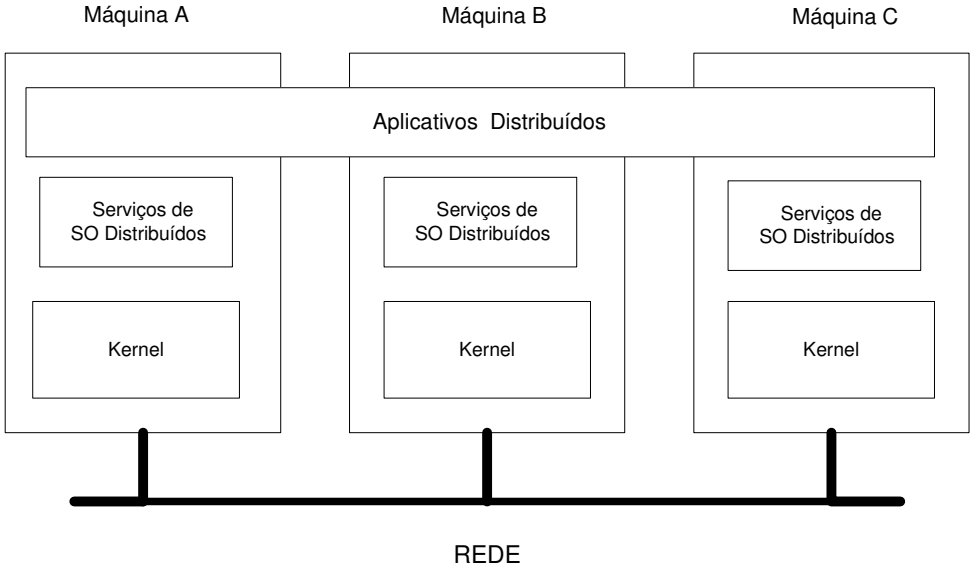

Figura 2 - Sistema Operacional de Rede
Fonte Tanenbaum et al [TAN2002]

Os sistemas operacionais de rede permitem ao usuário fazer o uso de serviços disponíveis em máquinas específicas da rede, como execução de processos, cópia de arquivos. Para ter acesso aos serviços de outro computador da rede, o usuário deve se *logar* na máquina remota. O acesso às máquinas remotas é inteiramente manual e o usuário deve ter seu acesso disponibilizado em cada ponto da rede para ter acesso a estes.

Para prover escalabilidade, independência de sistema operacional e transparência em sistemas operacionais distribuídos são utilizados *middleware*, que consistem de uma camada adicional de software sobre sistemas operacionais de rede.

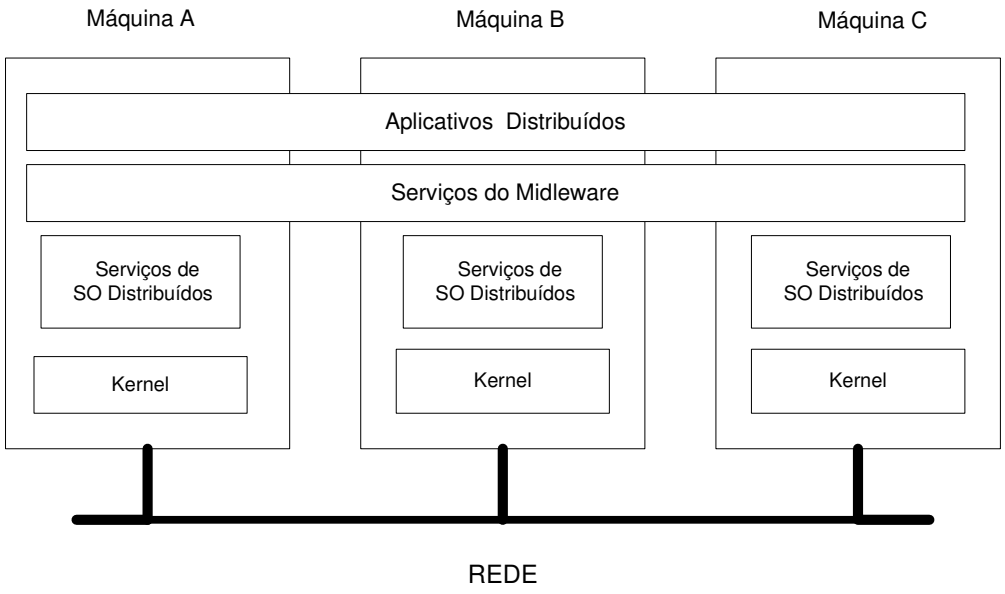

Figura 3 - Sistema Operacional Distribuído – Middleware
Fonte Tanenbaum et al [TAN2002]

Uma importante função dos *middleware* em sistemas distribuídos é viabilizar a heterogeneidade de plataformas nos níveis abaixo, através de seus serviços. Existem vários modelos de implemantação de *middleware*, dentre os quais, sistemas de arquivos distribuídos, Chamada Remota de Procedimentos (RPC), objetos distribuídos e documentos distribuídos (WWW). Os *middleware* provêem distribuição e transparência em sistemas operacionais distribuídos.

2.3 Comunicação e sincronização entre processos

Em aplicações paralelas, se faz necessária a comunicação entre partes de programas executando em diferentes processadores, assim como a sincronização de suas ações. Existem duas categorias de comunicação entre processos: a *troca de mensagens* e o compartilhamento de dados através de *threads*.

2.3.1 Troca de mensagens

Muitos fatores devem ser considerados durante o processo de troca de mensagens entre processos: quem está enviando, o que é enviado, para quem é enviado, qual a garantia que a mensagem chegou e se foi aceita pelo processo ou processos destino, se o processo destino já está criado, se existe retorno da mensagem, o que acontece em caso de erro. Os fatores acima estão colocados de maneira geral, sendo que podemos especificar quatro modelos de troca de mensagens que são *mensagens ponto-a-ponto, rendezvous, chamada remota de procedimentos* e *mensagens de um para muitos.*

A troca de *mensagens ponto-a-ponto* é a mais comum e elementar entre a linguagens e provê a comunicação entre um processo emitente e um processo recebedor. O emitente inicia a comunicação de forma explícita, por exemplo, enviando uma mensagem ou invocando uma *chamada remota de procedimento*. Pelo lado do recebedor a mensagem pode ser recebida de forma explícita, através de um comando *accept*, ou de forma implícita pela invocação do processo. Isto normalmente cria uma nova *thread* de controle dentro do processo recebedor. O recebimento explícito de mensagens permite um maior controle por parte do processo recebedor, que pode estar em diferentes estados e aceitar diferentes tipos de mensagens para cada estado. Um controle mais acurado é possível caso a aceitação da mensagem possa ser feita de forma condicional, dependendo dos argumentos da mensagem. Por exemplo, um servidor de arquivo pode aceitar uma requisição de abertura de um arquivo somente se o arquivo não estiver alocado, assim como em algumas linguagens o programador é quem controla a ordem da troca das mensagens, ou ainda a ordem das mensagens depende de seu conteúdo ou emitente. As mensagens podem ser trocadas de forma *direta* ou *indireta*. Na forma *direta* as mensagens são endereçadas a processos específicos. A forma *indireta* pode ser ilustrada da seguinte forma: as mensagens são enviadas através de um correio e depositadas em caixas postais, que são acessadas pelos processos que possuírem a chave da caixa postal. As mensagens podem

ser trocadas de forma *síncrona* ou *assíncrona*. Na troca de mensagens *síncrona*, o processo emitente fica bloqueado até que o recebedor tenha acolhido a mensagem (implicitamente ou explicitamente).

O modelo *rendezvous* é baseado em três conceitos: a *declaração de entrada, a chamada de entrada* e *a declaração de aceitação*. A *declaração de entrada* e *a declaração de aceitação* fazem parte do código do servidor, enquanto a *chamada de entrada* fica no cliente. A *declaração de entrada* é sintaticamente parecida com uma declaração de procedure e a *chamada de entrada* é similar à chamada de procedimento. A interação *rendezvous* ocorre entre dois processos, X e Y, quando X chama uma entrada de Y, e Y executa uma declaração *accept* para a entrada. A interação é síncrona, então, quando o primeiro processo, que está pronto para interagir, espera pelo outro. Enquanto executa a declaração, Y acessa os parâmetros de entrada fornecidos por X. Y pode transferir valores para os parâmetros de saída, que são passados de volta para X. Após a execução desta operação, X e Y continuam sua execução em paralelo. Y pode ainda continuar executando a solicitação de X, sendo que X não estará mais bloqueado.

Na *chamada remota de procedimentos*, o processo X chama um procedimento remoto P do processo Y, passando parâmetros. Quando Y recebe a invocação, executa o procedimento P e passa o resultado através de parâmetro de volta ao processo X. Durante a execução de P, X e Y ficam bloqueados. O processo X só é desbloqueado quando recebe de volta os parâmetros de Y. Esta é a diferença para o mecanismo *rendezvous*, onde o processo que chama é desbloqueado logo que a declaração estiver sendo executada [TAN89]. Assim como *rendezvous*, a *chamada remota de procedimentos* é uma interação síncrona.

Mensagens de um para muitos: muitas redes suportam o envio de mensagens por *broadcast* ou *multicast*. Uma mensagem de *broadcast* é enviada para todos os processadores conectados à rede e uma mensagem *multicast* é enviada para conjuntos específicos de processadores.

Geralmente não existe a garantia de que todas as mensagens cheguem ao seu destino.

2.3.2 Threads

O compartilhamento de dados da memória em um computador pode ser feito através do recurso de *threads*. As *threads* são um recurso dos sistemas operacionais e viabilizam uma forma de comunicação muito rápida e de sincronização entre as partes de programas concorrentes. Múltiplos processos podem ser concorrentemente compartilhados em um mesmo processador e outros hardwares de forma transparente [TAN2002]. Para cada processo que é criado o sistema operacional deve criar um espaço de endereço, transferir o programa associado e setar a pilha dos dados temporários.

As *threads* podem ser usadas em sistemas não distribuídos em ambientes mono-processados ou multi-processados. Nos ambientes multi-processados as *threads* são distribuídas pelos vários processadores, compartilhando a mesma área de memória. Uma importante propriedade das *threads* é que elas podem prover um eficiente bloqueio de chamadas sem bloquear o processo inteiro, enquanto a *thread* está executando [TAN2002]. Esta propriedade é muito interessante em sistemas distribuídos, pois permite a comunicação com múltiplas conexões lógicas ao mesmo tempo através de aplicações cliente/servidor *multi-thread.*

A comunicação e sincronização entre processos é utilizada no desenvolvimento de aplicações paralelas e distribuídas, onde devem ser consideradas a concorrência e o paralelismo dos processos. A concorrência permite que partes de um programa sejam executadas independentemente. No paralelismo temos partes concorrentes do programa que podem ser executadas ao mesmo tempo em elementos de processamento separados. A concorrência é uma propriedade do programa e o paralelismo é uma propriedade do equipamento.

2.4 Sockets

Socket ou *soquete* é uma abstração feita pelo Sistema Operacional e um mecanismo básico para a comunicação entre processos [COU2001]. *Sockets* são originados do BSD UNIX, mas também estão presentes no Windows NT/2000 e Macintosh. Antes de ser manipulado o *socket* deve ser criado. Um processo utiliza o comando *socket* para criar um *socket*. Na criação do *socket* são definidos os parâmetros de protocolo comunicação (TCP, UDP, etc). Um *socket* é formado por um endereço IP concatenado com um número de porta. O processo de comunicação consiste em transmitir mensagens entre o *socket* de um processo para o *socket* de outro processo. Os *sockets*, em geral utilizam uma arquitetura cliente-servidor. O servidor espera por pedidos de clientes ouvindo uma porta específica e assim que um pedido é recebido, o servidor aceita uma conexão do *socket* cliente para completar a conexão [SIL2000]. Mensagens são enviadas para um endereço de internet e uma porta específica, só podem ser recebidas por processo que estão associados com aquele endereço de internet e porta. Processos podem ter múltiplas portas para receber mensagens, mas os processos não compartilham portas com outros processos no mesmo computador. Existe uma exceção para o compartilhamento de portas caso o processo esteja utilizando *IP Multicast* [COU2001]. Em geral, servidores podem ter vários pedidos concorrentes e o tempo que um cliente pode ter de esperar para ser atendido por um servidor *mono-thread* pode ser inaceitável. Para resolver essa situação, um servidor pode lidar com pedidos concorrentes atribuindo uma *thread* separada para atender cada pedido que chega.

Existem vários tipos de *sockets* que representam classes de serviço. Os três principais tipos são:

- *Sockets* de Fluxo (Pacotes TCP): Fornecem fluxos de dados confiáveis, duplex e seqüenciados. Não há perda ou duplicação de dados na entrega e não há limites de registros. O processo Cliente e Servidor se comunicam através de um túnel e utilizam as funções *connect()*, do lado do

Cliente, e *accept()* do lado do Servidor. Quando dois processos estabelecem uma conexão, um executa o processo Cliente e o outro executa o processo Servidor e depois podem ser pares [COU2001]. O processo Cliente cria, o *socket* e estabelece a conexão com uma porta do servidor e solicita uma conexão com o processo Servidor. O processo Servidor cria vários *sockets* e fica aguardando as solicitações do(s) processo(s) Cliente(s). Este tipo de comunicação é caracterizado por definir claramente quem é o Cliente e quem é o Servidor.

- *Sockets* de Datagrama (Pacotes UDP): Transferem mensagens de tamanho variável, chamadas de datagramas, nas duas direções. Não existe garantia que essas mensagens chegarão na mesma ordem em que foram enviadas, ou que serão duplicadas, ou mesmo se chegarão, mas o tamanho da mensagem original é preservado em qualquer datagrama que chegue [SIL2000]. Não existe canal de comunicação para a troca de mensagens entre os processos. Os pacotes são jogados na rede com o endereço de destino. São utilizados os comandos *send()*, para enviar as mensagens, e *receive()* para recebê-las. Estes pacotes podem até não chegar ao seu destino. Para enviar ou receber mensagens, deve ser primeiro criado um *socket* para um endereço de internet e uma porta local, sendo que o Sistema Operacional irá ligar o *socket* à porta do Servidor, que indica aos Clientes que pode ser enviada uma mensagem para eles. O Cliente se conecta a uma porta local livre e envia o endereço de internet e porta para o Servidor, sendo que este completa o envio da mensagem [COU2001]. Não há uma distinção entre Cliente e Servidor. Existem *sockets* de mensagens entregues com confiabilidade, ou datagramas confiáveis.

- *Sockets* Brutos: Permitem o acesso direto por processo aos protocolos que suportam os outros tipos de *sockets*. Os protocolos acessíveis incluem não só os de nível mais alto, mas também os de nível inferior. Por exemplo, no domínio da Internet, é possível acessar o TCP, o IP abaixo dele, ou um protocolo Ethernet abaixo dele. É um recurso útil para desenvolver novos protocolos.

Para que outro processo enderece um *socket*, este deve ter um nome, que é a ele associado pela chamada ao sistema *bind*, que utiliza um descritor do *socket*, um ponteiro para o nome e o tamanho do nome como um string de bytes. O tamanho e conteúdo do string de bytes dependem do formato de endereço. A chamada de sistema *connect* é usada para iniciar a conexão.

Um processo servidor utiliza *socket* para criar um *socket* e *bind* para associar ao endereço conhecido do serviço a esse *socket*. A chamada *listen* é utilizada para informar ao *kernel* que ele está pronto para aceitar conexões dos clientes e para especificar quantas conexões pendentes o *kernel* poderá colocar na fila até que o servidor possa atendê-las. Finalmente, o servidor utiliza a chamada ao sistema *accept* para aceitar conexões individuais. O *listen* e o *accept* assumem como argumento o descritor de *socket* original. O *accept* retorna um novo descritor de *socket* correspondente à nova conexão; o descritor de *socket* original ainda está aberto para novas conexões. O servidor geralmente utiliza o *fork* para gerar um novo processo depois do *accept*, para atender ao cliente enquanto o processo servidor original continua a ouvir mais conexões. Existem também chamadas de sistema para definir parâmetros de uma conexão e para retornar o endereço do *socket* externo depois de um *accept*. Quando uma conexão para um *socket* de fluxo (TCP) for estabelecida, os endereços das duas extremidades são conhecidos, e não há necessidade de informações de endereçamento adicionais para transferir dados e as chamadas de sistema *read* e *write* podem ser usadas. A forma mais simples de encerrar uma conexão e destruir um *socket* associado é usar a chamada de sistema *close* sobre o descritor do *socket*. Quando há a necessidade de encerrar apenas uma direção de comunicação em uma conexão duplex , a chamada ao sistema *shutdown* pode ser usada. Para os s*ockets* tipo datagrama, que não oferecem suporte a conexão, são usadas as chamadas de sistema *sendto* e *recvfrom*.

Primitiva	Significado
Socket	Cria uma nova conexão
Bind	Conecta um endereço local ao um *socket*
Listen	Declara disponibilidade para aceitar conexões
Accept	Bloqueia até a resposta da conexão chegar
Connect	Estabelece uma conexão
Send	Envia dados pela conexão
Receive	Recebe dados pela conexão
Select	Multiplexa a transferência de dados
Close	Libera a conexão

Quadro 1 - Primitivas de Sockets

A chamada de sistema *select* pode ser usada para multiplexar transferência de dados em vários descritores de arquivos e/ou *sockets* e também para permitir que um processo servidor ouça conexões de clientes para muitos serviços e efetue um *fork* de um processo para cada conexão, assim que ela for estabelecida. O servidor efetua *socket*, *bind* e *listen* para cada serviço: em seguida, faz um *select* sobre todos os descritores de *socket*. Quando o *select* indicar atividade em um descritor, o servidor fará um *accept* nele e criará um novo processo para o novo descritor retornado, deixando o processo pai fazer um *select* novamente.

2.5 Chamada Remota de Procedimento

A comunicação usando *sockets* é considerada uma forma de comunicação de baixo nível entre *threads* e processos distribuídos. A Chamada Remota de Procedimento (RPC-Remote Procedure Call) é um método de comunicação em um nível de abstração mais alto e permite que uma *thread* chame um procedimento ou função de outro processo. Este outro processo pode estar em um espaço de endereçamento separado no mesmo computador ou em um computador distinto conectado à rede [SIL2000]. A RPC não faz uso de portas explicitamente, possibilitando a qualquer máquina executar um procedimento remotamente. Uma função é criada e registrada como responsável por atender pedidos de RPC.

Quando alguma máquina solicita uma função de RPC, esta função é executada passando parâmetros e obtendo o retorno especificado. A função RPC atende a pedidos como criar processos, enviar e receber mensagens. O processo Cliente inclui uma *procedure stub* ou *proxy* para cada processo na lista de serviços e envia uma mensagem de *request* com sua identificação e parâmetros. O processo Servidor possui um *dispatcher* ou *skeleton* para cada processo Cliente, que tem a função de selecionar no serviço o *stub* identificado na mensagem de *request* e executá-lo. O retorno da mensagem é feito através do *reply*.

A vantagem das RPC em relação aos *sockets* é que o RPC gerencia o canal de comunicação, por isso os aplicativos podem ser escritos de modo que a localização de um procedimento, quer local ou remoto, seja transparente [SIL2000].

2.6 Ambiente de Computação Distribuída DCE

O ambiente de computação distribuída DCE (Distributed Computing Environment) é um *middleware* desenhado para executar como uma camada de abstração entre um sistema operacional de rede e aplicações distribuídas [TAN2002]. É uma tecnologia que foi desenvolvida pelo Open Group, que é um consórcio de usuários e fornecedores que trabalham em conjunto visando o desenvolvimento de tecnologias para sistemas abertos. O DCE foi projetado para trabalhar independente do sistema operacional ou tecnologia de rede, permitindo a interação entre cliente e servidor em qualquer tipo de ambiente.

A tecnologia inclui serviços de software que residem acima do sistema operacional, fornecendo uma interface para os recursos de baixo nível do sistema operacional e recursos de rede. O modelo de programação do DCE é cliente-servidor, onde os clientes acessam os serviços providos remotamente pelos servidores de processos. Alguns destes serviços são parte do próprio DCE, mas outros podem ser aplicações desenvolvidas por programadores em ambientes operacionais

diversos. Toda a comunicação entre os clientes e servidores é feita por recursos do RPC.

