Instituto de Computação
Universidade Estadual de Campinas

Dissertação de Mestrado

Uma modelagem para Comércio Eletrônico usando CORBA e Agentes Móveis

Eduardo José Rodríguez

Fevereiro de 1999

Banca Examinadora:

- Prof. Dr. Edmundo Roberto Mauro Madeira (Orientador)
  Instituto de Computação - UNICAMP
- Prof. Dr. Edson dos Santos Moreira
  Instituto de Ciências Matemáticas e de Computação - USP São Carlos
- Prof. Dr. Rogério Drummond Burnier Pessoa de Mello Filho
  Instituto de Computação - UNICAMP
- Prof. Dr. Luiz Eduardo Buzato (Suplente)
  Instituto de Computação - UNICAMP

BC
Ex
37208
229/99
X
R$ 11,00
07/04/99

FICHA CATALOGRÁFICA ELABORADA PELA
BIBLIOTECA CENTRAL DA UNICAMP

-00122049-5

Rodríguez, Eduardo José
R618m Uma modelagem para comércio eletrônico usando CORBA e agentes móveis / Eduardo José Rodríguez. – Campinas, SP : [s.n.], 1999.

Orientador : Edmundo Roberto Mauro Madeira.
Dissertação (mestrado) - Universidade Estadual de Campinas, Instituto de Computação.

1. CORBA (Programação de computador). 2. Comércio eletrônico. 3. Catálogo eletrônico. I. Madeira, Edmundo Roberto Mauro. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Resumo

O crescimento da Internet e o avanço das tecnologias de redes têm contribuído positivamente ao desenvolvimento do **Comércio Eletrônico** nos últimos anos. O surgimento de plataformas proprietárias para responder às crescentes necessidades na área, somado à falta de padronização, tem originado problemas de interoperabilidade que nem sempre são fáceis de resolver principalmente em aplicações de busca de produtos numa rede.

Esta é a questão que abordaremos neste trabalho, modelando uma aplicação de Comércio Eletrônico sobre a plataforma CORBA que consiste de um agente móvel que procura produtos numa rede em nome de um cliente. A modelagem compreende também o objeto catálogo que contém as ofertas de produtos e apresenta suas interfaces através de um ORB. Adotamos um modelo de três componentes onde o primeiro deles é o agente, o segundo está representado pelas interfaces do catálogo num ORB e o terceiro por um SGBD que gerencia o banco de dados que contém os dados do catálogo.

Abstract

The fast growth of the Internet and the network technology advance have contributed to develop the Electronic Commerce in the last years. The emergence of proprietary platforms to meet the growing requirements in this area and the lack of standards have originated several problems of interoperability, mainly in areas that deal with searching products in a network.

This is the problem that we aim to address. We model an Application that implements a mobile agent to search products in a CORBA based Electronic Commerce environment. We also model the catalogue object that interacts with the mobile agent and presents its interface through an ORB. We adopt a three tiers model where the first tier is the mobile agent, the second one is represented by the catalogue interface and the third tier is implemented using a DBMS that manages the database that stores the catalogue data.

Uma modelagem para Comércio Eletrônico usando CORBA e Agentes Móveis

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Eduardo José Rodríguez e aprovada pela Banca Examinadora.

Campinas, 05 de fevereiro de 1999

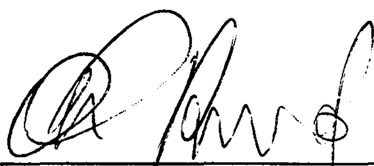
Prof. Dr. Edmundo Roberto Mauro Madeira
(Orientador)

Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação

TERMO DE APROVAÇÃO

Dissertação defendida e aprovada em 05 de fevereiro de 1999, pela Banca Examinadora composta pelos Professores Doutores:

Porf. Dr. Edson dos Santos Moreira
ICMSC - USP



Prof. Dr. Rogério Drummond Burnier Pessoa de Mello Filho
IC - UNICAMP

Prof. Dr. Edmundo Roberto Mauro Madeira
IC - UNICAMP

A mi esposa Ruth

Agradecimentos

Desejo expressar meus sinceros agradecimentos ao Prof. Dr. Edmundo R.M. Madeira, meu orientador, pela discussão franca de idéias, pelos valiosos aportes para a pesquisa e, principalmente, pela sua calidez humana.

Agradeço aos professores do Instituto de Computação, por suas valiosas contribuições para a minha formação.

Meu profundo agradecimento a minha família e amigos da Argentina pelo acompanhamento à distancia e pelo amparo afetivo que tornou mais leve estar longe de nossa terra.

Não posso deixar de agradecer a todas as pessoas que aqui no Brasil nos acompanharam durante esta etapa, obrigado por tudo.

Aos meus colegas do Instituto de Computação obrigado pela amizade e pelo tempo compartilhado.

Um agradecimento especial a minha companheira Ruth a cuja teimosia devo a possibilidade de estar obtendo o meu mestrado.

Este trabalho foi desenvolvido com auxílio da FAPESP (processo n° 97/11129-0).

SUMÁRIO

INTRODUÇÃO 1

CAPÍTULO 1

FUNDAMENTAÇÃO 4

1.1.- CORBA (*COMMON OBJECT REQUEST BROKER ARQUITECTURE*) 4

1.1.1.- Introdução *4*

1.1.2.- IDL (Interface Definition Language) *6*

1.1.3.- ORB (Object Request Broker) *6*

1.1.4.- Serviços CORBA *7*

1.1.5.- Facilidades CORBA *8*

1.1.6.- Interfaces de Domínio *9*

1.2.- COMÉRCIO ELETRÔNICO 9

1.2.1.- Introdução *9*

1.2.2.- Escopo do Comércio Eletrônico *10*

1.2.3.- Questões em aberto *12*

1.3.- COMÉRCIO ELETRÔNICO E CORBA 13

1.3.1.- Facilidades CORBA *13*

1.3.2.- Um Modelo de Referência para Comércio Eletrônico *14*

1.3.2.1.- A arquitetura 14

1.3.2.2.- Facilidades para Dados semânticos 15

1.3.2.3.- Facilidade de Catálogo 16

1.3.2.4.- Facilidade de Corretagem (Brokerage) 17

1.3.2.5.- Facilidade de Agência 18

1.4.- AGENTES MÓVEIS 19

1.4.1.- Definição e tipos *19*

1.4.2.- Conceitos principais *19*

CAPÍTULO 2

MODELAGEM DO SERVIÇO DE COMÉRCIO ELETRÔNICO........ 22

2.1.- PRELIMINARES........ 22

2.1.1.- Sistema de Agentes........ *24*

2.1.2.- Semântica dos dados........ *24*

2.1.2.1.- Transporte: SDOs........ 24

2.1.2.2.- Armazenamento........ 28

2.1.3.- Semântica das Consultas........ *30*

2.2.- DESENVOLVIMENTO DO MODELO DE OBJETOS........ 32

2.2.1.- Modelo geral........ *32*

2.2.1.1.- Cliente........ 33

2.2.1.2.- Trader........ 34

2.2.1.3.- Sistema de Agentes........ 35

2.2.1.4.- Agência........ 35

2.2.2.- Catálogo........ *36*

2.2.2.1.- Interface Loockup........ 38

2.2.2.2.- Interface Admin........ 39

2.2.3.- Agente........ *42*

CAPÍTULO 3

ASPECTOS DA IMPLEMENTAÇÃO........ 45

3.1.- CONSIDERAÇÕES GERAIS........ 45

3.2.- SEMANTIC DATA OBJECTS........ 46

3.2.1 Classe Sdo........ *46*

3.2.2.- Classe Convert........ *48*

3.3.- CATÁLOGO........ 49

3.3.1.- Sobre o ORB........ *49*

3.3.2.- Sobre a implementação do catálogo........ *51*

3.3.2.1.- O Servidor........ 51

3.3.2.2.- A API JDBC........ 51

3.3.2.3.- A estrutura do Banco de Dados........ 53

3.3.2.4.- Os métodos........ 53

3.3.3.- O SGBD........ *56*

3.4.- AGENTE........ 57

3.4.1.- Sobre o Sistema de Agentes *57*
3.4.2.- Sobre o Agente *59*
3.4.2.1.- Classe QWorker 59
3.4.2.2.- Classe DecisionModule 60
3.4.2.3.- Classe Query 60
3.5.- INTERFACE DO USUÁRIO 62
3.5.1.- Classe Launcher *62*
3.5.2.- Classe Receiver *64*
3.6.- SISTEMA GERAL 65
3.7.- ALGUMAS CONSIDERAÇÕES SOBRE SEGURANÇA 68
3.8.- TRABALHOS RELACIONADOS 68
CONCLUSÃO **70**
TRABALHOS FUTUROS 72
REFERÊNCIAS BIBLIOGRÁFICAS **74**
APÊNDICE A: IDL DO CATÁLOGO **77**
APÊNDICE B: SISTEMA DE AGENTES ODYSSEY **79**
B-1.- API ODYSSEY 79
B-1.1.- Agentes *79*
B-1.2.- Workers *79*
B-1.3.- Sistema de agentes *79*
B-1.4.- Lugares *80*
B-1.5.- A hierarquia de classes Odyssey *80*
B.2.- QUESTÕES OPERACIONAIS 81

Introdução

Nos últimos cinco anos tem-se desenvolvido um grande interesse em Comércio Eletrônico como uma nova maneira de levar adiante transações comerciais ou de negócios.

Nos primeiros anos de Comércio Eletrônico o nível de transações foi relativamente baixo, mas em 1997 explodiu e chegou a níveis de vários bilhões de dólares. Previsões atuais esperam que este crescimento explosivo continue, atingindo um volume de negócios de U$S 300 bilhões no ano 2001. Alguns sugerem que será maior e superará U$S 1 trilhão nos próximos 5 anos. Esta expetativa pode ser real se considerarmos que nos próximos 10 anos 1 bilhão de pessoas estarão conectadas à Internet [8].

Por outro lado, além dos necessários mecanismos de segurança e pagamento, Comércio Eletrônico depende cada vez mais do surgimento de capacidades ou funcionalidades que possam ser usadas pelos compradores para rápida e facilmente obterem os dados necessários sobre produtos para poderem efetuar decisões de compra corretas.

Esta é a questão que pretendemos abordar na nossa pesquisa modelando um aplicação de Comércio Eletrônico sobre a plataforma CORBA para busca de produtos numa rede utilizando também agentes móveis.

Entre as capacidades necessárias para Comércio Eletrônico temos:

a) disponibilidade de mecanismos de busca aceitos universalmente.

b) formatos padrões para apresentação da informação.

c) mecanismos que permitam conhecer itens ou serviços que complementam um dado produto.

Além disso e considerando catálogos eletrônicos, a interoperabilidade é crucial, não somente porque permite juntar dados de fontes diversas, mas também porque permite a fornecedores e consumidores localizar e obter qualquer informação que eles desejarem[4].

O surgimento de plataformas proprietárias para responder às crescentes necessidades na área, somado à falta de padronização, tem originado problemas de interoperabilidade que nem sempre são fáceis de resolver.

Em ambientes para processamento distribuído aberto orientados a objeto vem sendo desenvolvida uma iniciativa de padronização liderada pelo OMG (Object Management Group), a plataforma **CORBA** *(Common Object Request Broker Architecture).*

O OMG em parceria com outras organizações está produzindo alguns avanços na padronização de funcionalidades básicas de Comércio Eletrônico. No capítulo 1 apresentamos o Modelo de Referência para Comércio Eletrônico que está sendo objeto de estudo por estas organizações.

Do ponto de vista das tecnologias, já em 1995 a nova tecnologia de agentes móveis era considerada como uma facilidade para as tarefas de Comércio Eletrônico[10].

Hoje podemos achar várias implementações nesta área feitas com agentes cobrindo desde busca de produtos e preços, negociação, venda e entrega até serviços pós-venda[9].

No campo da procura de produtos e preços em nome de um cliente (*brokering*), a maioria das abordagem compreende agentes interagindo com catálogos implementados em páginas *web*. Devido ao fato de que os *Web sites* constróem suas requisições CGI e as respostas HTML de formas diferentes e as vezes estas mudam facilmente, manter implementações que levem em conta todas as possibilidades e mudanças é muito difícil[2].

Considerando este estado do desenvolvimento e que a habilidade para realizar uma busca efetiva por um produto específico é crucial para que o Comércio Eletrônico na Internet possa realizar todo seu potencial, é que nos propomos desenvolver uma implementação, baseada em agentes móveis, para procura de produtos num ambiente de Comércio Eletrônico baseado em CORBA, modelando principalmente o objeto catálogo com o qual o agente vai interagir e que apresentará suas interfaces através de um ORB.

Na modelagem do agente consideraremos dotar o agente de uma certa "inteligência" nos mecanismos de escolha para que o usuário possa fornecer alternativas para o produto a procurar.

No capítulo 1 apresentamos alguns conceitos sobre a plataforma CORBA, Comércio Eletrônico, o Modelo de Referência para Comércio Eletrônico apresentado pelo OMG, e Agentes Móveis, os quais consideramos relevantes para o desenvolvimento de nossa pesquisa.

O capítulo 2 apresenta a modelagem do sistema geral e das diversas partes do sistema com ênfase especial no modelo do catálogo. Também apresentamos aqui os *Semantic Data Objects* que serão a estrutura utilizada para transporte de dados.

A implementação é detalhada e analisada no capítulo 3.

Finalmente apresentamos a conclusão e possíveis trabalhos futuros baseados nesta pesquisa.

Capítulo 1

Fundamentação

1.1.- CORBA (*Common Object Request Broker Arquitecture*)

1.1.1.- Introdução

Em 1989, foi formado um grupo, o OMG (*Object Management Group*), que hoje é um consórcio formado por mais de 800 entidades da área de computação, com o propósito de desenvolver padrões que admitissem a interoperabilidade e portabilidade de aplicações distribuídas orientadas a objeto. O trabalho deste grupo concentra-se no desenvolvimento da especificação da Arquitetura de Gerenciamento de Objetos - **OMA** (*Object Management Architecture*), dentro dela o elemento mais importante é a arquitetura de comunicação denominada *Common Object Request Broker Architecture* ou, simplesmente, **CORBA**. A seguir apresentamos alguns conceitos que caraterizam esta arquitetura.

Em CORBA os serviços que um objeto oferece são exportados numa *interface* entre o objeto e o resto do sistema. A especificação desta interface estabelece os serviços que o objeto provê e

como devem ser construídas as mensagens para invocar esses serviços. Além disso, cada objeto possui uma única identificação dentro da infra-estrutura que permite a entrega das mensagens para ele.

As interfaces estão especificadas através de uma linguagem padrão definida pelo OMG e denominada **IDL** (*Interface Definition Language*) de forma que o acesso aos objetos é independente da linguagem na qual as aplicações foram feitas.

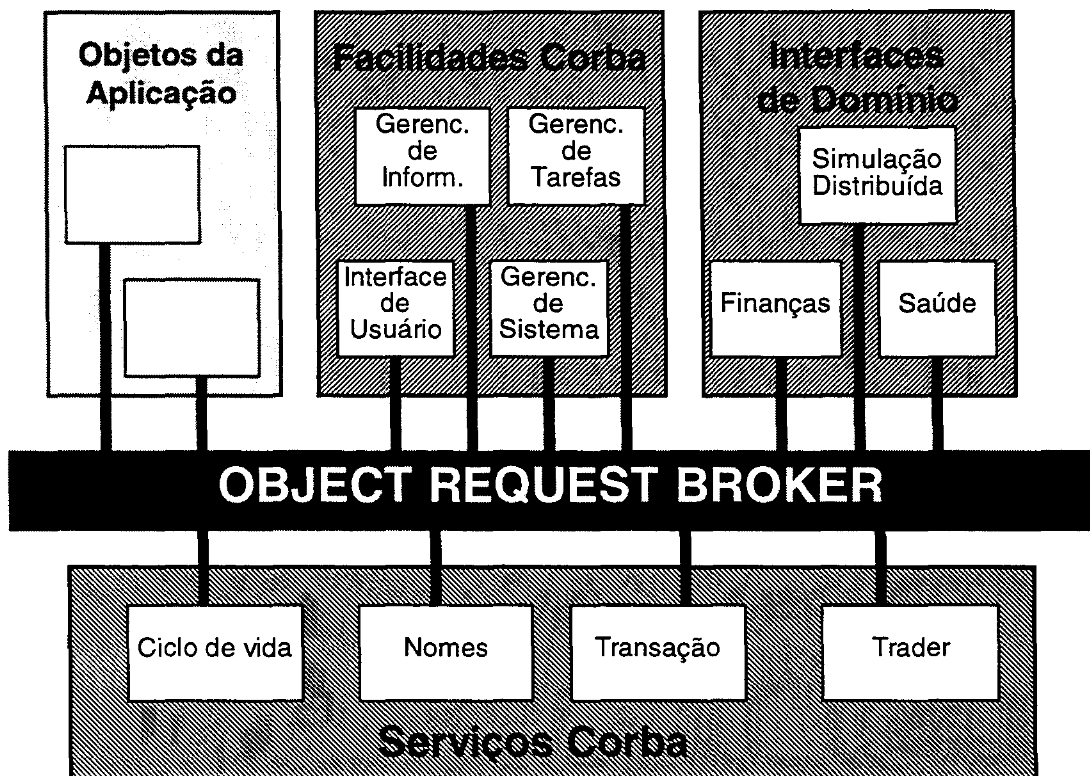


Figura 1 - Arquitetura OMA

Os cinco elementos principais da arquitetura OMA [19] são mostrados na Figura 1.

- O **Negociador de Requisições de Objetos** ou *Object Request Broker* (ORB) define o meio que permite a interação entre objetos do ambiente distribuído.

- Os **Serviços CORBA** (*CORBA Services*) definem uma coleção de serviços (interfaces e objetos) que oferecem funções básicas para uso e implementação de objetos.
- As **Facilidades CORBA** (*CORBA Facilities*) definem um conjunto de serviços com funções diversas para uso por parte das aplicações.
- As **Interfaces de Domínio** (*Domain Interfaces*) são interfaces que fornecem funcionalidades comuns para um segmento específico de mercado.
- Os **Objetos de Aplicação** (*Application Objects*) definem as aplicações e serviços específicos não padronizados pelo OMG, e são as aplicações de usuário.

1.1.2.- IDL (Interface Definition Language)

Como já foi dito, IDL é a linguagem padrão que permite a especificação das interfaces dos objetos. A IDL é usada para especificar os atributos dos componentes, as classes das quais eles herdam, as exceções que podem ocorrer, os tipos de eventos que emitem e os métodos que suas interfaces suportam, incluindo os parâmetros de entrada e saída requeridos e seus tipos. A IDL é puramente declarativa, não inclui detalhes de implementação. A gramática da IDL é um subconjunto de C++ com adições para suportar conceitos de distribuição[14].

1.1.3.- ORB (Object Request Broker)

O ORB é o meio que permite o estabelecimento de uma relação cliente/servidor entre objetos.

Utilizando o ORB, um objeto cliente pode, transparentemente, invocar um método em um objeto do servidor (chamado de Implementação de Objeto), que pode estar na mesma máquina ou em qualquer ponto da rede.

O ORB recebe a requisição do cliente com os parâmetros necessários e é responsável por achar o objeto que satisfaça a requisição, preparar esse objeto para receber a requisição, invocar nele os métodos apropriados e retornar os resultados se for o caso.

