Migração Transparente em Sistemas de Agentes usando CORBA

Bruno Richard Schulze

Tese de Doutorado

Instituto de Computação
Universidade Estadual de Campinas

Migração Transparente em Sistemas de Agentes usando CORBA

Bruno Richard Schulze

Julho de 1999

Banca Examinadora:

- Prof. Dr. Edmundo Roberto Mauro Madeira (Orientador)
  Instituto de Computação - UNICAMP
- Prof. Dr. Manoel Camillo de Oliveira Penna Neto
  PUC-PR e CEFET-PR
- Prof. Dr. Eleri Cardozo
  Faculdade de Engenharia Elétrica e Computação - UNICAMP
- Prof. Dr. Rogério Drummond Burnier Pessoa de Mello Filho
  Instituto de Computação - UNICAMP
- Prof. Dr. Nelson Luís Saldanha da Fonseca
  Instituto de Computação - UNICAMP
- Prof. Dr. Maurício Ferreira Magalhães (Suplente)
  Faculdade de Engenharia Elétrica e Computação - UNICAMP
- Prof. Dr. Ricardo de Oliveira Anido (Suplente)
  Instituto de Computação - UNICAMP

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP

Schulze, Bruno Richard

Sch85m Migração transparente em sistemas de agentes usando CORBA / Bruno Richard Schulze -- Campinas, SP : [s.n.], 1999.

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

1. CORBA (Programação de computador). 2. Programação orientada a objetos (Computação). 3. Migração. 4. Agentes de serviço de inteligência. I. Madeira, Edmundo Roberto Mauro. II. Universidade Estadual de Campinas, Instituto de Computação. III. Título.

Resumo

Este trabalho apresenta o paradigma de mobilidade em sistemas de agentes e a sua utilização de forma *explícita* determinada pelo próprio agente ou de forma *implícita* determinada por mudanças no meio externo a ele. Exploramos a inserção destas formas de mobilidade e de sistemas de agentes sobre o modelo OMG/CORBA através de um suporte de *Mobilidade* de Agentes e um suporte de *Disponibilidade* para a localização de componentes e a migração dos clientes no sentido dos componentes alvos ou dos componentes alvos no sentido dos clientes. É apresentado um estudo de caso de aplicação de agentes móveis em gerenciamento de sistemas distribuídos abertos baseado em monitoração e configuração, e mobilidade explícita e implícita de componentes.

Abstract

This work presents the paradigm of mobility in agent systems and either in its explicit form, determined by the agent itself, or in its implicit form, determined by external changes in the environment. We explore the insertion of these forms of mobility and agent systems based on the OMG/CORBA model with an Agent *Mobility* support and an *Availability* support for location of components and migration of the client in the direction of the target component or the target in the direction of the client. A case study is presented applying mobile agents in management of open distributed systems based on monitoring and configuration, and with use of explicit and implicit mobility of components.

Migração Transparente em Sistemas de Agentes usando CORBA

Este exemplar corresponde à redação final da Tese devidamente corrigida e defendida por Bruno Richard Schulze e aprovada pela Banca Examinadora.

Campinas, 12 de julho de 1999.

Edmundo R. M. Madeira

Prof.Dr. Edmundo Roberto Mauro Madeira
(Orientador)

Tese apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Doutor em Ciência da Computação.

TERMO DE APROVAÇÃO

Tese defendida e aprovada em 12 de julho de 1999, pela Banca Examinadora composta pelos Professores Doutores:

Prof. Dr. Manoel Camillo Oliveira Penna Neto
PUC-PR

Prof. Dr. Eleri Cardozo
FEEC - UNICAMP

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

Prof. Dr. Nelson Luís Saldanha da Fonseca
IC - UNICAMP

Prof. Dr. Edmundo Roberto Mauro Madeira
IC - UNICAMP

Amassa-se o barro, fazem-se os tijolos,
erguem-se as paredes.
Mas é preciso deixar lacunas
para as portas e janelas,
que tornarão a casa habitável.
Corta-se o tronco, desbasta-se a madeira,
faz-se a roda.
Mas é preciso cavar o buraco,
que permite a introdução do eixo.
Portanto, o ser produz o útil, mas é o não ser,
que o torna eficaz.
Lao Tsé

Em memória de meu irmão Werner Arthur Schulze
e de meu filho(a)

Agradecimentos

Diante da vastidão do espaço
e da imensidade do tempo,
é uma alegria para mim,
partilhar um planeta
e uma época com você.

Carl Sagan

Ao Edmundo pela orientação na realização do doutorado e deste trabalho de tese.

A minha esposa Raquel pelas sugestões.

A meus Pais e familiares pela atenção.

Aos colegas do CBPF/CNPq pelo incentivo. Em especial aos colegas do CAT e LAFEX.

Aos docentes e funcionários do IC pela interação cordial e produtiva no dia a dia.

Aos colegas discentes pela convivência.

A Patrícia e Francisco pelo trabalho em equipe.

Ao Prof. Dr. M. Mendes pelas contribuições no tema de agentes.

A UNICAMP pelo acolhimento no programa de doutorado em Ciência da Computação, através do então Instituto de Matemática, Estatística e Ciência da Computação (IMECC) e posteriormente do Instituto de Computação (IC), quando da sua criação.

Ao Centro Brasileiro de Pesquisas Físicas - CBPF / CNPq / MCT, pelo suporte, apoio institucional e interesse no desenvolvimento do trabalho e aplicações em Computação Científica. Em especial: ao grupo do CAT por todo o apoio durante o doutoramento, ao grupo do LAFEX pelas oportunidades em Computação nos grandes experimentos em Física de Partículas no Fermilab e CERN, aos então grupos ACP (Advanded Computer Program) e TPL (Tagged Photon Lab) no Fermilab, ao Grupo de Trigger do DELPHI - CERN, INFN Bologna e particularmente INFN Roma II.

Ao Departamento de Engenharia de Computação e Automação da Faculdade de Engenharia Elétrica e Computação da Unicamp, pela cooperação no contexto original da plataforma *Multiware*.

Ao GMD FOKUS (Berlim, Alemanha) e Prof.Dr.Dr.h.c. Radu Popescu-Zeletin pela acolhida e oportunidade de interação com os especialistas do grupo e de enriquecimento do trabalho.

Ao intercâmbio com o Departamento de Engenharia de Informática, da Universidade de Coimbra (Coimbra, Portugal), dentro do projeto conjunto: "Estudo de Desenvolvimento de Aplicações de Gerenciamento de Sistemas de Comunicação com garantia de Qualidade de Serviço".

Ao Marcelo Schiffer pelo companheirismo.

Aos amigos Ana Monteiro, José Roberto, Julio, M.Emilia pelo coleguismo.

Ao Paulo e o Oficina Coral pelos ensaios da voz e da caricatura. À Christina e o Cais do Canto e pela afinação da dissonância.

Ao suporte de: FAPESP, CAPES, CBPF/CNPq, PROTEM/MCT e FAEP.

A este Deus *desconhecido*.

Conteúdo

A nitidez
é uma conveniente
distribuição de luz e sombra
Goethe

CAPÍTULO 1 *Introdução* ***1***

Motivação e Abordagem **2**
Contribuições deste trabalho **3**
Estrutura de Trabalho **4**

CAPÍTULO 2 *Agentes Móveis* ***5***

Código Móvel em Processamento Distribuído **5**
Agentes como serviços **6**
Sequencial e Concorrente **6**
Paradigmas e Tecnologias de Código Móvel **8**
Estimativas de Custo de Tráfego **11**
Paradigmas e Tecnologias **15**
Agentes **18**
Mobilidade **19**
Autonomia e flexibilidade **20**
Mobilidade e segurança **20**
Agentes móveis sobre CORBA **21**
Trabalhos Relacionados em Tecnologias de Agentes Móveis **23**
Aplicações **28**
Aplicações de Interesse **29**

CAPÍTULO 3 *Um Serviço de Disponibilidade Para Migração de Agentes* ***31***

Arquitetura Orientada a Serviços **31**
Serviços **33**
Agência Orientada a Serviços **34**
Ciclo de Vida e Persistência **35**
Mobilidade **35**
Disponibilidade **35**
Domínios Distribuídos Dinâmicos **38**
Migração Transparente em Sistemas de Agentes **40**

Agência **43**
Serviço de Disponibilidade **45**
Oferta de Disponibilidade ***48***
Propriedades Dinâmicas ***50***
Suporte de Mobilidade **50**
Serviços de Armazenamento **52**
Diretórios ***52***
Repositórios ***54***

CAPÍTULO 4 *Detalhes de Implementação* ***55***

Descrição Geral do Sistema de Agentes **55**
Facilidade de Interoperabilidade entre Sistemas de Agentes ***56***
Conexão de um Agente Cliente a um Agente Servidor ***58***
Implementação a partir de CORBA **60**
Implementação a partir de uma Plataforma de Agentes **63**
Comentários **64**
Especificações CORBA IDL Relacionadas ***65***
Definição de Classes **67**
Transparência ***67***
Disponibilidade ***68***
Seleção ***69***
Cache ***70***
Mobilidade ***71***
Nomes ***72***
Trader ***75***

CAPÍTULO 5 *Resultados* ***77***

Prototipagem na Passagem de Código **77**
Prototipagem na Migração de Componentes **79**
Fase de Emergência ***79***
Fase de Recuperação ***80***
Comparação de Desempenho com Java ***86***
Comentários sobre a (Re)distribuição de Serviços **88**
Prototipagem em Gerenciamento **89**

CAPÍTULO 6 *Aplicação: Gerenciamento de Serviços usando Agentes Móveis em Ambiente CORBA* ***91***

Introdução **91**
Metodologia **92**
Esquemas de Gerenciamento ***92***
Paradigma de Agentes Móveis ***93***
Ajustes ***94***
Trabalhos Relacionados ***96***
Arquitetura **97**
Infra-estrutura de Gerenciamento ***97***
Agentes de Gerenciamento ***98***

Infra-estrutura de Ajuste ***99***
Serviço de Mobilidade ***100***
Aspectos de Implementação **100**
Infra-estrutura de Gerenciamento ***100***
Infra-estrutura de Ajuste ***104***
Resultados **105**
Comentários **107**

CAPÍTULO 7 *Conclusão* ***109***

Aspectos de Implementação ***110***
Aplicação em Gerenciamento ***110***
Aplicação em Computação Móvel ***111***
Trabalhos Futuros ***111***

REFERÊNCIAS ***113***

Do Autor Neste Trabalho **113**
Outras do Autor **114**
Demais Autores **115**
Da Internet **121**

APÊNDICE A *CORBA em JAVA* ***123***

CORBA em Java: OrbixWeb e javaIDL **123**
Inicialização do ORB **127**
Serviço de Nomes **130**
Adicionando Objetos ao Espaço de Endereçamento ***130***
IDL do Serviço de Nomes **133**

APÊNDICE B *MASIF* ***135***

IDL CfMAF **135**
Interface MAFAgentSystem **139**
Interface do MAFFinder **151**

APÊNDICE C *PortableServer* ***159***

Diagrama UML **159**
IDL do PortableServer **160**

APÊNDICE D *IT_daemon* ***163***

ÍNDICE REMISSIVO ***165***

LISTA DE FIGURAS

Figura 2.1 Processamento distribuído com execução concorrente e sequencial. 7
Figura 2.2 Cliente-Servidor (CS) 9
Figura 2.3 Avaliação Remota (REV) 9
Figura 2.4 Código Sob Demanda (COD) 9
Figura 2.5 Agentes Móveis (MA) 10

Figura 3.1 Uma Aplicação Serviço-Orientada. 32
Figura 3.2 Agentes estacionários e móveis 33
Figura 3.3 Uma agência baseada no modelo CORBA. 34
Figura 3.4 Plataforma Multiware. 34
Figura 3.5 Agências interligadas pelo ORB 36
Figura 3.6 Utilização de um serviço de catálogo genérico 37
Figura 3.7 Serviços de catálogos federados e hierárquicos 38
Figura 3.8 Traçado de um agente baseado em proxies 38
Figura 3.9 (a) federação de Catálogos. (b) um domínio de agente anexado a um domínio de Catálogo 39
Figura 3.10 Migração Transparente: a) localização e b) (re)configuração dinâmica 41
Figura 3.11 Migração Transparente: Migração Final 42
Figura 3.12 A migração ao nível de requisições e re-chamadas entre pares de agentes 42 (43)
Figura 3.13 Nó, agência, cluster(s) de serviços locais e cluster(s) de agentes 43
Figura 3.14 Componentes de uma agência 44
Figura 3.15 Serviço de Disponibilidade sobre CORBA 45
Figura 3.16 O macro modelo do serviço de Disponibilidade 46
Figura 3.17 Diagrama de Seqüências da Migração 47
Figura 3.18 Diagrama de classes do suporte a mobilidade 51
Figura 3.19 Fluxograma de execução do suporte a mobilidade 51
Figura 3.20 Estruturação do espaço de Nomes 53
Figura 3.21 Trader e suas interfaces, versão completa e stand-alone 53

Figura 4.1 Aspecto Genérico de uma Agência com o Serviço de Disponibilidade. 56
Figura 4.2 Estrutura de Sistemas de Agentes e sua interconexão via Rede. 57
Figura 4.3 Arquitetura de uma Região. 57
Figura 4.4 Relacionamento do MAF com um ORB. 57
Figura 4.5 Serviços e Facilidades CORBA relacionados a uma Agência. 58
Figura 4.6 Conexão Cliente / Servidor com gerente de ativação (orbixd). 60
Figura 4.7 Integração de agências baseadas no Orbixd. 61
Figura 4.8 Serviços de Disponibilidade e Mobilidade registrados com o orbixd. 62

Figura 4.9 Serviço de Disponibilidade é ativado como servidor persistente do ORB e agente estático da plataforma de agentes. 63

Figura 5.1 Seqüência IDL via ethernet em ambiente não dedicado: (a) cliente/servidor co-localizados e (b) cliente/servidor remotos. 78
Figura 5.2 Seqüência de instanciação e comunicação de objetos: 1, 2, 3, 4, 1, e assim por diante 81
Figura 5.3 Diferentes alocações de hospedeiros 83

Figura 6.1 Esquemas de Gerenciamento 93
Figura 6.2 Ambiente de Agentes Distribuídos 94
Figura 6.3 Migração Transparente 95
Figura 6.4 O macro modelo da estrutura de gerenciamento 99
Figura 6.5 Pontos de monitoração em um processo e o cascateamento de filtros. 101
Figura 6.6 Cenário de ativação de sensores e a interface IDL dos filtros 101
Figura 6.7 Agente de ativação de filtros, ags tipo 6. 102
Figura 6.8 Agente de medida de ocupação de cpu e memória, ags tipo 7. 103

Figura A.1 CORBA simplificado. 124
Figura A.2 Comunicação Servidor / Servidor em CORBA. 124

Figura C.1 Diagrama em UML da parte principal do PortableServer 159

LISTA DE TABELAS

Tabela 2.1 Paradigmas de Código Móvel [Vigna98] 10
Tabela 2.2 Termos e definições usados nas equações de estimativa de custo tráfego 11
Tabela 2.3 Uma classificação de mecanismos de mobilidade [Vigna98] 16
Tabela 2.4 Relação entre paradigmas e tecnologias [Vigna98] 17
Tabela 2.5 Características gerais de Agentes 19
Tabela 2.6 Paradigmas de Código Móvel sobre CORBA. 21

Tabela 3.1 Tipos básicos de disponibilidade. 48
Tabela 3.2 Tipos de ofertas de disponibilidade comuns a agentes e agências. 49
Tabela 3.3 Tipos adicionais de ofertas de disponibilidade para agentes e agências. 49
Tabela 3.4 Organização do Serviço de Nomes para agentes e agências. 52

Tabela 4.1 Estratégia geral na migração transparente. 59
Tabela 4.2 Especificações IDL Mínimas: ORB e PortableServer. 65
Tabela 4.3 Especificação IDL Mínima do CfMAF. 66

Tabela 5.1 Comparação entre hospedeiros e benchmark. 83
Tabela 5.2 Medidas de execução. 84
Tabela 5.3 Semelhante a Tabela 5.1, mas em máquinas diferentes. 87
Tabela 5.4 Resultados para as configurações (a) e (d) da Figura 5.3. 87

Tabela 6.1 Sensores Modelados. 97
Tabela 6.2 Agentes de gerenciamento utilizando os filtros propostos na Tabela 6.1. 98
Tabela 6.3 Tabela de níveis de decisão para ocupação de cpu, memória e throughput. 105
Tabela 6.4 Relatório gerado por um Agente. 106

ACRÔNIMOS

API	Application Programming Interface
ANSA/APM	Advanced Network Systems Architecture / Architecture Projects Management
BOA	Basic Object Adaptor
CORBA	Common Object Request Broker Architecture
CSCW	Computer Supported Cooperative Work
DII	Dynamic Invocation Interface
DSI	Dynamic Skeleton Interface
GIOP	General Inter-Orb Protocol
GMD/FOKUS	Forschungszentrum Informationstechnik GmbH / Forschungsinstitut für offene Kommunikationssysteme (German National Research Center for Information Technology / Research Institute for Open Communitation Systems)
HTTP	Hipper Text Transfer Protocol
IIOP	Interoperable Inter-Orb Protocol
IOR	Interoperable Object Reference
JDK	Java Development Kit
JDMK	Java Dynamic Management Kit
JMAPI	Java Management Application Programming Interface
JVM	Java Virtual Machine
MASIF	Mobile Agent System Interoperability Facility
OMA	Object Management Architecture
OMG	Object Management Group
ORB	Object Request Broker
RM-ODP	Reference Model for Open Distributed Processing
OMT	Object Modeling Technique
POA	Portable Object Adaptor
RMI	Remote Method Invocation
RPC	Remote Procedure Call
SNMP	Simple Network Management Protocol
TCP/IP	Transmission Control Protocol / Internet Protocol
UML	Unified Modeling Language
WWW	World Wide Web

CAPÍTULO 1

Introdução

Escrever é fácil:
você começa com uma letra maiúscula
e termina com um ponto final.
No meio você coloca as idéias.
Pablo Neruda

A tecnologia de objetos em computação distribuída permite simplificar a programação de objetos componentes separadamente da distribuição e configuração destes componentes. O paradigma de mobilidade de componentes permite estender aspectos de configuração e distribuição, gerando contribuição significativa em aplicações de serviços distribuídos abertos.

O trabalho se insere no contexto de arquiteturas multi-agentes, propondo um suporte de migração transparente. Na apresentação do trabalho procuramos padronizar termos e conceitos [Schulze97-1, Vigna98, OMG97-2] na tentativa de auxiliar em novas definições. Os termos e conceitos básicos considerados são: *componentes*, *interações* e *sítios*. *Componentes* podem ser do tipo: *código*, *recurso* e *computacional*, onde este último tipo corresponde a uma instância em execução. *Interações* são eventos envolvendo dois ou mais *componentes*. *Sítios* hospedam e viabilizam a execução de *componentes computacionais*, representando a noção intuitiva de localização.

A arquitetura apresentada é orientada a *agentes* e baseada no modelo orientado a objetos do OMG/CORBA [CORBA98]. São acrescentados de forma associada um suporte de *Mobilidade* e um suporte de *Disponibilidade* para tratar migração de *agentes* de forma *explícita* e *implícita* (ou transparente). De forma resumida, um *agente* age autonomamente representando uma pessoa ou organização, a migração *explícita de um agente* é aquela determinada somente por ele em um contexto estático, enquanto a migração *implícita* de um *agente* é aquela determinada por mudanças num contexto dinâmico e *agentes* externos. Ressaltamos que num caso de uso genérico os dois tipos de migração podem estar intimamente associados, não sendo claramente separáveis.

De forma genérica, um componente cliente requisita um componente externo e migra no sentido do componente servidor selecionado. O suporte de *Disponibilidade* atua na *seleção* do componente servidor e repassa ao suporte de *Mobilidade* migrar o cliente ao servidor. O serviço de Disponibilidade é baseado em serviços CORBA tais como *Trader* e *Nomes*.

Uma *seleção* utiliza uma fase preliminar baseada numa consulta por componentes disponíveis dentro de uma faixa de disponibilidade. A partir do conjunto retornado é possível uma *seleção*

baseada numa avaliação de disponibilidade de componentes baseada no tempo de resposta e número de tentativas. O serviço de Disponibilidade, na média de uso, deve ser ágil e de processamento leve.

Uma extensão do trabalho é a classificação de interfaces e métodos de disponibilidade de uso geral ou especializados, como por exemplo, em gerenciamento de sistemas distribuídos abertos. Agentes móveis envolvem (re)configuração de ambientes distribuídos particularmente quando são autônomos e flexíveis, isto é, capazes de redefinir seu plano de execução.

Motivação e Abordagem

Diversos mecanismos e facilidades foram concebidos em passado recente para migração de código. Existem mecanismos de migração de processos e objetos suportados ao nível de sistemas operacionais distribuídos. Uma revisão de sistemas suportando migração de objetos pode ser encontrado em [Nuttall94].

A migração de objetos permite granularidade de mobilidade mais fina em relação à migração de processos. Nem todos os sistemas fornecem migração completamente transparente. Um exemplo de sistema provendo migração transparente de objetos é o COOL [Vigna98], implementado sobre o sistema operacional *Chorus* [Rozier88]. É capaz de mover objetos sem intervenção ou conhecimento do usuário. A abordagem para a migração de objetos (e processos) considera, em princípio, hospedeiros fracamente acoplados, sistemas distribuídos de pequena escala, elevada largura de banda, latência previsivelmente pequena, confiabilidade, e frequentemente homogeneidade [Vigna98].

Sistemas de código móvel [Vigna98] exibem diversas inovações com respeito às abordagens existentes de migração de objetos. Endereçam sistemas distribuídos heterogêneos em larga escala, controlados por autoridades diferentes, com diferentes níveis de confiabilidade, e largura de banda desde sem-fio (*wireless*) até ótico. Oferecem ao programador o controle sobre a mobilidade.

Sistemas atuais de agentes móveis [GrassHopper98, Aglets, Voyager97-1/2, Odyssey97] usam a abordagem geral de agentes móveis como agentes somente clientes seguindo seu roteiro e interagindo com serviços e objetos estáticos. Ressaltamos a importância de agentes com métodos públicos que podem ser requisitados por outros agentes externos e conseqüentemente se moverem devido a uma requisição e/ou à migração de um componente externo ao qual estejam conectados.

Contribuições deste trabalho

Deseja-se unir a mobilidade implícita existente na migração de objetos com a mobilidade explícita disponível nos sistema atuais de agentes móveis. Nossa abordagem é explorar CORBA como estrutura geral de componentes, transformando objetos em agentes e adicionando o suporte para a mobilidade explícita e implícita. Adicionar a mobilidade a CORBA é uma fusão da estrutura de objetos e infra-estrutura definidos no modelo com o paradigma de agentes móveis.

Construímos um serviço de Disponibilidade associado a um serviço de mobilidade para suporte à migração transparente de componentes computacionais. Componentes podem encapsular *hardware* ou *software*. Serviços de Catálogo, como o OMG/Trader [OMG96, Trader95, Bearman96], participam no processo de localização, especialmente quando diferentes domínios constituem o domínio da aplicação.

Atingimos um grau de transparência na migração de agentes, procurando ir na direção do idealmente proposto no modelo de referência ODP (*Open Distributed Processing*) [RMODP95] para transparência de localização e migração, onde um cliente não toma ciência de qualquer alteração na localização / migração de um servidor.

A transparência de migração é considerada na forma de dois casos básicos de interação: agente-agência e agente-agente. Numa interação agente-agência, o agente cliente solicita (ao serviço de Disponibilidade) uma agência a partir de propriedades com valores e migra para uma qualquer que satisfaça as restrições. Esta solicitação pode ser iniciada a partir de uma métrica insatisfatória do agente (ex. desempenho). Numa interação agente-agente, a migração transparente é tratada na forma de uma exceção entre cliente e servidor, solicitando (ao serviço de Disponibilidade) um determinado agente de serviço a partir de propriedades com valores. A partir da nova referência de agente (retornada pelo serviço de Disponibilidade) a comunicação é restabelecida após a migração na direção do objetivo.

Apresentamos uma aplicação de gerenciamento desenvolvida sobre o paradigma de agentes móveis. Agentes são usados em gerenciamento de monitoração enquanto o serviço de Disponibilidade participa no gerenciamento de configuração, explorando o paradigma de mobilidade em gerenciamento de sistemas heterogêneos distribuídos abertos. Um conjunto de agentes é definido para explorar o ambiente gerenciado utilizando um *detalhamento sucessivo* de potenciais problemas. A execução considera um ambiente distribuído aberto baseado em objetos CORBA. Uma extensão é apresentada para ajustar o sistema gerenciado através de reconfiguração com a ajuda do serviço de Disponibilidade.

Estrutura do Trabalho

A tese está estruturada de acordo com a descrição de capítulos a seguir:

Capítulo 2: Apresenta conceitos sobre Agentes Móveis numa Arquitetura Comum de Intermediador de Requisições de Objetos (CORBA) e alguns trabalhos relacionados.

Capítulo 3: Propõe um serviço da disponibilidade para um ambiente móvel do agente.

Capítulo 4: Descreve detalhes da implementação do serviço de Disponibilidade em uma arquitetura orientada a serviços.

Capítulo 5: Reporta resultados da arquitetura proposta.

Capítulo 6: Descreve uma aplicação em gerenciamento de serviços baseada em agentes móveis e uma infra-estrutura de ajuste do sistema gerenciado.

Capítulo 7: Contém observações finais.

CAPÍTULO 2

Agentes Móveis

Não vemos as coisas
como elas são
mas como nós somos.
Anais Nin

Este capítulo é uma revisão de tópicos relacionados aos paradigmas e às tecnologias de agentes móveis. Está dividido nas seguintes seções:

- 2.1 Código Móvel em Processamento Distribuído, página 5,
- 2.2 Paradigmas e Tecnologias de Código Móvel, página 8,
- 2.3 Agentes, página 18,
- 2.4 Agentes móveis sobre CORBA, página 21,
- 2.5 Trabalhos Relacionados em Tecnologias de Agentes Móveis, página 23 e
- 2.6 Aplicações, página 28.

2.1 Código Móvel em Processamento Distribuído

O modelo tradicional de processamento distribuído envolve processos estaticamente localizados transferindo dados e/ou comandos. Os dados e os comandos são a parte móvel de uma computação enquanto os programas são estáticos. Há um número de cenários de computação em rede que não podem ser eficazmente endereçados por paradigmas de interação estática [Baldi98, Goldszmidt96, Vigna98]. Um paradigma alternativo é o de agentes móveis com vantagem de desempenho em sistemas com banda de comunicação estreita e latência alta.

De acordo com OMG/MASIF [OMG97-2] um *agente* é um programa de computador que age autonomamente em nome de uma pessoa ou de uma organização. Um *agente móvel* não está limitado ao sistema de sua instanciação. Pode migrar de um sistema a outro que contenha o objeto com o qual o agente deseja interagir, podendo utilizar os serviços de objeto do destino. Uma *agência*, ou sistema de agentes, é uma plataforma que pode criar, interpretar, executar, transferir e terminar agentes. Como um agente, está associado a uma autoridade que identifica a pessoa ou organização a qual representa. Um hospedeiro pode conter um ou mais sistemas de agentes.

Uma solução interessante para o problema de latência é *caching* inverso de dados [Goldszmidt96] onde a aplicação é movida para onde os dados são encontrados ou produzidos. No *caching* normal de dados, move-se os dados para onde são consumidos, sendo eficaz quando há um elevado grau de localidade de uso e de consistência. Não é próprio para dados distribuídos altamente voláteis, que são rapidamente desatualizados, tornando o *cache* inconsistente. Por exemplo, em instrumentos de monitoração remota, se as medidas mudam frequentemente, é mais eficiente mover o código para perto da coleta de dados. Obviamente, *caching* inverso só é vantajoso enquanto o custo de mover o código é menor do que o ganho obtido.

Agentes móveis permitem a transferência implícita da informação, carregando todo seu estado interno, eliminando a necessidade de etapas separadas de comunicação. Em segundo, as tarefas como gerenciamento de rede, recuperação de informação e workflow, se adaptam naturalmente a este saltar-executar-saltar de agentes móveis. Um agente migra para uma máquina, executa uma tarefa, migra para outra máquina, executa outra tarefa que pode ser dependente da saída da tarefa precedente, e assim por diante. Finalmente, uma aplicação pode dinamicamente distribuir seus componentes computacionais ao começar a execução.

Agentes móveis são uma extensão interessante do modelo tradicional cliente/servidor, onde cliente e servidor podem se reprogramar, estendendo a funcionalidade que uma aplicação pode oferecer. Uma vantagem é a não necessidade da prévia configuração dos hospedeiros destinatários. Estado e código devem ser mantidos leves utilizando bibliotecas do ambiente local quando possível. Em hospedeiros móveis é especialmente importante que agentes sejam leves, em atenção às limitações de recursos e transmissão de dados. *Browsers* se tornam ambientes locais favoráveis em hospedeiros móveis, especialmente incorporando ORBs.

Agentes como serviços

Uma aplicação orientada a serviço deve utilizar serviços disponíveis tanto quanto possível e inicializar novos serviços quando não estão disponíveis. Como em situações do cotidiano, deve-se empregar preferencialmente serviços de prateleira e confeccionar somente o que não está disponível. O desenvolvimento e execução de aplicações orientadas a serviço envolve contratação de serviços, e distribuição e inicialização de serviços não encontrados. Durante a execução, uma aplicação deve consultar e responder a serviços, cuidando da eventual substituição de serviços falhos. Finalmente uma aplicação deve permitir uma interrupção programada em algum ponto. Os serviços específicos para suportar a arquitetura proposta são redistribuídos durante a execução, atendendo a balanceamento de carga e caching inverso [Goldszmidt96].

Sequencial e Concorrente

Em processamento distribuído, agentes móveis são considerados migrando de nó para nó, como executando uma tarefa seqüencial. Na Figura 2.1, uma tarefa distribuída pode empregar a execução seqüencial e concorrente, especialmente se considerarmos a facilidade de código móvel. A execução concorrente é possível pela execução simultânea em hospedeiros diferentes enquanto a execução seqüencial é possível passando um agente como uma ficha a cada etapa na execução.

Ambientes de processamento distribuído baseados no paradigma de agentes móveis apresentam uma situação particular que surge quando um agente participa em tarefas de longa duração e tarefas onde parte do código pode adormecer esperando sincronização ou requisições esparsas. A situação anterior sugere o agente ser removido do ambiente de execução e armazenado em um repositório de execução. Este repositório de execução pode simplesmente armazenar uma proxy contendo o estado do agente e uma referência à classe objeto correspondente.

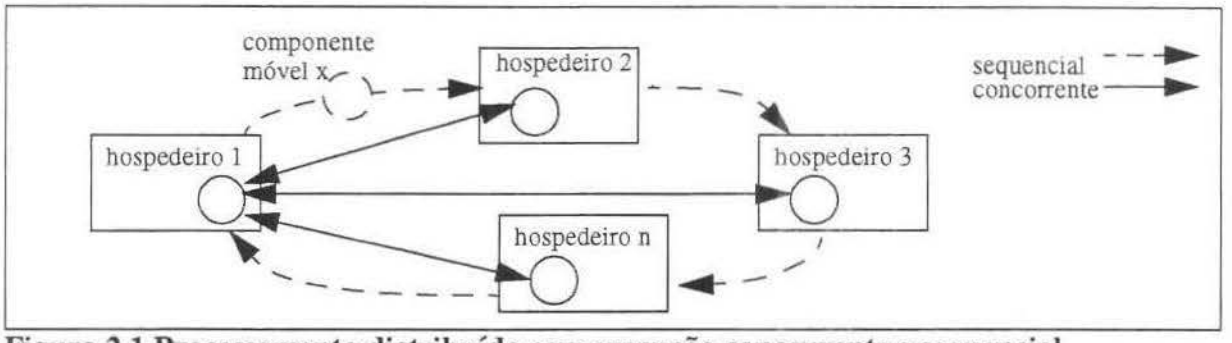


Figura 2.1 Processamento distribuído com execução concorrente e sequencial.

Cada instanciação nova da mesma classe de objeto origina um novo agente com seu próprio estado separado de execução. Manter um agente continuamente na memória pode não ser nem necessário nem desejado. De fato se um agente está inativo deve ser removido da memória para retornar os recursos de execução ao sistema. Isto sugere que um agente ao ficar inativo deve armazenar o seu estado e o local onde encontrar as suas classes de objetos.

A possibilidade de mover um componente segundo uma trajetória, como numa execução sequencial, permite reduzir o custo de tráfego de mensagens que circulam na rede. Este aspecto é explorado mais adiante neste capítulo e ao longo deste trabalho, estando presente na escolha das aplicações.

2.2 Paradigmas e Tecnologias de Código Móvel

Os conceitos de arquitetura apresentados em [Vigna98] são usados neste trabalho a fim de manter o uso de uma padronização de termos ao referir elementos de código móvel. Assim, os elementos de arquitetura considerados são: *componentes*, *interações* e *sítios*. *Componentes* são os elementos de uma arquitetura e classificados em *código*: o receituário para uma computação em particular, *recursos*: os dados ou dispositivos que participam em uma computação, e *componentes computacionais*: os executores ativos de uma computação descrita por um código correspondente. *Interações* são os eventos que envolvem dois ou mais componentes. *Os sítios* são os locais que suportam a execução de componentes computacionais. Interações entre componentes residindo num mesmo sítio são consideradas de custo mais baixo que interações entre componentes em sítios distintos. Uma computação só se realiza quando estão localizados no mesmo *sítio* o *receituário* descrevendo a computação, os *recursos* usados durante a computação, e o *componente computacional* responsável pela computação.

Paradigmas de projeto são descritos em termos de padrões de interação que definem a realocação de componentes e a coordenação entre eles, sendo ambas necessárias para o desempenho de um serviço. O cenário a ser considerado adiante é o de um componente computacional A, localizado em um sítio S_A que necessita dos resultados de um serviço. Assume-se também a existência de um outro sítio S_B que é envolvido na realização do serviço.

São identificados três paradigmas que exploram código móvel: *avaliação remota* (REV), *código sob demanda* (COD) e *agentes móveis* (MA). Estes paradigmas são caracterizados pela localização de componentes antes e depois da execução do serviço, pelo componente computacional responsável pela execução do código, e pela localização onde a computação efetivamente acontece.

Cliente-Servidor. O paradigma cliente-servidor é amplamente conhecido e utilizado. Neste paradigma, ver Figura 2.2, um componente computacional *B* (o servidor) oferece um conjunto de serviços e está localizado no *sítio* S_B. Os *recursos* e o *receituário* necessários para a execução do serviço estão sediados também no *sítio* S_B. O componente cliente *A*, localizado em S_A, requisita a execução de um serviço através de uma interação com o componente servidor S_B. Em resposta, *B* desempenha o serviço requisitado executando o *receituário* correspondente e acessando os recursos co-localizados com *B*. Em geral, o serviço gera algum tipo de resultado que é entregue de volta ao cliente por uma interação adicional. Um servidor pode depender de outros componentes para desempenhar partes do serviço requisitado ou para recuperar parte dos dados requisitados, mas, neste caso, o servidor estará atuando como cliente em outra interação cliente-servidor. Do ponto de vista do cliente original, o servidor possui todos os dados e receituário.

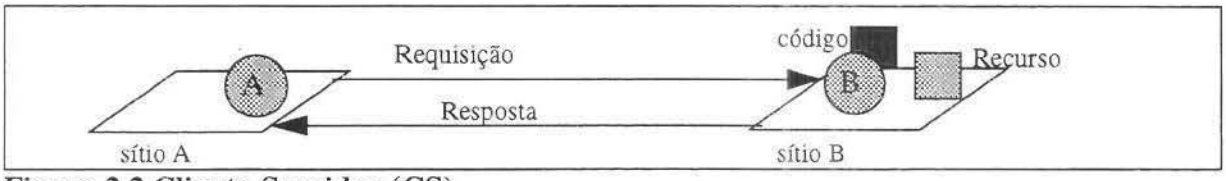


Figura 2.2 Cliente-Servidor (CS).

Avaliação Remota. No paradigma de avaliação remota, ver Figura 2.3, um componente *A* tem o *receituário* necessário para desempenhar o serviço mas não dispõe dos *recursos* requeridos, que ocorrem de estar em um *sítio* remoto S_B. Conseqüentemente, A envia o *receituário* do serviço a um componente computacional *B* localizado no *sítio* remoto. *B*, em contrapartida, executa o código utilizando os *recursos* disponíveis lá. Uma interação adicional retorna o resultado para *A*. Com esta definição, REV pode ser visto simplesmente como um caso especial do paradigma CS em que o servidor exporta um serviço *executar_código* que recebe um fragmento de código como parâmetro.

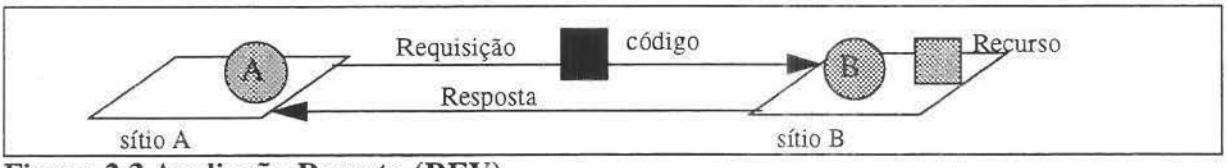


Figura 2.3 Avaliação Remota (REV).

Código Sob Demanda. No paradigma de código sob demanda, ver Figura 2.4, um cliente C interage com o componente *A* que é capaz de acessar os *recursos* que necessita e que estão co-localizados com A em S_A. Entretanto, nenhuma informação sobre como manipular tal *recurso* está disponível em S_A. Portanto, *A* interage com um componente *B* em S_B através da requisição do *receituário* do serviço, que também está localizado em S_B. Uma segunda interação ocorre quando *B* entrega o *receituário* a *A*, que pode subseqüentemente executá-lo.

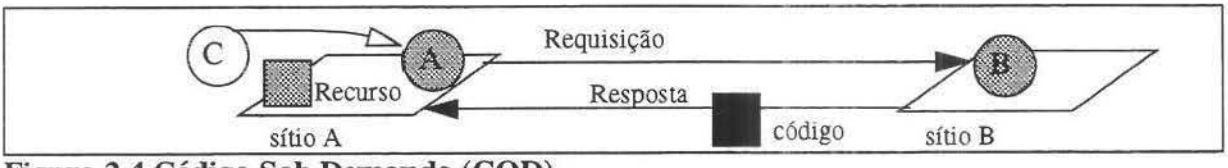


Figura 2.4 Código Sob Demanda (COD).

Agentes Móveis. No paradigma de agentes móveis, ver Figura 2.5, o receituário é parte de A, que está inicialmente hospedado em S_A, mas alguns dos recursos requeridos estão localizados em S_B. Portanto, A migra para S_B carregando consigo o receituário e possivelmente alguns resultados intermediários. Após ter se movido para S_B, A completa o serviço usando os recursos disponíveis em S_B. O paradigma de agentes móveis é diferente dos demais paradigmas de código móvel no que as interações associadas envolvem a mobilidade de um componente com-

putacional existente. Ou seja, enquanto com REV e COD o foco está na transferência de código entre os componentes, em MA um componente computacional inteiro é movido para o destinatário, junto com seu estado, o código necessário, e alguns recursos requeridos para realizar a tarefa.

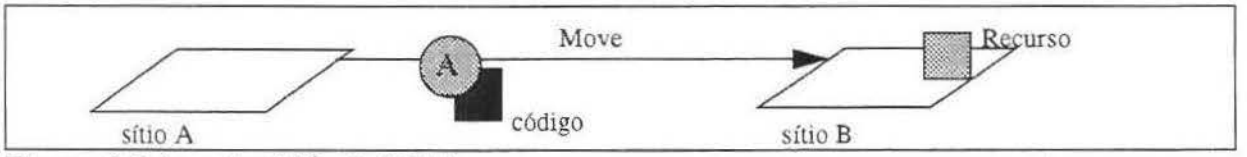


Figura 2.5 Agentes Móveis (MA).

A Tabela 2.1 resume as descrições anteriores apresentando a localização de componentes *antes* e *depois* da execução do serviço. Para cada paradigma, o componente computacional em **negrito** é aquele que executa o código. Os componentes em *itálico* são aqueles que foram movidos.

Tabela 2.1 Paradigmas de Código Móvel [Vigna98]

Paradigmas	Antes		Depois	
	Sítio A	**Sítio B**	**Sítio A**	**Sítio B**
Cliente-Servidor (CS)	A	receituário recurso B	A	receituário recurso **B**
Código sob Demanda (COD)	recurso A	receituário B	*receituário* recurso **A**	B
Avaliação Remota (REV)	receituário A	recurso B	A	*receituário* recurso **B**
Agentes Móveis (MA)	receituário A	recurso	--	*receituário* recurso **A**

Observações. Uma solução baseada em componentes móveis não é necessariamente melhor que uma baseada em componentes não móveis. Há aplicações que se adaptam melhor a um ou a outro paradigma, enquanto algumas se beneficiam da união dos dois. A seguir, resumimos alguns aspectos relevantes a serem considerados na escolha de utilização de agentes móveis e não móveis:

- considere a relevância do tráfego de mensagens e a sua minimização;
- descreva o problema considerando a localização dos dados e o tamanho dos agentes;
- considere se o problema pode ser particionado em problemas menores e planos topológicos separados;

- considere se o problema e a solução podem ser particionados contendo mobilidade e não mobilidade em planos topológicos separados;
- considere agentes móveis associados à execução de tarefas seqüências;
- considere agentes estáticos associados à execução de tarefas concorrentes.

Estimativas de Custo de Tráfego

Usamos uma estimativa de custo de tráfego apresentada em [Baldi98] para cada um dos paradigmas mencionados anteriormente, isto é: CS, REV, COD e MA. Na Tabela 2.2 apresentamos termos e definições utilizados nas estimativas de custo de tráfego discutidas a seguir.

Tabela 2.2 Termos e definições usados nas equações de estimativa de custo tráfego

Termo	Definição	Termo	Definição
N	# de unidades gerenciadas	I_q	tamanho da requisição
Q	número de consultas	$\bar{I}_q$	valor médio de I_q
S	tamanho do estado	I_f	requisição de código
C	tamanho do código	I_R	tamanho da requisição de registro remoto
U	número de execuções	R_{qn}	tamanho da resposta da $q^{ésima}$ requisição à $n^{ésima}$ unidade
X	tamanho dos dados	$\bar{R}_{qn}$	valor médio R_{qn}
T	custo de tráfego	η	*overhead* da função de protocolo nas requisições
p	paradigma	$\bar{\eta}$	*overhead* da função de protocolo nas respostas
		ΔO	diferença de overhead na transmissão dos resultados

A equação a seguir apresenta a forma genérica de custo de tráfego (T) durante um número de execuções (U), considerando os paradigmas cliente-servidor, avaliação remota e agentes móveis.

$$T_p(U) = UT_p, \forall p \in \{CS, REV, MA\} \ . \qquad \text{(eq. 1)}$$

Para código sob demanda, p=COD, temos a forma alterada a seguir,

$$T_{COD}(U) = T_{COD,\,inicializacao} + UT_{COD,\,estavel} \ . \qquad \text{(eq. 2)}$$

A etapa de inicialização se refere a distribuição inicial de código para os sítios onde o código não se encontra. A etapa estável se refere a utilização posterior à distribuição inicial e portanto o código pode ser encontrado localmente em cada sítio considerado.

Associado ao tráfego de uma mensagem X temos uma parcela devida a um *overhead*, O(X), diferente em cada paradigma. A forma geral da função *overhead*, $\eta(X)$, ou simplesmente η, é apresentada a seguir. O termo $\alpha(X)$ representa o *overhead* introduzido pela troca de mensagens na fase de estabelecimento da conexão num protocolo orientado a conexão. O termo $\beta(X)$

representa o *overhead* introduzido pelo encapsulamento das mensagens. X é o tamanho da mensagem original e X' é o tamanho final efetivamente enviado com os acréscimos para o envio.

Podemos representar a função *overhead* como:

$$\eta(X) = \frac{X'}{X} \text{ onde } X' = \alpha(X) + \beta(X)X \text{ e portanto } \eta_p(X) = \frac{\alpha_p(X)}{X} + \beta_p(X) .$$

Uma forma alternativa de representação é $\eta_p(X) = \frac{X + O_p(X)}{X}$.

A seguir transcrevemos as equações para a estimativa de custo de tráfego de cada um dos diferentes paradigmas. Na eq. 3, para CS, é considerado que as requisições que partem de um cliente tem tamanho I_q, e Q mensagens devem ser enviadas para cada um de N nós para a realização da tarefa. A unidade *n* responde a q^{ésima} requisição com uma resposta de tamanho R_{qn}. Considera-se funções de *overhead* distintas para requisições e respostas.

$$T_{CS} = U \sum_{n=1}^{N} \sum_{q=1}^{Q} (\eta_{CS} I_q + \tilde{\eta}_{CS} R_{qn}) . \quad \text{(eq. 3)}$$

Na eq. 4, para REV, as Q requisições estão contidas no fragmento de código de tamanho C_{REV} enviado para a unidade *n*. A avaliação remota de código produz os Q resultados R_{qn} que são enviados de volta ao cliente. Este par de interações ocorre para cada um dos N nós.

$$T_{REV} = U \sum_{n=1}^{N} \left(\eta_{REV} C_{REV} + \tilde{\eta}_{REV} \sum_{q=1}^{Q} R_{qn} \right) . \quad \text{(eq. 4)}$$

Na eq. 5, para MA, o cliente envia um componente móvel que visita cada um dos N nós e coleta informação localmente. O código e a parte do estado necessária para a sua execução estão representados por C_{MA} separados da parte do estado relevante à aplicação e denotada por S_{MA}. S_{MA} pode crescer ao longo da trajetória de componentes. O último termo na eq. 5 se refere ao custo de tráfego gerado pelos Q resultados R_{qn} enviados de volta ao cliente. Uma alternativa de projeto é fazer o agente retornar ao cliente inicial e gerar a comunicação intermediária com o cliente, fazendo sumir este último termo na eq. 5. Na eq. 6, o estado carregado a caminho do nó *(n)* é relativo aos *(n-1)* nós visitados anteriormente. Para n=1, o estado é zero pois o componente ainda não carrega consigo nenhum estado quando se move para o primeiro nó. O estado deve ser processado pelo componente e mantido reduzido de forma a não crescer simplesmente na proporção de N^2.

$$T_{MA} = U \sum_{n=1}^{N} \left(\eta_{MA}(C_{MA} + S_{MA,n}) + \tilde{\eta}_{MA} \sum_{q=1}^{Q} R_{qn} \right) , \quad \text{(eq. 5)}$$

$$\text{onde } S_{MA,n} = \begin{cases} 0, para\ n=1 \\ \sum_{m=1}^{n-1} \sum_{q=1}^{Q} R_{qm}, para\ n>1 \end{cases} . \qquad \text{(eq. 6)}$$

Para COD, conforme a eq. 2, temos um termo relativo a etapa de inicialização e um termo relativo a etapa estável. Em COD, o componente de código é enviado para fazer as Q operações localmente. I_{COD} representa as assinaturas de operações enviadas pelo cliente. Se as operações já estão carregadas então é enviada uma resposta contendo o resultado das Q consultas. Conseqüentemente, na eq. 7, temos a expressão para o custo de tráfego em regime estável:

$$T_{COD,estavel} = U \sum_{n=1}^{n} \left(\eta_{COD} I_{COD} + \tilde{\eta}_{COD} \sum_{q=1}^{Q} R_{qn} \right) . \qquad \text{(eq. 7)}$$

Se, por outro lado, o código não está carregado então uma requisição, de tamanho I_{fetch}, é enviada solicitando dinamicamente o carregamento do código. O código, de tamanho C_{COD}, é transferido e conectado (*linked*) ao componente destino onde se torna disponível para futuras invocações. Na inicialização, na eq. 8, existe um *overhead* representado pela mensagem enviada pelo componente destino requisitando o carregamento (I_f) somado a transferência do código (C_{COD}), e portanto:

$$T_{COD,inicializacao} = \eta_{COD} \sum_{n=1}^{n} (I_f + C_{COD}) . \qquad \text{(eq. 8)}$$

Comparações entre Paradigmas. Usando as equações de custo de tráfego anteriores para fazer comparações entre paradigmas chega-se aos resultados a seguir. Comparando CS (eq. 3) e REV (eq. 4), só é vantagem utilizar REV se: $T_{CS} \geq \mu T_{REV}$, assim:

$$N \sum_{q=1}^{Q} \eta_{CS} I_q + \sum_{n=1}^{n} \sum_{q=1}^{Q} \tilde{\eta}_{CS} R_{qn} \geq N \eta_{REV} C_{REV} + \sum_{n=1}^{n} \tilde{\eta}_{REV} \sum_{q=1}^{Q} R_{qn}$$

e substituindo as requisições (I_q) e respostas (R_q) por valores médios $\bar{I}$ e $\bar{R}$, temos:

$$NQ(\eta_{CS} I + \tilde{\eta}_{CS} R) \geq N(\eta_{REV} C_{REV} + \tilde{\eta}_{REV} QR) ,$$

considerando $Q\tilde{\eta}_{CS} R \geq \tilde{\eta}_{REV} QR$, o que equivale a $\Delta O_{CS,REV} \geq 0$, resulta em

$$\eta_{REV} C_{REV} \leq \Delta O_{CS,REV} + Q\eta_{CS} I .$$

Portanto, REV apresenta vantagem sobre CS se o tamanho da mensagem contendo o código a ser avaliado remotamente é menor que o tamanho das requisições enviadas (em CS) mais a diferença de *overhead* introduzido pela transmissão dos resultados (ΔO). Nitidamente REV é

conveniente se o número de instruções, Q, necessárias para desempenhar uma consulta é alto e o código (C_{REV}) efetivamente compacta a representação das interações locais (I_q).

Podemos aplicar as mesmas considerações e acertivas para comparar REV (eq. 4) e MA (eq. 5). Para manter generalidade e permitir simplificações, consideraremos a eq. 5 com o último termo não zero, isto é, existe comunicação do agente com o cliente ao longo de sua trajetória. Portanto, consideraremos $T_{MA} \geq T_{REV}$ e chegamos a forma a seguir:

$$N\eta_{MA}C_{MA} + \tilde{\eta}_{MA}QN\bar{R} + \sum_{n=1}^{N} \eta_{MA}Q(n-1)\bar{R} \geq N(\eta_{REV}C_{REV} + \tilde{\eta}_{REV}Q\bar{R}) \ .$$

Para Q grande pode-se assumir $\tilde{\eta}_{REV}QNR \equiv \tilde{\eta}_{MA}QNR$ e $\eta_{REV} \equiv \eta_{MA}$. Neste caso mais resultados são empacotados juntos ao invés de enviados separadamente, como em CS, e a diferença de *overhead* (ΔO) tende a ser desprezível. De forma geral, temos $C_{REV} \equiv C_{MA}$ e aplicado na inequação anterior resulta a eq. 9, contendo o estado do agente crescendo a cada nó:

$$\sum_{n=1}^{N} \eta_{MA}Q(n-1)\bar{R} \geq 0 \ . \qquad \textbf{(eq. 9)}$$

A conclusão, portanto, é outra diretiva no projeto de agentes, isto é, favorecer a simplificação do estado acumulado dos nós já visitados mantendo o tamanho do estado transportado constante ou quase, a exemplo: o resultado de uma busca de determinado mínimo (ou máximo).

Para fazer a comparação entre REV (eq. 4) e COD (eq. 7 e 8) devemos considerar que a aplicação do paradigma de COD depende da freqüência de invocações U (eq. 2). Consideraremos $T_{REV} \geq T_{COD}$ e assumindo $I \equiv I_{COD} \equiv I_{fetch}$ e $\tilde{\eta}_{REV}QR \geq \tilde{\eta}_{COD}QR$, resulta em

$$\eta_{REV}C_{REV} \geq \frac{U+1}{U}\eta_{COD}\bar{I} + \frac{1}{U}\eta_{COD}C_{COD} \ .$$

Se a funcionalidade de código e a tecnologia são as mesmas para os dois paradigmas, podemos fazer $C_{REV} \equiv C_{COD} \equiv C$ e $\eta_{REV} \equiv \eta_{COD} \equiv \eta$.

Para U >> 1, a equação anterior simplifica para, $\eta_{REV}C_{REV} \geq \eta_{COD}I$ e

$$C \geq I \qquad \textbf{(eq. 10)}$$

Para U pequeno por sua vez temos $\eta C \geq \frac{U+1}{U}\eta\bar{I} + \frac{1}{U}\eta C$ e

$$C \geq \frac{U+1}{U-1}\bar{I} \ , \qquad \textbf{(eq. 11)}$$

o que confirma a noção intuitiva que para uma função usada mais de uma vez em seguida, convém armazenar seu código temporariamente para economizar custo de tráfego de mensagens.