O DCE apresenta os seguintes serviços:

- Comunicação: o acesso dos recursos através da rede é efetuado por meio de chamadas de procedimento (RPC).
- Serviço de arquivos distribuídos: provê transparência para a localização e acesso de arquivos do sistema.
- Serviço de diretórios: permite a localização dos recursos do sistema. Os recursos incluem máquinas, impressoras, servidores e dados distribuídos pela rede.
- Serviço de segurança: autentica usuários, autoriza o acesso a recursos em redes distribuídas e contabiliza usuários e servidores. Incorpora a tecnologia Kerberos.
- Serviço de relógio: mantém sincronizados os relógios das máquinas da rede.

O DCE, através do sistema RPC pode fazer a conversão automática de tipos de dados entre o cliente e servidor, permitindo a interação entre diferentes arquiteturas com diferentes tipos de dados. Uma grande variedade de protocolos de rede e representações de dados são também suportados pelo DCE.

2.7 Invocação Remota de Método

A Invocação Remota de Método (RMI – Remote Method Invocation), é um recurso semelhante a RPC, que permite a processos cooperarem com clientes chamando programas em diferentes computadores. A RMI implementa a aplicação do modelo de Objetos Distribuídos que é suportada pela programação orientada a objeto, sendo que as RPCs suportam programação *procedural*. Os parâmetros dos procedimentos remotos na RPC são estruturas de dados comuns; com a RMI é possível passar

objetos como parâmetros para os métodos remotos. Exemplos de sistemas de invocação remota são o CORBA e Java RMI [COU2001].

A invocação de métodos é feita através do *middleware*. A camada de *middleware* usa protocolos baseados em mensagens. Um aspecto importante do *middleware* é a provisão de transparência de localização e independência de protocolos de comunicação, sistemas operacionais e hardware. Algumas formas de *middleware* permitem aplicar componentes escritos em diferentes linguagens de programação. Os sistemas de objetos distribuídos podem adotar a arquitetura cliente-servidor. Neste caso, os objetos são gerenciados por servidores e seus clientes invocam seus métodos usando RMI. No RMI, o *request* do cliente invoca o método do objeto e uma mensagem é enviada ao servidor que gerencia os objetos, e este direciona para a execução do método no servidor e o resultado é retornado ao cliente através de outra mensagem. Objetos em um servidor podem se transformar em clientes de objetos em outros servidores.

Para tornar os métodos remotos transparentes ao cliente e ao servidor, a RMI implementa o objeto remoto usando *stubs* e *skeletons*. Um *stub* é um Proxy (representante) do objeto remoto; ele reside junto ao cliente e quando o cliente invoca um método remoto o *stub* é chamado. Este *stub* no cliente é responsável por criar um pacote, que consiste no nome do método a ser invocado no servidor e nos parâmetros deste método, um processo conhecido como *mashalling* de parâmetros. O *stub* envia então esse pacote para o servidor, onde ele é recebido pelo *skeleton* do objeto remoto. O *skeleton* é responsável por efetuar a operação de *unmarshallling* (extração) dos parâmetros e por invocar o método desejado no servidor. O *skeleton* agrega o valor de retorno (ou exceção, se houver) em um pacote e retorna-o ao cliente. No cliente, o *stub* efetua a extração do valor de retorno e o passa ao cliente.

Um meio usual de prover suporte RMI é especificar as interfaces dos objetos em IDL (Interface Definition Language). Outra alternativa é utilizar uma linguagem orientada a objeto como o JAVA, que controla o *stub*

automaticamente [TAN2002]. A *invocação estática* requer que as interfaces dos objetos sejam conhecidas pelas aplicações que vão utilizar estes objetos. Isto implica que em caso de mudança das interfaces as aplicações clientes tenham que ser recompiladas. Já na *invocação dinâmica*, a aplicação seleciona um programa executável como método, e este irá invocar o objeto remoto.

Capítulo 3
Bases de Dados e Ambientes Distribuídos

Base de dados é uma coleção de dados relacionados. Dados são fatos que podem ser armazenados e que tem um significado implícito [ELM2000]. As bases de dados têm implícitas as seguintes propriedades:

- Representam algum aspecto do mundo real, e as mudanças do mundo real se refletem nas bases de dados.
- São uma coleção lógica e coerente de dados com significado inerente.
- São designadas, construídas e populadas com dados para um propósito específico.

As bases de dados podem ter qualquer tamanho e variação de complexidade. As bases de dados computadorizadas podem ser criadas e mantidas por uma coleção de programas escritos especificamente para esta finalidade ou por sistemas gerenciadores de banco de dados (SGDBs).

Os sistemas de computação distribuída, ou ambientes distribuídos, são constituídos de uma coleção de elementos de processamento, não necessariamente homogêneos, que são interconectados por uma rede de computadores, que cooperam para executar determinadas tarefas.

As bases de dados distribuídas (BDDs), que são uma coleção de múltiplas e logicamente inter relacionadas bases de dados, distribuídas em uma rede de computadores. As bases de dados distribuídas (BDDs) surgem com a evolução de duas tecnologias: i) tecnologias de bancos de dados, e (ii) tecnologias de redes e comunicação de dados [ELM2000].

No início de sua utilização, as bases de dados seguiram o caminho da centralização, que resultou em gigantescas e monolíticas bases de dados. A partir dos anos oitenta, a tendência passou a ser a descentralização e autonomia de processamento. Com o avanço na

computação distribuída e as possibilidades oferecidas pelos sistemas operacionais distribuídos, as pesquisas de bases de dados seguiram para a distribuição de dados, processamento de transações distribuídas, gerenciamento de metadados. Surgem então os sistemas gerenciadores de bancos de dados distribuídos (SGBDDs), que são sistemas de software que gerenciam bases de dados distribuídas, de forma transparente para o usuário [OZS2001].

3.1 Sistemas Aplicativos e Bases de Dados

As novas demandas por informação exigem que sejam constantemente implantados novos sistemas nas organizações. Acompanhando as demandas de novos aplicativos, vem a evolução dos sistemas gerenciamento de bases de dados. Os primeiros sistemas utilizavam arquivos sequênciais, que consistiam de uma lista de informações manipuladas e armazenadas em memória, fitas magnéticas e posteriormente em discos rígidos. Junto com a evolução dos meios de armazenamento de informações, veio também a evolução dos sistemas de gerenciamento de bases de dados. Surgiram os sistemas de gerenciamento de arquivos indexados, onde as informações podiam ser armazenadas e recuperados de forma não seqüencial. Posteriormente surgiram os sistemas gerenciadores de banco de dados, que aumentaram significativamente a capacidade de controle e armazenamento de dados.

Em função dos custos e tempo de implementação, as organizações normalmente oferecem grande resistência ao desenvolvimento de novos sistemas para os quais já existe uma solução implantada. Em outros casos pode existir a incompatibilidade entre as tecnologias dos sistemas existentes com as tecnologias aplicadas nos novos sistemas. A heterogeneidade, juntamente com os sistemas abertos, possibilita a combinação de componentes de hardware e software, permitindo a integração entre diferentes ambientes operacionais, que quando corretamente aplicada, produz sistemas operacionalmente viáveis.

Atualmente existem muitas interfaces e pacotes de software que podem ser utilizados para produzir sistemas heterogêneos, mas poucas ferramentas ajudam a lidar com a integração de sistemas isolados em um ambiente distribuído e heterogêneo.

3.2 Bancos de Dados Relacionais Distribuídos

A arquitetura de um sistema define sua estrutura. A especificação da arquitetura de um sistema de software exige identificação de vários módulos, com suas interfaces e inter-relacionamentos em termos de fluxo de dados e de controle. Em um Sistema Gerenciador de Banco de Dados Distribuído (SGBDD), podemos definir três arquiteturas de referência: sistemas cliente-servidor, SGBDD não-hierárquicos e sistemas de vários bancos de dados.

O suporte completo para bancos de dados distribuídos implica em uma única aplicação, que deve ser capaz de operar de modo transparente sobre dados dispersos em uma variedade de máquinas diferentes, gerenciados por SGBDs diferentes, em execução em máquinas diferentes, em sistemas operacionais diferentes e redes de comunicação de dados diferentes [DAT2000]. Considerando que vários bancos de dados podem compartilhar diversos sistemas gerenciadores de banco de dados (SGBDs), podemos classificar sua organização em relação à *autonomia*, *distribuição* e *heterogeneidade*.

- Autonomia: refere-se à distribuição de controle, não de dados. Ela indica até que grau os SGBDs individuais podem operar independentemente. Podemos ter uma integração estreita, na qual uma única imagem do banco de dados está disponível para qualquer usuário que queira compartilhar as informações, que podem residir em vários bancos de dados. Da perspectiva dos usuários, os dados estão logicamente centralizados em um único banco de dados. Em seguida, identificamos sistemas semi-autônomos que consistem em SGBDs que podem operar independentemente. Cada um desses SGBDs determina

que partes de seu próprio banco de dados estarão disponíveis para usuários de outros SGBDs. A última alternativa é o isolamento total, no qual os sistemas individuais são SGBDs independentes, que não tem conhecimento da existência de outros SGBDs e nem se comunicam com eles.

- Distribuição: considera a distribuição física dos dados sobre os diversos sites. Existem duas maneiras de distribuir SGBDs: a distribuição *cliente-servidor* e distribuição *não-hierárquica* (ou distribuição total). A distribuição cliente-servidor, que se tornou bastante popular nos últimos anos, concentra as tarefas de gerenciamento de dados em servidores, enquanto os clientes se concentram em fornecer o ambiente de aplicativo, incluindo a interface com o usuário. Em sistemas *não hierárquicos*, não existe nenhuma distinção entre máquinas clientes e servidores. Cada máquina tem toda a funcionalidade do SGBD e pode se comunicar com outras máquinas para executar consultas e transações.

- Heterogeneidade: A heterogeneidade pode ocorrer de várias formas nos sistemas distribuídos, variando desde a heterogeneidade de hardware, diferenças em protocolos de interligação de redes, variações em gerenciadores de dados até heterogeneidade em linguagens de consulta, que utilizam o mesmo modelo de dados, mas selecionam métodos diferentes para expressar solicitações idênticas [OZS2001].

3.2.1 Sistemas Cliente-Servidor

Um sistema cliente-servidor é um sistema distribuído em que: i) alguns *sites* são *sites clientes* e outros são *sites servidores*, ii) todos os dados residem nos *sites servidores*, iii) todas as aplicações são executadas nos *sites clientes* e iv) a operação não é uniforme [DAT2000].

Em sistemas cliente-servidor o *servidor* faz a maior parte do trabalho, gerenciando transações e o armazenamento de dados. O *cliente*, além da aplicação e da interface com o usuário, tem o módulo cliente de SGBD e é responsável pelo gerenciamento dos dados que são

colocados no *cache* do cliente e algumas vezes pelo bloqueio de transações. O cliente repassa consultas SQL ao servidor, sem compreendê-las ou otimizá-las e o servidor trata a solicitação e devolve ao cliente a relação resultante. Em uma arquitetura mais sofisticada, existem vários servidores no sistema. Neste caso são possíveis duas estratégias alternativas de gerenciamento: cada cliente gerencia sua própria conexão com o servidor adequado, ou cada cliente conhece apenas seu servidor local, que então se comunica com os outros servidores conforme necessário. A primeira abordagem simplifica o código do servidor, mas carrega as máquinas clientes com responsabilidades adicionais conduzindo aos chamados "clientes pesados". Por outro lado, a segunda abordagem concentra a funcionalidade de gerenciamento de dados nos servidores, levando a "clientes leves".

Sob a perspectiva da lógica de dados, os SGBDs cliente-servidor fornecem a mesma visão dos dados que os sistemas não hierárquicos (ponto a ponto), ou seja, eles dão ao usuário a aparência de um banco de dados logicamente único, enquanto no nível físico os dados podem estar distribuídos [OZS2001].

3.2.2 Sistemas Distribuídos Não-Hierárquicos

Em sistemas distribuídos não-hierárquicos, a organização dos dados físicos em cada máquina pode ser diferente, portanto há a necessidade de definir um esquema interno individual em cada *site*, chamado de Esquema Interno Local (EIL). Já a visão da estrutura lógica dos dados em todos os *sites* é chamada de Esquema Conceitual Global (ECG). Os dados em um banco de dados distribuídos, em geral, são fragmentados e replicados. Para lidar com a fragmentação e replicação, a organização lógica dos dados em cada *site* tem de ser descrita no Esquema Conceitual Local (ECL). Por fim, o acesso de aplicativos do usuário e o acesso de usuários ao banco de dados são admitidos pelos

Esquemas Externos (EEs), localizados acima do esquema conceitual global.

A transparência de localização e replicação é admitida pela definição dos esquemas conceitual global e local, e pelo mapeamento intermediário. Por outro lado, a transparência de rede é aceita pela definição do esquema conceitual global. O SGBDD converte consultas globais em grupos de consulta local, que são executadas por componentes do SGBDD em diferentes *sites,* que se comunicam uns com os outros. Uma das principais motivações do processamento distribuído é o desejo de exercer o controle local sobre a administração dos dados [OZS2001].

O modelo de arquitetura dos Sistemas de Vários Bancos de Dados Distribuídos (SVBDD), tem como diferença fundamental, em relação aos SGBDD, a definição do esquema conceitual global. No caso de SGBDD logicamente integrados, o esquema conceitual global define a visão conceitual do banco de dados inteiro; por outro lado, no caso dos SVBDD, ele representa apenas a coleção de alguns bancos de dados locais que cada SVBD local quer compartilhar. A hipótese da homogeneidade estrita é sem dúvida forte demais: o que precisamos é que os SGDBs em diferentes *sites* admitam a mesma interface [DAT2000]. As pesquisas em SVBDD são relativamente novas e ainda não geraram soluções de propósito geral.

O projeto de sistemas de computadores distribuídos trata da disposição de *dados* e *programas* pelos *sites* de uma rede de computadores, além do projeto da própria rede. Nos SGBDD, a *distribuição dos programas* inclui dois itens: a distribuição do software do SGBDD e a distribuição dos programas aplicativos que funcionam nele. A solução para estas questões é a distribuição de uma cópia do software SGDBB em cada *site*.

Na *distribuição dos dados,* a questão crucial é o *nível de compartilhamento,* onde existe a possibilidade de não haver nenhum

compartilhamento, onde cada aplicativo e seus dados são executados no mesmo *site,* sem acesso a qualquer arquivo de dados ou programa em outros *sites*. No nível de compartilhamento de dados, todos os programas são replicados em todos os *sites*, mas não os arquivos de dados. Desta forma, as solicitações dos usuários são tratadas nos *sites* de onde se originam e os arquivos de dados não são movidos pela rede. No compartilhamento de dados e programas, onde tanto os dados como os programas podem ser compartilhados, um programa em um determinado *site* pode solicitar um serviço de outro programa em um segundo *site* que, por sua vez, pode ter que acessar dados localizados em um terceiro *site* [OZS2001].

O padrão de acesso às solicitações de usuários pode se alterar no decorrer do tempo. A dinâmica das mudanças de padrões de acesso exige constante administração das estruturas de dados, e sua distribuição e é um fator a ser considerado no projeto. O acesso a múltiplos bancos de dados pode ser feito através de *gateways*, também chamados de *middleware* ou *mediadores*, que são fragmentos de software cujo propósito geral é atenuar as diferenças entre sistemas distintos que devam funcionar juntos de algum modo [DAT2000].

3.3 Banco de Dados de Objetos e Relacional-Objeto

Uma das áreas mais novas na pesquisa de bancos de dados envolve a aplicação do paradigma da orientação a objetos na construção de um banco de dados. Segundo Brookshear et al [BRO2000], a motivação para estes esforços tem, no mínimo, quatro razões:

- A independência de dados, que pode ser alcançada através do encapsulamento.
- Os conceitos de classe e de herança aparecem prontos para a descrição e sub esquemas de bancos de dados.
- A construção de bancos de dados constituídos de objetos inteligentes de dados, capazes de responder às consultas por si próprios, em vez de serem argüidos por um programa supervisor.

- A possibilidade de superar algumas restrições inerentes a outros modelos de banco de dados, como por exemplo, a facilidade de recuperar parte de informação de um único atributo.

Um banco de dados orientado a objetos manipula classes de objetos, onde estão inseridos os atributos e os métodos que tratam estes atributos. Se comparado ao um banco de dados relacional, este manipula tabelas e colunas, sendo que o método que trata estas estruturas é de responsabilidade do sistema aplicativo ou do sistema de gerenciamento do banco de dados. Os adeptos da tecnologia de bancos de dados orientados a objetos argumentam que a imagem de um banco de dados composto de objetos retrata mais fielmente o ambiente do usuário do que um banco de dados que trata relações de tabelas e seus atributos.

Para que os métodos internos aos objetos executem suas tarefas de uma maneira eficiente, deve ser mantida alguma forma de encadeamento entre os diferentes objetos. Uma técnica para obter este encadeamento é estender a composição de objetos para além da encontrada em ambientes tradicionais orientados a objetos. Objetos tradicionais são compostos de estruturas de dados (atributos) e métodos. Para se construir um banco de dados orientado a objetos, essa composição deve ser estendida para incluir um terceiro tipo de componente, uma lista de outros objetos. Portanto, um banco de dados orientado a objetos é constituído de estrutura de dados (atributos), métodos e listas de objetos. O resultado é um banco de dados constituído de objetos, que mantém registros sobre a existência de outros objetos, sendo capazes de identificar os objetos apropriados e com eles estabelecer comunicação para responder solicitações de informação.

Nos últimos anos, vários fornecedores lançaram produtos de SGDB “relacional-objeto”, também conhecidos como *servidores universais*, sendo que estes produtos são tentativas de uma aproximação entre as tecnologias de objetos e relacional [DAT2000]. Esta aproximação está

firmemente baseada no modelo relacional, basicamente por admitir o conceito de domínio relacional (ou tipos) de maneira apropriada.

3.4 Conclusão

A evolução dos recursos de hardware, software e as metodologias disponíveis para a implementação de sistemas viabiliza a otimização dos sistemas atuais e a implementação de novos sistemas. A implementação de novos sistemas nas organizações, na maioria das vezes é feita de forma gradativa, sem comprometer a continuidade das atividades das organizações. A parte mais complexa deste processo é a implementação de sistemas aplicativos, onde a implantação de novas tecnologias implica normalmente na mudança das linguagens de programação dos gerenciadores de banco de dados. Em função do tamanho e complexidade dos sistemas e da incompatibilidade das bases de dados e compiladores das linguagens de programação, nem sempre é possível migrar todas as bases de dados para um único ambiente de gerenciamento. Na maioria das vezes, os dados de um tipo de base de dados são replicados para as bases de dados dos novos sistemas. Este procedimento é complexo por exigir o desenvolvimento de interfaces para replicar os dados, além de um controle do processo de réplicas, que nem sempre trazem resultados satisfatórios.

A grande maioria dos sistemas gerenciadores de bancos de dados distribuídos (SGBDDs) que implementaram as funcionalidades e técnicas propostas nas pesquisas de bases de dados distribuídas (BDDs), nunca emergiram comercialmente como produtos viáveis. A maioria dos grandes fornecedores de SGBDDs redirecionaram seus esforços para o desenvolvimento de SGBDDs “puros”, ao invés de desenvolver sistemas baseados em cliente-servidor ou sistemas gerenciadores de bancos de dados heterogêneos [ELM2000].

Portanto, a interoperabilidade em bases de dados distribuídas e heterogêneas ainda não pode ser viabilizada integralmente pelas tecnologias disponíveis até o momento. Para resolver este problema, temos atualmente três alternativas: i) a utilização de interfaces ODBC que

permitem a interligação e troca de dados - quando disponíveis nos bancos de dados, ii) o desenvolvimento de aplicações específicas, normalmente baseadas em troca de arquivos, ou iii) quando possível, a utilização de mediadores ou *middleware*, que estão disponíveis em alguns ambientes de desenvolvimento e sistemas operacionais.