1.1.4.- Serviços CORBA

Os Serviços CORBA são uma coleção de serviços em nível de sistema. Eles podem ser pensados como aumentando e completando a funcionalidade do ORB[27]. A seguir enumeramos estes serviços:

- **Ciclo de Vida**, define operações para criar, copiar, mover, e deletar objetos.
- **Persistência**, permite o armazenamento de objetos em forma persistente em vários modelos, como por exemplo em bancos de dados orientados a objeto.
- **Nomes**, permite que os objetos sejam localizados por nomes associados a eles.
- **Eventos**, através do Serviço de Notificação de Eventos os componentes do sistema podem registrar seu interesse em eventos específicos e serem notificados quando estes acontecem.
- **Controle de Concorrência**, provê um sistema de gerenciamento de acesso a objetos ou dados.
- **Transação**, provê coordenação do processo de efetivação em duas fases em componentes, usando transações.
- **Relacionamento**, fornece uma maneira para a criação dinâmica de associações entre objetos que não tem informação uns dos outros.
- **Externalização**, fornece uma maneira padrão para serializar e desserializar um objeto. É usado quando se quer transportar um objeto a diferentes processos, máquinas ou ORBs.
- **Consulta**, provê operações de consulta para objetos.
- **Licenciamento**, fornece operações para o estabelecimento de licenças de uso sobre objetos.
- **Propriedades**, provê operações para associar propriedades dinâmicas aos componentes.

- **Tempo**, este serviço fornece interfaces para sincronização do tempo em ambientes de objetos distribuídos.
- **Segurança**, provê um *framework* para segurança de objetos distribuídos. Suporta autenticação, listas de controle de acesso, confidencialidade e não repudiação.
- **Coleção**, fornece interfaces CORBA para criar e manipular os agrupamentos mais comuns de objetos tais como filas, pilhas, listas, matrizes, árvores, etc.
- **Trader**, fornece um serviço de "páginas amarelas" para objetos.

Um dos serviços que merece atenção especial é o serviço de *trading* ou *Trader* [20]. Este é um dos serviços mais importantes e com maior funcionalidade dentro da arquitetura OMA.

O ***Trader*** pode ser visto como um objeto através do qual outros objetos podem anunciar seus serviços. Ele também pode ser consultado para a localização de um determinado serviço com determinadas características e retorna, se o serviço procurado estiver no seu domínio, a referência de interface do objeto que oferece aquele serviço e as características da citada interface para que o cliente possa fazer a sua requisição.

1.1.5.- Facilidades CORBA

As Facilidades CORBA [21] são um conjunto de componentes definidos em IDL que fornecem serviços diretamente aos objetos de aplicação. Tanto as Facilidades CORBA, quanto as Interfaces de Domínio , definem as regras de acordo que os componentes de negócio (objetos da aplicação) necessitam para colaborar efetivamente.

Estas facilidades incluem funções partilhadas por vários ou pela maioria dos sistemas. Quatro domínios são identificados para estas facilidades:

- **Interface de usuário**: torna um sistema de informação acessível aos seus usuários e responde às suas necessidades. Por exemplo, a apresentação de documentos compostos pertence a este domínio.

- **Gerenciamento de informação**: cobre a modelagem, definição, armazenamento, recuperação, gerenciamento e intercâmbio de informação.
- **Gerenciamento de sistema**: define interfaces para gerenciamento, instrumentação, configuração, instalação, operação e reparo de componentes de objetos distribuídos.
- **Gerenciamento de tarefas**: envolve a automação de trabalhos. Inclui workflow, transações longas, agentes, scripting, regras e e-mail.

1.1.6.- Interfaces de Domínio

As **Interfaces de Domínio** representam tecnologias que suportam vários segmentos específicos de mercado tais como Saúde, CAD, Sistemas financeiros, Telecomunicações, entre outros. Esta área de padronização foi separada das Facilidades Comuns no início de 1996 e era antigamente nomeada Facilidades Verticais. Exemplos desta área são as *Common Business Object Facilites*[33].

1.2.- Comércio Eletrônico

1.2.1.- Introdução

Existem diversas definições possíveis de Comércio Eletrônico. Uma pode ser: "Qualquer tipo de transação de negócios na qual as partes interajam eletronicamente em lugar de fazê-lo através de intercâmbio físico ou com contato direto". [6]

Uma outra seria considerar "Comércio Eletrônico como uma aplicação das tecnologias de informação, especificamente inteligência computacional e redes de computadores, aos problemas do comércio". [25]

Os negócios modernos estão caracterizados por um aumento crescente: das capacidades dos fornecedores, da competência global e das expectativas dos clientes. Em resposta, os negócios no mundo inteiro estão mudando sua organização e seu modo de operação, como por

exemplo horizontalizando suas estruturas hierárquicas e eliminando as barreiras entre companhias, clientes e fornecedores.

Comércio Eletrônico é considerado o meio para permitir e suportar estas mudanças em escala global, permitindo que as companhias sejam mais eficientes e flexíveis em sua operação, trabalhem mais perto de seus fornecedores e estejam mais ligadas às necessidades e expectativas de seus clientes.

O Comércio Eletrônico pode ser subdividido em quatro categorias :

- **Negócio - negócio** : compreende as relações comerciais entre companhias. Ex.: relação com os fornecedores.
- **Negócio - cliente**: a mais conhecida representada principalmente pelo comércio varejista.
- **Negócio - administração**: envolve todas as transações entre as companhias e as organizações do governo.
- **Cliente - administração**: transações entre o governo e seus contribuintes como por exemplo, pagamento de taxas e benefícios sociais.

1.2.2.- Escopo do Comércio Eletrônico

O Comércio Eletrônico é caraterizado pela sua diversidade. Como pode-se ver na Figura 2, compreende um amplo escopo de operações de negócios e transações que incluem:

- Estabelecimento do contato inicial, por exemplo entre cliente e fornecedor.
- Intercâmbio de informação.
- Suporte de serviços pré e pós venda.
- Venda.
- Pagamento eletrônico.

- Distribuição dos produtos.
- Empresas virtuais, grupos de companhias independentes que juntam suas capacidades de maneira a oferecer produtos ou serviços que não poderiam ser fornecidos por qualquer uma delas individualmente.
- Processos de negócios partilhados que são operados conjuntamente por uma companhia e seus parceiros comerciais

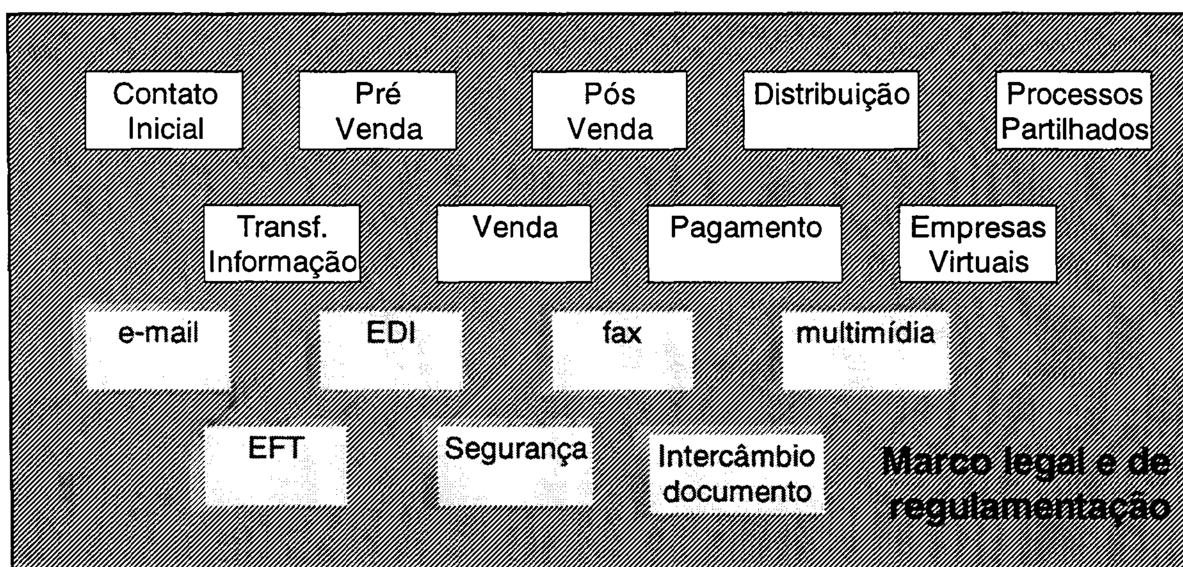


Figura 2 - Caraterísticas de Comércio Eletrônico

Como também pode-se observar na Figura 2, o Comércio Eletrônico encerra uma grande variedade de tecnologias de comunicação como e-mail, fax, multimídia, intercâmbio eletrônico de dado (EDI) e transferência eletrônica de fundos (EFT).

Além do enunciado é necessário um marco legal e de regulamentação bem definido que facilite as transações comerciais eletrônicas. Dado que a possibilidade de interação global é um dos fundamentos do comércio eletrônico, este marco legal e de regulamentação deverá ter um escopo global.

1.2.3.- Questões em aberto

Enquanto o comércio eletrônico cresce rapidamente, existem ainda algumas questões que devem ser resolvidas para que todo seu potencial possa ser desenvolvido. Dentre elas temos:

Globalização: potencialmente por meio das redes globais fica mais fácil fazer negócios entre companhias espalhadas por todo o mundo. Porém o meio de comunicação sozinho não é suficiente. Como as empresas podem ter conhecimento da existência umas das outras e dos produtos ou serviços oferecidos ou requeridos? Como uma companhia pode saber a maneira como se efetuam os negócios do outro lado do mundo? Como são consideradas as diferenças culturais e de idioma das distintas comunidades?

Questões financeiras e contratuais: dentro de um mercado globalizado onde uma empresa pode fazer um negócio com uma outra em nome de uma terceira, todas em países distintos, surgem algumas questões importantes a serem consideradas. Que mecanismos vão ser usados para acordar os termos do contrato? Quando um contrato vai ser estabelecido e qual seu marco legal? Quem vai ter jurisdição legal sobre o contrato? Como o pagamento vai ser efetuado e conferido? Quais as taxas que devem ser aplicadas aos produtos ou serviços? A quem devem ser pagas?

Propriedade: no caso de produtos que podem ser distribuídos eletronicamente, e portanto copiados facilmente, surge a necessidade de regulamentação e proteção da propriedade intelectual desse produto.

Privacidade e Segurança: dado que o Comércio Eletrônico se realiza através de redes abertas existe a necessidade de mecanismos para privacidade e segurança. Estes mecanismos devem fornecer principalmente confidencialidade, autenticação e não repudiação (assegurar que as partes não possam negar depois a sua participação numa transação).

Interconectividade e interoperabilidade: para realizar todo o potencial do comércio eletrônico é necessário que qualquer companhia ou cliente possa acessar qualquer organização oferecendo produtos ou serviços, independentemente da localização geográfica da mesma e da rede onde ela estiver conectada. Isto demanda acordos sobre padrões para conexão de redes e

interoperabilidade entre elas e também sobre os tipos de dados a trocar e a sua forma de apresentação.

1.3.- Comércio Eletrônico e CORBA

1.3.1.- Facilidades CORBA

Dentro do marco das, agora nomeadas, Interfaces de Domínio, o OMG emitiu em janeiro de 1996 o Requerimento para Propostas RFP-4 [17], chamando por submissões para suportar as seguintes Facilidades CORBA:

Objetos de negócios comuns (*Common Business Objects*): objetos que representam coisas ou conceitos tão comuns a todos os negócios que algo da semântica dessas coisas ou conceitos possa ser padronizado.

Facilidades para objetos de negócio (*Business Object Facilities*): fornecem a infra-estrutura interoperável ou marco requerido para suportar objetos como componentes de aplicação de uma solução de negócio. Provê a interface e os protocolos para que os componentes de aplicação possam colaborar e a infra-estrutura para suportá-los como componentes "plug & play".

Varias submissões foram apresentadas e alguns dos conceitos nelas expostos são referenciados na arquitetura apresentada na próxima seção.

Por outro lado, em abril de 1995 o OMG criou o Grupo de Interesse Específico (SIG) sobre Comércio Eletrônico encarregado de determinar se havia interesse suficiente em estender a OMA a Comércio Eletrônico. Comprovado um claro interesse na área, o SIG do Domínio de Comércio Eletrônico solicitou ao OMG, em junho de 1996, sua elevação ao status de Força de Tarefas para poderem enviar Requerimentos para Informação (RFI), desenvolver uma arquitetura de serviços para Comércio Eletrônico e enviar Requerimentos de Propostas (RFP).

A Força de Tarefas do Domínio de Comércio Eletrônico (EC DTF) [24] existe para definir e promover a especificação de tecnologias de objetos distribuídos, segundo o padrão do OMG, para o desenvolvimento e uso de aplicações de Comércio Eletrônico.

Dentro de seu trabalho a EC DTF emitiu dois RFIs. O primeiro [22](*Asset and Content Management*), foi lançado em agosto de 1996, solicitando informação sobre grupos de tecnologias e interfaces necessárias para o gerenciamento de bens e conteúdo. O segundo [23]data de novembro de 1996 (*Enabling Technologies and Services for Electronic Commerce*) e seu objetivo é requisitar informação que ajude na adoção de especificações que facilitem o Comércio Eletrônico em formato de objetos. Este RFI solicita informações nas áreas de: requerimentos, cenários, arquitetura, desenho, projeto, protocolos e padrões. Estes RFIs são a base da arquitetura apresentada na seção a seguir.

1.3.2.- Um Modelo de Referência para Comércio Eletrônico

1.3.2.1.- A arquitetura

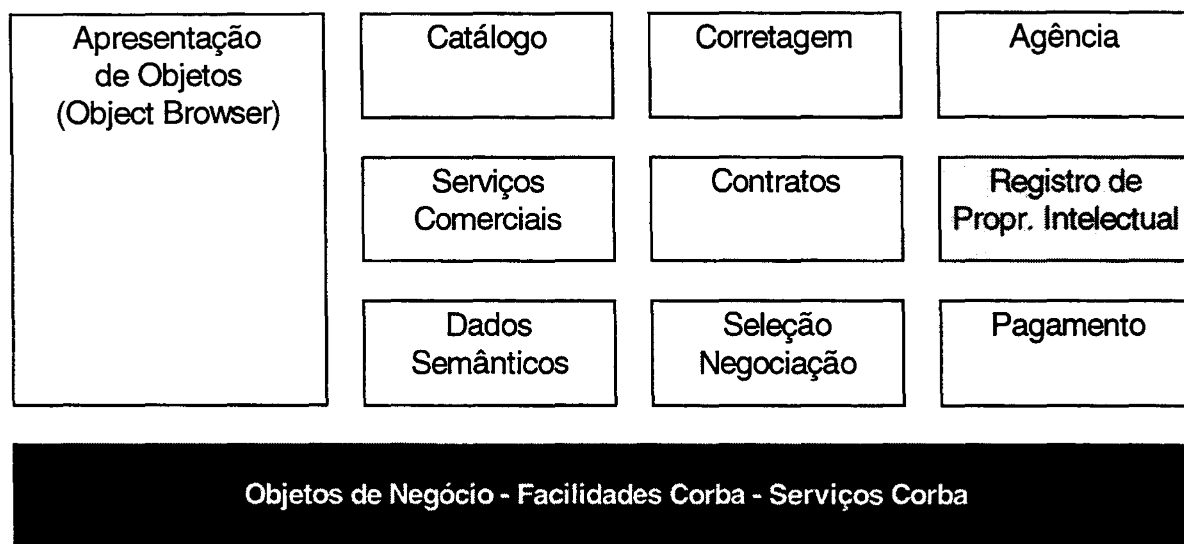


Figura 3 - Modelo de Referência para Comércio Eletrônico

Esta arquitetura, que podemos ver na Figura 3, foi apresentada num *Whitepaper* [25] conjunto do OMG e a associação CommerceNet, que foi preparado a partir das respostas enviadas pela indústria aos RFI 1 e 2 da EC DTF do OMG, e se encontra atualmente em discussão pela EC DTF.

O objetivo desta arquitetura é aprofundar a especificação dos requerimentos de alto nível para comércio eletrônico e mercados eletrônicos abertos. Ela é composta de três grupos principais denominados (1) **Serviços de comércio eletrônico** de baixo nível incluindo serviços de pagamento, facilidades de dados semânticos e serviços de negociação/seleção; (2) **Facilidades de comércio** suportando contratos, gerenciamento de serviços e facilidades referidas *a desktop*; e (3) **Facilidades da infra-estrutura de mercado** cobrindo catálogos, corretagem (*brokerage*) e agências.

A seguir detalhamos algumas destas facilidades que são de interesse para nossa pesquiça.

1.3.2.2.- Facilidades para Dados semânticos

O requerimento principal para comércio eletrônico é uma estrutura comum para a troca de dados semânticos descrevendo produtos, serviços, conteúdos e bens. O segundo requerimento se refere à necessidade de construir dinamicamente descrições de serviços para satisfazer uma demanda e a posterior modificação e extensão dessa descrição.

Dados semânticos são objetos de dados fornecidos por um provedor de serviços ou um consumidor e descrevem serviços oferecidos ou requeridos. Eles podem ser considerados como recipientes portáveis e persistentes para objetos de dados de qualquer tipo.

O principais requerimentos para um Objeto Semântico de Dados (*Semantic Data Object*) ou SDO é a sua capacidade para descrever qualquer serviço que possa ser oferecido num mercado eletrônico.

Os requerimentos para estas facilidades são:

1. Devem ser suficientemente gerais para incluir informação de qualquer tipo.
2. Deve ser possível agregar, modificar ou apagar informação dentro de um SDO.
3. Para cada objeto de informação dentro de um SDO deve ser possível determinar seu tipo e sua estrutura.

4. Deve ser possível navegar dentro de um SDO por iteração e por busca por tipos e informação.
5. Deve existir uma forma de converter a representação em memória de um SDO a uma forma persistente e vice-versa.
6. A forma persistente de um SDO deve ser portável e independente de plataforma.

Além dos requerimentos funcionais acima se requer a padronização de etiquetas específicas não-industriais que permitam a evolução de mercados abertos por meio do reconhecimento de termos comuns.

Estas facilidades guardam estreita relação com os Serviços CORBA de Nomes e Externalização.

Ainda que não tenha havido um outro documento oficial do OMG, posterior ao anteriormente citado, sobre esta arquitetura, na página *Web* da EC DTF [24] se apresentam algumas mudanças. Uma delas é a substituição do termo SDO pelo termo *Profile*, nesta Facilidade de Dados Semânticos que agora chama-se de Facilidade de *Profile* (Intercâmbio). Provavelmente isto foi feito para dar mais generalidade à facilidade dado que existe uma submissão feita ao OMG pela *System Software Association* (SSA)[31] propondo uma abordagem para este problema com o nome de *Semantic Data Object.*

1.3.2.3.- Facilidade de Catálogo

O catálogo é um objeto estruturado que pode ser inspecionado, apresentado e transferido através da rede. Ele provê uma interface que permite reordenar o catálogo de distintas maneiras, estender o catálogo agregando entradas, modificar ou apagar entradas existentes. Sua principal aplicação é conter informação sobre serviços e contratos.

A definição dada corresponde ao que chamamos **objeto catálogo**, para diferenciá-lo do **serviço catálogo** que é um serviço que fornece um objeto catálogo.

Os requerimentos básicos para um objeto catálogo são:

- Controle sobre reordenamento, agregação e remoção de itens.
- Uma interface formal para consulta.
- Meio para explicitar uma associação não repudiável de um catálogo com uma agência.

A infra-estrutura do catálogo deve suportar integração transparente e interoperabilidade entre catálogos de fontes diferentes. Diferentes implementações desta facilidade devem suportar a mesma interface.

Serviço de Assinatura: é uma facilidade genérica que permite aos consumidores efetuarem uma operação de assinatura sobre catálogos.

Esta facilidade tem relação estreita com a Facilidade de Gerenciamento de Serviço, com a Facilidade de Dados Semânticos e com os Serviços CORBA de Coleção e Consulta.