Finalmente, comparamos CS (eq. 3) e COD (eq. 7 e 8). Assumiremos $T_{CS} \geq T_{COD}$ e $I \equiv I_{COD} \equiv I_{fetch}$, assim como as considerações anteriores onde $I_q \equiv I$ e $R_q \equiv R$. Se $\Delta O_{CS,COD}$ é a diferença de *overhead* entre os dois paradigmas no envio de resultados de volta ao cliente, então, resulta a eq. 12 onde

$$UN(Q\eta_{CS}I + Q\tilde{\eta}_{CS}R) \geq UN(\eta_{COD}I + \tilde{\eta}_{COD}R) + N(\eta_{COD}I + \eta_{COD}C_{COD}) \text{ e}$$

$$U \geq \frac{\eta_{COD}I + \eta_{COD}C_{COD}}{Q\eta_{CS}I - \eta_{COD}I + \Delta O_{CS,COD}}, \qquad \text{(eq. 12)}$$

ou ainda, para $C_{COD} \equiv C$ e $\eta_{CS} \equiv \eta_{COD} \equiv \eta \equiv \tilde{\eta}_{CS} \equiv \tilde{\eta}_{COD}$, então

$$(I + C) \leq U(Q - 1)(I + R). \qquad \text{(eq. 13)}$$

Da eq. 13 concluímos que para COD ser superior a CS, o custo de envio de código deve ser U vezes menor que o custo de comunicação remota.

Paradigmas e Tecnologias

Mobilidade em *sistemas de código móvel* (SCM) pode ser caracterizada pelas suas *unidades de execução* (UE) que podem migrar. *Mobilidade forte* é a habilidade de um *SCM* permitir a migração de código e estado de execução de uma *UE* para um *ambiente computacional* (AC) diferente. *Mobilidade fraca* é quando num *SCM* é possível transferir código através de diferentes *ACs* podendo ser acompanhado de alguns dados de inicialização, mas nenhuma migração de estado de execução está envolvida. A Tabela 2.3 resume uma classificação dos mecanismos de mobilidade

Mobilidade forte é suportada por dois tipos de mecanismos: migração e *clonagem remota*. O mecanismo de migração suspende uma *UE*, a transmite ao *AC* destino e retoma a execução. Migração pode ser *proativa* ou *reativa*. Em migração *proativa*, o tempo e destinatário para migração são determinados autonomamente pela *UE* em migração. Em migração *reativa*, o movimento é disparado por uma *UE* diferente que tem algum tipo de relacionamento com a *UE* a ser movida. O mecanismo de clonagem remota cria um cópia de uma *UE* em um *AC* remoto. Difere dos mecanismos de migração porque a *UE* original não é desconectada de sua *AC* corrente, ou seja, é um mecanismo de duplicação. Como para migração, clonagem remota pode ser *proativa* ou *reativa*.

Mecanismos de *mobilidade fraca* oferecem a possibilidade de transferir código através de *ACs* e conectá-lo dinamicamente a uma *UE* ativa ou usá-lo como fragmento de código para uma nova *UE*. Tal mecanismo pode ser classificado de acordo com a direção da transferência de código, a natureza do código movido, a sincronização envolvida, e o instante em que o código é efetivamente executado no destinatário. Pela direção de transferência de código, uma *UE*

pode ou puxar o código a ser conectado (*linked*) dinamicamente e/ou executado, ou simplesmente enviar tal código a outro *AC*. O código pode ser migrado como código auto-contido ou como um fragmento. Código auto-contido é usado para instanciar uma nova *UE* no destinatário enquanto um fragmento de código deve ser conectado (*linked*) no contexto de um código já em execução e eventualmente executado. Os mecanismos que permitem *mobilidade fraca* podem ser *síncronos* ou *assíncronos*, dependendo da *UE* que requisita a transferência ficar suspensa ou não até o código ser executado. Em mecanismos assíncronos a execução do código transferido pode ser *imediata* ou *adiada*. No primeiro caso, o código executa tão logo é recebido, enquanto no segundo esquema a execução é realizada a partir de uma condição satisfeita.

Tabela 2.3 Uma classificação de mecanismos de mobilidade [Vigna98]

Mecanismo de Mobilidade	Gerenciamento de Código e Estado	Mobilidade Forte	Migração	Proativa		
				Reativa		
			Clonagem Remota	Proativa		
				Reativa		
		Mobilidade Fraca	Código Enviado	Auto contido	Síncrono	
					Assíncrono	Imediato
						Adiado
				Fragmento	Síncrono	
					Assíncrono	Imediato
						Adiado
			Código Puxado	Auto contido	Síncrono	
					Assíncrono	Imediato
						Adiado
				Fragmento	Síncrono	
					Assíncrono	Imediato
						Adiado

A Tabela 2.4 resume paradigmas e tecnologias. Tecnologias de código não móvel são apropriadas para a implementação de arquiteturas baseadas no paradigma CS. Se usadas para implementar arquiteturas baseadas em REV, forçam a implementação a usar código como dados e programar a avaliação de tal código explicitamente. Mesmo se tecnologias de código não móvel forem usadas para implementar arquiteturas de MA, o programador também deve codificar o gerenciamento, i.e., variáveis auxiliares devem ser usadas para manter o estado da computação e estruturas de código não natural devem ser usadas para recuperar o estado de um componente após a migração ao destinatário.

Tabela 2.4 Relação entre paradigmas e tecnologias [Vigna98]

Tecnologia	Paradigmas		
	CS	REV	MA
Não Móvel	Apropriado	Código como dados Interpretação de programa	Código e estado como dados Restauração de estado de programa Interpretação de programa
Mobilidade Fraca	Código é uma única instrução Cria unidades de execução desnecessárias	Apropriado	Estado como dados Restauração de estado de programa
Mobilidade Forte	Código é uma única instrução Cria unidades de execução desnecessárias Move estado p/frente e p/trás	Gerencia migração Move estado p/ frente e p/trás	Apropriado

Tecnologias de *mobilidade fraca* que permitem a execução de segmentos de código remotamente são naturalmente orientadas para a implementação de aplicações desenvolvidas de acordo com o paradigma de REV. Estas tecnologias são ineficientes para implementar arquiteturas CS uma vez que forçam a execução remota de segmentos de código composto de uma única instrução. Portanto, uma nova *UE* é criada para executar este código degenerado. Ao contrário disto, para implementar aplicações baseadas no paradigma de MA, o programador precisa gerenciar, ao nível do programa, o (des)empacotamento de variáveis representando o estado e a recuperação de fluxo de execução da *UE*.

Tecnologias de *mobilidade forte* são orientadas para aplicações baseadas no paradigma de MA não sendo apropriadas para a implementação de aplicações baseadas nos paradigmas de CS e REV. No primeiro caso, o programador precisa sobre-codificar um agente para tê-lo movido para o lado do servidor, executar um única instrução e saltar de volta com os resultados. Tais implementações podem ser ineficientes uma vez que todo estado da *UE* é transmitido para frente e para trás pela rede. No segundo caso, em adição ao código a ser executado remotamente, o implementador deve adicionar os procedimentos de migração. Além disto, o estado da *UE* deve ser transmitido através da rede.

Resumindo, tecnologias podem se revelar muito poderosas ou muito limitadas para implementar uma arquitetura em particular. No primeiro caso, são desperdiçados recursos, resultando em ineficiência. No segundo caso, o programador deve codificar todos os mecanismos e políticas que a tecnologia não implementa.

2.3 Agentes

Uma noção interessante é a de componentes [Orfali96,Orfali98] como objetos autônomos que podem ser conectados e executados através das redes, das aplicações, das linguagens, das ferramentas e dos sistemas operacionais. Objetos distribuídos são, por definição, componentes por causa da forma como são encapsulados. Um agente é distinto de simplesmente um objeto por sua parte autônoma de execução que é iniciada na instanciação do agente e evolui ao longo de seu ciclo de vida. Toda nova ativação após a instanciação recupera o estado da execução para prosseguimento.

Uma analogia [Russel95] entre engenharia de conhecimento e programação apresenta o paradigma da programação a um nível mais alto de abstração. A abordagem é que requer menos trabalho decidir somente quais objetos e relações valem a pena representar e quais relações se mantém entre quais objetos. Não há nenhuma necessidade de computar as relações entre objetos. Há a necessidade de especificar somente o que é verdadeiro enquanto um procedimento de inferência descobre como tornar os fatos em uma solução do problema. Se considerarmos que, em um mesmo contexto, um fato é verdadeiro não obstante que tarefa está tentando resolver, então bases de conhecimento podem ser reutilizadas, sem modificação, em uma variedade de diferentes tarefas. A tarefa de eliminar erros é esperada ser mais simples pois qualquer sentença é verdadeira ou falsa por si, enquanto a exatidão de um programa depende fortemente do seu contexto.

A noção acima introduz o campo de programação baseada em agentes [Genesereth94] como uma tentativa de fazer todo tipo de sistemas e recursos interoperáveis fornecendo uma interface declarativa baseada em lógica de primeira ordem. "A noção de agente pretende ser uma ferramenta para analisar sistemas, não uma caracterização absoluta que divida o mundo em agentes e não agentes." [Russel95]. Um agente não necessita ser um programa [Franklin96], mas agentes de *software* são, por definição, programas que devem atingir algumas características para serem um agente.

Na Tabela 2.5 encontramos algumas das características gerais de agentes. Neste trabalho exploramos algumas das características gerais de agentes mas concentramos atenção em especial na propriedade de *mobilidade* e as tecnologias associadas. Outras características de agentes são tratadas com menor atenção e são tratadas no trabalho pela forma como influenciam os aspectos de mobilidade considerados. Estas outras características são por exemplo: comunicação, reação, autonomia, orientação a objetivo, continuidade no tempo, flexibilidade e caráter (persistência de estado). A característica de aprendizado particularmente não está presente, sendo uma etapa conseqüente do trabalho.

Tabela 2.5 Características gerais de Agentes

Propriedade	Outros Nomes	Significado
reatividade	(sentindo e agindo)	responde de uma maneira temporal a mudanças no ambiente
autonomia		exerce controle sobre suas próprias ações
orientação a objetivo	proativo c/objetivo	não atua simplesmente em resposta ao ambiente
continuidade		é um processo executando continuamente
comunicabilidade	sociabilidade	comunica com outros agentes, talvez inclusive pessoas
aprendizado	adaptatividade	muda seu comportamento baseado em experiências prévias
mobilidade		capaz de transportar-se de uma máquina para outra
flexibilidade		ações não prescritas
caráter		personalidade e estado (emocional) confiáveis
representatividade		agir representando uma pessoa ou organização

Mobilidade

Um objeto pode mover-se somente em função de uma demanda externa enquanto um agente pode mover-se em função de ambos: demanda externa e interna (autônoma). Nós distinguimos mobilidade em explícita e em implícita. A primeira é de iniciativa apenas do agente enquanto a segunda é conseqüência de mudanças no ambiente, como por exemplo o movimento de um componente externo. Como conseqüência, um agente se move em função de outro agente. Definimos este segundo tipo de mobilidade como implícita ou transparente e como mostrado mais adiante neste texto ela ocorre ao nível das requisições e respostas.

As plataformas de agentes existentes limitam-se ao que definimos como mobilidade explícita. Uma abordagem para transparência de migração em sistemas de agentes móveis não é explorada na maioria dos trabalhos relacionados [Aglets, GrassHopper98, Odyssey97, Voyager97-1, Voyager97-2]. As técnicas da migração são descritas na maioria em ambientes não orientados a objeto e sob um controle central de migração que faz a otimização global do processamento distribuído [Ciupke96, Golubski96, Goldszmidt96]. Evitamos otimização global mas usamos uma estratégia baseada em instantâneos (*snapshots*) e em balanceamento final aleatório. Um instantâneo é uma primeira aproximação que seleciona um conjunto de recursos que satisfazem a uma faixa de disponibilidade. A partir deste conjunto, tentamos conectar aleatoriamente ao primeiro recurso disponível. Balanceamento final de componentes acontece ao nível de requisições ou de rechamadas (*callbacks*).

Nosso objetivo é dar suporte a mobilidade explícita e implícita em um mundo de multi-agentes e usar o objetivo da tarefa para identificar os componentes complementares e guiar sua migração de maneira transparente. Propomos a implementação de um suporte a transparência de migração de agentes em uma plataforma de multi-agentes baseada em um *middleware* OMG/ CORBA. A localização de um componente é transparente desde que esteja registrado em algum sítio no ambiente.

Autonomia e flexibilidade

Autonomia de um agente sugere ele poder (re)escalonar autonomamente as tarefas de um roteiro enquanto flexibilidade sugere ele poder redefinir ou adaptar o roteiro ao cenário. Estes aspectos se referem à *configuração* de ambientes de objetos distribuídos e podem ser vistos como uma *configuração dinâmica*.

Autonomia e flexibilidade de um agente ao invés de irrestritas estariam melhor definidas como suficientes, isto é, o agente tem um grau de liberdade para ajustar suas tarefas e desempenho mas esta autonomia está limitada por ser demasiado vago dar autonomia plena e poder envolver áreas críticas. É difícil definir autonomia irrestrita senão dentro de um contexto. Isso sugere que fora deste contexto a autonomia irrestrita não é mais verdadeira.

Em gerenciamento de sistemas, um agente pode atingir o seu limite de autonomia e entrar em áreas críticas, e neste caso deve solicitar autorização para prosseguir. Isto pode acontecer diversas vezes para uma mesma situação. Depois que algum histórico que relata os repetidos pedidos de autorização seguidos de atuações bem sucedidas um agente poderia obter a extensão de sua autonomia. Se concedida, isto permite ao agente atuar autonomamente em situações semelhantes sem nova solicitação prévia. Para ser mais geral, um agente obtém um grau de autonomia e ao necessitar maior autonomia deve solicitá-la e obter autorização a partir de históricos bem sucedidos.

Mobilidade e segurança

Considerar a mobilidade de agentes implica em alguns aspectos de segurança quanto ao agente ser confiável pelo hospedeiro assim como o agente confiar no novo hospedeiro. No primeiro caso há métodos propostos de autenticação baseados em credenciais do agente a ser confiado pelo hospedeiro. No segundo caso uma aproximação similar de autenticação pode ser usada. Um agente móvel pode levar com ele ou obter de algum sítio a lista contendo o domínio de hospedeiros contratados como confiáveis. Uma trajetória de um agente pode ser predefinida contendo hospedeiros confiáveis com a prévia autenticação do agente em cada um deles. Segurança não é o tema deste trabalho, sendo consideradas soluções disponíveis.

2.4 Agentes móveis sobre CORBA

Nós propomos a implementação do paradigma de Agentes Móveis (MA) sobre a tecnologia de objetos do modelo OMG/CORBA. Para implementar MA sobre CORBA consideramos outros paradigmas disponíveis na tecnologia, conforme considerado na Tabela 2.6. Nesta tabela mostramos como cada paradigma está presente em cada fase do ciclo de vida de um agente móvel. Também está presente uma correspondência com Java e seus aspectos não distribuídos. Esta correspondência é feita porque Java é uma linguagem orientada a objeto em desenvolvimento onde conceitos modernos de tecnologia de objetos estão sendo incorporados. No particular aspecto de distribuição estão sendo transportados e incorporados os aspectos presentes em CORBA, como por exemplo *JavaIDL*, *RMI* e o esforço de unificação *RMI/IDL* [OMG98-1,OMG98-3, OMG98-4].

Tabela 2.6 Paradigmas de Código Móvel sobre CORBA.

Paradigmas	CORBA (distribuído)	Java (local)
Cliente-Servidor (CS)	- O cliente requisita componente computacional ativo instanciado no servidor.	- Cliente requisita instância de componente no Servidor.
Código sob Demanda (COD)	- O cliente requisita componente computacional a ser ativado no servidor por um gerente de ativação, a partir de um Repositório de Implementação. - A operação continua em modo CS até a desativação.	- Semelhante ao carregamento de uma classe do *classpath*
Avaliação Remota (REV)	- O cliente registra um componente computacional no servidor, i.e., um procedimento de configuração. - A operação segue no modo COD e posteriormente CS, até a desativação.	- Semelhante a uma nova instância de uma classe.
Agentes Móveis (MA)	- O suporte de MA faz diversos procedimentos REV, i.e., ocorre uma reconfiguração baseada em diversas operações de registro, CODs e CS.	-

Utilizamos as estimativas de custo de tráfego apresentadas anteriormente nas eq. 3 a 8 e as considerações da Tabela 2.6 para obter uma estimativa de custo de tráfego para a nossa implementação de MA sobre CORBA. Se a funcionalidade do código e a tecnologia são as mesmas para os paradigmas podemos fazer as simplificações: $\forall p \in \{CS, REV, COD, MA\}$ temos $C_p \equiv C$, $\eta_p \equiv \eta$, $\tilde{\eta}_p \equiv \tilde{\eta}$. Substituindo nas eq.3 a 8 e rearrumando para a situação particular obtemos a equação a seguir (eq.14):

$$T_{MA,\,CORBA}(U) = T_{MA,\,CORBA,\,migracao}(U) + T_{MA,\,CORBA,\,comunicacao-remota}(U) \quad \text{(eq. 14)}$$

Na eq. 14 particionamos a nossa estimativa de custo de tráfego para agentes móveis sobre CORBA no termo *migração*, relativo à migração, e no termo *comunicação-remota*, relativo aos mecanismos de comunicação remota em CORBA.

Para a *migração*, temos de considerar o custo de registro remoto do agente e o custo de requisitar o código para cada unidade, em adição temos de considerar o custo de migrar o código para cada unidade e finalmente temo o custo de enviar o estado da unidade anterior, e portanto temos:

$$T_{MA,CORBA,migracao}(U) = N\eta(I_R + I_f) + N\eta C + U\sum_{n=1}^{N}\eta(S_{MA,n}) \qquad \text{(eq. 15)}$$

Para o termo *comunicação-remota*, após a *migração*, temos de considerar o custo das requisições e respostas remotas durante cada execução em cada unidade, e resulta:

$$T_{MA,CORBA,comunicacao-remota}(U) = U\sum_{n=1}^{N}\sum_{q=1}^{Q}(\eta I_q + \tilde{\eta}R_{qn}) \; . \qquad \text{(eq. 16)}$$

A fase de *migração* envolve o registro remoto (I_R), a passagem do código (C) quando não disponível localmente e a transferência do termo relativo ao estado do agente. O estado do agentes está representado pelo último termo:

$$U\sum_{n=1}^{N}\eta(S_{MA,n}) \; . \qquad \text{(eq. 17)}$$

Este termo é o somatório de estados ao longo das N unidades vezes o número de execuções (U) em cada unidade. Em princípio o estado pode crescer linearmente, entretanto o agente pode realizar um processamento e compressão dos dados para manter o tamanho do estado quase constante.

A fase *comunicação-remota* está relacionada às requisições (I_q) e respostas (R_{qn}) remotas entre cliente e servidor e cresce com o número de requisições por execução por unidade. Basicamente, o custo de tráfego na eq. 16 é linearmente dependente do número de requisições (Q), enquanto na eq. 15, este termo não está presente. Em uma aplicação, se o número de requisições remotas sem mobilidade aumenta acima do custo de migração, na eq. 15, então uma solução baseada em mobilidade se torna vantajosa em relação a uma sem mobilidade, do ponto de vista de custo de tráfego gerado.

A fase de *migração*, na eq. 15, é semelhante à fase de *inicialização* na formulação de COD, na eq. 7, com a inclusão da parte referente ao estado do agente, $S_{MA,n}$, da eq. 5. O mesmo resultado pode ser obtido a partir da formulação para REV, na eq. 4, com $R_q \equiv I_f$ e a inclusão do estado do agente. A fase de *comunicação-remota* de nossa implementação de MA sobre CORBA, na eq. 16, é semelhante à formulação de custo de tráfego para CS na eq. 3.

2.5 Trabalhos Relacionados em Tecnologias de Agentes Móveis

Distinguimos mobilidade em explícita e em implícita. A primeira é demandada pelo agente e a segunda é conseqüência de mudanças no meio externo. A migração implícita ocorrerá ao nível de requisições e rechamadas (*callbacks*). Externamente a um agente não é possível distinguir a origem da migração.

Técnicas de migração são descritas em geral para ambientes não orientados a objeto e sob um controle centralizado de migração e otimização global do processamento distribuído. Não é tema central do trabalho explorar algorítmos ótimos de migração baseados em otimização global. Apenas utilizamos uma estratégia conciliando duas etapas: *primeiro* uma escolha inicial baseada em busca para uma primeira aproximação e *segundo* uma seleção final aleatória para evitar efeito *ping-pong*. O balanceamento de carga [Ciupke96, Golubski96, Goldszmidt96] utilizado é principalmente no sentido de tolerância a falhas [Cristian94, Cristian93, Cristian91], isto é, de sustentação da operacionalidade dos componentes de serviço e dentro de uma qualidade de operacionalidade, ao invés do sentido usual de maximização de desempenho.

A migração depende da seleção e da localização de componentes. Nossa estratégia de seleção é baseada em instantâneos e num balanceamento final aleatório. Os instantâneos são uma primeira aproximação com um conjunto de componentes que satisfazem critérios de disponibilidade. Deste conjunto, tentamos conectar aleatoriamente ao primeiro hospedeiro que atender a uma requisição. Localização é possível se o componente estiver registrado em algum lugar no ambiente. Para ser transparente, a migração é orientada a objetivo. O balanceamento final acontece ao nível de requisições e de rechamadas. Técnicas bem conhecidas podem permitir novos resultados se aplicadas a novos paradigmas como por exemplo o modelo de agentes.

A seguir faremos uma breve descrição de algumas implementações. As plataformas de agentes existentes concentram-se no que definimos como mobilidade explícita. Uma abordagem para transparência de migração em sistemas de agentes móveis não é explorada na maioria de trabalhos relacionados [Aglets, GrassHopper98, Odyssey97, Voyager97-1, Voyager97-2].

Java. A linguagem Java desenvolvida pela Sun Microsystems [Java96, Java97, Java98-1, Java98-2, Java98-3] despertou a maior parte da atenção e expectativas recentes em relação a código móvel. O objetivo inicial de Java foi prover uma linguagem orientada a objeto de uso geral, portável, limpa e fácil, tendo sido este objetivo redirecionado com o crescimento da Internet. Um compilador traduz os programas em uma forma intermediária, independente de plataforma denominada *byte-code*. Este *byte-code* é interpretado por um ambiente computacional (AC) denominado *máquina virtual Java* (*JVM*). Um *class loader* recupera e conecta dinamicamente classes em uma JVM em execução. Quando o código em execução possui um nome de classe não resolvido o *loader* recupera a classe de um hospedeiro remoto e a carrega na JVM. Portanto Java suporta mobilidade fraca utilizando mecanismos de recuperação de fragmentos de código. Tais mecanismos são assíncronos e o código não traz estado de execução nem relacionamentos remotos. Entretanto, são disponibilizados mecanismos para serialização de estado e relacionamentos. A maioria das implementações atuais, mesmo as que iniciaram com versões não Java, dispõem de versão sobre Java que implementa mobilidade fraca.

Grasshopper. As implementações Grasshopper e MAGNA [GrassHopper98, Krause97] são ambientes de agentes móveis segundo o padrão OMG/MASIF. Permitem a criação de aplicações distribuídas com agentes se beneficiando de comunicação e acesso a dados localmente a velocidade mais alta. Grasshopper implementa uma facilidade de agentes móveis que poderia ser usada em várias aplicações, inclusive como a estrutura para o desenvolvimento de um sistema de gerenciamento baseado no paradigma de agentes móveis. Outras implementações disponíveis de facilidades de agentes são basicamente sobre Java/RMI, como por exemplo, Odyssey [Odyssey97], Aglets [Aglets], Voyager [Voyager97-1, Voyager97-2] entre outras.

Aglets. A implementação de Aglets, desenvolvida pelo *IBM Tokyo Research Laboratory*, estende Java com um suporte para mobilidade fraca. As UEs são *threads* do interpretador Java que constitui o ambiente computacional (AC). A API provê a noção de contexto como uma abstração do AC. O contexto de um aglet provê um conjunto básico de serviços. Possui duas primitivas básicas de migração: *dispatch*, para envio de código; e a simétrica *retract*, para recuperação (*fetch*) de código.

Odyssey. A implementação de Odyssey, de forma semelhante, estende Java com um suporte para mobilidade fraca. As UEs são *threads* do interpretador Java que constitui o ambiente computacional (AC). Se baseia em Java/RMI para registro remoto do agente e recuperação de código. Foi implementado pela *General Magic*, anteriormente criadora do *Telescript*, uma linguagem orientada a objeto desenvolvida para aplicações distribuídas em larga escala, com um enfoque em mobilidade forte. Utiliza uma linguagem intermediária portável chamada *Low Telescript*. As ACs são responsáveis por executar *agentes* e *lugares* que são EUs *Telescript*. Agentes podem mover utilizando a operação *go* que implementa migração proativa. Também está disponível uma operação *send* que implementa replicação proativa remota. *Lugares* são UEs estacionárias que podem conter outras EUs.

Voyager. A implementação de Voyager, de forma semelhante, estende Java com um suporte para mobilidade fraca. As UEs são igualmente *threads* do interpretador Java que constitui o AC. Se baseia numa versão anterior não Java implementando mobilidade forte. Possui uma característica própria no tratamento de rechamadas (*callbacks*), pela qual mantém um servidor remoto atualizado da nova localização do cliente. Isto diminui a necessidade de re-localização do cliente em caso de migração.

Este Trabalho. A nossa abordagem se baseia na utilização de Java e CORBA, com um suporte para mobilidade fraca. As UEs também são *threads* do interpretador Java que constitui o AC. As UEs podem ser *agentes* ou *lugares*. *Lugares* estão definidos na especificação MASIF como ambientes de execução, isto é, contextos dentro da agência onde um agente pode executar. Em nosso caso são processos CORBA ou aglomerados (*clusters*) de agentes.

MASIF. A especificação OMG/MASIF [OMG97-2] começou em novembro de 1995 como uma proposta de *facilidade de agentes móveis* (MAF) e foi aprovada em março de 1998 como *Facilidade de Interoperabilidade entre Sistemas de Agentes Móveis* (*MASIF*) de diferentes fabricantes. O padrão consiste de várias especificações CORBA IDL para a comunicação remota entre plataformas de componentes distribuídos. Não cobre todas as funcionalidades requeridas para o desenvolvimento de um sistema de agentes, mas somente os aspectos

comuns a maioria das plataformas de agentes atuais. Para conformidade com o MASIF espera-se modificações ou restrições substanciais ao *kernel* das plataformas existentes. Existe uma abordagem tratando da mobilidade de objetos em fase de discussão e especificação no contexto do OMG (*Object by Value*) [OMG98-4]. Esta proposta de *passagem de objeto por valor* trata da migração de objetos incluindo o seu estado de execução.

A seguir resumimos os principais conceitos de agentes móveis propostos no OMG/MASIF. A descrição usa terminologia de orientação a objeto. Para sistemas de agentes móveis que não usam orientação a objeto, substituir a palavra *classe* por *código* nas definições.

Agente Um agente é um programa de computador que age autonomamente representando pessoa ou organização. Atualmente, a maioria dos agentes são programados em uma linguagem interpretada (por exemplo, Tcl e Java) para portabilidade. Cada agente tem a sua própria *thread* de execução, assim as tarefas podem ser executadas por sua própria iniciativa.

Agente Estacionário Um agente estacionário executa somente no sistema onde inicia a sua execução. Se o agente necessita de informação que não está no sistema, ou necessita interagir com outro agente ou sistema de agentes diferente, utiliza um mecanismo de transporte de comunicação tal como chamada remota de procedimento (*RPC*). A comunicação necessária a agentes estacionários se encontra atualmente em sistemas de objetos distribuídos tal como CORBA, DCOM, e RMI.

Agente Móvel Um agente móvel não está limitado ao sistema onde inicia a sua execução. Possui a habilidade única de se transportar de um sistema a outro, em uma rede. A habilidade de viajar permite a um agente móvel se mover para um sistema de agentes destino que contém o objeto com o qual o agente deseja interagir. Além disso, o agente pode utilizar os serviços de objeto do sistema de agentes destino. Um agente móvel tem mais potencialidades e exigências que muitos sistemas atuais de objetos distribuídos podem endereçar.

Estado de Agentes Quando um agente viaja, seu estado e código são transportados com ele. Neste contexto, o estado do agente pode ser o seu estado de execução, ou os valores de atributos que determinam o que deve ser feito quando o agente recomeça a execução no sistema de agentes destino. Os valores de atributos do agente incluem o estado do sistema de agentes associado ao agente (por exemplo tempo de vida).

Estado de Execução de Agentes O estado de execução de um agente é o seu estado de *run-time*, incluindo o contador de programa e a estrutura de pilhas.

Autoridade de Agentes A autoridade de um agente identifica a pessoa ou a organização em nome da qual o agente age. Uma autoridade deve ser autenticada.

Nomes de Agentes Agentes requerem nomes que podem ser identificados em operações de gerenciamento, e podem ser localizados por um Serviço de Nomes. Os agentes são nomeados utilizando a sua autoridade, identidade e tipo de sistema de agentes. A identidade de um agente é um valor único dentro do espaço de autoridade que identifica uma instância particular do

agente. A combinação de autoridade, identidade e tipo de sistema de agentes é sempre um valor global único. Como o nome de um agente é globalmente único e imutável, o nome pode ser usado como chave em operações que se referem a uma instância particular de agente.

Localização de Agentes A localização de um agente é o endereço de um *lugar* (*place*), definido mais adiante. Um lugar reside dentro de um sistema de agentes. Conseqüentemente, uma localização de agente deve conter o nome do sistema de agentes onde o agente reside e de um nome de lugar. Note que se a localização não contém um nome de lugar, o sistema de agentes destino escolhe um lugar *default*.

Sistema de Agentes Um sistema de agentes é uma plataforma que pode criar, interpretar, executar, transferir e terminar agentes. Como um agente, um sistema de agentes está associado a uma autoridade que identifica a pessoa ou organização para a qual o sistema de agentes age. Por exemplo, um sistema de agentes com autoridade *joão* executa políticas de segurança do *joão* que protejam recursos do *joão*. Um sistema de agentes é identificado excepcionalmente por seu nome e endereço. Um hospedeiro pode conter um ou mais sistemas de agentes.

Tipo de Sistema de Agentes O tipo de sistema de agentes descreve o perfil de um agente. Por exemplo, se o sistema de agentes é do tipo Aglet, o sistema de agentes é implementado pela IBM, suporta java como linguagem de agentes, usa itinerário para se mover, e usa serialização de objetos java para a sua serialização. A especificação reconhece tipos de sistemas de agentes que suportam múltiplas linguagens, e linguagens que suportam múltiplas serializações de método. Conseqüentemente, um cliente requisitando uma função de um sistema de agentes deve especificar o perfil de agente (tipo de sistema de agentes, linguagem, e serialização de método) de forma a identificar de forma única a funcionalidade.

Lugar Quando um agente se transfere, ele viaja entre ambientes de execução chamados *lugares*. Um lugar é um contexto dentro do sistema de agentes onde um agente pode executar. Este contexto pode fornecer funções tais como controle de acesso. O lugar fonte e lugar destino podem residir no mesmo sistema de agentes, ou em sistemas de agentes diferentes que suportam o mesmo perfil de agente. Um lugar está associado a uma localização, que consiste do nome do lugar e endereço do sistema de agentes dentro do qual o lugar reside. Um sistema de agentes pode conter um ou mais lugares e um lugar pode conter um ou mais agentes. Mesmo que o lugar seja definido como o ambiente onde um agente executa, se o sistema de agentes não implementa *lugar*, então *lugar* é definido como o lugar *default*. Quando um cliente pede a localização de um agente, recebe o endereço do lugar onde o agente está executando.

Regiões Uma *região* é um conjunto de sistemas de agentes que tem a mesma autoridade, mas não são necessariamente do mesmo tipo. O conceito de região permite que mais de um sistema de agentes represente a mesma pessoa ou organização. As regiões permitem a escalabilidade através da distribuição de carga entre múltiplos sistemas de agentes. Uma região fornece um nível de abstração a clientes se comunicando de outras regiões. Um cliente desejando contatar um agente ou sistema de agentes pode não estar ciente da localização do agente. Ao invés, o cliente tem um endereço para a região (basicamente, o endereço de um sistema de agentes que está designado como ponto de acesso a região), e o nome do agente ou lugar. Desta forma é possível contatar e comunicar com um agente ou sistema de agentes com somente esta infor-

mação. Um agente pode também ter a mesma autoridade que a região em que está atualmente residindo e executando. Isto significa que o agente representa a mesma pessoa ou organização que a região. Normalmente, a configuração da região pode conceder um conjunto de privilégios maior a tal agente do que a outro agente residente com uma autoridade diferente. Por exemplo, um agente que tem a mesma autoridade que a região pode conceder privilégios administrativos. Uma região interconecta completamente sistemas de agentes dentro do seu perímetro e permite a transferência de informação ponto a ponto entre eles. Cada região contém um ou mais pontos de acesso à região e desta forma regiões estão interconectadas na forma de uma rede.

Interconexão Sistema de Agentes - Sistema de Agentes Toda comunicação entre sistemas de agentes é através da infra-estrutura de comunicação (CI). O administrador da região define serviços de comunicação para comunicação intra-região e inter-região.

Interconexão Região - Região Regiões são interconectadas através de uma ou mais redes e podem compartilhar o mesmo Servidor de Nomes baseado num acordo entre as autoridades de região e a implementação específica das regiões. Um sistema de não-agentes pode também comunicar com o sistema de agentes dentro de qualquer região contanto que o sistema de não-agentes tenha autorização para tanto. Uma região contém um ou mais sistemas de agentes. Sistemas de agentes e clientes fora da região acessam a região através do sistema de agentes que está visível à parte exterior, similar a um *firewall*. Estes sistemas de agente são definidos como pontos de acesso da região. Direitos de acesso são alocados aos agentes, baseados na mesma autoridade da região na qual está correntemente executando. Utilizando esta definição, uma região é considerada um domínio de segurança no contexto do MASIF.

Serialização/Deserialização A serialização é o processo de armazenar o agente de forma serializada. Deserialização é o processo de restaurar o agente da sua forma serializada. A chave para armazenar e recuperar agentes está na representação do estado de um agente de forma serializada que seja suficiente para a reconstrução do agente. Note-se que a forma serializada deve poder identificar e verificar as classes das quais os campos foram armazenados. Para os sistemas de agentes que não são orientados a objeto, o estado do agente é a extração dos dados de *runtime* do agente, e as classes são o código que o agente executa.

Base de Código *(Codebase)* A base de código especifica a localização das classes usadas por um agente. Pode ser um sistema de agentes ou objeto não CORBA tal como servidores de Web. Se um sistema de agentes for responsável por fornecer as classes necessárias, a *base de código* deve ter informação suficiente para encontrar o sistema de agentes. Tal sistema de agentes é chamado de fornecedor de classes (*class provider*). É possível para um agente mover-se para tal sistema de agentes onde a *base de código* não está diretamente acessível (por exemplo, devido à um *firewall*). Para maiores detalhes, consulte "*Class Transfer*" na seção 1.3.1 do MASIF, "*Remote Agent Creation*".

Infra-estrutura de Comunicação A infra-estrutura de comunicações provê serviços de transporte de comunicação (por exemplo RPC), nomes, e serviços de segurança para um sistema de agentes.

Localidade No contexto de agentes móveis, localidade é definida como sendo próxima ao sistema de agentes de destino sendo no mesmo hospedeiro ou na mesma rede.

2.6 Aplicações

Como aplicações alvo para agentes móveis destacamos aquelas que podem se beneficiar de uma adaptatividade do ambiente computacional, seja às tarefas a realizar seja às limitações nos recursos (temporárias ou constantes).

Aplicações Cliente/Servidor frequentemente exigem ambientes com bastante capacidade computacional. Muitas aplicações distribuídas são adaptadas para executarem em estações de trabalho de alto desempenho e comunicarem via conexões de banda larga. Estas aplicações frequentemente não podem ser executadas em ambientes de recursos limitados, tais como unidades móveis. Unidades móveis conectam intermitentemente às redes sem-fio, em conexões de largura de banda estreita, particularmente fora de escritórios. Dispositivos móveis tem capacidade computacional limitada em parte devido à sua operação sobre baterias.

Aplicações baseadas em componentes e agentes móveis podem ter facilitadas a sua distribuição, isto é, somente um núcleo pequeno necessita ser enviado com funções adicionais carregadas conforme a necessidade. Aplicações distribuídas podem armazenar dados provisórios em servidores intermediários de *proxy* e usar agentes móveis para paginá-los.

Agentes móveis necessitam de uma infra-estrutura de mobilidade para despachar o agente para um hospedeiro remoto, invocá-lo e controlar sua execução. Aplicações onde agentes móveis são interessantes compartilham algumas das seguintes características:

- são de longa duração,
- executam sobre ambientes heterogêneos distribuídos,
- devem adaptar-se às mudanças no ambiente,
- têm restrições de tempo-real e
- devem executar sobre hospedeiros com recursos computacionais limitados ou sobre redes de largura de banda estreita.

Embora alternativas baseadas em outras tecnologias existentes possam ser propostas, em determinados casos agentes móveis apresentam vantagens sobre abordagens convencionais nas etapas de: projeto, implementação e execução. A seguir, está uma lista de algumas vantagens da tecnologia de agentes móveis para certos tipos de aplicação [AGREV97]:

- eficiência,
- redução do tráfego da rede,
- interação autônoma assíncrona,
- interação com entidades de tempo-real,
- adaptação dinâmica,

- tratar de volumes de dados grandes,
- robustez e tolerância a falha,
- sustentação para ambientes heterogêneos,
- comportamento personalizado do servidor,
- paradigma conveniente de desenvolvimento.

Aplicações de Interesse

Alocação Dinâmica de Funcionalidade . Agentes móveis permitem a distribuição dinâmica do *software* para dispositivos, permitindo uma atualização dinâmica de funcionalidade conforme as exigências evoluem, adaptando-se às mudanças da rede e dos recursos computacionais, assim como beneficiar-se de novos recursos disponíveis. Dispositivos de rede, por exemplo, estão sendo continuamente equipados com mais potência computacional, incluindo cpus mais rápidas e mais memória. É eficiente em termos de custo mover dinamicamente a funcionalidade para dispositivos menos caros.

Autonomia . Suponha parte de uma rede sendo interrompida por algumas horas. Nesta situação, agentes podem mover-se para executar o gerenciamento de dispositivos críticos quando não houver nenhuma conexão com a unidade central de gerenciamento.

Atualização de Software . Aplicações distribuídas de longa duração não podem antecipar todas as atualizações e adaptações funcionais requeridas após instaladas em cada local. Atualizar *software* requer tipicamente a substituição de processos por novas versões recentemente compiladas. Em muitos casos, os custos de *shutdown* do sistema são substanciais. Componentes e agentes móveis permitem substituições parciais sem a necessidade de uma parada programada global.

Configuração de Software . Muitos dispositivos podem ser dinamicamente adaptados usando código móvel. Por exemplo, a programação de um dispositivo interativo, onde a aplicação pode dinamicamente substituir, adicionar, ou remover algoritmos específicos. Tal agente pode ser desenvolvido baseado na experiência de usuários e situações após a aplicação original ter sido disponibilizada.

Monitoração e Depuração de *Software*. Outro uso de agentes móveis é estender uma aplicação distribuída com capacidades de monitoração para depuração de erros. Podendo monitorar e acompanhar determinados eventos em hospedeiros distribuídos para, por exemplo, descobrir padrões de acesso a dados, coletar e correlacionar eventos, e assim por diante.

Interoperabilidade de Aplicações . Muitas aplicações distribuídas devem dar suporte a um crescente número de protocolos de interoperabilidade entre diferentes aplicações. Um exemplo é o suporte a diferentes protocolos de representação de dados (NDR, XDR, BER) em um aplicação de *gateway* para suporte de interoperabilidade entre aplicações em sistemas distribuídos heterogêneos. O suporte a todos protocolos resulta em adições grandes de código, sejam usados ou não. Outro exemplo é um laptop cliente dinamicamente recuperando o código de criptografia somente quando necessário, poupando e se adaptando dinamicamente aos recursos.

Ambientes Distribuídos de Experiências de *Collaboratory* (DCEE). A palavra *collaboratory* foi criada para definir ferramentas integradas de computação e comunicação para suporte de colaborações científicas. Um passo importante na realização deste potencial é combinar o interesse da comunidade científica em grande escala com aqueles das comunidades de computação e de engenharia [Kouzes96, Johnston97].

Computação Móvel. Um hospedeiro móvel pode necessitar desconectar-se da rede estacionária, por exemplo, por mudança de estação de comunicação, ou economia de energia. Isto é semelhante a uma queda ou falha de hospedeiro em um ambiente onde a disponibilidade e funcionalidade dos serviços deve ser mantida. Serviços e objetos ativos que executam no hospedeiro móvel podem ter que migrar para algum outro hospedeiro no ambiente [Schulze97-2].

Comércio Eletrônico. Aplicações em Comércio Eletrônico podem ser de formas bastante variadas, considerando que esta é uma área recente em desenvolvimento. A utilização de agentes móveis está presente basicamente na forma de um agente coletando informações de produtos em diferentes pontos de oferta e realizando um fechamento de posição de compra. Serviços afins são um serviço de transação e um serviço de catálogo [Rodríguez99], entre outros [CommerceNet97, Bichler98].

Qualidade de Serviço em Sistemas de Multimídia. Aplicações em Sistemas de Multimídia envolve a negociação de Qualidade de Serviço (QoS) na escolha do serviços associados às tarefas realizadas. A utilização de agentes móveis é proposta com uma alternativa com vantangens em relação aos métodos convencionais [Guedes97, 97-1,97-2, Oliveira98].

CAPÍTULO 3

Um Serviço de Disponibilidade Para Migração de Agentes

Quem me navega não sou eu
quem me navega é o mar.
Paulinho da Viola

Este capítulo apresenta um Serviço de Disponibilidade em ambientes de Agentes Móveis baseados em CORBA. Está dividido nas seguintes seções:

- 3.1 Arquitetura Orientada a Serviços, página 31,
- 3.2 Migração Transparente em Sistemas de Agentes, página 39,
- 3.3 Agência, página 42,
- 3.4 Serviço de Disponibilidade, página 44,
- 3.5 Suporte de Mobilidade, página 49 e
- 3.6 Serviços de Armazenamento, página 51.

3.1 Arquitetura Orientada a Serviços

Nossa proposta de arquitetura orientada a serviços é baseada em componentes distribuídos ativos usando o modelo OMG/CORBA como intermediador de componentes [Orfali96, Orfali98]. Componentes podem estar numa fase estacionária ou numa fase móvel [Schulze97-1, OMG97-2]. O conceito de serviços é associado a serviços disponibilizados por componentes ativos. A denominação componente abrange objetos e agentes.

Uma aplicação baseada nesta arquitetura utiliza, tanto quanto possível, serviços que estão disponíveis. Quando os serviços não estão disponíveis em lugar algum, então a aplicação: ou espera, ou aborta, ou confecciona um serviço substituto. Esta etapa de confecção envolve a migração de componentes a fim otimizar o desempenho da execução como um todo.

Estendemos a localização transparente de componentes do modelo CORBA de forma a permitir a transparência na mobilidade de componentes. Os recursos de um hospedeiro são disponibilizados como serviços através de uma agência e sua disponibilidade publicada por um *serviço de Disponibilidade*. Deste modo uma aplicação contrata recursos específicos de uma agência para um componente particular poder ser carregado.

Utilizando métricas de desempenho em conjunto com um *serviço de Disponibilidade* e um *serviço de Mobilidade* pode-se distribuir transparentemente código na inicialização da aplicação. A noção de serviços como componentes ativos apresentada está associada ao nosso conceito de agentes.

O diagrama da Figura 3.1 ilustra os blocos básicos de uma aplicação orientada a serviço, ou orientada a agente, como discutido a seguir.

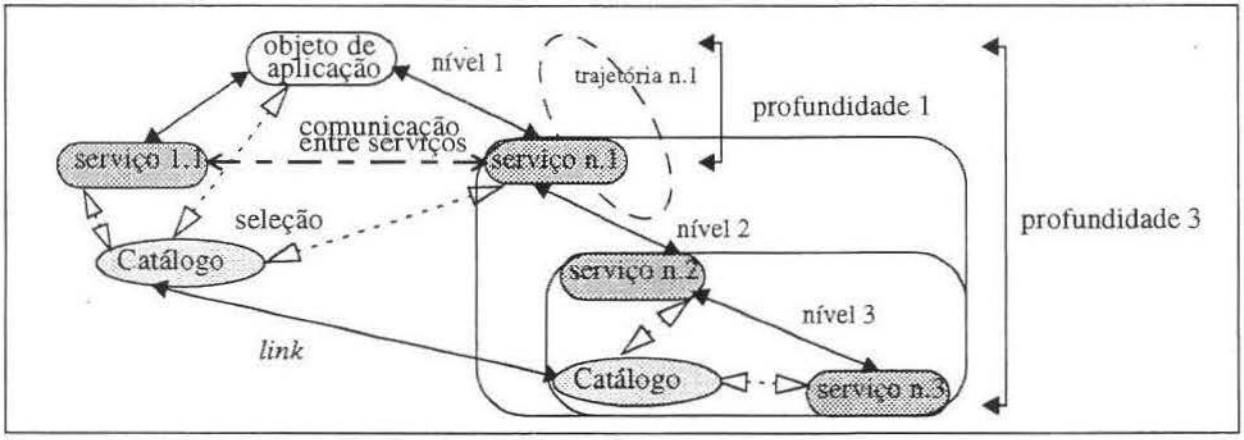


Figura 3.1 Uma Aplicação Serviço-Orientada.

Agentes são os componentes básicos que oferecem um serviço a ser utilizado por uma aplicação. Serviços *disponíveis* são oferecidos em uma agência. Serviços *não-disponíveis* são confeccionados, por exemplo pela aplicação, e a *posteriori* tratados como serviços disponíveis. Uma agência é um componente básico capaz de oferecer agentes e seus respectivos serviços a uma aplicação. A negociação de serviços pode ser tratada por um serviço de Catálogo como especificado pelo OMG/Trader [Trader95, OMG96, Lima95] ou outro tipo genérico de Catálogo [CommerceNet97, Bichler98, Rodríguez99]. O Trader é um serviço externo para encontrar outros serviços dentre um conjunto de agências contratadas. Cada agente móvel se locomove dentro de uma trajetória própria e pode se comunicar com outros agentes através de chamadas de métodos públicos. A comunicação pode ser remota ou local, e neste caso envolver o deslocamento do cliente ou do servidor para que fiquem co-localizados. Um serviço oferecido por um agente pode utilizar serviços oferecidos por outros agentes, remotos ou não. Conseqüentemente, no caso de chamadas remotas pode estar associada uma profundidade diferente de 1 (um). O termo *profundidade* está associado a seqüência de conexões remotas numa mesma chamada de método.

Na Figura 3.2, serviços estão representados como agentes distribuídos em um domínio. Podem estar ou estacionários conectados a infra-estrutura local do ORB ou em movimento em direção a um novo hospedeiro.

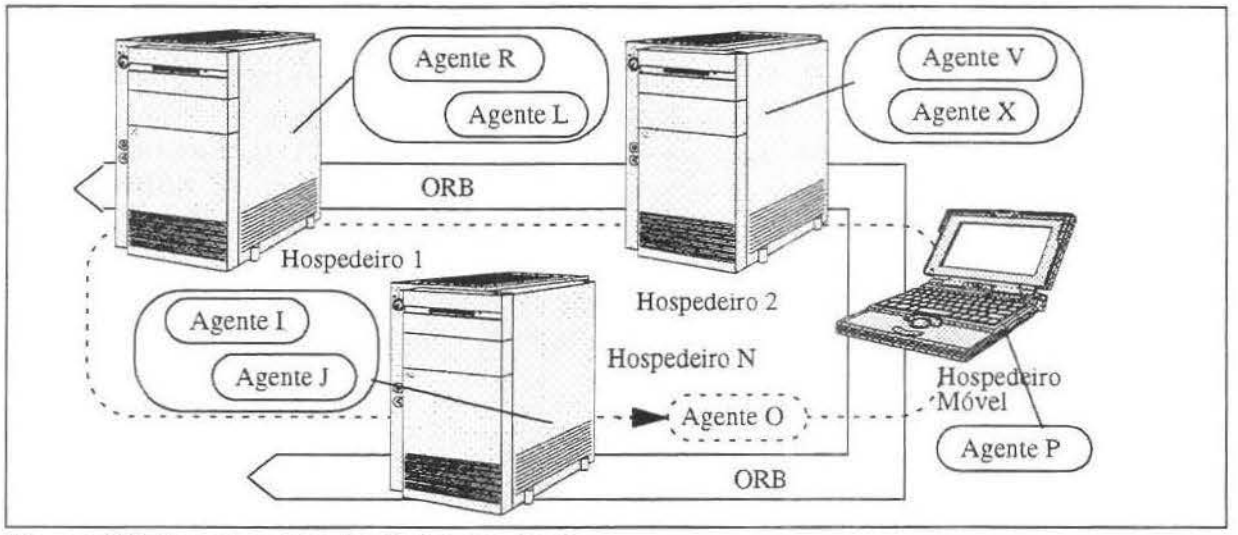


Figura 3.2 Agentes estacionários e móveis.

3.1.1. Serviços

A computação com serviços representa um nível mais alto de abstração na implementação de aplicações reduzindo o esforço ao desenvolvimento de componentes específicos não disponíveis e à interconexão dos componentes ativos relacionados com a aplicação. A interconexão destes componentes envolve: contratar, localizar, requisitar e responder. Os nossos componentes ativos [Sichman95, Orfali96, Orfali98] são agentes de serviços em um ambiente de multi-agentes.

Serviços disponíveis podem ser componentes remotos, podendo ser componentes de *software*, agentes de gerenciamento de sistemas, co-processadores, bases de dados, compressão de dados, arquivos, e assim por diante. Serviços não-disponíveis podem ser tipos semelhantes mas por alguma razão não disponíveis no contexto. A não disponibilidade pode ter significado variado como:

- a aplicação não tem autorização de acesso a um serviço;
- um serviço específico não está disponível onde necessário;
- um serviço está desconectado temporariamente;
- o serviço é uma computação específica da aplicação e tem que ser confeccionado.

Uma aplicação pode tratar esta indisponibilidade e confeccionar um serviço substituto. A confecção de um serviço lida com transporte de código, alocação de recursos para a execução, nomes e registro do serviço. Após a confecção, a aplicação utiliza o novo serviço como todo outro serviço disponível. Um serviço pode empregar outros serviços remotos e para tanto é possível a comunicação entre serviços.

3.1.2. Agência Orientada a Serviços

A arquitetura da agência que propomos é composta de agentes da aplicação, de um intermediador de componentes (ORB), de Serviços e Facilidades CORBA, de uma coleção de agentes de serviços, de um serviço de Mobilidade de agentes (com *Gerente de Ativação* e *Repositório de Implementação*) e de um serviço de Disponibilidade [Schulze97-2]. Consideramos um Gerente de ativação associado ao serviço de ciclo de vida de agentes, o que inclui um Adaptador (de Objetos) [OMG98-1, 98-5] com suporte para *Registro Remoto*. Uma agência com serviço de Mobilidade e de Disponibilidade é capaz de carregar novos componentes, isto é, a agência está aberta a novos agentes de acordo com a demanda de uma aplicação. Fazendo um paralelo com o modelo CORBA, um serviço oferecido por uma agência é um agente de serviços e o ORB é um mediador de agentes, como na Figura 3.3.

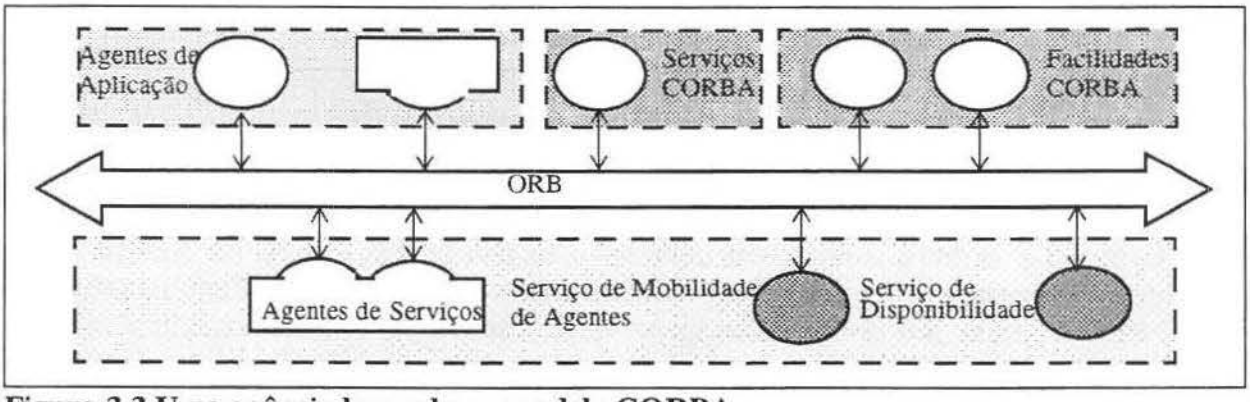


Figura 3.3 Uma agência baseada no modelo CORBA.

Middleware: Este trabalho foi concebido como parte de um projeto de plataforma distribuída denominada *Multiware*. A plataforma se encontra representada na Figura 3.4, com as suas camadas e funcionalidades.