Capítulo 4
Modelos e Tecnologias de Mediadores

4.1 Introdução

A maioria dos aplicativos desenvolvidos nas organizações, recaem na existência de uma administração central dos dados [BUS2002]. Isto ocorre mesmo quando a organização é constituída por diversas unidades independentes, cada uma com um repositório de dados local, ou quando são utilizados mais de um sistema de gerenciamento de base de dados.

O aumento da demanda por desenvolvimento de novos aplicativos, a necessidade de manutenção de sistemas legados e a possibilidade de interligação de múltiplos equipamentos de ambientes operacionais diferentes, evidenciam a necessidade de utilização de componentes que viabilizem a interoperabilidade dos dados dos diversos sistemas. Neste contexto, os mediadores surgem como uma alternativa que viabiliza a interoperabilidade nestes ambientes.

O termo "mediação" pode ser relacionado com termo "*brokering*" que é utilizado no *middleware* CORBA. A possibilidade de combinar dados e funcionalidades, característico no paradigma de orientação a objetos, nos permite um grande número de possibilidades de estilos de integração e interoperabilidade entre um grande número de infra-estruturas de informação. Não somente dados estáticos são tratados por sistemas de mediação, mas também as operações sobre os dados.

4.2 Sistemas Federados de Informação

Um Sistema Federado de Informações (FIS – Federated Information System), consiste de um conjunto de sistemas de informação distintos e autônomos, que colaboram com os demais sistemas da federação [BUSSE99]. Sua principal característica é sua construção como uma

camada de integração entre aplicações legadas e bases de dados. A taxonomia de Özsu e Valduriez [ÖZS2001], classifica em 3 dimensões as alternativas para distribuição de bases de dados: autonomia, distribuição e heterogeneidade. A autonomia refere-se à capacidade de manter o controle de uma transação global entre as diversas bases de dados, podendo haver uma forte integração, semi-autonomia e total autonomia. A interoperabilidade fica mais complexa à medida em que aumenta a distribuição e heterogeneidade das bases de dados. A interoperabilidade da informação não está somente relacionada aos meios físicos que suportam a troca de informações entre os vários subsistemas, mas também com os padrões de funcionalidade e estrutura da informação [BUS2002]. A modularização de bases de dados é utilizada para definir módulos de dados que serão compartilhados pelos diversos subsistemas através de objetos integradores.

O acesso integrado às múltiplas, distribuídas, heterogêneas e autônomas bases de dados ou outros recursos de informação pode ser classificada em *lazy* ou *eager* [WID96]. No acesso *lazy* as informações são tratadas sob demanda, diretamente nos repositórios de dados, utilizando-se para isto de mediadores. Nos acesso *eager* as informações são previamente extraídas e armazenadas em repositórios logicamente centralizados, no qual serão feitos os acessos às informações. A abordagem *lazy* é mais adequada para informações que mudam rapidamente e tem pouca previsibilidade em relação às demandas dos diversos recursos e sistemas envolvidos, como por exemplo, na *world-wide-web*. Todavia pode implicar em ineficiência pela demora no acesso ou pela indisponibilidade dos dispositivos de armazenamento das bases de dados, podendo ser inviável em sistemas que não permitam o acesso *ad hoc* a suas informações. Já na abordagem *eager* ou *wharehousing*, as informações integradas estão disponíveis para acesso imediato, sendo recomendada para sistemas com requisitos específicos, que necessitem

de alta performance no acesso e não necessariamente a informação mais atualizada.

Em um ambiente de bases de dados homogêneas, os SGBDs normalmente apresentam um alto grau de abstração, provendo recursos de acesso distribuído e réplica das diversas bases de dados. A arquitetura dos SGBDs homogêneos e distribuídos, segundo o modelo ANSI/SPARC [ÖZS2001], é composta pelo Esquema Conceitual Global (ECG), Esquemas Externos (EE) e Esquema Conceitual Local (ECL), que descrevem a estrutura lógica dos dados distribuídos pelos diversos sites. Estes recursos também podem ser implementados em *stored procedures,* que permitem a definição de procedimentos públicos e privados de cada módulo em um ambiente com vários SGDBs. Já em um ambiente com bases de dados heterogêneas, existe a necessidade de modelar e desenvolver componentes de software para gerenciar as conexões e troca de informações entre as bases de dados. Estes componentes podem aplicar e integrar recursos existentes nos diversos SGBDs existentes no ambiente em questão. Em [BUS2002] os objetos integradores são caracterizados e aplicados em diversos tipos de conversão, tais como: dos dados, suas estruturas, valor semântico, regras de validação, procedimentos, gerenciadores e interfaces. Suas funcionalidades são distribuídas pelos serviços de identificação, métodos, mapeamento, regras ativas, conectividade e execução. A utilização de objetos integradores pressupõe o conceito de desenvolvimento de sistemas e acesso a servidores de dados em ncamadas. Objetos integradores, definidos em [KAT2002] como mediadores, devem permitir o compartilhamento e integração de dados em um grande número de autônomos, distribuídos e heterogêneos recursos e serviços de computação. Os Mediator-base Information Systems (MBIS) são definidos em [BUSSE99] como componentes de software que mediam os usuários e o recurso de dados físico.

4.3 Sistemas Integradores de Bases de Dados (Mediadores)

A principal tarefa no desenho de sistemas integradores de dados é estabelecer o mapeamento entre os recursos de dados e o esquema global, assim como permitir um controle apropriado do sistema [LEN2002]. Os mediadores podem ser concebidos com uma arquitetura centralizada, onde existe um gerenciamento em um único local dos acessos às diversas bases de dados. Já a arquitetura *peer-to-peer* caracteriza-se pela utilização de diversos mediadores, fisicamente distribuídos, com especialidades específicas, integrando um pequeno número de recursos e compartilhando os dados com níveis superiores de mediação.

A área de interesse de nossa pesquisa está focada em soluções de *middleware* como objetos integradores ou mediadores. Como exemplo, temos a arquitetura Peer Mediator System (PMS) proposta por Katchaounov [KAT2002], que se caracteriza por bases de dados virtuais que definem um processo lógico único para acesso aos vários recursos de armazenamento de informações, com todos os acessos e transformações das informações sendo efetuadas em tempo de execução. O PMS apresenta as seguintes características:

- A materialização das recuperações e atualizações das informações nas bases de dados;
- Faz o acesso aos dados através da base de dados virtual, sem o prévio conhecimento do tipo de acesso à base de dados real;
- Requer somente uma pequena parte das informações da base de dados real;
- Otimiza o acesso às informações através de planos de execução, reduzindo ao mínimo o acesso às bases de dados reais;
- Reutiliza os recursos computacionais já disponíveis, como algoritmos de extração e atualização das bases de dados;
- Permite o controle de acesso, segurança e contabilização das transações.

A arquitetura do PMS é baseada em componentes de software funcionalmente distribuídos em três camadas: *recursos de dados*, *mediadores* e *aplicações de usuários*. Os *recursos de dados* provêem primitivas de acesso a componentes externos, que invocam recursos computacionais a fim de enviar e receber dados.

As primitivas de acesso formam a interface de baixo nível. Em [KAT2002], estas interfaces são caracterizadas como *recursos de dados globais* e *recursos de dados locais*. Os recursos de dados globais são globalmente identificados e acessados por sistemas remotos da rede, como por exemplo: Web sites, dispositivos de pesquisa na Internet, Web Services, DNS, etc. Os recursos de dados locais não possuem uma identificação única e global, mas sim métodos de acesso para serem invocados por componentes externos da rede. Temos como exemplos de recursos ODBC (Open DataBase Connectivity), JDBC (Java DataBase Connectivity), arquivos locais e componentes de software acessados por API (Application Program Interface). Para permitir que os recursos de dados locais sejam acessados por todos os *peers* no PMS, um ou mais mediadores servem de intermediários entre os recursos locais e o resto do PMS. Cada mediador do PMS é tratado como um recurso de dados.

Para utilizar as aplicações legadas em seu ambiente operacional, um mediador deve ser capaz de suportar acesso a dados (como ODBC e JDBC) e permitir uma fácil implementação de novas interfaces. Para isto, um mediador deve conter dois níveis de interfaces especializadas: *i)* interface de baixo nível de rede implementada sobre a camada de transporte como o TCP/IP que permite a interconexão entre a aplicação e o mediador, independente de linguagens de programação, sistemas operacionais e hardware e *ii)*interface de alto nível que serve como um *gateway* de aplicação para os outros mediadores, que é necessária para prover as atuais e futuras aplicações a habilidade de acesso simples e transparente aos mediadores.

As funcionalidades dos mediadores estão distribuídas em duas camadas: a *camada DBMS (Data Base Management System)*, que é responsável pela integração das consultas dos usuários e a *camada wrapper* que é responsável pelo acesso aos recursos de acesso aos dados. *Wrappers* são desenhados de forma genérica, sendo que um *wrapper* pode acessar múltiplas instâncias de um mesmo tipo de recurso de dados.

A arquitetura *Mediator-based Information Systems* [BUSSE99] é composta por uma federação de mediadores provendo serviços de mediação, onde cada mediador tem seu próprio esquema de dados e podem utilizar outros mediadores como recursos de dados, formando uma rede de mediadores. Nesta arquitetura os *wrappers* implementam a heterogeneidade ao modelo, pois a forma como os dados são acessados é transparente ao mediador.

Quando os mediadores são implementados na topologia *cliente-servidor*, a camada DBMS tem a função de servidor e os *wrappers* são os clientes. Na topologia *peer to peer*, caracterizada no PMS, cada componente interage com os demais de forma descentralizada, não existindo um esquema global com informações e uma coordenação sobre todos os *peers*, e como conseqüência:

- Não existe *peer* com conhecimento global sobre os demais.
- Sendo os mediadores totalmente autônomos, poderão existir vários mediadores definindo bases de dados virtuais sobre o mesmo recurso de dados.
- A única forma de um *peer* ter acesso aos dados ou metadados é solicitando (normalmente por *querys*) a *peers* conhecidos.

A vantagem dos mediadores da topologia *peer to peer* é de possibilitar um controle próprio independente dos outros mediadores e da

mesma forma que os donos dos recursos de dados têm total controle sobre seu conteúdo.

4.4 OMG/CORBA

Os Objetos CORBA (Common Object Request Broker Architecture) compõem um poderoso *middleware* que pode estar distribuído em qualquer parte de uma rede. São pacotes binários que clientes remotos podem acessar via invocação de métodos. Esta camada de software permite a comunicação entre aplicações cliente-servidor heterogêneas. Por exemplo, um programa C++ pode usar o CORBA para acessar um serviço de banco de dados escrito em Cobol. Já a RMI é uma tecnologia Java nativa, por isso requer que todas as aplicações sejam escritas em Java. As linguagens e compiladores usados para criar os objetos servidores são totalmente transparentes aos clientes. Os clientes não precisam saber onde estão distribuídos (localizados) e qual o sistema operacional onde estes são executados. O CORBA utiliza Interface Definition Language (IDL) para fazer a interface com os potenciais clientes. As IDLs do CORBA não provêem detalhes de implementação, sendo puramente declarativas [ORF98]. As especificações de métodos das IDLs podem ser escritas e invocadas por qualquer linguagem que possua mapeamento (*mappings)* CORBA, tais como C, C++, Java, Cobol, Ada, Smalltalk. As especificações do CORBA são de responsabilidade dos membros da OMG (Object Management Group).

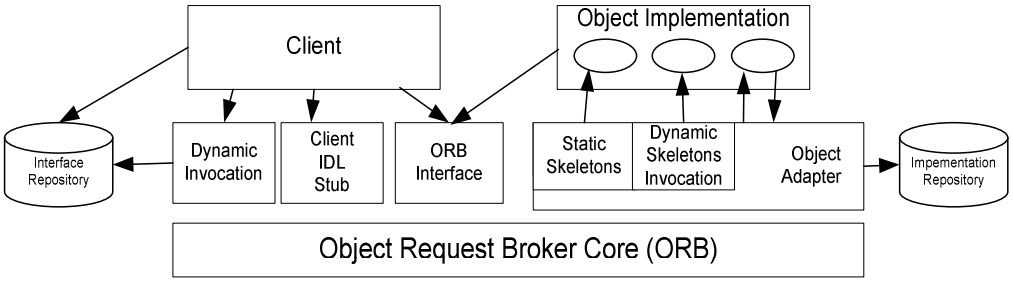

Figura 4 - Arquitetura do CORBA
Fonte Orfali et al [ORF98]

O Object Request Broker (ORB) ilustrado na figura 4, tem a função de intermediar as solicitações e respostas de outros objetos alocados localmente ou remotamente. Esta comunicação é proporcionada pelas IDLs alocadas junto ao objeto cliente e objeto servidor. Os *Stubs de IDL* do cliente provêem uma interface estática para os serviços dos objetos e definem como os clientes invocam os serviços correspondentes no servidor. Na perspectiva dos clientes, os *stubs* atuam como uma chamada local. As interfaces são definidas usando IDL, e ambos os *stubs* do cliente e servidor são gerados por um compilador IDL. Os stubs efetuam o *marshaling*, que é a codificação e decodificação das mensagens que são trocadas entre o cliente e servidor, além de invocar métodos no servidor para linguagens como C, C++, Java e Smalltalk para obter serviços remotos.

O ORB apresenta os seguintes benefícios:

- Invocação de métodos de forma estática, definindo o seu método em tempo de compilação ou dinamicamente, descobrindo-os em tempo de execução;

- Auto-descrição do sistema em tempo real, provendo uma base de metadados, com a descrição do todas as interfaces servidoras conhecidas no sistema com seus parâmetros. Os clientes usam os metadados para descobrir como invocar os serviços em tempo de execução;

- Transparência para tratar objetos localizados em processos locais, assim como vários processos executando em diferentes máquinas distribuídas em redes e sistemas operacionais diferentes;

- Possui mecanismos de segurança e transações no encaminhamento das mensagens pelos processos;

• Possui a característica de polimorfismo, podendo invocar uma função remota definindo seu método. Uma mesma chamada de função pode ter resultados diferentes, dependendo do objeto que a recebe;

• Coexistência de sistemas, por separar a definição dos objetos de sua implementação, permitindo o encapsulamento de aplicações já existentes.

O CORBA ORB provê uma grande variedade de serviços de *middleware* que são divididos nas categorias *CORBAservices* e *CORBAfacilities*, e estes se relacionam com os *objetos da aplicação*.

CORBAfacilities são um conjunto de definições em IDL que provêem serviços de uso direto para o relacionamento de colaboração entre objetos.

Serviços CORBA (CORBAservices)

Os *CORBAservices* são uma coleção de dezesseis serviços, que são disponibilizados através de interfaces IDL e complementam as funcionalidades do ORB:

• Serviço de Ciclo de Vida: define operações de criação, cópia, movimentação e exclusão de componentes do ambiente.

• Serviço de Persistência: provê uma interface para armazenar componentes persistentes em uma variedade de sistemas de armazenamento, incluindo banco de dados de objetos, banco de dados relacionais e arquivos simples.

• Serviço de Nomes: permite tratar componentes por seus nomes, utilizando diretórios de nomes existentes incluindo o NDS da Novell, NIS da SUN e DCE do OSF.

• Serviço de Eventos: permite registrar e tirar do registro eventos de forma dinâmica.

• Serviço de Controle de Concorrência: provê o gerenciamento de alocação de transações ou threads.

• Serviço de Transação: provê a coordenação da confirmação(*commit*) de componentes de transações recuperáveis.

• Serviço de Relacionamento: provê um meio para criar associações dinâmicas (ou *links*) entre componentes que não conhecem nada um sobre o outro. Pode ser usado para tratar qualquer tipo de *link* entre componentes, como a integridade referencial.

• Serviço de Externalização: provê padrões para a troca de dados entre os componentes através de mecanismos de *streams.*

• Serviço de Query: provê operações de *query* para objetos. É um superset do SQL baseado na especificação SQL3 do ODMG (Object Database Management Group).

• Serviço de Licenças: provê operações para medir o uso de componentes e assegurar a compensação pelo seu uso, através de pontos de controle nos componentes. Permite a cobrança por sessão, módulo, instância de criação ou por *site*.

• Serviço de Propriedades: provê operações que permitem obter as propriedades de qualquer componente. Usando este serviço, podemos associar dinamicamente as propriedades com os estados dos componentes, tais como: título, nome, data.

• Serviço de Tempo: provê interfaces de sincronização de tempo em ambientes de objetos distribuídos.

• Serviço de Segurança: provê recursos para tratar segurança em um ambiente de objetos distribuídos. Oferece autenticação, controle de listas de acesso, confidencialidade e não repudiação. Também gerencia a delegação de credenciais entre objetos.

• Serviço de Negociação (*Trader*): provê o serviço de 'páginas amarelas' para os objetos. Permite aos objetos publicar os seus serviços.

• Serviço de Conjuntos: provê interfaces para genericamente criar e manipular conjuntos.

• Serviço de Startup: permite solicitações para iniciar automaticamente um ORB quando invocado.

Com CORBA, provedores de componentes podem desenvolver seus objetos sem preocupar-se com o sistema de serviços. Então, dependendo do que o cliente necessita, o desenvolvedor ou integrador de sistemas pode misturar os componentes originais com os serviços do CORBA para criar as suas funcionalidades.

4.5 DCOM

O DCOM (Distributed Component Object Model) é uma extensão do Microsoft COM (Component Object Model) utilizado para a comunicação entre objetos em diferentes computadores, conectados à rede local (LAN), WAN ou Internet. É um modelo de programação binário e um padrão de interoperabilidade para computação, usando objetos distribuídos. Utilizado na construção de aplicações em três camadas, de forma a centralizar regras de negócio e processos, obter escalabilidade e facilitar a manutenção. O DCOM funciona de forma transparente, tanto para a aplicação cliente quanto para o servidor, que são codificadas de acordo com o COM. O DCOM permite que os componentes se comuniquem com ou sem orientação a conexão e suporta os protocolos TCP e UDP, IPX/SPX, Apple Talk e HTTP. O COM e DCOM foram iniciados pela Microsoft, não sendo mais um ambiente proprietário. Atualmente o consórcio independente ActiveX é responsável pelo gerenciamento deste padrão. Apesar de ser mais comum no ambiente Microsoft Windows, está disponível também no Unix e Apple. O DCOM é oferecido como um complexo sistema, no qual, muitas coisas similares podem ser feitas de forma diferente, todavia sua coexistência em diferentes soluções é algumas vezes impossível [TAN2002].

4.5.1 Visão Geral da Arquitetura

A arquitetura básica do DCOM permite que uma aplicação possa ser desenvolvida de maneira que automaticamente permita uma distribuição futura e com escalabilidade. Quando ocorre um aumento da demanda para a aplicação podemos aumentar a capacidade do servidor para atender esta necessidade.

Caso a aplicação estiver no padrão DCOM, podemos distribuir partes da aplicação por outros servidores, otimizando a utilização dos recursos disponíveis. Para se comunicar com um componente que não seja local, o DCOM emprega um mecanismo de comunicação inter-processos, que é totalmente transparente para a aplicação, substituindo o mecanismo de comunicação local por um protocolo de comunicação de rede. Assim como feito no COM, a aplicação cliente cria objetos através de uma chamada à função CoCreateInstance, que utiliza o SCM (Service Control Manager). O SCM no computador cliente se comunica com o SCM no computador servidor que utiliza a função CoCreateInstance para criar o servidor COM desejado.