1.3.2.4.- Facilidade de Corretagem (Brokerage)

Seja qual for o papel de um participante num mercado eletrônico (cliente, consumidor, comerciante, fornecedor) ele terá que lidar com informação e desempenhará o papel de consumidor ou fornecedor de informação.

Os consumidores de informação necessitam buscar, recolher e filtrar informação; os fornecedores de informação necessitam enviar e rotear informação.

O propósito principal da facilidade de corretagem é permitir aos usuários (consumidores ou fornecedores de informação) serem mais expertos no tratamento de informação sobre serviços comerciais num mercado eletrônico global. Esta facilidade permite focalizar os requerimentos de informação nas fontes pertinentes.

Um outro requerimento de alguns clientes é a confidencialidade da informação requerida, o que significa anonimato nas transações comerciais. Também pode ser requerido por pessoas participantes numa transação a privacidade da mesma.

O método geral para obter privacidade e anonimato é a existência de uma terceira parte confiável que os garantisse. Esta será uma função da facilidade de corretagem.

A facilidade de corretagem introduz duas interfaces específicas denominadas ***recruiting*** e ***forwarding*** correspondendo aos conceitos de **busca** e **anúncio** .

Para explicitar melhor os requerimentos desta facilidade necessitamos definir:

habilidade em termos de serviços comerciais de um fornecedor e

interesse expresso em forma de uma especificação que pode descrever um tipo de serviço ou contrato.

Os requerimentos funcionais para esta facilidade são:

1. mecanismos através dos quais um consumidor possa expressar seu interesse em algo.
2. mecanismos através dos quais um fornecedor possa expressar sua habilidade para efetuar alguma coisa.
3. Mecanismos através dos quais possa ser estabelecida uma política de corretagem.
4. Mecanismos para suportar capacidades de *recruiting*, ou seja, casamento entre interesses declarados por um consumidor e habilidades declaradas por um fornecedor.
5. Mecanismos para suportar capacidades de *forwarding*, seja que for, para fornecedores ministrando descrição de suas habilidades para serem enviadas a consumidores registrados ou consumidores fornecendo descrição de interesses para serem enviadas a fornecedores registrados.

Requerimentos de assinatura: a facilidade de corretagem oferecerá serviços de assinatura baseados nos serviços de assinatura de catálogo.

Esta facilidade guarda relação com as já especificadas facilidades de Gerenciamento de Serviços, Catálogo e Assinatura e com os Serviços CORBA de Eventos, Consulta e *Trader*.

1.3.2.5.- Facilidade de Agência

Esta facilidade suporta requerimentos gerais para a padronização de um ponto de presença no mercado. Ela estabelece um ponto formal de acesso e uma interface pública para um participante num mercado eletrônico.

1.4.- Agentes Móveis

Apresentaremos alguns conceitos introdutórios para facilitar o entendimento da proposta. Os conceitos desta seção correspondem à especificação adotada pelo OMG a respeito da Facilidade de Interoperabilidade de Sistemas de Agentes Móveis (MASIF)[18].

1.4.1.- Definição e tipos

Um **agente** é um programa de computador que atua de forma autônoma em nome de uma pessoa ou organização. A maioria dos agentes hoje são programados em linguagens interpretadas, como por exemplo Tcl ou Java, por questões de portabilidade. Cada Agente tem seu próprio *thread* de execução de modo que ele pode efetuar tarefas por iniciativa própria.

Existem dois tipos de agentes. **Agentes estacionários** que são aqueles executados somente no sistema onde começa sua execução, se precisarem informação ou efetuarem interação com componentes de outro sistema, deverão usar algum mecanismo de comunicação como RPC. Um **agente móvel** não está confinado no sistema onde começa sua execução, ele possui a habilidade de transportar a si próprio de um sistema para outro numa rede. Esta habilidade de viajar permite a um agente móvel se movimentar para o sistema de agentes que contém o objeto com o qual deseja interagir.

1.4.2.- Conceitos principais

Quando um agente viaja, ele carrega consigo seu **estado** e seu código. O estado de um agente pode ser seu estado de execução ou os valores de seus atributos que permitam determinar o ponto a partir do qual sua execução vai continuar em seu destino.

A **autoridade** do agente identifica a pessoa ou organização em nome da qual o agente atua. O agente possui um **nome** para ser identificado em operações de gerenciamento e localização. Este nome é dado pela autoridade e ele é único dentro do escopo dessa autoridade.

A **localização** de um agente é o endereço de um "lugar"(*place*). Um lugar reside dentro de um sistema de agentes, portanto a localização do agente contém o nome do sistema de agentes donde o agente reside e o nome do lugar. Este conceito e os seguintes podem ser vistos na Figura 5.

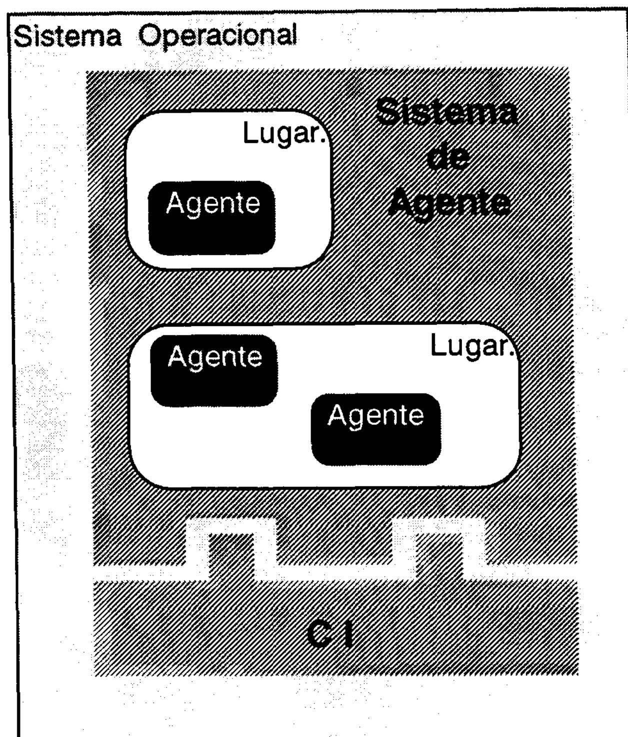


Figura 5 - Ambiente de Agentes

Um **sistema de agentes** é uma plataforma que pode criar, executar, interpretar, transferir e terminar um agente. Assim como o agente, ele está associado a uma autoridade. Ele é identificado unicamente pelo seu nome e seu endereço.

Um host pode conter mais de um sistema de agentes e cada sistema de agentes pode conter mais de um "lugar"(*place*).

Todas as comunicações entre sistemas de agentes ocorrem através de uma **Infra-estrutura de Comunicação (CI)**, a qual fornece também o serviço de nomes e de segurança.

Quando um agente se transfere a si próprio, ele viaja entre ambientes de execução denominados "lugares"(*places*). Como visto anteriormente um **lugar** é um contexto dentro de um sistema de agentes onde o agente pode ser executado. Este lugar pode fornecer funções tal como o controle de acesso.

Vários sistemas de agentes que possuem a mesma autoridade constituem o que se chama de **região**. Este conceito permite que mais de um sistema de agentes representem uma pessoa ou autoridade. O conceito de região pode corresponder ao de domínio em redes.

Um elemento importante que não pertence ao sistema de agentes, mais é anexo a ele, é o chamado ***Finder.*** Ele fornece operações para registrar, desregistrar e localizar agentes, lugares e sistemas de agentes.

A definição de agente que adotaremos para a nossa pesquisa é coincidente com a que já temos apresentado, " **agente** é um programa de computador que atua de forma autônoma em nome de uma pessoa ou organização" e em nosso casso o agente e um agente móvel.

Capítulo 2

Modelagem do Serviço de Comércio Eletrônico

2.1.- Preliminares

Como já foi introduzido no capítulo 1, este trabalho pretende implementar um agente móvel trabalhando em nome de um cliente para recolher informação sobre produtos ou serviços num ambiente de comércio eletrônico suportado pela plataforma CORBA. O agente fornecerá eventualmente facilidades para que o cliente possa fazer uma negociação do produto procurado.

O cenário proposto é esquematizado na Figura 6 e se descreve a seguir .

1.- Um cliente (objeto ou não) invocará o agente passando os dados do produto a procurar, os requerimentos de busca e eventualmente algum itinerário.

2.- O agente poderá invocar um objeto *Trader* para localizar os fornecedores da informação que ele procura, isto poderá ser feito em qualquer ponto do itinerário o que quer dizer que o itinerário poderá ser mudado durante a operação do agente.

3.- O agente se moverá para interagir com fornecedores, *Brokers* (Corretores) ou *Malls* (o equivalente eletrônico de nossos shoppings) onde poderá interagir com uma ou várias agências procurando a informação desejada.

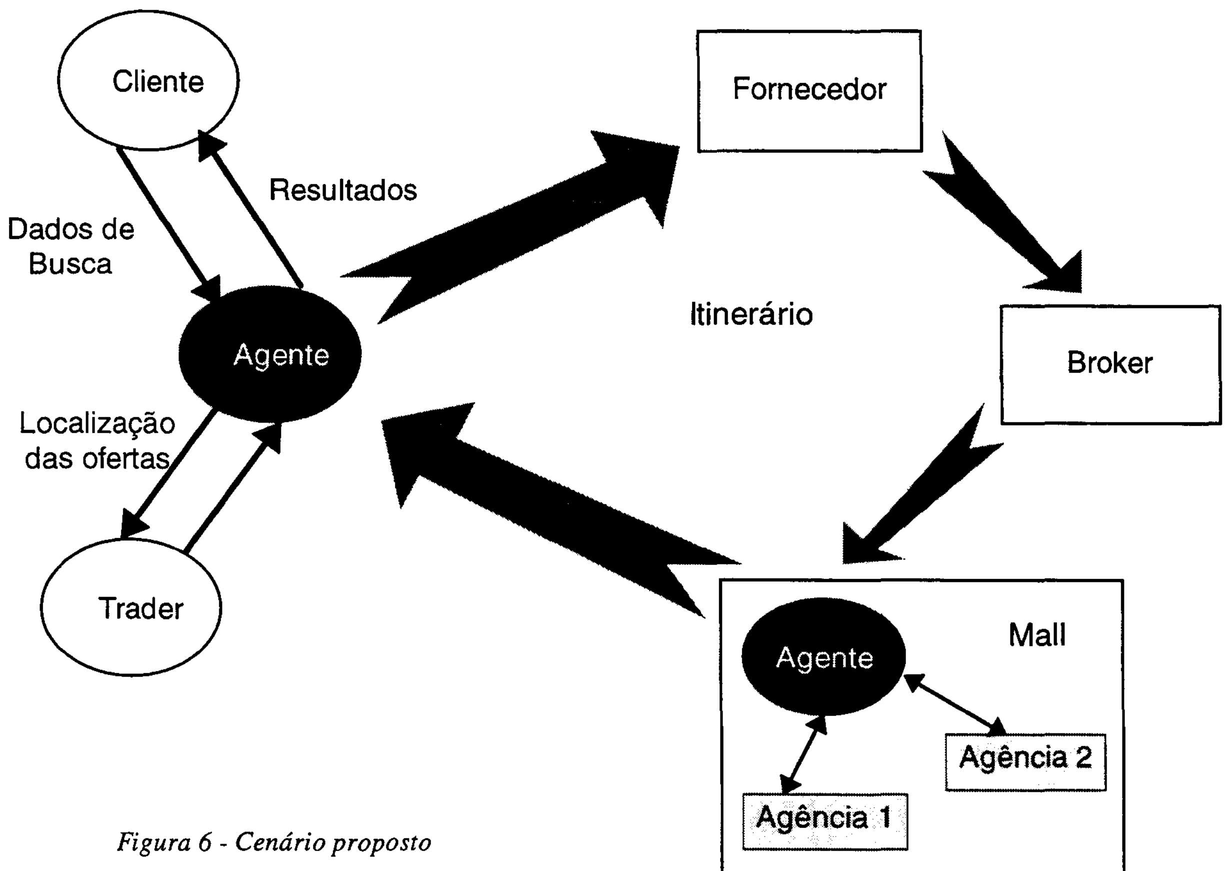


Figura 6 - Cenário proposto

4.- Eventualmente podem existir lugares que não tenham suporte para agentes, nesse caso a comunicação com eles será feita via rede de comunicação desde o ponto mais perto deles.

5.- Os resultados da operação do agente poderão ser retornados ao cliente parcialmente em qualquer ponto do itinerário ou quando o agente retornar ao ponto de partida.

A partir deste cenário surgem algumas questões que devem ser mais especificadas:

- O agente vai ser um objeto que como já foi dito vai interagir principalmente com objetos CORBA.

- Vamos considerar que os possíveis fornecedores da informação procurada (fornecedores, *brokers* ou agências) possuem seus dados na forma de um objeto catálogo, que fornece uma interface para consulta através de um ORB.
- Terá que ser definido um objeto cliente com uma interface que permita ao usuário um ambiente visual para invocar e gerenciar o agente e visualizar os resultados.

2.1.1.- Sistema de Agentes

Baseadas nas funcionalidades definidas para nosso sistema se estabeleceram algumas das características necessárias para o sistema de agentes, as quais se listam a seguir:

a) Interação com objetos CORBA

b) O agente deve carregar os dados e os resultados do seu trabalho no seu percurso

c) Definição das tarefas dentro do código do agente.

d) Deve ser possível alterar a lista de tarefas do agente em seu percurso

e) Capacidade de mobilidade em domínios diferentes

f) Capacidade de comunicação

g) Questões relativas a segurança (requisitos mínimos de identidade e autoridade do agente).

2.1.2.- Semântica dos dados

2.1.2.1.- Transporte: SDOs

Com relação ao transporte dos dados, necessitamos uma estrutura que possibilite o envio de dados entre duas ou mais partes e a posterior identificação desses dados sem que exista prévio acordo entre essas partes sobre a identificação desses dados.

Escolhemos utilizar, com algumas modificações, os *Semantic Data Objects* (SDOs) apresentados na submissão sobre *Bussines Object Facility* feita ao OMG pela *System Software Associates Inc.* (SSA)[31].

Os SDOs são objetos que carregam a semântica do dado que eles contêm e apresentam métodos que permitem manipulá-los.

Um SDO possui as seguintes propriedades:

label: é uma etiqueta que identifica o dado contido.

value: conteúdo correspondente ao dado identificado pelo *label*.

type: indicador do tipo de dado (inteiro, string, double, etc.)

container: uma estrutura que contém outros SDOs associados a este.

A Figura seguinte mostra um exemplo de SDO que representa a resposta obtida numa seleção de empregados:

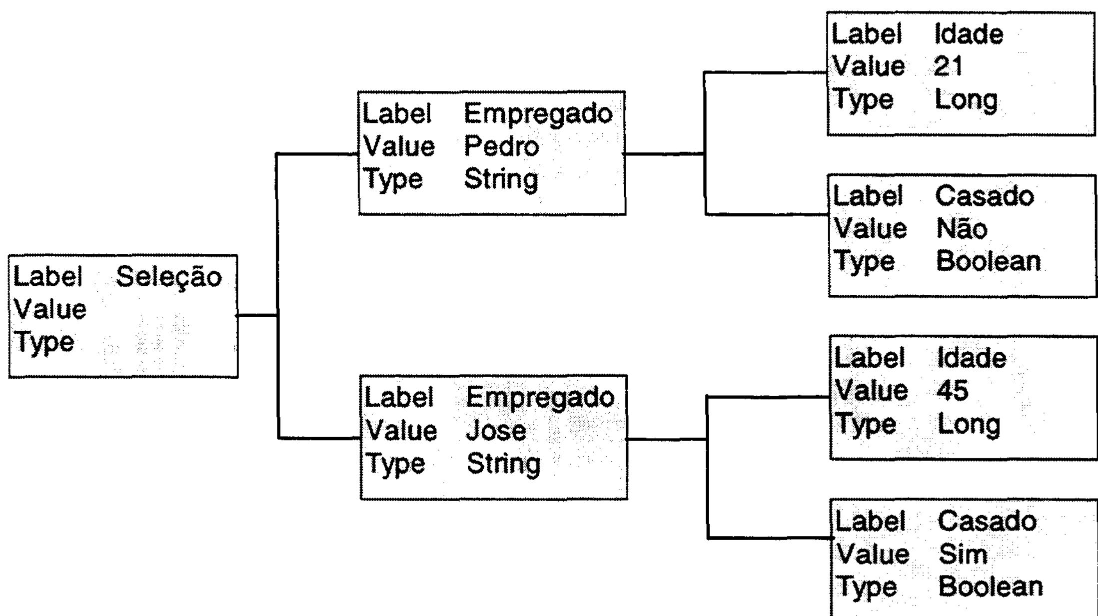


Figura 7 - SDO resposta de uma seleção de empregados

Os SDOs possuem métodos que permitem:

- navegar pela estrutura de SDOs em busca de um determinado *Label* ou *Value*

- obter um determinado filho de um SDO
- retirar ou eliminar um SDO da estrutura
- inserir um determinado SDO na estrutura
- criar um novo SDO num lugar determinado da estrutura

Estos métodos serão detalhados no capítulo sobre a implementação na seção 3.2.

Para a serialização dos SDOs para transporte se utilizaram, com algumas modificações, duas estruturas propostas na citada submissão da SSA cuja descrição em IDL podemos ver a seguir:

```
typedef struct SerialisedSdo
{
   short      version;
   short      type;
   sequence<SerializedSdoElement> sdoList;
{;
```

`version` e `type` estão relacionados com o formato usado para serialização

`sdoList` é uma seqüência que contém estruturas SerializedSdoElement que se descreve abaixo.

```
typedef struct SerializedSdoElement
{
   string     label;
   string     value;
   string     type;
   long       parentIndex;
   octet      flag;
};
```

O dado chave desta estrutura é o parentIndex que representa o índice dentro da seqüência de estruturas SerializedSdoElement, na estrutura SerializedSdo associada, do SerializedSdoElement que contém este SerializedSdoElement. Ou seja, a ordem dentro da estrutura SerializedSdo do pai deste SDO serializado.

O motivo desta escolha, foi o fato de que, como se verá mais à frente, o SDO serializado é a forma em que o catálogo entrega ao agente o resultado de uma consulta, e pelo fato do catálogo poder ser implementado em qualquer linguagem, necessitamos uma estrutura simples que possa ser construída e reconstruída usando qualquer linguagem.

Para permitir a visualização por parte de um usuário do conteúdo e da estrutura de um SDO usamos o processo de externalização que transforma um SDO numa string com a seguinte forma:

```
[Seleção,]""
{
        [Empregado,String]"Pedro"
        {
                [Idade,Long]"21"
                [Casado,Boolean]"Não"
        {
        [Empregado,String]"Juan"
        {
                [Idade,Long]"45"
                [Casado,Boolean]"Sim"
        {
}
```

Esta externalização corresponde ao SDO da Figura 7.

Todos os processos que transformam um SDO em suas formas serializadas ou externalizadas e permitem recuperar o SDO a partir das formas citadas serão agrupados numa classe nomeada **Convert**.

Na Figura a seguir podemos visualizar o modelo de objetos do **SDO** com a classe associada **Convert,** um maior detalhe destas classes e seus métodos correspondentes será apresentado na seção 3.2. Todos os modelos de objetos neste capítulo seguem a notação gráfica da Técnica de Modelagem de Rumbaugh (OMT)[30].