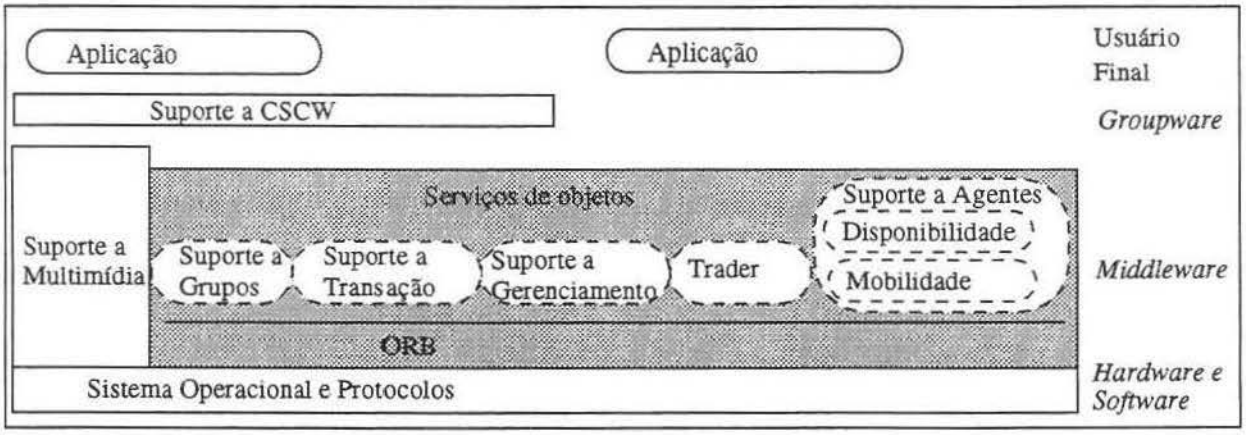


Figura 3.4 Plataforma *Multiware*.

A plataforma incorpora serviços de objetos com módulos para suporte a diferentes funcionalidades como: gerenciamento [Queiroz97], transação, grupos, *Trader*, e agentes. CORBA permite um bom grau de flexibilidade na implementação do núcleo ORB, podendo ser implementado como um conjunto de bibliotecas de execução, como um conjunto de processos ativos, como um servidor, ou como parte de um sistema operacional [Orbix96].

3.1.3. Ciclo de Vida e Persistência

Identificamos fases diferentes no ciclo de vida de um agente: inicialização, estacionário, migração e remoção. A fase de inicialização envolve contratar e distribuir. A fase estacionária de um agente pode ser provisória ou indefinida de acordo com as características do serviço. Disponibilizar agentes de uso geral envolve o gerenciamento e a distribuição destes agentes a fim de garantir sua disponibilidade tanto quanto possível. Podemos pensar em serviços estacionários a maior parte do tempo desde que não haja nenhum problema na rede ou no hospedeiro onde o agente se mantém funcionando. No caso de degradação no ambiente, o agente deve poder migrar de forma a preservar a sua disponibilidade. A fase de migração de um agente envolve persistência de código e estado, isto é, antes de migrar o agente deve armazenar seu estado de forma persistente. Estado e código são passados como seqüências (*streams*) e armazenados persistentemente na agência destinatária, seguido de remoção na agência remetente. No momento da (re)ativação o agente recupera seu estado para as variáveis originais. Em todas as ocasiões, o agente é reativado pelo suporte de agentes para recuperar o seu estado de execução, podendo inclusive seguir inativo, se este for o estado seguinte. A fase de remoção se refere a desinstanciação de um serviço. Esta fase pode ser realizada com o auxílio de um agente móvel passando como uma ficha *(token)* cuidando da desinstanciação de uma aplicação de forma apropriada.

3.1.4. Mobilidade

O suporte de mobilidade cuida da recepção de um agente, seu armazenamento persistente e o registro de sua interface no ORB. Uma chamada ao suporte de mobilidade envolve: (des)serialização, emissão(recepção) do agente e remoção(armazenamento) persistente do estado. No remetente é necessário: desabilitar qualquer nova requisição, terminar a execução de requisições atendidas, desativar e colocar o agente na fila de envio. Em contrapartida, no receptor é necessário: a publicação da interface do agente e a sua (re)ativação. O serviço de persistência é necessário para o armazenamento de código e estado.

3.1.5. Disponibilidade

Disponibilidade de uma Agência. Para um agente novo ser disponibilizado, é necessário encontrar e alocar recursos em uma agência. Para identificar agências abertas a novos agentes e seu nível de disponibilidade, é introduzido um serviço de Disponibilidade nas agências. Este serviço de Disponibilidade avalia a carga da agência utilizando métricas de desempenho [Queiroz97] como: tempo de resposta, *throughput* e utilização. A utilização se baseia em parâmetros diversos para avaliar a ocupação em termos de: cpu, memória, disco, atividade de rede, número de usuários (processos), e assim por diante. Estes valores são computados com

um *specmark* do hospedeiro para permitir um valor comparativo entre hospedeiros. O nível de disponibilidade de uma agência pode ser obtido através de consulta direta à agência ou através de um Catálogo com propriedades dinâmicas [Trader95].

Disponibilidade de um Agente. Para selecionar e localizar um agente de serviço recorremos ao serviço de Disponibilidade da agência local, conforme descrito acima. Em todos as situações, os domínios precisam ter um *gateway* de entrada que entenda o protocolo específico de agentes, por exemplo, IIOP [CORBA98]. Na Figura 3.5, duas ou mais agências podem estar interligadas por um ORB, ou seja, existe uma porta cujo endereço é previamente conhecido e configurado em nível de sistema operacional. Neste caso, um cliente pode utilizar os serviços do ORB para localizar um agente, desde que este esteja registrado no ORB local.

Para localização de agentes em agências que não estejam interligadas pelo ORB deve-se utilizar serviços de diretórios como Nomes e Catálogo. Um serviço de Nomes [OMG97-3] participa na localização de um agente a partir de propriedades, enquanto um serviço de Catálogo participa na localização a partir de propriedades com valores. Com um serviço de Catálogo é possível trabalhar faixas de QoS e diferentes formas de rastreamento de um agente. Alguns serviços de Catálogo do OMG foram padronizados [Trader95, OMG97-3], ou estão em fase de definição, por exemplo, para Comércio Eletrônico [CommerceNet97, Bichler98, Rodríguez99].

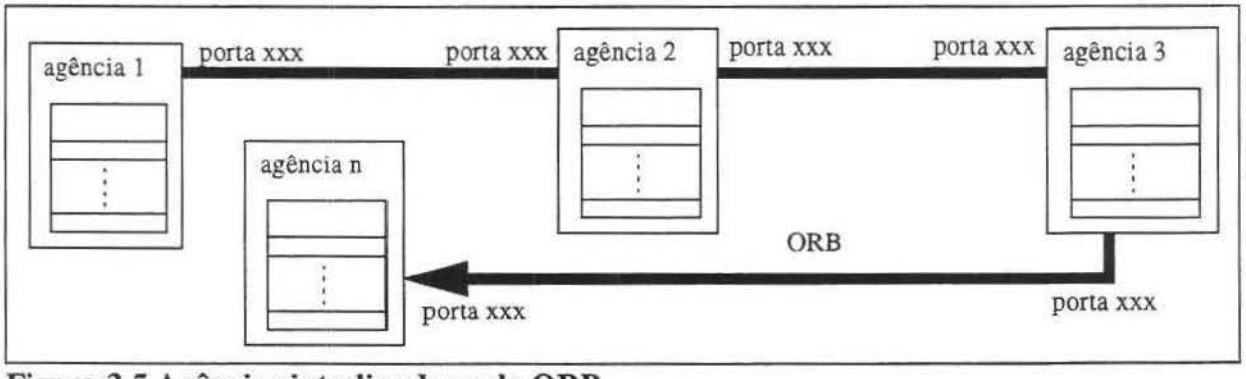


Figura 3.5 Agências interligadas pelo ORB.

Um serviço de diretório simples é o serviço de Nomes que armazena nomes e respectivas referências de objetos. Pode ser contactado através de uma porta predefinida e sua atuação é restrita a um determinado domínio, como representado na Figura 3.6.

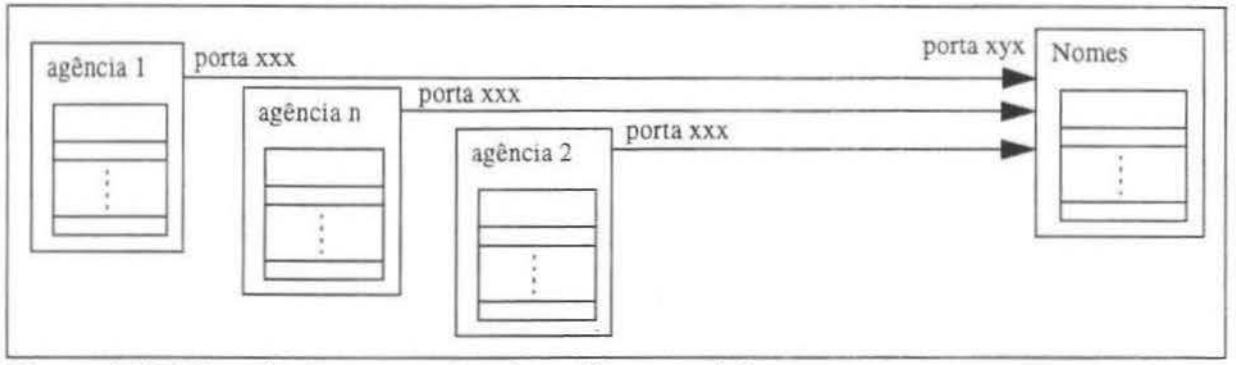


Figura 3.6 Utilização de um serviço de catálogo genérico.

Podemos localizar um agente a partir do seu domínio, isto é, se o agente tem um conjunto de agências as quais restringem a sua trajetória, então esta trajetória pode definir um domínio do agente e através de um membro do domínio pode ser possível localizar a instância ativa do agente, por exemplo, através do serviço de Nomes.

Uma federação de Catálogos pode retornar um apontador para o domínio do agente. Se a federação de Catálogos estiver interligada [Trader95] por conexões (*links*), então esta abordagem gera um custo de tráfego de mensagens entre os Catálogos. Uma alternativa para evitar estas mensagens entre Catálogos é utilizar um agente móvel de busca passando pelos catálogos. O agente de busca retorna com um conjunto de possibilidades a partir das propriedades declaradas na busca. Este procedimento apresenta maior latência mas com um volume menor de tráfego de mensagens.

O problema de latência pode ser contra-balançado com o uso de uma *cache* local onde o conjunto retornado pode ser armazenado e utilizado repetidamente antes de invalidado e nova busca ser necessária. Ainda para reduzir a latência, ressaltamos que o agente pode retornar quando obtém o conjunto desejado, não prosseguindo obrigatoriamente por toda a trajetória de Catálogos disponíveis. De forma resumida, uma busca passa primeiro por uma cache local, depois pelo serviço de Nomes e finalmente pelo serviço de Catálogo.

Podemos considerar uma federação de Catálogo, como na Figura 3.7, na qual um agente se registra através do Catálogo do respectivo domínio no qual se encontra. Quando sai da federação (domínio ou região) o agente se desregistra. Uma outra abordagem possível é o uso de uma *proxy* onde o agente mantém a sua localização. A localização da *proxy* é conhecida e aponta para a localização do agente.

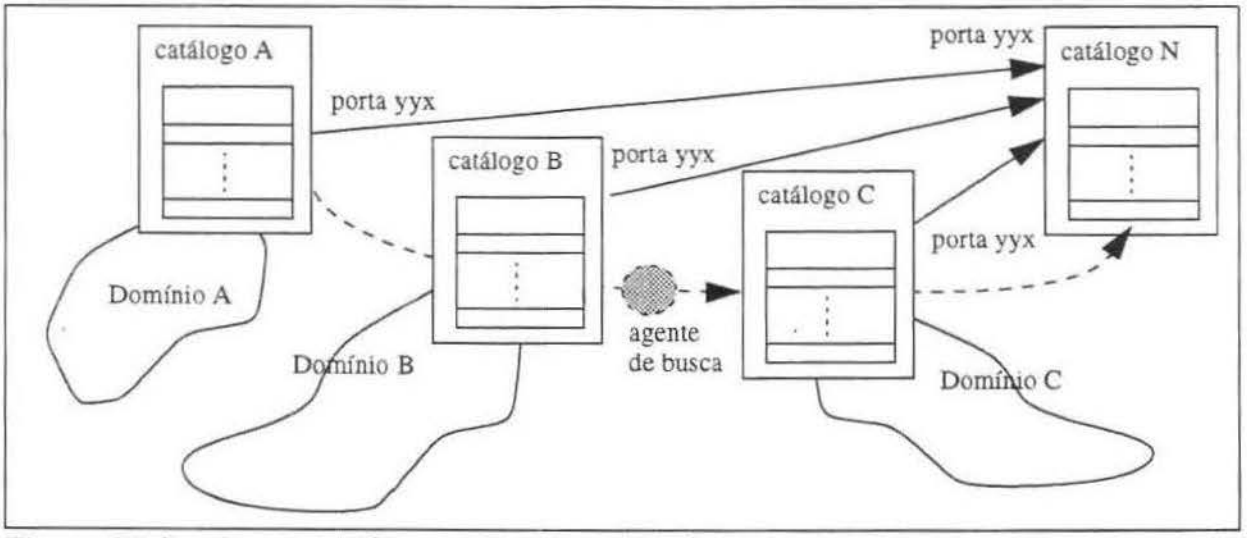


Figura 3.7 Serviços de catálogos federados e hierárquicos.

Uma *proxy* também pode apontar para uma outra *proxy* e assim por diante, como mostrado na Figura 3.8. Neste caso temos um menor custo na migração sem a necessidade de atualização dos serviços de diretório mas em compensação temos um maior custo na localização do agente com sucessivas chamadas de proxies remotas.

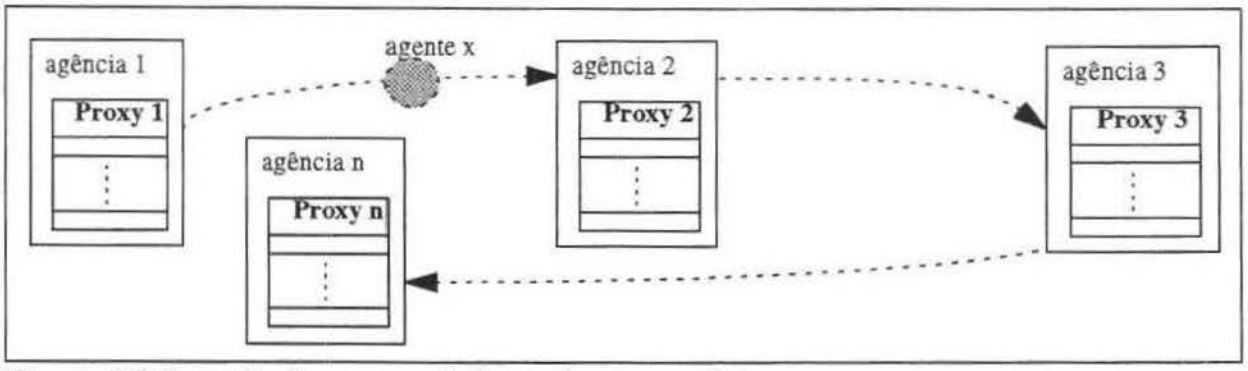


Figura 3.8 Traçado de um agente baseado em *proxies*.

3.1.6. Domínios Distribuídos Dinâmicos

Nesta seção abordamos a estruturação de domínios dinâmicos. Restrições com valores e tolerâncias são utilizados na busca de componentes em um domínio de Disponibilidade. Um domínio de Disponibilidade é composto de um conjunto de serviços de Disponibilidade e de serviços de Diretório: Nomes e Catálogos (Seção 3.6 - Serviços de Armazenamento). A estruturação de domínios dinâmicos é baseada na atualização de domínios de Agentes e domínios de Disponibilidade.

Um domínio de Agente é constituído de um conjunto de agências. Este é o domínio onde o agente está autorizado a ser instanciado e (re)ativado. O domínio é parte da informação do

agente disponibilizado num Catálogo. Um domínio de *Disponibilidade* é composto de domínios individuais de agentes e agências registrados num Catálogo. Catálogos podem conter informações com propriedades dinâmicas [Trader95] e obtê-las através de consultas remotas ou de um agente de consulta. Por exemplo, uma propriedade dinâmica pode ser a IOR de um agente e ser fornecida por um serviço de Nomes ou *serviço de Disponibilidade*.

Na Figura 3.9 (a e b), uma federação de Catálogo (B e C) é utilizada para compor domínios de agentes. Em (b), um novo domínio de agente é registrado no Catálogo-C e incorporado ao seu domínio. Um Catálogo tem um domínio composto das agências registradas nele.

Qualquer novo agente e agência deve ter sua informação registrada e disponibilizada em um Catálogo. A informação pode conter parâmetros de *autorização*, *propriedades* de disponibilidade, um repositório de *estado*, um repositório de *implementação*, um repositório de *interfaces*, e assim por diante. A migração de um agente depende do registro de parte desta informação em uma nova agência. Isto inclui IORs e propriedades (em serviços de Diretórios) e adaptadores de objetos portáveis (nas agências). A migração é intermediada pelos serviços de Disponibilidade das agências. O conceito de registro está relacionado com *export* no padrão ODP/Trader [Bearman96], *register* no padrão OMG/Trader [OMG96] e *region_registration* na especificação da Facilidade de Agentes Móveis [OMG97-2].

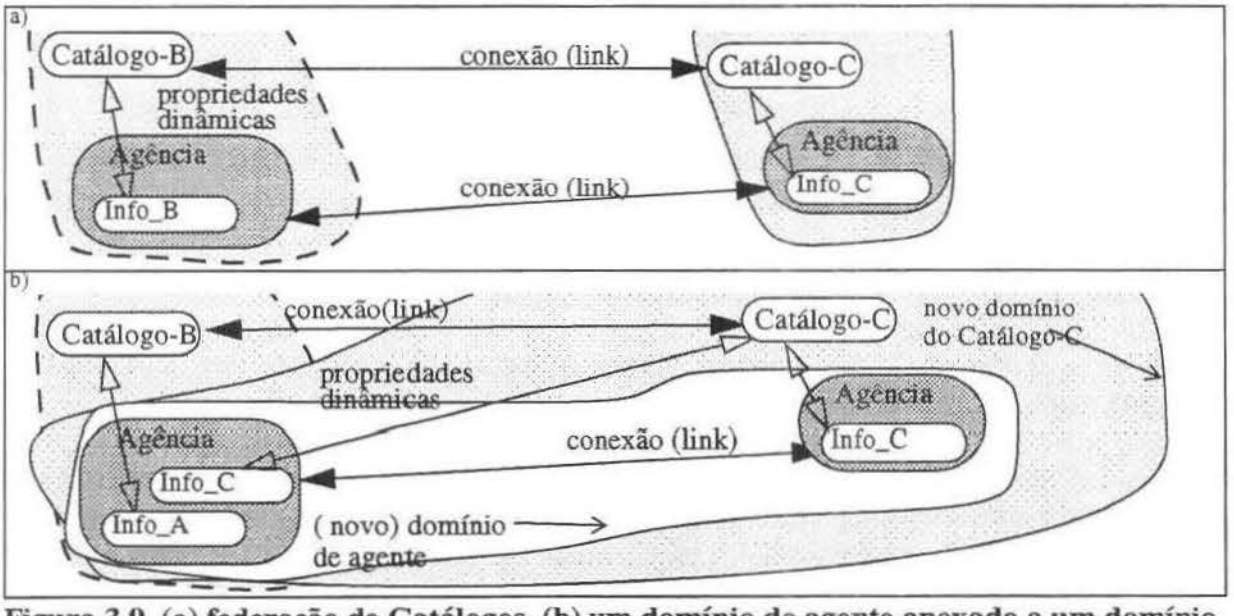


Figura 3.9 (a) federação de Catálogos. (b) um domínio de agente anexado a um domínio de Catálogo.

3.2 Migração Transparente em Sistemas de Agentes

Nesta seção tratamos de transparência de migração em sistemas de agentes [Schulze99-1]. Os sistemas correntes de agentes implementam a migração explícita de um agente, que é aquela determinada apenas pelo agente. Nesta seção apresentamos uma proposta de migração implícita, ou transparente, de um agente em direção a outro agente. Supondo uma variedade de ofertas de agentes, cabe estabelecer critérios de seleção e de negociação dos serviços oferecidos pelos agentes e ao final migrar o agente cliente em direção ao serviço selecionado.

A transparência de migração ocorre através de uma interface de transparência do serviço de Disponibilidade e sua associação ao serviço de Mobilidade. Estão presentes numa agência uma coleção de agentes (ativos ou apenas registrados), um Gerente de Ativação cuidando do (des)registro (via adaptador de objetos) e (des)ativação de agentes, e uma porta de comunicação entre agências. Em cada agência é incorporado um serviço de Disponibilidade. A migração transparente envolve reconfiguração dinâmica do ambiente. O mecanismo por trás do serviço de Disponibilidade é detalhado a seguir.

Estratégia Geral de Configuração Dinâmica:

- Agências e agentes estão publicados em um Catálogo.
- A informação de domínio de um agente pode estar inválida.
- A obtenção de um novo domínio é baseada em instantâneos e seleção aleatória de agências que satisfazem as propriedades e a qualidade declaradas na informação do agente.
- Instantâneos podem utilizar Catálogos com avaliação de propriedades dinâmicas.
- O agente é registrado em cada agência selecionada e a informação do agente atualizada.
- Balanceamento final de recursos através de requisições individuais. Utiliza o gerente de ativação do ciclo de vida de um agente (objeto) móvel.
- O serviço de Nomes é atualizado em cada (des)ativação do agente.

As Figuras 3.15a e 3.15b mostram respectivamente os processos de localização e reconfiguração, conforme descrito a seguir.

A Figura 3.10a mostra o processo de localização passando pela *cache* local, o serviço de Nomes e o Catálogo. Quando a informação é encontrada, a busca é concluída com uma atualização da *cache*. Na figura, um agente A requisita um agente B executando a estratégia a seguir:

(1) contacta serviço de Disponibilidade local;

(2a) consulta *cache;*

(2b) consulta Nomes;

(2c) consulta Catálogo;

(3) seleciona e armazena na *cache.*

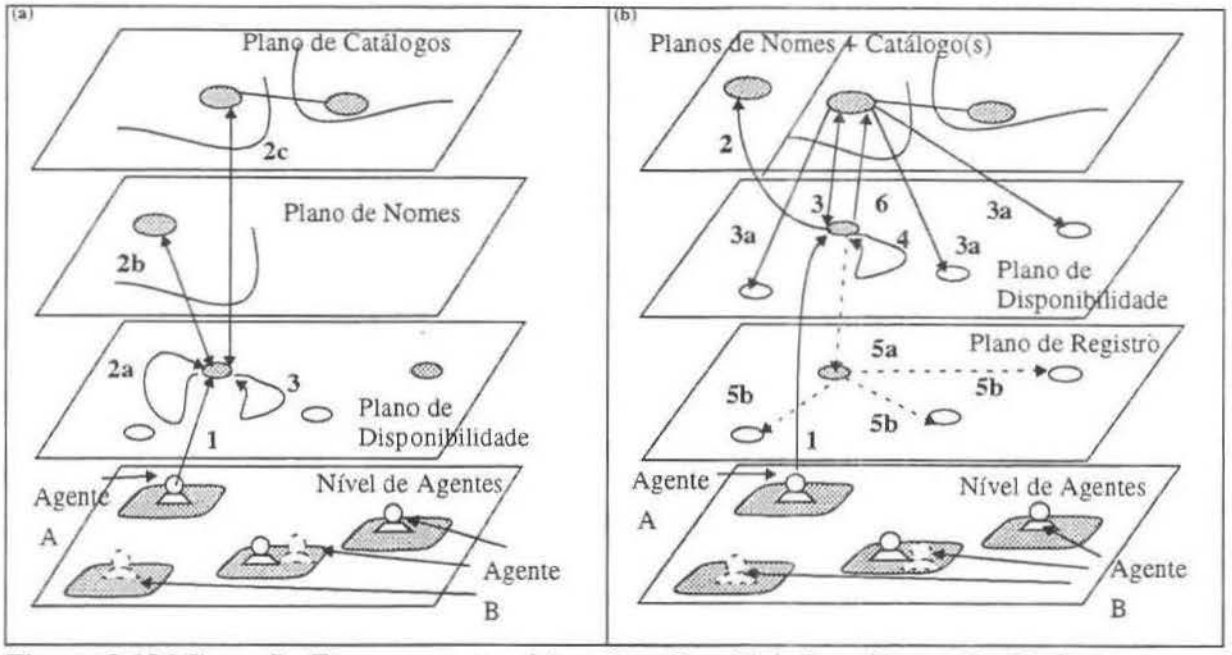


Figura 3.10 Migração Transparente: a) localização e b) (re)configuração dinâmica.

A Figura 3.10b mostra o processo de reconfiguração baseado na atualização do registro nas agências e atualização da informação no serviço de Nomes e Catálogo(s). Ao final da seqüência a informação da localização do agente B será armazenada na *cache*. Na figura, um agente A requisita um agente B, encontra um domínio de B inválido e constrói um novo domínio para B. A estratégia consiste dos passos a seguir:

(1)contacta serviço de Disponibilidade local;

(2) atualiza Nomes com instância ativa de A;

(3) consulta Catálogo **(3a)** propriedades dinâmicas;

(4) seleciona agências e armazena na *cache;*

(5a,b) registra nas agências selecionadas;

(6) atualiza informação de B.

A Figura 3.11 mostra a fase de balanceamento final de recursos em que a consulta via o Catálogo é evitada enquanto estiverem sendo satisfeitos os critérios da execução. Na figura, um agente A requisita um agente B e encontra a informação de B na cache local. A estratégia consiste dos passos a seguir:

(1) contacta o serviço de Disponibilidade local;

(2) consulta a *cache;*

(3) seleciona as agências disponíveis via requisições sucessivas;

(4) (re)ativa a instância de B com ajuda do gerente de ativação na agência remota;

(5) *A* faz conexão através do serviço de Disponibilidade local e migra na direção de *B*, ou

(6) *B* faz a re-chamada através do serviço de Disponibilidade local e migra na direção de *A*.

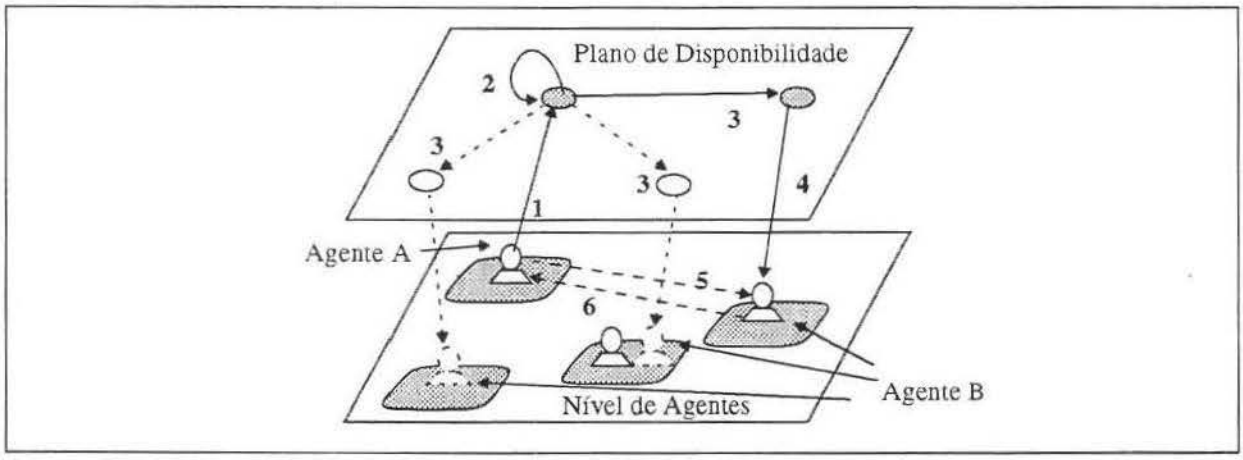


Figura 3.11 Migração Transparente: Migração Final.

As consultas via serviço de Catálogo são sempre mais custosas. Para minimizar inconsistência nas informações, a publicação de um agente somente é escrita em um Catálogo para posteriormente ser copiada por outro serviço de Disponibilidade para sua *cache* local. Somente o serviço de Disponibilidade escreve na sua *cache* local por razões de consistência. As estruturas da informação de agentes e agências estão na seção seguinte, nas Tabela 3.2 e 3.3.

Esquema de Migração Implícita: A partir das estimativas de tráfego [Baldi98, Vigna98] discutidas no Capítulo 2, tratamos a migração transparente na forma de um migração do agente cliente no sentido do agente servidor, Figura 3.12a. Se a mobilidade do agente cliente estiver vedada pela implementação da sua parte autônoma, então a migração poderá ocorrer no sentido contrário, isto é, do agente servidor em direção ao cliente durante a re-chamada do servidor ao cliente, quando os papéis se invertem, Figura 3.12b. Se esta migração também está vedada na parte autônoma do segundo agente, então a execução prossegue remotamente, Figura 3.12c. Se a execução remota não é tolerada por algum dos agentes então a conexão é abandonada. Uma avaliação do tamanho do estado do agente para decidir sobre a sua migração ou não fica a critério da implementação do agente. Como regra geral de projeto recomendamos que o estado de um agente seja mantido de tamanho reduzido, por exemplo, menor que o tamanho do código.

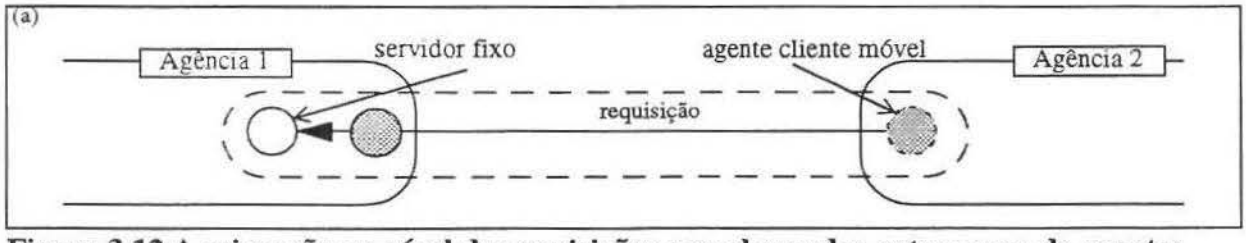


Figura 3.12 A migração ao nível de requisições e re-chamadas entre pares de agentes.

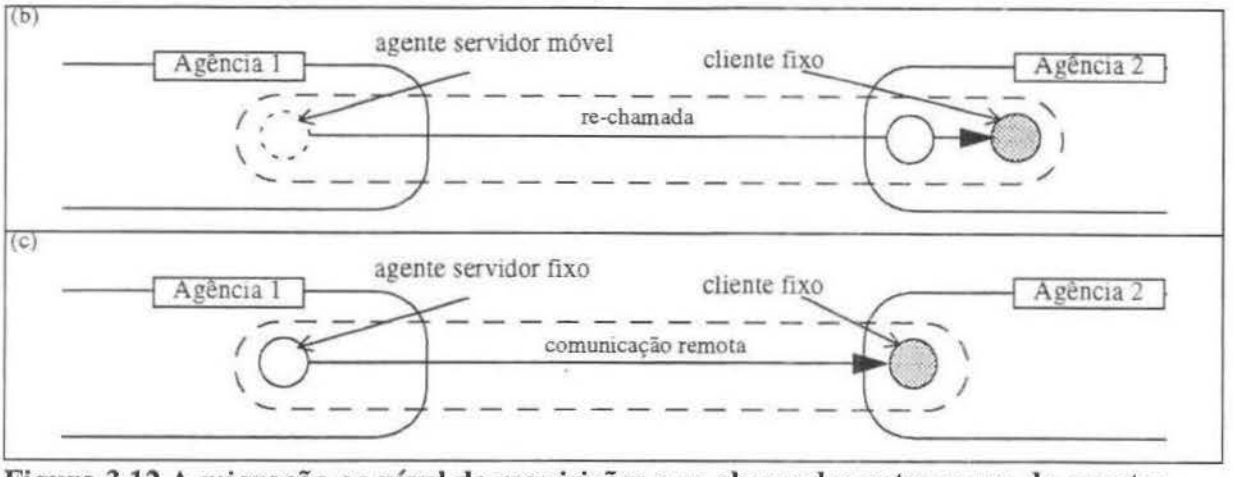


Figura 3.12 A migração ao nível de requisições e re-chamadas entre pares de agentes.

3.3 Agência

Para o modelo de uma agência, identificamos uma correspondência entre o modelo CORBA, proposto no MASIF [OMG97-2, Krause97] e o Modelo de Referência RM-ODP [RMODP95]. Na Figura 3.13 apresentamos este modelo comparativo a partir do modelo genérico de agência do RM-ODP.

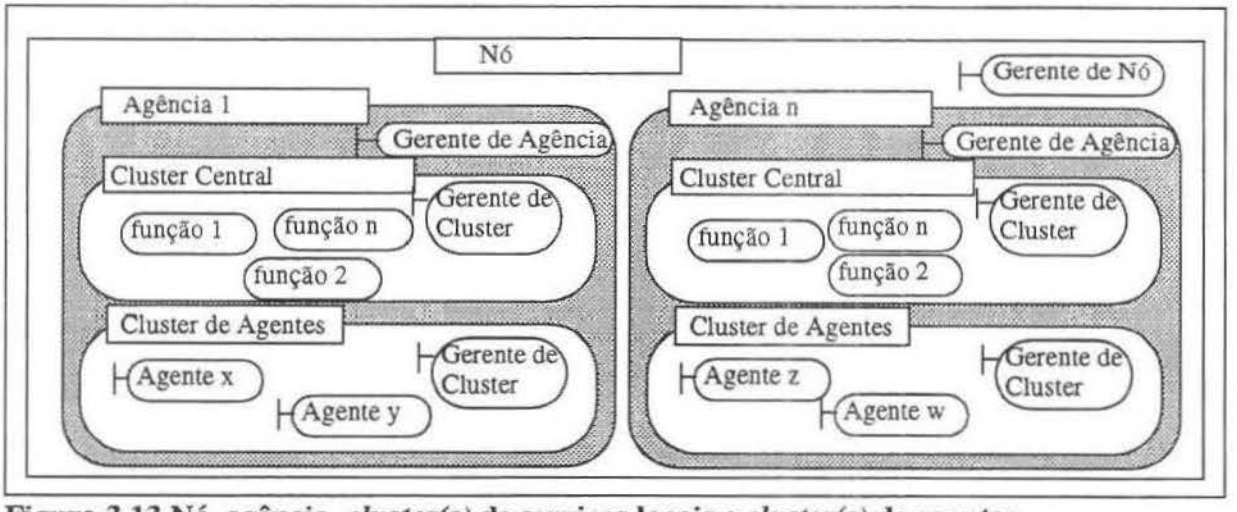


Figura 3.13 Nó, agência, *cluster(s)* de serviços locais e *cluster(s)* de agentes.

Na figura, em um nó (RM-ODP) podem residir uma ou mais agências (CORBA) ou sistemas de agentes (MASIF). A cada agência ou sistema de agentes corresponde uma porta de comunicação. Ao gerente de nó corresponde o sistema operacional (do hospedeiro). A cada Gerente de Agência corresponde um Gerente de Ativação de sistema de agentes. No padrão CORBA corresponde ao *POAManager* presente no POA. Ao *cluster* central da agência correspondem os objetos de serviços locais de cada sistema de agentes, Em CORBA corresponde aos serviços

CORBA. A cada *cluster* de agentes corresponde um *lugar* no MASIF com os respectivos agentes. Em CORBA/POA corresponde a um processo servidor com objetos serventes (*servants*). A um gerente de *Cluster* não há correspondente no MASIF, mas em CORBA/POA corresponde ao *ServantManager*.

Na Figura 3.14 temos uma representação dos componentes da agência proposta, baseada em CORBA. Consiste do serviço de Disponibilidade com a interface de Transparência, Avaliador(es) de Propriedades(s) Dinâmicas (Trader) e o serviço de Mobilidade (com Gerente de Ativação e Repositórios diversos). Os serviços iniciais [Apêndice A] utilizados pelo Serviço de Disponibilidade incluem: Serviço de Nomes, Serviço de Catálogo, Repositório de Interfaces, Repositório de Implementação (contém informações sobre ativação e código), Gerente de Ativação (relacionado ao ciclo de vida do agente) e Repositório de Estado (persistência do agente). No caso dos serviços iniciais utilizados por um agente, é suficiente o Serviço de Disponibilidade local. Detalhes sobre os principais componentes se encontram nas seções a seguir.

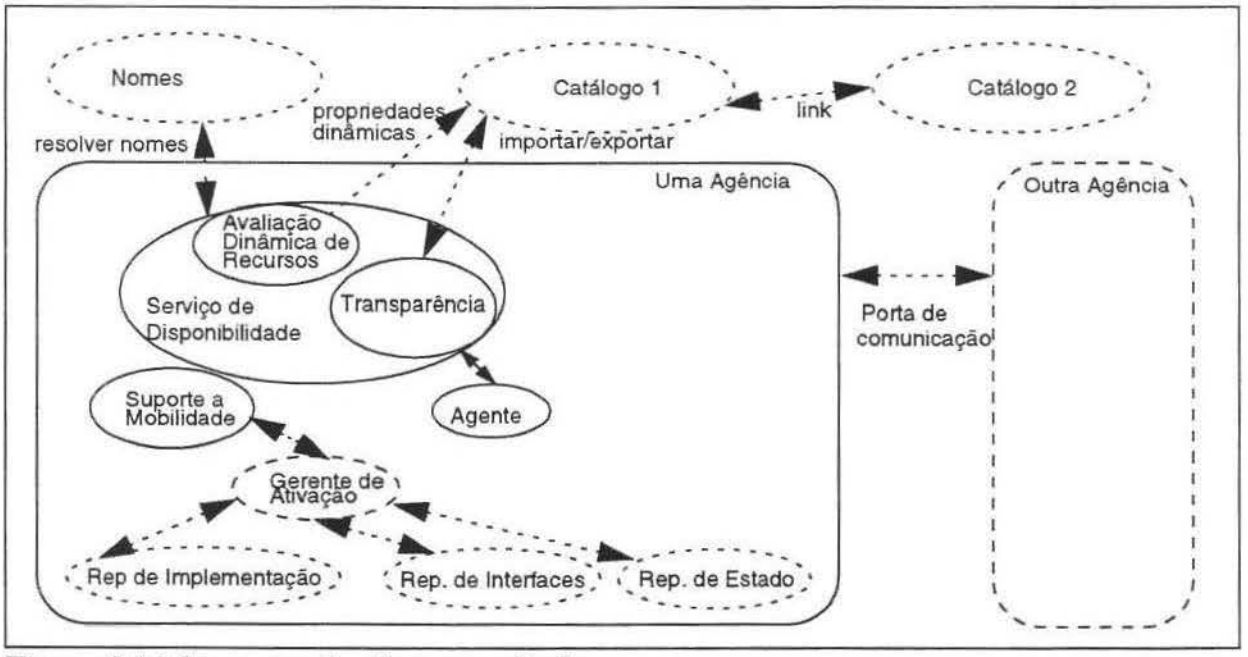


Figura 3.14 Componentes de uma agência.

3.4 Serviço de Disponibilidade

Em [Schulze97-1] introduzimos o conceito de um serviço de Disponibilidade a atuar em conjunto com um suporte de mobilidade para migrar transparentemente agentes ou serviços. Os critérios de migração são baseados em balanceamento de carga utilizando *caching direto* ou *inverso* [Goldszmidt96]. Ou seja, considera-se preferencialmente co-localizar cliente e servidor na mesma agência através da migração do cliente em direção ao servidor, ou alternativamente do servidor em direção ao cliente. A co-localização tem por objetivo reduzir o tráfego de mensagens na rede. Se a migração não se concretiza então considera-se a execução remota de cliente e servidor.

Como mencionado anteriormente, a migração de agentes pode ser tratada transparentemente se passar a fazer parte da realização do objetivo da computação a ser realizada. Por exemplo, métricas de desempenho podem ser incluídas no objetivo, e para atingir estas métricas consulta-se um serviço de Disponibilidade local. Na Figura 3.15 está representado o serviço de Disponibilidade interfaceado com um ORB.

O serviço de Disponibilidade é importante para oferecer recursos a novos agentes em fase de implantação. A fim de identificar uma agência aberta para receber um agente novo, o serviço da disponibilidade informa o nível de disponibilidade da agência. Este nível é publicado de modo que possa ser obtido através de consulta à agência ou através de um Catálogo. Uma disponibilidade deve considerar alguma retroatividade no tempo, refletindo a duração do novo serviço. Isto exige um histórico de ocupação em cada agência.

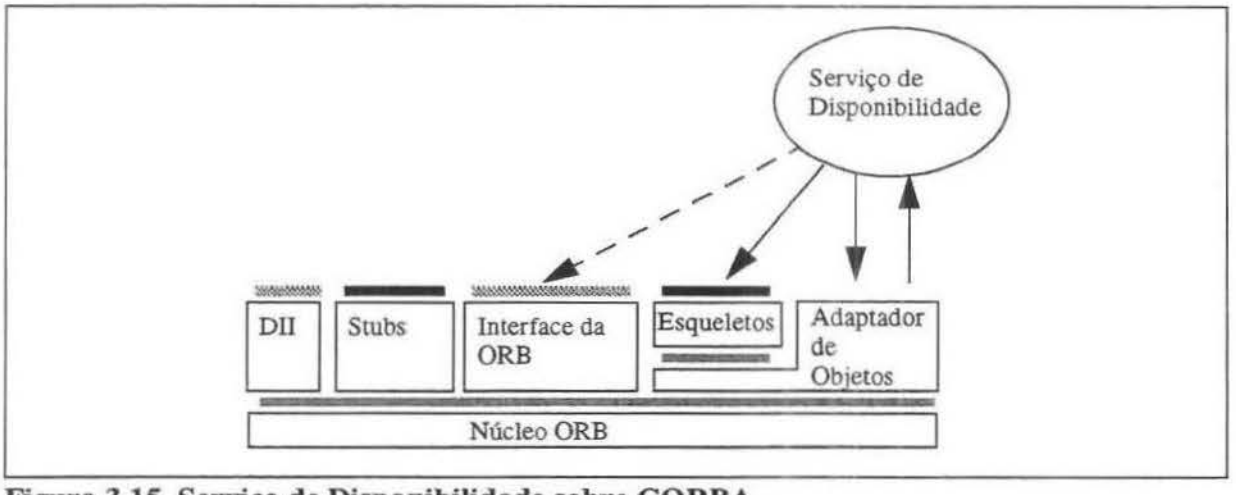


Figura 3.15 Serviço de Disponibilidade sobre CORBA.

Na Figura 3.16 está o diagrama proposto de classes e relacionamentos, em notação UML [BOOCH99], com uma representação macroscópica dos suportes de mobilidade e de disponibilidade com outros serviços CORBA mencionados mas não detalhados.

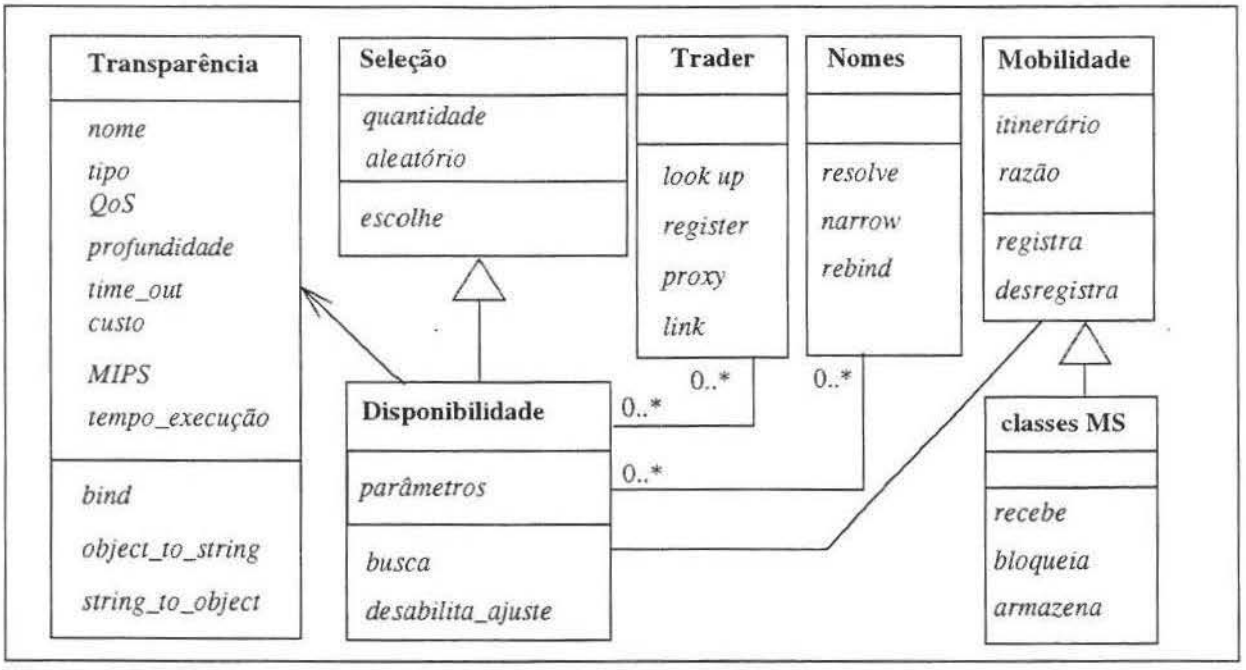


Figura 3.16 O macro modelo do serviço de Disponibilidade.

Transparência: Esta é a interface disponível localmente a um agente. Os *parâmetros* são gerais. O atributo nome define um identificador para o agente desejado. O atributo *tipo* é a razão para a transparência ou tipo de aplicação, por exemplo: desempenho, tolerância a falha e assim por diante. O atributo *QoS* é usado conjuntamente com o *tipo* e influencia na faixa de qualidade de serviço(s) desejada na seleção. O atributo *profundidade* serve para limitar a profundidade nas conexões remotas e junto com *time_out* limita a pendência das conexões. O atributo MIPS é usado conjuntamente com *tempo_execução* para obter o desempenho desejado de cpu. O método *bind* é usado para resolver o nome de um objeto ou agente remoto. O método *object_to_string* transforma uma referência de objeto em *string* e o método *string_to_object* converte de volta a referência de objeto.

Mobilidade: Um atributo *itinerário* pode ser uma agência ou um domínio de agências. Um atributo adicional *razão* indica o tipo da estratégia de migração, isto é, explícita ou implícita. O itinerário pode ser seguido sequencialmente ou aleatoriamente. Um método *(des)registra* (des)registra a referência de agente em cada agência do domínio. A classe mobilidade aparece como super classe no modelo de suporte à mobilidade apresentado mais adiante. A classe MS faz parte do suporte à mobilidade [Vasconcellos98] e será detalhada adiante. Um método *bloqueia* o agente para novas requisições. Um método *armazena* o estado. E um método *recebe* o agente: ativa e recupera o estado.

Disponibilidade: Esta é a implementação da classe *Disponibilidade* no servidor de disponibilidade. Os atributos *parâmetros* são os recursos a serem buscados, como ilustrado na interface de transparência. O método *busca* procura os componentes (agentes ou agências) e conecta a eles. A estratégia é: 1.busca na cache; 2.busca no serviço de Nomes; 3.busca no Trader; 4. seleciona; e 5. conecta a instância ativa. O método *desabilita_ajuste* (ou *desabilita_migração*)

simplesmente comuta a mobilidade para implícita através do atributo *razão*, na classe *mobilidade*.

Seleção: É uma classe interna de Disponibilidade contendo o método *escolhe*. A escolha pode ser aleatória ou sequencial e retornar um número de elementos especificado pelo atributo *quantidade*.

A Figura 3.17 apresenta um diagrama geral de eventos para a migração explícita e implícita. A forma genérica é uma requisição ao serviço de Disponibilidade por um componente com suas propriedades. A seqüência de migração pode começar a partir de uma requisição por um agente do qual pode-se conhecer completamente as propriedades inclusive a localização ou apenas parcialmente algumas propriedades e restrições de uso.

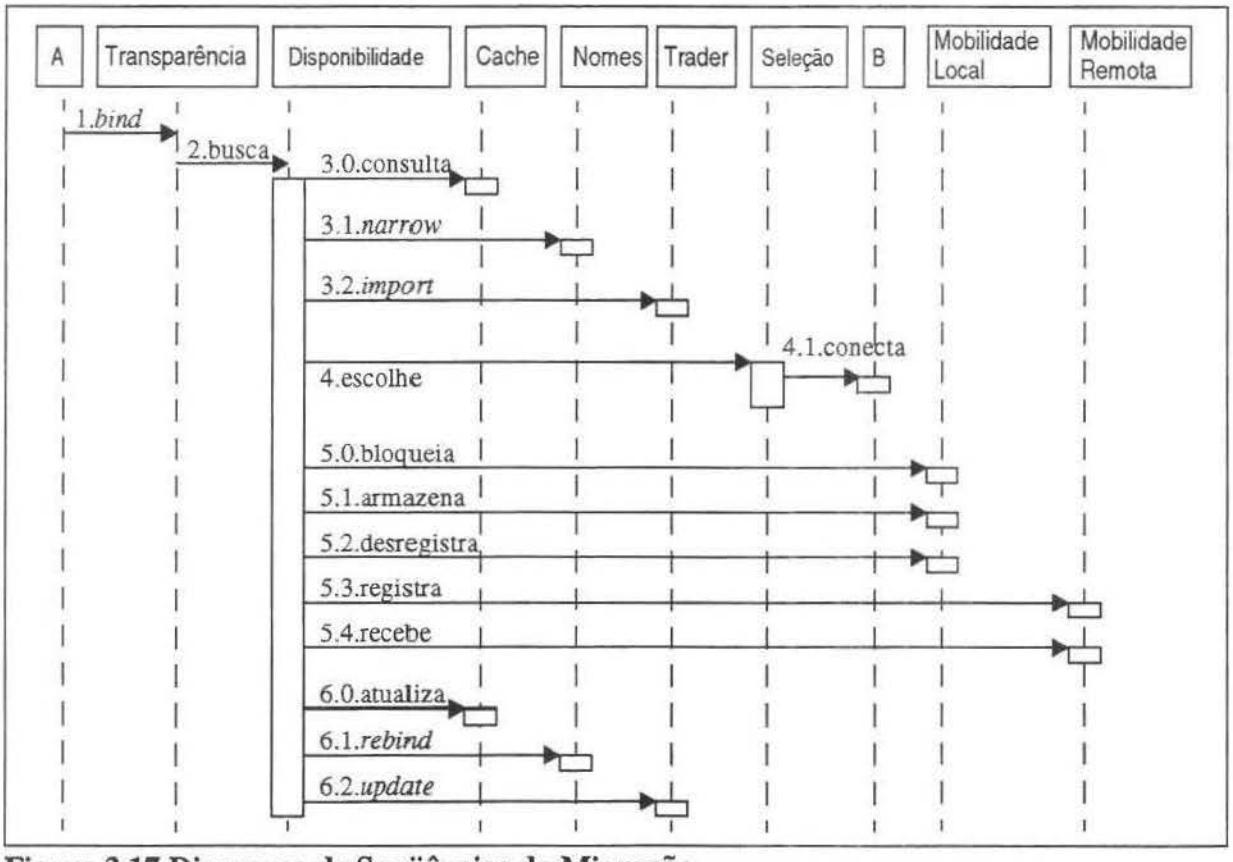


Figura 3.17 Diagrama de Seqüências da Migração.

A partir do diagrama da Figura 3.17, em notação UML, analisamos o caso de uma migração implícita onde um *agente A* necessita de um agente B. Se usar a chamada de método especificada por CORBA, A executa remotamente os métodos de B. Entretanto, usando migração implícita, ao requisitar uma operação de B, A vai migrar em direção a B. Para tanto, o *agente A* requisita o método *busca* do serviço de Disponibilidade. Este método consulta o domínio local e não encontrando o recurso localmente, passa ao serviço de Nomes ou Trader. O retorno da *busca* é o conjunto das alternativas encontradas. O retorno de um conjunto vazio resulta uma

exceção devido a um insucesso na *busca*. A partir de um conjunto não vazio é feita uma escolha (*escolhe*). O comportamento do método *escolhe* da classe *Seleção* se modifica em função do parâmetro *aleatório*. Sem o parâmetro *aleatório* habilitado, o método escolhe aquele componente que primeiro atendeu a sua solicitação. Este procedimento se baseia no parâmetro *quantidade* de elementos considerados. Com o parâmetro *aleatório* habilitado, o método faz uma escolha aleatória a partir do conjunto obtido da(s) consulta(s). A escolha retorna tipicamente um elemento, mas pode retornar também mais de um, por exemplo, a escolha de um conjunto de agências para compor o domínio de um novo agente.

Terminadas as etapas de busca e seleção (1, 2, 3 e 4) começa a etapa de migração (5) seguida das atualizações (6) necessárias. Para mover um agente, a agência remetente *bloqueia* novas requisições e *armazena* o estado do agente. Uma agência excluída do domínio *desregistra* o agente. Uma agência incluída no domínio *registra* o agente. A agência destino que *recebe* o agente lê o estado e ativa a instância. Uma exclusão e/ou inclusão no domínio requer uma atualização na cache da agência remetente e destino, no serviço de Nomes (*rebind*) e no Trader (*update*).

A partir do mesmo diagrama de eventos analisamos o caso de um migração explícita onde um *agente A* migra para uma determinada agência X da sua trajetória. Como a agência destino é conhecida, as etapas 3 e 4 não são efetivadas. Introduzindo aleatoriedade na informação de destinatário observamos o surgimento de uma componente de migração implícita. Por exemplo, o agente quer se locomover para outro nó de sua trajetória não importando qual, contanto que percorra todos eles sequencialmente. A requisição ao serviço de Disponibilidade informa o domínio do agente mas não o nó específico e a seqüência do diagrama inclui a etapa 3 de busca e a etapa 4 de escolha com o parâmetro *aleatório*.