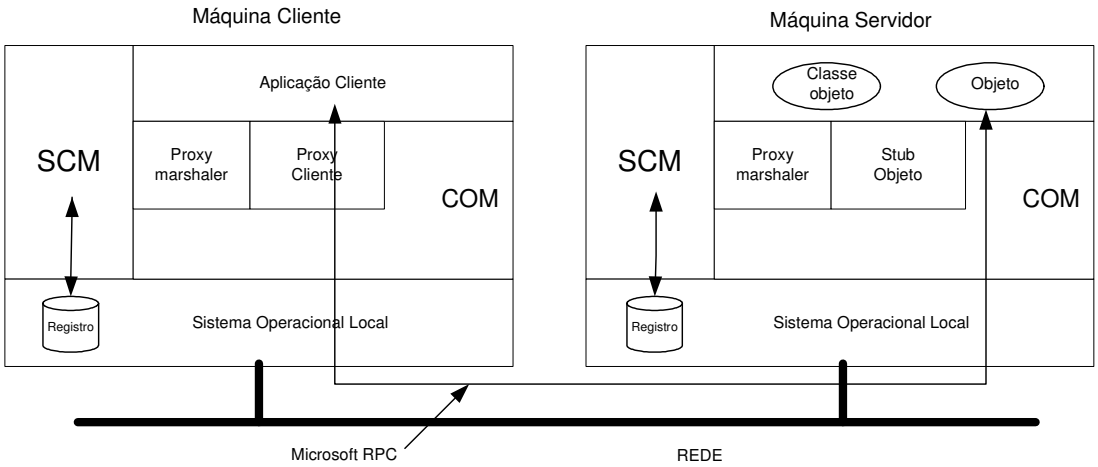

Figura 5 - Arquitetura do DCOM- Fonte Tanenbaum et al [TAN2002]

A aplicação cliente recebe um ponteiro para um objeto *proxy*, que implementa a mesma interface do servidor COM. Este é responsável pela

serialização dos parâmetros de entrada, pela utilização das funções do DCOM para comunicação com o computador servidor e pela desserialização dos parâmetros de saída.

Um objeto *stub* é responsável por desserializar os parâmetros de entrada das chamadas de métodos do servidor COM, efetuar a chamada local, serializar os parâmetros de saída e utilizar as funções do DCOM para comunicação com o computador cliente.

Localização de Objetos: Assim como no COM, cada classe possui um identificador globalmente único (GUID) de 128 bits, chamado de Class ID (CLSID). A aplicação cliente informa o GUID do objeto desejado e, opcionalmente, o endereço do servidor que tem o arquivo executável ou DLL(Dynamic Link Library) que será responsável pela sua execução. O SCM no computador cliente se conecta ao SCM no computador servidor, requisita a criação do objeto e retorna um ponteiro para um objeto *proxy* local. Caso a aplicação cliente não tenha informado o endereço do servidor, o SCM obtém essa informação no registro do Windows.

Chamada Remota de Métodos: Os nomes de instância do DCOM são referenciados como *monikers*. Os *monikers* são objetos que contém as informações necessárias para localizar a instância que está sendo executada do objeto que ele está se referindo [GOU2002]. Quando a aplicação cliente chama um método do objeto remoto, o objeto *proxy* precisa efetuar a serialização (*marshaling*) dos parâmetros, para que eles possam ser transmitidos pela rede. Os parâmetros podem ser tipos simples, mas também podem ser *arrays* ou objetos complexos, compostos de vários objetos. No servidor, um objeto *proxy* (chamado de *stub*) realiza a desserialização (*unmarshaling*) dos parâmetros, chama o método do objeto, serializa os parâmetros de saída e transmite-os para o computador cliente. No cliente, o objeto proxy desserializa o retorno do método e repassa esses dados para a aplicação cliente. O mecanismo de

serialização do DCOM foi construído a partir da infra-estrutura de RPC (Remote Procedure Call).

Gerência de Conexão: O COM realiza a liberação de memória alocada não mais utilizada (*garbage collection*) através da contagem de referências, com chamadas aos métodos AddRef e Release da interface IUnknown, implementada por todas as classes de objetos COM. Para melhorar a performance da comunicação e suportar o término anormal de aplicações cliente, DCOM utiliza *pinging* e o agrupamento de chamadas de contagem de referência.

Gerência de Concorrência: Do ponto de vista do programador, a gerência de concorrência no DCOM funciona da mesma forma que o COM. Na opção *Single-Threaded Apartment* (STA), cada método de um objeto não recebe chamadas simultâneas, ou seja, cada objeto executa em uma *thread*. No entanto, várias instâncias do objeto podem ser criadas para atender a requisições simultâneas. Na opção STA *main thread only*, todas as instâncias são criadas na mesma *thread*. Na opção *Multi-Threaded Apartment* (MTA), um método pode receber várias chamadas ao mesmo tempo e deve ser codificado com a utilização de sincronização, caso utilize algum recurso compartilhado como um campo do objeto.

Segurança: O controle de acesso, que verifica se um usuário está autorizado a executar um método, pode ser configurado de forma declarativa ou ser embutido no código. O controle de criação de objetos somente pode ser feito de forma declarativa, para evitar ataques de negação de serviço. DCOM fornece o serviço de autenticação de clientes. A autenticação pode ser configurada para funcionar com vários *security providers*, dentre eles o Windows NT LAN Manager. Na configuração padrão, a autenticação ocorre no estabelecimento da conexão. No entanto, ela pode ser configurada para ocorrer em cada chamada.

Clientes ou objetos podem requisitar que os pacotes de dados possuam informação adicional que garante integridade e criptografia dos pacotes.

4.6 Resumo de outros Projetos de Mediadores

Information Manifold [BUSSE99]: O Information Manifold é um projeto desenvolvido pela AT&T, que integra estruturas de recursos de dados na WWW. Os componentes do sistema são mapeados em um esquema de mediação que é desenhado de acordo com a necessidade de informação no topo do sistema (a estrutura é *top-down*). As características de cada componente do esquema são relatadas ao mediador através de uma linguagem declarativa. Quando o usuário executa uma *query*, o sistema utiliza descrição para identificar os recursos, executar as *sub-querys* e retornar com os resultados. Novos componentes ou objetos do sistema podem ser implementados pelo usuário. Pela independência entre o mediador e o esquema de componentes e a descrição explícita de seu relacionamento, os componentes mantém sua autonomia.

Garlic [ROT96, JOS2002]: Garlic é um projeto da IBM Reserch, que visa à integração de recursos de dados multimídia, utilizando como componentes sistemas especializados para pesquisar e armazenar os mais diversos tipos de dados. O Garlic é orientado a objeto, podendo interpretar *querys*, criar planos de execução para encaminhar parte das *querys* para os servidores de dados apropriados e retornar os resultados para as aplicações. A heterogeneidade das bases de dados é provida pela utilização de um *otimizador de query*. O Garlic utiliza *wrappers* e a execução das *querys* depende da interatividade de comunicação entre o mediador e os *wrappers*. A autonomia entre os componentes é alta.

SIMS – Search in Multiple Sources [ARE93]
O SIMS provê um ponto de acesso global para componentes de dados heterogêneos, através da uma descrição lógica do modelo de dados. A

grande expressividade da descrição lógica permite a integração de diferentes modelos de dados, dando ao modelo heterogeneidade semântica. O processamento das *querys* no SIMS considera bases de dados replicadas, onde as *querys* podem ser solicitadas mesmo sem os componentes dados não estarem disponíveis. Como no Information Manifold, o SIMS possibilita a autonomia dos componentes e a evolução do sistema. Novos componentes podem ser integrados ao sistema pela declaração de suas características ao mediador, independente dos outros componentes ou mediadores. Modificações e extensões dos mediadores podem ser feitas independentemente.

TSIMMIS – The Stanford-IBM Manager of Multiple Information Sources [LI98]

O TSIMMIS suporta a integração de recursos de dados heterogêneos, tanto de componentes estruturados ou semi-estruturados. Ao contrário dos projetos descritos acima, não possui um único esquema de mediação, mas propaga-os nos componentes *wrappers* nos usuários. O modelo de heterogeneidade é resolvido utilizando um semi-estruturado "modelo de troca de objetos", que é um modelo simples de objetos e sub objetos, sem herança e classes. Para resolver os conflitos de semântica entre os componentes foi proposto um dicionário de serviços, mas não foi implementado. As especificações de integração são feitas através da MSL (Mediator Specification Language). Os *wrappers* são robustos, pois assumem muitas tarefas que geralmente são de responsabilidade do mediador, como a decomposição das *querys* e a compensação de faltas de capabilidade das *querys* em relação aos recursos de dados.

InterDB Project [THI2002]

O InterDB Project é desenvolvido por uma equipe de pesquisadores do Departamento de Informática da Universidade de Namur-Bélgica. O modelo é composto de uma arquitetura geral, metodologia e ferramenta CASE, que provê a usuários e programadores uma interface abstrata

para bases de dados independentes, distribuídas e heterogêneas. A arquitetura é composta por uma hierarquia de mediadores, que dinamicamente transformam dados reais em uma base de dados virtual e homogênea. A independência de locação e semântica é garantida por um mediador. Cada camada de mediadores tem algum tipo de dependência e cada gerenciador de base de dados é provido de seu próprio *wrapper*. Os *wrappers* podem interagir como um *middleware* padrão, como pôr exemplo o ORB. A metodologia definida no modelo permite a recuperação dos esquemas conceituais de bases de dados existentes, permitindo a sua engenharia reversa. A ferramenta engenharia de software DB-MAIN CASE é utilizada para definir uma imagem comum e abstrata das diversas bases de dados independentes e construir os *wrappers* e mediadores, além de recursos de engenharia reversa, esquemas de mapeamento e gerador de aplicações.

IRO-DB Project – Interoperable Relational and Object-Oriented Databases [FAN98]

A arquitetura do IRO-DB é formada pelas três clássicas camadas de componentes. A *camada lógica* é composta por adaptadores de bases locais (LDA), situados nos servidores de base de dados. A base de dados local exporta esquemas relacionais e orientados a objeto com a descrição de seu tipo, características e disponibilidade. A *camada de comunicação* implementa o acesso remoto a base de dados orientada a objeto, o servidor e os clientes. A *camada de interoperabilidade* mapeia os componentes do sistema através dos esquemas importados da camada lógica, que são armazenados em um dicionário de dados. As classes de atributos e seus métodos são implementados. Uma ferramenta interativa chamada *Integrator Workbench* ajuda o administrador da base de dados a definir uma visão integrada dos diversos componentes. O IRO-DB permite a construção de federações de bases de dados em bases de dados hierárquicas e relacionais. O sistema está baseado no modelo e linguagem do padrão ODMG (Object Database Management Group).

DISCO – Distributed Information Search COmponent [TOM95]

O mediador DISCO oferece uma nova semântica para o processamento de *querys*, recursos para a modelagem de objetos de dados (Modelo de dados do Mediador) e suporta facilmente as transformações e incorporações de novos recursos de dados no mediador, além de uma interface para a construção de *wrappers*. A arquitetura do mediador DISCO é composta pela: i) *aplicação de usuários*, que acessam a representação de dados através de uma linguagem de query; ii) *mediadores* encapsulam representações de múltiplas bases de dados e permitem um acesso uniforme aos dados, além de resolver a compatibilidade do modelo de dados e representação das *querys*. Os mediadores podem ser combinados permitindo a utilização de um grande número de recursos de dados; iii) o *catálogo* (ou esquema global) relaciona as bases de dados, mediadores e *wrappers* do sistema. O catálogo não tem o conhecimento de todos os elementos, mas apresenta uma visão geral do sistema; iv) os *wrappers* são mapeados pelo mediador para acessar bases de dados em suas linguagens específicas. Durante o processamento, a *query* é otimizada pelo mediador gerando uma expressão lógica para o *wrapper,* que gera o comando apropriado para a linguagem da base de dados. Quando os recursos de dados são objetos, uma solicitação de *query* pode ser uma solicitação parcial e pode referenciar o objeto do recurso de dados ou o objeto de dados atuais, permitindo uma resposta adequada mesmo com a indisponibilidade de um determinado recurso de dados.

4.7 Conclusão

A utilização de mediadores permite o acesso a servidores de dados em n-camadas e passam a ser elementos de flexibilização e independência para a criação de interfaces de acesso às bases de dados.

Um grande número de sistemas e modelos de mediadores tem sido propostos, com variações em termos de arquitetura, níveis de distribuição, autonomia e modelo de dados. Muitos deles possuem a arquitetura centralizada, que consiste de um único componente mediador interagindo com os dados e recursos através de componentes *wrapper*. Temos como exemplos de mediadores centralizados os sistemas Garlic [ROT96] e DB2 Federeted DBMS [JOS2002]. Na arquitetura distribuída encontramos os protótipos de sistema AURORA [ABA2003], o Projeto DIOM [LEE97] e o Mediador DISCO [TOM95]. Outra característica dos mediadores diz respeito às possibilidades de acesso aos dados, onde na maioria dos projetos, permitem um acesso read-only (ex: Garlic, Information Manifold, SIMS e TSIMMIS). O FIS [BUSSE99] permite, além da leitura, a inserção de dados (read-and-write access) nos componentes da federação. Em geral, a inserção de dados implica na perda de autonomia e performance ao sistema.

Algumas linguagens de programação do mercado já incorporam recursos para acesso a vários SGDBs, essa funcionalidade induz a implementação da solução cliente-servidor para as aplicações, fazendo com que cada interface da aplicação tenha que replicar as informações de acesso aos servidores de dados [BUS2002].

Apesar dos esforços empreendidos no desenvolvimento de sistemas mediadores, não encontramos nenhum modelo de mediador que atenda todas as demandas, atuais e futuras, da interoperabilidade entre bases de dados distribuídas e heterogêneas. Concluímos que o baixo índice de utilização das soluções de mediadores existentes, se devem ao fato de que estas soluções são apresentadas como *produtos* com arquitetura proprietária, e que, para serem utilizadas, precisam ser implementados para cada ambiente operacional ou linguagem de programação.

Capítulo 5
Um Modelo para o Compartilhamento de Bases de Dados Heterogêneas e Distribuídas

Nesta seção será apresentado um modelo que viabiliza o Compartilhamento de Bases de Dados Distribuídas e Heterogêneas. A modelagem será feita na notação UML, onde serão apresentados através de seus diagramas os aspectos estruturais e comportamentais do modelo.

5.1 Justificativa

O modelo aqui proposto se apresenta como uma alternativa às propostas e tecnologias de mediadores, ou *middleware*, atualmente disponíveis. Todas as tecnologias disponíveis atualmente são implementações específicas, com restrições de sistemas operacionais e ambientes de desenvolvimento. Podemos citar como exemplo os *middleware* CORBA e DCOM, que apesar de apresentarem em suas definições uma gama enorme de recursos, apresentam restrições de implementação em ambientes operacionais heterogêneos.

O CORBA, para ser utilizado, precisa estar implementado em uma linguagem de programação. Das centenas de linguagens de programação que são utilizadas comercialmente, poucas possuem implementação do CORBA. A implementação do DCOM, até pouco tempo atrás, estava restrita aos sistemas operacionais Windows (Windows 3.11, Windows 95/98, Windows NT/2000/2003). A portabilidade do DCOM a outros ambientes operacionais está sendo possível com a aplicação da tecnologia .net™, que é licenciada pela Microsoft.

Os esforços para o desenvolvimento de sistemas mediadores, têm sido no sentido de construir produtos (pacotes) que invariavelmente precisam ser implementados em ambientes de desenvolvimento e sistemas operacionais. Nossa proposta é no sentido de desenvolver um modelo de mediador que atenda aos seguintes requisitos:

- Disponível aos pesquisadores e desenvolvedores como um modelo estrutural e comportamental, no qual possam ser incrementados novos recursos e funcionalidades;
- Possa ser implementado no maior número de ambientes operacionais;
- Não apresente restrições de ambiente de desenvolvimento;
- Possível de ser utilizado para a interoperabilidade com todos os tipos e modelos de dados;
- Possa ser facilmente portado e adaptado a novas tecnologias de ambientes operacionais, tipos e modelos de dados e modelagem de sistemas;
- Implementação do Catálogo de Agentes (que é o agente mediador do modelo) na linguagem Java, ficando o código disponível para a comunidade de pesquisadores e desenvolvedores.

A caracterização dos requisitos descritos acima, será apresentada durante a modelagem e implementação do modelo.

5.2 Modelagem através da UML

Para criar o software de uma aplicação, é necessária a descrição do problema e de seus requisitos – o que é o problema e o que o sistema deve fazer. A *análise* enfatiza a *investigação do problema*, de como uma solução é definida. O *projeto* enfatiza a *solução lógica*, ou seja, como o sistema atende os requisitos [LAR2000].

A *análise e o projeto orientados a objeto* enfatiza a consideração de um domínio de problema e uma solução lógica, segundo a perspectiva de

objetos (coisas, conceitos ou entidades). A UML (Unified Modeling Language) é uma notação para modelagem de sistemas, usando conceitos orientados a objetos. A UML é uma sintaxe geral para criar um modelo lógico de um sistema e foi projetada para ser independente de quaisquer linguagens-alvo, processo de software ou ferramenta, porém suficientemente genérica e flexível a ponto de poder ser utilizada sob uma forma personalizada, usando extensões definidas pelo próprio usuário para acomodar praticamente qualquer linguagem, ferramenta ou requisito de processo [LEE2001].

Na análise e projeto de sistemas distribuídos podemos aplicar a notação UML em todas as suas etapas. A UML possui doze tipos de diagramas, mas para efeito deste estudo, apresentaremos apenas aqueles aplicados na representação das características de distribuição e os componentes de interconexão: *diagramas de caso de uso, diagramas de classe, diagramas de colaboração e diagramas de implementação*.

5.3 Arquitetura do Modelo

O modelo de mediador aqui proposto é uma definição metodológica suportada por um framework, formado por uma comunidade de artefatos de software aqui denominados *agentes*, que cumprem duas funções: i) a função de relacionamento com o aplicativo, e ii) a função de relacionamento com os demais agentes da comunidade. A comunicação entre os agentes será feita com a utilização de *sockets*, recurso disponível em praticamente todos os sistemas operacionais utilizados comercialmente. Pretendemos focar nosso trabalho na construção da função de relacionamento dos agentes com a comunidade de agentes, que é a função fundamental de um mediador.

A comunidade de agentes é formada de maneira virtual através do reconhecimento dos agentes e seus respectivos serviços pelos outros agentes, através do Diretório de Agentes. Cada Agente Servidor se cataloga (registra) no Catálogo de Agentes, para o qual os Agentes

Cliente recorrem para identificar a disponibilidade e endereço do serviço. O Catálogo de Agentes é um objeto não persistente, existindo apenas quando o agente está em execução.

A arquitetura do modelo é distribuída (peer-to-peer), onde cada agente da comunidade interage com o aplicativo e os demais agentes, baseando-se no conhecimento das demandas do aplicativo e os serviços ofertados pelos demais agentes através doCatálogo de Serviços.

Os Agentes Cliente são construídos nos aplicativos e os Agentes Servidores são construídos junto as bases de dados ou aplicativos. As demandas do aplicativo são tratadas (programadas) no Agente Cliente e relacionadas aos serviços dos demais agentes. É necessário conhecer o serviço dos demais agentes quando o agente é integrado (construído) no aplicativo. Os serviços dos agentes são identificados através de nomes.

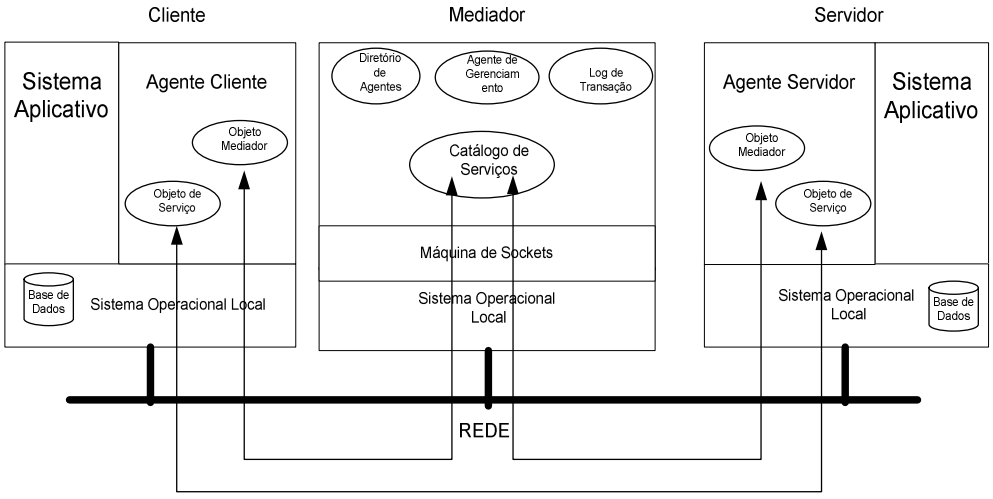

Figura 6 - Arquitetura do Modelo de Mediador

O Diretório de Agentes é uma classe persistente onde são identificados os agentes e os nomes dos serviços disponíveis na comunidade. Todos os agentes da comunidade conhecem o endereço do Diretório de Agentes, através dos parâmetros de inicialização dos

serviços, e nele devem estar inseridos junto com seus serviços, para que sejam reconhecidos pela comunidade.