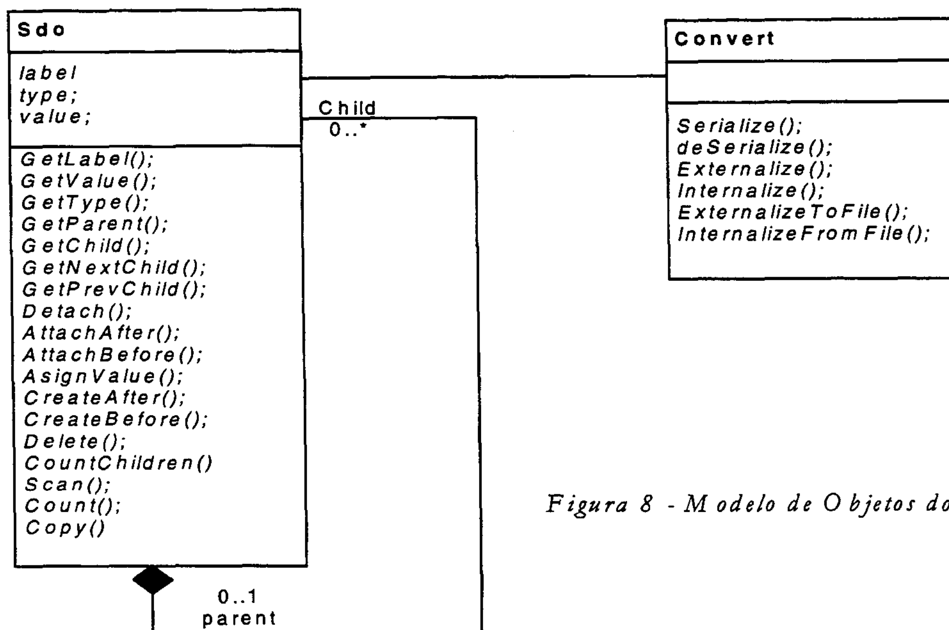


Figura 8 - Modelo de Objetos do SDO

Uma questão a ser considerada é a necessidade da padronização das etiquetas (*labels*) que identificam o conteúdo dos SDOs. Isto implica na existência de um repositório de etiquetas de modo que quando uma etiqueta é usada possa ser verificada a sua existência, ou possa ser registrada quando é criada, evitando duplicações. Se não existir esta padronização, a utilidade dos SDOs nesta tarefa de intercâmbio de informação é muito limitada. Em geral a falta de padronização nesta área é o principal obstáculo para a implementação de catálogos interoperáveis[4].

2.1.2.2.- Armazenamento

Num primeiro momento achamos que poderíamos usar a estrutura dos SDOs para armazenar os dados do catálogo, o que significaria um casamento perfeito com a estrutura de transporte.

Isto implicava em dispor de um sistema de banco de dados orientado a objeto para fornecer o gerenciamento desses dados. Além disso a estrutura dos SDOs apresenta uma certa

complexidade para a implementação de métodos para reordenar o catálogo, dada a sua estrutura em árvore.

Um outro elemento determinante é o fato da perda de generalidade de nossa abordagem ao impor uma determinada estrutura para o catálogo.

Visto o desenvolvimento por parte da Sun Microsystems do *Java Database Conectivity* (JDBC) que permite que as aplicações Java manipulem dados contidos em bases de dados quaisquer e dada a liberdade que isto significa para quem implementa o catálogo, e comprovada a flexibilidade que esta abordagem permite em soluções reais [28, 26], decidimos adotá-la na nossa modelagem.

JDBC consiste de duas camadas. A camada superior é o JDBC *Application Programming Interface* (API) que fornece ao programador os métodos para manipular a base de dados. Esta API se comunica com o JDBC *manager driver* API, enviando várias sentenças SQL. Este gerente se comunicará (transparentemente para o programador) com vários *drivers*, fornecidos por terceiros,

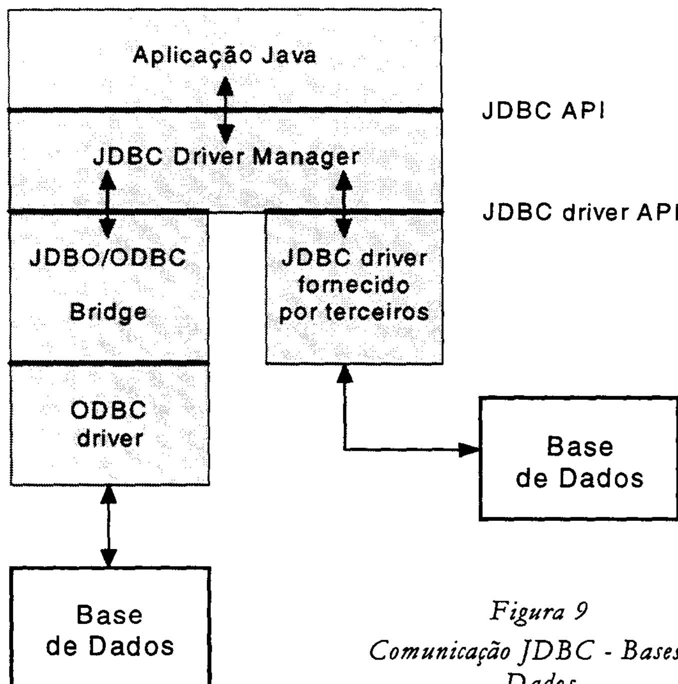


Figura 9
Comunicação JDBC - Bases de Dados

que atualmente se conectam às bases de dados e retornam informação das consultas ou efetuam ações especificadas[11]. Na Figura 9 pode-se visualizar o descrito acima.

Em suma, um programador pode escrever aplicações Java para acessar bases de dados, usando sentenças SQL padrão seguindo as convenções da linguagem Java.

Com esta abordagem deixamos para o fornecedor do catálogo a liberdade de escolha para o armazenamento dos seus dados e somente forneceremos a definição da interface IDL que o catálogo deve implementar para sua interação com o agente.

2.1.3.- Semântica das Consultas

Neste item definiremos a estrutura dos dados fornecidos pelo usuário e a forma em que o agente faz as consultas.

a) Dados fornecidos pelo usuário: os dados fornecidos pelo usuário devem conter a informação suficiente para identificar os itens procurados e por sua vez permitir uma certa flexibilidade podendo-se fornecer variações para algumas propriedades.

O usuário fornecerá:

Tipo do produto, que permitirá identificar o catálogo que se deve procurar. Ex.: carro, livro, etc.

Propriedades que identifiquem o produto, a maioria delas com valores fixos (tomando o exemplo de carros, Marca = Volkswagen e Modelo = Gol, etc.); uma, para a qual podem-se fornecer até três valores (Ex.: cor = verde, branco ou preto) e uma outra com intervalo contínuo de possibilidades (Ex.: ano entre 1996 e 1998).

Um intervalo para o preço desejado. Ex.: preço < 12.000.

As condições expressas acima são capturadas pelo modelo de objetos da Figura 10. Neste modelo de objetos temos a classe **UserData** que contém os atributos *product* e *price*, sendo uma agregação das classes **FixProp, EscPro, ContProp** e **Bound**. A classe **Bound** contém dois

parâmetros *up* e *bt* que representam os limites superior e inferior de uma quantidade, no caso da agregação com relação a UserData representam os limites para o parâmetro preço.

A classe **FixProp** representa as propriedades com valores fixos, que podem ser mais de uma, e que têm um parâmetro *priority* que indica se essa propriedade pode ser deixada de lado na procura e qual é o seu peso com relação às outras propriedades com valores fixos. A classe **EscProp** corresponde à propriedade da qual se podem fornecer até três valores e contém um parâmetro *value* do tipo *array*. A classe **ContProp** contém os dados para a propriedade com intervalo contínuo, por isso ela tem associada a classe **Bound** que fornece os limites para o citado intervalo. Estas três classes tem como agregado a classe **PropId** que fornece os parâmetros *name* e *type* para cada uma delas.

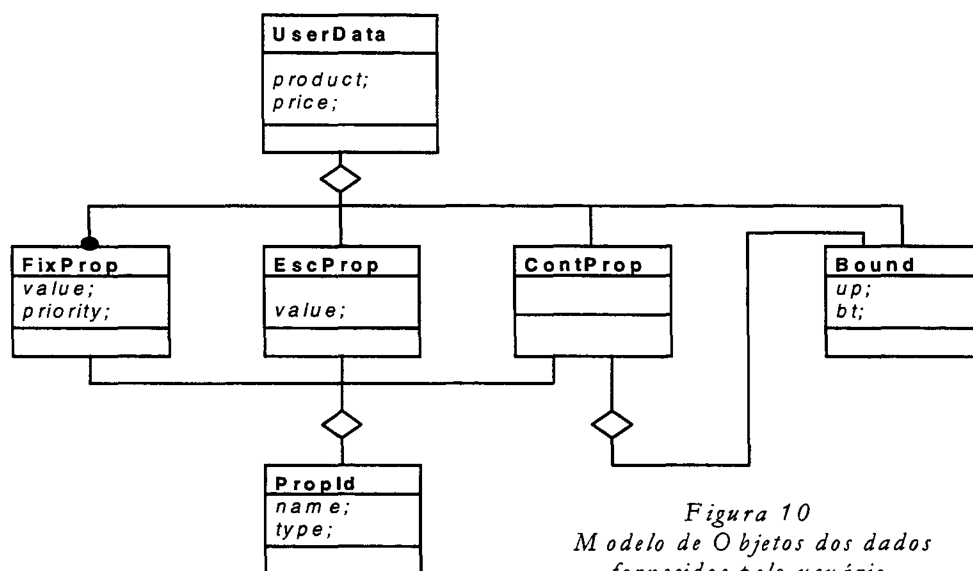


Figura 10
Modelo de Objetos dos dados fornecidos pelo usuário

b) Modularização da decisão sobre consultas: o agente consta de um módulo de decisão que alimentado com os dados fornecidos pelo usuário e baseado na sua lógica interna aciona um outro módulo encarregado de efetuar as consultas sobre o catálogo. As respostas a estas consultas serão analisadas pelo módulo de decisão o qual produzirá os resultados ou eventualmente ordenará novas consultas. A Figura 11 mostra o processo descrito acima.

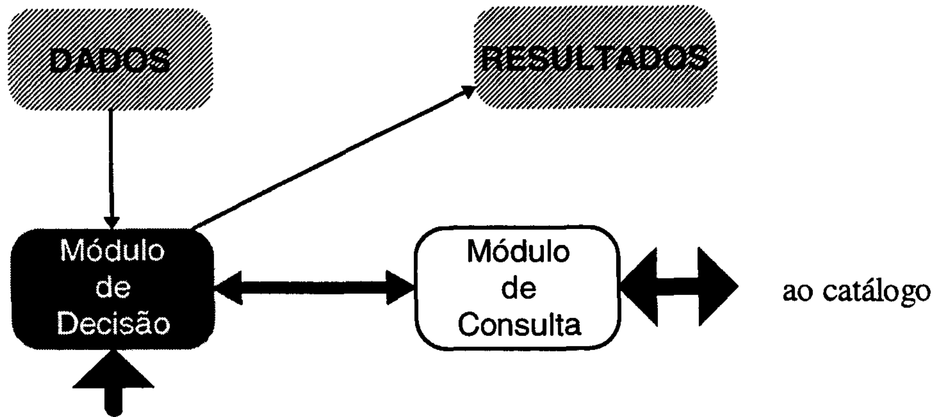


Figura 11 - Modularização do Agente

O motivo desta abordagem é deixar aberta a possibilidade do módulo de decisão poder ser trocado, permitindo que módulos com tipo distinto de "inteligência" ou com funcionalidades diferentes possam ser utilizados.

O fato do módulo de consulta estar separado do módulo de decisão permite também que o agente possa ter mais de um módulo de consulta, os quais podem ser utilizados para interagir com ORBs de versões ou fabricantes diferentes ou com catálogos não implementados através de ORBs.

2.2.- Desenvolvimento do modelo de objetos

2.2.1.- Modelo geral

Foi desenvolvido o modelo geral do sistema, indicando as interações entre seus componentes, que pode-se visualizar na Figura 12.

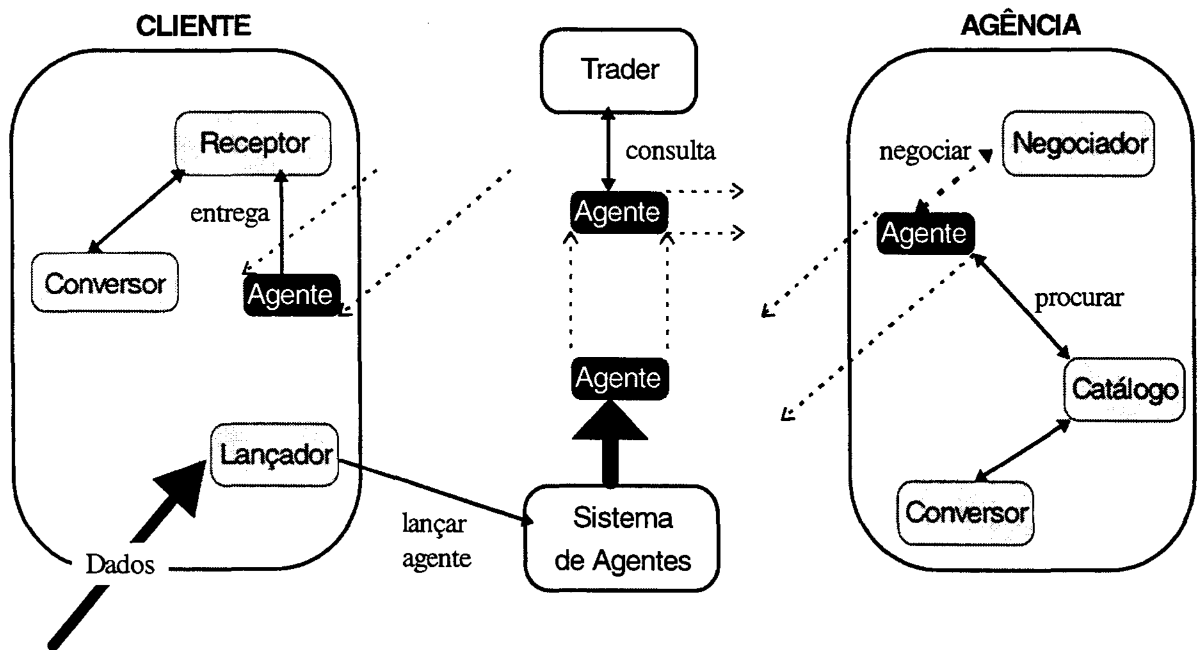


Figura 12 - Modelo geral

Nosso sistema supõe a interação de 4 subsistemas: Cliente, Sistema de Agentes, Trader e Agência, a seguir descrevemos cada um deles.

2.2.1.1.- Cliente

Composto por três objetos que fornecem os serviços que são necessários para quem envia o agente e que são:

Lançador

Proporciona a interface entre o sistema e o usuário e permite:

- descrever o produto desejado;
- fornecer variações para suas propriedades;
- estabelecer o itinerário;

- lançar o agente interagindo com o sistema de agentes.

Conversor

Este objeto é o encarregado de receber um SDO serializado ou como objeto e transformá-lo para ser utilizado como objeto (desserialização) ou para ser transportado (serialização).

Receptor

Seu objetivo é receber do agente os dados coletados no seu percurso e apresentá-los ao usuário. Este objeto interage com o Conversor para desserializar os SDOs que recebe do agente como resultado do seu trabalho.

Na Figura 13 podemos ver o modelo de objetos do cliente.

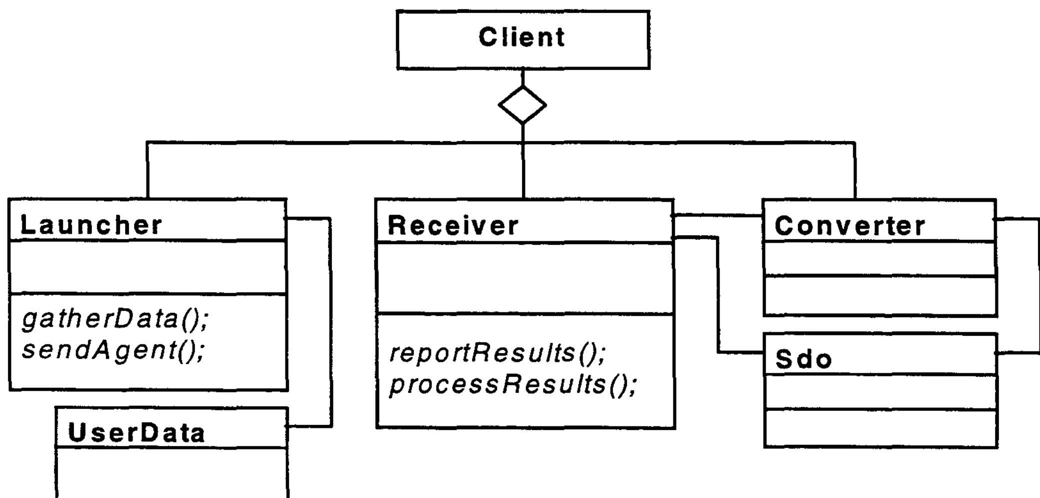


Figura 13 - Modelo de Objetos do Cliente

2.2.1.2.- Trader

Este subsistema pertence a um sistema CORBA e tem por objetivo fornecer referências de interfaces onde podem ser localizadas ofertas de serviços, pode-se tratar de um *Trader* só o uma

Federação de *Traders*. Será consultado pelo agente para obter as referências dos catálogos para achar os produtos que procura.

2.2.1.3.- Sistema de Agentes

É composto principalmente por uma "*Engine*" que é o próprio sistema de agentes e que gerencia o agente e os lugares em cada destino do agente. Na seção 2.1.1 já foram definidas as características necessárias e no capítulo 3, correspondente à implementação, forneceremos uma descrição do sistema de agentes utilizado.

2.2.1.4.- Agência

Este subsistema é uma versão simplificada da Facilidade de Agência do Modelo de referência para Comércio Eletrônico apresentado no capítulo 1. Seu objetivo é ser um ponto de presença no mercado. É composta por três objetos, a saber:

Catálogo

Este objeto armazena ofertas de produtos e fornece as interfaces necessárias para efetuar consultas sobre esses produtos e para gerenciar o catálogo. Receberá do agente consultas sobre os produtos e interagirá com o Conversor para serializar as respostas às consultas.

Conversor

Tem as mesmas funcionalidades do Conversor do subsistema cliente.

Negociador

Este objeto fornecerá as funcionalidades necessárias para efetuar negociações sobre produtos, ou seja uma vez que o Cliente tenha escolhido o produto baseado na informação coletada pelo agente, poderá utilizar a interface fornecida por este objeto para negociar com o fornecedor as condições de venda e de entrega do produto. A referência a este objeto negociador será fornecida pelo objeto catálogo quando da consulta dos produtos. O objeto negociador deverá

fornecer ao menos um método *negotiate()* que o cliente invocará passando como parâmetros o código do produto e eventualmente condições de compra que ele deseja.

Na Figura 14 podemos ver o modelo de objetos da Agência.