As classes e seqüências apresentadas acima implementam as estratégias descritas anteriormente neste capítulo, na Seção 3.2, Figuras 3.10 e 3.11.

3.4.1. Oferta de Disponibilidade

As ofertas de disponibilidade, como apresentadas na Tabela 3.1, podem estar organizadas em métricas básicas de uso geral e métricas especializadas de uma classe de aplicação além de métricas específicas da aplicação em particular. Os tipos especializados herdam os tipos básicos e podem incluir métricas para: *desempenho*, *gerenciamento*, *segurança*, entre outras.

Ofertas Básicas		**Classe de Aplicação para**
Estáticas	**Dinâmicas**	**Ofertas Especializadas**
recursos de *hardware*	alocação de recursos	desempenho
localização física	ocupação de recursos	gerenciamento
comunicação	...	segurança
protocolos		...
tempo médio entre falhas - MTBF		
...		

Tabela 3.1 Tipos básicos de disponibilidade.

Os tipos básicos de ofertas de disponibilidade para agentes e agências estão organizados como nas Tabelas 3.2 e 3.3, podendo ter propriedades dinâmicas, isto é, propriedades que são atualizadas em tempo de consulta. A funcionalidade de propriedade dinâmica deve ser utilizada com critério pois pode introduzir custo adicional no tráfego de mensagens e latência. Os valores atribuídos a cada um dos tipos apresentados na Tabela 3.3 são utilizados para selecionar um agente como também para eventualmente selecionar novas agências para o agente executar.

Tabela 3.2 Tipos de ofertas de disponibilidade comuns a agentes e agências.

Tipo	Comentário
nome	identificador de nome
proprietário	identificador de proprietário
grupo(s)	identificador de grupo(s)
domínio(s)	identificador de domínio(s)
custo	identificador de custo
protocolo comunicação	tipo(s) de protocolo
modo ativação	
protocolo de segurança	tipo(s) de protocolo de segurança
palavra_chave(s)	palavra_chave(s) que identifica(m) alguma(s) funcionalidade(s)
localização	endereço de localização
método(s)	nenhum ou os métodos públicos
criação	data de instanciação
expiração	data prevista de expiração

Tabela 3.3 Tipos adicionais de ofertas de disponibilidade para agentes e agências.

Agentes		Agências	
Tipos	**Comentários**	**Tipos**	**Comentários**
cpu	desempenho mínimo necessário	cpu	desempenho oferecido
memória	mínimo necessário	memória	memória oferecida
canal de comunicação	velocidade mínima de canal de comunicação (KBytes/s)	canal de comunicação	velocidade do canal de comunicação (KBytes/s)
faixa de tempo de resposta	tempo max. de resposta sugerido e faixa percentual	tempo de resposta	tempo médio de resposta da agência
atualização	intervalo sugerido de atualização da localização de um agente	ocupação	ocupação global (%)
agências	número de agências	hospedeiros	número de hospedeiros
classes	http://~AOdir.AOclass	*throughput*	KBytes/s
estado	http://~AOdir.Statefile	porta #	porta de comunicação da agência

A disponibilidade permite avaliar ocupação em termos de diferentes parâmetros como: cpu, memória, disco, atividade de rede, número de usuários (processos), e assim por diante. A seleção pode computar estes números com *benchmarks* dos hospedeiros para uma decisão comparativa e finalmente usar uma computação implícita ao privilegiar as conexões efetivadas. Esta

computação implícita significa uma função de ativação onde o número de tentativas e o tempo de resposta totalizados correspondem a um nível de ocupação:

$$\gamma = \sum k(t)t \qquad \text{(eq. 1)}$$

para $k(t) = [0,1]$.Um limiar $\Gamma = NT$ estabelece o nível máximo de ocupação tolerado, $\gamma < \Gamma$, onde N é número máximo de tentativas, T é tempo máximo de espera de cada tentativa.

3.4.2. Propriedades Dinâmicas

A avaliação de propriedades dinâmicas deve considerar um tempo de amostragem. Uma forma de estimar este tempo de amostragem é que seja igual ou maior ao tempo de execução pretendido. Esta avaliação exige agentes alocando recursos de execução e armazenamento. Se esta utilização de recursos se torna invasiva, pode-se considerar unidades em *firmware* dedicado, por exemplo, JavaChips [Javachip96] para tarefas de monitoração em aplicações distribuídas.

3.5 Suporte de Mobilidade

É uma facilidade CORBA adicionada ao ORB onde um agente móvel tem a estrutura de um objeto CORBA e visto como tal por qualquer outro agente. Agentes CORBA se registram em toda agência destino do itinerário como um objeto CORBA, e a fim de mover-se, solicitam desativação na origem e reativação no destino. Distinguimos um objeto móvel CORBA de um agente móvel CORBA estendendo a descrição anterior, isto é, um objeto se move devido a uma demanda externa enquanto um agente pode mover-se devido a demandas externas e internas.

Definimos previamente mobilidade explícita como a demanda de mobilidade determinada pelo agente. Externamente um agente associado poderá se mover em conseqüência do movimento explícito do primeiro agente. Esta mobilidade implícita ocorre ao nível de uma re-chamada do primeiro agente em resposta a um requisição do segundo, como representado na Figura 3.10.

O diagrama de classes para o suporte à mobilidade está representado na Figura 3.18, em notação UML, adaptando o suporte de [Vasconcellos98]. Estas classes são usadas na mobilidade explícita dos agentes de gerenciamento. Para a mobilidade implícita com a infra-estrutura de ajuste, usamos o *serviço de Disponibilidade* proposto em conjunto com o serviço de mobilidade. A classe *MS_1i* é usada para a primeira ativação de um agente enquanto *MS_2i* garante todas as ativações subseqüentes em outros hospedeiros. *MSc* interfaceia o controle da *Aplicação Gerente*. *Callbackimpl* implementa *Callback_abs* e permite o gerente ser re-chamado pelos agentes. A classe *BaseAgent* é uma classe abstrata e estendida para implementar o agente com *Agimpl*. A classe *Mthread* é responsável pela execução do agente. A classe *MScliente* permite lançar os agentes alternativamente através de linha de comando.

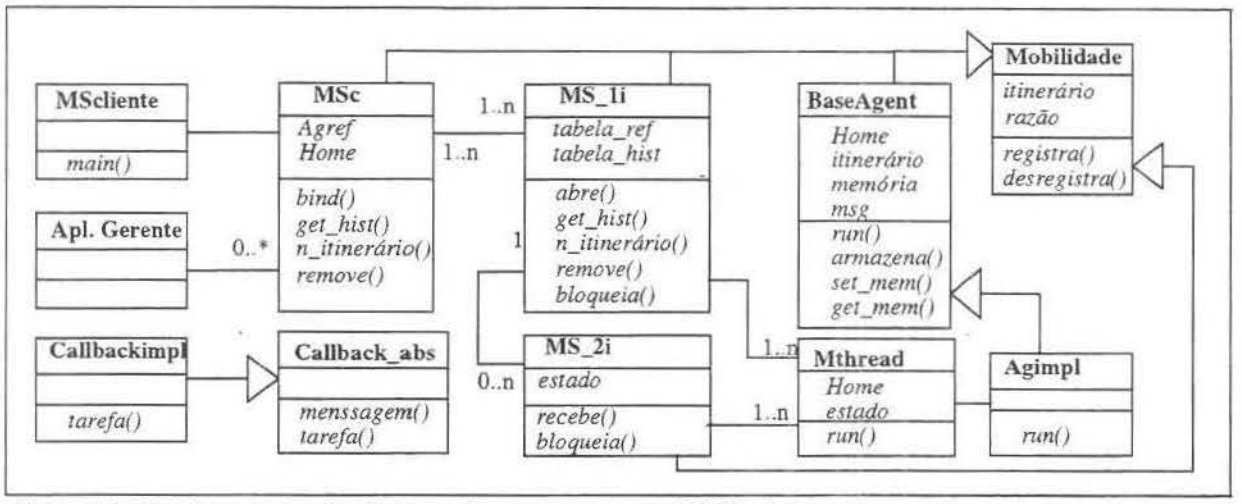


Figura 3.18 Diagrama de classes do suporte a mobilidade.

Um fluxograma do uso é apresentado na Figura 3.19. Neste fluxograma, *MSc* interfaceia a Aplicação Gerente que envia os agentes (1a e 2a). *MS_1i* é usado para a ativação de um agente na primeira agência (1b e 2b) e *MS_2i* para a ativação nas agências subseqüentes (1c e 2c). A *Mthread* executa o agente. Um agente 1 é instânciado no hospedeiro X, passa ao hospedeiro A e registra a sua nova referência no *MS_1i* correspondente (1d). Um segundo agente, é instanciado no hospedeiro Y, passa ao hospedeiro B e registra a sua nova referência no *MS_2i* correspondente (2d). Para comunicação, os agentes podem obter a referência atualizada de um agente com o respectivo *MS_1i* do agente procurado (1e e 2e).

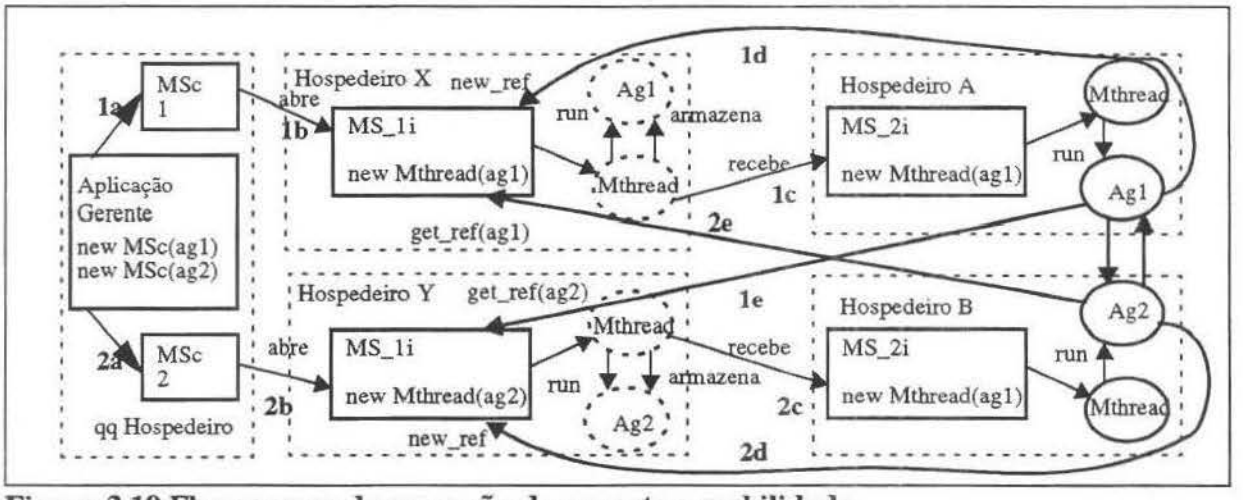


Figura 3.19 Fluxograma de execução do suporte a mobilidade.

3.6 Serviços de Armazenamento

Utilizamos o termo Serviço de Armazenamento de acordo com a definição de ODP para serviços que servem para consulta e atualização de informações sobre componentes de tipos diversos. Os Serviços de Armazenamento apresentados nesta seção estão organizados em Diretórios e Repositórios. Serviços de Armazenamento são tratados como componentes separados e que podem ser remotos. No nosso caso utilizamos estes serviços numa consulta onde a informação armazenada localmente na cache está inconsistente. A sua utilização pode gerar tráfego adicional de mensagens, seja para inserção, consulta ou remoção de uma entrada. Portanto recomendamos a utilização criteriosa de acordo com a aplicação desenvolvida. Em CORBA, este serviços são localizados utilizando a função *resolve_initial_services()* por ocasião da inicialização do ORB e seu adaptador de objetos portáveis [OMG98-5].

3.6.1. Diretórios

Serviço de Nomes. O serviço de Nomes [OMG97-3] é um componente estático com uma interface de escrita e consulta e uma porta de comunicação definida e conhecida ao nível de um domínio. O serviço de nomes é utilizado para armazenar entradas contendo os pares *nome* e *IOR* de um componente ativo. Uma nova entrada ou alteração no serviço de nomes normalmente é feita por ocasião da ativação ou desativação de um componente em alguma agência do mesmo domínio. Não são permitidas entradas com propriedades dos componentes que facilitem a sua identificação e seleção a partir de um conhecimento parcial. Neste caso deve-se utilizar um serviço de catálogo com propriedades, como o Trader, entretanto deve-se ter em mente que a latência de uma consulta fica maior. Em outros termos, deve-se manter a latência das consultas a mais baixa possível, tolerando uma latência maior quando a situação o exigir para contornar alguma inconsistência ou falha, mas não deve ser utilizado via de regra.

Na Tabela 3.4 temos a organização do espaço de nomes para o caso de agências e para o caso de agentes. A estruturação do espaço de nomes relativa a estes dois casos está representada na Figura 3.20 com uma raiz, os (sub)diretórios e as folhas terminando em IORs ou em agências.

Tabela 3.4 Organização do Serviço de Nomes para agentes e agências.

	Agências
Nome	IOR (Hospedeiro)
	Agentes
Nome	[Agência 1, ..., Agência n]
Nome	[Subdomínio 1, ..., Subdomínio n]
Nome	[IOR 1, ..., IOR n]

As árvores da Figura 3.20 representam a organização de nomes para agências e para agentes. Para uma agência, a árvore termina simplesmente numa folha com uma IOR, uma vez que uma agência é um componente fixo. No caso de um agente, a árvore começa numa raiz comum mas pode ser organizada de três formas diferentes. A primeira, no caso de um agente que se move pouco (ou nada), contém o nome do agente e a folha com a respectiva IOR do agente. A

segunda forma, contém o nome do agente e a(s) folha(s) com uma (ou mais) agência(s). A terceira forma, contém o nome do agente, passa por um nível com os *subdomínios* do agente e termina em folhas contendo as respectivas agências que compõem cada subdomínio. A IOR da agência é recuperada pelo nome da agência.

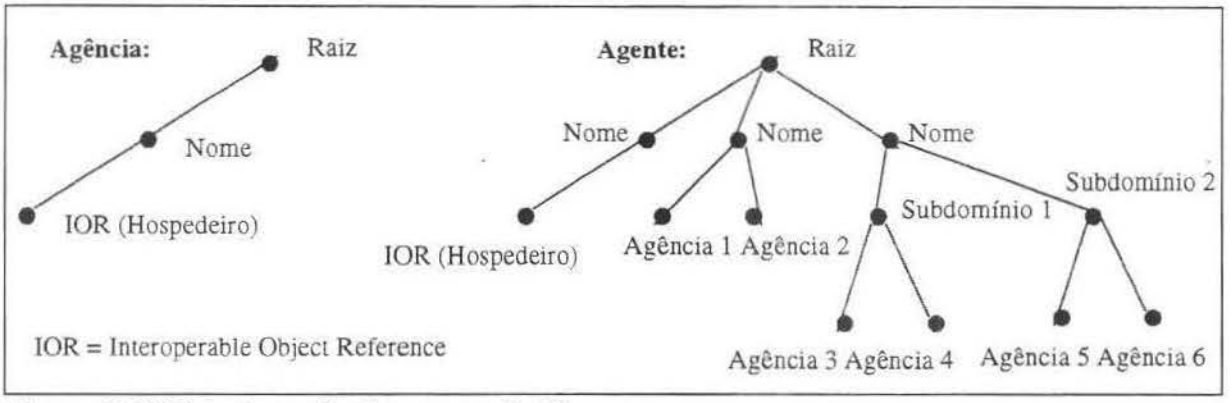


Figura 3.20 Estruturação do espaço de Nomes.

Serviço de Catálogo. Consultas através de um Catálogo [Trader95, Lima95, Rodríguez99] incluem uma faixa de disponibilidade de um tipo específico de componente (ou recurso). O Catálogo responde retornando uma lista. A fase de seleção pode incluir uma interrogação direta antes de contratação do componente. Na Figura 3.21 está uma representação genérica do de Catálogo proposto pelo OMG/Trader e suas interfaces na versão completa e *stand-alone* [Bearman96, OMG96].

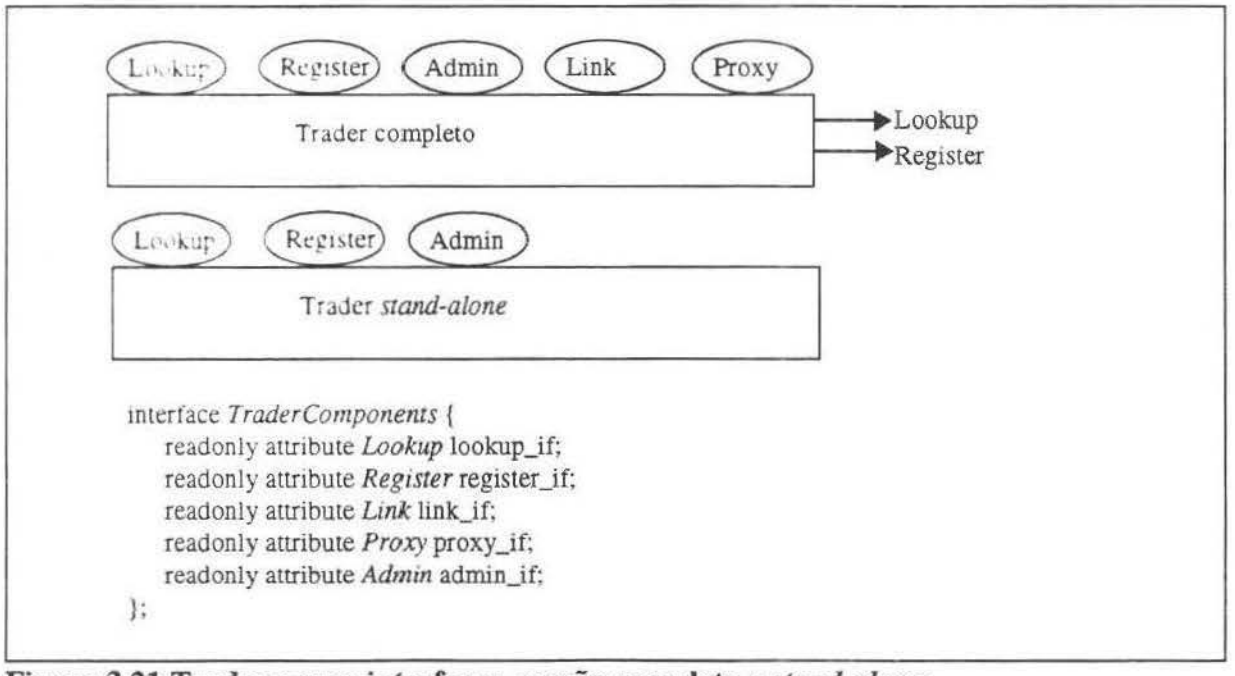


Figura 3.21 Trader e suas interfaces, versão completa e *stand-alone*.

A interface *Lookup* permite operações de consulta sendo a única operação nesta interface. *Register* serve para a entrada de novas ofertas. *Admin* permite a administração do Trader com a criação e remoção de estruturas de ofertas. *Proxy* permite as operações de aceitar, retirar, e descrever as ofertas de *proxy*. Se uma operação de consulta resulta em uma oferta de *proxy*, o Trader executa uma consulta aninhada para um Trader secundário indicado na oferta de proxy. *Link* permite que um Trader passe uma consulta a outro Trader dentro de uma federação de Traders. Através de *Funções Específicas* são permitidas propriedades dinâmicas, isto é, o Trader consulta um exportador de uma propriedade dinâmica através de uma interface *DynamicPropEval*. Está interface não fica no Trader mas no componente externo utilizado por ele. Uma consulta à interface *TraderComponents*, na Figura 3.21, retorna *nil* quando o Trader não dá suporte a interface em particular e uma exceção de *NotImplemented* é levantada pela invocação da operação da interface não suportada.

3.6.2. Repositórios

As implementações de CORBA permitem ainda diferentes serviços de repositório. Dentre eles ressaltamos:

- *Repositório de Implementação* - é um repositório onde ficam armazenados arquivos com detalhes de ativação de um servidor assim como informação sobre a localização (*:// repositorio_classes/classe.class*) da classe a ser utilizada. Se considerarmos mobilidade, nesta informação poderia ser incluída a localização (*://repositorio_estado/agente.ser*) do arquivo de estado serializado.
- *Repositório de Interface* - é um repositório onde ficam armazenados arquivos com as interfaces em IDL para utilização numa invocação dinâmica de CORBA, (*:// repostitorio_interfaces/*).
- *Repositório de Estado* - é um repositório que sugerimos para armazenar os arquivos contendo o estado serializado de agentes (*://repostitorio_estado/*). Pode vir a ser o serviço de Persistência [OMG97-3].

CAPÍTULO 4

Detalhes de Implementação

Quando você pára de esperar, tem todas as coisas.
Buda

Neste capítulo apresentamos uma visão geral das etapas de implementação. O capítulo está dividido nas seguintes seções:

- 4.1 Descrição Geral do Sistema de Agentes página 55,
- 4.2 Implementação a partir de CORBA página 60,
- 4.3 Implementação a partir de uma Plataforma de Agentes página 63
- 4.4 Comentários página 64 e
- 4.5 Definição de Classes página 67.

4.1 Descrição Geral do Sistema de Agentes

De acordo com o proposto no Capítulo 3 (Seção 3.1), deseja-se ter um sistema de agentes (Seção 3.3) que reúna as funcionalidades de agentes móveis (Seção 3.5) às funcionalidades do modelo CORBA de forma que os agentes possam ser tratados como objetos CORBA, isto é, um agente herda as funcionalidades de um objeto CORBA. A partir desta infra-estrutura agrega-se ao sistema a funcionalidade de migração transparente de agentes (Seção 3.2) baseada em um Serviço de Disponibilidade (Seção 3.4) e seu algorítmo de decisão de migração apresentado mais adiante na Tabela 4.1. Para a infra-estrutura podemos considerar um sistema de agentes implementado a partir de CORBA ou a adaptação de um sistema de agentes aos mecanismos de CORBA. A especificação do OMG/MASIF permite as duas abordagem conforme apresentado adiante. A implementação final é em *Java,* sendo independente de sistema operacional, mas o ambiente de desenvolvimento é *Solaris* (*SunOS 5.x*).

Na Figura 4.1 apresentamos uma agência genérica (Seções 3.1 e 3.3). Representamos o sistema operacional do hospedeiro e o sistema de agentes. Basicamente o sistema de agentes (Seção 3.3) consiste de uma interface de comunicação, de diferentes agrupamentos de agentes denominados locais, e facilidades das quais ressaltamos o Serviço de Disponibilidade (Seção 3.4) e um Gerente de Ativação. Adiante apresentaremos duas implementações básicas diferentes. Uma considera a infra-estrutura CORBA disponível em *OrbixWeb3.x* [OrbixWeb97-1,

OrbixWeb97-2] na qual dispomos de um Gerente de Ativação para objetos CORBA com uma interface IDL proprietária, ver *IT_Daemon* no Apêndice D. A outra considera a infra-estrutura disponibilizada pelo sistema de agentes, *Grasshopper1.2* [GrassHopper98], no qual dispomos de um Gerente de Ativação para agentes e a integração com CORBA através das interfaces IDL do MASIF, ver *MAFFinder* e *MAFAgentSystem* no Apêndice B.

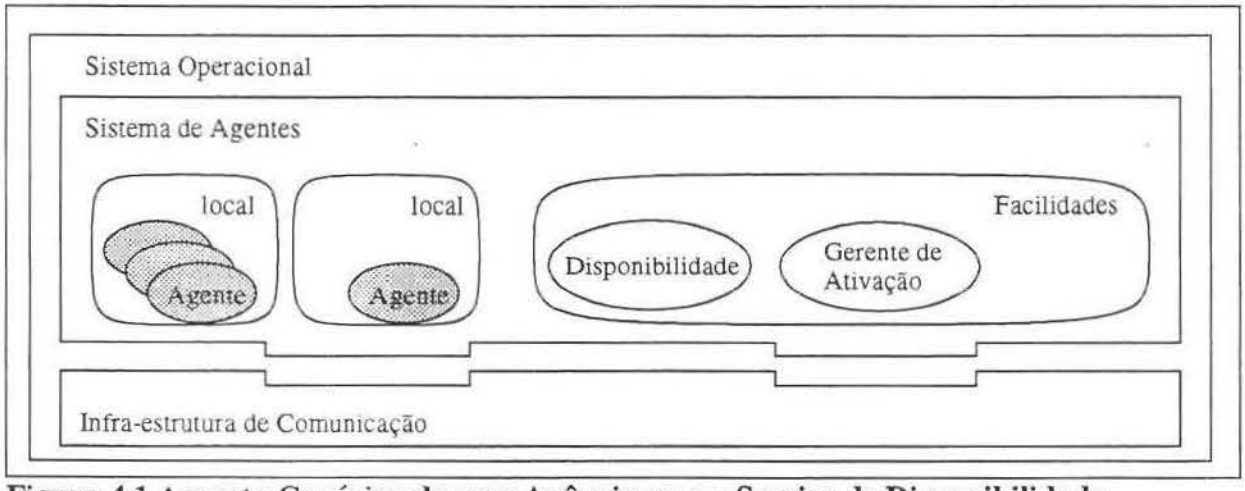


Figura 4.1 Aspecto Genérico de uma Agência com o Serviço de Disponibilidade.

A especificação OMG/MASIF [OMG97-2] evoluiu de uma *Facilidade de Agentes Móveis* (MAF) para uma *Facilidade de Interoperabilidade entre Sistemas de Agentes Móveis* (MASIF) de diferentes fabricantes, para a comunicação remota entre plataformas de componentes distribuídos. Detalha o uso de serviços CORBA especificados em ambientes orientados a agentes [Schulze97-1]. O padrão não se propõe a abordar todas as funcionalidades requeridas para o desenvolvimento de um sistema de agentes, mas somente aspectos que devem ser comuns às plataformas de agentes.

4.1.1. Facilidade de Interoperabilidade entre Sistemas de Agentes

Na Figura 4.2 temos uma representação genérica do sistema de agentes proposto na especificação do MASIF. Temos o sistema de agentes com uma interface de comunicação e diferentes agrupamentos (*locais*) de agentes. Na Figura 4.3 é apresentado o conceito de Região do MASIF sendo basicamente um domínio de agências, conforme visto na Seção 3.1. Para cada Região é proposto um serviço de nomes denominado *MAFFinder*. Na Figura 4.4 é apresentado o relacionamento do MAF com um ORB. A implementação MAF passa a ser disponibilizada e localizada no ORB através da interface IDL do MAFAgentSystem e do MAFFinder. Estes devem ser registrados no ORB como servidores (CORBA).

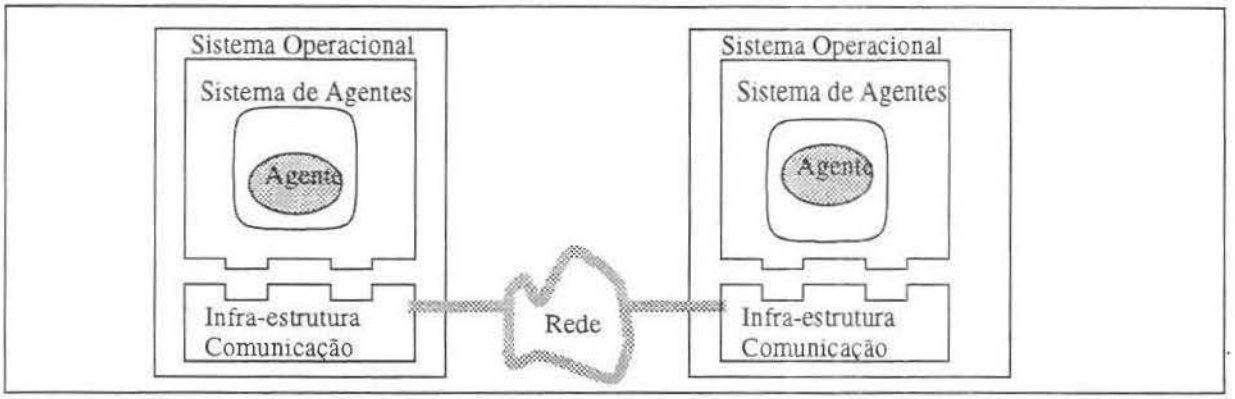


Figura 4.2 Estrutura de Sistemas de Agentes e sua interconexão via Rede.

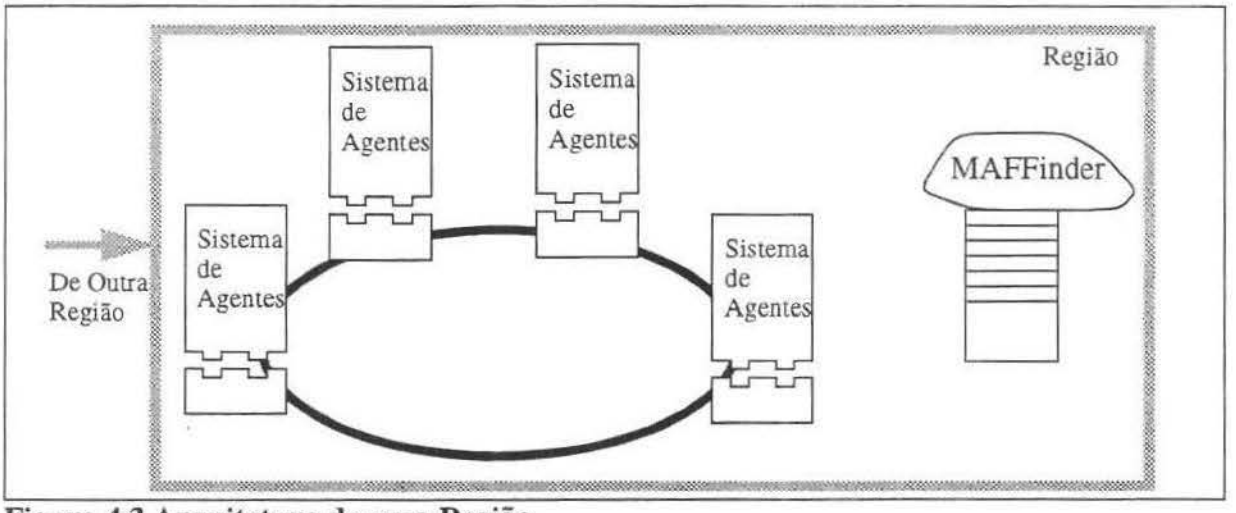


Figura 4.3 Arquitetura de uma Região.

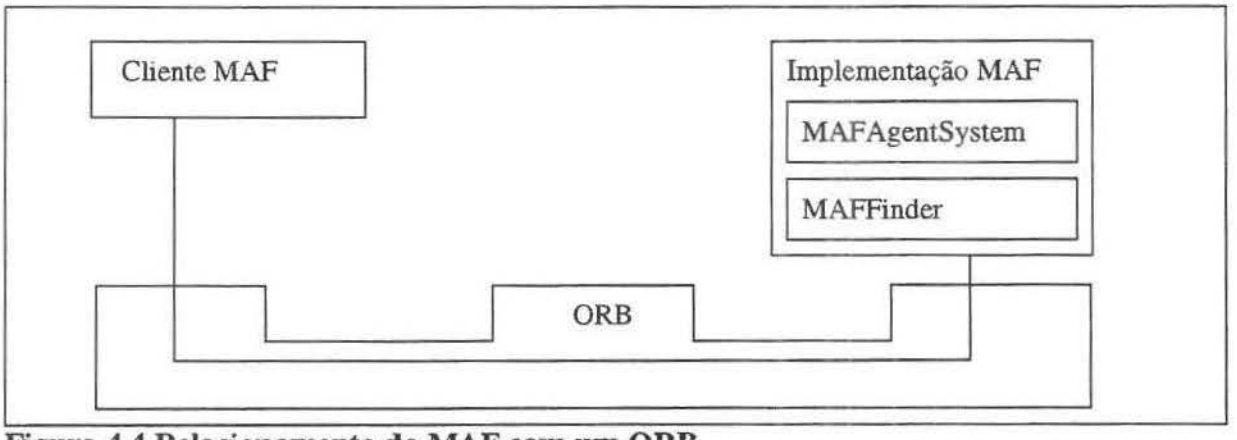


Figura 4.4 Relacionamento do MAF com um ORB.

Na Figura 4.5 representamos a integração do sistema de Agentes com os serviços e facilidades CORBA. A agência considerada pode ser endereçada através do serviço de Disponibilidade ou através de um Gerente de Ativação. O serviço de Disponibilidade implementa a interface de transparência e as chamadas aos serviços externos envolvidos. A implementação de um *serviço de Disponibilidade* envolve familiaridade com os mecanismos de um ORB, de um serviço

de Nomes, de um serviço de Catálogo, de um Suporte a Mobilidade e das funcionalidades *Java*.

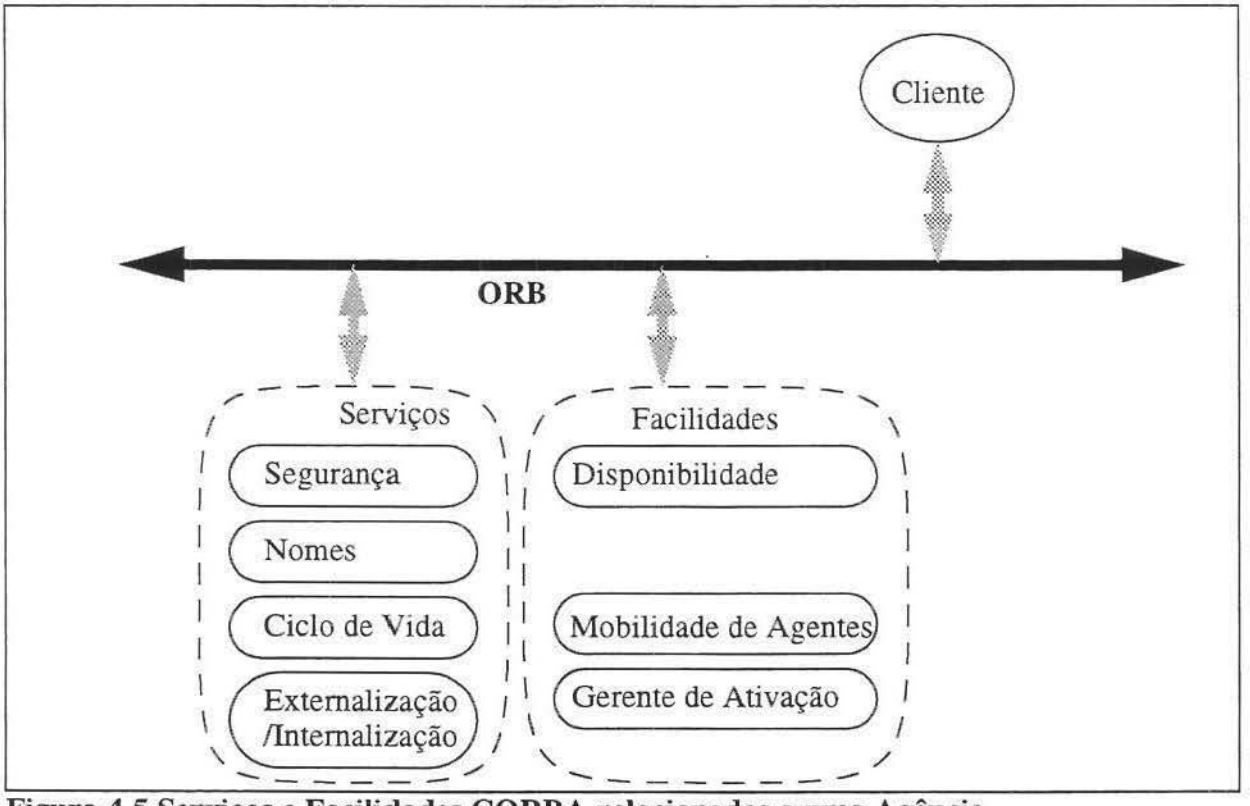


Figura 4.5 Serviços e Facilidades CORBA relacionados a uma Agência.

O Ciclo de Vida de um agente envolve e se relaciona com a Mobilidade e (Re)Ativação de agente, assim como a atribuição de Nomes. A Mobilidade demanda o serviço de Externalização (Internalização) para a (Des)Serialização do estado do agente. O serviço de Segurança também está relacionado a demanda de segurança na mobilidade. O serviço de Disponibilidade envolve o serviço de Nomes e de Ciclo de Vida.

O serviço de Externalização está relacionado à serialização do estado de um agente quando da sua migração pelo suporte de mobilidade. A mobilidade de agentes está por sua vez relacionado com o ciclo de vida de um agente móvel.

4.1.2. Conexão de um Agente Cliente a um Agente Servidor

A forma genérica é uma requisição ao serviço de Disponibilidade por um componente com suas propriedades. O serviço de Disponibilidade é responsável por encontrar um agente (ou componente) que satisfaça as propriedades requeridas. Se o agente requisitante é móvel, ele se moverá no sentido dos recursos requisitados. Para atingir os recursos desejados, é utilizado o método *bind* da interface de Transparência (Seção 3.4), com a sintaxe:

```
Referência_Componente = bind (nome_componente, propriedades, nome_operação, QoS);
Referência_Componente.nome_operação(argumentos_operação);
```

Um primeiro exemplo é um caso em gerenciamento onde um agente necessita de mais desempenho de cpu e comunicação. O agente cliente procura por uma agência com os recursos necessários e requisita migração. Neste caso, o cliente necessita ser móvel e realiza uma migração direta no sentido do componente alvo.

Um outro exemplo é o caso de um agente necessitar de informações de gerenciamento de um agente móvel de gerenciamento. O agente cliente procura pelo agente de gerenciamento do qual necessita e se move no sentido do agente requisitado, entretanto se este cliente não é móvel, ele não pode migrar e envia uma requisição remota. Quando o agente requisitado está pronto para responder, ele faz um re-chamada (callback) ao cliente requisitante e se move no sentido dele. Neste caso, há uma migração inversa com o agente alvo se movendo no sentido do cliente inicial.

Um desdobramento deste caso ainda gera um terceiro caso possível em que o agente de gerenciamento requisitado não pode se mover e neste caso o cliente inicial deve decidir pela execução remota da interação com o agente requisitado ou por fazer uma nova tentativa mais tarde.

Na Tabela 4.1 apresentamos uma chamada genérica com as estratégias adotadas em cada um dos casos possíveis:

- o agente iniciador do processo é móvel,
- o agente iniciador do processo não é móvel,
- o agente requisitado faz um callback sendo possível um dos casos anteriores.

Tabela 4.1 Estratégia geral na migração transparente.

Sinopse: Any agent_bind(nome, propriedades, qualquer)
Descrição: Move Ag1 na direção de Ag2 e retorna a sua IOR para requisições subseqüentes.
Ag1 requisita os métodos de Ag2 através de *stubs* e esqueletos, estáticos e/ou dinâmicos.

Caso 1: Ag1 é móvel {
1. Ag1 requisita Ag2 ao serviço de disponibilidade local (Av1)
2. Av1 localiza Ag2 em Av2
3. Av1 bloqueia Ag1
4. Av1 armazena Ag1
5. Av1 envia Ag1 para Av2
6. Av1 desregistra Ag1
7. Av2 registra Ag1
8. Av2 retorna para Av1 o IOR de Ag2 }

Caso 2: Ag1 não é móvel {
1. Ag1 requisita Ag2 ao serviço de disponibilidade local (Av1)
2. Av1 localiza Ag2 em Av2
3. Exceção quando Av1 tenta bloquear/armazenar/enviar Ag1
4. Av2 retorna a Av1 a IOR de Ag2 }

Caso 3: Numa rechamada de Ag2 para Ag1, um dos casos anteriores descritos para Ag1 se aplica agora a Ag2. {
1. caso 1 ou caso 2 }

Resumindo, o processo de migração executa:

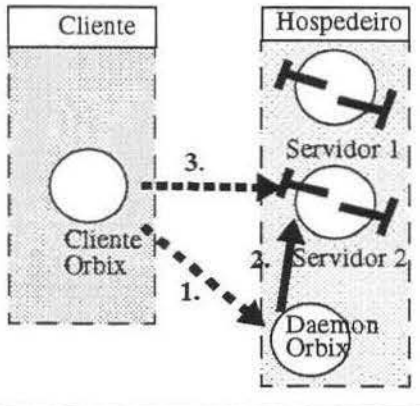
- uma migração direta do agente iniciador,
- uma migração inversa ou
- uma execução remota.

4.2 Implementação a partir de CORBA

Nesta seção aprofundamos uma implementação a partir de CORBA, enquanto na seção seguinte apresentamos um estudo comparativo de uma implementação a partir de uma plataforma de agentes.

Uma implementação inicial simplificada foi realizada para os testes utilizando serviços disponíveis na Orbix C++ e OrbixWeb para Java os quais, entretanto, estão padronizados pelo OMG. Dentre estes serviços destacamos o gerente de ativação de servidores (*orbixd*) e a classe *locator* associada ao método *bind* de obtenção de referência de objeto remoto através do ORB.

O ORB consiste de um biblioteca de funções de cliente e de servidor junto com um executável. Esta abordagem pretende um melhor *throughput* e distribuição da aplicação. A biblioteca de servidor engloba o envio e o recebimento de requisições de operações de objetos remotos, enquanto a biblioteca de cliente permite somente enviar tais requisições. Um *daemon* é responsável por ativar processos de servidores dinamicamente conforme requisitado, e de acordo com as políticas de ativação descritas na especificação CORBA. Aplicações Cliente / Servidor no mesmo espaço de endereçamento podem ser construídas utilizando a biblioteca de servidor somente. A Figura 4.6 mostra como o gerente de ativação (*Orbix daemon*) estabelece as conexões Cliente / Servidor.



1. O cliente contacta o *daemon*, explicitamente definido ou identificado via *Localizador* (*Locator*) ou *Servidor de Nomes*.
2. *Daemon* abre conexão com o servidor.
3. A referência do servidor é passada ao cliente e o *daemon* não atua mais.

- *Daemon* aguarda novas requisições, atuando como gerente de ativação.

Figura 4.6 Conexão Cliente / Servidor com gerente de ativação (*orbixd*).

O Gerente de Ativação permite a invocação remota de objetos não ativados e gerenciar a sua (re)ativação. Este gerente possui uma porta predefinida de acesso pela *Internet* e é o componente fundamental de uma agência na migração e ciclo de vida dos agentes. Este gerente tam-

bém possui uma interface IDL *IT_daemon* com funções e atributos que esboçam as funcionalidades oferecidas no *MAFAgentSystem* no MASIF [OMG97-2] e do *ServantActivator* no POA [OMG97-3, OMG98-2, OMG98-5].

A classe *Locator* por sua vez é o componente fundamental na recuperação de referências de objetos no protocolo Orbix (i.e., não IIOP), ver Figura 4.7. Possui uma implementação *default* a qual utiliza um arquivo de dados (*orbixhome/config/orbix.hosts*) como um serviço de Nomes simplificado (ou *cache*) contendo entradas do tipo:

- nome do serviço e hospedeiro de localização, ou
- nome do serviço e lista de hospedeiros de localização, ou
- nome do serviço e lista de grupos de hospedeiros de localização.

Este arquivo pode receber entradas pelo terminal ou através de comandos do cliente ao *orbixd* pela interface *IT_daemon*. Incluímos nesta lista uma entrada de nome do *IT_daemon* com um (ou mais) hospedeiro(s) *default* para utilização em caso de falha.

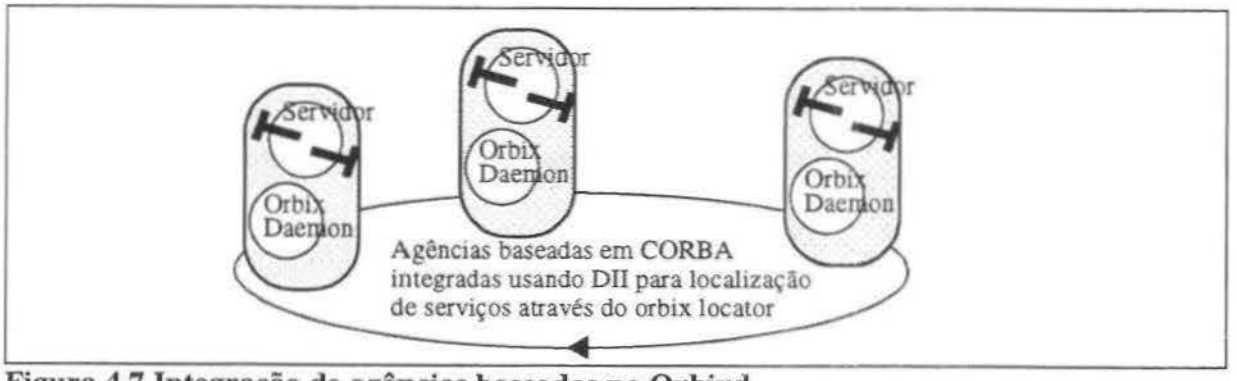


Figura 4.7 Integração de agências baseadas no Orbixd.

A funcionalidade do *Locator* é equivalente de certa forma a oferecida por *MAFFinder* no MASIF e *ServantLocator* no POA. Entretanto a implementação *default* não utiliza referências de objetos interoperáveis (*IORs*) nem a localização de componentes remotos utilizando propriedades e/ou propriedades com valores. Entretanto a classe *Locator* permite uma implementação em substituição a *default* a ser implementada pelo desenvolvedor.

Ressaltamos que a funcionalidade da classe Locator está presente no *daemon orbixd* disponível para as versões Orbix e OrbixWeb. Na implementação java deste *daemon* (*orbixdj*) o *locator* só está presente a partir da versão OrbixWeb3.1.

A possibilidade de utilização de um *Locator* personalizado permite utilizá-la na implementação do Serviço de Disponibilidade, no que se refere a localização de componentes utilizando o serviço de Nomes (*OMG/Naming*) para componentes com propriedades e um serviço de Catálogo como o *OMG/Trader* para componentes com propriedades com valores. Desta forma componentes podem ser localizados, escolhidos e requisitados através de IORs.

Para a atualização de referências pode-se utilizar diretamente chamadas ao serviço de Nomes e serviço de Catálogo ou ao serviço de Disponibilidade local que está registrado com o gerente de ativação (orbixd). Na Figura 4.8 temos os serviços de disponibilidade e mobilidade registrados com o gerente de ativação (*orbxid*).

Para o protocolo Orbix, no hospedeiro(n-1) estes serviços estão registrados como servidores CORBA e incluídos na lista utilizada pelo *Locator* do hospedeiro. A entrada na lista é da forma:

- *SD(n-1) : hospedeiro(n-1)*, para localização do serviço de disponibilidade da agência(n-1)
- *SM(n-1) : hospedeiro(n-1)*, para localização do serviço de mobilidade da agência(n-1)
- *IT(n-1) : hospedeiro(n-1)*, - para localização do *IT_daemon* da agência(n-1)

Estes três serviços compõem uma agência no hospedeiro(n-1). As referências de objeto dos serviços SD(n-1) e SM(n-1) são obtidas através da intermediação do orbixd (IT_daemon) conforme descrito no Apêndice A. O daemon é contactado transparentemente pela função *bind* da classe <Tipo>Helper gerada pelo compilador IDL, onde <Tipo> é o nome de um tipo definido em IDL pelo desenvolvedor.

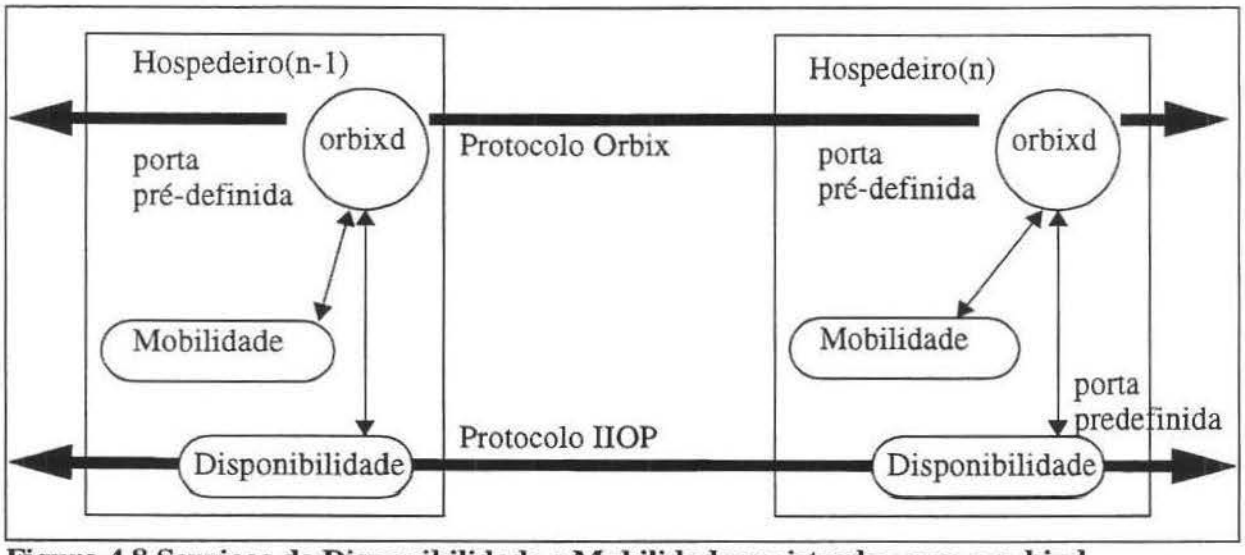


Figura 4.8 Serviços de Disponibilidade e Mobilidade registrados com o orbixd.

Para o protocolo IIOP, o serviço de disponibilidade passa a ser a porta predefinida de entrada. Para tanto:

- o serviço de disponibilidade é ativado como servidor persistente no hospedeiro
- uma IOR é gerada no instante da ativação
- o nome SD(n-1) é associado a IOR no serviço de Nomes e/ou Trader do domínio ou região.

As interfaces do serviço de Disponibilidade desempenham papel semelhante a *MAFAgentSystem* e *MAFFinder* do MASIF e a interface *IT_daemon* do orbixd. Através do serviço de Dispo-

nibilidade passa a ser possível contactar os demais servidores. A ativação de servidores é repassada ao orbixd e a mobilidade repassada ao serviço de Mobilidade.

A classe *locator* e o arquivo associado orbix.host (e orbix.hostgroups) correspondem respectivamente a uma simplificação da classe Seleção, *Cache*, Nomes e Trader apresentadas na Seção 3.4. Estas estão detalhadas adiante na Seção 4.5, juntamente com as demais (Transparência, Disponibilidade, Mobilidade).

Um caso representativo de mobilidade implícita é apresentado na Seção 5.2 (Prototipagem na Migração de Componentes) enquanto um caso representativo de migração explícita é apresentado no Capítulo 6, associado a um mecanismo de ajuste baseado na migração implícita dos serviços gerenciados (agentes móveis ou objetos móveis CORBA).

4.3 Implementação a partir de uma Plataforma de Agentes

Nesta seção apresentamos uma comparação da implementação anterior a partir de CORBA com uma a partir de uma plataforma de agentes.

Como na implementação anterior, duas formas são possíveis numa implementação a partir de uma plataforma de agentes móveis como GrassHopper (GH), como representado na Figura 4.9, onde o serviço de disponibilidade incorpora o sistema de agentes ou é incorporado ao sistema de agentes.

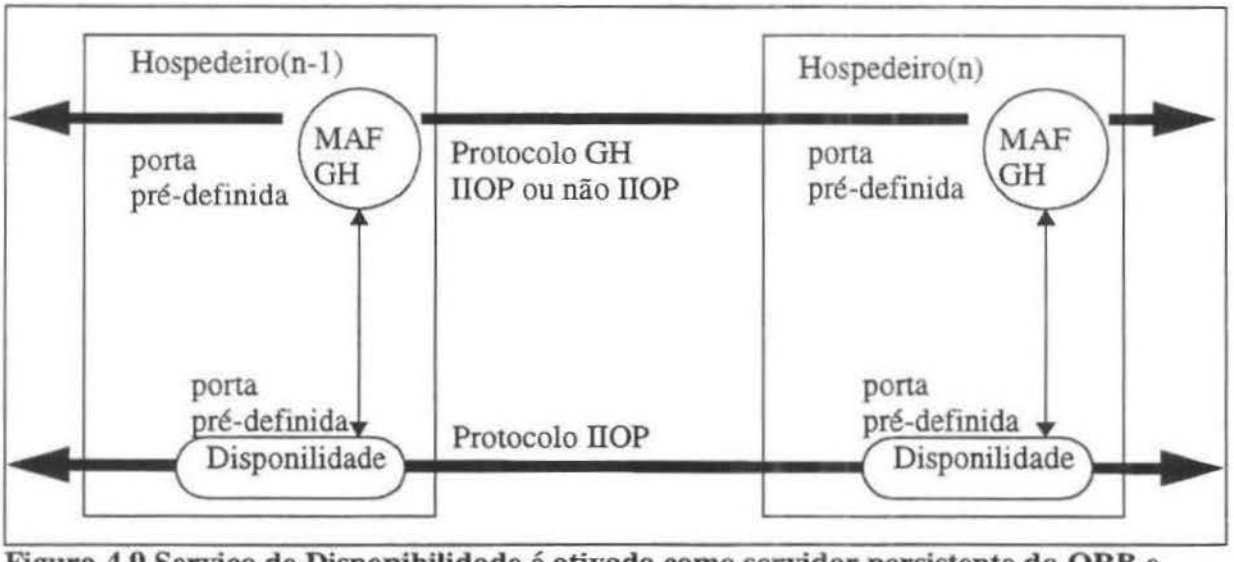


Figura 4.9 Serviço de Disponibilidade é ativado como servidor persistente do ORB e agente estático da plataforma de agentes.