Os agentes da comunidade, Clientes e Servidores, podem ser construídos e implementados em qualquer ambiente operacional que utilize *sockets*.

A administração do sistema é feita com a utilização do Agente de Gerenciamento, que tem a funções de permitir a manutenção do Diretório de Agentes e apresentar a situação de atividade dos agentes da comunidade nos seus Catálogos de Serviços Ativos e o Log de Transações gerado por cada agente. O Diretório de Agentes e o Log de Transações são os objetos persistentes do modelo. Podem ser verificados os logs de agentes inativos no endereço padrão das estações da rede. O agente de gerenciamento pode ser ativado por qualquer usuário que tenha acesso a ele. Cada comunidade de agentes tem o seu próprio agente de gerenciamento que identifica somente os agentes da comunidade através do Diretório de Agentes.

O mesmo agente (serviço) poderá estar ativo em vários pontos da rede, podendo ser usados em caso de contingência pelos outros agentes do grupo. Da mesma forma, um aplicativo poderá integrar agentes diferentes, que acessam bases de informação diferentes, para atender a mesma demanda de informação.

Para dar mais robustez ao modelo pode ser utilizada a tecnologia de agentes móveis, que dependendo da aplicação, pode agregar mais segurança e eficiência ao serviço pela interação local nas bases de dados por parte dos Agentes Servidor.

5.4 Casos de Uso

Os casos de uso descrevem as funcionalidades e os usuários (atores) do sistema. Ele é utilizado para mostrar os relacionamentos entre

os atores que empregam o sistema e os casos de uso utilizados por eles, assim como os relacionamentos entre casos de uso.

Os dois conceitos em um diagrama de caso de uso são: i) *ator* que representa usuários do sistema, incluindo pessoas, dispositivos e outros sistemas. São externos ao sistema e, ii) *caso de uso* que representa serviços ou funcionalidade provida pelo sistema aos usuários.

Um caso de uso é representado por uma figura oval. Uma descrição do caso de uso é anexada ao caso de uso sob a forma de um atributo e descreve em prosa (ou em um diagrama de seqüência) a seqüência de interações que ocorrem pela fronteira do sistema. O caso de uso se relaciona, no mínimo, a um ator por uma linha denominada relacionamento que 'comunica'. Um caso de uso pode estar relacionado com outros casos de uso. As fronteiras do sistema são ilustradas em um diagrama de caso de uso como um retângulo limite que inclui os casos de uso, com os atores fora dele. A um diagrama de casos de uso normalmente estão associados os detalhes correspondentes ao caso de uso, que podem ser complementados com diagramas de seqüência ou descrição textual.

5.4.1 Identificação dos atores

Agente Cliente - Aplicativo: sistema ou software, que utiliza um Agente Cliente para fazer o tratamento de informações em bases de dados, que podem estar em outro ambiente operacional ou em outra estrutura de dados. O Agente Cliente pode estar inserido no aplicativo ou ser executado (invocado) pelo sistema ou software (ex: objeto, DLL). As regras de relacionamento entre o aplicativo e o agente devem ser definidas e construídas considerando o ambiente operacional do aplicativo. O mesmo aplicativo pode ter diferentes agentes, que utilizam bases de dados diferentes, para atender uma mesma demanda de informação ou funcionalidade. Esta possibilidade traz redundância ao modelo, que pode ser útil como contingência.

Agente Servidor - Base de Dados: Neste modelo, são consideradas bases de dados qualquer fonte de informação disponível em ambientes computacionais ou em suas interfaces. Podem ser construídos Agentes Servidor para tratar informações de sistemas de arquivos, bancos de dados, informações de estado de sistemas operacionais, sensores, etc.

Administrador do Sistema: Utiliza o Agente de Gerenciamento da comunidade para inserir novos agentes no Diretório de Agentes e acompanhar o funcionamento do serviço através do log de transações e a atividade dos agentes através doCatálogo de Serviços.

5.4.2 Descrição de Casos de Uso

Ativação de Catálogo de Serviços: O objeto do Catálogo de Serviços pode ser ativado na inicialização dos nós da rede e através de um comando de execução. Ao ser inicialialized, o objeto do Catálogo de Serviços carrega o objeto Diretório de Agentes, identifica o endereço IP da máquina e fica escutando sua porta fixa (ex: 2323). Podem existir vários Catálogos de Serviços na mesma comunidade de agentes sendo que um mesmo Agente Servidor poderá estar catalogado em mais de um Catálogo de Serviços.

Apresentação do Serviço do Agente Servidor: Para ser utilizado pelos agentes da comunidade, um serviço deve estar registrado e ativo no Catálogo de Serviços. Os Agentes Servidores podem ser ativados na inicialização do nó da rede, por um usuário ou ainda, por Agentes Cliente que ativam seus serviços através da execução comando_de_execução do Diretório de Agentes. Na inicialização do Agente Servidor é definida a porta do Catálogo de Serviços e porta de serviço do Agente Servidor. A ativação do serviço ocorre conforme segue:

- Ao ser executado, o Agente Servidor envia a mensagem de ativação em *multicast* com seu endereço IP e porta para o Catálogo de Serviços, para disponibilizar o serviço à comunidade de agentes.

A mensagem de ativação é enviada para todos os endereços IP e para a porta do Catálogo de Serviços, obtida dos parâmetros de inicialização (ex: 2323), que contém, além do seu nome, o seu endereço IP e porta do serviço.

• Ao reconhecer a mensagem, os Catálogos de Serviços identificam o serviço no Diretório de Agentes, registram o serviço com seu endereço IP e porta e retornam seu número IP para o endereço e porta de serviço do Agente Servidor solicitante.

• Ao receber as mensagens de retorno dos Catálogos de Serviços, o Agente Servidor registra o(s) endereço(s) dos catálogos e fica aguardando mensagem de solicitação de serviços dos Agentes Cliente. Caso o Agente Servidor não receba o retorno doCatálogo de Serviços ou o agente não seja encontrado em nenhum Diretório de Agentes é registrada ocorrência no Log de Transações, que pode ser acessado Agente de Gerenciamento da comunidade.

• Periodicamente o Agente Servidor sinaliza sua atividade para os Catálogos de Serviços através de uma mensagem de atividade.

O mesmo Agente Servidor, e conseqüentemente seu serviço, pode ser ativo em vários endereços da rede, assim como pode estar registrado em mais de um Catálogo de Serviços. Estas características e funcionalidades implementam redundância e transparência de localização ao modelo.

Sinalização de Atividade: A Sinalização de Atividade tem como objetivo manter atualizada a situação de atividade do Agente Servidor. O processo de sinalização ocorre conforme segue:

• Cada Agente Servidor envia periodicamente (ex: tempo de sinalização = 5 segundos) uma mensagem sinalização de atividade para os endereços dos Catálogos de Serviços Ativos indicando sua situação (atividade). A mensagem de sinalização de atividade pode

também ser envidado pelo Agente Servidor por solicitação do Catálogo de Serviços.

• O Catálogo de Serviços recebe a mensagem a atualiza a situação do Agente Servidor como ativo. Caso o Catálogo de Serviços não receba a mensagem de sinalização de atividade dos Agentes Servidor no tempo determinado (ex: tempo de sinalização * 4), o serviço é desativado no catálogo. O Catálogo de Serviços envia então uma solicitação de atividade para o Agente Servidor e caso na recebe o retorno, registra a ocorrência no Log de Transações.

Localização do Serviço: Ao ser inicializado, o Agente Cliente recebe as informações da porta do Catálogo de Serviços e porta de serviço do Agente Cliente. Os Agentes Cliente precisam estar registrados no Diretório de Agentes, assim como o Agente Servidor do serviço que está sendo requerido.Esta verificação é feita pelo Catálogo de Serviços conforme segue:

• O Agente Cliente envia uma mensagem *multicast* para todos os endereços IP e para a porta do Catálogo de Serviços, obtida dos parâmetros de inicialização (ex: 2323), com seu nome, seu endereço IP, sua porta de serviço, e o nome do serviço requerido. O Agente Cliente fica então aguardando o retorno de um Catálogo de Serviços. Caso não obtenha a resposta em determinado tempo (ex: tempo de sinalização * 4), registra a ocorrência no Log de Transações e avisa o aplicativo da inexistência de um Diretório de Agentes na comunidade.

• Ao receber a mensagem, o Catálogo de Serviços identifica se os serviços Cliente e Servidor estão disponíveis para a comunidade no Diretório de Agentes. Caso positivo, retorna ao Agente Cliente, o endereço IP e porta de serviço do(s) Agente(s) Servidor. Caso negativo, retorna a justificativa da negativa de serviço, junto com seu endereço IP e inicializa o Agente Servidor utilizando o endereço e

comando de execução registrado no Diretório de Agentes (caso estejam cadastrados).

• Ao receber a primeira mensagem de retorno do Diretório de Agentes, indicando a porta e endereço IP do Agente Servidor do serviço, o Agente Cliente passa a ter acesso ao serviço, através da troca direta de mensagens entre o Agente Cliente e o Agentes Servidor através da porta e endereço IP obtidos. Caso o Agente Cliente não receba nenhuma mensagem indicando a porta e o endereço IP de um Agente Servidor para o serviço, retorna ao início do processo de solicitação de serviço, pois o Agente Servidor poderá já estar ativado pelo Catálogo de Serviços. Caso não obtenha êxito em determinado número de vezes (que pode ser definido na construção do Agente Cliente), registra a ocorrência no Log de Transações e avisa o aplicativo da indisponibilidade do serviço.

O Catálogo de Serviços poderá observar a situação de atividades e prioridade dos serviços registrados. A indisponibilidade de serviço deve ser tratada pelo Agente Cliente e o aplicativo. A mesma solicitação poderá ser enviada a mais de um agente (mesmo agente em outro endereço ou outro agente com o mesmo serviço).

Solicitação de Serviços do Agente Cliente: Após ter identificado o endereço IP e porta do Agente Servidor, o Agente Cliente passa transacionar com o Agente Servidor conforme segue:

• O Agente Cliente envia mensagem (request) para o endereço IP e porta do Agente Servidor, juntamente com os atributos pertinentes ao serviço.

• Caso o serviço exija retorno de mensagem contendo a resposta da solicitação ou confirmação de recebimento, o Agente Cliente fica aguardando a resposta do Agente Servidor. Caso esta resposta não retorne em tempo hábil (ex: tempo de sinalização * 20), o Agente

Cliente registra a ocorrência no Log de Transações e comunica o aplicativo. Neste caso o Agente poderá voltar a enviar a mensagem ou localizar o serviço do Agente Servidor.

O tratamento das exceções deve ser feito pelo Agente Cliente e pelo aplicativo, pois cada caso deve ser considerado no contexto da aplicação.

Atendimento de Solicitação de Serviço pelo Agente Servidor: O Agente Servidor, estando ativo, fica aguardando solicitações de serviço dos Agentes Cliente conforme segue:

- O Agente Servidor aguarda pacotes de mensagem com solicitação de serviços a ele endereçados pelos Agentes Cliente. Após o reconhecimento da mensagem, o Agente Servidor executa a solicitação, que pode ser a leitura ou atualização de informação em uma base de dados ou dispositivo a ele acoplado.

- Após a execução da solicitação, o Agente Servidor registra a tarefa em seu Log de Transações. O retorno (reply) para o Agente Cliente poderá ser necessário em função do tipo de serviço, como leitura de uma tabela ou confirmação de atualização de um arquivo. A definição do retorno da solicitação deve ser feita na Solicitação do Serviço e devendo estar prevista no Diretório de Agentes. O retorno é feito para o endereço IP e porta do Agente Cliente.

Administração do Sistema: A função de administração do sistema é implementada pelo objeto Agente de Gerenciamento, que interage com os objetos de Diretório de Agentes e Log de Transação, que são objetos persistentes, e com os Catálogos de Serviço da comunidade de agentes, conforme segue:

- A *manutenção dos serviços* permite incluir, alterar e excluir serviços no objeto Diretório de Agentes. No Diretório de Agentes são registradas informações sobre a comunidade de agentes e seus

respectivos serviços. Ele armazena informações como: identificação da comunidade, identificação e tipo dos agentes, identificação do serviço (para Agentes Servidor) serviços, endereços dos agentes na rede, comando de execução do Agente Servidor, etc. Os Diretórios de Agentes podem ser replicados em vários pontos da rede, e esta réplica é efetuada pelo Administrador do sistema.

• A *apresentação da situação atual da comunidade* é feita pela troca de mensagens do Agente de Gerenciamento com os Catálogos de Serviços da comunidade. Podem ser visualizados os diversos catálogos de serviço, sendo que a busca das informações é feita por uma mensagem broadcast enviada pelo Agente de gerenciamento para todos os endereços IP e porta dos Catálogos de Serviços Ativos, que é definida na inicialização do serviço.

• A *verificação do log de transações dos agentes da comunidade* é feita pela leitura dos arquivos Log de Transações localizados nos endereços de rede onde executam os agentes. Pode ser verificado o Log de Transações de agentes inativos, desde que exista acesso endereço de rede. Para obtenção dos Log de Transações, o Agente de Gerenciamento busca em um diretório específico em cada nó da rede (ex: \maquina_socket\log_transação.log).

Poderão também ser implementados Agentes de Gerenciamento para a verificação automática de atividade dos agentes pelo Catálogo de Serviços e Log de Transações, integrados a serviço de mensagens.

5.4.3 Diagrama de Casos de Uso

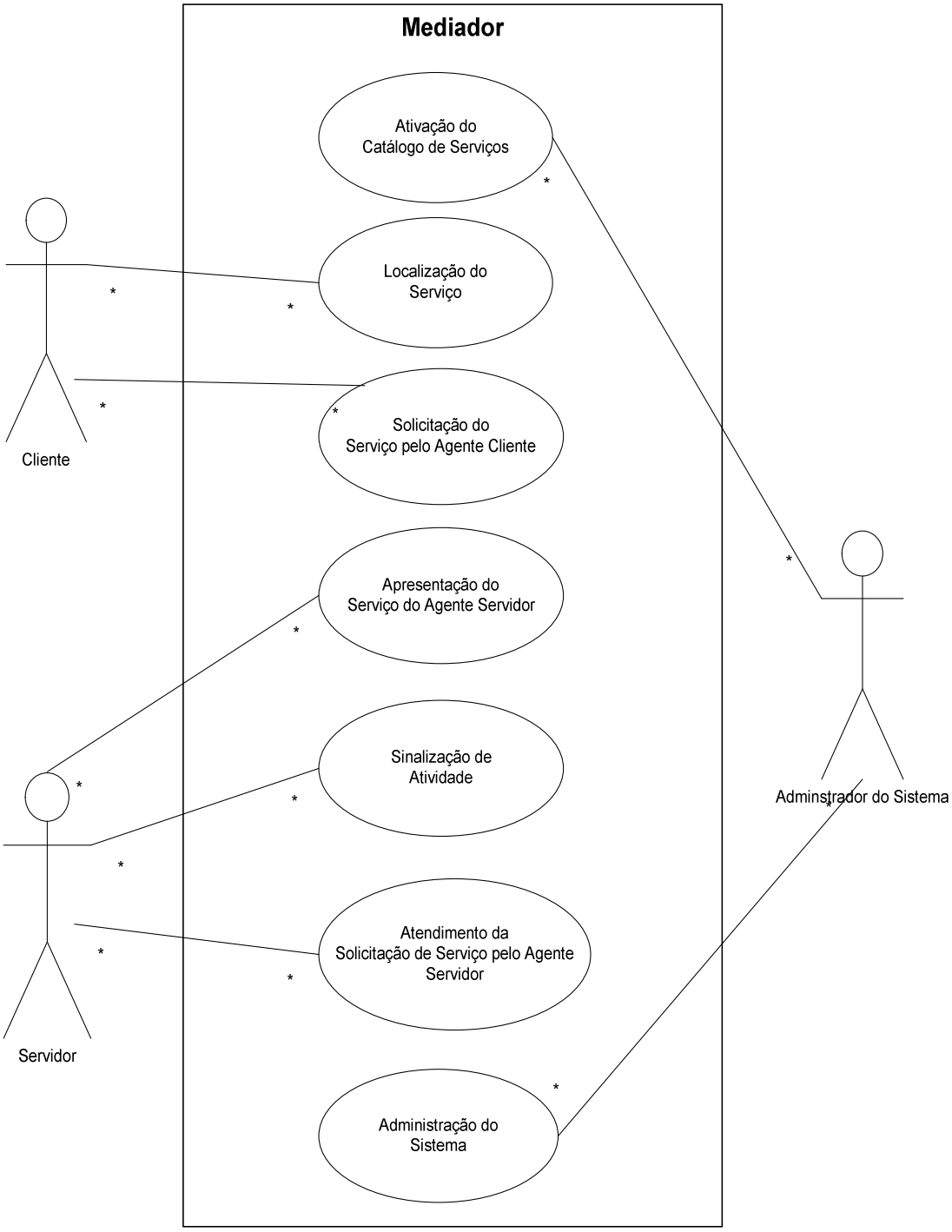

Figura 7 - Diagrama de Casos de Uso do Modelo

5.5 Projeto do Modelo

No processo de desenvolvimento de um aplicativo, após ter sido definido o comportamento do sistema, através do diagrama e descrição de casos de uso, passamos para a definição da estrutura do sistema, onde são definidas as classes e as mensagens trocadas entre os seus componentes.

Na notação UML, os *diagramas de classe* são usados para representar a *visão estática* dos padrões de interconexão [GOM]. Um *componente* é uma classe que é definida nos termos de sua interface, que é visível por outros componentes. Um *conector* esconde os detalhes da interação entre os componentes. Um recurso é usado para demonstrar a infra-estrutura de comunicação que suporta a interconexão entre os componentes e conectores.

Os *diagramas de interação* (seqüência e colaboração) são usados para representar a visão dinâmica das interações entre os componentes, conectores e objetos.

Os *diagramas de implementação* permitem representar os componentes de software e a disposição física de um sistema. Em sistemas distribuídos esta representação é fundamental para o entendimento da disposição e relacionamento de seus componentes distribuídos.

Os *diagramas de componentes* mostram as dependências de compilação e tempo de execução entre os componentes de software, tais como arquivos-fonte e DLLs.

5.5.1 Diagrama de Classes

As classes são os blocos de construção mais importantes que qualquer sistema orientado a objetos. São a descrição de um conjunto de objetos que compartilham os mesmos atributos, operações,

relacionamentos e semântica [BOO2000]. O Diagrama de Classes apresentado a seguir representa a estrutura estática do modelo.