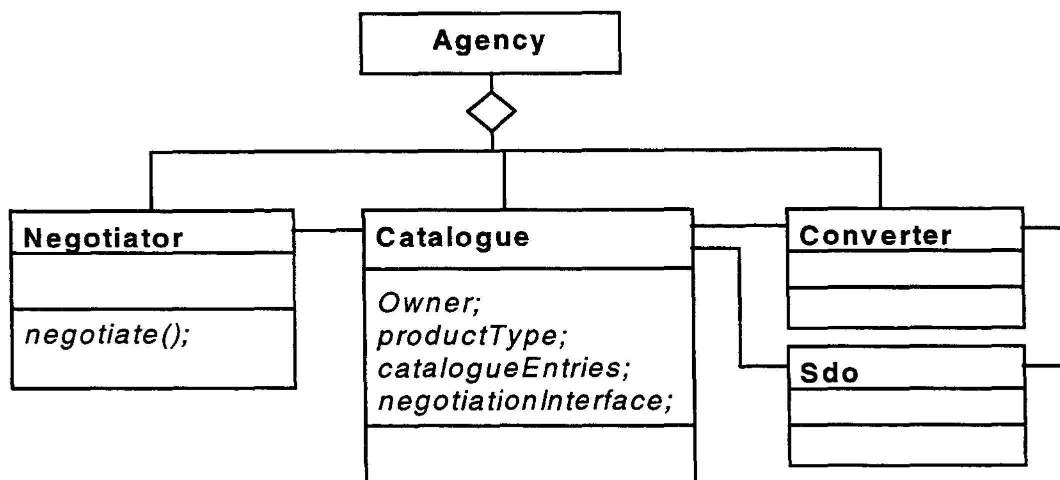


Figura 14 - Modelo de Objetos da Agência

2.2.2.- Catálogo

Levando em conta os requerimentos básicos para a Facilidade de Catálogo apresentados na seção 1.3.2.3, o objeto catálogo possui duas interfaces: a interface *Loockup* que fornecerá os métodos que implementam uma interface formal de consulta ou seja as funcionalidades necessárias para o usuário do catálogo descobrir a identificação do catálogo e buscar itens, e a interface *Admin* com métodos para controle sobre reordenação, agregação e remoção de itens, todas elas tarefas correspondentes à administração do catálogo.

Do ponto de vista de uma especificação CORBA nosso catálogo é uma implementação de objeto contendo dois objetos denominados *Loockup* e *Admin*, com se pode ver na Figura 15.

O catálogo possui **atributos** que o identificam e que são os seguintes:

- **Owner:** identifica o responsável pela informação contida no catálogo, ou seja, quem fornece os produtos que o catálogo contém.

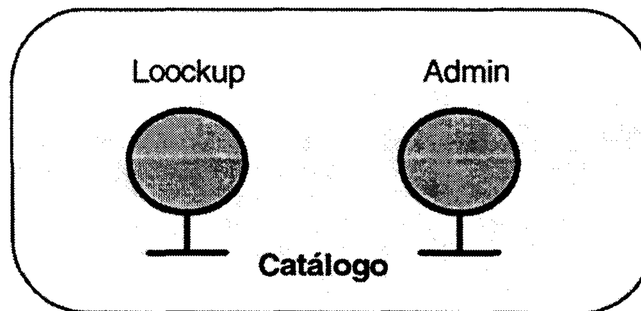


Figura 15 - Esquema do catálogo

- **Product_type:** identifica o tipo de produto dos quais o catálogo contém dados. Ex.: carros, livros, ferramentas, etc.

- **Catalogue_entries:** indica a quantidade de itens presentes no catálogo, esta é uma medida do tamanho do catálogo e pode ser usada para decidir se o catálogo será requisitado ou se fará uma consulta remota.

-**Negotiation_interface:** referencia uma interface donde se pode efetuar pedido e negociação sobre os produtos contidos no catálogo.

O catálogo poderá conter mais de uma lista de preços, nesse caso haverá um conjunto destas propriedades para cada uma das listas.

A seguir detalharemos as interfaces dos dois objetos que compõem o catálogo:

2.2.2.1.- Interface Loockup

Esta interface fornece ao usuário os métodos necessários para pesquisar o catálogo e obter informação sobre o mesmo, identificação e estrutura, e sobre os itens que ele armazena. Os métodos desta interface são os seguintes:

getTables

Função: fornece uma listagem das listas de preços contidas no catálogo.

Parâmetros: nenhum

Retorno: seqüência contendo os nomes das listas.

getId

Função: fornece a identificação de uma lista contida no catálogo

Parâmetros: nome da lista da qual se requer identificação

Retorno: estrutura contendo os valores dos atributos da lista indicada acima.

getSquema

Função: fornece a estrutura de uma determinada lista dentro do catálogo indicando o nome e tipo de cada uma das propriedades dos itens contidos na mesma. A ordem em que as propriedades são entregues pode indicar como foi construída a chave para ordenar o catálogo. Fazendo uma analogia com Banco de Dados seria fornecido o esquema de uma tabela dentro do banco indicando os nomes e tipos de cada um dos campos.

Parâmetros: nome da lista

Retorno: seqüência de pares (nome da propriedade, tipo).

query

Função: busca na lista especificada um item ou vários cujas propriedades casem com os valores fornecidos para as mesmas, retornando os valores de todas as propriedades que

esses itens possuem na lista. Eventualmente o usuário poderá não fornecer valor para alguma propriedade, neste caso fornecerá uma listagem dos itens que casam com as outras condições prescritas. Pode-se indicar a ordem em que as propriedades devem aparecer na resposta.

Parâmetros: são três : (1) o nome da lista na qual se deseja procurar; (2) uma seqüência de valores correspondentes às propriedades fornecidas na invocação **getSquema,** e na mesma ordem, que contém os valores desejados para as propriedades; (3) uma seqüência contendo os nomes das propriedades na ordem em que se deseja que estas sejam visualizadas na resposta.

Retorno: SDO serializado contendo a resposta à busca.

execQuery

Função: recebe uma sentença SQL de busca e a executa sobre o banco de dados que constitui o catálogo.

Parâmetros: são dois : (1) o nome da lista na qual se deseja procurar; (2) variável de tipo *string* contendo a sentença SQL.

Retorno: SDO serializado contendo a resposta à busca.

2.2.2.2.- Interface Admin

Esta interface é fornecida para ser utilizada pelo responsável do catálogo para suas tarefas de administração do mesmo. A seguir detalhamos seus métodos:

setId

Função: permite preencher os atributos de uma lista no catálogo.

Parâmetros: são dois: (1) nome da lista à qual se quer preencher a identificação; (2) estrutura contendo os valores dos atributos da lista indicada.

Retorno: valor booleano indicando se a operação teve sucesso ou não.

add

Função: adiciona um item a uma lista no catálogo no lugar correspondente.

Parâmetros: são dois : (1) o nome da tabela na qual se deseja adicionar; (2) seqüência de valores correspondentes às propriedades fornecidas na invocação ao método *getSchema,* na interface *Loockup*, e na mesma ordem, que contem os valores correspondentes ao item a adicionar.

Retorno: valor booleano indicando se a operação teve sucesso ou não.

delete

Função: efetua uma busca de um item ou vários itens de acordo com os parâmetros e os elimina de uma lista no catálogo.

Parâmetros: são dois : (1) o nome da tabela na qual se deseja operar; (2) seqüência de valores correspondentes às propriedades fornecidas na invocação ao método *getSchema,* na interface *Loockup*, e na mesma ordem, que contem os valores correspondentes do item ou itens a eliminar.

Retorno: valor numérico indicando quantas filas da lista foram eliminadas na operação.

update

Função: efetua uma busca de um item numa lista e troca os valores de suas propriedades de acordo com os parâmetros fornecidos.

Parâmetros: são três : (1) o nome da tabela na qual se deseja operar; (2) seqüência de valores correspondentes às propriedades fornecidas na invocação ao método *getSchema,* na interface *Loockup*, e na mesma ordem, que contem os valores correspondentes dos itens a buscar; (3) seqüência de valores correspondentes às novas propriedades .

Retorno: valor numérico indicando quantas filas da lista foram processadas na operação.

execUpdate

Função: recebe uma sentença SQL de add, delete ou update, e a executa sobre uma lista do banco de dados que constitui o catálogo.

Parâmetros: são dois : (1) o nome da lista na qual se deseja operar; (2) variável de tipo *string* contendo a sentença SQL.

Retorno: valor numérico indicando quantas filas da lista foram processadas na operação.

index

Função: ordena uma lista do catálogo considerando como chave a seqüência fornecida como parâmetro.

Parâmetros: são três : (1) o nome da lista na qual se deseja operar; (2) nome do índice; (3) seqüência contendo os nomes das propriedades que serão concatenadas para formar a chave de ordenação.

Retorno: valor booleano indicando se a operação teve sucesso ou não.

No apêndice A é apresentada a especificação IDL do catálogo e na Figura 16 podemos ver

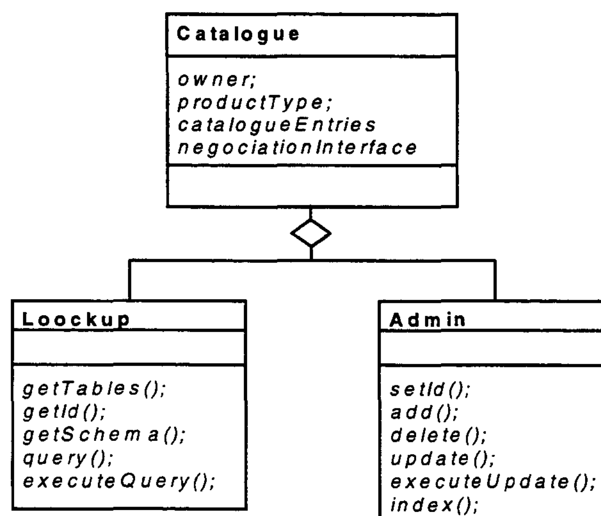


Figura 16 - Modelo de Objetos do Catálogo

o modelo de objetos.

2.2.3.- Agente

O agente não é um objeto CORBA mas como já foi dito na seção 2.1 ele interage com objetos CORBA. Levando em conta as funcionalidades detalhadas para o Sistema de Agentes na seção 2.1.1, as quais estabelecem critérios para a modelagem do agente, podemos desenhar um esquema do agente que pode-se ver na Figura 17. Nesta Figura a referência ao armazenamento somente indica os dados que o agente carrega consigo.

A Lógica de Busca e Casamento contém os módulos mencionados na seção 2.1.3 b.

O módulo de negociação fornecerá funcionalidades para que o cliente possa negociar com a agência sobre os produtos procurados pelo agente. Este módulo não inclui, e nem nenhum outro do agente, funcionalidades de pagamento.

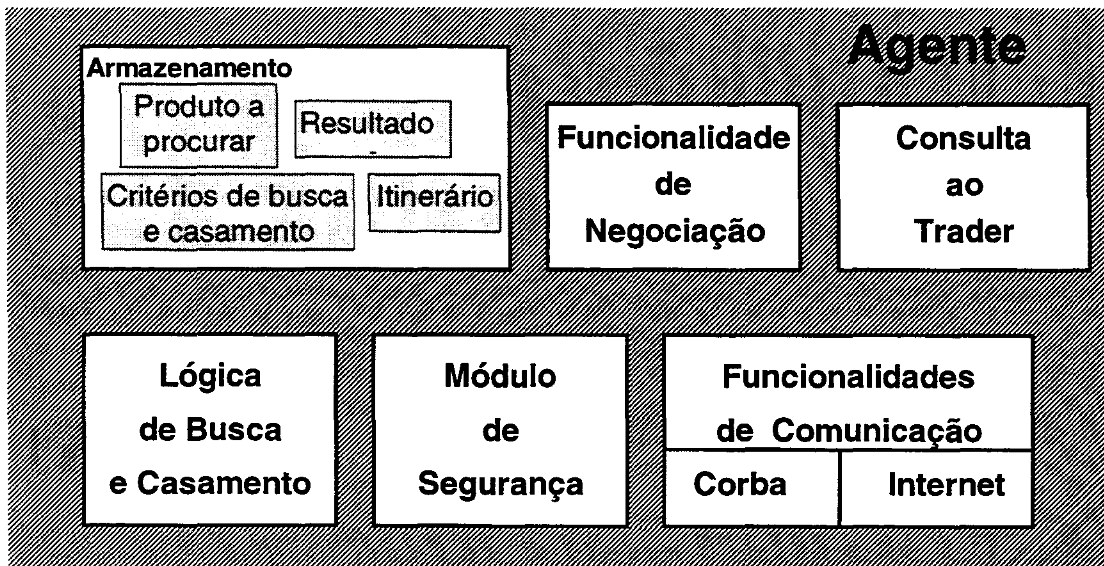


Figura 17 - Esquema do Agente

Consulta ao *Trader* indica um módulo com os métodos de acesso a um *Trader.* A sua modularização tem a ver com o fato de que, ainda que o *Trader* possua uma interface IDL padronizada pelo OMG, pode ser, como no caso do catálogo, que ele esteja presente em ORBs de diferentes fabricantes ou versões.

Com relação às Funcionalidades de Comunicação consideraremos que elas são fornecidas pelo Sistema de Agentes, fato hoje que pode ser comprovado na maioria dos sistemas proprietários existentes.

Não abordaremos neste trabalho questões de segurança do cenário que estamos modelando, não pelo fato de não considerá-las importantes, mas sim porque consideramos que essas questões sozinhas podem ser motivo de outro trabalho. Porém o modelo proposto não inviabiliza os modelos ou mecanismos de segurança desenvolvidos para Agentes Móveis ou Comércio Eletrônico. Alguns comentários a esse respeito serão feitos no capítulo da implementação com base nos testes efetuados.

Na Figura 18 apresentamos o modelo de objetos do agente, *Qagent.* Dado que utilizaremos um sistema de agentes já existente, nosso agente herdará dele a definição do agente e de acordo com o já exposto será composto por: um módulo de decisão; um ou mais módulos de consulta; um módulo de negociação e um módulo de interação com o *Trader.* O agente carrega consigo as tarefas, o itinerário e contém os dados fornecidos pelo usuário. A relação entre *Qagent e UserData* indica esta relação dos dados com o agente.

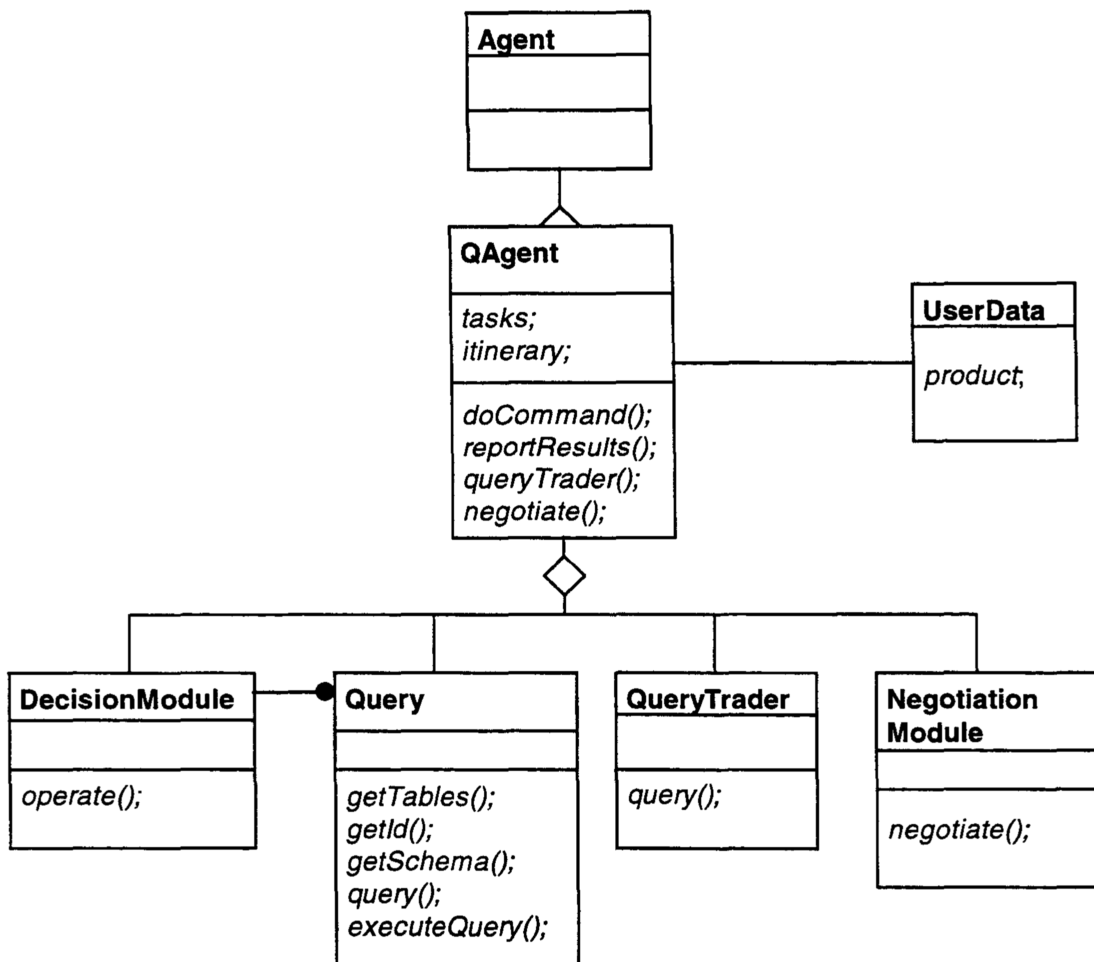


Figura 18 - Modelo de Objetos do Agente

Capítulo 3

Aspectos da Implementação

3.1.- Considerações gerais

Esta implementação compreende o desenvolvimento de um agente móvel com suas interfaces com o usuário, as classes que manipulam os SDOs, e a duas interfaces do catálogo utilizando um banco de dados para armazenamento.

A implementação foi desenvolvida utilizando a linguagem Java, principalmente por ser a linguagem de desenvolvimento dos Sistemas de Agentes, e o ORB OrbixWeb da Iona pelo fato de estar desenvolvido em Java, o qual possibilita a interação como o agente.

No início foi desenvolvida a classe que implementa os SDOs e uma classe auxiliar que implementa, entre outras, as funcionalidades descritas para o objeto Conversor no capítulo anterior.

Efetuou-se a especificação em IDL do catálogo como um módulo com suas duas interfaces, Loockup e Admin, e as estruturas auxiliares necessárias, e a partir dela efetuamos a implementação do mesmo utilizando a API Java DataBase Connectivity (JDBC) [11] e o Sistema

Gerenciador de Banco de Dados (SGBD) MiniSQL (mSQL)[12], conectados através de um *driver* escrito em Java.

Analisando as descrições de diversos sistemas de agentes fizemos a escolha de utilizar o Odyssey da General Magic[7] pelas razões que se expõem na seção 3.4.1 .

O Agente carrega consigo as classes necessárias para a interação com o catálogo, que foram geradas pela compilação da especificação IDL, e as classes que implementam os módulos mencionados na seção 2.1.3 b. Não foram implementadas as funcionalidades de Negociação e de Consulta ao Trader.

O ambiente utilizado foi Unix mas dado que a implementação é feita em Java e utiliza um ORB, não existem restrições quanto a sistema operacional.

Nas seções a seguir faremos uma descrição das partes da implementação.

3.2.- Semantic Data Objects

A implementação dos SDOs consta de duas classes, a classe **Sdo,** que implementa o SDO e possui métodos que permitem navegar por um SDO e modificar a sua estrutura, e a classe **Convert** com métodos para transformar um SDO.

3.2.1 Classe Sdo

A classe **Sdo** é a que implementa os Objetos Semânticos de Dados, SDOs. Nesta classe os atributos *label, value e type* estão contidos em variáveis de tipo *String*, e o *container*, que contém os filhos do SDO, está implementado usando a classe *java.util.Vector* que permite armazenar qualquer objeto e apresenta métodos para manipular seu conteúdo.

Esta classe possui os seguintes métodos:

(1) Métodos que permitem obter o conteúdo de um SDO

⇒ **String GetLabel(),** para obter o label;

⇒ **String GetValue(),** para obter o valor do dado contido;

⇒ **String GetType(),** para obter o tipo de dado;

⇒ **Sdo GetParent(),** que fornece a referência de objeto do SDO pai deste SDO;

⇒ **Sdo GetChild(String label, int position),** permite obter a referência a um filho deste SDO com um determinado *label* numa determinada posição, se houver. Variações deste método permitem especificar também o *value* ou somente o *label* e o *value.*

⇒ **Sdo GetNextChild(Sdo currChild, String label);**

⇒ **Sdo GetPrevChild(Sdo currChild, String label),** estes dois métodos permitem, a partir da referência de um determinado filho(currChild), obter a referência de um outro filho, posterior ou anterior respetivamente, com um determinado *label.*

(2) Métodos que modificam a estrutura e o conteúdo do SDO

⇒ **void Detach(),** o SDO é tirado da estrutura que o contém e se torna independente.