Primeiro descreveremos a implementação onde o serviço de Disponibilidade incorpora o GH utilizando seu suporte de mobilidade de agentes e gerente de ativação. Neste caso podemos utilizar uma versão de CORBA sem um gerente de ativação e suporte de mobilidade. O gerente de ativação começa a ser padronizado na especificação OMG/POA [OMG98-5] mas ainda não

está padronizado completamente. Para mobilidade está especificado o OMG/MASIF havendo outras especificações em andamento como *Object by Value* [OMG98-4] e de portabilidade como o POA. Para armazenamento de estado utiliza-se a serialização, em Java.

Nesta implementação o serviço de disponibilidade é a porta predefinida de entrada. Para tanto:

- o serviço de disponibilidade é ativado como servidor persistente no hospedeiro
- uma IOR é gerada no instante da ativação
- o nome SD(n-1) é associado a IOR no serviço de Nomes e/ou Trader do domínio ou região.

As interfaces do serviço de disponibilidade utilizam as funcionalidades disponibilizadas nas interfaces *MAFAgentSystem* e *MAFFinder*. Através do serviço de disponibilidade são contactados os demais agentes. A mobilidade e ativação de agentes é repassada ao GH.

Na outra implementação utilizamos o(s) protocolo(s) e a infra-estrutura do GH enquanto o serviço de disponibilidade passa a ser um agente estático da plataforma para a migração transparente de agentes. Neste caso devem ser consideradas particularidades internas da plataforma que não necessariamente seguem um padrão OMG/CORBA.

4.4 Comentários

Uma padronização maior de OMG/CORBA no que se refere a portabilidade de objetos deve facilitar a implementação de mobilidade em CORBA conciliando melhor objetos e agentes, sejam móveis ou não móveis.

A especificação do POA pretende corrigir definições vagas existentes no BOA e que levou a diferentes implementações. No POA especifica-se a estrutura de adaptadores de objetos a ser padronizada por diferentes implementadores na utilização de interfaces dinâmicas (DII e DSI) e conseqüentemente garantir a portabilidade de objetos entre eles.

O POA procura também padronizar gerentes de ativação e localização que participam na recuperação de referências de objetos e são parte do ciclo de vida de objetos distribuídos e móveis.

Algumas implementações de CORBA começam a disponibilizar as padronizadas org.omg, entre estas implementações encontram-se JavaIDL [Java98-3], OrbixWeb3.x e Visibroker for Java 3.x. O pacote de classes org.omg inclui:

- org.omg.CORBA
- org.omg.CORBA.DynAnyPackage
- org.omg.CORBA.ORBPackage
- org.omg.CORBA.portable
- org.omg.CORBA.TypeCodePackage
- org.omg.CosNaming

- org.omg.CosNaming.NamingContextPackage

A versão 3.1 da OrbixWeb dispõe de um Gerente de Ativação totalmente em Java (orbixdj) com as mesmas funcionalidades do executável não Java (orbixd). A versão Java disponibiliza uma funcionalidade adicional de ativação de objetos servidores (*servants*) dentro de um mesmo processo (*cluster* de objetos) utilizando reflexão computacional disponível em Java. Esta funcionalidade facilita a migração de objetos e agentes para dentro de um mesmo processo, onde o processo passa a ter a mesma funcionalidade de *lugar* (*place*) da especificação do MASIF.

A obtenção de uma referência de objeto transparentemente através do ORB ainda não está padronizada sendo um fator limitante para suporte a migração nas implementações atuais. A especificação do POA permite a localização de objetos com propriedades.

4.4.1. Especificações CORBA IDL Relacionadas

Este trabalho encontra correspondência em especificações IDL de padrões OMG como *ORB* [OMG97-3], *POA* [OMG97-3, OMG98-1, OMG98-5] e *CfMAF* [OMG97-2]. A Tabela 4.1 apresenta um conjunto reduzido das especificações correspondentes a este trabalho. As especificações são extensas e por isto sugere-se a consulta aos documentos originais e fazemos uma apresentação de forma resumida nos parágrafos que se seguem.

A interface *ORB* do módulo *CORBA* apresenta métodos relativos à inicialização do ORB { *ORB_init*() | *list_initial_services*() | *resolve_initial_references*() } e métodos para o tratamento e recuperação de referências de objetos { *object_to_string*() | *string_to_object*() } pelo lado cliente numa conexão. A IDL da especificação completa do módulo *CORBA* pode ser encontrada em [CORBA98, Java98-3].

O módulo *PortableServer* especifica a portabilidade de objetos entre implementações CORBA, sendo importante na padronização de agentes (componentes) migratórios. Está organizado nas interfaces: *POA* e *ServantManager* { *ServantActivator* e *ServantLocator* }. Na Tabela 4.2, resumimos as especificações das interfaces ORB e POA, pertinentes ao trabalho.

Tabela 4.2 Especificações IDL Mínimas: ORB e PortableServer.

ORB	**PortableServer**	

object_to_string()
string_to_object()
list_initial_services()
resolve_initial_references()
ORB_init()
	POA{
	create_POA()
	destroy()
	activate_object()
	deactivate_object()
	servant_to_reference()
	reference_to_servant()
	}
	ServantManager{
	}
		ServantActivator:ServantMana
		ger{
		incarnate ()
		etherealize()
		}
		ServantLocator:ServantManag
		er{
		preinvoke()
		postinvoke()
		}

A interface POA apresenta métodos relacionados ao ciclo de vida de um objeto portável { *create_POA*() | *destroy*() | *activate_object*() | *deactivate_object*() } e ao tratamento de referências de objetos pelo lado servidor { *servant_to_reference*() | *reference_to_servant*() }. A interface *ServantActivator:ServantManager* apresenta o método de ativação e desativação de objetos respectivamente { *incarnate*() | *etherialize*() }, enquanto a interface *ServantLocator:ServantManager* apresenta métodos para localização e ativação de objetos respectivamente { *preinvoke*() | *postinvoke*() }. Os diagramas UML e a IDL para a especificação completa do módulo *PortableServer* podem ser encontrados no Apêndice C. Na especificação, um *Servant ou servente* é um objeto servidor dentro de um processo servidor, sendo um *ServantManager* o gerente deste servente.

O módulo *CfMAF* especifica a interoperabilidade entre sistemas de agentes e apresenta as interfaces *MAFAgentSystem* e *MAFFinder*. Na Tabela 4.3, resumimos as especificações das interfaces do CfMAF, pertinentes ao trabalho. A IDL da especificação completa do módulo *CfMAF* pode ser encontrada no Apêndice B.

Tabela 4.3 Especificação IDL Mínima do CfMAF.

CfMAF	
AgentSystemInfo{	*MAFAgentSystem*{
agent_system_name;	create_agent()
agent_system_type;	fetch_class()
language_maps;	get_agent_system_info()
agent_system_description;	list_all_agents()
major_version;	receive_agent()
minor_version;	resume_agent()
properties;	suspend_agent()
};	terminate_agent()
	}
AgentProfile{	
language_id;	*MAFFinder*{
agent_system_type;	register_agent()
agent_system_description;	register_agent_system()
major_version;	lookup_agent()
minor_version;	lookup_agent_system()
serialization;	unregister_agent()
properties;	unregister_agent_system()
};	}

A interface *MAFAgentSystem* especifica métodos relativos ao ciclo de vida de agentes, ressaltando-se {*create_agent*() | *fetch_class*() | *get_agent_system_info*() | *list_all_agents*() | *receive_agent*() | *resume_agent*() | *suspend_agent*() | *terminate_agent*() }. A interface *MAFFinder* especifica métodos relativos a registro, desregistro e localização de agentes e agências, utilizando as propriedades em *AgentProfile* e *AgentSystemInfo* respectivamente, ressaltando-se {*register_agent*() | *register_agent_system*() | *lookup_agent*() | *lookup_agent_system*() | *unregister_agent*() | *unregister_agent_system*() }.

4.5 Definição de Classes

A seguir temos a definição de classes de acordo com o discutido no Capítulo 3. As classes Transparência, Disponibilidade e Seleção são discutidas na Seção 3.4. A classe Cache é uma classe interna do serviço de Disponibilidade, sendo apresentado como uma *Hashtable*. A cache pode ser também baseada num Objeto de Dados Semânticos *(SDO)* para dados importados por um agente móvel de diferentes serviços de Catálogo [Rodríguez99]. Um *SDO* [CommerceNet97] é uma estrutura de dados em árvore com métodos de busca (e armazenamento) onde os nós são propriedades com valores. A classe Mobilidade é discutida na Seção 3.5. As classes Nomes e Trader são discutidos na Seção 3.6.

4.5.1. Transparência

É a interface disponível ao usuário para chamadas com migração transparente.

Transparência.bind() Usado para resolver o nome de um objeto, agente ou agência.

Sinopse	Object *bind* (in String *nome*, in NVList *propriedades*, *in* Any *qq*)
Parâmetros	*nome* define um identificador para o agente desejado. *propriedades* é uma lista de propriedades com valores utilizada para qualificar os recursos a serem atingidos. Por exemplo, as propriedades apresentadas na Seções 4.1.2 e 3.4 (QoS, operação, tipo, custo, MIPS e tempo_execução). A lista também pode ser vazia. *qq* é um parâmetro que pode ser usado isoladamente para localizar um componente, podendo ser nulo.
Nota	A classe pode ser estendida para implementar um comportamento de método atendendo uma particularidade do usuário/aplicação.
Ver	Disponibilidade.bind()

Transparência.object_to_string() Transforma uma referência de objeto interoperável em uma string.

Sinopse	String *object_to_string*(Object *objref*)
Parâmetros	*objref* é a referência interoperável de objeto a ser convertida em string
Nota	Utiliza org.omg.CORBA.ORB.object_to_string().
Ver	IOR, string_to_object, interface ORB

Transparência.string_to_object() Converte de volta a referência de objeto.

Sinopse	Object *string_to_object*(String *str*)
Parâmetros	*str* é a referência interoperável de objeto em string
Nota	Utiliza org.omg.CORBA.ORB.string_to_object().
Ver	IOR, object_to_string, interface ORB

4.5.2. Disponibilidade

A partir do descrito anteriormente e das especificações apresentadas na Tabela 4.2, o Serviço de Disponibilidade pode ser implementado a partir de CORBA e das IDLs *ORB* e *PortableServer* e disponibilizar uma interface *CfMAF* para compatibilidade com outros sistemas de agentes ou implementado a partir de um Sistema de Agentes que disponibilize a interface *CfMAF* e interação com CORBA.

Disponibilidade.busca() Localiza os recursos a serem atingidos e retorna um conjunto de localizações.

Sinopse	stringSeq *busca*(in NVList *parâmetros*, out NVList *prop*)
Parâmetros	*parâmetros* são os recursos a serem atingidos. É uma lista de propriedades com valores utilizada para qualificar os recursos a serem atingidos. Por exemplo, as propriedades apresentadas na Seções 4.1.2 e 3.4 (QoS, operação, tipo, custo, MIPS e tempo_execução) *prop* é uma estrutura alternativa de retorno de dados contendo propriedades com valores obtidas a partir de um Trader (ou Catálogo), para iteração local.
Estratégia	1. busca na cache 2. busca nos serviços de Nomes 3. busca nos serviços de Trader (ou Catálogo)
Nota	São serviços de diretório: Nomes, Trader e/ou outro Catálogo.
Ver	Objetos de Dados Semânticos [Rodríguez99].

Disponibilidade.bind() Atinge os recursos solicitados, obtém uma referência de objeto e associa o nome à referência de objeto retornada.

Sinopse	Object *bind* (in String *nome*, in NVList *parâmetros*, *in* Any *qq*)
Parâmetros	*nome* é um parâmetro que pode ser usado isoladamente para localizar um componente *parâmetros* é uma lista de propriedades com valores utilizada para qualificar os recursos a serem atingidos. Por exemplo, as propriedades apresentadas na Seções 4.1.2 e 3.4 (QoS, operação, tipo, custo, MIPS e tempo_execução). A lista pode ser vazia. *qq* é um parâmetro que pode ser usado isoladamente para localizar um componente, podendo ser nulo *qq* = *quantidade* representa o número de elementos a selecionar *qq* = *hospedeiro* representa a agência destinatária a contactar *qq* = *tipo* representa uma classe (tipo) de aplicação de migração
Estratégia	1. busca 2. seleciona 3. atinge uma instância ativa 4. retorna referência de objeto
Nota	Implementa transparência.bind() e utiliza Disponibilidade.busca(). A classe pode ser estendida para implementar um comportamento do método atendendo uma particularidade do usuário/aplicação.
Ver	classe Disponibilidade

Disponibilidade.desabilita_ajuste() Desabilita a emigração de agente(s) de (para) uma agência.

Sinopse	void *desabilita_ajuste*(Boolean *In*, Boolean *Out*)
Parâmetros	*In* parâmetro que desabilita a imigração *Out* parâmetro que desabilita a emigração
Nota	Atua sobre todos os agentes em uma dada agência
Sinopse	void *desabilita_ajuste*(Object *objref*, Boolean *In*, Boolean *Out*)
Parâmetros	*objref* é a referência de um objeto específico a ser bloqueada a migração *In* parâmetro que desabilita a imigração *Out* parâmetro que desabilita a emigração
Nota	Atua seletivamente sobre um agente em uma dada agência

4.5.3. Seleção

Seleção.escolhe( ) Escolhe um subconjunto de localizações de um conjunto maior e retorna o novo subconjunto de localizações.

Sinopse	stringSeq *escolhe*()
Parâmetros	nenhum
Nota	Neste caso o resultado é apenas um hospedeiro. Utilizado na localização de uma agência (hospedeiro) específico, como por exemplo para uma migração explícita.
Sinopse	stringSeq *escolhe*(in stringSeq *destinatários*, out string *tipo*)
Parâmetros	*destinatários* é uma lista de localizações resultante de uma busca/escolha anterior *tipo* pode ser IOR ou hospedeiro.
Nota	Uma IOR inclui, não necessariamente nesta ordem: hospedeiro:porta:referência_objeto_interna. Utilizado para selecionar um conjunto, por exemplo, de agências para compor um domínio, ou de componentes (agências ou agentes) para uma seleção aleatória subseqüente.
Ver	Referência de Objetos [Voyager97-1, Java98-3].
Sinopse	stringSeq *escolhe*(in NVList *Sdo*)
Parâmetros	*Sdo* representa uma estrutura de dados obtidos a partir de uma busca em serviços de diretórios
Ver	Objetos de Dados Semânticos [Rodríguez99].
Sinopse	stringSeq *escolhe*(in int *quantidade*)
Parâmetros	*quantidade* representa o número de elementos no conjunto a retornar
Sinopse	stringSeq *escolhe*(in int *quantidade*, in boolean *aleatório*)
Parâmetros	*aleatório*=true define uma escolha aleatória ao invés de sequencial, =false preserva um escolha sequencial *quantidade* representa o número de elementos a selecionar

4.5.4. Cache

A cache é utilizada localmente para armazenamento e recuperação de destinatários que foram utilizados mais recentemente, sendo a informação mantida durante um período predeterminado de tempo. É baseada em *java.util.Hashtable*.

Cache.clear() Limpa a *hashtable* de forma a não conter qualquer chave.

Sinopse void clear()

Cache.remove() Remove a chave (e seu valor correspondente) desta *hashtable*, retornando a nova *hashtable*.

Sinopse Object remove(Object *chave*)

Cache.size() Retorna número de chaves desta *hashtable*.

Sinopse int size()

Cache.put() Mapeia a chave especificada para o valor especificado nesta *hashtable*, retornando a nova *hashtable*.

Sinopse Object put(Object *chave*, Object *valor*)

Cache.keys() Retorna uma enumeração das chaves nesta *hashtable*.

Sinopse Enumeration keys()

Cache.get() Retorna o valor para o qual a chave especifcada é mapeada nesta *hashtable*.

Sinopse Object get(Object *chave*)

Cache.contains() Testa se alguma chave mapeia para o valor especificado nesta *hashtable*.

Sinopse boolean contains(Object *valor*)

Cache.containsKey() Testa se o objeto especificado é uma chave nesta *hashtable*.

Sinopse boolean containsKey(Object *chave*)

4.5.5. Mobilidade

Esta classe implementa a interface do serviço de disponibilidade com a infra-estrutura que implementa a mobilidade de agentes. Dispomos de uma implementação própria [Vasconcellos98] implementada sobre CORBA para OrbixWeb(2x). Entretanto podemos utilizar outra implementação de mobilidade de agentes [Aglets, Odyssey97, GrassHopper98, Voyager97-1]. Em cada uma das extremidade, isto é, hospedeiro remetente e destinatário, deve estar disponível uma agência.

Atributos

Mobilidade.itinerário representa a nova localização de um agente, podendo ser uma agência ou domínio de agências

Sinopse	Vector *itinerário*

Mobilidade.razão indica o tipo da estratégia de migração, isto é, explícita ou implícita.

Sinopse	Boolean *razão*
Nota	para a migração implícita, o domínio (itinerário) é seguido aleatoriamente

Métodos

Mobilidade.registra() registra a referência de agente em cada agência do domínio

Sinopse	void *registra*(in String *nome*, in NVList *propriedades_classe*)
Parâmetros	*propriedades_classe* é o conjunto de informações necessárias para a (re)ativação de um agente
Ver	*desregistra*()

Mobilidade.desregistra() desregistra a referência de agente em cada agência do domínio

Sinopse	void *desregistra*(in String *nome*)
Parâmetros	*nome* é o identificador do agente
Ver	*registra*()

Mobilidade.MS.bloqueia() bloqueia o agente para novas requisições

Sinopse	void *bloqueia*(Object *objref*)
Parâmetros	*objref* é a referência de objeto do agente a ser bloqueado

Mobilidade.MS.armazena() *armazena* o estado

Sinopse	void *armazena*(Object *objref*)
Parâmetros	*objref* é a referência de objeto do agente a ter os estado armazenado

Mobilidade.MS.recebe() *recebe* o agente: ativa e recupera o estado

Sinopse	void recebe(Object *objref*)
Parâmetros	*objref* é a referência de objeto do agente

4.5.6. Nomes

Esta implementação utiliza o pacote *org.omg.Naming* presente em Java2 (JDK-1.2) [Java98-3] e OrbixWeb3.x [OrbixWeb97-1, OrbixWeb97-2]. O Serviço de Nomes fornece uma árvore de diretórios para referências de objetos semelhante a uma estrutura de diretório de sistemas de arquivos, ver Apêndice A.

As referências de objeto são armazenadas no espaço de nomes e cada par *referência-nome de objeto* é denominada uma associação (*binding*) conhecida. As associações conhecidas podem ser organizados em contextos de nomes. Contextos de nomes são também associações e tem a mesma função organizacional de um subdiretório de arquivos. Todas as associações são armazenadas sob o contexto inicial de Nomes.

Nomes.bind_context() Nomeia um objeto que é um contexto de nomes.

Sinopse	void bind_context(NameComponent[] *n*, NamingContext *nc*)
Parâmetros	*n* nome do componente *nc* um contexto de nomes
Ver	interface org.omg.NamingContext

Nomes.bind_new_context() Cria um novo contexto e o associa ao nome fornecido como argumento.

Sinopse	NamingContext bind_new_context(NameComponent[] *n*)
Parâmetros	*n* nome do componente
Ver	interface org.omg.NamingContext

Nomes.bind() Cria uma associação de um nome e um objeto no contexto de nomes.

Sinopse	void bind(NameComponent[] *n*, Object *obj*)
Parâmetros	*n* nome do componente *obj* referência de objeto
Ver	interface org.omg.NamingContext

Nomes.destroy() Remove o contexto de nomes corrente.

Sinopse	void destroy()
Parâmetros	*nenhum*
Ver	interface org.omg.NamingContext

Nomes.list() Permite um cliente iterar através de um conjunto de associações em um contexto de nomes.

Sinopse	void list(int *how_many*, BindingListHolder *bl*, BindingIteratorHolder *bi*)
Parâmetros	*how_many* quanto elementos buscar *bl* lista de elementos *bi* lista de retorno de iterações subseqüentes
Ver	interface org.omg.NamingContext e org.omg.BindingIterator

Nomes.new_context() Retorna um contexto de nomes implementado pelo mesmo servidor de nomes que o contexto sobre o qual a operação foi invocada.

Sinopse	NamingContext new_context()
Parâmetros	*nenhum*
Ver	interface org.omg.NamingContext

Nomes.rebind_context() Cria uma associação de um nome e um contexto de nomes no mesmo contexto de nomes mesmo que o nome já esteja associado no contexto.

Sinopse	void rebind_context(NameComponent[] *n*, NamingContext *nc*)
Parâmetros	*n* nome do componente *nc* um contexto de nomes
Ver	interface org.omg.NamingContext

Nomes.bind_new_context() Cria um novo contexto e o associa ao mesmo nome fornecido como argumento.

Sinopse	NamingContext bind_new_context(NameComponent[] *n*)
Parâmetros	*n* nome do componente
Ver	interface org.omg.NamingContext

Nomes.rebind() Cria uma associação de um nome e um objeto no contexto de nomes mesmo que o nome já esteja associado no contexto.

Sinopse	void rebind(NameComponent[] *n*, Object *obj*)
Parâmetros	*n* nome do componente *obj* referência de objeto
Ver	org.omg.NamingContext

Nomes.resolve() A operação de *resolve* é o procedimento de recuperação de um objeto associado a um nome em um dado contexto.

Sinopse	Object resolve(NameComponent[] *n*)
Parâmetros	*n* nome do componente
Ver	org.omg.NamingContext

Nomes.unbind() A operação de *unbind* remove a associação de um nome de um contexto.

Sinopse	void unbind(NameComponent[] *n*)
Parâmetros	*n* nome do componente
Ver	org.omg.NamingContext

Contextos de nomes são utilizados para inserir uma sub-árvore de propriedades abaixo de um dado nó. Isto é importante no caso de entradas de nomes com propriedades (sem valores) como representado na Figura 3.20, para agentes. A utilização de contexto de nomes está detalhada no Apêndice A (Seção A.3).

4.5.7. Trader

Definimos como catálogo os serviços de diretório que permitem consulta e armazenamento de componentes e de suas propriedades com valores. A implementação de um Trader permite deslocar algumas funcionalidades definidas nas especificações do OMG e integrá-las a outros serviços e ambientes. Como serviço de Catálogo há duas implementações possíveis de uso, uma considerando uma implementação [Rodríguez99] em Java e outra considerando um OMG/ Trader em C++ [TOI98]. Os métodos a seguir são implementações particularizadas dos métodos das interfaces do Trader. Os métodos originais das interfaces do Trader são métodos internos destas implementações.

Trader.ExportAg() Exporta um agente ou agência, com as suas propriedades com valores, e propriedades dinâmicas se houver (no caso de agência).

Sinopse	String ExportAg(in NVList *properties*, in String *type*, in int *stat_props*, in int *dyn_props*)
Parâmetros	*properties* lista de propriedades com valores *type* tipo (e.g. agente ou agência) *stat_props* quantidade de propriedas estáticas *dyn_props* quantidade de propriedas dinâmicas
Nota	retorna String *offerid* - identificação da entrada de oferta
Ver	OMG/Trader [OMG96]

Trader.ImportAg() Importa um agente ou agência, através de suas propriedades com valores.

Sinopse	CosTrading.Offer[] ImportAg(in String *type*, in CosTrading.LookupPackage.SpecifiedProps *desired_properties*, in String *constr*, in int *how_many*)
Parâmetros	*type* tipo (e.g. agente ou agência) *desired_properties* propriedade(s) desejada*(s)* *constr* restições quanto aos valores de propriedades *how_many* quantidade de ofertas desejadas
Nota	retorna CosTrading.Offer[] *offer* - lista de ofertas
Ver	OMG/Trader [OMG96, Bearman96, Trader95, Lima95]

Trader.ModifyAg() Atualiza as propriedades com valores de um agente ou agência.

Sinopse	void ModifyAg (in String *offerid*, in String *type*, in NVList *properties*)
Parâmetros	*offerid* identificação da entrada de oferta *type* tipo (e.g. agente ou agência) *properties* propriedade(s) desejada*(s)*
Ver	OMG/Trader [OMG96, Bearman96, Trader95, Lima95]

CAPÍTULO 5

Resultados

Nenhum vento sopra a
favor de quem não sabe
para onde ir.
Sêneca

Neste capítulo apresentamos resultados obtidos em diferentes fases do trabalho. Os resultados envolvem avaliação de alternativas consideradas nas diferentes fases, como suporte às escolhas a serem feitas. Duas aplicações de uso de migração com o suporte de um serviço de Disponibilidade foram analisadas: uma em computação móvel e outra em gerenciamento de serviços distribuídos. O capítulo está dividido nas seguintes seções:

- 5.1 Prototipagem na Passagem de Código, página 77,
- 5.2 Prototipagem na Migração de Componentes, página 79,
- 5.3 Comentários sobre a (Re)distribuição de Serviços, página 88 e
- 5.4 Prototipagem em Gerenciamento, página 89.

5.1 Prototipagem na Passagem de Código

Nesta seção apresentamos um teste de migração de código realizado sobre uma plataforma CORBA passando código como mensagem. Os resultados foram obtidos sobre uma implementação de CORBA 1.1 da Iona - Orbix v.1.3 para linguagem C++.

O código de uma implementação é obtido a partir da informação contida no Repositório de Implementação (no remetente) e enviado ao destinatário como uma seqüência binária na forma de uma seqüência em IDL. Um Repositório de Implementação contém (arquivos de) informação para a ativação de um servidor como localização da classe, sendo necessário em implementações CORBA utilizando um Gerente de Ativação. A seqüência é enviada como argumento e armazenada no repositório de código do destinatário. O repositório de código é baseado em arquivos, onde cada agente está em um arquivo próprio. Uma abordagem melhor para um número maior de agentes é uma base de dados orientada a objetos onde estado e código são armazenados persistentemente. Isto envolve o serviço CORBA de persistência que não é aprofundado neste teste.

As curvas na Figura 5.1 apresentam a variação da taxa de transferência de seqüências em função do tamanho das seqüências. Na Figura 5.1a, cliente e servidor estão colocados numa mesma máquina enquanto na Figura 5.1b cliente e servidor estão em máquinas remotas. O resultados são baseados em máquinas sparc 1+ e 2 não dedicadas, rede *Ethernet* e tamanhos de seqüências de 10KB a 10MB. Estas máquinas de recursos computacionais, atualmente limitados, sugerem um limite *superior* no tamanho da mensagem e *inferior* na taxa de transferência. Estes limites não representam uma inviabilização da proposta. Os resultados sugerem a utilização de mensagens do tipo *ascii* e do tipo *binário,* até o tamanho de 1MByte.

Os resultados correspondem às expectativas. Na Figura 5.1a, a taxa de transmissão cai rapidamente com um tamanho de mensagem acima de 2 MB. Isto se deve à alocação de memória para a mensagem consumir a memória disponível do sistema operacional. Como cliente e servidor estão co-localizados na mesma máquina, o consumo de memória avança mais rápido mas são possíveis taxas de transmissão mais elevadas se comparado com a Figura 5.1b onde cliente e servidor estão remotos em máquinas diferentes. Na Figura 5.1b a alocação de memória conta com duas vezes mais recursos disponibilizados pelas duas máquinas, entretanto a comunicação está limitada pela taxa possível pela *ethernet* (< 1,25 MByte/s).

A memória alocada para a seqüência impõe um limite prático ao tamanho da mensagem, e um compromisso pode ser obtido no particionamento da mensagem em pacotes que refletem um tamanho de *buffer* predefinido no protocolo de rede. Na referência [Schmidt95] arquivos de imagem, de até 64 MBytes, são passados em pacotes de 124 KBytes usando um canal externo adicional de alta velocidade.

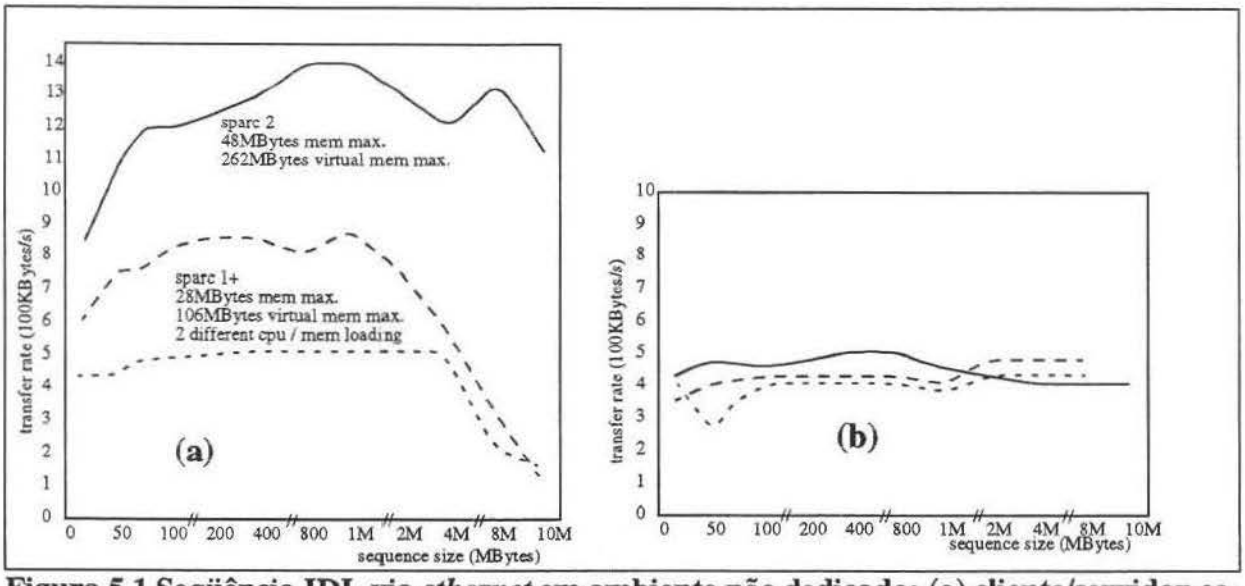


Figura 5.1 Seqüência IDL via *ethernet* em ambiente não dedicado: (a) cliente/servidor co-localizados e (b) cliente/servidor remotos.

Estes testes antecedem os avanços na linguagem Java [Java96, Java97, Java98-1, Java98-2, Java98-3] com o desenvolvimento de uma superclasse que permite a serialização de uma ins-

tância de objeto, contendo código e estado. Uma vez serializado um objeto pode ser enviado para um novo destinatário e deserializado. Para o novo padrão CORBA 3, o OMG propõe um padrão para a migração de objetos, denominado *Object by Value*. Especifica também mudanças no procedimento de *marshaling* e particionamento da mensagem. Atualmente o *marshaling* deve ser feito em toda a mensagem para um posterior envio. A nova especificação sugere o particionamento da mensagem anterior ao *marshaling* que passa a ser feito em mensagens de tamanho predeterminado (e menor) favorecendo a alocação de memória.

5.2 Prototipagem na Migração de Componentes

Nesta seção apresentamos uma prototipagem que simula o estudo de caso em que um objeto (ou agente) cliente procura por uma instância de um servidor móvel. O cliente faz requisições aleatórias às agências do domínio fazendo a instância do servidor migrar para a primeira agência que atender a uma das requisições. Em seguida o cliente faz uma série de requisições aleatórias às agências do domínio fazendo uma nova instância do cliente migrar para a primeira agência que atender a uma das requisições. Este é um caso representativo de migração implícita cuja prototipagem valida a migração implícita proposta no trabalho.

Esta prototipagem pretende simular um hospedeiro móvel que se desconecta de um conjunto de hospedeiros (estáticos) sendo tratado de forma similar a um hospedeiro que fica indisponível, seja por desligamento, perda de desempenho ou falha. Nestas situações a aplicação cliente deve migrar para outro hospedeiro disponível da rede estática.

O mecanismo foi avaliado na versão de Orbix2.1 (CORBA 2) / C++, OrbixWeb3.0 / Java, ambiente SunOS 5.5.1 e hospedeiros sparc-5 até sparc-ultra. Esta prototipagem não utilizou persistência de estado de cliente e servidor.

Estratégia. A estratégia é baseada na migração dos componentes em um hospedeiro móvel para outro hospedeiro na rede. Supõe-se a existência na rede de repositório (espelho) do cliente contendo: estado, execução, interfaces e dados. Um hospedeiro móvel que desconecta e reconecta pode ser visto como em recuperação de falha [Cristian94, Cristian93, Cristian91]. Este teste se concentra na realocação de recursos e na reativação de serviços. Neste processo, duas fases são sugeridas na migração de componentes de um hospedeiro móvel: uma fase de emergência e uma fase de recuperação.

5.2.1. Fase de Emergência

1. O cliente requisita uma nova ativação no último servidor em contato.
2. Esta ativação também pode ser solicitada pelo *mestre* no caso de falta de resposta do cliente.
3. Para certificar a ativação, o servidor requisita um cancelamento ao cliente.
4. Se o cancelamento não vier, o cliente é ativado no mesmo servidor.
5. A nova ativação passa a atender as novas requisições e inicia a fase de recuperação.

5.2.2. Fase de Recuperação

1. A nova ativação é responsável por migrar para um novo hospedeiro disponível;
2. Começa a busca pelo hospedeiro disponível;
3. O hospedeiro que melhor satisfaz a busca é o hospedeiro móvel original;
4. O hospedeiro móvel não estando disponível, outro hospedeiro é selecionado aleatoriamente, a partir de uma lista;
5. Novo cliente é ativado no hospedeiro selecionado;
6. Somente uma cópia ativada prossegue a execução.

Conexão Cliente/Servidor. Um cliente CORBA faz uma chamada de *_bind* a um servidor CORBA. Associado ao *_bind* ocorre todo o processo de seleção e alocação. Diversas estratégias são possíveis para o cliente migrar de volta ao hospedeiro móvel. A abordagem neste trabalho é usar cada *_bind* como ponto de verificação para uma nova tentativa de migração.

Procedimento de seleção do hospedeiro. Parâmetros de QoS e uma faixa estimada devem auxiliar na seleção de hospedeiro(s). A escala deve permitir uma escolha mais flexível na seleção final e reduzir a possibilidade de oscilações de min/max que podem surgir com critérios de minimalidade ou maximalidade. O uso de uma escala facilita também o processo de seleção. Uma estimativa de custo de execução pode ser traduzida em uma escala da disponibilidade expressa em termos de valores de desempenho dos hospedeiros (*benchmarks*).

Quando o hospedeiro móvel não responde para a recuperação, uma nova requisição é enviada seguida de requisição de nova ativação de cliente em um outro hospedeiro (estacionário). O teste é baseado numa versão simplificada do procedimento de seleção de hospedeiro. O Trader não está realmente participando do processo, mas uma lista com um conjunto de possíveis hospedeiros organizada por serviço. Quando um *_bind* a um serviço é requisitado, um hospedeiro é selecionado aleatoriamente do conjunto. Se o hospedeiro não responder, um outro hospedeiro é selecionado de forma semelhante do subconjunto restante. O protocolo de *_bind* usa parâmetros como o número de tentativas e tempo de resposta. Estes parâmetros permitem ajustar o nível de discriminação de disponibilidade de um hospedeiro.

A implementação utilizada para testes de avaliação e desempenho foi baseada no uso do protocolo Orbix com *Locator default*. Por exemplo, para uma configuração com seis hospedeiros, no arquivo *orbix.hosts* descrito anteriormente, foram colocados entradas contendo:

- clienteR : hospedeiro1, hospedeiro2, hospedeiro3, hospedeiro4, hospedeiro5, hospedeiro6
- Servidor : hospedeiro1, hospedeiro2, hospedeiro3, hospedeiro4, hospedeiro5, hospedeiro6
- IT_daemon : hospedeiro1,hospedeiro2,hospedeiro3,hospedeiro4,hospedeiro5,hospedeiro6

A cada chamada do método *<Tipo>Helper.bind (String NomeServidor, String Hospedeiro, X x)* o *Locator* é chamado para a obtenção de uma nova referência de objeto a ser utilizada nas requisições subseqüentes. Este método *bind* equivale a uma simplificação do método bind apresentado na interface de Transparência na Figura 3.16 (Seção 3.4) e na Tabela 4.1 (Seção 4.1).

Associado ao *Locator* é possível utilizar uma configuração alternativa através de um arquivo *orbix.hostgroups* onde são definidos nomes de grupos associados a hospedeiros ou a outros grupos. Neste caso teríamos arquivos do tipo:

orbix.hosts

- clienteR : redefixa : outrogrupo
- Servidor : redefixa

orbix.hostgroups

- hospedeiromóvel : hospedeiroM
- redefixa : hospedeiro1, hospedeiro2, hospedeiro3, hospedeiro4, hospedeiro5, hospedeiro6
- outrogrupo : ...
- IT_daemon : hospedeiromóvel : redefixa : outrogrupo

Neste caso criamos um grupo *hospedeiromóvel* composto do *hospedeiroM*, um grupo *redefixa* composto dos *hospedeiro1* a *6* e o grupo *IT_daemon* composto dos grupos definidos anteriormente, onde ' : ' é usado para separar os nomes dos grupos.

Num caso genérico, podemos ter estes arquivos configurados de forma (um pouco) diferente em cada agência, i.e., a configuração de nomes e grupos vista por um agente pode variar para mesmos nomes e grupos de acordo com a agência onde está residindo. Isto permite compor itinerários de hospedeiros e/ou grupos de hospedeiros.

Ciclo de Teste. O teste da avaliação é baseado em uma execução cíclica de um conjunto de comandos que envolvem a carga de comunicação e processamento. Este teste cíclico é composto de uma instância de um *servidor mestre* e de uma instância de um *cliente*. O cliente pode ser uma instância *original* ou uma *réplica*. A instância original é ativada em um hospedeiro pré-determinado, enquanto a réplica e o servidor mestre se movem para balanceamento de carga, selecionando aleatoriamente o primeiro hospedeiro que atender. A seqüência de teste está representada na Figura 5.2.

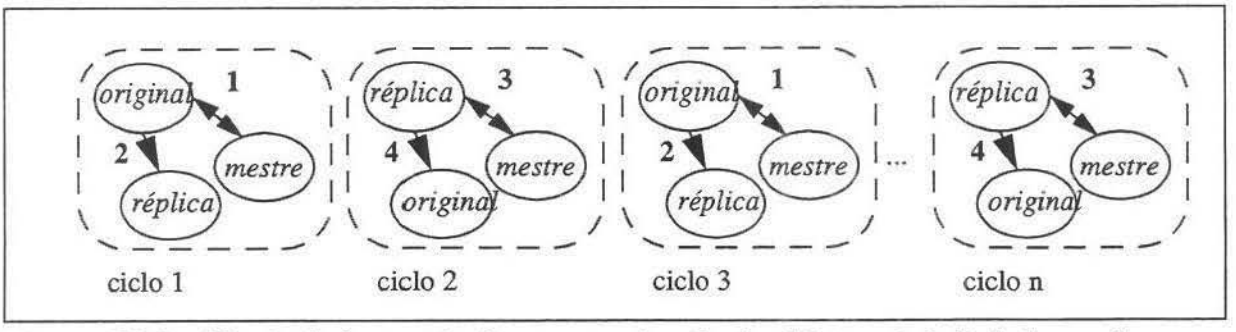


Figura 5.2 Seqüência de instanciação e comunicação de objetos: 1, 2, 3, 4, 1, e assim por diante.

Interfaces. A seguir está uma descrição das interfaces dos objetos representados na Figura 5.2 em um pseudo-código orientado a objeto.

```
// Mestre
interface ObjetoMestre_1 {
   oneway void any_method_a ();
   oneway void any_method_b ();
}
interface ObjetoMestre_2 {
   long void any_method_aa ( in typedef var1aa, inout typedef var2aa, out typedef var3aa);
}
// Cliente Original
interface ClienteO {
   oneway void main_method ();
   oneway void any_other_entry_point_method ();
}
// Cliente Réplica
interface ClienteR {
   oneway void main_method ();
   oneway void qualquer_outro_metodo_de_entrada ();
}
```

Repositório de Implementação. Em Orbix, é um diretório onde o registro de um serviço é armazenado. Configurações diferentes são possíveis para o Repositório de Implementação da ORB:

1. Um Repositório de Implementação compartilhado por todos os hospedeiros como uma agência (ORB) multiprocessada.
2. Um Repositório de Implementação separado para cada hospedeiro como várias agências (ORBs) monoprocessadas.
3. Uma abordagem mista.

Uma agência multiprocessada significa que qualquer hospedeiro da agência pode atender a uma requisição permitindo um balanceamento ao nível interno da agência. Um domínio de hospedeiros pode ser usado como uma única agência ou particionado em mais agências. Na Figura 5.3 estão representadas diferentes configurações de hospedeiros: a) um para o cliente original e seis para o servidor mestre e cliente réplica; b) um para o cliente original, três para a réplica e três para o servidor mestre; c) um para o cliente original, um para a réplica e dois para o servidor mestre; d) todos em um único hospedeiro.

Benchmark. O desempenho de execução é relativo a um *benchmark* baseado no desempenho de execução do cliente e servidor co-localizados na mesma agência. Este benchmark é obtido para duas condições de carga diferentes obtidas com um programa de carregamento explícito; são elas: normal e carregada. O programa de carga consome os mesmos recursos que o programa de teste, de modo a representar uma carga real do ponto de vista do programa de teste. O benchmark, na Tabela 5.1, é obtido para o hospedeiro mais rápido e o mais lento, estabelecendo um limite inferior e um limite superior de desempenho para a execução distribuída nos diversos hospedeiros. O benchmark (3x1) não é uma situação real. Neste teste,

a comunicação não passa pela rede e é 50% mais alta de quando cliente e servidor estão remotos comunicando via *ethernet* [Schulze97-2].

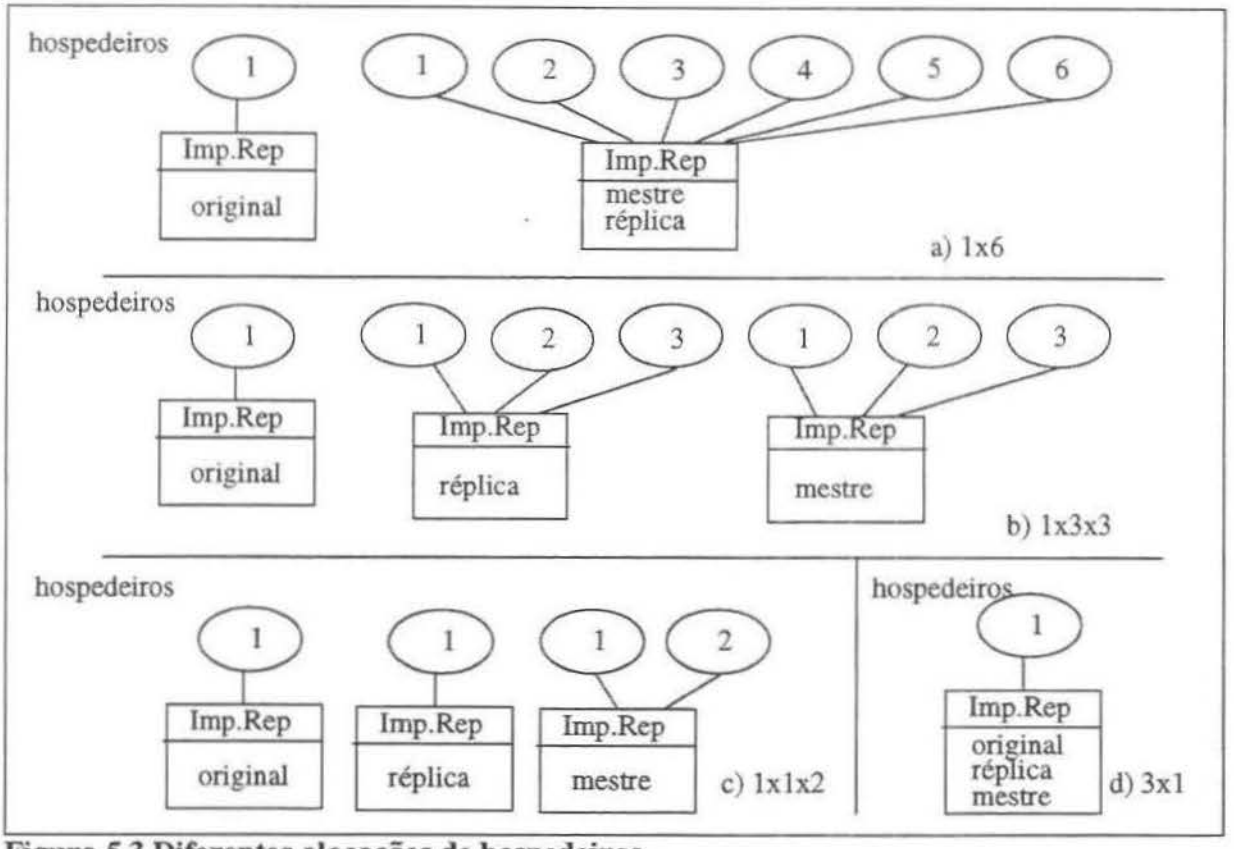


Figura 5.3 Diferentes alocações de hospedeiros.

Tabela 5.1 Comparação entre hospedeiros e *benchmark*.

Nome	Clock (MHz)	Memória (MBytes)	Comparativo Clock | Mem.	Tipo	Benchmark normalizado* (min / ciclo) normal | c/carga
itapoa	2x 60	96	1.7 | 3.0	Sparc-20	3.5 | 4.8
aracati	110	64	1.6 | 2.0	Sparc-5	|
tambau	110	64	1.6 | 2.0	Sparc-5	|
pajussara	110	32	1.6 | 1.0	Sparc-5	4.1 | ----
ilhabela	85	32	1.2 | 1.0	Sparc-5	|
tutoia	40	32	0.6 | 1.0	Sparc-5	4.7 | ----
juquei	70	32	- 1.0 | 1.0 -	Axil-235	5.5 | 8.7

Os valores na Tabela 5.2 são resultados médios computados em ocasiões diferentes selecionadas aleatoriamente com valores mínimos e máximos.

Tabela 5.2 Medidas de execução.

Configuração (Figura 5.3)	**Resultados** (min / ciclo) hospedeiros c/ carga normal	hospedeiros c/carga extra metade	todos	**Comentários**
(a) 1 x 6	4.1 - 5.2*	5.4	10.4 - 11.5	1 de 6 hospedeiros falhando
(b) 1 x 3 x 3	~5.			
(c) 1 x 1 x 2	~7.			
(-) 1 x 1 x 6	~5.			original=1 / réplica=1 / mestre=6
(d) 3 x 1	3.6 - 5.5	--	4.8 - 8.7	*benchmark* empírico para o hospedeiro mais rápido e o mais lento

Na Tabela 5.2 apresentamos todos os resultados como um faixa entre o *mínimo* obtido para o hospedeiro de mais desempenho e o *máximo* obtido para o hospedeiro de menos desempenho. A melhor situação de execução acontece quando os processos estão co-localizados no hospedeiro mais rápido e sem carga. Em função disto utilizamos este valor como *benchmark* empírico para comparação de desempenho entre os hospedeiros. Ainda para efeito de comparação de desempenho obtém-se o mesmo benchmark para os hospedeiros com carga, isto é, comparação de alteração de desempenho com a carga. Como pior situação de execução teremos os processos todos remotos e os hospedeiros todos carregados, pois neste caso além da falta de desempenho da cpu soma-se o custo da transmissão. De forma prática, estima-se que o mecanismo de migração seja eficiente enquanto houver um número de hospedeiros livres maior ou igual ao número de processos remotos envolvidos. O desempenho esperado com migração deve ficar entre o resultado do *benchmark* sem carga e com carga. Na Tabela 5.1, configuração (d), pode-se observar como o benchmark se degrada quando adicionamos carga às máquinas listadas. Para a configuração (a) com os hospedeiros sem carga, observa-se como a faixa de valores cabe perfeitamente dentro da faixa de benchmark sem carga para o melhor e o pior hospedeiro, como esperado. Quando carregamos metade dos hospedeiros o comportamento se mantém. Quando carregamos todos os hospedeiros, então a migração deixa de ser interessante e o resultado é pior que todos os processos co-localizados no mesmo hospedeiro.

Crescendo o número de hospedeiros crescem também as possibilidades de hospedeiros não carregados e o desempenho médio com migração é maior do que sem migração. Para um computador móvel, o uso de migração de componentes aplica-se de maneira similar e a vantagem é a sustentação da execução das tarefas e da disponibilidade dos serviços. Podemos concluir que não há perda significativa de desempenho com a migração, pois a condição em que ocorre perda é uma situação em que provavelmente será melhor esperar ou contratar novos recursos.

Classes dos Objetos. Abaixo está a estrutura básica das classes para cada um dos objetos da Figura 5.2, apresentados em pseudo-código orientado a objeto.

```
// Classe Mestre
public class mestre {
   _ObjetoMestre_1Ref newobject1 = null;
   _ObjetoMestret_2Ref newobject2 = null;
   try {
       newobject1 = new ObjetoMestre_1Impl();
       newobject2 = new ObjetoMestre_2Impl();
    _CORBA.Orbix.impl_is_ready("qq_nome_para_mestre");
   } catch( Exception) {
       exception_handling;
   }
}

// Implementação de ObjetoMestre_1
class ObjetoMestre_1Impl extends _alguma_classe_básica_de_ObjetoMestre_1 {
   // construtor e métodos desta classe devem ser declarados aqui //
}

// O mesmo para a Implementação do ObjetoMestre_2

// Classes Cliente
public class clienteo {
   _ClienteORef newobject = null;
   try {
       newobject = new ClienteOImpl();
       _CORBA.Orbix.impl_is_ready("qq_nome_para_clienteo");
   } catch( Exception ) {
       exception_handling;
   }
}

// Implementação de ClienteO
class ClientOImpl extends _alguma_classe_basica_de_ClienteO {
   // construtor da classe //
   public ClienteOImpl() throws SystemException {
       super();
   }
   public final void main_method() {
        qualquer_metodo_particular ();
   }
   public final qualquer_outro_metodo_de_entrada { };

   // Um método de finalização requisita a ativação de um novo cliente
   public final void finalize() {
       _ClienteRRef clientR;
       try {
           clienteR = ClienteR._bind("qq_nome_para_clienter", null);
       } catch ( Exception ) {
           exception_handling; return;
       }
       try {
```

```
            clienteR.main_method();
        } catch ( Exception ) {
            exception_handling; return;
        }
    }
}

// Similar para clienteR //

// Classe Comum de Implementação de Cliente
public class cliente {
    // typedef declarations
    public static void qualquer_metodo_particular () {
        _ObjetoMestre_1Ref newobject1 = null;
        _ObjetoMestre_2Ref newobject2 = null;
        try {
            newobject1 = ObjetoMestre_1._bind("qq_nome_para_mestre", null);
        } catch ( Exception ) {
            exception_handling;
        }
        try {
            ObjetoMestre_1.qualquer_metodo_x();
        } catch ( Exception ) {
            exception_handling;
            return;
        }
        try {
            newobject1 = ObjetoMestre_2._bind("qq_nome_para_mestre", null);
        } catch ( Exception ) {
            exception_handling;
        }
        try {
            MasterObject_2.qualquer_metodo_xx();
        } catch ( Exception ) {
            exception_handling;
            return;
        }
    // Qalquer outra requisição para outro servidor / objeto
    }
}
```

5.2.3. Comparação de Desempenho com Java

O primeiro teste em C++ foi modificado e repetido para uma comparação com Java. A versão original do teste em C++ passava 10 vezes uma seqüência de tamanho 100000. Mantendo o teste com estes valores resultou numa excessiva alocação de memória pelo processo da *Java Virtual Machine*. Houve uma degradação demasiada do teste com alguns hospedeiros não conseguindo ativar a JVM nos tempos de resposta e número de tentativas estabelecidos. Como não era possível alterar as condições de carga das máquinas e desejada uma comparação de resultados em condições idênticas para as duas linguagens, preferiu-se a alteração dos parâmetros do

programa. Conseqüente o teste foi modificado para passar 10000 vezes uma seqüência de tamanho 100. Os resultados são apresentados nas Tabelas 3 e 4, de acordo com a análise anterior para as Tabelas 1 e 2 respectivamente.

Tabela 5.3 Semelhante a Tabela 5.1, mas em máquinas diferentes.

Nome	Clock	Mem.	Comparativo Clock | Mem.	Tipo	Benchmark (min/ciclo) Java/OrbixWeb		C++/Orbix2.1	
	(MHz)	(MBytes)			normal	c/carga	normal	c/carga
hawk	2x 50	128	1.4 | 4.0	sparc-20	-	-	4.8 [2]	-
mimosa	2x 50	128	1.4 | 4.0	sparc-20				
marconi	2x 50	64	1.4 | 2.0	sparc-20				
goose	143	64	2.0 | 1.0	sparc-ultra	8.4 [2]	-	3.8 [2]	-
					-	-	3.3 [1]	-
owl	110	64	1.6 | 2.0	sparc-5				
metis	110	64	1.6 | 2.0	sparcs-5	9.4 [2]			
hydra	70	82	1.0 | 2.5	sparcs5	22.0 [2]	-	6.8 [2]	-
					-	-	4.3 [1]	-

1. teste antigo 2. teste novo

Na Tabela 5.4 estão resumidos os resultados para as duas linguagens, para o *benchmark* da configuração da Figura 5.3d e para o uso com migração na configuração da Figura 5.3a. Observa-se que os resultados para a configuração (a) cabem dentro do intervalo obtido para a configuração (d) conforme esperado e explicado anteriormente para a Tabela 5.2. Na comparação das duas linguagens observa-se a degradação de Java em relação a C++, como esperado, tanto para o benchmark quanto para o resultado com migração. O problema de alocação de memória que levou a modificação do teste já permite esperar resultados deste tipo. A degradação pode ser minimizada ou eliminada com o uso de JVMs em *hardware* dedicado.