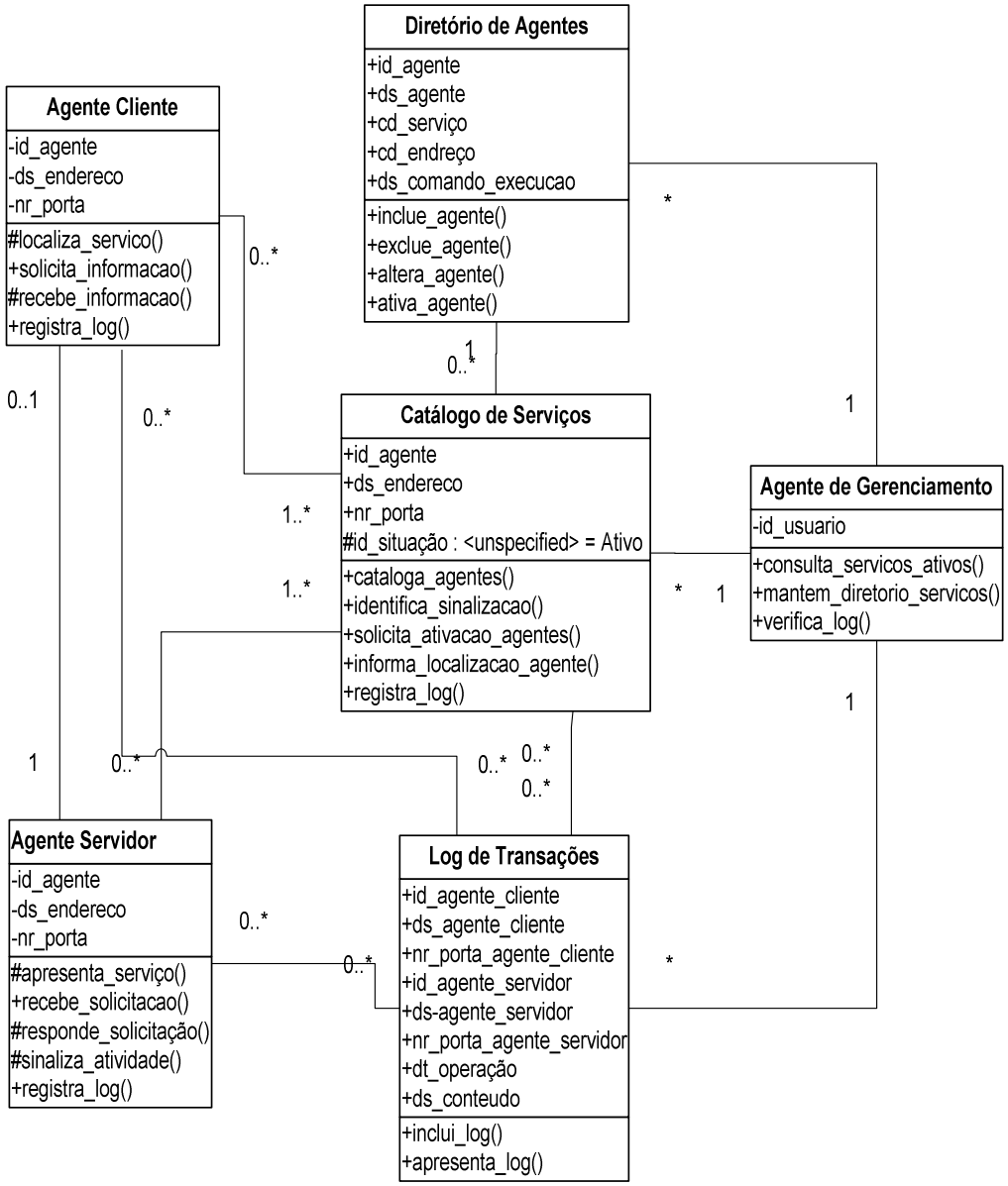

Figura 8 - Diagrama de Classes do Modelo

5.5.2 Descrição das Classes do Modelo

As classes apresentadas no diagrama acima apresentam as seguintes características e funcionalidades:

• Diretório de Agentes: É uma classe de entidade persistente, que cataloga todos os agentes e seus respectivos serviços. Pode ser replicada para mais de um ponto da sub-rede.

• Catálogo de Serviços: É uma classe de entidade não persistente, que cataloga os serviços ativos da comunidade de agentes. É utilizada pelos Agentes Servidor para registrar seus serviços e pelo Agentes Cliente para identificar os endereços dos Agentes Servidor ativos.

• Log de Transações: É uma classe de entidade persistente, onde são registradas as atividades dos agentes. Fica armazenada localmente no endereço de rede onde executa o agente. Pode ser acessada pelo Agente de Gerenciamento para fornecer informações ao Administrador do Sistema.

• Agente de Gerenciamento: É uma classe de fronteira que permite a manutenção do Diretório de Agentes e apresenta a situação dos serviços dos Catálogos de Agentes e o Log de Transações da comunidade.

• Agente Cliente: É o componente do serviço que faz solicitações de informações para outros Serviços (Agentes Servidor). O Agente Cliente é parte integrante do aplicativo, suprindo-o de informações de outras bases de dados, através do mediador. Normalmente é implementado com os recursos do ambiente de desenvolvimento do aplicativo.

• Agente Servidor: É o componente do serviço que responde as solicitações de informações dos Agentes Cliente. Os Agentes Servidor são componentes independentes que interagem com uma base de dados, disponibilizando seus serviços a comunidade de

agentes. Sua implementação pode ser feita com qualquer ferramenta de desenvolvimento que possuam recursos de acesso à base de dados. Um mesmo Agente Servidor pode oferecer mais de um serviço a comunidade de agentes.

5.5.3 Mensagens

Os objetos do modelo interagem através da troca de mensagens. As mensagens do modelo aqui proposto se dividem em duas categorias:

- *Mensagens de mediação*: são trocadas entre o Catálogo e os Agentes Cliente e Servidor e servem para viabilizar o registro dos serviços dos Agentes Servidor e a localização do serviço pelos Agentes Cliente. As mensagens de mediação são trocadas através de datagramas do protocolo UDP.

- *Mensagens de serviço*: são as mensagens trocadas entre os Agentes Servidor e Cliente, após ter sido feito o registro do serviço por parte do Agente Servidor e o serviço ter sido localizado pelos Agentes Cliente. Estas mensagens implementam o serviço propriamente dito e são trocadas após o estabelecimento de uma conexão através de *socket*. O modelo permite a troca de qualquer tipo e formato de dado entre os agentes.

A identificação, sentido e características das mensagens que são trocadas entre os objetos do modelo, estão identificadas no diagrama de seqüência da figura 9.

5.5.4 Diagrama de Interação

Os diagramas de interação da UML são usados para representar a visão dinâmica das interações entre os componentes, conectores e objetos [GOM]. As interações podem ser representadas por dois tipos de diagrama: i) o *diagrama de seqüência* representa a troca de mensagens

entre os componentes de forma seqüencial, onde o seqüenciamento das interações é definido pela ordem em que são apresentados (de cima para baixo e da esquerda para a direita); ii) o *diagrama de colaboração* representa a interação entre os objetos sem apresentar o seu seqüenciamento. No diagrama de colaboração, as mensagens podem ser numeradas para demonstrar a seqüência das ocorrências.

Na figura 9, é apresentado o diagrama de seqüência do modelo.

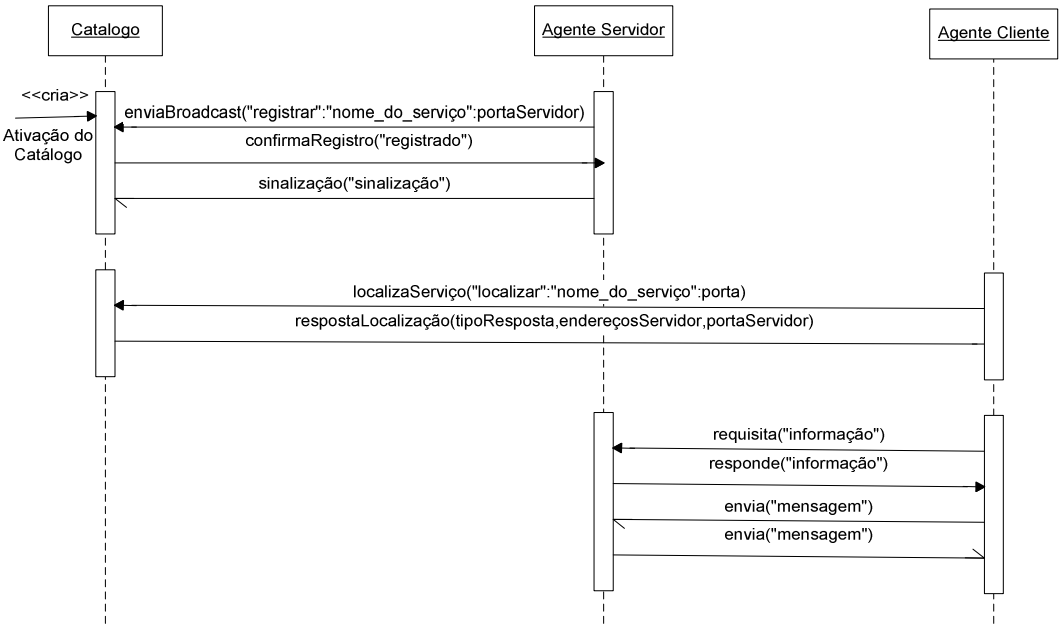

Figura 9 - Diagrama de Seqüência do Modelo

Os diagramas de seqüência são importantes instrumentos de modelagem de software, pois permitem a definição do fluxo e conteúdo das mensagens trocadas entre os objetos de um sistema. Facilitam também o processo de codificação de software por apresentarem de forma gráfica a interações entre os objetos.

5.6 Implementação do Modelo

O modelo aqui proposto foi implementado na linguagem de programação JAVA e a comunicação entre os seus componentes é feita através de *sockets*. A escolha de JAVA e *sockets* para a implementação do modelo é em função da portabilidade oferecida pela utilização da JVM (Java Virtual Machine) e pela disponibilidade de *sockets* nos principais sistemas operacionais utilizados comercialmente.

5.6.1 Construção dos Componentes do Modelo

Para atender os requisitos transparência de localização e redundância de serviços definidos na modelagem, foi necessária a construção de componentes de software que atendessem os requisitos definidos. Durante o processo de implementação do modelo foi necessário identificar e definir os recursos disponíveis nas ferramentas de desenvolvimento e nos ambientes operacionais. A implementação do modelo exigiu pesquisas para viabilizar as seguintes funcionalidades na linguagem de programação JAVA:

- Identificação das portas de comunicação que componentes utilizam para viabilizar a comunicação. Foi definido que seria utilizado um arquivo de configuração onde são definidas a portas de comunicação, que seria acessado por todos componentes. São definidas uma porta UDP padrão para envio de respostas ao catálogo, uma porta UDP para respostas ao agente, e uma porta TCP para comunicação entre o agentes. Por padrão foi assumido que a porta UDP de resposta ao agente e TCP de comunicação entre agentes teriam o mesmo número. Ao se registrar em um catálogo, um agente-servidor publica a porta TCP na qual está operando.
- Definição dos tipos de mensagens trocadas pelos componentes. Foi desenvolvida uma classe destinada a gerar mensagens de envio e analisar as mensagens recebidas. Esta classe utiliza o método

split, da classe String, para decompor a mensagem.

- Forma de identificação do endereço IP da máquina em que estão localizados os componentes. Para a identificação do endereço ip foi utilizado o método getHostAddress, da classe InetAddress.
- Criação de uma abstração de um canal de comunicação entre os agentes e o catálogo. Para isto, foi implementado uma classe nomeada CanalCatalogo. Esta classe encapsula a comunicação pelas portas UDP entre os agentes e o catálogo. Para a abertura do *socket* UDP foi utilizado a classe DatagramSocket, e para os envios das mensagens o método send (da própria classe DatagramSocket)
- Construção do dispositivo de localização do Catálogo pelos Agentes Servidor e Cliente. Para localizar os possíveis catálogos disponíveis na rede, os agentes clientes e agentes servidores enviam uma mensagem de registro (no caso dos servidores) ou de localização de serviço (no caso dos clientes) por UDP em broadcast.
- Definição do recurso de armazenamento dos registros dos serviços do Catálogo de Serviços. Para o armazenamento dos registros de serviços no catálogo, foi criado uma classe que implementa a abstração de uma lista. Esta é uma derivação da classe ArrayList, disponibilizada pelo Java.

Considerando que a concepção do modelo já estava definida na modelagem, foi necessária a pesquisa de componentes do JAVA e a construção de classes e métodos. Os componentes do mediador trocam *mensagens de mediação* através de pacotes UDP, que permite o envio de mensagens *multicast*. A comunicação entre o Agente Cliente e Agente Servidor, ou *mensagens de serviço*, pode ser feita pela troca de qualquer tipo e formato mensagem.

Nesta etapa da implementação foram demandados esforços para a construção dos mecanismos de identificação e localização de componentes do modelo. A seguir, iremos apresentar a implementação

da comunicação entre os Agentes Cliente e Servidor. Um dos requisitos definidos para o modelo aqui proposto é do isolamento do mecanismo de mediação e de comunicação entre o Catálogo de Agentes e os Agentes Cliente e Servidor da interação entre os Agentes Servidores aos Agentes Clientes. A separação entre os serviços de mediação e comunicação do mediador e da comunicação entre Agentes Clientes e Servidores é necessária, pois a implementação dos últimos é específica para cada serviço, sendo que cada serviço que vier a ser implementado pode apresentar características específicas e estar localizado em ambientes operacionais distintos. Esta separação é obtida pela utilização de interfaces do JAVA. As interfaces são empregadas para visualizar, especificar, construir e documentar a coesão interna do sistema [BRO2000]. A utilização das interfaces permite o encapsulamento das classes e métodos do mediador, permitindo aos Agentes Cliente e Servidor ter acesso apenas a classes e métodos definidos nas interfaces.

5.6.2 Comunicação dos Agentes com o Mediador

A comunicação entre os agentes e o mediador deve ser feita através de pacotes denominados mensagens, aqui definidas como *mensagens de mediação*. As mensagens de mediação implementadas até o momento são as seguintes:

O Agente Servidor envia ao Catálogo as seguintes mensagens:

registrar_serviço("REGISTRAR", "nome_do_serviço", porta_do_servidor)

sinalização("SINALIZACAO")

O Catálogo envia ao Agente Servidor a seguinte mensagem UDP:

confirma_registrado("REGISTRADO")

O Agente Cliente envia ao Catálogo a seguinte mensagem UDP:

localiza_serviço("LOCALIZAR", "nome_do_serviço", porta_do_cliente)

O Catálogo envia ao Agente Cliente as seguintes mensagens UDP:

resposta_localização("tipo de resposta", endereço do servidor, porta do servido)

Nota: Os *tipos de resposta* podem ser: *"LOCALIZADO" e "NÃO LOCALIZADO"*

Temos até o momento a implementação das funções básicas do modelo, sendo que na sua evolução novas funcionalidades poderão ser incrementadas e novas mensagens poderão ser inseridas ao protocolo do modelo.

5.6.3 Comunicação entre os Agentes

Para viabilizar a comunicação entre o Agente Cliente e Agente Servidor do modelo são utilizadas as *mensagens de serviço*. Deve ser estabelecido um protocolo para padronizar a troca de mensagens entre os Agentes Cliente e Servidor. A partir da conexão dos Agentes Cliente ao Agente Servidor, poderão ser trocados todos os tipos de mensagens, sendo que estes pacotes de informação podem conter *strings* de dados, documentos XML (Extensible Markup Language), instruções contendo invocação de métodos, comandos de execução de DLLs e APIs.

Os Agentes Cliente e Servidor podem trocar *mensagens síncronas*, onde Agente Cliente solicita informação para o Agentes Servidor com a mensagem *requisita("informação")* e o Agente Servidor retorna informação para o Agentes Cliente com a mensagem *responde("informação")*. Podem também ser utilizadas *mensagens assíncronas*, onde tanto a Agentes Cliente quanto o Agentes Servidor enviam mensagens um para o outro sem aguardar retorno, ou seja, sem bloquear a execução dos próximos comandos. Neste modelo a mensagem *envia(mensagem)* é assíncrona.

O conteúdo dos pacotes de informação e mensagens, o sincronismo na troca de mensagens e o tratamento das exceções na interação entre os agentes, devem ser definidas na implementação dos Agentes Cliente e Servidor.

5.6.4 Implementação do JAVA com Códigos Nativos

A função principal do mediador aqui proposto é de prover a interoperabilidade entre repositórios de dados distribuídos e

heterogêneos. É comum encontrarmos ambientes operacionais com diversos repositórios de dados sendo acessados por diversas linguagens de programação. Existem situações que inviabilizam a implementação de Agentes Cliente e Servidor em JAVA, como por exemplo: i) a existência de grande quantidades de componentes já implementados, que tornam inviável que estes componentes sejam portados para a linguagem JAVA, em função do tempo desenvolvimento e dos custos de implementação; ii) impossibilidade de implementação em JAVA para sistemas operacionais específicos; iii) necessidade de otimizar tempo de resposta dos componentes, que quando compilados apresentam melhor desempenho que o JAVA, que é interpretado.

A implementação dos Agentes Cliente e Servidor poderá ser feita em qualquer linguagem de programação e sistema operacional que implemente socktes devendo apenas ser respeitados o formato e protocolo definidos na comunicação com o Catálogo (mediador).

5.7 Conclusão

Com a implementação do modelo, com seus componentes Catálogo de Serviços, Agentes Servidor e Agente Cliente em JAVA, podemos confirmar a viabilidade técnica e operacional do modelo aqui proposto. Conseguimos até esta etapa validar as seguintes características:

- Independência de Ambiente Operacional (Portabilidade): necessita apenas de uma JVM (Java Virtual Machine). As diferenças de implementação de *sockets* para os diversos sistemas operacionais são resolvidas pela JVM.
- Independência física de dados: O encapsulmento das funções específicas dos agentes, como acesso as bases de dados, ficam isoladas dos demais agentes.
- Transparência de Localização: o Catálogo de Agentes, Agente Cliente e Agente Servidor interagem de qualquer ponto da sub-

rede. Estes componentes podem ser transferidos de local (máquina) sem afetar o funcionamento do modelo.

- Redundância dos Serviços: tanto o Catálogo de Agentes quanto os Agentes Servidor podem estar executando em mais de um posto da sub-rede.
- Gerenciamento do Ambiente: o Agente de Gerenciamento pode verificar atividade dos diversos Catálogos de Agentes ativos na sub-rede, assim como acessar os Logs de Transação dos Agentes Cliente e Servidor.

Os requisitos definidos no modelo foram viabilizados na implementação do modelo. Foram feitas diversas simulações e conseguimos obter resultados satisfatórios, tanto em execuções dos componentes em um mesmo computador, quanto em execuções de componentes distribuídos em vários computadores de uma sub-rede. Nas duas situações o funcionamento do modelo, ou comportamento, assim como desempenho, foram equivalentes.

A próxima etapa de nosso estudo avaliar o desempenho do modelo em outros ambientes de desenvolvimento e sistemas operacionais. Esta implementação será feita através de um estudo de caso, apresentado no próximo capítulo.

Capítulo 6
Estudo de Caso: Localização de Recurso Técnico

A implementação de novos sistemas ou o incremento de funcionalidades nos sistemas das organizações, na maioria dos casos, exige que sejam feitas integrações para acesso às bases de dados de sistemas já existentes. O objetivo deste estudo de caso é avaliar o esforço de implementação e a viabilidade operacional do modelo proposto, em ambientes operacionais heterogêneos.

6.1 Definição do Problema

A situação descrita a seguir apresenta a necessidade de integração entre dois sistemas, que neste caso, exige uma interação em tempo real entre duas bases de dados distintas, localizadas em ambientes operacionais distintos.

O Sistema de Gerenciamento de Manutenção (SGM) de uma planta industrial precisa obter informações da sobre a localização de técnicos, a fim de agilizar o atendimento de solicitações de serviço de manutenção corretiva. As informações sobre a localização dos técnicos encontram-se no Sistema de Controle de Acesso (SCA), que é composto por uma rede de leitores de códigos de barras instalados junto às portas de acesso das diversas áreas da planta. As pessoas, funcionários ou não, precisam se identificar através de um cartão (crachá), para ter acesso às áreas.