⇒ **void AttachBefore(Sdo parent, Sdo currChild);**

⇒ **void AttachAfter(Sdo parent, Sdo currChild),** nestes dois métodos o SDO é colocado como filho de *parent,* antes ou depois, respectivamente, na ordem relacionada com *currChild.*

⇒ **void AssignValue(Sdo sdo),** permite dar a este SDO o *value* e o *type* de um outro.

⇒ **Sdo CreateBefore(String label, String value, String type, Sdo currChild);**

⇒ **Sdo CreateAfter(String label, String value, String type, Sdo currChild),** estes métodos criam um SDO como filho do atual e o colocam antes ou depois, respectivamente, de *currChild* na estrutura de filhos.

⇒ **void Delete(),** elimina este SDO da estrutura.

(3) Métodos que permitem navegar e obter dados da estrutura

⇒ **int CountChildren(String label),** retorna a quantidade de filhos deste SDO com um determinado *label.*

⇒ **Sdo Scan(String label, String value, Sdo start),** vasculha este SDO, em profundidade, procurando um SDO com um determinado *label* e *value.* O parâmetro *start* pode indicar um outro SDO, distinto do atual, para começar a procura. Variações deste método permitem buscar um *label,* ou um *label* e um *value.*

⇒ **int Count(Sdo start),** vasculha este SDO, em profundidade, e retorna a quantidade de SDOs contidos na estrutura embaixo dele.

Existe também um método **Sdo Copy()** que fornece uma cópia do atual SDO, sem os filhos.

3.2.2.- Classe Convert

Como já dissemos no capítulo 2 a classe Convert fornece os métodos necessários para transformar um SDO na sua forma serializada para transporte, utilizando a estrutura também apresentada no citado capítulo, ou externalizada para compreensão do usuário, e para obter o SDO de volta a partir de alguma dessas formas. Seus métodos são:

⇒ **SerializedSdo Serialize(Sdo sdo)**, para serializar o SDO.

⇒ **Sdo deSerialize(SerializedSdo serializedsdo),** para obter um SDO a partir de sua forma serializada.

⇒ **String Externalize(Sdo rootSdo, int nivel)** retorna uma *String* que representa o SDO com a estrutura apresentada na seção 2.1.2.1 .

⇒ **Sdo Internalize(String externalizedSdo),** para obter o SDO a partir de sua forma externalizada.

⇒ **void ExternalizeToFile(Sdo rootSdo, String filePath),** externaliza um SDO e o armazena num arquivo com o nome *filePath.*

⇒ **Sdo InternalizeFromFile(String filePath),** a partir da forma externalizada contida no arquivo indicado por *filePath* retorna um SDO

3.3.- Catálogo

Como já foi visto no capítulo 2 o catálogo apresenta suas interfaces através de um ORB e foi implementado usando a API JDBC, que por meio de um *driver* se comunica com um SGBD, o mSQL, que gerencia as tabelas que contêm os dados do catálogo.

3.3.1.- Sobre o ORB

Como já foi dito, o catálogo apresenta suas interfaces através de um ORB, no caso do protótipo foi usado a OrbixWeb 3.0[13] tanto do lado cliente, no agente, como do lado servidor, no catálogo.

A compilação da IDL do catálogo gera as classes necessárias para comunicação com o ORB e as que definem os métodos que permitem em seguida desenvolver as classes que implementam cada interface.

Usamos a abordagem *ImplBase* para implementar as interfaces do catálogo. Esta abordagem é o método de implementação definido na especificação CORBA. Para cada interface IDL, OrbixWeb gera uma classe abstrata nomeada _<type>*ImplBase*, onde <type> representa o nome da interface IDL. Para indicar que uma classe Java, no nosso caso *LoockupImplementation*, implementa uma interface IDL, *Loockup*, essa classe deve herdar da correspondente classe *ImplBase*, no nosso caso *_LoockupImplBase*. Além disso, a classe *LoockupImplementation* deve implementar efetivamente os métodos definidos na interface IDL. O explicado acima pode ser visualizado no seguinte trecho de código.

```
/*
 * Implementação da interface Loockup
 */
public class LoockupImplementation extends _LoockupImplBase
   {
```

Para um cliente poder fazer requisições numa interface, *Loockup* por exemplo, é necessária uma aplicação servidora onde um objeto *LoockupImplementation* é instanciado. O OrbixWeb deve ser informado que a interface está acessível aos clientes conectando a implementação ao *runtime.*

Como o OrbixWeb 3.0 usa o padrão OMG de mapeamento IDL-Java , todos os clientes e servidores devem chamar *org.omg.CORBA.ORB.init()* para inicializar o ORB, como pode ser visto no código da classe *LoockupServer* a seguir.

```
package catalogo;

import IE.Iona.OrbixWeb._OrbixWeb;
import IE.Iona.OrbixWeb._CORBA;
import org.omg.CORBA.ORB;

/* Servidor para a interface Loockup
 * arg[0]  nome do host que aloja o catálogo
 * arg[1]  login para conexão ao catálogo
 * estes parâmetros são passados quando o servidor é registrado
 */
public class LoockupServer
   {
    public static void main (String args[])
      {

       Loockup loockupImpl = null;
       org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init();

          try {
               loockupImpl = new LoockupImplementation
                              ("com.imaginary.sql.msql.MsqlDriver",
                                "jdbc:msql://"+args[0]+"/catalogo",
                                                      args[1], "");

               _CORBA.Orbix.impl_is_ready("LoockupSrv");

               System.out.println("Shutting down server...");
               orb.disconnect(loockupImpl);

              }
               catch(org.omg.CORBA.SystemException se) {
               System.out.println("Exception raised during creation
                 of Loockup-Implementation" + se.toString());
               return;
              }

          System.out.println ("Server exiting....");
```

```
        }
    }
```

O programa servidor cria um objeto *LoockupImplementation* e indica ao OrbixWeb que a inicialização do servidor foi completada chamando *impl_is_ready()* com um parâmetro que é o nome com o qual o servidor é registrado no Repositório de Implementação (IR).

Esta abordagem e o registro do servidor no IR permitem a ativação automática do servidor da interface, através do ativador *orbixd*, quando o cliente invoca o método *bind()* na aplicação cliente.

3.3.2.- Sobre a implementação do catálogo

3.3.2.1.- O Servidor

Antes de entrar na implementação vamos fazer algumas observações sobre o servidor. O servidor ao ser registrado requer dois parâmetros, como pode ser visto na seção anterior, que indicam o host onde está o SGBD e o *login* para se conectar a ele. Estes parâmetros são logo repassados ao objeto *LoockupImplementation,* quando este é instanciado. Esta abordagem permite mudar a localização do SGBD sem modificar nenhuma linha de código e, por sua vez, permite também que o parâmetro *login* possa ser mudado.

3.3.2.2.- A API JDBC

Faremos aqui alguns comentários sobre a utilização da API JDBC principalmente na implementação da interface *Loockup*, dado que a abordagem na interface *Admin* é muito similar.

Como já pôde ser observado no código do *LoockupServer,* a classe *LoockupImplementation* recebe, na instanciação, o nome da classe do *driver* necessário para se conectar com o SGBD, a *url* que indica a localização do mesmo, o *login* e o *password* para a conexão. Com esses dados, no método construtor, é carregada a classe que implementa o *driver* e em seguida é feita a conexão com o SGBD, como pode-se visualizar no código a seguir.

```
/* Método construtor
 * Parâmetros:
 *    driver: driver jdbc para conexão à base de dados
 *    url: url para fazer a conexão com a base de dados
 *    login: login para conexão
 *    passwd: password para conexão
 */

   public LoockupImplementation(String driver, String url,
                                String login, String passwd)
      {
       try
         {
          // carga do driver;
          Class.forName(driver);

          // conexão com a base de dados que contém o catálogo;
          con = DriverManager.getConnection(url, login, passwd);
         }
          catch( Exception e ) {
          System.out.println(e.getMessage());
             e.printStackTrace();
       }

      }
```

Ao efetuar a conexão com o SGBD é retornado um objeto da classe *Connection* que é o objeto que permite efetuar todas as operações sobre o banco de dados.

Os resultados de consultas ao Banco de Dados são retornados em objetos de tipo *ResultSet* que permitem obter cada uma das filas do resultado.

A API fornece também uma classe *Statement* através da qual podem ser criadas sentenças SQL, e a mesma classe apresenta métodos para enviar essas sentenças para que sejam executadas pelo SGBD.

Também é possível obter metadados do Banco de Dados tais como: nome das tabelas, dos índices, esquemas das tabelas, etc.

O *driver* usado é uma versão beta implementada por George Reese que pode ser obtido sem custo na Internet[29].

3.3.2.3.- A estrutura do Banco de Dados

O Banco de Dados que contém os dados fornecidos pelo catálogo é composto por dois tipos de tabelas:

tabela 1: tabela cujo nome é o mesmo do tipo de produto procurado pelo usuário (Ex. tipo de produto = carro = nome da tabela), e que contém registros com dados sobre esse tipo de produto (Ex. marca, modelo, cor, preço, etc.). Este tipo de tabela representa a lista de preço dentro do catálogo e existe uma para cada tipo produto;

tabela 2: esta tabela recebe o nome de **tableId** e contém registros com dados que identificam as listas de preços e que já foram definidos na seção 2.2.2 *(owner, product_type, catalogue_entries, negotiation_interface*). Existe um registro nesta tabela para cada lista de preço no catálogo, e a palavra chave é o tipo do produto.

3.3.2.4.- Os métodos

Neste item faremos um comentário da implementação de cada método nas interfaces.

LoockupImplementation

⇒ **String[] getTables(),** utilizamos o método *getMetaData()* sobre o objeto *Connection*, que nos retorna os metadados do Banco de Dados, e em seguida *getTables()* sobre o resultado para obter as tabelas.

⇒ **Id getId(String table),** fazemos uma consulta na tabela *tableId*, utilizando uma sentença SQL, procurando pelo tipo de produto indicado pelo usuário.

⇒ **Column[] getSchema(String table),** dado que o *driver* não tem implementado o método *getColumn()* sobre os metadados utilizamos um método auxiliar **getMetaData(),**que faz uma consulta sobre a tabela e em seguida utilizamos métodos sobre metadados no resultado desta consulta.

⇒ **SerializedSdo query(String table, String[] property, String[]orderBy)**

⇒ **SerializedSdo execQuery(String table, String sqlStatement),** nestes dois métodos o procedimento é similar, com a diferença que no primeiro a sentença SQL deve ser preparada com o conteúdo de *property* e *orderBy*. Um objeto *Statement* é criado a partir do objeto *Connection* e logo é executado nele o método *executeQuery()* com a sentença SQL como parâmetro. O resultado é um conjunto de filas que é transformado num SDO pelo método auxiliar **rs2Sdo()**. O código a seguir apresenta o descrito acima para o caso de *execQuery()*.

```
public catalogo.SerializedSdo execQuery(String table,
                                         String sqlStatement)
  {
   Convert conv = new Convert();

   try
      {
       // execução da sentença SQL;
       stmt = con.createStatement();
       rs = stmt.executeQuery(sqlStatement);
       stmt.close();
      }
      catch( Exception e ) {
      System.out.println(e.getMessage());
         e.printStackTrace();
   }
   // conversão do conjunto de resultados em Sdo;
   sdo.Sdo rSdo = this.rs2Sdo(rs);

   // serialização do Sdo;
   catalogo.SerializedSdo  serSdo = conv.Serialize(rSdo);

   return serSdo;
  }
```

Este é o mesmo procedimento usado no método getId() para consultar a tabela *tableId.*

AdminImplementation

⇒ **boolean setId(String table, Id newId),** a implementação deste método é similar ao método *getId()* da interface Loockup, somente que neste casso estamos adicionando um registro.

⇒ **Boolean add(String table, String[] newRow),**

⇒ **int delete(String table, String[] row2delete),**

⇒ **int update(String table, String[] actualRow, String[] newRow),**

⇒ **int executeUpdate(String table, String sqlStatement),** a operação destes métodos é similar a do método *execQuery()* da interface *Loockup*. Nos três primeiros a sentença SQL é preparada com base nos parâmetros. A diferença principal com os métodos de *Loockup* é que o método invocado no objeto *Statement,* ao qual se repassa a sentença SQL, é o método *executeUpdate()*. Este método é usado quando a sentença SQL produz modificações na tabela, como se pode visualizar no trecho de código a seguir.

```
public int execUpdate(String table, String sqlStatement)
        {
         int i,rowCount = 0;
         boolean uId=false;

         try
           {
            // execução da sentença SQL;
            stmt = con.createStatement();
            rowCount = stmt.executeUpdate(sqlStatement);
            stmt.close();
           }
         catch( Exception e ) {
         System.out.println(e.getMessage());
         e.printStackTrace();
         }

         if (rowCount>0)
            {
             if(sqlStatement.startsWith("INSERT"))
               {
                uId = this.updateId(table,rowCount);
               }
             else
               {
                if(sqlStatement.startsWith("DELETE"))
                      uId = this.updateId(table,(-1*rowCount));
               }
            }

         return rowCount;
```

```
}
```

Como pode-se observar no código estes métodos atualizam na tabela *tableId* a quantidade de entradas da tabela modificada.

⇒ **boolean index(String table, String indexName, String[] newKey),** o procedimento é o mesmo que nos métodos anteriores, somente mudando a sentença SQL que neste caso é construída da seguinte forma:

```
String ndxSql ="CREATE INDEX "+indexName+" ON "+table+" (";

for(i=1;i<newKey.length;i++)
           {
            if(i!=1)
                ndxSql = ndxSql +",";
            ndxSql = ndxSql +newKey[i];
           }
         ndxSql = ndxSql +")";
```

Foram implementados também alguns métodos auxiliares como o método **String constroiSql(String table, String separator, String[] row)** que constrói a parte de condições sobre as propriedades numa sentença SQL levando em conta o tipo de cada uma delas.

3.3.3.- O SGBD

O SGBD utilizado foi o Mini SQL, ou mSQL, da Hughes Technologies[12], em sua versão 2.01, que é um SGBD de pequeno porte desenhado para prover acesso rápido a um conjunto de dados com o menor trabalho possível para o sistema. Não é um software de domínio público mas pode ser requisitada uma licença livre para uso em entidades de ensino. É fornecido o código fonte em C com os *makefiles* para compilação e instalação automática.

O sistema compreende um servidor de banco de dados e várias ferramentas que permitem ao usuário ou a uma aplicação cliente se comunicar com o servidor. O mSQL usa SQL, mas não fornece uma implementação completa do padrão ANSI SQL. Nós somente utilizamos aquelas

funcionalidades que pertencem ao padrão ANSI SQL, de modo que a implementação possa ser independente do SGBD.

A filosofia do mSQL é fornecer um SGBD que possa gerenciar tarefas simples com rapidez. Ainda que ele não seja apropriado para ambientes de finanças, nesta versão ele fornece acesso rápido e consistente a conjuntos de dados com até 1 milhão de registros.

Para seu funcionamento é necessário iniciar um *daemon*, **msql2d**, que gerencia as chamadas ao SGBD. O *msql2d* ao ser iniciado carrega um arquivo de configuração que permite, entre outros, estabelecer a localização do banco de dados, o usuário e o administrador do banco, e o nome da porta TCP onde o mSQL aceitará conexões cliente/servidor.

O sistema apresenta também ferramentas interativas para manipulação do banco de dados, dentre elas as mais importante são: **msqladmin** para administração do banco de dados (criação do banco de dados, *shutdown* do servidor, etc), e o monitor **msql** que é uma interface interativa com o servidor mSQL que permite enviar comandos SQL.

A simplicidade do sistema e sua facilidade de manipulação o fazem muito adequado para a utilização em nosso protótipo.

3.4.- Agente

3.4.1.- Sobre o Sistema de Agentes

Advertidos sobre alguns problemas do sistema de agentes da IBM, o Aglets, a respeito de sua interação com objetos CORBA, não consideramos este sistema e iniciamos o estudo do sistema de agentes Odyssey, da General Magic[7], cujo código Java se acha disponível para ser utilizado.

O estudo das classes do sistema e os testes efetuados permitiram constatar a simplicidade do sistema e sua capacidade para fornecer as funcionalidades requeridas no seção 2.1.1 para a implementação do nosso protótipo.

Um outro sistema analisado foi Voyager[15], da ObjectSpace. Este sistema implementa muitas funcionalidades além das necessárias para nosso sistema, de fato ele apresenta algumas características de um ORB. Para que um cliente Voyager possa interagir com objetos CORBA[16] é preciso o seguinte: 1) a interface IDL do objeto CORBA deve ser processada previamente com um utilitário nomeado **vcc** que gera uma interface Java e uma referência virtual funcional; 2) deve-se obter por algum método a IOR do objeto CORBA, por exemplo lendo-a de um arquivo, e em seguida convertê-la em uma referência virtual Voyager utilizando um método próprio do Voyager. Obtida esta referência pode-se invocar nela os métodos do objeto CORBA. Se o resultado de uma invocação é um objeto CORBA o Voyager converte automaticamente a referência numa referência virtual Voyager. Como se pode ver esta metodologia de interação depende do sistema Voyager e não pode ser usada com outros sistemas de agentes, por isso achamos não ser adequada devido à generalidade que queremos obter.

Analisando as descrições de outros sistemas de agentes [5, 3] fizemos a escolha de utilizar o Odyssey da General Magic pelas seguintes razões: a) ele se acha dentro do especificado pelo OMG na "Mobile Agent System Interoperbility Facilities Specification"[18], além de ser a General Magic uma das empresas que apresentaram a proposta que foi aprovada; b) as funcionalidades que ele apresenta são básicas e podem ser achadas em qualquer outro sistema de agentes, o que não restringe a generalidade da abordagem que tentamos apresentar.

No apêndice B.1 apresentamos uma descrição do Sistema Odyssey.

Do ponto de vista da iniciação do sistema, esta se realiza rodando a *Engine* através de uma máquina Java e utiliza nessa iniciação um arquivo *xxx.boot* que permite escolher valores para variáveis do sistema que definem algumas funcionalidades, entre elas, nome da classe que implementa o lugar inicial para um agente, host do *Finder* e classe que o implementa, níveis de mensagens de controle e outros que podem ser vistos no apêndice B.2. Esta *Engine*, em cada host, instancia um lugar inicial, *QPlace*, que é o encarregado de hospedar o agente.

3.4.2.- Sobre o Agente

Como já foi comentado no capítulo 2, nosso agente é implementado através da classe **QWorker**, que é uma especialização da classe *genmagic.odyssey.Worker*. Esta classe *Worker* é por sua vez uma especialização da classe *genmagic.odyssey.Agent* e tem a particularidade de carregar uma lista de tarefas, cada uma delas com um destino associado onde essa tarefa deve ser executada. Esta facilidade do *Worker* o faz muito apropriado para a tarefa que tem que realizar nosso protótipo.

Também formam parte do "módulo" agente as classes **DecisionModule** e **Query**. A seguir fazemos uma descrição delas e da classe do agente.

3.4.2.1.- Classe QWorker

Esta classe implementa as funcionalidades próprias de um agente com as características do *Worker,* ou seja, ele carrega uma lista de tarefas (*tasks*) e para cada uma delas tem especificado: a) um método que pertence ao agente, e define a tarefa, e b) um *ticket* que indica em qual host efetuar essa tarefa e o modo de transporte para chegar a esse host. Em cada host o agente realiza até o fim a tarefa especificada para ele.

Nosso agente apresenta um método **void doCommand()** que é a tarefa a efetuar em cada host. Este método somente invoca a classe que implementa o módulo de decisão e armazena num *array* de **SerializedSdo** os resultados das tarefas.