Tabela 5.4 Resultados para as configurações (a) e (d) da Figura 5.3.

Linguagem	Resultados min/ciclo (d)3x1	(a)1x6	Comentários
C++	3.8 - 6.8	6.2	*benchmark* empírico p/hospedeiro +
Java (interpretado)	8.4 - 22.0	13.2	rápido e o + lento

Testes com o OMG/Trader não foram realizados. A versão do Trader atual em C++ do GMD FOKUS foi considerada, mas a avaliação poderia incluir a versão Java. Outras comparações de desempenho poderiam levar em conta outros ORBs como: Visibroker da Visigenic, COOL para tempo real, JavaIDL de JDK1.2, outras plataforma de agentes móveis, e compiladores instantâneos - *Just In Time*. Um JIT compila métodos um após o outro ao serem chamados. Se um método for chamado mais de uma vez, então compensa por não ser preciso recompilá-lo mas apenas reexecutá-lo. Às vezes o código gerado pelo JIT não é mais rápido que o código interpretado. Embora raro, compilar os *bytecodes* pode ser mais lento que interpretá-los.

5.3 Comentários sobre a (Re)distribuição de Serviços

A inicialização de uma aplicação pode utilizar o mecanismo, descrito na Seção 3.1 [Schulze97-1], de contratação de serviços disponíveis e de carregamento de serviços novos, isto é, serviços existentes são contratados e serviços indisponíveis são distribuídos a partir de um Repositório de Implementação (e classes) local ou remoto.

Há diferentes possibilidades para a distribuição de serviços indisponíveis com relação a localização na inicialização. Em um caso simples, o repositório de classes está local na máquina de inicialização da aplicação. Em outro caso, uma aplicação mais susceptível demanda maior confiabilidade e portanto pode-se adicionar um servidor contendo o repositório de implementação (e classes) a fim de manter uma cópia dos serviços a serem (re)distribuídos naquele contexto. Portanto, a (re)distribuição parte deste servidor de repositórios. A possibilidade de falha deste servidor de repositório existe e pode ser tratada usando a mesma abordagem, isto é, adicionando novo servidor e assim por diante até o nível de relação custo / confiabilidade aceitável.

Independente do tipo do repositório de implementação (e classes), serviços novos são (re)distribuídos de acordo com a demanda, seja para balanceamento de carga e/ou *caching* inverso. Isto significa que se a distribuição não for necessária, a execução pode ser local. Uma alternativa é a aplicação esperar por recursos numa fila de inicialização.

Para *chaching* inverso as circunstâncias são consideradas como a seguir:

- um objeto necessita se mover para um local específico a fim de executar junto a um serviço contratado;
- o objeto móvel contacta o suporte local de agentes e solicita a migração para onde um serviço específico está;
- o suporte de agente localiza o serviço específico e envia o agente ao suporte de agente remoto.

Para balanceamento de carga, acrescentamos:

- agências são contratadas de acordo com critérios de consulta;
- uma instância é selecionada, ativa ou não, usando *multicast*;
- o serviço de Disponibilidade utiliza um Gerente de Ativação local para a ativação.

A avaliação se baseia em um cliente e um servidor móveis. A requisição do cliente move o servidor para uma nova agência com memória e *cpu* suficientes. O cliente contacta aleatoriamente um hospedeiro no grupo até encontrar um disponível. O estado da execução é mantido no cliente enquanto o servidor reinicia em seu estado inicial.

A fim de desacoplar este teste de desempenho do teste de passagem de código, disponibilizamos o código do servidor em cada hospedeiro de modo que somente a execução seja passada ao hospedeiro seguinte. O processo permite que a mesma aplicação seja executada diversas vezes simultaneamente.

5.4 Prototipagem em Gerenciamento

Apresentamos resultados de aplicação de agentes móveis em gerenciamento de sistemas distribuídos como uma aplicação de características particularmente favoráveis ao paradigma de agentes móveis. Estes resultados estão apresentados no Capítulo 6 [Schulze99-2].

CAPÍTULO 6

Aplicação: Gerenciamento de Serviços usando Agentes Móveis em Ambiente CORBA

Nunca se percebe o que já foi feito;
a gente só nota o que ainda está por fazer.
Emily Dickinson

Este capítulo apresenta a aplicação de agentes móveis em gerenciamento de serviços distribuídos baseados em CORBA [Schulze99-2]. O capítulo compreende:

- 6.1 Introdução página 91,
- 6.2 Metodologia página 92,
- 6.3 Arquitetura página 97,
- 6.4 Aspectos de Implementação página 100,
- 6.5 Resultados página 105 e
- 6.6 Comentários página 107.

6.1 Introdução

Exploramos mobilidade de computação no âmbito de gerenciamento de sistemas distribuídos abertos e apresentamos uma arquitetura com agentes móveis. As vantagens são consideradas do ponto de simplicidade e adaptabilidade do gerenciamento aos novos ambientes gerenciados ao invés da otimização de desempenho em problemas de soluções bem estabelecidas.

O modelo tradicional envolve processos estáticos transferindo dados e/ou comandos. Os dados e os comandos são a parte móvel de uma computação e os programas são a parte estática. Um crescente número de cenários não pode ser abordado de forma eficaz por tais paradigmas de interação estática. Um paradigma alternativo é o de agentes móveis, i.e., um programa identificado de forma única que pode migrar de máquina para máquina em um ambiente heterogêneo. Uma agência existe em cada máquina destino para receber e executar os agentes. A expectativa de desempenho maior está em redes de largura de banda estreita e de latência elevada.

Do ponto de vista arquitetural, gerenciamento pode variar de componentes inteiramente estáticos a componentes inteiramente móveis. Os esquemas estáticos são baseados em um controle de gerenciamento, e recursos distribuídos, ou em um controle com agentes distribuídos e associados aos recursos controlados. Os esquemas móveis são baseados em um controle estático e

agentes móveis que se conectam aos recursos controlados, ou em um controle móvel com agentes móveis que se conectam a recursos móveis.

São apresentados resultados do ponto de vista de uma implementação e de um estudo de caso de gerenciamento. A implementação é baseada em um *middleware* OMG/CORBA [CORBA98, OMG97-3] e o ambiente controlado é um mundo de objetos CORBA, isto é, objetos de aplicação e objetos de serviço [OMG97-1, OMG97-2, OMG98-1, OMG98-2]. A definição do problema é apresentada dos pontos de vista topológico, funcional e operacional. Do ponto de vista topológico, a população de objetos se encontra aglomerada em pequenos conjuntos (*clusters*) e distribuída no ambiente controlado. Em sistemas orientados a processo, estes conjuntos são processos. O uso de agentes móveis simplifica a visão topológica de gerenciamento dos objetos distribuídos.

Do ponto de vista funcional, os agentes móveis alteram dinamicamente a funcionalidade de gerenciamento e recolhem informação. Do ponto de vista operacional, uma solução de compromisso entre latência e velocidade de processamento é mover os dados em direção ao código, ou o código em direção aos dados - *caching* inverso [Goldszmidt96]. Manter os dados próximos de onde são consumidos é eficaz quando há um grau elevado de localidade no uso e de coerência, sendo impróprio para dados distribuídos altamente voláteis, que se desatualizam rapidamente e tornam a *cache* inconsistente. Em resumo, mover código para perto dos dados pode reduzir excedentes de comunicação de dados enquanto mover o código a destinos remotos pode minimizar a ausência de recursos computacionais limitados.

Soluções baseadas em tecnologia de código não móvel podem sempre ser propostas. Entretanto, tarefas como gerenciamento e recuperação de informação, se ajustam naturalmente ao saltar-executar-saltar de agentes móveis. Um agente migra para uma máquina, executa uma tarefa, migra para outra, executa outra tarefa dependente da saída precedente, e assim por diante.

O objetivo do trabalho é gerenciar aplicações baseadas em um *middleware* objeto-orientado que seja baseado em CORBA. Um conjunto de agentes é definido para explorar o ambiente controlado baseado no *detalhamento* de problemas potenciais. Uma extensão desta abordagem é proposta para permitir ajustes no sistema gerenciado. Este ajuste é baseado no balanceamento dos recursos computacionais com a ajuda de um *serviço de Disponibilidade*.

6.2 Metodologia

Nesta seção exploraremos aspectos dos diferentes esquemas de gerenciamento, o paradigma de agentes móveis em gerenciamento e uma proposta de ajuste usando migração de agentes.

6.2.1. Esquemas de Gerenciamento

Na Figura 6.1 apresentamos quatro esquemas de gerenciamento. No esquema 1, a unidade de controle conecta-se diretamente aos recursos controlados. No esquema 2, os agentes estáticos fazem as conexões aos recursos controlados, coletam a informação de gerenciamento e a arma-

zenam na base de informações de gerenciamento [Stallings93, Queiroz97]. No esquema 3, os agentes estáticos são substituídos por agentes móveis e funcionalidade adicional é delegada do gerente aos agentes [Schulze98-1]. No esquema 4, as partes envolvidas podem se mover, isto é, os agentes e os serviços ou recursos. No trabalho atual, exploramos os esquemas com mobilidade, com detalhes de projeto e de implementação para o esquema 3 e considerações de projeto para o esquema 4.

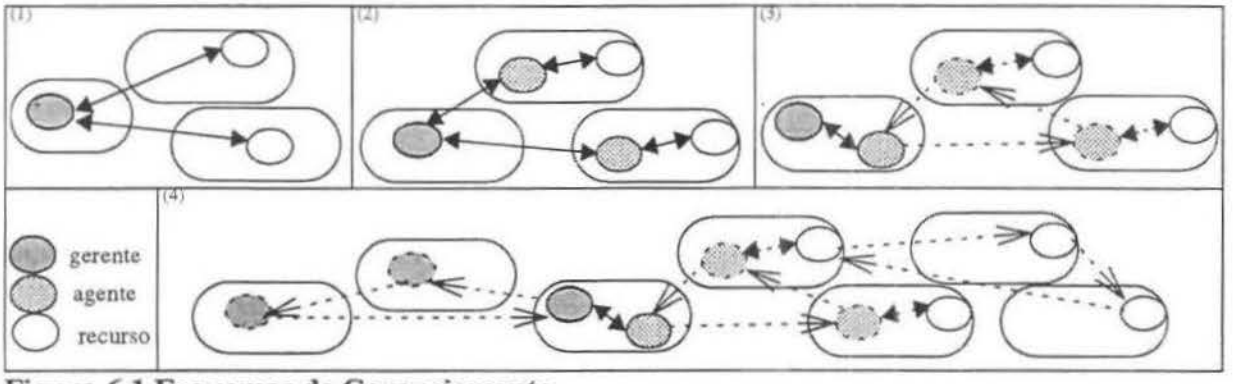


Figura 6.1 Esquemas de Gerenciamento.

Começamos com o caso mais simples onde somente os agentes são móveis. Nesta situação exploramos um estudo de caso no qual queremos detectar objetos distribuídos que estejam potencialmente causando problemas. A avaliação dos objetos é baseada nas requisições que chegam ou saem dos objetos. Baseado nas requisições estimamos a atividade, isto é, *throughput*, tempo de resposta, volume de dados e mais tarde a comparamos com a ocupação de cpu e de memória. Esta abordagem é baseada na estimativa de potenciais problemas considerando o tipo de aplicação que prevalece no domínio e suas características. Para adaptar o gerenciamento ao ambiente, o gerente deve utilizar uma biblioteca de agentes, selecionando e combinando-os de acordo, ou criando novos agentes e adicionando-os à biblioteca existente.

Uma extensão deste trabalho é o esquema de gerenciamento onde todos os componentes podem ser móveis, se movendo explicitamente como definido em seu código ou se movendo transparentemente por uma decisão externa do middleware ou de um cliente fazendo uma requisição. Maiores detalhes são apresentados mais adiante no texto.

6.2.2. Paradigma de Agentes Móveis

Na Figura 6.2, representamos o mapeamento topológico do plano de agentes para o plano correspondente de processamento distribuído e o plano físico subseqüente. A figura apresenta uma agência agrupando dois nós correspondentes a duas máquinas diferentes no plano físico. A abordagem assíncrona de agentes móveis necessita menor organização de mensagens e sincronização entre processamentos. Espera-se abordar problemas de gerenciamento de forma mais objetiva no plano de agentes móveis do que em nível dos outros dois planos.

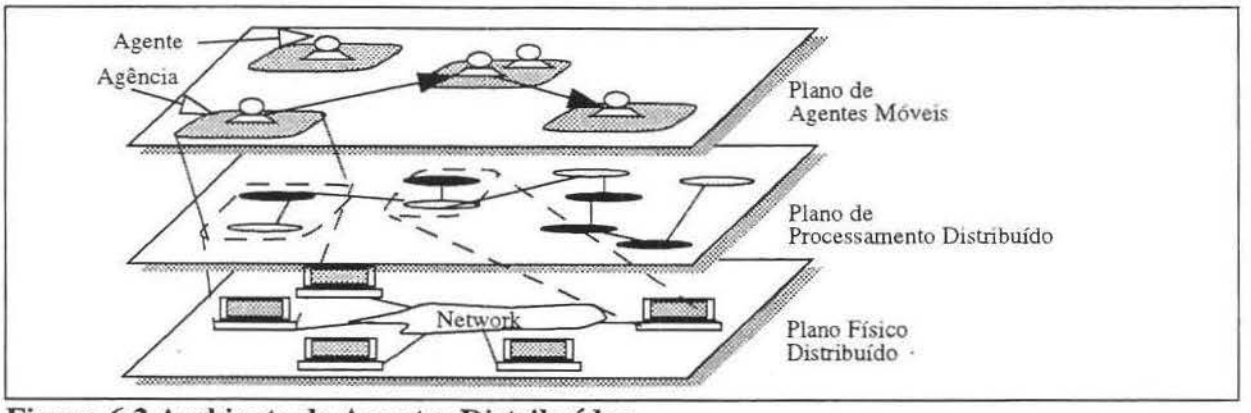


Figura 6.2 Ambiente de Agentes Distribuídos.

Agentes móveis permitem uma arquitetura dinâmica para enviar serviços de gerenciamento, com aplicações leves e flexíveis, que podem ser criadas, desdobradas, estendidas, ou suprimidas em tempo real [Schulze99-1, Schulze97-1, Krause97, Expertsoft, Odyssey97, Aglets, GrassHopper98]. Permite a criação e distribuição de serviços necessários para novos dispositivos orientados a serviços. Serviços podem ser incorporados diretamente nos agentes para executar tarefas de gerenciamento autonomamente e de intervenção sem ajuda humana.

6.2.3. Ajustes

Nesta seção descrevemos sucintamente os mecanismos de ajuste para o balanceamento dos recursos gerenciados por meio de um *serviço de Disponibilidade* [Schulze99-1, Vasconcellos98]. Serviços indisponíveis, podem ser (re)distribuídos atendendo uma demanda de balanceamento de carga e/ou *caching* inverso.

Na Figura 6.3 representamos um agente genérico A enviando uma requisição a um agente móvel B. O agente B pode ser encontrado em uma das agências representadas. A requisição segue o procedimento de localização com um multicast ao nível final a fim de selecionar um hospedeiro que atenda à solicitação. Os mesmos passos são seguidos numa rechamada do agente B ao A. O mesmo procedimento também ocorre quando o agente A deseja migrar transparentemente e envia uma requisição para uma nova instância ativa, isto é, um agente A envia uma requisição a um novo agente A. O primeiro hospedeiro a atender prossegue com a execução do agente.

Definimos mobilidade explícita como a demanda por mobilidade que está codificada explicitamente no agente. Também definimos mobilidade implícita ou transparente como sendo conseqüência de uma ação ou razão externa. Por exemplo, um movimento explícito do agente B pode resultar implicitamente em um movimento do agente A se o segundo tiver que executar associado ao primeiro. Propomos que um movimento implícito ocorra no momento de uma rechamada do primeiro agente em resposta a uma requisição do segundo, como na Figura 6.3. Com esta abordagem baseada na rechamada esperamos adiar a migração transparente até o agente e os recursos associados responderem. Assim esperamos minimizar migrações desnecessárias ou um efeito *pingue-pongue*.

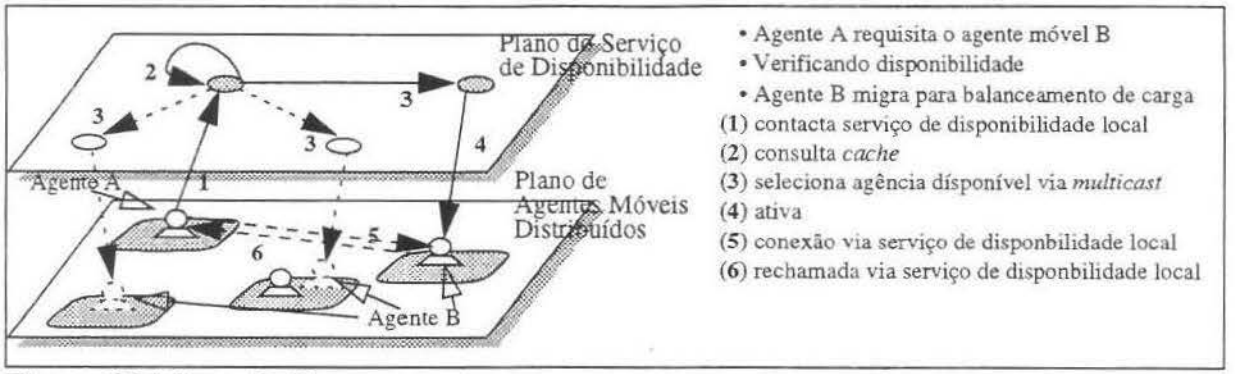


Figura 6.3 Migração Transparente.

Procurando e Alocando Recursos. Um agente cliente requisita um outro agente através do serviço de disponibilidade local o qual obtém o domínio deste outro agente, seja de um cache local, de um serviço de Nomes ou Negociador. A partir do domínio obtido o cliente procura conectar-se ao agente servidor, selecionando uma agência aleatoriamente. A conexão à agência passa pelo gerente da agência que consulta o serviço de Nomes local pela IOR correspondente (Referência Interoperável de Objetos). Se somente uma cópia ativa for permitida, o gerente tenta se conectar retornando a IOR. Se falhar, tenta uma ativação local e retorna a IOR. Se falhar retorna o controle ao lado cliente.

Balanceamento de Recursos . Dois cenários diferentes de serviços e agentes são possíveis. Um cenário é baseado nos serviços que existem somente durante o tempo de vida de uma aplicação, isto é, são serviços transitórios. Este cenário é dominado pelo tempo de inicialização e distribuição dos serviços individualmente. Um outro cenário consiste de serviços e agentes que participam em mais de uma aplicação e que persistem além do tempo de vida individual das aplicações, isto é, serviços persistentes. Persistente não significa estático.

Durante a migração a computação de um serviço está suspensa. Assim, assumimos o tempo de computação (θ) de um serviço como a adição (eq.1) de seu tempo ativo (λ) e seu tempo inativo (δ):

$$\theta \geq \lambda + \delta \qquad \textbf{(eq. 1)}$$

e tomamos como figura de mérito (ε) expressa na eq.2:

$$\varepsilon \leq \lambda/(\lambda + \delta) \qquad \textbf{(eq. 2)}$$

As eq.1 e 2 são desigualdades devido à possibilidade de uma falha de uma máquina ou de uma conexão. De acordo com o expresso anteriormente, num cenário do tipo 1 um agente deve evitar vários níveis de requisições e evitar arrastar outros agentes.

Se considerássemos os agentes executando ao longo de uma trajetória predefinida ou aleatória, a eq.1 seria expressa como um somatório de pares de tempo ativo e inativo, das etapas computacionais individuais (eq.3).

$$\theta_t = \sum_{i=1}^{n} \theta_i \therefore \theta_t \geq \sum_{i=1}^{n} (\lambda_i + \delta_i) \qquad \text{(eq. 3)}$$

Do acima tomamos uma outra figura de mérito (υ):

$$\upsilon = \sqrt{\sum_{i=1}^{n} \theta_i^2 / \theta_t^2} \therefore \upsilon \leq \frac{1}{n} \qquad \text{(eq. 4)}$$

O ajuste da carga de processamento e de comunicação é baseado no tempo de computação de cada conexão cliente/servidor, que chamamos de um *doublet*. Um *doublet* é a conexão de um agente cliente com um agente servidor através de uma interface. A interface estabelece as fronteiras da célula de migração do agente. As cargas de processamento e de comunicação não são tratadas separadamente mas como um todo.

6.2.4. Trabalhos Relacionados

Apresentamos sucintamente algumas iniciativas que fundem gerenciamento distribuído e código móvel, em particular os que estão sendo realizados em Java.

A arquitetura do *Java Dynamic Management Kit* (JDMK) [JDMK98, JMAPI99] consiste de um conjunto de agentes estáticos que recebem *beans* dinamicamente distribuídos pela rede, sendo (des)encaixados no agente, a fim de adicionar, modificar, ou suprimir serviços, tal como componentes de *hardware* que são (des)encaixados em um bastidor. Há uma biblioteca com um núcleo de serviços de gerenciamento implementados como componentes JavaBeans enquanto serviços novos de gerenciamento podem ser criados. Os agentes JDMK podem colaborar diretamente na resolução de problemas, ao invés de tradicionalmente repassá-los a um gerente num nível mais elevado. A inteligência integrada nos dispositivos pretende: reduzir o tráfego de gerenciamento na rede, permitir que problemas de baixo nível sejam tratados localmente sem geração de alarmes, permitir uma resposta mais rápida aos eventos e reduzir os custos de administração. Na arquitetura JDMK os beans são a parte móvel do código visto que os agentes são estáticos. Os benefícios são: integração rápida, compatibilidade, reuso de código, economia de tempo e escalabilidade dinâmica.

O *Object Management Group (OMG)* trabalha atualmente na especificação de uma estrutura para suporte à mobilidade de agentes através da especificação do *MASIF - Mobile Agent System Interoperability Facilities Specification* - [OMG97-2]. O Grasshopper e MAGNA [Krause97, GrassHopper98] são ambientes de agentes móveis inteligentes segundo o padrão OMG/MASIF. Permitem a criação de aplicações distribuídas com agentes se beneficiando de comunicação e acesso a dados localmente a velocidade mais alta. Grasshopper implementa uma facilidade de agentes móveis que poderia ser usada como a estrutura para o desenvolvimento de um sistema de gerenciamento baseado no paradigma de agentes móveis.

Este trabalho usa um *middleware* CORBA e agentes móveis CORBA para gerenciar o *middleware* e as aplicações associadas. Em adição, o *serviço de disponibilidade* para orientar ajustes.

6.3 Arquitetura

Nesta seção apresentamos nossa arquitetura MomentA baseada no modelo OMG/CORBA para o middleware e para as aplicações gerenciadas. Esta arquitetura baseia-se nos esquemas de gerenciamento com componentes móveis mencionados na seção 1. Nós começamos com uma console estática e agentes móveis distribuídos que se conectam aos recursos controlados. Antevemos, uma console móvel e agentes móveis que se conectam a recursos controlados também móveis. A proposta de gerenciamento é orientada basicamente à monitoração da atividade dos objetos no ambiente gerenciado. Os componentes básicos são: um *intermediator de requisições de objetos* (ORB), um *conjunto de agentes de gerenciamento* da atividade dos objetos, um *serviço de disponibilidade* para o suporte de ajustes no domínio e no ambiente controlados e um *suporte para mobilidade*.

6.3.1. Infra-estrutura de Gerenciamento

A infra-estrutura de gerenciamento é composta basicamente de agentes que monitoram a atividade dos objetos. A *monitoração* é *evento-orientada* e baseada em *contabilidade* e *desempenho*. Os *eventos* básicos são as requisições enviadas e recebidas pelos objetos [Schulze98-1, Friderich95]. A abordagem é detectar um problema através de agentes com funcionalidades diferentes e o detalhamento sucessivo do problema. Os agentes coletam informação de um conjunto de *sensores* que interagem com *pontas-de-prova* que interceptam as requisições. Os sensores são usados para monitorar hospedeiros no ambiente com um detalhamento até a granularidade de objetos ativos. Os sensores implementados consideram as métricas apresentadas na Tabela 6.1 e a partir deste ponto também nos referimos aos sensores como filtros. O suporte a mobilidade de agentes usa uma infra-estrutura de desenvolvimento próprio. Nossa infra-estrutura de mobilidade executa agentes móveis como objetos CORBA que se movem de uma agência para outra e se (des)registram no ambiente CORBA na chegada (partida).

Tabela 6.1 Sensores Modelados.

Sensores	Métricas
F0	- número de requisições enviadas (recebidas) - número de respostas - *throughput* total
F1	- tempo total de residência - tempo total de resposta
F2	- taxa de dados enviada por requisição/ resposta - taxa de dados recebida por requisição/ resposta
F3	- tempo de *marshaling* - tempo de *unmarshaling*
F4	- método (operação) mais requisitado
F5	- ocupação de memória e cpu por requisição

6.3.2. Agentes de Gerenciamento

Dois tipos de informação de gerenciamento são considerados: de visão geral com pouca informação coletada em um domínio grande, e de visão mais detalhada em um domínio menor. De acordo com esta abordagem, propomos um conjunto de agentes de dois tipos principais: agentes *em_largura* e agentes *em_profundidade*. Alguns agentes sugeridos são apresentados na Tabela 6.2.

Tabela 6.2 Agentes de gerenciamento utilizando os filtros propostos na Tabela 6.1.

tipo	agentes em_largura (*agw*)	filtro	agentes em_profundidade(*agd*)	filtro
1	requisições e *throughput*	F0	requisições, *throughput* e dados	F0+F2
2	tempos de residência e resposta	F1	tempos	F1+F3
3	requisições e tempos de residência/resposta	F0+F1	objetos	F0+F2+F4
4			cpu e memória	F1+F5
5			geral	F0+F1+F2+F3
	agentes especiais (ags)			
6	ativação de filtros	nenhum		
7	ocupação de cpu e memória (%)	nenhum		

O agente deve ser configurado pelo gerente para refletir as características do ambiente gerenciado, isto é, o tipo de usuário. Por exemplo, um agente *em_largura* pode coletar o *throughput* total e a ocupação de cpu e memória em um grupo de hospedeiros, usando *agws* do tipo 1 e *ags* do tipo 7. O gerente analisa esta informação e decide sobre a emissão de um agente *em_profundidade* ao(s) hospedeiro(s) em aparente situação crítica, a fim de monitorar em detalhes, a exemplo, por processo: o *throughput* e o número de requisições (*agd* tipo 1), o tempo médio de resposta (*agd* tipo 2), e assim por diante. Toda informação coletada é repassada a um gerente que decide sobre etapas adicionais, isto é, emitir outros agentes (*agw* e *agd*), intervir diretamente na realocação de processos, ou permitir a realocação pela infra-estrutura de ajuste. Para cada agente há um outro agente correspondente de ativação de filtros, *ags* do tipo 6. Há sempre ao menos duas etapas a serem seguidas: um agente para ativar os filtros desejados, e os agentes a coletar e trabalhar a informação filtrada.

A estrutura de gerenciamento baseada em agentes está representada na Figura 6.4, com um diagrama geral do sistema. As seguintes seqüências de gerenciamento podem ser observadas: (1,2) enviando um agente *em_largura*; (3) relatando ao gerente (móvel); (4,5) enviando um agente *em_profundidade*; e (6) relatando ao gerente.

Figura 6.4 O macro modelo da estrutura de gerenciamento.

6.3.3. Infra-estrutura de Ajuste

Nesta seção discutimos o uso de um *serviço de Disponibilidade* no contexto de gerenciamento de configuração a fim de permitir ajustes no ambiente gerenciado [Schulze99-1, Vasconcellos98]. O serviço de Disponibilidade intervém fazendo uma contínua migração da computação, de acordo com a demanda e a disponibilidade. Esta intervenção co-existe com os agentes de gerenciamento. Cabe ao gerente permitir este ajuste contínuo ou simplesmente bloqueá-lo. Uma abordagem simples é o bloqueio completo deste ajuste pelo gerente enquanto um refinamento do gerenciamento seria um bloqueio seletivo em determinados hospedeiros ou processos (*clusters*) de objetos.

Até o momento descrevemos como o serviço de Disponibilidade proposto participa em nível dos ajustes. Toda requisição com o propósito de ajuste passa pelo serviço de Disponibilidade local existente em cada hospedeiro. O serviço de Disponibilidade é responsável por encontrar um componente disponível de acordo com o conjunto de restrições passadas na requisição. Se for permitido ao componente se mover no momento da requisição, move-se no sentido dos recursos de prontidão e tem-se uma sintaxe como:

```
requisição (referencia_agente, nome_agente, operação, restrições, QoS)
```

Esta requisição encapsula a recuperação da referência de agente e a requisição da operação:

```
Component_Object_Reference = bind (component_name, constraints, operation_name, QoS);
Component_Object_Reference.operation_name(operation_arguments);
```

A infra-estrutura é uma facilidade adicionada ao ORB e baseada em outros serviços CORBA. É adicionada para ajustar o ambiente controlado, baseado na migração dos componentes e o macro modelo apresentado na Figura 6.5. Neste modelo, dois serviços foram adicionados às facilidades CORBA para permitir migração transparente: o *serviço de Disponibilidade* com uma *interface de Transparência* e o *serviço de Mobilidade*. Ver Capítulo 3, seção 3.4.

6.3.4. Serviço de Mobilidade

É uma facilidade CORBA adicionada ao ORB onde um agente móvel tem a estrutura de um objeto CORBA e visto como tal por qualquer outro agente. O que distingue um agente CORBA de um simples objeto CORBA é a existência de um construtor especial no agente que inicializa a sua parte autônoma do código na instanciação. Toda nova ativação após a instanciação simplesmente recupera o estado do agente e prossegue a execução.

Agentes de CORBA se registram em toda agência destino do itinerário como um objeto CORBA, e a fim de mover-se, solicitam desativação na origem e reativação no destino. Distinguimos um objeto móvel CORBA de um agente móvel CORBA estendendo a descrição anterior, isto é, um objeto se move devido a uma demanda externa enquanto um agente pode mover-se devido às demandas externas e internas.

Definimos previamente mobilidade explícita como a demanda interna de mobilidade que está codificada no agente. Externamente um agente associado poderá se mover em conseqüência do movimento explícito do primeiro agente. Esta mobilidade implícita ocorre ao nível de uma rechamada do primeiro agente em resposta a um requisição do segundo, como representado na Figura 6.3.

O serviço de Mobilidade está descrito no Capítulo 3, seção 3.5. No suporte a mobilidade explícita dos agentes de gerenciamento são usadas estritamente as classes dos diagramas das Figuras 3.18 e 3.19. Para o suporte a mobilidade implícita na infra-estrutura de ajuste, contamos com o serviço de Disponibilidade proposto, em conjunto com o serviço de Mobilidade.

6.4 Aspectos de Implementação

Nesta seção detalhamos alguns aspectos de implementação a respeito do porte de nossa arquitetura orientada a agentes de gerenciamento para um ambiente de computação distribuída baseado em CORBA. A implementação considera Java + JavaORB e as classes org.omg.CORBA disponíveis em *OW3.0* [OrbixNames], *javaIDL - JDK1.2* [Java98-3] e *Vbj3.2* [Visibroker97]. Em adição, temos o serviço de Nomes disponível em javaIDL (org.omg.Naming) e uma versão de Trader [TOI98, Trader95]. Trabalhamos basicamente em Unix (SunOS 5.5) e consideraremos MSWindows mais tarde.

6.4.1. Infra-estrutura de Gerenciamento

O gerenciamento de monitoração está baseado em contabilidade e desempenho. A infra-estrutura de gerenciamento é composta basicamente de um conjunto de agentes que monitoram a atividade dos objetos através das requisições enviadas e recebidas.

A Figura 6.5 esboça a estrutura de filtragem de eventos disponível na OrbixWeb2x [OrbixWeb97-1, OrbixWeb97-2] e como é introduzida nos processos controlados para posterior leitura por um agente de gerenciamento. Os pontos de inserção desta estrutura são: *saída-requisição-pre(pós)-marshal* e *entrada-resposta-pre(pós)-marshal* no lado cliente, e *entrada-*

requisição-pre(pós)-marshal e *saída-resposta-pre(pós)-marshal* no lado do servidor. Em cada ponto é possível cascatear diferentes processamentos do evento. Combinamos estes pontos de inserção com o cascateamento de processamento de eventos para construir nossos filtros.

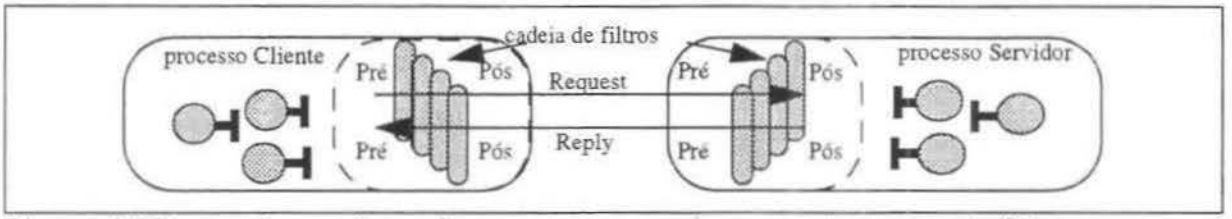
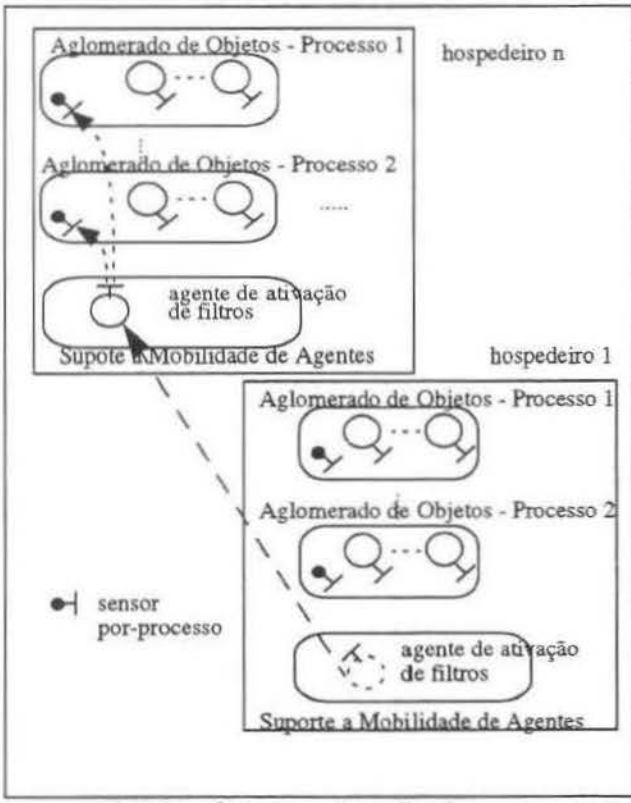


Figura 6.5 Pontos de monitoração em um processo e o cascateamento de filtros.

Do lado esquerdo, na Figura 6.6, estão respresentados os processos aglutinando objetos, em hospedeiros diferentes. Cada processo ou objeto pode ter ativado tipos diferentes de sensores através de um agente móvel. Os sensores escrevem dados em uma estrutura de informação de gerenciamento existente em cada processo. A interface de ativação de filtros em um processo é usada também para leitura da estrutura de informação de gerenciamento. Os agentes móveis são representados movendo-se de um hospedeiro para outro através do suporte a mobilidade de agentes. No lado direito, está a interface IDL com a estrutura de informação de gerenciamento (*dataFilter*) e os métodos que (des)ativam o(s) filtro(s) e lêem a estrutura. A estrutura de informação de gerenciamento suporta todos os filtros da Tabela 6.1.




```
typedef sequence<boolean> seqbool;
interface EncapsFilterProcess {
 struct dataFilter {
 long number_of_request_received;
       long  number_of_request_sent;
       long  number_of_response_received;
       long  number_of_response_sent;
       double residence_time;
       double response_time;
       double rate_of_data_received;
       double rate_of_data_sent;
       double marshaling_time;
       double unmarshaling_time;
       long  memory_occupancy;
       long  cpu_occupancy;
       double throughput;
       char operation_name;
       char marker_name;
       char server_name;
       short activation_mode;
       char principal;
       double startup_time;
       double final_time; };
 void Activate_Deactivate_Proc(in seqbool filtrosproc);
 dataFilter AcessData( );
 };
```

Figura 6.6 Cenário de ativação de sensores e a interface IDL dos filtros.

Agentes de Gerenciamento: Implementamos todos os filtros da Tabela 6.1 e um esqueleto de agente que permite implementar os agentes da Tabela 6.2. Para melhor descrever a implementação de agentes de acordo com a arquitetura descrita na seção anterior, apresentamos um estudo de caso simples e os agentes associados. Para o detalhamento de problemas definimos agentes *em_largura* e *em_profundidade* e um abordagem por-hospedeiro e por-processo. Um processo é um aglomerado de objetos como representado nas Figura 6.5 e 6.6.

Na Figura 6.7 e 6.8 ilustramos *ags* - tipo 6 e tipo 7 respectivamente ilustrando o código de uma implementação de agente.

```
package AgAct;
import ...;
public class AgAct extends BaseAgent
{
...
public AgAct( ) { super( ); ... ; }
public void run( ) {
/* Ativação de Filtro(s) */
_EncapsFilterProcessRef ref1;
_IT_daemonRef daemon;
serverDetailsSeq servers =
new serverDetailsSeq();
seqbool filprocess1 = new seqbool( );
filprocess1.length = 6;
filprocess1.buffer[0] = true;
filprocess1.buffer[1] = false;
filprocess1.buffer[2] = false;
filprocess1.buffer[3] = false;
filprocess1.buffer[4] = false;
filprocess1.buffer[5] = false;
String host = itinerary.buffer[0];
try {
daemon =
IT_daemon._bind("", host);
daemon.listActiveServers(servers);
} catch (... ) {... ; return; }
for(int i=0;i<servers.length;i++) {
try {
srv = servers.buffer[i].server;
ref1 =
EncapsFilterProcess._bind(":" + servers.buffer[i].server, host);
}
} catch (...) { ... ; return; }
}
tinitial = System.currentTimeMillis( );
time = Long.parseLong((String)memory.elementAt(0));
}
```

Figura 6.7 Agente de ativação de filtros, *ags* tipo 6.

No cenário de implementação esperamos detectar um conjunto de hospedeiros críticos e entregar um relatório ao gerente. Cabe ao gerente informar os níveis críticos determinados pelo tipo predominante de aplicação e usuário. Baseado nos relatórios o gerente decide-se por ações subseqüentes dos agentes de gerenciamento. O agente *ags* - tipo 6 ativa os filtros desejados nos

processos dos hospedeiros. Os agentes *ags* - tipo 7 são similares e uma instância diferente é enviada a cada hospedeiro controlado. Devem permanecer lá por um período verificando a ocupação de cpu e memória para serem lidos mais tarde por um agente *em_largura*, *agw*. Um agente *agw* (tipo 1, 2, 3) circula no ambiente gerenciado, avalia os hospedeiros que estão acima de um ponto crítico e emite um relatório ao gerente. Destes hospedeiros críticos resulta um conjunto que pode ser o itinerário de um agente *em_profundidade* subseqüente, *agd* (tipo 1, 2, 3, 4, 5). Os agentes *agw* e *agd* usam combinações diferentes de filtros.

A estrutura de decisão interna de agentes é baseada em níveis, podendo ter valores fixos predefinidos ou médios a partir de mínimos e máximos coletados em uma passagem anterior do agente. Pode-se considerar decisão baseada na combinação de sensores diferentes. Cabe ao gerente enviar outros agentes e/ou realocar processos pela via direta ou pela infra-estrutura de ajuste.

```
package AgCPM;
import ...;
public class AgCM extends BaseAgent
implements _CollectCMOperations {
...
public AgCM( ) { super( ); ... ; }
public void run( ) {
_EncapsFilterProcessRef ref1;
_IT_daemonRef daemon;
String host = itinerary.butter[0];
tinitial = System.currentTimeMillis( );
time = Long.parseLong((String)memory.elementAt(0));
while ( ) {
try {
/* Ler ocupação de cpu *
child1 = exec("sar -u 1 10");
in1 = child1.getInputStream( );
...
in1.close( );
// Esperar o término do subprocesso
try { child1.waitFor( );
} catch ( ... ) { ... }
/* Ler ocupação de memória *
child2 = exec("/usr/ucb/top 0");
in2 = child2.getInputStream( );
...
in2.close( );
try { child2.waitFor( );
} catch ( ... ) { ... ; }

} catch ( ... ) { ... ; }
... ;
}
memory.addElement(host);
memory.addElement(data); }
}
```

Figura 6.8 Agente de medida de ocupação de cpu e memória, *ags* tipo 7.

6.4.2. Infra-estrutura de Ajuste

Os blocos básicos do serviço de Disponibilidade são: um Trader, um serviço de Nomes, um avaliador de recursos dinâmicos, uma interface de transparência e um gerente de ativação. As etapas seguintes ocorrem durante uma requisição a um agente. Envolve a interação do serviço de Disponibilidade local com o Trader, o serviço de Nomes e os agentes, ou agências, requisitadas através do serviço de disponibilidade remoto.

Localização na Cache: O agente requisitado existe na cache local: 1) Requisita um agente; 2) Encontra uma tabela de localização para o agente na cache local. 3) Faz requisições sucessivas; 4) Termina com uma conexão bem sucedida; 5) Retorna a execução ao objeto de aplicação.

Exceção [1]: As agências não sabem sobre o agente requisitado: 1) Registra o agente no conjunto de agências definidas na tabela de localização; 2) Reinicia a localização na cache.

Exceção [2]: Nenhuma agência pode atender a requisição em tempo: 1) Começa a localização no serviço de Nomes.

Exceção [3]: Nenhuma tabela de localização para o agente foi encontrada na cache local: 1) começa a localização no serviço de Nomes (ou salta para a localização no Trader).

Localização no Serviço de Nomes: Há uma referência do agente no serviço de Nomes. O serviço de Nomes é atualizado em cada (des)ativação e reativação.

Exceção [1]: Nenhuma localização para o agente foi encontrada no serviço de Nomes: 1) Começa a localização no Trader.

Localização no Trader: O agente requisitado existe no Trader: 1) Importa a informação do Trader; 2) Copia a tabela de localização na cache local; 3) Reinicia a localização na cache.

Exceção [1]: O Trader sabe sobre o agente mas não sabe sobre a tabela de localização: 1) Solicita informação sobre agências com as características sugeridas; 2) Importa a lista de agências que combinam as características anteriores; 3) Seleciona o número sugerido de agências; 4) Registra o agente no novo conjunto de agências; 5) Reinicia a localização no Trader.

Exceção [2]: O Trader não sabe sobre o agente requisitado: 1) Retorna a execução ao objeto de aplicação para novas instruções.

6.5 Resultados

Nesta seção apresentamos alguns resultados no gerenciamento de processos CORBA baseados na seção de implementação. Usamos um programa de distribuição de carga para simular hospedeiros críticos. A carga distribuída é composta de processos CORBA, onde os processos são aglomerados de objetos. Usamos um domínio de teste com três hospedeiros SunOS5.5 e esperamos que os agentes de gerenciamento detectem os problemas simulados. Consideramos todos os processos gerenciados com uma única *thread* de execução, isto é, uma única requisição é atendida de cada vez. Consideramos todas as requisições com resposta, i.e., nenhuma de sentido único.

Experimentação de Monitoração. Enviamos um agente ags6 para ativar o filtro desejado (F0) e diferentes agentes *ags7* para cada hospedeiro. Este permanece lá e mede a ocupação de cpu e memória. O código para estes agentes está ilustrado na Figura 6.9 e 6.10. Após 4 minutos enviamos um agente *agw1* para navegar pelos hospedeiros, coletando dados e relatando ao gerente quais hospedeiros devem ser inspecionados por um outro agente *agd1*. A avaliação do *agw1* é baseada nos padrões da Tabela 6.3. Durante a primeira passagem os valores são prefixados. Um recurso adicional seria o *agw1* atualizar os níveis para a média dos valores mínimo e máximo coletados na primeira passagem e fazer uma segunda (ou mais) passagem(ns).

Tabela 6.3 Tabela de níveis de decisão para ocupação de cpu, memória e *throughput*.

ocupação de cpu	ocupação de memória	throughput	enviar agente em_profundidade (?)
acima	acima	acima	sim
acima	acima	abaixo	sim
acima	abaixo	acima	não
acima	abaixo	abaixo	sim
abaixo	acima	acima	não
abaixo	acima	abaixo	sim
abaixo	abaixo	acima	não
abaixo	abaixo	abaixo	não

Na Tabela 6.4 apresentamos o relatório retornado por *agw1* e por *agd1*. Do relatório o gerente pode decidir enviar um agente adicional ao hospedeiro 3. Mas pode decidir enviar um agente adicional ao hospedeiro 1, por exemplo, para obter um relatório por-processo baseado num *agd1*. O relatório de *agd1* não sendo conclusivo resulta em um *agd2* e/ou *3*, e assim por diante.

Tabela 6.4 Relatório gerado por um Agente.

Agente: agw1	**Mínimo**	**Máximo**	**Hospedeiro1**	**Hospedeiro2**	**Hospedeiro3**
ocupação de cpu (%)	40	50	40	42	50
ocupação de memória(%)	80	94	80	92	94
throughput	0.0132	0.0198	0.0137	0.0198	0.0132
requisições			1000+1000+160	1000+1000+160	1000+1000+160
enviar outro agente			não	não	sim
Agente: agd1			**Hospedeiro1**	**Processo 1**	**Processo 2**
ocupação de cpu (%)			54		
ocupação de memória(%)			83		
throughput				0.0003833	0.0132
requisições				2000	80
enviar outro agente			sim / não		

Observações. Apresentamos a seguir uma comparação de tráfego de mensagens na rede gerado para cada um dos três primeiros esquemas de gerenciamento da Figura 6.1. Incluímos como mensagem qualquer tipo de acesso remoto. Consideramos a *struct* de cada processo para armazenar a saída dos filtros, como nossa estrutura de informação de gerenciamento. O tráfego (T) de mensagens gerado na rede seria portanto:

Esquema 1 T1 = (estrutura de informação de gerenciamento) * (# hospedeiros) * (# processos)

Esquema 2 T2 = (estado do agente) * (# agentes estáticos) * (# hospedeiros)

Esquema 3 T3 = (código do agente + estado do agente) * (# agentes móveis) * (# hospedeiros +1)

Em princípio, o tráfego de mensagens gerado por agentes estáticos e agentes móveis é comparável se no primeiro, o tempo de *polling* for comparável ao tempo de coleta do agente móvel, por hospedeiro. A vantagem dos agentes móveis é permitir uma comparação no gerenciamento de hospedeiros sem custo extra de mensagens. Com agentes estáticos, esta comparação só é possível a um custo mais elevado de mensagens. Com agentes móveis temos como limitação uma latência maior.

Casos de Ajuste. Descrevemos aqui como o serviço de Diponibilidade participa ao nível dos ajustes no ambiente gerenciado. Qualquer requisição de serviço passa pelo serviço de Diponibilidade local presente em cada agência. O serviço de Diponibilidade é responsável pela localização de um serviço de acordo com um conjunto de propriedades (com valores). Se o serviço solicitante é um componente móvel (por exemplo um agente móvel), ele poderá se mover na direção do componente de serviço solcitado.

Um primeiro exemplo de caso em gerenciamento pode ser um objeto ou agente necessitando mais velocidade de cpu ou de canal de comunicação. O cliente busca por uma agência com

estes recursos para se mover na direção dela. Neste caso, o cliente deve ser móvel e faz uma migração direta na direção do componente alvo.

Um outro exemplo é o de um objeto ou agente que necessita informação de um agente móvel de gerenciamento. O cliente busca pelo agente de gerenciamento requerido e move na sua direção. Entretanto, se o cliente não é móvel, ele não será capaz de se mover e envia uma requisição remota. Quando o agente requisitado está pronto para responder, ele faz uma rechamada ao cliente requisitante e se move na direção dele. Neste caso temos uma migração inversa do alvo na direção cliente iniciador do processo. Em ainda outra situação, o agente requisitado não pode se mover (momentaneamente) e neste caso o cliente requisitante (iniciador) deve decidir por uma execução remota ou uma nova tentativa posterior.

6.6 Comentários

As contribuições da arquitetura MomentA são: gerenciamento de monitoração com agentes móveis para serviços em ambiente CORBA; gerenciamento de configuração com um serviço de disponibilidade; suporte à mobilidade de agentes de gerenciamento; e suporte à migração transparente de componentes.

Os agentes móveis suportados pelo serviço de mobilidade são objetos CORBA, isto é, têm referência de objeto. Quando um objeto CORBA se move de um hospedeiro para outro, sua referência muda. O suporte à mobilidade guarda a nova referência, retornando-a quando solicitado. Agentes CORBA se comunicam como objetos CORBA, facilitando as interações.

O gerenciamento é flexível no sentido de agentes em_largura e em_profundidade refletirem um procedimento de gerenciamento a partir do detalhamento do problema. Domínios, hospedeiros, processos e objetos podem ser a trajetória da análise. O gerenciamento de serviços usa os filtros fornecidos pela OrbixWeb [OrbixWeb97-1, OrbixWeb97-2] como sensores. Estes filtros demonstram ser apropriados para coletar dados e executar contadores e temporizadores. O gerente escolhe os agentes a partir de um conjunto e os envia aos hospedeiros desejados. O agente incorpora parte do processamento a fim de minimizar o tráfego de mensagens e não necessita estar previamente lá podendo ser enviado de maneira assíncrona.

Destacamos que, com agentes móveis, se torna possível tratar gerenciamento de maneira assíncrona e simplificar a descrição dos problemas de gerenciamento. Em adição, é possível basear o gerenciamento na comparação entre hospedeiros sem custo adicional de mensagens que seria necessário num esquema com agentes estáticos. Desdobramentos possíveis deste trabalho são aspectos de gerenciamento de configuração e o gerenciamento de hospedeiros móveis baseado no paradigma de agentes móveis.

A parte do aspecto desempenho a mobilidade de componentes acrescenta fortemente um aspecto de adaptatividade e consequentemente disponibilidade dos serviços oferecidos.

CAPÍTULO 7

Conclusão

Não é triste mudar de idéias,
triste é não ter idéias para mudar.
Barão de Itararé

Neste trabalho apresentamos uma arquitetura de agentes a qual é baseada sobre o modelo CORBA e parcialmente sobre algumas especificações recentes do OMG como o MASIF e o POA. A arquitetura introduz um serviço de Disponibilidade como forma de combinar o uso de componentes fixos e componentes móveis em um ambiente onde componentes disponíveis são utilizados e componentes não disponíveis são configurados e então utilizados. O serviço de Disponibilidade oferece uma interface para a localização e migração de componentes. A parte do aspecto desempenho a mobilidade de componentes acrescenta um forte aspecto de adaptatividade e consequentemente disponibilidade dos serviços oferecidos.

Uma contribuição do trabalho é a apresentação de um modelo com uma requisição genérica de um componente remoto onde o agente iniciador do processo pode ser móvel ou não móvel. Quando o componente requisitado faz uma rechamada ao agente iniciador o procedimento recai em um dos casos anteriores. Conseqüentemente, ocorre ou uma migração do agente iniciador no sentido do componente alvo, ou uma migração do componente alvo no sentido do agente iniciador, ou uma execução remota entre o agente iniciador e componente alvo. Neste sentido estendemos o conceito de mobilidade implementado nas atuais plataformas de agentes móveis [Aglets, Odyssey97, Voyager97-1, GrassHopper98]. Não incluímos na estratégia de migração uma avaliação do tamanho do estado do agente, ficando isto a critério da implementação do agente. Como recomendação de implementação sugerimos o estado ser mantido pequeno, isto é, próximo ao tamanho do código.

Os objetivos principais da estratégia de migração proposta residem na tentativa de migrar componentes seja:

- para obter recursos computacionais que ofereçam uma qualidade de execução (*QoS*) melhor,
- ou minimizar o tráfego de mensagens remotas e portanto mantendo cliente e servidor colocalizados no mesmo hospedeiro ou região.

O serviço de Disponibilidade e a estratégia de migração citados contribuem no gerenciamento de configuração de sistemas distribuídos abertos através de ajuste dos recursos gerenciados. A

migração ou a mobilidade possibilitam reduzir o tráfego de rede, otimizar o balanceamento de carga, e obter menor latência. A mobilidade pode ser considerada como um componente essencial de sistemas abertos do futuro, melhorando largamente a distribuição de *software*, especialmente se a tecnologia funcionar através de múltiplas plataformas. São introduzidas exigências adicionais, particularmente em segurança e interoperabilidade que dificultam o mesmo desempenho de algumas aplicações em código nativo estático.