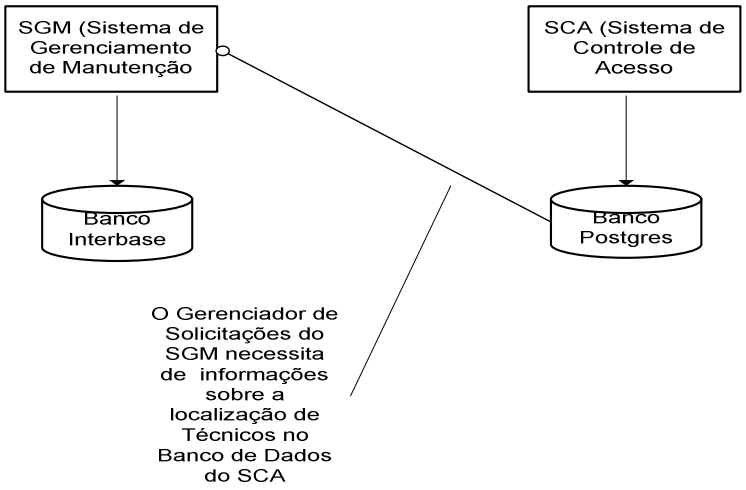

Figura 10 – Sistemas SGM e SCA

Por questões de segurança, o acesso às áreas da planta é exclusivo a determinados profissionais. As informações referentes ao acesso dos profissionais às áreas alimentam uma base de dados que permite a localização das pessoas em "tempo real". O incremento desta funcionalidade ao SGM trará uma redução significativa nos riscos de acidente e de prejuízo econômico para a organização.

6.2 Definição do processo de Localização de Recursos Técnicos

Na modelagem de sistemas de informação, devem ser definidos os processos e o fluxo de controle das suas operações. Para atender a esta necessidade utilizamos os Diagramas de Atividade da UML. Os Diagramas de Atividade permitem visualizar, especificar, construir e documentar a dinâmica de uma sociedade de objetos, ou poderão ser utilizados para fazer a modelagem do fluxo de controle de uma operação [BOO2000].

O processo de Localização dos Recursos Técnicos está representado no Diagrama de Atividade da figura 11.

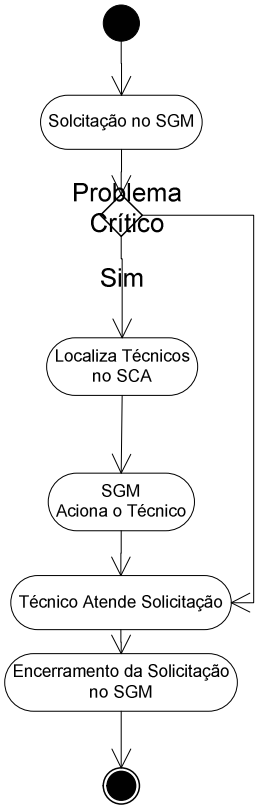

Figura 11 – Diagrama de Atividades – Localização de Recursos Técnicos

Para uma melhor compreensão do diagrama acima, segue a descrição das atividades por ele representadas:

1) As solicitações de serviço, que são abertas no Sistema de Gerenciamento de Manutenção, identificam, além de tipo de problema, a localização dos equipamentos na planta.

2) Caso seja um problema ou equipamento crítico, o SGM, através de seu Gerenciador de Solicitações, identifica o(s) técnico(s) apto(s) a atender a ocorrência. A partir disto, o Gerenciador de Solicitações precisa saber a localização dos técnicos selecionados, sendo que a localização dos técnicos encontra-se no SCA. A solicitação desta informação deve ser feita através do mediador, pois os dois sistemas, e conseqüentemente suas bases de dados, encontram-se em ambientes operacionais distintos.

3) O SGM faz a solicitação de localização ao SCA, através do envio de uma mensagem com a lista dos técnicos selecionados, que recebe e processa a solicitação, retornando uma mensagem com a localização que cada técnico da lista.

4) Com base na localização dos técnicos, o SGM identifica o profissional mais próximo e o aciona através de um sistema de mensagens (bip ou telefone celular).

5) Após ser acionado, o técnico deve efetuar o aceite da solicitação, sendo que a partir deste momento, a Solicitação de Serviço passa para a situação de atendimento.

6) No final da intervenção, o técnico deve encerrar a Solicitação de Serviço no SGM.

6.3 Implementação do Agente Servidor

O Agente Servidor, que atende as solicitações de informação da base de dados Postgres do SCA, foi implementado na linguagem de programação Visual Basic. Este componente interage com os demais componentes do mediador através de mensagens, conforme demonstrado no diagrama da figura 12.

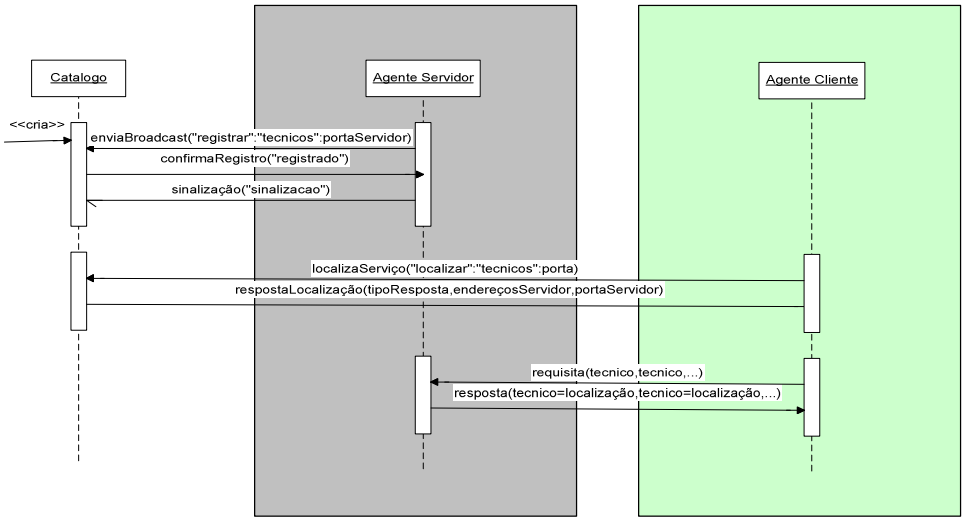

Figura 12 - Diagrama de Seqüência dos Agentes Cliente e Servidor

Para viabilizar a implementação do Agente Servidor, foi necessário apenas construir o componente de software, respeitando a seqüência definida no diagrama da figura 12, o que caracteriza as propriedades de independência de ambiente e de encapsulamento dos componentes do modelo.

Para facilitar a padronização da comunicação e interoperabilidade entre os futuros agentes da comunidade, as informações trocadas entre os agentes Cliente e Servidor foram armazenadas em documentos XML. A estrutura dos documentos XML é mostrada nos quadros 2 e 3.

```
<resposta>
        <tecnico>
        </tecnico>
        <localizacao>
        </localizacao>
        <tecnico>
        </tecnico>
        <localizacao>
        </localizacao>
</resposta>
```

Quadro 2 – Estrutura do documento XML da resposta do Agente Servidor

A utilização da UML para a modelagem do mediador facilitou o processo de desenvolvimento dos Agentes Servidor e Cliente, pela capacidade de representação das características comportamentais e estruturais oferecidas pelos seus diversos diagramas.

O Visual Basic, assim como a maioria das linguagens de programação, apresenta uma biblioteca de componentes que viabilizam a implementação de vários tipos de protocolos de comunicação. Para a construção das funcionalidades de comunicação do Agente Servidor foi necessária a utilização do componente Microsoft Winsock Control 6.0.

6.4 Implementação do Agente Cliente

O Agente Cliente foi implementado na linguagem de programação JAVA e interage com a base de dados Interbase/Firebird, buscado informações sobre a localização dos técnicos junto ao Agente Servidor. Considerando que a Agente Cliente foi implementado na linguagem de programação JAVA, não tivemos maiores dificuldades em construir os componentes de mediação, pois foram utilizadas as mesmas bibliotecas utilizadas para a implementação do Catálogo de Serviços.

```
<requisita>
        <tecnico>
        </tecnico>
        <tecnico>
        </tecnico>
</requisita>
```

Quadro 3 – Estrutura do documento XML da requisição do Agente Cliente

6.5 Descrição do Ambiente de Simulação

A simulação do serviço de Localização de Recursos Técnicos apresentado acima foi feita em dois ambientes operacionais distintos, tanto a nível gerenciadores de banco de dados, quanto ao nível de sistemas operacionais. Por se tratar de uma simulação, as informações das bases de dados utilizadas pelos dois sistemas foram inseridas (populadas) diretamente com as ferramentas disponíveis nos gerenciadores dos bancos de dados Interbase/Firebird e Postgres.

A representação do ambiente dos dois sistemas está representada na figura 13:

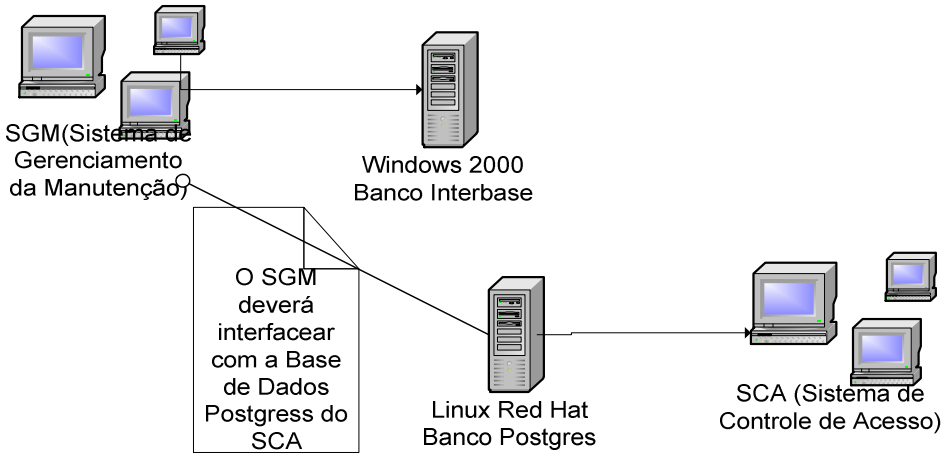

Figura 13 - Diagrama de Implantação do SGM e SCA.

O Sistema de Manutenção Industrial (SGM) executa no seguinte ambiente operacional:

- Plataforma de desenvolvimento JAVA, utilizando o banco de dados Interbase/FireBird.
- Sistema operacional Linux Red Hat.

Por sua vez, o ambiente operacional do Sistema de Controle de Acesso (SCA) apresenta as seguintes características:

- Plataforma de desenvolvimento MS Visual Basic com, implementação em banco de dados Postgres.
- Sistema operacional Windows 2000.

A simulação do serviço foi feita em uma rede TCP-IP, em estações com processadores Pentium 2.3 Mhz e 256 Mb de memória, operando a velocidade de comunicação de 10 e 100 mega bits por segundo.

6.6 Conclusão

A simulação do estudo de caso foi feita na rede da GATI – Gestão e Assessoria em Tecnologia da Informação. Com base no diagrama de atividade da figura 12, o programador Jefferson Thiago Leitholdt implementou o Agente servidor na linguagem de programação Visual Basic, acessando o banco de dados Postgres. O tempo de implementação do Agente Servidor, com o tratamento das mensagens de mediação e de serviço, foi de aproximadamente 35 minutos.

O modelo do mediador implementado em Java já estava instalado na rede. Considerando que o tratamento das mensagens de mediação já estava implementado no Agente Cliente, foi necessário apenas implementar as mensagens de serviço e acesso ao banco de dados Interbase/Firebird, atividade que demorou aproximadamente 15 minutos e foi desenvolvida pelo analista de sistemas Diogo Vinícius Wink. Não houve a necessidade de fazer qualquer modificação no componente Catálogo de Serviços.

Com a implementação e simulação do estudo de caso, conseguimos comprovar a viabilidade técnica e operacional do modelo proposto, evidenciando sua flexibilidade e facilidade de implementação.

Capítulo 7
Conclusão e Trabalhos Futuros

7.1 Aplicabilidade do Modelo

A proposta deste trabalho foi de apresentar as principais ferramentas e recursos atualmente disponíveis para viabilizar a interoperabilidade de aplicações e bases de dados distintas em ambientes distribuídos e heterogêneos, e sugerir um modelo de mediador que atenda a praticamente todas as demandas relacionadas a esta questão. Nossa constatação foi de que, apesar da existência de recursos a nível dos sistemas operacionais como *sockets* e RPC e mediadores como CORBA e DCOM, sua utilização é muito pequena e sem qualquer padrão metodológico. Na grande maioria dos casos são utilizadas aplicações específicas para cada caso, implicando em altos custos de implementação e manutenção das integrações. O esforço e a complexidade aumentam consideravelmente quando as bases de dados estão localizadas em sistemas operacionais distintos.

A interoperabilidade entre as bases de dados do estudo de caso apresentado, não poderia ser viabilizada com utilização dos modelos de mediadores disponíveis em seus ambientes operacionais. Pelas características de suas linguagens de programação e sistemas operacionais, seria necessário o desenvolvimento de aplicativos específicos nos dois ambientes. Com a aplicação do mediador proposto, o esforço de implementação foi mínimo e sem a necessidade da aquisição ou implementação de qualquer outro recurso ou ferramenta.

A simplicidade e facilidade de implementação, viabilizadas pela modelagem em UML, aliadas à utilização de *sockets* como mecanismos de comunicação, fazem do modelo de mediador proposto uma alternativa que permite a interoperabilidade de bases de dados de dados

heterogêneas, em ambientes operacionais heterogêneos, de forma transparente e padronizada. As simulações feitas na implementação do estudo de caso apresentaram as seguintes vantagens:

- Viabilidade Operacional: o modelo mostrou-se viável operacionalmente, por atender todas a demandas de interoperabilidade das aplicações e utilizando os recursos disponíveis os dois ambientes operacionais.
- Transparência de Localização: funcionamento do mediador, assim como dos agentes Cliente e Servidor, não foi comprometido pela mudança de sua localização física nos nós da sub-rede.
- Expansibilidade: a inserção de novos agentes na comunidade, não afeta o funcionamento do mediador, possibilitando a utilização dos serviços já implementados pelo Agente Servidor por qualquer Agente Cliente, independente de sua implementação ou localização.
- Estabilidade e Desempenho: pelo fato de se utilizar *sockets* como infra-estrutura de comunicação entre os nós da rede, observamos estabilidade e um excelente desempenho do mediador, mesmo submetido a um grande volume de transações.
- Esforço de Implementação: considerando que o Catálogo de Serviços já estava implementado em JAVA, o esforço de implementação dos Agentes Cliente e Servidor foi mínimo. A apresentação do padrão de comunicação entre os componentes no diagrama de seqüência da UML facilita a compreensão do modelo pelos programadores.

Apesar do desenvolvimento de um grande número de propostas que tem como objetivo solucionar questões relacionadas à integração de bases de dados heterogêneas e distribuídas, não encontramos soluções que atendam todas as demandas geradas nestes ambientes. A aplicação de mediadores, nas suas mais variadas arquiteturas, apresenta-se como uma alternativa viável, mas se considerarmos a variedade de ambientes operacionais e bases de dados, sua implementação torna-se complexa e

dispendiosa. Na prática, onde já existe um ambiente em que os recursos de informação são homogêneos e distribuídos, esta integração é feita de forma específica para cada caso, através do desenvolvimento de aplicativos *ad hoc,* que permitem o acesso e compartilhamento das informações. O desenvolvimento de aplicativos na Internet vem criando uma demanda crescente de recursos e metodologias para a integração de bases de dados. Em função desta realidade, consideramos ser oportuno o desenvolvimento de padrões que tragam suporte metodológico a esta árdua tarefa. Nossa contribuição foi a definição de um modelo, definido na notação UML, que permite a padronização, transparência e agilidade no desenvolvimento e manutenção de sistemas que acessam bases de dados distribuídas e heterogêneas.

7.2 Dificuldades Encontradas

Inicialmente pesquisamos as características e recursos disponíveis na infra-estrutura dos sistemas operacionais, ferramentas de desenvolvimento de aplicativos e gerenciadores de bancos de dados.

Em seguida pesquisamos as principais propostas e ferramentas de *midlleware*, onde em alguns casos, encontramos um vasto material bibliográfico de cunho eminentemente teórico, mas muito pouco material que tratasse dos aspectos práticos de implementação. Constamos que a maioria dos projetos de mediadores, nunca foram sequer implementados e os poucos que foram implementados apresentam um baixo índice de utilização pelos desenvolvedores e integradores de sistemas.

O próximo passo foi buscar uma alternativa às propostas de mediadores que viabilizasse a interoperabilidade de bases de dados, independentemente do sistema operacional e sem a necessidade de aquisição ou implementação de qualquer outro recurso. Após a definição do protótipo do modelo, tratamos de sua implementação. Foram feitas pesquisas no sentido de identificar uma ferramenta de desenvolvimento que permitisse sua implementação, independente de sistema operacional

ou base de dados. Encontramos no JAVA uma alternativa para viabilizar o mediador.

A implementação do modelo de mediador em JAVA, exigiu um esforço na pesquisa dos componentes que viabilizassem a identificação do endereço IP, envio de mensagens *multicast* e a comunicação através de *sockets*. Para a implementação do estudo de caso, foi necessária também identificação destes componentes na linguagem de programação Visual Basic.

Finalmente, tivemos que montar um ambiente para simular o estudo de caso, onde além dos objetos do mediador, foi necessária a utilização dos bancos de dados Interbase/FireBird e Postgres e os sistemas operacionais Linux Red Hat e Windows 2000.

7.3 Trabalhos Futuros

Considerando que o objetivo deste trabalho foi buscar soluções para resolver a problemática da interoperabilidade entre bases de dados distribuídas e heterogêneas, nosso foco principal foi de apresentar um modelo de mediador que contemple as demandas não atendidas pelas propostas e soluções de mediadores existentes. Para efeito desta dissertação, foram modelados e implementados os principais componentes do mediador proposto e consideramos que os objetivos inicialmente propostos foram alcançados. Na evolução de nossas pesquisas poderão ser considerados os seguintes pontos:

- Implementação do Agente de Gerenciamento do modelo proposto.
- Implementação da funcionalidade de ativação dos Agentes Servidor pelo Catálogo de Serviços.
- Desenvolvimento de interfaces para a especificação do serviço e representação do formato das mensagens, trocadas entre os Agentes Cliente e Agentes Servidor.

• Desenvolvimento de um estudo comparativo, onde poderão ser comparadas as vantagens e desvantagens do modelo de mediador proposto e os demais mediadores existentes.

• Implementação das mensagens de serviço em arquivos XML.