Os dados fornecidos pelo usuário, que foram armazenados na estrutura *UserData,* os resultados e as tarefas são carregados durante o percurso do agente.

Existe também um método **void reportResults()** que é a tarefa que o agente realiza quando retorna a sua origem e tem por objetivo invocar o mesmo método no lugar inicial para entregar os resultados do trabalho do agente para serem apresentados ao usuário.

O método **queryTrader()** que permite fazer consultas a um *Trader* para obter os lugares onde procurar catálogos, não foi implementado neste protótipo.

Também não foi implementado o método **negotiate()** para que o Agente possa negociar os produtos.

3.4.2.2.- Classe DecisionModule

Esta classe implementa o módulo de decisão do agente, ela apresenta um único método público **SerializedSdo operate(UserData data, String host)**. Este método tem três funções: a) preparar uma sentença SQL para ser enviada ao catálogo baseado nos dados contidos na estrutura *UserData*, b) instanciar a classe *Query* e invocar o método *execQuery* com a sentença SQL preparada e c) analisar os resultados e decidir novas consultas.

A "capacidade" de decisão deste módulo está baseada num campo *priority* que contém as propriedades fixas da estrutura *UserData*, este campo pode ter valores de 0 a 9. No caso do resultado da consulta não conter nenhum item de resposta, o módulo analisa as propriedades fixas com prioridade distinta de zero e escolhe a de maior valor, constrói uma nova sentença SQL excluindo essa propriedade e faz uma nova consulta, este procedimento se repete até obter uma resposta ou até não existirem mais propriedades com *priority* distinto de zero. Denominamos este processo de flexibilização de propriedades fixas.

3.4.2.3.- Classe Query

Esta classe implementa uma ponte entre o catálogo, cuja interface está disponível através de um ORB, e o agente.

A classe *Query* é instanciada pela classe *DecisionModule* a qual repassa para ela o parâmetro host que indica o host do catálogo. Este parâmetro é usado no método construtor da classe para fazer um *bind* com a interface escolhida do catálogo, neste caso a interface *Loockup*. Para fazer isto o método construtor inicia o ORB e em seguida invoca o método *bind()*, próprio da OrbixWeb, na interface *Loockup* usando a classe *LoockupHelper*, repassando o nome do servidor e o host onde ele se acha. Este método *bind()* aciona, através do ORB, o *daemon* encarregado de lançar o servidor da interface no lado servidor e cria, no lado cliente, um objeto *proxy* para a interface, que recebe e

repassa todas as invocações para essa interface no lado servidor[13]. O trecho de código correspondente é o seguinte:

```
public class Query implements java.io.Serializable
 {
  Loockup lRef = null;

  /**
   * Initialize
   */
  public Query(String srvHost) throws Exception
         {
          ORB.init();

          try{
              lRef = LoockupHelper.bind (":LoockupSrv",srvHost);
             }
              catch (org.omg.CORBA.SystemException ex) {
              System.out.println("Exception during bind");
              System.out.println(ex.toString());
              ex.printStackTrace();
              throw new Exception("Bind to object failed");
             }

         }
```

Os métodos da classe *Query* são os mesmos da interface *Loockup* do catálogo, eles somente repassam os parâmetros e invocam o método no catálogo dentro de um bloco *try* que faz o controle de exceções exigido pelo ORB, como se pode visualizar no seguinte trecho de código.

```
public String[] getTables()
   {
    try {
         return lRef.getTables();
        }
         catch (org.omg.CORBA.SystemException ex) {
         System.out.println("FAIL\tException during getTables");
         System.out.println(ex.toString());
         return null;
        }
```

Esta abordagem faz com que a classe *Query* seja a única classe dentro do "módulo" agente que depende do ORB, isto diz muito a respeito da flexibilidade do sistema.

3.5.- Interface do usuário

A interface do usuário é composta por duas classes, **Launcher** e **Receiver** que são interfaces gráficas entre o usuário e o sistema e implementam as funcionalidades definidas para os objetos Lançador e Receptor, respetivamente, da seção 2.2.1.1. A seguir descrevemos cada uma delas.

3.5.1.- Classe Launcher

Esta classe como já foi colocado acima é uma interface gráfica que permite ao usuário especificar o produto a procurar, suas propriedades e o itinerário a seguir pelo agente.

O **Launcher** é instanciado pelo lugar inicial, *QPlace*, e possui um método **sendAgent()** que realiza as seguintes tarefas:

a) colocar os dados coletados na estrutura *UserData*
b) obter os números IP dos nomes de hosts do itinerário.
c) construir as tarefas e coloca-las na estrutura *tasks*
d) lançar o *QWorker* repassando para ele as tarefas (*tasks*) e os dados (*UserData*).

Na figura 19 apresentamos a interface gráfica que o *Launcher* apresenta para o usuário.

Uma vez enviado um agente, o mesmo *Launcher* pode ser usado para enviar outro ou outros agentes para procurar produtos distintos com itinerários distintos.

Enviar Agentes

======================= D A D O S =======================

Produto : carro

------------ PROPRIEDADES ------------

Fixas ----------------

Nome : modelo Tipo Prioridade: 1

Valor : Uno string

marca, String, Fiat, 0
modelo, String, Uno,

Adicionar Limpar

Discreta ------------

Nome : cor Tipo : string

Valor : branco preto verde

Continua ------------

Nome : ano Tipo : integer

Valor Inf.: 1996 Valor Sup.: 1998

------------ PRECO ------------

Valor Inf.: 9000 Valor Sup.: 13000

======================= D E S T I N O =======================

TRADER : nao consultar Trader

Destino : solimoes.dcc.unicamp.br

Adicionar Destino

gaivota.dcc.unicamp.b
solimoes.dcc.unicamp

Limpar Destinos

Enviar

Figura 19 - Interface gráfica do Launcher

3.5.2.- Classe Receiver

Esta classe é uma interface gráfica para apresentação ao usuário dos resultados obtidos pelo agente em seu percurso.

O **Receiver** é instanciado quando o agente, *QWorker*, retorna e invoca o método *reportResults()* em *QPlace*. O método *reportResults()* em *Qplace* somente instancia o *Receiver* e invoca o método **reportResults()** do objeto *Receiver*, repassando um *array* de SDOs serializados que foram entregues ao *QPlace* pelo agente.

Esta classe além do método *reportResults()* possui um método **procResults(SerializedSdo res)** para processar cada SDO serializado, resultado da busca num host determinado, e transformá-lo num estrutura de tabela, de fácil compreensão pelo usuário.

Na Figura 20 podemos visualizar a interface gráfica do *Receiver*, com a resposta da busca efetuada pelo agente em dois catálogos diferentes.

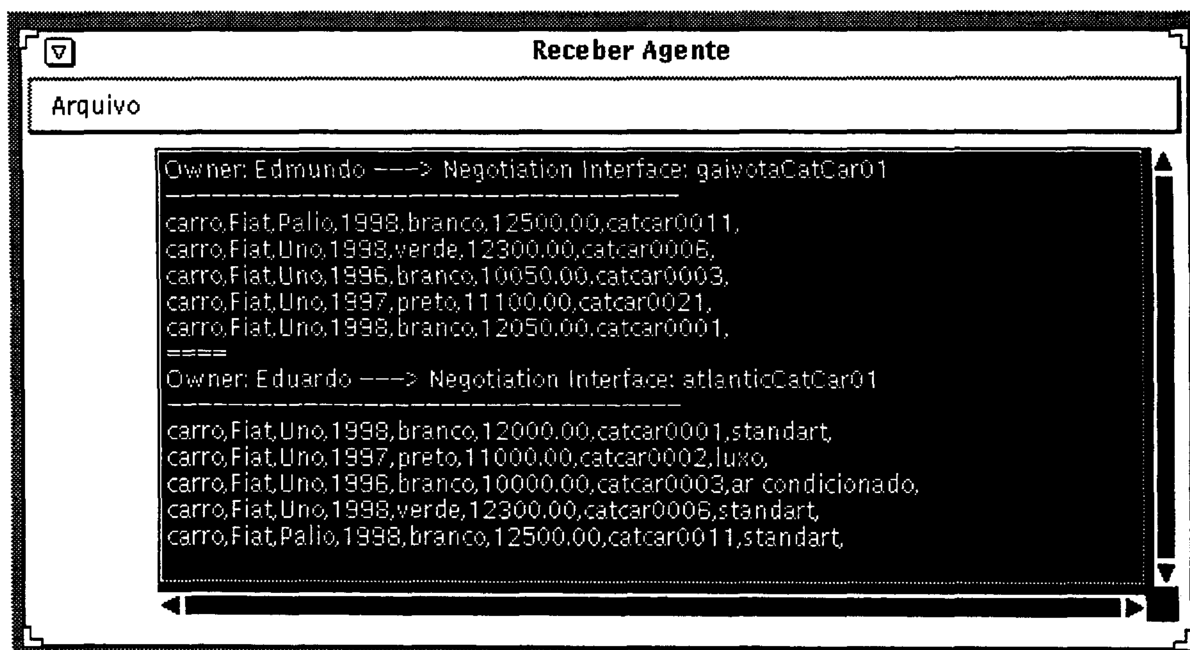


Figura 20 - Interface gráfica do Receiver

3.6.- Sistema geral

O sistema foi testado usando três *hosts*, um para lançar o agente, Host1, e os outros dois, Host2 e Host3, como lugares que o agente percorre para efetuar a busca do produto desejado pelo usuário.

Vejamos os **requerimentos** para cada um deles:

Host1 deve conter: uma máquina Java; as classes do sistema de agentes Odyssey; as classes que implementam o agente, incluindo a parte cliente do catálogo, e a interface de usuário.

Host2 e Host3 devem conter: uma máquina Java; as classes do sistema de agentes Odyssey; uma ORB; as classes que implementam o catálogo, incluindo o *driver* JDBC e o SGBD com o Banco de Dados que contém os dados do catálogo.

As classes que implementam o agente não são necessárias nos *hosts* 2 e 3 porque elas podem ser carregadas via rede do *site* de origem do Agente. No caso de nosso protótipo elas se acham presentes nos *hosts* 2 e 3 pelo fato de que para elas poderem ser buscadas via rede devem ser implementados mecanismos de segurança através do *QPlace* para que não ocorram exceções ao agente invocar o catálogo, voltaremos sobre este assunto no ponto sobre segurança.

Agora vejamos o **funcionamento** do sistema:

Host1: a *Engine* do Sistema de Agentes é inicializada usando um arquivo de *boot* nomeado *Master.boot* que estabelece, entre outras coisas, o nome da região e indica que este host é o host do *Finder*. Esta *Engine* instancia a classe *QPlace* como lugar inicial do Agente e por sua vez o *QPlace* instancia o *Launcher* para que o usuário preencha o dados e envie o agente. Quando o usuário o ordena, o *Launcher* instancia o *QWorker* e invoca nele seu método *start()*, indicando que pode começar suas tarefas. O *QWorker* verifica a primeira tarefa e pede para o sistema de agentes se movimentar até o host onde deve efetuá-la. O sistema de agentes transfere o agente e o seu conteúdo para o host de destino. No retorno, como já foi visto, o agente entrega os resultados ao *Receiver*, via *QPlace*, que os apresenta ao usuário.

Host2 e Host3: a *Engine* do Sistema de Agentes é inicializada usando um arquivo de arranque nomeado *Slave.boot* que indica a que região pertence este host e qual é o host que

hospeda o *Finder*, neste caso Host1. Esta Engine instancia o QPlace para receber o agente e o sistema de agentes avisa ao *Finder* do Host1 que tem um lugar pronto no Host2 repassando o número IP do *host*. Em cada um destes *hosts* deve ser inicializado o *daemon* que atende as chamadas ao SGBD e o ativador de servidores do ORB, e o servidor do catálogo deve ser registrado com o ativador dado que usamos a abordagem de ativação automática do catálogo quando este é invocado. Em cada um destes *hosts* o agente é recebido, efetua a sua tarefa, e em seguida pede ao Sistema de Agentes local para se movimentar até o *host* onde deve efetuar a próxima tarefa.

Fazendo uma análise do ponto de vista do grau de acordo prévio entre as partes do sistema, tanto o "dono" do agente como os "donos" do catálogo devem conhecer a interface IDL do catálogo e os host que hospedam os catálogos devem ter um sistema de agentes concordante com o sistema do agente. Se o sistema fosse Odyssey, a única classe necessária nos *hosts* 2 e 3 é a classe *QPlace*, além do arquivo *Slave.boot*.

A seguir descrevemos o cenário da interação do agente com o catálogo.

Quando o agente chega num site ele traz com ele: a) a lista de tarefas; b) a estrutura *UserData* com os dados fornecidos pelo usuário para efetuar a busca e c) os resultados coletados em outros sites. Ao chegar o agente é instanciado pelo sistema de agentes local e verifica na sua lista de tarefas o que ele tem a fazer. No caso de nosso protótipo em todos os lugares o agente invoca seu proprio método *doCommand()*. Este método instancia a classe *DecisionModule*, invocando nela seu método *operate()* com *UserData* e o nome do *host* como parâmetros. Este método prepara, com base nos dados contidos em *UserData*, uma sentença sql de consulta para o catálogo. A seguir é intanciada a classe *Query* com o nome do *host* como parâmetro, ao ser instanciada, a classe *Query* inicia o ORB e faz um *bind* com a interface *Loockup* do catálogo. No lado servidor a classe *LoockupServer* é instanciada pelo *orbixd*, se ela não estiver instanciada, a qual por sua vez instancia a classe *LoockupImplementation* repassando para ela o *driver* Java para conexão com o SGBD, a url do mesmo e os parâmetros de acesso *login* e *password*. Com esses parâmetros a classe *LoockupImplementation* carrega o driver e faz a conexão com o SGBD. Voltando à classe *DecisionModule*, no método *operate* é logo invocado o método *execQuery* na classe *Query* com o nome do produto e a sentença sql como parâmetros. Este método *execQuery* repassa esses parâmetros ao

mesmo método da interface *Loockup* do catálogo usando a referência de interface obtida ao fazer o *bind*. Na implementação da interface *Loockup*, *LoockupImplementation*, a invocação do método *execQuery* envia a senteça sql ao SGBD o qual a executa sobre a tabela indicada pelo tipo de produto. O resultado é convertido num SDO e logo serializado para ser entregue ao agente. Quando a classe *DecisionModule* recebe um resultado através da classe *Query* ela examina esse resultado e se contém algum item, retorna o controle à classe do agente a qual armazena o resultado. Se a resposta não contém itens, a classe *DecisionModule* realiza o processo de flexibilização das propriedades fixas. Este processo se repete até obter um resultado com itens ou até não existirem mais propriedades fixas para flexibilizar. Obtida uma resposta, como já foi dito, o controle retorna para o agente que armazena essa resposta, verifica na sua lista de tarefas e pede, ao sistema de agentes local, para se movimentar ate o seu próximo destino.

Na interação do agente com o ORB podemos levantar duas questões:

1) A classe *Query* necessita das classes do lado cliente (*stub*) geradas ao compilar a IDL do catálogo e ela mesma foi implementada levando em conta um ORB específico que pode não ser o mesmo do catálogo, em cujo caso a interação pode não ser possível. Este problema pode ser superado se o ORB do lado do agente fosse um ORB Java dado que a mesma pode ser requisitada via rede, somente necessitamos a parte cliente, instalada no host do catálogo e fazer comunicação via IIOP, local, com o ORB que contém a interface do catálogo.

2) Se usarmos, como no casso do protótipo, algum método proprietário para ativação automática do servidor do catálogo, *bind()*, a conexão não seria possível se o ORB do catálogo não fosse o mesmo. Isto pode ser solucionado se usarmos uma opção de compilação da IDL do catálogo, presente ao menos na OrbixWeb 3.0, que gera classes somente do padrão CORBA e usando uma abordagem onde o servidor seja registrado num Serviço de Nomes e a classe Query consulte esse Serviço para obter a referência do objeto catálogo. O ORB do catálogo deverá conter as classes padrões CORBA. Esta abordagem pode ser usada também para evitar o problema da questão 1.

3.7.- Algumas considerações sobre segurança

Embora não consideremos segurança na nossa abordagem o modelo apresentado pode ser usado com os mecanismos de segurança existentes. Porém queremos fazer os seguintes comentários:

a) A linguagem Java apresenta hoje classes e métodos para encriptação e assinatura do código que podem ser usados com o agente, implementando alguns controles de acesso através do *QPlace*.

b) Um outro ponto de controle de acesso pode ser a invocação do catálogo, e este controle pode ser implementado usando: ou os recursos da ORB, ou o Serviço de Segurança se estiver implementado, ou recursos do SGBD, como *login* e *password* de acesso.

3.8.- Trabalhos relacionados

Apresentaremos a seguir três trabalhos relacionados com a nossa pesquisa.

Existe uma categoria de agentes ou sistemas baseados em agentes para busca de produtos na Internet que interagem com catálogos embutidos em páginas Web. O primeiro trabalho a ser comentado é o *BargainBot*[1] que pertence a esta categoria. O *BargainBot* é catalogado como um agente de busca para Comércio Eletrônico na Internet que permite a um usuário a busca simultânea em várias livrarias virtuais. O *BargainBot* foi feito usando PERL e a partir dos dados fornecidos pelo usuário o sistema envia vários "sub-agentes", cada um dos quais busca numa determinada livraria. O resultado da busca é integrado numa única pagina em HTML para que o usuário possa comparar os resultados.

Estes sistemas são substancialmente diferentes do nosso pelo fato que nós propomos principalmente trabalhar na interação do agente com catálogos que apresentam interfaces padronizadas num ORB. Além disso a nossa pesquisa também apresenta o modelo e define as interfaces para o catálogo. Nossa abordagem não necessita informação a respeito da implementação do catálogo como no caso do *BargainBot*.

Um segundo trabalho relacionado é o efetuado por dois pesquisadores da Universidade de Stuttgart, na Alemanha[32]. Eles apresentam uma modelagem para agentes móveis genéricos para mercados eletrônicos. O modelo propõe agentes de compra móveis e agentes de vendas fixos estabelecendo como obrigatórias as funcionalidades da fase de informação, ou seja busca de produtos ou fornecimento de informação sobre os mesmos, e compra ou venda desses produtos. É proposta a interação com um *Trader* ou mecanismo similar para achar os agentes de vendas cujas interfaces são apresentadas em IDL e fornecem dois métodos: *inquiry*, para requisitar dados sobre um produto e *order*, para comprar o produto.

A nossa pesquisa somente modela a fase de informação ou procura de produto e deixa em aberto a parte de negociação. A principal diferença está na interface do catálogo, que poderia comparar-se com a do agente de venda, pelo fato que não é apresentada como um agente, e além disso as funcionalidades que definimos dão ao usuário a possibilidade de fazer uma busca mais seletiva, enquanto que a interface do agente de venda só permite fazer requisições especificando o nome do produto.

Finalmente um terceiro trabalho que achamos importante é o projeto OFFER (Object Framework for Electronic Requisitioning)[2], o qual apresenta um modelo avançado para corretagem (brokerage) na Internet. O principal objetivo deste projeto é especificar e desenhar componentes de um sistema de corretagem aberto com capacidade para suportar várias áreas de negócio. A abordagem adotada na definição do sistema OFFER é similar aos pontos de vista do Modelo de Referência para Processamento Distribuído Aberto (RM-ODP) da ISO. O serviço de corretagem é descrito do ponto de vista da empresa, de informação, computacional e de tecnologia e são fornecidas especificações de alto-nível de seus componentes fundamentais.