Duas aplicações de agentes móveis são tratadas no trabalho: uma em gerenciamento e outra em computação móvel. Uma solução baseada no paradigma de mobilidade não é necessariamente melhor que uma baseada em não-mobilidade. Há aplicações que se adaptam melhor a um e outro paradigma, enquanto outras podem se beneficiar da junção de ambos. Atualmente, a maioria de aplicações de gerenciamento são baseadas em agentes estáticos, enquanto novos desenvolvimemtos tem surgido baseados em CORBA com agentes estáticos e móveis.

Na abordagem de agentes móveis proposta, sobre CORBA, introduzimos uma formulação que apresenta a contribuição da parcela relativa a migração e da parcela relativa a comunicação remota. A formulação pretende auxiliar na decisão de qual paradigma (móvel e/ou não-móvel) utilizar.

As aplicações que de uma forma em geral pouco ou nada se beneficiam do uso de mobilidade e de agentes móveis são aquelas que necessitam de muita troca de mensagens para sincronização de processos concorrentes.

Aspectos de Implementação

A estratégia de implementação se baseou em avaliações iniciais explorando e conciliando conceitos novos e tecnologias emergentes. A implementação utiliza ao máximo componentes de *software* disponíveis e compatíveis com o padrão CORBA os quais podem ser atualizados ou substituídos de acordo com os conceitos propostos pela arquitetura. As partes as quais não é possível manter este procedimento deve se restringir àquelas que apesar de padronizadas ainda não dispõem de implementação ou que estão numa fase anterior de proposta à padronização.

Aplicação em Gerenciamento

Uma aplicação de gerenciamento de monitoração baseada em agentes móveis foi implementada [Schulze98-1, Schulze99-2, Schulze99-3] com a introdução de migração transparente de componentes [Schulze99-1] em gerenciamento de configuração baseada no serviço de Disponibilidade.

O gerenciamento pode ser de aplicações ou de agências. O gerenciamento de aplicações está relacionado com o gerenciamento de tarefas. O gerenciamento das agências consiste em controlar a coleção de agências do ponto de vista do fornecedor de serviço e está relacionado com a gerência de sistemas distribuídos. Um serviço requisitando e interagindo com outros serviços remotos é, por si, um tipo de gerenciamento de tarefa.

Níveis podem ser utilizados para uma autorização por etapas no que se refere a monitoração e reconfiguração. O histórico do sistema é importante para novas sugestões de atuação. Conseqüentemente, uma fase de aprendizado sobre o ambiente é necessária seguida de uma fase de execução. A repetição sucessiva de ambas deve manter o sistema atualizado. Pode haver agentes com estratégias diferentes e o mais eficaz dando a resposta mais prontamente.

Um teste de desempenho foi realizado para um ambiente homogêneo com os agentes móveis usando uma implementação *C++* em comum. O desempenho de execução de código interpretado em contraposição a compilado não é necessariamente uma restrição se todo novo componente remoto tem um similar compilado.

Uma extensão deste trabalho em gerenciamento é a utilização da API de gerenciamento presente em JMAPI [JMAPI99] para a confecção de agentes móveis de gerenciamento. O JMAPI é uma extensão da linguagem Java baseada em *JavaBeans*, utilizada em outros desenvolvimentos para gerenciamento [JDMK98]. Outra extensão é a incorporação de agentes móveis (CORBA) às infra-estruturas de gerenciamento baseadas em agentes estáticos e CORBA, como por exemplo, o *IBM Power Broker* [IBM99].

Aplicação em Computação Móvel

É possível manipular um hospedeiro móvel como um hospedeiro usual sendo desativado em um ambiente onde deseja-se sustentar a funcionalidade através da migração dos componentes do hospedeiro móvel. É possível considerar a migração de componentes para outro hospedeiro no ambiente e posterior migração de volta ao hospedeiro móvel. Esta migração é possível através do serviço de Disponibilidade para identificação de um novo destino e do suporte de agentes móveis para despachar os componentes ao destino.

Trabalhos Futuros

Diferentes abordagens são possíveis para um serviço disponibilidade, de acordo com a área de aplicação, por exemplo *desempenho* e *gerenciamento de sistemas distribuídos abertos*. Uma extensão do trabalho é a classificação de interfaces e métodos de disponibilidade para diferentes objetivos. Para balanceamento de carga, o serviço de disponibilidade deve ser rápido e de processamento leve. Por esta razão os testes feitos usam um avaliação intrínseca de carga, baseada no tempo de resposta dos hospedeiros em atender a cada tipo de serviço.

Algumas aplicações científicas em grandes experimentos necessitam funcionar continuamente durante longos períodos. Devem tolerar atualizações constantes do *hardware e software* experimental e mudanças na busca por eventos especiais. Para o gerenciamento destas aplicações é muito importante a possibilidade de atualizar as plataformas ao longo do seu ciclo de vida de forma flexível, substituindo componentes. Um trabalho futuro que se mostra interessante é a integração de agentes em trabalhos anteriores do autor [Schulze96-1/2, 95, 94-1/2, 93, 92-1/2].

Uma vez liberada e em funcionamento, pode ser impróprio tirar uma aplicação de operação para atualização seguida da reinicialização e depuração. Em aplicações baseadas em agentes, cada agente pode ser removido e reiniciado separadamente sem uma interrupção da aplicação inteira. A atividade de atualização e depuração pode ser reduzida aos agentes.

Algumas considerações adicionais podem ser feitas sobre atualizações que demandam replicação, fracionamento ou inclusão de um agente novo no contexto. Isto pode envolver a atualização da interface do agente ou grupo de agentes. No caso de uma fusão, o agente resultante pode herdar todas as interfaces do grupo de agentes.

REFERÊNCIAS

A informação que temos
não é a que desejamos.
A informação que desejamos
não é a que precisamos.
A informação que precisamos
não está disponível.
John Peers

Consiste das referências do autor e demais referências bibliográficas utilizadas no texto, sendo as referências encontradas somente na *Internet* agrupadas em uma seção própria:

- R.1 Do Autor Neste Trabalho, página 113,
- R.2 Outras do Autor, página 114,
- R.3 Demais Autores, página 115 e
- R.4 Da Internet, página 121.

R.1 Do Autor Neste Trabalho

Schulze99-3 B.Schulze, E.R.M.Madeira, P.Ropelatto e F.Vasconcellos. MomentA: Service Management using Mobile Agents in a CORBA Environment, Journal of Network and Systems Management (JNSM), Editor: Dr. M. Malek, 25 pgs., submetido em Maio de 1999.

Schulze99-2 B.Schulze, E.R.M.Madeira e P.Ropelatto. MomentA: Gerenciamento de Serviços usando Agentes Móveis em Ambiente CORBA, Anais do 17º Simpósio Brasileiro de Redes de Computadores SBRC'99, Salvador, Bahia, Brasil, pgs. 665-680, 25-28 de Maio de 1999.

Schulze99-1 B.Schulze e E.R.M.Madeira. Migration Transparency in Agent Systems, Proceedings of the Fourth International Symposium on Autonomous Decentralized Systems - ISADS'99, IEEE Computer Society, Tóquio, Japão, pgs. 320-323, 21-23 de Março de 1999.

Schulze98-1 P.Ropelatto, E.R.M.Madeira, B.Schulze e F.Vasconcellos, Distributed Objects Management in Corba Environment using Mobile Agents, Proceedings of the Conference on Network and Communication System - NCS'98, Iasted, Pittsburgh - EUA, pgs.13-16, Maio de 1998.

Schulze98-2 B.Schulze e E.R.M.Madeira. Transparent Service Mobility Using CORBA, Relatório Técnico IC-98-27, 23 pgs., Unicamp, Campinas - SP, Agosto de 1998.

Schulze98-3 B.Schulze, E.R.M.Madeira, F.Assis Silva, S.Choy, I.Busse e S.Covaci. Service Migration and a Transparency Service, Relatório Técnico IC-98-28, 15 pgs., Unicamp, Campinas - SP, Agosto de 1998.

Schulze98-4 B.Schulze e E.R.M. Madeira, MomentA: Mobilidade e Gerenciamento com Agentes em CORBA, Palestras do Objetos Distribuídos - OD'98, Curitiba - PR, 1-4 de Dezembro de 1998.

Schulze97-1 B.Schulze e E.R.M.Madeira, Contracting and Moving Agents in Distributed Applications Based on a Service-Oriented Architecture, Lecture Notes in Computer Science - LNCS no 1219 - Mobile Agents, First International Workshop, MA'97, Berlim - Alemanha, Proceedings, Springer-Verlag, pgs.74-85, Abril de 1997.

Schulze97-2 B.Schulze e Edmundo R.M. Madeira, Using an availability service for transparent service migration in mobile computing, Relatório Técnico IC-97-11, 13 pgs., Unicamp, Campinas - SP, Agosto de 1997.

Schulze96 B.Schulze e Edmundo R.M. Madeira, Contracting and moving agents in distributed applications based on a service-oriented architecture, Relatório Técnico IC-96-20, 11 pgs., UNICAMP, SP, Brasil, Dezembro de 1996.

R.2 Outras do Autor

Schulze96-1 V.Bocci, A.Branco, J.Buytaert, S.Cairanti, V.Chorowicz, F.Formenti, L.Gorn, M.Oesterle, U Rossi, T.Rovelli, B. Schulze, Z.Tomsa, G.Valenti , PAND2 Test Results, Delphi Note 96-16 DAS 172, CERN, Suíça, 19 de fevereiro de 1996. (delphiwww.cern.ch/~pubxx/delwww/delsec/delnote/private/96_16_das_172.ps.gz)

Schulze96-2 R.Schulze, B.Schulze, The DELPHI Run DataBase Presenter, Delphi Note 94-2 DAS 146, CERN, Suíça, 4 de fevereiro de 1994. (delphiwww.cern.ch/~pubxx/delwww/delsec/delnote/private/94_2_das_146.ps.gz)

Schulze95 DELPHI Trigger Group, Architecture and performance of the DELPHI trigger system, Nuclear Instruments and Methods in Physics Research A 362 (1995) 361-385.

Schulze94-1 DELPHI Trigger Group. *Basic concepts and architectural details of the DELPHI trigger system,* 1994 IEEE Nuclear Science Symposium and Medical Imaging Conference (NCS/MIC), Norfolk, Virgínia - EUA, 30 de Outubro a 5 de Novembro de 1994. (Artigo selecionado para o IEEE Transactions on Nuclear Science, Vol.42 No.4 August 1995, 837-843).

Schulze94-2 B.Schulze, A.Branco, J.Buytaert, S.Cairanti, F.Formenti e G.Valenti, *A test system for the Local Trigger Supervisor of the DELPHI experiment at LEP*, Int. Conference on Computing in High Energy Physics - CHEP'94, S.Francisco, Califórnia - EUA, 21-27 de Abril, 1994, LBL-35822 (CONF-940492) UC-405 pgs. 510-512.

Schulze93 DELPHI Collaboration, Determination of alpha S using the Next-to-Leading-Log Approximation of QCD Delphi Internal Number: 0058, PPE Number: 93--043 Zeit. Phys. C59 (1993) 21.

Schulze92-1 M.Miranda, C.Barros, M.Mendes, A.Nigri, E.Paiva, A.Santoro, B.Schulze, C.Silva R.Valois, H.Areti, J.Biel, A.Cook, J.Deppe, M.Edel, M.Fishler, I.Gaines, M.Gao, B.Haynes, D.Husby, M.Isley, T.Nash, T.Zmuda, RISC Architecture Microprocessor Farm for off-line analysis, CERN Report 92-07, Proceedings of the International Conference on Computing in High Energy Physics'92, pgs. 467-470, Annecy, França, 21-25 de Setembro de 1992.

Schulze92-2 DELPHI Trigger Group, J.Valls et al., *Architecture and performance of the DELPHI trigger system*, 1992 IEEE Nuclear Science Symposium and Medical Imaging Conference (NCS/MIC), Orlando, Flórida - EUA, 25-32 de Outubro de 1992. (Delphi Note 92-162 DAS 135)

R.3 Demais Autores

APM93 APM. An Overview ANSAware 4.1. Architecture Projects Management Ltd., Cambridge UK, 1993.

Baldi98 M.Baldi e G.P. Picco, Evaluating the Tradeoffs of Mobile Code Design Paradigms in Network Management Applications, 20th InternationalConference on Software Engineering (ICSE '98), Kyoto, Japão, Abril de 1998. (http://www.polito.it/~baldi/curriculum/curriculum.htm#conference)

Bearman96 M.Bearman. Tutorial on ODP Trading Function, DSTC, Faculty of Information Sciences & Engineering, University of Camberra, Austrália, revisado Junho de 1996.

Bichler98 M.Bichler e A.Segev. A Brokerage Framework for Internet Commerce, Fisher Center for Management & Information Technology, Walter A. Hass School of Business, University of California, Berkeley, Estados Unidos, CMIT Working Paper 98-WP-1031, 1998. (hass.berkeley.edu/~cmit/OFFER)

Bigus98 Joseph P.Bigus e Jennifer Bigus, Constructing Intelligent Agents with Java. Wiley & Sons, 1998.

Booch94 G.Booch, Object-Oriented Analysis and Design with Applications. The Benjamin/Cummins Publishing Company, Inc., 1994.

Cardozo E.Cardozo, J.S.Sichman e Y.Demazeau, Using the Active Object Model to Implement Multi-Agents Systems, Proc.of the 5th IEEE Conf.on Tools with Artificial Intelligence, Boston, Estados Unidos, 1993.

Ciupke96 O.Ciupke, D.A.Kottmann e H-D.Walter, Object Migration in Non-Monolithic Distributed Applications, Proc.16th Int. Conf. on Distributed Computing and Systems, Hong Kong, pgs.529-536, Maio 27-30, 1996.

CORBA98 The Common Object Request Broker: Architecture and Specification. Object Managemnt Group, 1998.

CommerceNet97 Catalogs for Digital Marketplace, CommerceNet Research Report, 1997. (www.commerce.net/about/membership/rrsample.hmtl)

Cristian91 F. Cristian, Understanding Fault-Tolerant Distributed Systems, Commun. ACM 32(2):56-78, Fevereiro de 1991.

Cristian93 F. Cristian, Automatic Reconfiguration in the Presence of Failures, Software Engineering Journal, IEE and British Computer Society, pgs. 53-60, Março de 1993.

Cristian94 F. Cristian, Abstractions for Fault-Tolerance, 13th IFIP World Computer Congress, 28 de Agosto - 2 de Setembro, Hamburgo - Alemanha, 1994.

Drummond96 R.Drummond, C.Gonçalves e C.Furuti. LegoShell: A Visual Language for Distributed Objects, A-HAND, IC/Unicamp, Abril de 1996. (www.ahand.unicamp.br/papers/artigosAHAND.p.html)

Fischer95 G.Fischer, Rethinking and Reinventing Artificial Intelligence from the Perspective of Human-Centered Computational Artifacts, Lecture Notes in Artificial Intelligence, #991 Springer, Outubro de 1995, pgs. 1-11.

Franklin96 S.Franklin e A.Graesser, Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents, Proc.of the 3rd International Workshop on Agent Theories, Architectures, and Languages, Springer, 1996. (http://www.msci.memphis.edu/~franklin/AgentProg.html)

Friderich95 R.Friedrich et al., Standardized Performance Instrumentation and Interface Specification for Monitoring DCE based Application. OSF DCE RFC 33.0, 1995.

Genesereth94 M.R.Genesereth e S.P.Ketchpel, Software Agents . Commun. ACM 37(7), 1994.

Goldszmidt96 G.S.Goldszmidt, Distributed Management by Delegation, PhD Thesis, Graduate School of Arts and Sciences, Columbia University, 1996.

Golubski96 W.Golubski, D.Lammers e W-M.Lippe. Theoretical and Empirical Results on Dynamic Load Balancing in an Object-Based Distributed Environment, Proceedings: 16th International Conference on Distributed Computing and Systems, Hong Kong, pgs.537-544, 27-30 de Maio de 1996.

Guedes97 L.A.Guedes, P.C.Oliveira e E.Cardozo. An Agent-based Approach for Quality of Service Negotiation and Management in Distributed Multimedia Systems, Lecture Notes in Computer Science - LNCS no 1219 - Mobile Agents, First International Workshop, MA'97, Berlim - Alemanha, Proceedings, Springer-Verlag, pgs., Abril de 1997.

Guedes97-1 L.A.Guedes e E.Cardozo. Especificação de um Protocolo para Negociação de Qualidade de Serviço em Sistemas Multimídia, 15º Simpósio Brasileiro de Redes de Computadores, São Carlos - SP, Maio de 1997.

Guedes97-2 L.A.Guedes, P.C.Oliveira, L.F.Faina e E.Cardozo. QoS Agency: An Agent-based Architecture for Supporting Quality of Service in Distributed Multimedia Systems, IEEE Conference on Protocols for Multimedia Systems - Multimedia Networking - PROMSMmNet'97, Santiago, Chile, Novembro de 1997.

Haggerty98 P.Haggerty e K.Seetharaman . The Benefits of CORBA-Based Network Management. Commun. ACM 41(10), Outubro de 1998.

Java96 OSF, Java Mobile Code Paper, 15 de Janeiro de 1996. (http://www.gr.osf.org/projects/web/java/whit_pap.htm)

Java97 M.Wutka. Hacking Java: The Professional's Resource Kit, Que Corporation, 1997.

Java98-1 C.S.Horstmann e G.Cornell, Core JAVA 1.1, Volume I - Fundamentals. Sun Microsystems, The SunSoft Press Java Series, Printice Hall, 1998, ISBN 0-13-766957-7.

Java98-2 C.S.Horstmann e G.Cornell, Core JAVA 1.1, Volume II - Advanced Features. Sun Microsystems, The SunSoft Press Java Series, Printice Hall, 1998, ISBN 0-13-766965-8.

Javachip96 Sun Microsystems Inc., SunnyVale, CA, Estados Unidos, 2 de Fevereiro de 1996. (http://www.sun.com/sparc/newsreleases/nr95-042.html)

Johnston97 W.E.Johnston e S.Sachs. Distributed, Collaboratory Experiment Environments (DCEE) Program1: Overview and Final Report, Fevereiro de 1997. (www-itg.lbl.gov/DCEEpage/DCEE_Overview.html#1021081)

Kouzes96 R.T.Kouzes, J.Myers e W.A.Wulf . Collaboratories: Doing Science On The Internet, IEEE Computer, Vol.29 #8, Agosto de 1996.

Kramer85 J.Kramer e J.Magee. Dynamic configuration for distributed systems, IEEE Transactions on Software Engineering, SE-11 (4), Abril 1985.

Kramer90 J.Kramer. Configuration Programming - A framework for the development of Distributed Systems, Proc. IEEE COMPEURO'90, Tel-Aviv, Israel, pgs. 374-384, Maio de 1990.

Kramer92 J.Kramer, J.Magee e M.Sloman. Configuring Distributed Systems, Proc.5th ACM SIGOPS Workshop on Models and Paradigms for Distributed Systems Structuring, Setembro de 1992.

Krause96 S.Krause e T.Magedanz. Mobile Service Agents enabling "Intelligence on Demand" in Telecommunications, IEEE GLOBECOM'96, Londres, Reino Unido, pgs. 78-84, Novembro de 1996.

Krause97 Krause S. et al. MAGNA - A DPE-based Platform for Mobile Agents in Electronic Service Markets, 3rd Int. Symposium on Autonomous Decentralized Systems - ISADS'97, Berlim, Alemanha, pgs. 93- 102, 1997.

Iglesias96 C.Iglesias, J.C.Gonzalez e J.R.Velasco, MIX: A General Purpose Multiagent Architecture, Lecture Notes in Artificial Intelligence, #1037 Springer, 1996, pgs. 251-266.

Lange D.B.Lange e D.T.Chang. Aglets Workbench, IBM Corporation. (http://www.ibm.co.jp/trl/aglets)

Lima95 L.A.P.Lima Jr. e E.R.M.Madeira. A Model for a Federative Trader, Open Distributed Processing: Experiences with Distributed Environment, pgs.173-184, Chapman&Hall, 1995.

Loyolla94 W.P.C.Loyolla, E.R.M.Madeira, M.JMendes, E.Cardozo e M.F.Magalhães. Multiware Platform: An Open Distributed Environment for Multimedia Cooperative Applications, IEEE Computer Software & Applications Conference, COMPSAC'94, Taipei, Taiwan, Novembro de 1994.

Nuttall94 M.Nuttall. Survey of systems providing process or object migration, Technical Report Doc. 9410, Dept. of Computing, Imperial College, Reino Unido, Maio de 1994.

Oliveira98 P.C.Oliveira, L.A.Guedes e E.Cardozo. Monitoramento de Qualidade de Serviço em Sistemas Multimídia Distribuídos: um Estudo de Caso para Sistemas Baseados em Agentes Móveis, 16º Simpósio Brasileiro de Redes de Computadores - SBRC'98, Rio de Janeiro - RJ, 25-28 de Maio de 1998, pgs. 481-500.

OMG98-1 ORB Portability IDL/Java Language Mapping. OMG TC Document orbos/98-01-06, 1998.

OMG98-2 Fault Tolerant CORBA. OMG TC Document orbos/98-03-05, 1998.

OMG98-3 Java Language to IDL Mapping. OMG TC Document orbos/98-07-19, 1998.

OMG98-4 Object by Value. OMG TC Document orbos/98-01-18, 1998.

OMG98-5 Portable Object Adaptor. OMG TC Document orbos/98-07-19, 1998.

OMG97-1 CORBA Component Model: Initial Submission. OMG Document orbos/97-11-07, 1997.

OMG97-2 Mobile Agent System Interoperability Facilities Specification. Object Managemnt Group, OMG Document: orbos/97-10-05, Novembro 10, 1997.

OMG97-3 CORBA Services. Object Managemnt Group, 1997.

OMG96 OMG Trader Object Service, RFP5 Submission, 10 de Maio de 1996.

OMG95-1 Common Facilities Architecture, Revision 4.0, OMG Document orbos 95-1-2, 3 de Janeiro de 1995.

OMG95-2 OMG, Object Startup Service, IBM submission, OMG TC Document 95.10.7, 1995.

Orbix96 OrbixTM Programmer's Guide. Iona Technologies PLC, 1996.

OrbixWeb97-1 OrbixWebTM Programmer's Guide. Iona Technologies PLC, Novembro de 1997.

OrbixWeb97-2 OrbixWebTM Programmer's Reference. Iona Technologies PLC, Novembro de 1997.

Oliveira97 L.AG.Oliveira, P.C.Oliveira e E.Cardozo. An Agent-Based Approach for Quality of Service Negotiation and Management in Distributed Multimedia Systems, Mobile Agents - MA97, LNCS #1219, Springer-Verlag, pgs. 1-12, 1997.

Orfali96 R.Orfali, Dan Harkey e J.Edwards. The Essential Distributed Objects Survival Guide, John Wiley & Sons, 1996.

Orfali98 R.Orfali e Dan Harkey. Client/Server Programming with CORBA and JAVA 2nd Ed., John Wiley & Sons, 1998.

Queiroz97 A Queiroz e E.R.M.Madeira, Management of CORBA objects monitoring for the Multiware platform proceedings of the ICODP'97, Toronto, Canadá, Maio de 1997, pgs. 122-133.

RMODP95 ODP Reference Model, 1995. (ftp://ftp.dstc.edu.au/pub/arch/RM-ODP)

Rodríguez99 E.J.Rodríguez, Uma Modelagem para Comércio Eletrônico usando CORBA e Agentes Móveis, Tese de Mestrado, Instituto de Computação - Unicamp, Fevereiro de 1999.

Rothermel97 K.Rothermel e R.Popescu-Zeletin, (Eds.), Mobile Agents: 1st International Workshop MA'97, volume 1219 of LNCS. Springer, Abril de 1997.

Rozier88 M.Rozier, V.Abrossimov, F.Armand, I.Boule, M.Gien, M.Guillemont, F.Hermann, C.Kaiser, P.leonard, S.Langlois e W. Neuhauser, Chorus Distributed Operating Systems. Computing Systems, 1:305-379, Outubro de 1988.

Rumbaugh91 J.Rumbaugh, M.Blaha, W.Premerlani, F.Eddy e W.Lorensen. Object-Oriented Modeling and Design, Prentice Hall, 1991.

Russel95 S.Russel e P.Norvig. Artificial Intelligence, A Modern Approach, Prentice Hall Series in Artificial Intelligence, New Jersey, Estados Unidos, 1995, pr. 33.

Sichman95 J. S.Sichman e Y.Demazeau. Exploiting Social Reasoning to Enhance Adaptation in Open Multi-Agent Systems, Lecture Notes in Artificial Intelligence, #991 Springer, Outubro de 1995, pgs. 253-263.

Schmidt95 D.C.Schmidt, T.Harrison e E.Al-Shaer. Object Oriented Components for High-speed Network Programming, Proc.of the USENIX Conf. on Object-Oriented Technologies, Monterey, CA, Estados Unidos, Junho de 1995. (http://siesta.cs.wustl.edu/~schmidt/publications.html)

Sloman89 M.Sloman, J.Kramer e J.Magee. Configuration Support for System Description, Construction and Evolution, Proc.5th Int.Workshop on Software Specification and Design, Pittsburgh, Estados Unidos, Maio de 1989.

Stallings93 W.Stallings . SNMP, SNMPv2 and CMIP: The Practical Guide to Network-Management Standards. Addison-Wesley, 1993.

Trader95 Trading Functions, ISO/IEC JTC1/SC 21, 20 de Junho de 1995. (ftp:// ftp.dstc.edu.au/pub/arch/RM-ODP)

Vasconcellos98 F.J.S.Vasconcellos e E.R.M.Madeira, Projeto e Desenvolvimento de um Suporte a Agentes Móveis baseado em CORBA, Anais do 16o SBRC, Rio de Janeiro, RJ, Brasil, Maio de 1998, pgs. 501-520.

Vigna98 G.Vigna, Mobile Code Technologies, Paradigms, and Applications, Tesi di Dotorado, Politécnico di Milano, Milão, Itália, 1998. (www.polimi.it/)

Vinoski93 S.Vinoski, Distributed Object Computing With CORBA, C++ Report Magazine, Julho / Agosto de 1993.

Visibroker97 Visibroker for Java Programmer's *Guide, Release 3.0.* Visigenic Software, Inc., 1997.

Vogel98 A.Vogel e K.Duddy. Java Programming with CORBA 2nd Edition, John Wiley & Sons, 1998.

R.4 Da Internet

Aglets Aglets Workbench. IBM Corporation. (www.trl.ibm.co.jp/aglets)

AGREV97 S.Green, L.Hurts, B.Nangle, P.Cunningham, F.Somers e R.Evans, Software Agents: A review, 27 de Maio de 1997. (www.cs.tcd.ie/research_groups/aig/iag/toplevel2.html)

Expertsoft Tutorial on Distributed Objects. Power Broker CORBAplus, Expertsoft Corporation. (www.omg.org/news/begin.htm)

GrassHopper98 Grasshopper: An Intelligent Mobile Agent Platform. IKV++ GmbH, 1998. (www.ikv.de/products/grasshopper/grasshopper.html)

IBM99 IBM Component Broker: Introducing Component Broker, IBM Corporation, Março de 1999. (www.ibm.com/)

Java98-3 JavaTM Development Kit - JDK 1.2. Sun Microsystems, 1998. (Inc, java.sun.com/products/jdk/1.2/index.html)

JDMK98 Java Dynamic Management Kit - JDMK. Sun MicroSystems, 1998. (www.sun.com/software/java-dynamic/ds-jdmk.html)

JMAPI99 Java Management API, Sun MicroSystems, java.sun.com/products/JavaManagement/index.html, Abril, 1999.

Odyssey97 Introduction to the Odyssey API, General Magic, 1997. (www.genmagic.com/agents/odysseyIntro.ps)

OrbixNames OrbixNames Programming Guide, Iona Technologies, Dublin, Ireland. (www-usa.iona.com/support/kb/OrbixNames/articles/604.482.html)

TOI98 TOI: OMG Trading Object Service, IKV++ GmbH, 1998. (www.ikv.de/products/toi.html)

Voyager97-1 Object Space Core Package Technical Overview, Dezembro de 1997. (www.objectspace.com/product/voyager/white/VoyagerTechOverview.pdf)

Voyager97-2 Voyager CORBA Integration Technical Overview, Dezembro de 1997. (www.objectspace.com/developers/voyager/white/VoyagerCORBAIntegrationW97.pdf)

APÊNDICE A

CORBA em JAVA

Antes de começar o trabalho de modificar o mundo, dê três voltas dentro de sua casa.
Provérbio Chinês

Este Apêndice compreende alguns detalhes sobre CORBA em *Java*, particularmente em duas implementações: *OrbixWeb* e *javaIDL*. A implementação padronizada de *javaORBs* tem sido disponibilizada na forma das classes *org.omg.(CORBA e Naming)*. A seções estão organizadas em:

- A.1 CORBA em Java: OrbixWeb e javaIDL, página 123,
- A.2 Inicialização do ORB, página 127,
- A.3 Serviço de Nomes, página 130 e
- A.4 IDL do Serviço de Nomes, página 133.

A.1 CORBA em Java: OrbixWeb e javaIDL

CORBA é a arquitetura de objetos distribuídos padronizada pelo consórcio OMG [CORBA98]. Esta arquitetura permite que objetos invoquem outros objetos sem saber onde residem e/ou em que linguagem estão implementados. A linguagem IDL (*Linguagem de Definição de Interface*) é usada para definir as interfaces dos objetos. Objetos CORBA diferem de outros modelos de objetos por poderem:

- estar situados em qualquer lugar em uma rede;
- interoperar com objetos em outras plataformas;
- ser escritos em qualquer linguagem de programação para a qual haja mapeamento a partir de IDL, atualmente: *Java*, *C++*, *C*, *Smalltalk*, *COBOL* e *Ada*.

Os conceitos introduzidos pelo modelo OMG/CORBA podem ser obtidos em detalhes na especificação CORBA/IIOP [CORBA98]. O diagrama da Figura A.1 mostra uma requisição de método enviada de um cliente para uma implementação de objeto servidor CORBA. A Figura A.2 mostra interfaces do ORB em uma comunicação entre objetos servidores ou serventes.

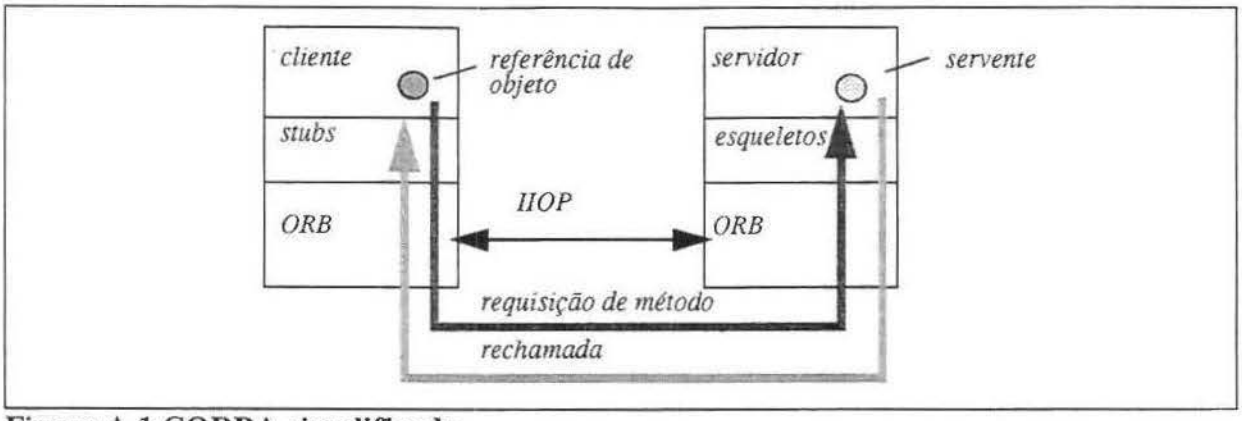


Figura A.1 CORBA simplificado.

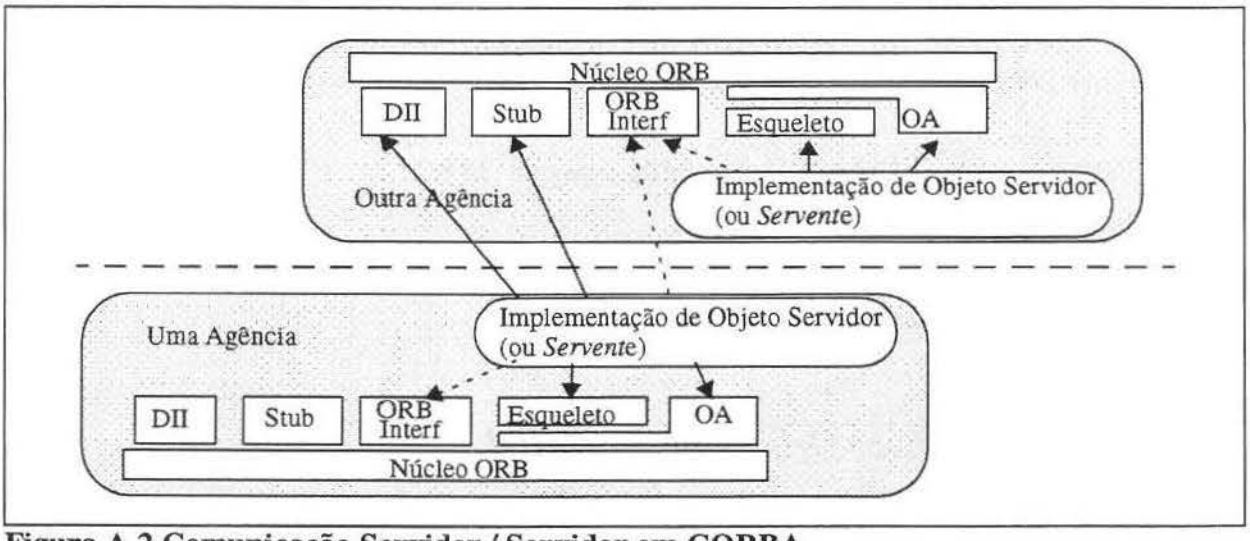


Figura A.2 Comunicação Servidor / Servidor em CORBA.

API: Uma etapa importante na geração de uma aplicação distribuída é a definição das interfaces pelas quais clientes e servidores se comunicarão. As interfaces IDL são interfaces formalmente padronizadas que definem os serviços fornecidos e como os clientes os invocarão.

IDL: O compilador IDL usa um arquivo *.idl* para gerar os *stubs* de cliente e os *esqueletos* de servidor. Se cliente e servidor estiverem em plataformas diferentes, então o mesmo IDL será compilado em cada plataforma gerando o código apropriado de cliente e servidor respectivamente.

Stub & Esqueleto: Os *stubs* são usados pelo cliente como invocações locais de método do objeto remoto, enquanto do lado do servidor estes métodos, isto é, *esqueletos* devem ser implementados para produzir o resultado desejado. A elaboração do servidor pode ser completamente independente da elaboração do cliente.

Invocação Dinâmica: Uma invocação dinâmica de método estabelece uma comunicação entre cliente e servidor iniciada transparentemente pelo *daemon (OrbxiWeb)*. Se o hospedeiro implementa o servidor em particular, o *daemon* estabelece a conexão com o servidor, passa a conexão ao hospedeiro cliente e sai do processo. A interoperabilidade entre Cliente e Servidor é estabelecida enquanto a localização e a natureza do hospedeiro do servidor são mantidas transparentes. Ao *daemon* compete somente a disponibilidade dos servidores e a conexão Cliente / Servidor, sendo parte do ORB.

Adaptador de Objetos: Dentre as funções do adaptador de objetos estão: publicar a interface de um objeto servidor (ou *servant*) no *Repositório de Implementação*, remover a interface, listar todas as interfaces registradas, listar detalhes de uma interface, mostrar o estado ativo de um servidor e desativar um servidor. Na Orbix/OrbixWeb, um *localizador* está disponível para localização transparente de um servidor, através da comunicação entre *daemons*. É também possível associar um conjunto de interfaces a um *grupo* ou criar um grupo de hospedeiros.

Um cliente pode ser um objeto CORBA (ou não CORBA) que invoca um método de outro objeto CORBA. O *servente* é uma instância da implementação do objeto servidor. O cliente de um objeto CORBA usa uma *referência de objeto* para enviar as requisições de método(s). Se o objeto servidor for remoto, a *referência de objeto* aponta para um *stub*, que usa o ORB para enviar invocações ao objeto servidor. Um *stub* usa o ORB para identificar a máquina que executa o objeto servidor e solicita ao ORB desta máquina uma conexão ao objeto servidor. Quando o *stub* obtém a conexão, envia a referência de objeto e os parâmetros ao *esqueleto* ligado a implementação de objeto destino. O *esqueleto* transforma a chamada e os parâmetros no formato requerido específico da implementação e chama o objeto. Todos os resultados ou exceções são retornados pelo mesmo trajeto.

O cliente não tem nenhum conhecimento da localização do objeto CORBA, detalhes de implementação, nem qual ORB é usado para acessar o objeto. ORBs diferentes comunicam-se através do protocolo *Internet InterORB* (IIOP). Um cliente pode somente invocar os métodos que estão especificados na interface do objeto CORBA. A interface de um objeto CORBA é definida em *IDL*. Uma interface define um tipo de objeto e um conjunto de métodos e parâmetros, assim como os tipos de exceção que estes métodos podem retornar. Um compilador IDL traduz as definições do objeto CORBA em uma linguagem de programação específica de acordo com o mapeamento da linguagem. Para Java, o compilador IDL traduz as definições IDL em construções Java de acordo com o mapeamento IDL-Java. Os códigos de *stubs* e *esqueletos* são geradas pelo compilador IDL para cada objeto. Um *stub* implementa pelo lado do cliente o acesso aos métodos remotos, na linguagem de programação do cliente. Um *esqueleto* liga a implementação de objeto ao executável do ORB. O ORB usa um *esqueleto* para despachar os métodos às instâncias de implementação de serventes.

O desenvolvimento de objetos CORBA inclui a criação e o registro de um objeto servidor, ou simplesmente chamado *servente*. Um processo servidor é um programa que contém a implementação de um ou mais serventes e que foi registrado com o ORB. Todo objeto CORBA suporta uma *interface IDL* que define o objeto. Uma interface pode herdar uma ou mais interfaces. A sintaxe IDL é muito semelhante a de Java ou de C++, e um arquivo IDL é funcionalmente análogo a um arquivo *header* de C++. Para cada interface IDL, o compilador IDL gera

uma interface Java e arquivos *.java* associados, incluindo um *stub* de cliente e um *esqueleto* de servidor. Uma interface IDL declara o conjunto de métodos disponíveis ao cliente, o conjunto de exceções e de atributos tipados. Cada método tem uma assinatura que inclui *nome*, *parâmetros*, *resultados*, e *exceções*. Uma interface IDL simples que descreve uma aplicação genérica, é apresentada a seguir:

```
module AplicacaoGenerica {
    interface Aplicacao {
        string mensagem();
    };
};
```

Um método pode levantar uma exceção quando uma condição de erro ocorre. O tipo de exceção indica o tipo de erro que foi encontrado. Os clientes devem estar preparados para tratar: exceções CORBA e exceções definidas para cada método, além dos resultados normais. Uma vez que as interfaces IDL foram definidas e o compilador executado sobre arquivo *.idl*, os arquivos *.java* que contêm as implementações dos métodos podem ser editados. Os arquivos de implementação em java são então compilados e ligados à biblioteca do ORB para criar um objeto servidor.

Uma *implementação de objeto* define o comportamento de todas as operações e atributos da interface que ele sustenta. Pode haver implementações múltiplas de uma interface. A implementação define o comportamento da interface e da criação e destruição do objeto. Somente serventes criam novos objetos CORBA, e portanto uma interface *factory object* deve ser definida e implementada para cada objeto. Por exemplo, se *Document* for um tipo de objeto, então um tipo de objeto *DocumentFactory* com um método *criar* deve ser definido e implementado como parte do servidor. O método *criar* não é reservado e qualquer nome de método pode ser usado. A implementação do método *criar* pode então usar *new* para criar o objeto. Por exemplo:

```
DocumentServant document = new DocumentServant();
orb.connect(document);
```

Um método *destruir* pode ser definido e implementado em *Document*; ou, o objeto pode ser previsto para persistir indefinidamente. O método *destruir* também não é reservado e qualquer nome pode ser usado.

Os *clientes* podem criar objetos CORBA somente através das interfaces publicadas de *factory* que o servidor fornece. Do mesmo modo, um cliente pode somente remover um objeto CORBA se esse objeto publicar um método de destruição. Como um objeto CORBA pode ser compartilhado por vários clientes em uma rede, somente o objeto servidor está em condição de saber quando o objeto pode ser removido. Uma forma do cliente enviar requisições a métodos de um objeto CORBA é através de sua *referência de objeto*. Uma referência de objeto interoperável (*IOR*) é uma estrutura opaca que identifica o hospedeiro de um objeto CORBA, a porta em que o servidor está escutando as requisições, e um ponteiro ao objeto específico no processo. No caso de objetos transientes, esta referência de objeto torna-se inválida se o processo servidor for removido e reiniciado. Os clientes obtêm tipicamente referências de objeto das seguintes maneiras:

- de um *factory object*. Por exemplo, o cliente invoca um método *criar* no objeto *DocumentFactory* a fim de criar um *new Document*. O método *criar DocumentFactory* retornaria ao cliente uma referência de objeto para *Document*;
- de um *servidor de nomes*. Por exemplo, o cliente obtém uma referência de objeto para *DocumentFactory* enviando uma requisição ao servidor de nomes;
- de uma *string* criada especialmente a partir de uma referência de objeto.

Uma vez que uma referência de objeto é obtida, o cliente deve *estreitá-la* (*narrow*) ao tipo apropriado. IDL suporta herança onde a raiz de sua hierarquia é *Object* em IDL e *org.omg.CORBA.Object* em Java. *org.omg.CORBA.Object* é uma subclasse de *java.lang.Object*. Algumas operações como *name lookup* e *unstringifying* retornam um *org.omg.CORBA.Object*, que é estreitado, usando uma classe *helper* gerada pelo compilador IDL, ao tipo derivado desejado. Os objetos CORBA devem ser estreitados explicitamente porque o executável Java não pode saber o tipo exato de um objeto CORBA.

javaORB. *JavaIDL* é um javaORB disponível no JDK 1.2 e com o compilador *idltojava*. Pode ser usado para definir, executar e acessar objetos CORBA na linguagem de programação Java. É uma implementação em conformidade com a especificação CORBA/IIOP 2.0 [CORBA98] e o mapeamento IDL-to-Java [OMG98-3, OMG98-1]. Suporta objetos transientes cujo tempo de vida é limitado pelo tempo de vida do seu processo servidor. JavaIDL fornece também um Servidor de Nomes transiente para armazenar *nomes* e *referências de objetos* em uma estrutura de árvore-diretório. A versão atual de JavaIdl não implementa um *Repositório de Interfaces* nem um Gerente de Ativação. A implementação atual requer o Servidor de Nomes já executando com *hospedeiro* e *porta* definidos pelas propriedades *ORBInitialHost* e *ORBInitialPort* ou por seus valores *default*.

A.2 Inicialização do ORB

Antes de um programa Java poder usar objetos CORBA, deve haver uma inicialização para criar o (objeto) ORB e obter uma ou mais referências iniciais de objetos, i.e., ao menos um contexto de nomes. Ao criar um objeto ORB uma aplicação obtém acesso às operações que estão definidas no ORB. Este fragmento de código mostra como uma aplicação pode criar um objeto ORB:

```
import org.omg.CORBA.ORB;
public static void main(String args[]) {
    try{
        ORB orb = ORB.init(args, null);
        ...
    }
}
```

Os argumentos para o método de inicialização são:

- *args* ou *this* : Passa ao ORB os argumentos (da aplicação).
- *null* : Um objeto java.util.Properties.

A operação *init()* utiliza estes parâmetros e propriedades do sistema, para obter a informação necessária para configurar o ORB. As propriedades de configuração são buscadas na seguinte ordem e lugares:

1. Os parâmetros da aplicação (primeiro argumento)
2. Um objeto *java.util.Properties* (segundo argumento), caso fornecido
3. O objeto *java.util.Properties* retornado por *System.getProperties()*

O primeiro valor encontrado para uma propriedade em particular é o valor utilizado por *init()*. Se uma propriedade de configuração não pode ser encontrada em nenhum destes lugares, a operação *init()* assume um valor específico da implementação para ela. Para a máxima portabilidade entre implementações de ORB, aplicações deveriam especificar explicitamente valores de propriedades de configuração que afetam suas operações ao invés de depender dos valores assumidos para o ORB onde estão executando.

Com relação às propriedades de sistema, note que a máquina virtual *Java* da *Sun* acrescenta o argumento *-D* na linha de comando. Outras máquinas virtuais Java podem não fazer o mesmo. Atualmente as seguintes propriedades são definidas para todas as implementações de ORB:

- org.omg.CORBA.ORBClass

  O nome de uma classe Java que implementa a interface *org.omg.CORBA.ORB*. O valor para ORB Java IDL é com.sun.CORBA.iiop.ORB

- org.omg.CORBA.ORBSingletonClass

  O nome de uma classe Java que implementa a interface *org.omg.CORBA.ORB*. Este é o objeto retornado pela chamada a orb.init() sem argumentos. É utilizada basicamente para criar instâncias de *typecode* que podem ser compartilhadas através de código não confiável (tal como um *unsigned applet*) em um ambiente seguro.

Em adição às propriedades listadas anteriormente, as seguintes propriedades são suportadas:

- org.omg.CORBA.ORBInitialHost

  O nome de um hospedeiro executando um servidor que provê o *bootstrap* de serviços iniciais como o serviço de Nomes. O valor *default* para esta propriedade é *localhost* para aplicações e para applets é o *applet host*, equivalente a *getCodeBase().getHost()*.

- org.omg.CORBA.ORBInitialPort

  A porta inicial de escuta do Servidor de Nomes. O valor *default* é 900.

Para invocar um objeto CORBA, uma aplicação deve ter uma referência para ele. Há três maneiras de se obter uma referência para um objeto CORBA:

- de uma *string* que foi especialmente criada de um referência de objeto;
- de um outro objeto, tal como um contexto de nomes;
- de operação do ORB *resolve_initial_references()*.

A primeira técnica de converter uma referência *stringified* para uma referência de objeto de fato é independente de implementação de ORB. Não importando o ORB Java em que a aplicação executa, ela converte a referência de objeto *stringified*. Entretanto, depende do desenvolvedor da aplicação:

- garantir que o objeto referido seja de fato acessível de onde a aplicação está executando;
- obter a referência *stringified*, talvez de um arquivo ou um parâmetro.

O seguinte fragmento de código mostra como um servidor converte uma referência de objeto CORBA para uma *string*:

```
org.omg.CORBA.ORB orb =    // obter um obejto ORB
org.omg.CORBA.Object obj = // criar a referência de objeto
String str = orb.object_to_string(obj);
// tornar a string disponível ao cliente
```

O fragmento de código a seguir mostra como um cliente converte uma referência de objeto *stringified* de volta em objeto:

```
org.omg.CORBA.ORB orb = // obter um obejto ORB
String stringifiedref = // ler string
org.omg.CORBA.Object obj = orb.string_to_object(stringifiedref);
```

Ao não utilizar uma referência *stringified* para obter um referência de objeto CORBA, pode-se utilizar o próprio ORB para fornecer a referência de objeto. Isto pode tornar a aplicação ORB-dependente, isto é, embora a especificação CORBA defina a interface para obter referências de objetos, ela não fornece ainda informação suficiente para uma implementação padronizada. Desenvolvedores de aplicações devem ser cautelosos ao usar esta operação até o padrão ser especificado mais claramente. Para ser ORB-independente, utilize preferencialmente a técnica de referência de objeto *stringified*.

A interface de ORB define uma operação chamada *resolve_initial_references()* pretendida para carregar (*bootstrapping*) referências de objeto em uma nova aplicação. A operação utiliza um argumento *string* que nomeia um ou alguns objetos identificados e retorna um objeto CORBA, que deve ser estreitado (*narrowed*) como o tipo que a aplicação conhece. Duas *strings* estão atualmente definidas:

NameService (Nomes) - Este valor retorna uma referência para uma raiz de contexto de nomes que, após estreitamento, pode ser utilizada para consultar referências de objetos cujos nomes são conhecidos da aplicação (assumindo que estes objetos estão registrados pelos seus nomes na raiz ou em contexto de nomes subordinado)

InterfaceRepository (Repositório de Interfaces) - Este valor retorna uma referência para um Repositório de Interfaces, isto é, um objeto CORBA que contém definições de interfaces. O argumento "*InterfaceRepository*" deve ser passado para *resolve_initial_references()*.

A.3 Serviço de Nomes

Este resumo considera o pacote *org.omg.Naming* de Java2 (JDK-1.2) [Java98-3] e OrbixWeb3.x [OrbixWeb97-1]. O Serviço de Nomes fornecido com javaIDL é uma implementação simples do Serviço de Nomes constante da especificação CORBA. O Serviço de Nomes fornece uma árvore de diretórios para referências de objetos semelhante a uma estrutura de diretório de sistemas de arquivos. As referências de objeto são armazenadas no espaço de nomes e cada par *referência-nome de objeto* é denominada uma associação (*binding*) conhecida. As associações conhecidas podem ser organizadas em contextos de nomes. Contextos de nomes são eles mesmos associações e tem a mesma função organizacional de um subdiretório de arquivos. Todas as associações são armazenadas sob o contexto inicial de Nomes.

O contexto inicial é a única associação persistente no espaço de nomes, sendo o restante perdido quando o serviço de Nomes é reiniciado. Para um *applet* ou aplicação usar o servidor de Nomes, o ORB deve saber o nome e a porta do hospedeiro onde o servidor executa ou ter acesso à IOR do servidor conhecido. Se não especificado de outra forma, o servidor de Nomes escuta na porta 900 para o protocolo de *bootstrap* utilizado para implementar os métodos *resolve_initial_references()* e *list_initial_references()* do ORB. Para especificar uma porta diferente, por exemplo: 1050:

tnameserv - ORBInitialPort 1050

Para informar aos clientes o novo número de porta deve-se modificar a propriedade *org.omg.CORBA.InitialPort* para o novo número ao criar um novo objeto ORB.