Referências Bibliográficas

[ABA2003] D. J. Abadi, et al, Aurora: a new model and architecture for data stream management, Brandeis University – Waltham, Brown University – Providence, M.I.T. – Cambridge, USA, ACM Computing Surveys, 2003

[ARE93] Y. Arens, C. Knoblock, SIMS: Retrieving and Integrating Information From Multiples Sources, USC/Information Sciences Institute, USA, ACM Computing Surveys, 1993

[BOO2000] Booch, G Rumbaugh J. Jacobson I. UML – Guia do Usuário, Editora Campus, 2000

[BRO2000] Brookshear, J. Gleen, Ciência da Computação – Uma Visão Abrangente, Bookman, 2000

[BUS2002] G. Busichia e J.E. Ferreira, Compartilhamento de Módulos de bases de Dados Heterogêneas através de Objetos Integradores. ICMC-USP - São Carlos e IME-DCC-USP-São Paulo, 2002

[BUSSE99] S. Busse, R. Kutsche, U. Leser, H. Weber, Federated Information Systems: Concepts, Terminology and Architectures, Tecnische Universität Berlin-Fachbereich 13 Informatik, Berlin, 1999

[COU2001] Coulouris, G. Dollimore, J. Kindberg, T. Distributed Systems: Concepts and Design. Eddison-Wesley. 2001

[DAT2000] Date, C.J. Introdução a Sistemas de Bancos de Dados. Editora Campus, 2000

[ELM2000] Elmasri, L. Navathe, S. B. Fundamentals of Database Systems. Addison-Wesley. 2000

[FAN98] P. Fankhauser, G. Gardarin, M. lopez, J. Munoz, A. Tomazic, Experiences in Federated Databases: From IRO-DB to MIRO-DB, GMD-Germany, University of Versailles – France, GIE Dyade – France, Ibermatica – pain, INRIA – France, 1998

[GOM] Gomaa, H. Menascé, Daniel A. Design and performance modeling of component interconection patterns for distributed software architectures. George Mason University

[GOU2002] Goulart, Ademir. Avaliação de mecanismos para Comunicação em Grupo em Ambiente WAN. UFSC, 2002

[JOS2002] V. Josifovski, P. Schwarz, L. Haas, E. Lin, Garlic: A New Flavor of Federated Query Processing for DB2, IBM Almaden Research Center, IBM Silicon Valley Laboratory, USA, ACM Computing Surveys, 2002

[KAT2002] T. Katchaounov, Query Processing for Peer Mediator Databases, Uppsala University, Sweden, 2003

[LAR2000] Larman, Craig. Utilizando UML e Padrões. Porto Alegre. Bookman. 2000

[LEE2001] Lee, Richard C. Tepfenhart, William M. UML e C++ - Guia Prático de Desenvolvimento Orientado a Objeto. Makron Books. São Paulo. 2001

[LEE97] Y. Lee, L. Liu, C. Pu, Towards Interoperable Heterogeneous Information Systems: An Experiment Using DION Approach, University of Alberta, Oregon Graduate Institute, USA, ACM Computing Surveys, 1997

[LEN2002] M. Lenzerini, Data Integration: A theoretical Perspective, Universitá di Roma "La Sapienza", Roma, 2002

[LI98] C. Li, et al, Capability Based Mediation in TSIMMIS, Stanford University, University of California, USA, ACM Computing Surveys, 1998

[ORF98] Orfali, R.,Harkey,D., Client/Server programming with JAVA and CORBA, Wiley, 1998

[OZS2001] Ozsu, M.T. Valduriez, P. Princípios de Sistemas de Banco de Dados Distribuídos. Editora Campus, 2001

[ROT96] M. T. Roth, et al, The Garlic Project, IBM Almaden Research Center, USA, ACM Computing Surveys, 1996

[SIL2000] Silberschatz, A. Galvin, Peter. Gagne, Greg. Sistemas Operacionais - Conceitos e Aplicações. Editora Campus, 2000

[TAN2002] Tanenbaum, A.S. Steen M. V. Distributed Systems Principles and Paradigms. Vrije Universiteit, Amsterdam, Prentice Hall. 2002

[TAN89] Tanenbaum, A.S. Bal, H. E, Steiner, J.G. Language Support for Programming Distributed Systems. ACM computing Surveys, Vol 21. 1989

[THI2002] P. Thiran, J. Hainaut, S. Bodart, A. Chougrani, A. Deflorenne, .Dumoulin, J. Hick, InterDB Project – Integration of Legacy and Heterogeneous Databases, University of Namur – Institut d`Informatique Belgium, 2002

[TOM95] A. Tomazic, L. Rashid, P. Valduriez, Scaling Heterogeneous Databases and Design of DISCO, INRIA - Institut National de Recherche en Informatique et an Automatique, France, 1995

[WID96] J. Widom, Integrating Heterogeneous Databases: Lazy or eager, Stanford University-CA, ACM Computing Surveys, 1996

Anexos

A – Parâmetros de Execução do Modelo

portaCatalogo	2000
portaServidor	2001
portaCliente	2003

B – Código Fonte JAVA do Mediador

```
package servidor;

import java.io.IOException;
import java.net.SocketException;
import java.net.SocketTimeoutException;
import java.net.UnknownHostException;

import mensagens.MensagemCatalogo;
import servicos.Servico;
import servicos.Servicos;
import suporte.CanalCatalogo;
import suporte.Configuracao;
import suporte.MediadorBase;

/**
 * @author Administrador
 *
 * TODO To change the template for this generated type comment go to
 * Window - Preferences - Java - Code Style - Code Templates
 */
public class Servidor extends MediadorBase {
   private int portaRecepcao;
   private String servico = "";
   public  Servicos catalogos;

        public Servidor(String servico, int portaCatalogo, int portaRecepcao)
throws UnknownHostException{
                super(portaCatalogo);
                this.servico = servico;
                this.portaRecepcao=portaRecepcao;
                //cria uma lista de catalogos
                catalogos = new Servicos();
        }

        public void executar() throws SocketException{
```

```
                canal = new CanalCatalogo(portaRecepcao, 2000);
                try{
                        MensagemCatalogo msg = new
MensagemCatalogo(MensagemCatalogo.REGISTRAR,servico,portaRecepcao);
                        canal.enviarBroadcast(msg, portaCatalogo);
                        receberRespostasCatalogos();
                        System.out.println("> Servico iniciado!");
                        //deveria disparar a thread do servico
                }
                catch(IOException e){
                        e.printStackTrace();
                }
        }
        protected void receberRespostasCatalogos() throws IOException
        {
                try
                {
                        do{
                        MensagemCatalogo resposta = canal.receber();
                        String msgServico               = resposta.getTxtMsg();
                                String msgIp                    = resposta.getIp();
                                int    msgPorta         = resposta.getPortaResposta();
                        Servico catalogo = new Servico(Servico.CATALOGO,
msgIp, portaCatalogo);
                                catalogos.add(servico);
                }while(true);
                }catch(SocketTimeoutException e){
                        System.out.println("Catálogos disponíveis:
"+catalogos.size());
                }

        }

        public static void main(String [] args) {
                try {
                        int     portaResposta;
                        int     portaCatalogo;
                        Servidor servidor;
                        String   file = args [0];

                        Configuracao.loadProperties(file);
                        portaResposta =
Integer.parseInt(Configuracao.propriedades.getProperty("portaServidor"));
                        portaCatalogo =
Integer.parseInt(Configuracao.propriedades.getProperty("portaCatalogo"));

                        servidor = new Servidor("TESTE", portaCatalogo,
portaResposta);
```

```
                servidor.executar();

            } catch (Exception e) {
                e.printStackTrace();
            }
        }
}
```

```
/*
 * Created on 29/01/2005
 *
 * TODO To change the template for this generated file go to
 * Window - Preferences - Java - Code Style - Code Templates
 */
package servicos;

import java.sql.Time;

public class Servico {

        public static final String CATALOGO = "catálogo";
        private String nome;
        private String ip;
        private int    porta;
        private Time tempo;

        public String getIp() {
            return ip;
        }

        public void setIp(String ip) {
            this.ip = ip;
        }

        public String getNome() {
            return nome;
        }

        public void setNome(String nome) {
            this.nome = nome;
        }

        public int getPorta() {
            return porta;
        }
```

```
	public void setPorta(int porta) {
		this.porta = porta;
	}

	public Time getTempo() {
		return tempo;
	}

	public void setTempo(Time tempo) {
		this.tempo = tempo;
	}

	public Servico(String nome,String ip,int  porta) {
		this.nome = nome;
		this.ip   = ip;
		this.porta= porta;
	}


}
```

```
/*
 * Created on 03/02/2005
 *
 * TODO To change the template for this generated file go to
 * Window - Preferences - Java - Code Style - Code Templates
 */
package servicos;

import java.util.ArrayList;

/**
 * @author Administrador
 *
 * TODO To change the template for this generated type comment go to
 * Window - Preferences - Java - Code Style - Code Templates
 */
public class Servicos extends ArrayList{

	public Servicos() {
		super();
	}

	public boolean add(Servico o) {
		return super.add(o);
	}
```

```
	public Servico localizar(String nome)
	{
		Servico aux;
		for(int i=0; i < size(); i++){
			aux = (Servico)get(i);
			if(aux.getNome().equals(nome))
				return aux;
		}
		return null;
	}
	public Object [] localizarTodos(String nome)
	{

		Servico aux;
		ArrayList vet = new ArrayList();
		for(int i=0; i < size(); i++){
			aux = (Servico)get(i);
			if(aux.getNome().equals(nome))
				vet.add(aux);
		}
		return vet.toArray();
	}
}
```

```
/*
 * Created on 29/01/2005
 *
 * TODO To change the template for this generated file go to
 * Window - Preferences - Java - Code Style - Code Templates
 */
package mensagens;

import java.net.DatagramPacket;

/**
 * @author Administrador
 *
 * TODO To change the template for this generated type comment go to Window
-
 * Preferences - Java - Code Style - Code Templates
 */
public class MensagemCatalogo {

	public static final String REGISTRAR  = "registrar";
	public static final String LOCALIZAR  = "localizar";
	public static final String REGISTRADO = "registrado";
```

```
	public static final String ERRO       = "erro";
	public static final String SINALIZACAO= "sinalizacao";

	public static final String SEPARADOR  = ":";



	protected String tipo;
	private String txtMsg;
	private int portaResposta;
	private DatagramPacket datagrama;


	public MensagemCatalogo(String tipo, String txtMsg, int portaResposta) {
		this.tipo = tipo;
		this.txtMsg = txtMsg;
		this.portaResposta = portaResposta;
		this.datagrama= null;
	}

	public MensagemCatalogo(String mensagem) throws
NumberFormatException {
		decompor(mensagem);
		datagrama = null;
	}

	public MensagemCatalogo(DatagramPacket rec)throws
NumberFormatException {
		datagrama = rec;
		String mensagem = new String(rec.getData()).trim();
		decompor(mensagem);
	}

	public void decompor(String mensagem){
		String [] aux = mensagem.split(SEPARADOR);
		tipo        = aux [0];
		txtMsg      = aux [1];
		portaResposta= Integer.parseInt(aux [2]);
	}

	public String getTipo(){
		return tipo;
	}

	public String getTxtMsg(){
		return txtMsg;
	}
```

```
   public void transformarResposta(String txtMsg)
   {
         if(tipo.equals(REGISTRAR))
               tipo = REGISTRADO;
         else throw new RuntimeException("Impossivel gerar reposta!");
         this.txtMsg = txtMsg;
   }
   public void transformarErro(String txtMsg)
   {
     tipo = ERRO;
     this.txtMsg = txtMsg;
   }

         public int getPortaResposta() {
               return portaResposta;
         }

         public String toString(){
               return
tipo+SEPARADOR+txtMsg+SEPARADOR+String.valueOf(portaResposta);
         }

         public DatagramPacket getDatagrama() {
               return datagrama;
         }

         public String getIp()
         {
               if(datagrama == null)
                     return "127.0.0.1";
               return datagrama.getAddress().getHostAddress();
         }
```

```
/*
 * Created on 03/02/2005
 *
 * TODO To change the template for this generated file go to
 * Window - Preferences - Java - Code Style - Code Templates
 */
package mensagens;

import java.net.DatagramPacket;

public class MensagemRespostaServico extends MensagemCatalogo {


         public static final String NAOLOCALIZADO= "servicoNaoLocalizado";
```

```
	public static final String LOCALIZADO   = "localizado";
	public static final String SEPARADOR_SERVICO  = "#";
	protected String ipServidor;
	protected int    portaServidor;

	public MensagemRespostaServico(String tipo, int portaResposta, String
servidor, int portaServidor) {
		super(tipo, servidor+SEPARADOR_SERVICO+portaServidor,
portaResposta);
		this.ipServidor     = servidor;
		this.portaServidor= portaServidor;
	}

	public MensagemRespostaServico(String mensagem)
			throws NumberFormatException {
		super(mensagem);
		decomporServico(this.getTxtMsg());
	}

	public MensagemRespostaServico(DatagramPacket rec)
			throws NumberFormatException {
		super(rec);
		decomporServico(this.getTxtMsg());
	}

	public void decomporServico(String servico)
	{
		String [] aux = servico.split(SEPARADOR_SERVICO);
		ipServidor     = aux [0];
		portaServidor= Integer.parseInt(aux [1]);
	}

	public void transformarResposta(String txtMsg)
	{
		if(txtMsg.equals(NAOLOCALIZADO))
			tipo = NAOLOCALIZADO;
		else super.transformarResposta(txtMsg);
	}

	public int getPortaServidor() {
		return portaServidor;
	}

	public String getIpServidor() {
		return ipServidor;
	}

}
```

```
package cliente;

import java.io.IOException;
import java.net.SocketException;
import java.net.SocketTimeoutException;
import java.net.UnknownHostException;

import mensagens.MensagemCatalogo;
import mensagens.MensagemRespostaServico;
import servicos.Servico;
import servicos.Servicos;
import suporte.CanalCatalogo;
import suporte.Configuracao;
import suporte.MediadorBase;

public class Cliente extends MediadorBase {

	public Servicos servidores;
	public Servicos catalogos;
	private String servico = "";
	private int portaRecepcao;

	public Cliente(String servico, int portaCatalogo,  int portaRecepcao)
throws UnknownHostException {
		super(portaCatalogo);
		//cria uma lista de servidores
		servidores = new Servicos();
		this.servico = servico;
		this.portaRecepcao=portaRecepcao;
		//cria uma lista de catalogos
		catalogos = new Servicos();
	}

	public void executar() throws SocketException {
		canal = new CanalCatalogo(portaRecepcao, 2000);
		try{
			MensagemCatalogo msg = new
MensagemCatalogo(MensagemCatalogo.LOCALIZAR,servico,portaRecepcao);
			canal.enviarBroadcast(msg, portaCatalogo);
			receberRespostasCatalogos();
			// comunicação com o servidor...
		}
		catch(IOException e){
			e.printStackTrace();
		}
	}
```

```
	protected void receberRespostasCatalogos() throws IOException
	{
		try
		{
			do{
				MensagemRespostaServico resposta =
canal.receberMensagemRespostaServico();
				String ipServidor	= resposta.getIpServidor();
				String msgIp		= resposta.getIp();
				int	portaServidor	= resposta.getPortaServidor();
				int	msgPorta	= resposta.getPortaResposta();
				Servico catalogo = new
Servico(Servico.CATALOGO, msgIp,	portaCatalogo);
				catalogos.add(servico);
				System.out.println("registrando catálogo em:
"+msgIp);
				System.out.println("Catálogos disponíveis:
"+catalogos.size());
				if(resposta.getTipo().equals(
MensagemRespostaServico.NAOLOCALIZADO))
					System.out.println("Servico indisponível!");

				else
				{
					if(resposta.getTipo().equals(
MensagemRespostaServico.ERRO))
						System.out.println("Ocorreu um erro
no Catálogo.");
					else{

						Servico servidor = new
Servico(this.servico, ipServidor, portaServidor);
						servidores.add(servico);
						System.out.println("registrando
servico em: "+ipServidor+" na porta:"+portaServidor);
					}
				}
			}while(true);
			}catch(SocketTimeoutException e){
				System.out.println("Servidores disponíveis:
"+servidores.size());
			}
	}

	public static void main(String [] args) {
		try {
			int	portaResposta;
```

```
                        int     portaCatalogo;
                        Cliente  cliente;
                        String   file = args [0];

                        Configuracao.loadProperties(file);
                        portaResposta =
Integer.parseInt(Configuracao.propriedades.getProperty("portaCliente"));
                        portaCatalogo =
Integer.parseInt(Configuracao.propriedades.getProperty("portaCatalogo"));

                        cliente = new Cliente("TESTE2", portaCatalogo,
portaResposta);
                        cliente.executar();

                } catch (Exception e) {
                        e.printStackTrace();
                }
        }
}
```

```
package suporte;

import java.io.IOException;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;
import java.net.NoRouteToHostException;
import java.net.SocketException;
import java.net.UnknownHostException;

import mensagens.MensagemCatalogo;
import mensagens.MensagemRespostaServico;


public class CanalCatalogo {
        protected DatagramSocket udpSocket;

        public CanalCatalogo(int porta) throws SocketException{
                udpSocket = new DatagramSocket(porta);
                System.out.println("Canal UDP aberto na porta: "+porta);
        }

        public CanalCatalogo(int porta, int timeout) throws SocketException{
                this(porta);
                udpSocket.setSoTimeout(timeout);
        }
```

```
	public void enviarMensagem(MensagemCatalogo msg, String ip, int
porta)throws UnknownHostException, IOException{
		byte [] buffer = msg.toString().getBytes();
		InetAddress group = InetAddress.getByName(ip);
		DatagramPacket data = new DatagramPacket(buffer, buffer.length,
group, porta);
		udpSocket.send(data);
	}

	public void enviarBroadcast(MensagemCatalogo msg, int porta) throws
UnknownHostException, IOException{
		byte [] buffer = msg.toString().getBytes();
		try{

		InetAddress group = InetAddress.getByName("255.255.255.255");
		DatagramPacket data = new DatagramPacket(buffer, buffer.length,
group, porta);
		udpSocket.send(data);
	    }
	    catch(NoRouteToHostException e)	{
		System.out.println("> Não foi possível encontrar rede, rodando
localmente.");
		InetAddress group = InetAddress.getByName("127.0.0.1");
		DatagramPacket data = new DatagramPacket(buffer, buffer.length,
group, porta);
		udpSocket.send(data);
		}
	}

	public MensagemCatalogo receber() throws IOException
	{
		DatagramPacket rec = new DatagramPacket(new byte [1024],
1024);
		udpSocket.receive(rec);
		return  new MensagemCatalogo(rec);
	}

	public MensagemRespostaServico receberMensagemRespostaServico()
throws IOException
	{
		DatagramPacket respostaServico = new DatagramPacket(new byte
[1024], 1024);
		udpSocket.receive(respostaServico);
		return  new MensagemRespostaServico(respostaServico);
	}


}
```

```
/*
 * Created on 29/01/2005
 *
 * TODO To change the template for this generated file go to
 * Window - Preferences - Java - Code Style - Code Templates
 */
package suporte;

import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;
import java.util.Properties;

/**
 * @author Administrador
 *
 * TODO To change the template for this generated type comment go to Window
-
 * Preferences - Java - Code Style - Code Templates
 */
public class Configuracao {

	public static Properties propriedades;

	public static void loadProperties(String fileName) throws IOException {
		InputStream propsFile;
		propriedades = new Properties();
		propsFile = new FileInputStream(fileName);
		propriedades.load(propsFile);
		propsFile.close();
	}

}
```

```
package suporte;

/**
 *
 * @author Administrator
 */

import java.net.InetAddress;
import java.net.UnknownHostException;

public class Local
```

```
{
        /** Creates a new instance of GetHostName */
        public static void printInfo() throws UnknownHostException
        {
                InetAddress addr = InetAddress.getLocalHost();
                System.out.println("Local IP:" +addr.getHostAddress());
                System.out.println("Local Host Name:" +  addr.getHostName());
        }

        public static String getIP()throws UnknownHostException
        {
                InetAddress addr = InetAddress.getLocalHost();
                return addr.getHostAddress();
        }

        public static String getHost()throws UnknownHostException
        {
                InetAddress addr = InetAddress.getLocalHost();
                return addr.getHostName();
        }

}
```

```
/*
 * Created on 03/02/2005
 *
 * TODO To change the template for this generated file go to
 * Window - Preferences - Java - Code Style - Code Templates
 */
package suporte;

import java.net.SocketException;
import java.net.UnknownHostException;


/**
 * @author Administrador
 *
 * TODO To change the template for this generated type comment go to
 * Window - Preferences - Java - Code Style - Code Templates
 */
abstract public class MediadorBase {

        protected int portaCatalogo;
        public static CanalCatalogo canal;

        public MediadorBase(int portaCatalogo)throws UnknownHostException {
```

```
                super();
                this.portaCatalogo = portaCatalogo;
                System.out.println("** "+this.getClass().getName()+" **");
                Local.printInfo();
        }
        abstract public void executar() throws SocketException;

        public int getPortaCatalogo() {
                return portaCatalogo;
        }

        public void setPortaCatalogo(int portaCatalogo) {
                this.portaCatalogo = portaCatalogo;
        }
}
```

```
/*
 * Created on 29/01/2005
 *
 * TODO To change the template for this generated file go to
 * Window - Preferences - Java - Code Style - Code Templates
 */
package suporte;

import java.util.Enumeration;

import suporte.Configuracao;

/**
 * @author Administrador
 *
 * TODO To change the template for this generated type comment go to
 * Window - Preferences - Java - Code Style - Code Templates
 */
public class Teste {

        public static void main(String [] args) {
                String file =
"D:\\eclipse\\workspace\\Mediador\\suporte\\configuracao.properties";
                Configuracao.loadProperties(file);
                System.out.println(Configuracao.propriedades.size());
                Enumeration aux = Configuracao.propriedades.propertyNames();
                 while ( aux.hasMoreElements())
                 {
              String nm = (String)aux.nextElement();
                        System.out.println(nm+":
"+Configuracao.propriedades.getProperty(nm));
```

```
            }
        }
}
```