Em nosso trabalho o agente realiza uma das funcionalidades definidas para o serviço de corretagem, a busca de produtos. Uma outra relação com nossa pesquisa é que utiliza como ponto de partida o Modelo de Referência para Comércio Eletrônico apresentado conjuntamente pelo OMG e CommerceNet[25] que já foi introduzido na seção 1.3.2. Outro ponto em comum é que o serviço OFFER deverá interagir com catálogos cujas interfaces podem ser como as definidas na nossa pesquisa.

Conclusão

Comércio Eletrônico depende cada vez mais do surgimento de capacidades ou mecanismos adequados para que um usuário possa obter os dados necessários sobre produtos para poder fazer a decisão certa de negócio.

Dentre estas capacidades consideramos mais importantes: 1) mecanismos universais de busca; e 2) padrões para apresentação da informação.

O principal obstáculo para a difusão mais ampla de catálogos interoperáveis é a falta de padronização na área de descrição e classificação de itens e a não adoção de um padrão para protocolos de comunicação entre eles.

As principais contribuições desta pesquisa são:

a) O modelo geral para a tarefa de busca de produtos numa rede utilizando agentes móveis interagindo com catálogos que apresentam suas interfaces através de um ORB, estabelecendo os componentes do sistema e as interações entre eles. Esta interação entre objetos CORBA e não CORBA incrementa a interoperabilidade do sistema.

b) Modelagem do catálogo e definição de suas interfaces em IDL descrevendo os métodos de cada uma delas. Achamos isto muito importante dado que em [2] é comentado que

"quase todos os projetos da áreas de corretagem (*brokerage*) que seguem o padrão OMG CORBA estão em seu estado inicial" e "não existem hoje catálogos na Internet acessíveis via interfaces OMG IDL".

É importante destacar que a **escalabilidade** do sistema é considerada na modelagem do catálogo em dois aspectos:

1) o catálogo pode possuir qualquer número de listas de preços e pode-se obter as caraterísticas delas com o método *getTables()* e elas podem ser manipuladas individualmente utilizando o parâmetro *table* dos métodos das interfaces.

2) a utilização de um modelo de componentes com o JDBC como ponte entre as interfaces do catálogo e o SGBD que armazena os dados permite um fácil *upgrade* do sistema de armazenamento dos dados se fosse necessário.

A **interoperabilidade** do sistema está considerada no seguinte:

1) especificação das interfaces do catálogo em IDL e sua implementação através de um ORB o qual faz as invocações independentes da implementação. De fato as aplicações que invocam o catálogo devem estar desenvolvidas em C, C++, SmallTalk ou Java que são linguagens para as quais existe mapeamento IDL.

2) modularização do agente, o qual permite incorporar nele módulos para iteração com outros sistemas.

3) utilização dos SDOs para transporte de dados, o qual elimina a necessidade de um acordo a respeito do formato e identificação dos dados. É importante repetir aqui que esta abordagem não é muito útil se não existir uma padronização das etiquetas para os SDOs com a correspondente implementação de um sistema de repositório e manipulação dos mesmos.

4) a implementação do método *getSchema()* na interface *Loockup* que permite descobrir a estrutura de armazenamento dos dados e o nome e tipo das propriedades antes de fazer as consultas, evitando um acordo prévio sobre este assunto.

A respeito da **implementação** queremos destacar alguns pontos:

⇒ não existem restrições quanto ao sistema operacional dado que a implementação necessita um ORB e uma máquina Java os quais são independentes de plataforma.

⇒ o sistema de agentes escolhido é o suficientemente geral para poder ser substituído por qualquer outro sem agregar restrições ao sistema.

⇒ a utilização do JDBC e a implementação do servidor do catálogo com parâmetros que são passados no momento do lançamento, e que definem a localização da fonte dos dados, e com os parâmetros de acesso, incrementam a escalabilidade do sistema. Não é o caso da nossa implementação, mas também o *driver* para conexão com o SGBD poderia ser passado como parâmetro, o qual facilitaria a mudança do SGBD e a fonte de armazenamento dos dados..

⇒ a implementação do sistema usando a linguagem Java permitiu a interação entre agentes, objetos não CORBA, e objetos CORBA. O fato das funcionalidades de rede serem "nativas" nesta linguagem, e o fato de que Java foi desenvolvido para aplicações nestes ambientes diz muito respeito à interoperabilidade e flexibilidade do sistema. Somente como exemplo o sistema poderia ser mudado facilmente de modo que o agente possa ser lançado utilizando uma página Web, que é carregada através de um *browser* qualquer, com um *Applet* que colhe os dados e em seguida os envia a seu host de origem para que com eles seja lançado um agente.

⇒ o sistema de consulta do agente implementado com duas classes, *DecisionModule* e *Query*, com distintas funcionalidades separa a parte dependente do ORB, *Query*, da parte que implementa a lógica de consulta, *DecisionModule*, dando maior flexibilidade ao permitir mudar a inteligência das consultas ou o ORB com a qual se tem que interagir trocando somente uma classe.

Trabalhos futuros

Como trabalhos futuros baseados nesta pesquisa podemos propor:

- Incorporar nesta abordagem os futuros padrões da OMG sobre *Tagged Data* e *Object by Value*.
- Completar o catálogo com as funcionalidades definidas no Modelo de Referência para Comércio Eletrônico[25] incorporando funcionalidades de interoperabilidade com outros catálogos e serviço de assinatura de modo a oferecer um Serviço de Catálogo.
- Considerar a interação do agente com catálogos embutidos em páginas Web ou catálogos implementados com aplicações baseadas em documentos considerando o futuro padrão XML(*eXtensible Markup Language*) que sugere uma forma simples de definir transações e fornece mensagens auto-descritivas[2].

Referências Bibliográficas

[1] **Bassam Aoun, IMAGE Technology Research Group** - *Agent Technology in Electronic Commerce and Information Retrieval on the Internet* - 1996 - http://www.scu.edu.au/asuweb96/auon/paper.html

[2] **Bichler Martin and Segev Arie** - *A Brokerage Framework for Internet Commerce* - Fisher Center for Management & Information Technology, Walter A. Haas School of Business, University of California, Berkeley, USA - 1998 - CMIT Working Paper 98-WP-1031 - http://hass.berkeley.edu/~citm/OFFER

[3] **Bigus Joseph P. and Jennifer** - *Constructing intelligent Agents with Java* - Wiley - 1998

[4] **CommerceNet** - *Catalogs for the Digital Marketplace* - CommerceNet Research Report - 1997 - http://www.commerce.net/about/membership/rrsample.html

[5] **Endler Markus** - *Agentes Móveis : Um novo paradigma para a Programação Distribuída* - Departamento de Ciência da Computação. IME. USP - SBRC98 - http://www.ime.usp.br/~endler/paperlinks/sbrcslides.ps

[6] **Esprit's Electronic Commerce Team** - *Electronic Commerce - An Introduction* - Maio 1996 - http://www.cordis.lu/esprit/

[7] **General Magic** - *Introduction to the Odyssey API* - 1997 - http://www.genmagic.com/agents/odysseyIntro.ps

[8] **Georgiou Christos, Stafaneas Petros** - *Strategies For Accelerating the Adoption of E-Commerce By Consumers WorldWide* - IBM Research Report RC 21277(94934) - Setembro 1998

[9] **Guttman Robert, Moukas Alexandros and Maes Pattie** - *Agent-mediated Electronic Commerce: A Survey* - Software Agents Group - MIT Media Laboratory - 1998 - http://ecommerce.media.mit.edu/papers/ker98.pdf

[10] **Harrison C.G; Chess D.A; Kershenbaum A.** - Mobile Agents: Are they a good idea? - Research Report - IBM T.J. Watson Research Center - Março 1995 - http://www.research..ibm.com/massive/mobag.ps

[11] **Hortsmann Cay S., Cornell Gary** - *Core Java Volume II - Advanced Features* - Sun Microsystems Press & Prentice Hall - 1998.

[12] **Hughes Technologies** - *Mini SQL 2.0 User Guide* - 1997 - http://Hughes.com.au/software/msql2/current.html

[13] **IONA Technologies** - *OrbixWeb Programming Guide* - http://www.iona.com.

[14] **Jon Siegel - OMG** - *CORBA Fundamentals and Programming* - John Wiley & Son - 1996

[15] **ObjectSpace** - *ObjectSpace Voyager Core Package. Technical Overview* - Dezembro 1997 - http://www.objectspace.com/product/voyager/white/VoyagerTechOview.pdf

[16] **ObjectSpace** - *Voyager CORBA Integration. Technical Overview* - Dezembro 1997 - http://www.objectspace.com/developers/voyage/white/VoyagerCORBAIntegrationW97.pdf

[17] **OMG** - *Common Facilities RFP-4 - Common Business Objects and Business Object Facilities* - Janeiro 1996 - http://www.omg.org/docs/cf/96-01-04.pdf

[18] **OMG** - *Mobile Agent System Interoperability Facilities Specification* - 1998 - OMG TC Document orbos/98-03-09

[19] **OMG** - *The Common Object Request Broker: Architecture and Specification* (CORBA) Revision 2.2 Fevereiro 1998 - ftp://ftp.omg.org/pub/docs/formal/98-07-01.ps

[20] **OMG** - *Trading Object Service Specification* - 1997 - ftp://www.omg.org/pub/docs/formal/97-12-23.ps

[21] **OMG** *Document 95-01-02 - Common Facilities Architecture Revision 4.0* - Novembro 1995 - ftp://www.omg.org/pub/docs/formal/98-07-11.pdf

[22] **OMG EC DTF** - *Asset and Content Management RFI* - Agosto 1996 - http://www.omg.org/arch2/ec/96-08-03.ps

[23] **OMG EC DTF** - *Enabling Technologies and Services for Electronic Commerce RFI* - Novembro 1996 - http://www.omg.org/arch2/ec/96-11-03.ps

[24] **OMG EC DTF**- *Electronic Commerce Domain Task Force* - http://www.osm.net/ecdtf.html

[25] **OMG/CommerceNet** - *The OMG/CommerceNet Join Electronic Commerce Whitepaper* - Julho 1997 - http://www.osm.net/upload/97-06-09.pdf

[26] **Orfali R. and Harkey D.**- *Client/Server Programming with Java and Corba, 2nd Edition* - John Wiley, 1998.

[27] **Orfali R., Harkey D., Edwards J.** - *Instant CORBA* - John Wiley & Son - 1997

[28] **Papaioannou Todd and Edwards John** - *Mobile Agent Technology Enabling The Virtual Enterprise: A Pattern for Database Query* - MSI Research Institute, Department of Manufacturing Engineering, Loughborough University, UK - 1998 - http://luckyspc.lboro.ac.uk/Docs/Papers/Agents98.html

[29] **Reese George** - *Imaginary Java Class Library JDBC Driver for mSQL Version 1.0 beta 2* - 1997-1998 - ftp://ftp.imaginary.com/pub/

[30] **Rumbaugh J.** e Outros - Object Modeling and Design - Prentice Hall - 1991

[31] **System Software Associates Inc.** - *OMG BODTF RFP-1 Submission Bussines Object Facility* - Unisys Corporation - Janeiro 1997 - ftp://ftp.omg.org/pub/docs/bom/97-01-05.ps

[32] **Villinger K., Burger C.** - *Generic mobile agents for electronic markets* - Institute of Parallel and Distributed High-Performance Systems, University of Stuttgart, Germany - 1997 - http://inforge.unil.ch/isdss97/papers/73.htm

[33] **Vogel Andreas, Duddy Keith** - *Java Programming with Corba, Second Edition* - John Wiley, 1998

Apêndice A: IDL do Catálogo

```
module catalogo
     {
      interface Loockup;
      interface Admin;

      struct Id
           {
            string owner;
            string productType;
            unsigned long entries;
            string negotiationInterface;
           };

      struct Column
           {
            string name;
            string type;
           };

      typedef sequence<string> TableSeq;

      typedef sequence<Column> SchemaSeq;

      typedef sequence<string> PropertySeq;

      typedef sequence<string> IndexSeq;

      struct SerializedSdoElement
           {
            string label;
            string value;
            string type;
            long parentIndex;
            octet flags;
           };

      struct SerializedSdo
           {
            short version;
            short type;
            sequence<SerializedSdoElement> sdoList;
           };

      interface Loockup
           {
```

```
      TableSeq getTables();

      Id getId(in string table);

      SchemaSeq getSchema(in string table);

      SerializedSdo query(in string table, in PropertySeq
                      property,  in IndexSeq orderBy);

      SerializedSdo execQuery(in string table,
                              in string sqlStatement);

     };

 interface Admin
     {
      boolean setId(in string table, in Id newId);

      boolean add(in string table, in PropertySeq newRow);

      long delete(in string table,
                       in PropertySeq row2Delete);

      long update(in string table,
          in PropertySeq actualRow,in PropertySeq newRow );
      long execUpdate(in string table,
                                in string sqlStatement);

      boolean index(in string table,in string indexName,
                                in IndexSeq newKey);

     };
};
```

Apêndice B: Sistema de Agentes Odyssey

Odyssey é um sistema de agentes implementado como uma biblioteca de classes Java.

B-1.- API Odyssey

O paradigma Odyssey para aplicações distribuídas móveis inclui agentes, sistemas de agentes e lugares (*places*).

B-1.1.- Agentes

Um agente é um processo que atua de forma autônoma em nome de uma pessoa ou organização. Cada agente tem seu próprio *thread* de execução. Agentes são criados por especialização da classe Odyssey *Agent* ou *Worker* (será descrito mais à frente).

Os agentes Odyssey são agentes móveis o que quer dizer que eles podem movimentar de um sistema a outro numa rede.

Para os processos Odyssey, sejam eles agentes ou lugares, seu método *live(int)* define o seu comportamento.

B-1.2.- Workers

A classe *Worker* é uma subclasse da classe *Agent*. Um *worker* é estruturado como um conjunto de tarefas e um conjunto de destinos. Em cada destino o *worker* executa a tarefa que foi estabelecida para esse destino.

Um *worker* pode manipular sua lista de tarefas em qualquer ponto durante seu percurso.

B-1.3.- Sistema de agentes.

Um sistema de agentes é uma plataforma que permite criar, gerenciar, interpretar, executar, transferir e "deletar" agentes. Um sistema de agentes detém a autoridade da região ou organização

que ele representa. O sistema de agentes Odyssey é um conjunto de classes que suportam agentes e lugares Odyssey.

B-1.4.- Lugares

Um lugar é um contexto em um sistema de agentes dentro do qual pode-se executar um agente. Um agente móvel viaja entre lugares.

Um lugar é a parte estacionária da aplicação que o usuário escreve. É necessária a existência de um lugar em cada host donde a aplicação será executada.

Em Odyssey, o lugar inicial criado quando o sistema é iniciado é distinto dos demais lugares e é derivado de uma classe nomeada *BootPlace*. Quando o *BootPlace* termina, o sistema de agentes Odyssey termina, acabando com todos os agentes e lugares que existem dentro dele.

Em geral um lugar é a porta de comunicação entre um agente visitante, o host e os recursos do host.

B-1.5.- A hierarquia de classes Odyssey

A hierarquia de classes Odyssey contém as classes que implementam o paradigma Odyssey: são duas classes que permitem construir agentes, *Agent* e *Worker*, e uma classe que implementa os lugares, nomeada *Place*.

Também existem classes de suporte para estas classes principais, que são:

Task: identifica a tarefa que um *worker* tem que executar.

Ticket: indica como e onde um agente viaja.

Means: especifica como um agente viaja.

Petition: identifica com quem um agente quer comunicar.

ProcessName: usado para gerar nomes únicos para todos os processos.

A hierarquia de classes inclui também três interfaces: *genmagic.odyssey.AgentSystem, genmagic.odyssey.Finder, e genmagic.odyssey.Transport* que permitem que o usuário possa desenvolver

qualquer uma delas com as funcionalidades que sejam necessárias para sua implementação. Odyssey fornece uma implementação *.class* para cada uma dessas interfaces, aliás no caso da interface de transporte são oferecidas duas versões, uma que usa RMI e outra que usa IIOP sobre um ORB. A classe *genmagic.odyssey.Finder* é usada para registrar e localizar lugares.

Existe uma classe auxiliar nomeada *genmagic.util.AuditTrail* que permite guardar registro de eventos de diversas categorias produzidos por um processo Odyssey.

B.2.- Questões operacionais

Para iniciar o sistema de agentes coloca-se o comando a seguir:

```
$ java genmagic.odyssey.Engine <boot config file>
```

A seguir mostramos o arquivo de configuração de *boot* o qual nos dá uma idéia das opções que podem ser configuradas.

```
# O arquivo de configuração de Boot deve especificar o nome da
# classe BootPlace que a "Engine" Odyssey deve instanciar.
# Esta é a única linha obrigatória deste arquivo.
#
BootPlace genmagic.demo.remoteCmd.CmdPlace

# A palavra-chave BootArgs especifica os argumentos passados ao
# construtor pelo BootPlace da aplicação.  Se os argumentos são
# especificados na linha de comando logo após do arquivo de
# configuração de boot quando a "Engine" Odyssey é iniciada,
# os argumentos especificados substituem os argumentos
# indicados neste arquivo. Isto permite incluir
# um conjunto de argumentos por "default" neste arquivo
#
# BootArgs pacific

# A palavra-chave CodeBase especifica uma URL que o RMI usa para ir
# buscar qualquer classe que não está disponível no CLASSPATH
# local.
# CodeBase http://Pacific/Code/

# O valor associado com a palavra-chave Finder identifica a classe
# que implementa  a interface genmagic.odyssey.Finder.  A "Engine"
# Odyssey usa a classe especificada para registrar todos os lugares
```

```
# que são criados dentro da "Engine", e localizar lugares de
# destino referenciados pelos agentes em viagem.
# Se não se acha esta palavra-chave
# a "Engine" atua como se a linha seguinte fosse especificada:
#
# Finder genmagic.odyssey.RMIFinder

# A palavra-chave Log pode ser usada para incrementar seletivamente
# a quantidade de mensagens de log que o sistema Odyssey produz.
# Cada mensagem de log no sistema Odyssey tem sido catalogado
# dentro de uma das seguintes categorias: process, travel, meeting,
# transport, finder, ou worker.
#
# Log process travel meeting transport finder worker

# A palavra-chave LogOutput redireciona as mensagens de log que o
# sistema Odyssey gera.  Os valores possíveis são none, stdout,
# file, window, e minimal, que enviam a saída ao correspondente
# output type fornecido pela classe genmagic.util.AuditTrail .  O
# valor por "default" é stdout. Se especificado 'file' as mensagens
# de log serão colocadas num arquivo nomeado Odyssey.log.
#
# LogOutput file

# Se você usa o RMIFinder  por "default" e o "Finder" nesta
# "Engine" não será o "Finder" mestre para esta região, então deve-
# se especificar a localização do "Finder" mestre para a região à
# qual esta "Engine" pertence .Para especificar o "Finder" mestre
# utilize a palavra-chave MasterFinder.
#
# MasterFinder localhost

# Se o "Finder" nesta "Engine" Odyssey será o "Finder" mestre para
# uma região, então pode-se usar a palavra-chave Region para
# especificar o nome da região para a qual o "Finder" será o
# mestre. Se não se especifica nome para a região, o nome da
# máquina local será usado como nome para a região.
#
# Region genmagic.com

# O valor associado com a  palavra-chave Transport identifica  um
# protocolo URL e a classe que implementa a interface
# genmagic.odyssey.Transport. A "Engine" Oyssey usará uma instância
# da classe especificada para transferir agentes a lugares cujos
# sistemas de agentes tenham registrado com o "Finder" que eles
# suportam o protocolo especificado. Um arquivo de configuração de
# boot  pode conter várias palavra-chave Transport. Nesse caso
# todos os protocolos especificados serão suportados.  Se não se
# acha nenhuma palavra-chave Transport no arquivo de configuração
```

```
# de boot a "Engine" se comportará como se a linha a seguir fosse
# especificada:
#
# Transport rmi genmagic.odyssey.RMITransport
```