A.3.1. Adicionando Objetos ao Espaço de Endereçamento

```
import java.util.Properties;
import org.omg.CORBA.*;
import org.omg.CosNaming.*;
public class NameClient {
    public static void main(String args[]) {
        try {
            // continua ...
```

Anteriormente o servidor foi inicializado na porta 1050. O seguinte código assegura que o programa cliente esteja ciente deste número de porta.

```
            // continuação ...
            Properties props = new Properties();
            props.put("org.omg.CORBA.ORBInitialPort", "1050");
            ORB orb = ORB.init(args, props);
            // continua ...
```

Este código obtém o contexto inicial de nomes e o atribui a *ctx*. A segunda linha copia *ctx* para uma referência de objeto temporária *objref* que ligaremos a vários nomes e adicionaremos ao espaço de endereçamento.

```
// continuação ...
NamingContext ctx = NamingContextHelper.narrow(orb.resolve_initial_references("NameService"));
NamingContext objref = ctx;
// continua ...
```

Este código cria um nome *planos* do tipo *texto* e o liga à referência de objeto temporária. *planos* é adicionado então sob o contexto inicial usando o *rebind*. O método *rebind* permite chamadas repetidas sem as exceções do método *bind*.

```
// continuação ...
NameComponent nc1 = new NameComponent("planos", "text");
NameComponent[] name1 = {nc1};
ctx.rebind(name1, objref);
System.out.println("rebind de planos com sucesso!");
// continua ...
```

Este código cria um contexto de nomes chamado *Pessoal* do tipo *directory*. A referência de objeto resultante, ctx2, é limitada ao nome e adicionada no contexto de nomes inicial.

```
// continuação ...
NameComponent nc2 = new NameComponent("Pessoal", "directory");
NameComponent[] name2 = {nc2};
NamingContext ctx2 = ctx.bind_new_context(name2);
System.out.println("novo contexto de nomes adicionado.");
// continua ...
```

O restante do código liga a referência de objeto temporária usando os nomes *programação* e *calendário* sob o contexto de nome *Pessoal* (ctx2).

```
        // continuação ...
        NameComponent nc3 = new NameComponent("programação", "text");
        NameComponent[] name3 = {nc3};
        ctx2.rebind(name3, objref);
        System.out.println(" rebind de programação com sucesso!");

        NameComponent nc4 = new NameComponent("calendário", "text");
        NameComponent[] name4 = {nc4};
        ctx2.rebind(name4, objref);
        System.out.println(" rebind de calendário com sucesso!");
    } catch (Exception e) {e.printStackTrace(System.err);}
  }
} // Fim.
```

Cliente. O código a seguir ilustra como varrer o espaço de nomes.

```
import java.util.Properties;
import org.omg.CORBA.*;
import org.omg.CosNaming.*;
public class NameClientList {
```

```
public static void main(String args[]) {
    try {
        // continua ...
```

Anteriormente o servidor foi inicializado na porta 1050. O seguinte código assegura que o programa cliente esteja ciente deste número de porta.

```
// continuação ...
Properties props = new Properties();
props.put("org.omg.CORBA.ORBInitialPort", "1050");
ORB orb = ORB.init(args, props);
// continua ...
```

O código seguinte obtém o contexto inicial de nomes.

```
// continuação ...
NamingContext nc = NamingContextHelper.narrow(orb.resolve_initial_references("NameService"));
// continua ...
```

O método *list* lista os associações no contexto de nomes. Neste caso, até 1000 associações do contexto inicial são retornados por *BindingListHolder*. Todos os associações restantes são retornados por *BindingIteratorHolder*.

```
// continuação ...
BindingListHolder bl = new BindingListHolder();
BindingIteratorHolder blIt= new BindingIteratorHolder();
nc.list(1000, bl, blIt);
// continua ...
```

Este código obtém a matriz de associações fora do *BindingListHolder* retornado. Se não houver nenhum ligamento, o programa termina.

```
// continuação ...
Binding bindings[] = bl.value;
if (bindings.length == 0) return;
// continua ...
```

O restante do código faz recursão através dos associações e imprime os nomes.

```
// continuação ...
for (int i=0; i < bindings.length; i++) {
    // obter a referência de objeto para cada binding
    org.omg.CORBA.Object obj = nc.resolve(bindings[i].binding_name);
    String objStr = orb.object_to_string(obj);
    int lastIx = bindings[i].binding_name.length-1;

    // verificar se este é um contexto de nomes
    if (bindings[i].binding_type == BindingType.ncontext) {
        System.out.println( "Context: " + bindings[i].binding_name[lastIx].id);
    } else { System.out.println("Object: " + bindings[i].binding_name[lastIx].id); }
}
    } catch (Exception e) { e.printStackTrace(System.err); }
  }
} // Fim.
```

A.4 IDL do Serviço de Nomes

A especificação IDL do Serviço de Nomes é apresentada a seguir. É composta basicamente do módulo CosNaming com as interfaces NamingContext e BindingIterator. A primeira se refere a estruturação do contexto de nomes enquanto a segunda se refere a uma interface iterativa para refinamento de um consulta.

```
//**************************************************
// Arquivo: CosNaming.idl
//**************************************************
// File: CosNaming.idl
#ifndef _COSNAMING_IDL_
#define _COSNAMING_IDL_
#pragma prefix "omg.org"
//**************************************************
module CosNaming {
  typedef string Istring;
  struct NameComponent {
    Istring id;
    Istring kind; };
  typedef sequence<NameComponent> Name;
  enum BindingType { nobject, ncontext };
  struct Binding {
    Name binding_name;
    BindingType binding_type; };

// Nota: Na struct Binding, binding_name é corretamente definido como Name invés de NameComponent.
// Esta definição não muda por razões de compatibilidade.

  typedef sequence <Binding> BindingList;
//**************************************************
  interface BindingIterator;
//**************************************************
  interface NamingContext {
    enum NotFoundReason { missing_node, not_context, not_object };
    exception NotFound { NotFoundReason  why; Name rest_of_name; };
    exception CannotProceed { NamingContext  cxt; Name rest_of_name; };
    exception InvalidName{ };
    exception AlreadyBound { };
    exception NotEmpty{ };
    void bind ( in Name n, in Object obj ) raises ( NotFound, CannotProceed, InvalidName, AlreadyBound );
    void rebind ( in Name n, in Object obj ) raises ( NotFound, CannotProceed, InvalidName );
    void bind_context( in Name n, in NamingContext nc ) raises ( NotFound, CannotProceed, InvalidName,
AlreadyBound );
    void rebind_context( in Name n, in NamingContext nc ) raises ( NotFound, CannotProceed, InvalidName );
    Object resolve ( in Name n ) raises ( NotFound, CannotProceed, InvalidName );
    void unbind (in Name n ) raises( NotFound, CannotProceed, InvalidName );
    NamingContext new_context( );
    NamingContext bind_new_context( in Name n ) raises ( NotFound, AlreadyBound, CannotProceed, Invalid-
Name );
    void   destroy() raises(NotEmpty);
    void   list( in unsigned long   how_many, out BindingList bl, out BindingIterator bi );
  };
```

```
//**************************************************
  interface BindingIterator {
    boolean next_one( out Binding b );
    boolean next_n( in unsigned long how_many, out BindingList bl );
    void destroy( );
  };
//**************************************************
  interface NamingContextExt: NamingContext {
    typedef string StringName;
    typedef string Address;
    typedef string URLString;
    StringName to_string( in Name n ) raises ( InvalidName );
    Name to_name( in StringName sn ) raises ( InvalidName );
    exception InvalidAddress { };
    URLString to_url (in Address addr, in StringName sn ) raises (InvalidAddress, InvalidName);
    Object resolve_str ( in StringName n ) raises ( NotFound, CannotProceed, InvalidName, AlreadyBound );
  };
//**************************************************
};
//**************************************************
#endif // _COSNAMING_IDL_
//**************************************************
```

APÊNDICE B

MASIF

A mente é como o vento
e o corpo como a areia;
se você quer conhecer o vento
observe o movimento da areia.
Anônimo

Este Apêndice compreende IDLs da especificação OMG/MASIF [OMG97-2] e descrição de cada operação e atributo. Está organizado nas seções:

- B.1 IDL CfMAF, página 135,
- B.2 Interface MAFAgentSystem, página 139 e
- B.3 Interface do MAFFinder, página 151.

B.1 IDL CfMAF

```
    module CfMAF {
// ***************************************************
// Tipos de Dados
// ***************************************************
    typedef sequence<octet> OctetString;
    typedef sequence<OctetString> OctetStrings;
    typedef OctetString Authority;
    typedef OctetString Identity;
    typedef short LanguageID;
    typedef short AgentSystemType;
    typedef short Authenticator;
    typedef short SerializationID;
    typedef sequence<SerializationID> SerializationIDList;
    typedef any Property;
    typedef sequence<Property> PropertyList;
    struct Name {
    Authority authority;
    Identity identity;
    AgentSystemType agent_system_type;
    };
    typedef sequence<Name> NameList;
    struct AuthInfo {
    boolean is_authenticated;
```

```
    Authenticator authenticator;
    };
    struct LanguageMap {
    LanguageID language_id;
    SerializationIDList serializations;
    };
    typedef sequence<LanguagedMap> LanguageMapList;
    struct AgentSystemInfo {
    Name agent_system_name;
    AgentSystemType agent_system_type;
    LanguageMapList language_maps;
    string agent_system_description;
    short major_version;
    short minor_version;
    PropertyList properties;
    };
    struct AgentProfile{
    LanguageID language_id;
    AgentSystemType agent_system_type;
    string agent_system_description;
    short major_version;
    short minor_version;
    SerializationID serialization;
    PropertyList properties;
    };
    struct ClassName{
    string name;
    OctetString discriminator;
    };
    typedef sequence<ClassName> ClassNameList;
    typedef sequence<octet> Arguments;
    typedef string Location;
    typedef sequence<Location> Locations;
    enum AgentStatus {
    CfMAFRunning, CfMAFSuspended, CfMAFTerminated
    };
// ***************************************************
// Exceções
// ***************************************************
    exception AgentNotFound { };
    exception AgentIsRunning { };
    exception AgentIsSuspended { };
    exception ArgumentInvalid { };
    exception ClassUnknown { };
    exception DeserializationFailed { };
    exception EntryNotFound { };
    exception FinderNotFound { };
    exception MAFExtendedException { };
    exception NameInvalid { };
    exception ResumeFailed { };
    exception SuspendFailed { };
    exception TerminateFailed { };
```

```
// ***************************************************
// Definições de Interfaces
// ***************************************************
    interface MAFFinder {
    void register_agent (
    in Name agent_name,
    in Location agent_location,
    in AgentProfile agent_profile,)
    raises ( NameInvalid );
    void register_agent_system (
    in Name agent_system_name,
    in Location agent_system_location,
    in AgentSystemInfo agent_system_info )
    raises ( NameInvalid );
    void register_place (
    in string place_name,
    in Location place_location )
    raises ( NameInvalid );
    Locations lookup_agent (
    in Name agent_name,
    in AgentProfile agent_profile )
    raises ( EntryNotFound );
    Locations lookup_agent_system (
    in Name agent_system_name,
    in AgentSystemInfo agent_system_info )
    raises ( EntryNotFound );
    Locations lookup_place (in string place_name)
    raises (EntryNotFound);
    void unregister_agent ( in Name agent_name )
    raises ( EntryNotFound );
    void unregister_agent_system ( in Name agent_system_name )
    raises ( EntryNotFound );
    void unregister_place ( in string place_name )
    raises ( EntryNotFound );
    };
    interface MAFAgentSystem {
    Name create_agent (
    in Name agent_name,
    in AgentProfile agent_profile,
    in OctetString agent,
    in string place_name,
    in Arguments arguments,
    in ClassNameList class_names,
    in string code_base,
    in MAFAgentSystemclass_provider )
    raises (
    ClassUnknown, ArgumentInvalid,
    DeserializationFailed,
    MAFExtendedException );
    OctetStrings fetch_class (
    in ClassNameList class_name_list,
    in string code_base,
    in AgentProfile agent_profile )
```

```
raises (
ClassUnknown,
MAFExtendedException );
Location find_nearby_agent_system_of_profile (
in AgentProfile profile )
raises ( EntryNotFound );
AgentStatus get_agent_status (
in Name agent_name )
raises( AgentNotFound );
AgentSystemInfo get_agent_system_info ( );
AuthInfo get_authinfo (
in Name agent_name )
raises ( AgentNotFound );
MAFFinder get_MAFFinder ( )
raises ( FinderNotFound );
NameList list_all_agents ( );
NameList list_all_agents_of_authority (
in Authority authority );
Locations list_all_places ( );
void receive_agent (
in Name agent_name,
in AgentProfile agent_profile,
in OctetString agent,
in string place_name,
in ClassNameList class_names,
in string code_base,
in MAFAgentSystemagent_sender )
raises (
ClassUnknown,
ArgumentInvalid,
DeserializationFailed,
MAFExtendedException );
void resume_agent (
in Name agent_name)
raises (
AgentNotFound,
ResumeFailed,
AgentIsRunning ) ;
void suspend_agent (
in Name agent_name )
raises (
AgentNotFound,
SuspendFailed,
AgentIsSuspended );
void terminate_agent (
in Name agent_name )
raises (
AgentNotFound,
TerminateFailed );
};
};
```

B.2 Interface *MAFAgentSystem*

Descrevemos a seguir, os métodos presentes na interface *MAFAgentSystem*.

create_agent()

Um sistema de agentes executa a operação *create_agent* para criar um agente de acordo com a requisição de um cliente remoto. O nome real do agente criado é retornado. Este pode ser o mesmo que o nome dado como parâmetro se o cliente for responsável por nomear.

Sintaxe

```
Name create_agent (
  in Name agent_name,
  in AgentProfile agent_profile,
  in OctetString agent,
  in string place_name,
  in Arguments arguments,
  in ClassNameList class_names,
  in string code_base,
  in MAFAgentSystem class_provider)
raises (ClassUnknown, ArgumentInvalid, DeserializationFailed, MAFExtendedException);
```

Parâmetros

- *agent_name*. É o nome do novo agente. A autoridade do agente é a autoridade do cliente. Seu *agent_system_type* é ou o *agent_system_type* do cliente, se o cliente é um sistema de agentes, ou é *NonAgentSystem* (valor 0, referir Apêndice B do MASIF, "Assigned Numbers", para maiores detalhes) se o cliente não é um sistema de agentes. A identidade deve ser fornecida se o cliente é responsável por nomear o novo agente. Se a responsabilidade é do sistema de agentes, o campo identidade é ignorado. O nome real para o agente novo é dado como valor de retorno.

- *agent_profile*. Contém informação sobre o agente, tal como o tipo do sistema de agentes que o criou, e o método usado para serializá-lo para transferência. Baseado no perfil do agente e perfil do sistema destino, o segundo pode descobrir se as exigências do agente e do suporte do sistema são *similares* o bastante a fim aceitar o agente no sítio.

  Similar é definido e interpretado pelo sistema que aceita o agente. *Similar* pode ser definido pelo tipo do fabricante. Por exemplo, em certos casos pode não haver interoperabilidade entre certos fabricantes, e ao mesmo tempo haver interoperabilidade entre outros. *Similar* também pode ser definido pelas versões do sistema. Agentes podem ter sido criados em sistemas de agentes do mesmo fabricante mas com uma versão obsoleta ou incompatível. *Não-similar* certamente vale para linguagens diferentes.

- *agente* . Este parâmetro é opaco, e pode conter qualquer coisa que o remetente necessita passar ao sistema remoto de agentes que não esteja incluso nos outros parâmetros deste método. Este parâmetro contem a informação que é única aos agentes de um perfil particular. O sistema de agentes que cria o agente deve poder decodificar o significado da informação contida neste parâmetro.

  O tipo de informação que pode estar neste parâmetro inclui, mas não está restrita à definição da classe do agente, aos estados do agente, e definições de alguma ou todas as classe necessárias para instanciar o agente no sistema remoto de agentes.

- *place_name*. É o nome do lugar onde o agente residirá. Se este parâmetro não for especificado, o sistema de agentes cria o agente em um lugar *defualt* dentro do sistema.

- *arguments* . Este parâmetro especifica os argumentos do construtor do agente.

- *class_names*. É a lista que contém o nome de cada classe necessária para instanciar o agente. Notar que a lista de classes é opcional, o que significa que pode ser vazia. Como CORBA não aceita *null* como argumento passado em um parâmetro especificado como *classNameList*, o *classNameList* com uma *string* vazia deve ser usada para indicar o caso onde nenhum nome de classe é requerido. Implementadores do MAF podem fornecer um nome especial para classe nula por conveniência.

  A lista pode ou não ser necessária dependendo do mecanismo de transferência de classe escolhido. Ver "*Class Transfer*" seção 1.3.1, "*Remote Agent Creation*" na especificação [OMG97-2] para uma discussão dos vários mecanismo e suas exigências.

  As classes listadas aqui podem ou não necessitar ser transferidas dependendo do sistema de agentes ter uma *cache* de classes. De fato, o sistema de agentes de recepção deve decidir se *fetch_class*() é necessário.

- *code_base*. É uma referência a base de código contendo as definições necessárias da classe. A sintaxe deste parâmetro pode variar entre tipos de sistemas de agentes sem afetar a interoperabilidade. É retornado ao cliente requisitador através de *fetch_class*(), se necessário. Conseqüentemente, somente o provedor de classes necessita conhecer a sintaxe desta *string* se a origem das definições de classe é um sistema de agentes. É possível para o sistema de agentes destino recuperar as definições de classe diretamente, usando a informação do *codebase*.

- *class_provider*. É uma referência à origem do cliente que fornece as necessárias definições de classes. Notar que o parâmetro *class_provider* deve ser *void* se as classes forem fornecidas por um não -sistema de agentes.

Exceções

- *ClassUnknown* . A definição de classe não foi encontrada.

- *ArgumentInvalid*. Os argumentos passados não correspondem a nenhuma assinatura de construtor de agente.

- *DeserializationFailed* . O sistema de agentes não pode instanciar o agente porque não pode decodificar o *OctetString* do agente.

- *MAFExtendedException.* Esta é uma exceção genérica. Use somente se nenhuma outra exceção se aplica à condição de erro.

Notas De Uso

Mesmo um cliente remoto sem facilidades de agente pode criar um agente utilizando este método.

O sistema de agentes destino está livre para enfileirar o agente ou direcioná-lo a um outro sistema de agentes dentro de sua região. Este é um detalhe da implementação que a especificação permite mas não exige.

---

fetch_class()

O método *fetch_class*() retorna as definições de uma ou mais classes. No caso de sistema de agentes não orientado a objeto, o método *fetch_class*() é usado puxar o código.

Sintaxe

OctetStrings *fetch_class*(
in ClassNameList *class_name_list*,
in string *code_base*, in AgentProfile *agent_profile*)
raises (*ClassUnknown*, *MAFExtendedException*);

Parâmetros

- *class_name_list.* Os nomes das definições de classes requisitadas.

- *code_base.* É uma referência à base de código que contém as definições necessárias da classe. A sintaxe deste parâmetro pode variar com os tipos de sistema de agente sem afetar a interoperabilidade. O cliente fornece o code_base no *create_agent*() ou no *receive_agent*(). É retornado ao cliente através de *fetch_class*(), se necessário. Portanto, somente o fornecedor da classe necessita entender a sintaxe desta *string*.

- *agent_profile* . Contém informação sobre a linguagem e o método de serialização usados para o contexto corrente de *create_agent*() ou de *receive_agent*().

Exceções

- *ClassUnknown* . Definição de classe não encontrada.

- *MAFExtendedException.* É uma exceção genérica. Usada somente se nenhuma outra exceção se aplica à condição de erro.

Notas De Uso

Usar este método para recuperar as classes do *code_base* e *cliente* especificados. O sistema de agentes requisitado pode saber se é o provedor de classe ou não examinando o code_base dado. Se não, o sistema de agente requisitado pode retornar as classes *cacheadas,* se alguma, ou pode direcionar esta requisição a um outro sistema de agentes. Este é um detalhe de implementação que a especificação permite mas não exige.

find_nearby_agent_system_of_profile()

O método *find_nearby_agent_system_of_profile*() requisita o *MAFFinder* para encontrar um sistema de agentes próximo que possa executar o agente que o cliente deseja enviar. Este é um método da interface *MAFAgentSystem*, que depende de *MAFFinder* para localizar um próximo sistema de agentes do tipo correto. A implementação do *MAFAgentSystem* deve obter esta funcionalidade usando o *MAFFinder*.

Sintaxe

Location *find_nearby_system_of_profile* (
in AgentProfile *profile*)
raises (*EntryNotFound*);

Parâmetros

- *profile*. O perfil do agente que está sendo enviado.

Exceções

- *EntryNotFound*. O agente especificado não está na lista de agentes do sistema de agentes corrente.

Notas De Uso

Por vezes um agente deseja se comunicar com um objeto que reside em um sistema de agentes de tipo diferente (i.e. um que não suporta o perfil do agente viajante), ou um não sistema de agentes. Este método deixa o agente requisitante procurar por um sistema de agentes do tipo correto que esteja mais perto do objeto com o qual deseja se comunicar, que reside em um tipo incorreto (*MAFAgentSystem* não suportado, não suportando a versão requerida, linguagem diferente, etc.). A fim de poder otimizar a comunicação com esse objeto, um outro *MAFAgentSystem* do tipo correto (a mesma versão como requerido pelo agente móvel) é encontrado.

Esta interface é altamente específica da aplicação. Seria extremamente difícil generalizar a métrica de proximidade. Portanto, a aplicação deve definir e tirar vantagem desta interface.

get_agent_status()

O método *get_agent_status*() retorna o status do agente especificado.

Sintaxe
```
Agent_Status get_agent_status (
in Name agent_name)
raises (AgentNotFound);
```

Parâmetros

• *agent_name* . É o nome do agente cujo estado o requisitante quer saber.

Exceções

• *AgentNotFound* . O sistema de agente não pode encontrar o agente especificado.

Notas De Uso

O parâmetro de retorno *Agent_Status* pode ser um de três valores:

- *funcionando*, significa que o agente está executando
- *suspenso*, significa que o agente não está executando
- *terminado*, significa que o agente terminou a execução

Consulte à definição para *ClassName* ("*MAF IDL Interfaces*" na página 86) para os valores de *Agent_Status*.

Este método é útil em aplicações de gerenciamento. Permite ao gerente do sistema monitorar o estado de um agente.

get_agent_system_info()

O método *get_agent_system_info*() retorna a estrutura de *AgentSystemInfo*. Esta estrutura contém a informação sobre o sistema de agentes, incluindo seu nome e o perfil de agentes que suporta.

Sintaxe
```
AgentSystemInfo get_agent_system_info();
```

Parâmetros

Não há nenhum parâmetro para este método. Retorna a informação sobre o sistema de agentes corrente.

Exceções

Nenhuma.

Notas De Uso

Um agente pode usar este método para encontrar uma informação sobre um sistema de agentes para o qual deseja ir.

A estrutura *AgentSystemInfo* que este método retorna fornece a seguinte informação sobre o sistema de agentes:

- *system_name* . Nome do sistema de agentes.
- *system_type* . Identifica o tipo de sistema de agentes (por exemplo: Aglets, MOA, ou Agent-Tcl).
- *language_maps* . É a linguagem de programação que o sistema de agentes suporta (por exemplo: Java, Tcl, Scheme, ou Perl), e os esquemas de serialização que cada uma destas linguagens usa (por exemplo: JavaObjectSerialization, ASN1_BER, ASN1_DER).
- *system_description* . É uma descrição resumida do sistema de agentes. A informação neste parâmetro não é padronizada; é dependente da implementação.
- *major_version*. É a informação sobre a versão de implementação do sistema de agentes.
- *minor_version* . informação sobre a versão de implementação do sistema do agentes *implementation.get_authinfo*()
- *serializations*. Identificam os esquemas de serialização que o sistema de agentes usa (por exemplo: JavaObjectSerialization, ASN1_BER, ASN1_DER).

get_authinfo()

O método *get_authinfo*() retorna a informação se um agente foi autenticado, e qual método de autenticação foi usado.

Sintaxe
AuthInfo *get_authinfo*(
in Name *agent_name*)
raises (*AgentNotFound*);

Parâmetros

- *agent-name* . É o nome do agente cuja informação de autenticação é requisitada.

Exceções

• *AgentNotFound* . O sistema do agente não pode encontrar o agente especificado.

Notas De Uso

Se segurança for desejada, o cliente deve autenticar o sistema de agentes antes de chamar este método.

---

get_MAFFinder()

Retorna uma referência a um *MAFFinder* para localizar *agentes*, *lugares* (*places*), e sistemas de agentes.

Sintaxe MAFFinder *get_MAFFinder*()
raises (*MAFFinderNotFound*);

Parâmetros

Nenhum.

Exceções

• *FinderNotFound* . Não foi possível encontrar o *MAFFinder* da *região* corrente.

Notas De Uso

Uma vez obtida a referência do *MAFFinder*, pode-se usar os método do *MAFFinder* para encontrar *agentes*, *lugares*, e sistemas de agentes dentro da *região* (*region*).

---

list_all_agents()

O método *list_all_agents*() lista todos os agentes registrados no sistema de agentes.

Sintaxe NameList *list_all_agents*();

Parâmetros

Nenhum.

Exceções

Nenhuma.

Nota De Uso

Esta é uma operação de gerenciamento que permite um gerente do sistema rastrear um agente dentro de um sistema de agentes.

---

list_all_agents_of_authority()

O método *list_all_agents_of_authority*() lista todos os agentes dentro do sistema de agentes que têm a autoridade especificada.

Sintaxe NameList *list_all_agents_of_authority* (
in Authority *authority*);

Parâmetros

• *authority* . Identifica a autoridade cujos agentes se quer listar.

Exceções

Nenhuma.

Nota De Uso

Esta é uma operação de gerenciamento que permite um gerente do sistema rastrear um agente dentro de um sistema de agentes.

---

list_all_places()

O método do list_all_places() lista todos os lugares dentro do sistema do agente.

Sintaxe Locations *list_all_places*();

Parâmetros

Nenhum.

Exceções

Nenhuma.

Nota De Uso

Esta é uma operação de gerenciamento que permite um gerente do sistema obter a lista de lugares (*places*) de um sistema de lugares registrados com *MAFFinder*.

receive_agent()

Um sistema de agentes usa *receive_agent*() para receber e instanciar um agente.

Sintaxe

```
void receive_agent (
in Name agent_name,
in AgentProfile agent_profile,
in OctetString agent,
in string place_name,
in ClassNameList class_names,
in string code_base,
in MAFAgentSystem agent_sender)
raises (ClassUnknown, DeserializationFailed, MAFExtendedException);
```

Parâmetros

- *agent_name* . Identificador único de um agente. Este identificador deve incluir a autoridade da pessoa ou organização que o agente está representando, e a identidade do agente para compor o nome único do agente.

- *agent_profile* . Contém informação do agente, tal como o tipo de sistema de agentes que o criou, e o método de serialização usado para transferí-lo.

- *agent* . Este parâmetro é opaco, e pode conter qualquer coisa que o remetente necessita fazer saber ao sistema de agentes remoto e que não esta em outro parâmetro deste método. Este parâmetro contém informação que é única aos agentes de um tipo de perfil particular. O sistema de agentes que recebe o agente deve poder decodificar o significado da informação neste parâmetro.

  O tipo informação que pode estar neste parâmetro inclui, mas está restrito à definição de classe do agente, estado de execução do agente, e definições de algumas ou todas as classes necessárias para instanciar o agente no sistema de agentes remoto.

  Note que as definições de classe incluídas neste parâmetro afetam o parâmetro *class_names*. Se uma classe é incluída na definição, não é necessário adicioná-la à lista de *class_names*. Alternativamente, se uma definição de classe não é incluída neste parâmetro, deve estar na lista de *class_names*.

- *place_name*. É o nome do lugar onde o agente residirá. Se este parâmetro não for especificado, o sistema de agentes cria o agente em um lugar *default* dentro do sistema.

- *class_names* . É uma lista contendo o nome de cada classe necessária para instanciar o agente. Notar que a lista de classe é opcional, o que significa que pode estar vazia. Como CORBA não aceita *null* passado como argumento para um parâmetro especificado como *classNameList*, implementadores do *MAF* devem criar um nome especial de classe nula com o campo nome definido por uma *string* vazia.

A lista pode ou não ser necessária dependendo do mecanismo de transferência de classe escolhido. Consultar "Class Transfer" seção 1.3.1, "Remote Agent Creation" na especificação [OMG97-2] para uma discussão dos vários mecanismos e suas exigências.

Este parâmetro e o parâmetro *agent* estão relacionados. Se o parâmetro *agent* contiver uma definição de classe, esta classe não necessita estar na lista de *class_names*. Alternativamente, qualquer classe requerida para instanciar o agente deve estar nesta lista se não estiver incluída no parâmetro *agent*.

As classes listadas aqui podem ou não necessitar serem transferidas dependendo de o sistema de agentes *cachear* classes. De fato, o sistema de agentes receptor deve decidir se o *fetch_class*() é necessário.

- *code_base* . É uma referência à base de código que contém as definições necessárias da classe. A sintaxe deste parâmetro pode variar com os tipos de sistema de agente sem afetar a interoperabilidade. É retornado ao cliente através de *fetch_class*(), se necessário. Portanto, somente o fornecedor da classe necessita entender a sintaxe desta *string*.

- *agent_sender*. É uma referência ao sistema de agentes que inicia a transferência do agente.

Exceções

- *ClassUnknown* . Definição de classe não encontrada.

- *DeserializationFailed*. O sistema de agentes não pode instanciar o agente porque não pode decodificar o OctetString do agente.

- *MAFExtendedException*. Esta é uma exceção genérica. Usada somente se nenhuma outra exceção se aplica à condição de erro.

Notas De Uso

Um algorítmo possível para a execução deste método é:

1. Verificar se as classes requeridas para instanciar o agente estão incluídas no parâmetro de entrada *receive_agent*, ou *cacheado* na plataforma do sistema de agentes.
2. Chamar *fetch_class*() se necessário, para recuperar algumas classes requeridas que não estão disponíveis.
3. Deserializar e instanciar o agente no lugar especificado na chamada, ou no lugar default do sistema de agentes se nenhum lugar for especificado.

O sistema de agentes destino é livre para enfileirar o agente ou redirecioná-lo a outro sistema de agentes dentro de sua região. Este é um detalhe de implementação que a especificação permite mas não exige.

resume_agent()

O método *resume_agent*() retoma a execução do agente especificado.

Sintaxe
void *resume_agent*(
in Name *agent_name*)
raises (*AgentNotFound, ResumeFailed, AgentIsRunning*);

Parâmetros

• *agent_name* . Nome do agente a retomar a execução.

Exceções

• *AgentNotFound* . O sistema de agentes não pode encontrar o agente especificado.

• *ResumeFailed* . O agente não pode retomar a execução.

• *AgentIsRunning* . O agente já está executando; não necessita ser retomado.

Notas De Uso

Este método fornece a função de gerenciamento de reiniciar um agente que foi suspenso, com *suspend_agent*().

suspend_agent()

O método do *suspend_agent*() suspende a execução do agente especificado.

Sintaxe
void *suspend_agent* (
in Name *agent_name*)
raises (*AgentNotFound, SuspendFailed, AgentIsSuspended*);

Parâmetros

• *agent_name* . Nome do agente a suspender.

Exceções

• *AgentNotFound.* O sistema de agentes não pode encontrar o agente especificado.

• *SuspendFailed* . Não foi possível parar a execução do agente.

• *AgentIsSuspended* . O agente já está suspenso.

Notas De Uso

Este método fornece a função de gerenciamento de suspender a execução de um agente. Usar *resume_agent* para reiniciar a execução do agente quando desejado. Para implementar este método, deve-se suspender a *thread* de execução do agente e manter os estados do agente.

Há diversas razões para suspender execução de agente. Por exemplo, pode-se suspender um agente para passar recursos a uma *thread* de prioridade mais alta. Pode-se também suspender um agente se houver suspeita de violação de segurança.

---

terminate_agent()

O método *terminate_agent*() para execução do agente especificado.

Sintaxe
void *terminate_agent* (
in Name *agent_name*)
raises (*AgentNotFound, TerminateFailed*);

Parâmetros

• *agent_name* . Nome do agente a terminar.

Exceções

• *AgentNotFound.* O sistema de agentes não pode encontrar o agente especificado.

• *TerminateFailed* . Não foi possível parar a execução do agente.

Notas De Uso

Este método fornece a função da gerenciamento de parar permanentemente a *thread* de execução de um agente.

---

terminate_agent_system()

O método *terminate_agent_system*() para a execução do sistema de agentes.

Sintaxe
void *terminate_agent_system* ()
raises (*TerminateFailed*);

Parâmetros

Nenhum.

Exceções

• *TerminateFailed* . Não foi possível parar a execução do sistema de agentes.

Notas De Uso

Dependendo da implementação de *terminate_agent_system*(), o sistema de agentes pode armazenar a informação importante, e informar a todos os agentes hospedados sobre a término pretendido.

B.3 Interface do MAFFinder

A interface *MAFFinder* fornece métodos para a manutenção da base de dados dinâmica de nome e de localização de agentes, lugares, e sistemas de agentes. A interface não dita que método um cliente deve usar para encontrar um agente. Ao invés, provê a forma de localizar agentes, sistemas de agentes, e lugares que suporta um larga escala de técnicas de localização.

Há várias maneiras possíveis de encontrar um agente. Estão aqui quatro possibilidades:

• *Busca por força bruta.* Encontrar cada sistema de agentes na região, então enviar um agente viajando através de cada sistema de agentes para encontrar o agente.

• *Registrar (Logging).* Um agente de um sistema de agentes sempre deixa uma marca que informa aonde está indo. Conseqüentemente, um sistema de agentes pode sempre rastrear os registros para encontrar esse agente. Deve também haver uma maneira de coleta de lixo dos registros depois que o agente morre.

• *Registro de Agente.* Cada agente registra sua localização corrente em uma base de dados. Esta base de dados tem sempre a informação a mais atual disponível sobre a localização de um agente. Notar que registrar a posição nova de um agente adiciona *overhead* à operação *go*() do agente. Conseqüentemente, as operações de base de dados podem ser um gargalo.

• *Publicação de Agente.* Registrar somente todos os lugares estacionários. A localização de um agente é registrada somente quando o agente se anuncia. Para encontrar um agente não-anunciado, o sistema de agentes pode usar busca por *força bruta* ou *logging*.

Embora a interface do *MAFFinder* não restrinja a implementação a um certo conjunto de esquemas de localização, assume uma estrutura de base de dados subjacente que pode suportar registro, desregistro, e localização de agentes, sistemas de agentes, e lugares.

Lookup_agent()

O método *lookup_agent*() retorna as localizações dos agentes especificados. Este método pode procurar por um agente específico por nome, ou pode procurar por um conjunto de agentes que satisfaça um perfil de agente específico.

Sintaxe

```
Locations lookup_agent (
in Name agent_name,
in AgentProfile agent_profile)
raises (EntryNotFound);
```

Parâmetros

- *agent_name* . Nome do agente que o cliente ou agente deseja encontrar.

- *agent_profile* . Perfil de informação de agente que pode ser usado para especificar um critério de busca (definido em "*OMG Naming Authority Identifiers*" na página 43 da especificação).

Exceções

- *EntryNotFound* . Nenhum agente pode ser encontrado que satisfaça os critérios especificados.

Notas De Uso

Um agente pode usar este método para encontrar um outro agente ou agentes com os quais deseja se comunicar.

É muito específico da aplicação e como a semântica de nomes está organizada. Em particular, várias implementações do MAF organizam nomes de famílias e de gerações de agentes de maneiras diferentes. Havendo ambos, nome do agente e perfil do agente, permite uma busca semântica mais rica, por exemplo seria possível procurar por uma família específica de agentes, ou um geração (definida pelo nome do agente) com certas características (definidas no perfil de agente). Portanto, é possível que ambos, o nome e o perfil de agente sejam não nulos (*null*). Isto porque não existe uma união, mas dois parâmetros para este método.

O agente requisitador pode especificar por nome o agente exigido, ou especificar um ou mais agentes usando o parâmetro *agent_profile*. Note que no último caso nem todos os componentes de *agent_profile* precisam ser especificados. Entretanto, um componente não restringido na busca deve ser zero, para um tipo inteiro, ou uma *string* vazia, para outros tipos além de inteiros.

Este método não pode garantir que um agente estará na localização que o método retorna durante qualquer intervalo de tempo.

Há também uma fonte potencial de localizações falsas. Se o sistema de agentes não remove agentes terminados do sistema de agentes com *unregister_agent*(), uma chamada a

lookup_agent() por este agente retorna a localização de um agente que não está mais disponível.

lookup_agent_system()

Retorna a localização de um sistema de agentes que foi registrado com o *MAFFinder*. Este método pode procurar um nome específico de sistema de agentes, ou pode procurar por um conjunto de sistemas de agentes que satisfaçam o parâmetro *AgentSystemInfo* especificado.

Sintaxe

Locations *lookup_agent_system* (
in Name *agent_system_name*,
in AgentSystemInfo *agent_system_info*)
raises (*EntryNotFound*);

Parâmetros

- *agent_system_name* . Nome do sistema de agentes a localizar.

- *agent_system_info* . É a informação de um sistema de agentes que pode ser usada para um critério específico de busca (definido em "*OMG Naming Authority Identifiers*" na página 43 da especificação [OMG97-2]).

Exceções

- *EntryNotFound* . Nenhum agente pode ser encontrado que satisfaça os critérios especificados.

Notas De Uso

Este método pode ser usado para procurar por sistemas de agentes que estão registrados com *MAFFinder*.

O cliente requisitante pode especificar por nome o sistema de agentes desejado, ou pode especificar um ou mais sistemas de agentes usando o parâmetro *agent_system_info*. Note que no último caso nem todos os componentes do *agent_system_info* devem ser especificados. Entretanto, um componente não restringido na busca deve ser zero, para um tipo inteiro, ou uma *string* vazia, para outros tipos além de inteiros.

lookup_place()

Retorna a localização de um *lugar* (*place*) registrado com o *MAFFinder*.

Sintaxe
```
Location lookup_place (
in string place_name)
raises (EntryNotFound);
```

Parâmetros

- *place_name* . Nome do lugar a localizar.

Exceções

- *EntryNotFound* . O *lugar* especificado não está registrado com o *MAFFinder*.

Notas de Uso

Por vezes o cliente tem somente o nome do *lugar* ao qual deseja enviar um agente. O cliente usa este método para obter a localização do *lugar* especificado.

register_agent ()

Adiciona o agente nomeado à lista de agentes registrados com o *MAFFinder*. Como um agente móvel viaja, esta operação pode ser invocada muito frequentemente por um agente durante o seu ciclo de vida. Se esta operação é invocada com um *agent_name* que já existe no *MAFFinder*, esta operação substitui a informação associada (localização e perfil) com a informação na invocação mais recente.

Sintaxe
```
void register_agent (
in Name agent_name,
in Location agent_location,
in AgentProfile agent_profile)
raises (NameInvalid);
```

Parâmetros

- *agent_name* . Nome do agente a adicionar à lista.

- *agent_location* . Localização do agente.

- *agent_profile*. Informação de perfil de agente que pode ser usada para especificar critérios de busca (definido em "*OMG Naming Authority Identifiers*" na página 43 da especificação).

Exceções

• *NameInvalid* . O pedido para atualizar o *MAFFinder* falhou.

Notas De Uso

Este método fornece o registro de um agente com o *MAFFinder*.

register_agent_system()

O método *register_agent_system*() adiciona o sistema de agentes nameado à lista de sistemas de agentes registrados com o *MAFFinder*. Como um sistema de agentes é um objeto estacionário, o *MAFFinder* não permite invocações múltiplas desta operação com o mesmo nome. Quando mover um sistema de agentes, é necessário desregistrá-lo antes de registrá-lo novamente com a nova localização.

Sintaxe

void *register_agent_system* (
in Name *agent_system_name*,
in Location *agent_system_location*,
in AgentSystemInfo *agent_system_info*)
raises (*NameInvalid*);

Parâmetro

• *agent_system_name* . Nome do sistema adicionado à lista.

• *agent_system_location* . Localização do sistema adicionado.

• *agent_system_info*. Informação do sistema de agentes que pode ser usada para especificar critérios de busca (definido em "*OMG Naming Authority Identifiers*" na página 43 da especificação).

Exceções

• *NameInvalid* . Já existe um sistema de agentes registrado com o mesmo nome.

Notas De Uso

Este método registra um sistema de agentes.

register_place()

O método *register_place*() adiciona a localização de *lugar* nomeado à lista de lugares registrados com o *MAFFinder*. Como um *lugar* é um objeto estacionário, o *MAFFinder* não permite invocações múltiplas desta operação com o mesmo nome. Se um *lugar* for movido, é necessário desregistrá-lo de sua posição inicial antes de registrá-lo na nova localização.

Sintaxe
void *register_place* (
in string *place_name*,
in Location *place_location*)
raises (*NameInvalid*);

Parâmetros

• *place_name* . O nome do *lugar* a adicionar na lista

• *place_location* . A localização do *lugar* (*place*).

Exceções

• *NameInvalid* . Já existe um *lugar* registrado como o mesmo nome.

Notas De Uso

Este método é de um grupo de métodos que pode ser usado para manter uma lista de lugares com o *MAFFinder*.

unregister_agent ()

Remove o agente especificado da lista de agentes que foram registrados com o *MAFFinder*.

Sintaxe
void *unregister_agent* (
in Name *agent_name*)
raises (*EntryNotFound*);

Parâmetros

• *agent_name* . Nome do agente a remover da lista de agentes.

Exceções

• *EntryNotFound*. O agente especificado não está registrado com o *MAFFinder*.

Notas De Uso

Este método é de um grupo de métodos que pode ser usado para manter uma lista de agentes dentro um sistema de agentes.

---

unregister_agent_system()

Remove o sistema de agentes especificado da lista de sistemas de agentes que foram registrados com o *MAFFinder*.

Sintaxe
void *unregister_agent_system* (
in Name *agent_system_name*)
raises (*EntryNotFound*);

Parâmetros

• *agent_system_name* . Nome do sistema a remover da lista.

Exceções

• *EntryNotFound* . O sistema de agentes especificado não está registrado com o *MAFFinder*.

Notas De Uso

Este método é de um grupo de métodos que pode ser usado para manter uma lista de sistema de agentes registrados com o *MAFFinder*.

---

unregister_place()

Remove o lugar especificado da lista de lugares registrados com o *MAFFinder*.

Sintaxe
void *unregister_place* (
in string *place_name*)
raises (*EntryNotFound*);

Parâmetros

• *place_name* . Nome do *lugar* a remover da lista.

Exceções

• *EntryNotFound* . O *lugar* especificado não está registrado com o *MAFFinder*.

Notas De Uso

Este método é de um grupo de métodos que pode se usado para manter uma lista de lugares registrados com o *MAFFinder*.

APÊNDICE C

PortableServer

É preciso ser um realista
para descobrir a realidade.
É preciso ser um romântico
para criá-la.
Fernando Pessoa

Este Apêndice compreende o diagrama UML do *PortableServer* e da especificação em IDL:

- C.1 Diagrama UML, página 159 e
- C.2 IDL do PortableServer [OMG97-3], página 160.

C.1 Diagrama UML

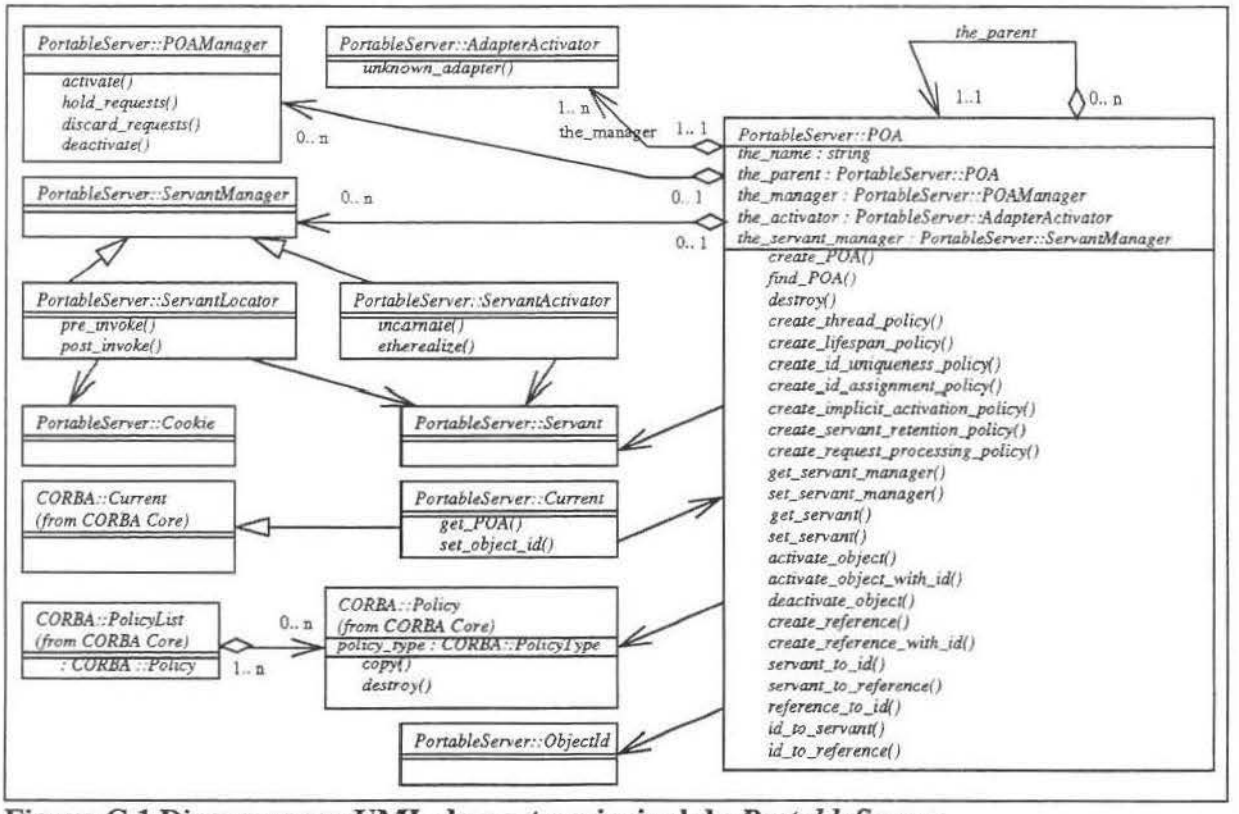


Figura C.1 Diagrama em UML da parte principal do *PortableServer*.

C.2 IDL do PortableServer [OMG97-3]

// IDL para módulo PortableServer

```
#pragma prefix "omg.org"
module PortableServer {
// Referências
    interface POA;
    native Servant;
    typedef sequence<octet> ObjectId;
    exception ForwardRequest { Object forward_reference; };
// Interfaces de Políticas
    enum ThreadPolicyValue { ORB_CTRL_MODEL, SINGLE_THREAD_MODEL };
    interface ThreadPolicy : CORBA::Policy { readonly attribute ThreadPolicyValue value; };
    enum LifespanPolicyValue { TRANSIENT, PERSISTENT };
    interface LifespanPolicy : CORBA::Policy { readonly attribute LifespanPolicyValue value; };
    enum IdUniquenessPolicyValue { UNIQUE_ID, MULTIPLE_ID };
    interface IdUniquenessPolicy : CORBA::Policy { readonly attribute IdUniquenessPolicyValue value; };
    enum IdAssignmentPolicyValue { USER_ID, SYSTEM_ID };
    interface IdAssignmentPolicy : CORBA::Policy { readonly attribute IdAssignmentPolicyValue value; };
    enum ImplicitActivationPolicyValue { IMPLICIT_ACTIVATION, NO_IMPLICIT_ACTIVATION };
    interface ImplicitActivationPolicy : CORBA::Policy { readonly attribute ImplicitActivationPolicyValue
value; };
    enum ServantRetentionPolicyValue { RETAIN, NON_RETAIN };
    interface ServantRetentionPolicy : CORBA::Policy { readonly attribute ServantRetentionPolicyValue value;
};
    enumRequestProcessingPolicyValue{USE_ACTIVE_OBJECT_MAP_ONLY,USE_DEFAULT_SERVANT,
        USE_SERVANT_MANAGER
    };
    interface RequestProcessingPolicy : CORBA::Policy { readonly attribute RequestProcessingPolicyValue
value; };
// Interface POAManager
    interface POAManager {
        exception AdapterInactive{ };
        void activate ( ) raise s(AdapterInactive);
        void hold_requests (in boolean wait_for_completion) raises(AdapterInactive);
        void discard_request (in boolean wait_for_completion) raises (AdapterInactive);
        void deactivate ( in boolean etherealize_objects, in boolean wait_for_completion) raises (AdapterInactive);
    };
// Interface AdapterActivator
    interface AdapterActivator { boolean unknown_adapter(in POA parent, in string name); };
// Interfaces ServantManager:
    interface ServantManager { };
    interface ServantActivator : ServantManager {
    Servant incarnate ( in ObjectId oid, in POA adapter ) raises (ForwardRequest);
    void etherealize ( in ObjectId oid, in POA adapter, in Servant serv,
                        in boolean cleanup_in_progress, in boolean remaining_activations );
    };
    interface ServantLocator : ServantManager {
        native Cookie;
        Servant preinvoke( in ObjectId oid, in POA adapter, in CORBA::Identifier operation, out Cookie
the_cookie )
```

```
            raises (ForwardRequest);
        void postinvoke( in ObjectId oid, in POA adapter, in CORBA::Identifier operation,
                         in Cookie the_cookie, in Servant the_servant);
    };
// POA interface
    interface POA {
        exception AdapterAlreadyExists {};
        exception AdapterInactive {};
        exception AdapterNonExistent {};
        exception InvalidPolicy { unsigned short index; };
        exception NoServant {};
        exception ObjectAlreadyActive {};
        exception ObjectNotActive {};
        exception ServantAlreadyActive {};
        exception ServantNotActive {};
        exception WrongAdapter {};
        exception WrongPolicy {};
// Criação e destruição de POA
        POA create_POA( in string adapter_name, in POAManager a_POAManager, in CORBA::PolicyList poli-
cies)
            raises (AdapterAlreadyExists, InvalidPolicy);
        POA find_POA(in string adapter_name, in boolean activate_it) raises (AdapterNonExistent);
        void destroy( in boolean etherealize_objects, in boolean wait_for_completion);
// Fábricas para objetos de Políticas
        ThreadPolicy create_thread_policy (in ThreadPolicyValue value);
        LifespanPolicy create_lifespan_policy (in LifespanPolicyValue value);
        IdUniquenessPolicy create_id_uniqueness_policy (in IdUniquenessPolicyValue value);
        IdAssignmentPolicy create_id_assignment_policy (in IdAssignmentPolicyValue value);
        ImplicitActivationPolicy create_implicit_activation_policy (in ImplicitActivationPolicyValue value);
        ServantRetentionPolicy create_servant_retention_policy (in ServantRetentionPolicyValue value);
        RequestProcessingPolicy create_request_processing_policy (in RequestProcessingPolicyValue value);
// Atributos do POA
        readonly attribute string the_name;
        readonly attribute POA the_parent;
        readonly attribute POAManager the_POAManager;
        attribute AdapterActivator the_activator;
// Registro do Servant Manager:
        ServantManager get_servant_manager ( ) raises (WrongPolicy);
        void set_servant_manager( in ServantManager imgr ) raises (WrongPolicy);
// Operações para a política USE_DEFAULT_SERVANT
        Servant get_servant ( ) raises (NoServant, WrongPolicy);
        void set_servant (in Servant p_servant) raises (WrongPolicy);
// Ativação e desativação de objetos
        ObjectId activate_object( in Servant p_servant ) raises (ServantAlreadyActive, WrongPolicy);
        void activate_object_with_id( in ObjectId id, in Servant p_servant)
            raises ( ServantAlreadyActive, ObjectAlreadyActive, WrongPolicy);
        void deactivate_object (in ObjectId oid) raises (ObjectNotActive, WrongPolicy);
// Operações de criação de referências
        Object create_reference ( in CORBA::RepositoryId intf ) raises (WrongPolicy);
        Object create_reference_with_id ( in ObjectId oid, in CORBA::RepositoryId intf ) raises (WrongPolicy);
// Operações de mapeamento de identidade:
        ObjectId servant_to_id(in Servant p_servant) raises (ServantNotActive, WrongPolicy);
        Object servant_to_reference(in Servant p_servant) raises (ServantNotActive, WrongPolicy);
```

```
        Servant reference_to_servant(in Object reference) raises (ObjectNotActive, WrongAdapter, WrongPol-
icy);
        ObjectId reference_to_id(in Object reference) raises (WrongAdapter, WrongPolicy);
        Servant id_to_servant(in ObjectId oid) raises (ObjectNotActive, WrongPolicy);
        Object id_to_reference(in ObjectId oid) raises (ObjectNotActive, WrongPolicy);
    };
// Interface Current
    interface Current : CORBA::Current {
        exception NoContext { };
        POA get_POA() raises (NoContext);
        ObjectId get_object_id() raises (NoContext);
    };
};
```

APÊNDICE D *IT_daemon*

O que a lagarta
chama de fim do mundo,
o mestre chama de borboleta.
Richard Bach

Este Apêndice compreende a IDL do Gerente de Ativação da OrbixWeb (*IT_daemon*) [OrbixWeb97-1, OrbixWeb97-2].

```
interface IT_daemon{
    boolean lookUp( in string service,out stringSeq hostList, in octet hops,in string tag);
    boolean addHostsToServer(in string server,in stringSeq hostList);
    boolean addHostsToGroup(in string group,in stringSeq hostList);
    boolean addGroupsToServer(in string server,in stringSeq groupList);
    boolean delHostsFromServer(in string server,in stringSeq hostList);
    boolean delHostsFromGroup(in string group,in stringSeq hostList);
    boolean delGroupsFromServer(in string server,in stringSeq groupList);
    boolean listHostsInServer(in string server,out stringSeq hostList);
    boolean listHostsInGroup(in string group,out stringSeq hostList);
    boolean listGroupsInServer(in string server,out stringSeq groupList);
    enum LaunchStatus { inActive, manualLaunch, automaticLaunch };
    struct serverDetails {
        string server;
        string marker;
        string principal;
        string code;
        string comms;
        string port;
        unsigned long OSspecific;
        LaunchStatus status; };
    void listActiveServers(out serverDetailsSeq servers);
    void killServer(in string name, in string marker);
    void newSharedServer(in string serverName,in stringSeq marker,
        in stringSeq launchCommand, in unsigned long mode_flags);
    public void newSharedServer2(String serverName,
        String[] marker,
        String[] launchCommand,
        int mode_flags,
        int nservers,
        int wellKnownPort);
    void newUnSharedServer(in string serverName,
        in stringSeq marker,
        in stringSeq launchCommand,
        in unsigned long mode_flags);
    void newPerMethodServer(in string serverName,in stringSeq method, in stringSeq launchCommand);
```

```
    void listServers(in string subdir,out stringSeq servers);
    void deleteServer(in string serverName);
    boolean serverExists(in string serverName);
    public void getIIOPDetails( String serverName,String markerName,
        String methodName,org.omg.CORBA.StringHolder iiopPort,
        org.omg.CORBA.StringHolder activationPolicy);
    public void getImplementationDetails( String serverName,
        String markerName,String methodName,
        org.omg.CORBA.StringHolder codeProtocol,
        org.omg.CORBA.StringHolder commsProtocol,
        org.omg.CORBA.StringHolder commsPort,
        org.omg.CORBA.StringHolder activationPolicy ) ;
    void getServer(in string serverName,out string commsProtocol,
        out string codeProtocol,out string activationPolicy,
        out unsigned long mode_flags,out string owner,
        out string invokeList,out string launchList,
        out stringSeq markers,out stringSeq methods,
        out stringSeq commands);
    public void getServer2(String serverName,
        org.omg.CORBA.StringHolder commsProtocol,
        org.omg.CORBA.StringHolder codeProtocol,
        org.omg.CORBA.StringHolder activationPolicy,
        org.omg.CORBA.IntHolder mode_flags,
        org.omg.CORBA.StringHolder owner,
        org.omg.CORBA.StringHolder invokeList,
        org.omg.CORBA.StringHolder launchList,
        org.omg.CORBA.IntHolder nservers,
        org.omg.CORBA.IntHolder port,
        stringSeqHolder markers,
        stringSeqHolder methods,
        stringSeqHolder commands) ;
    void addUnsharedMarker(in string serverName,in string markerName, in string newCommand);
    void removeUnsharedMarker(in string serverName, in string markerName);
    void addSharedMarker(in string serverName,in string markerName, in string newCommand);
    void removeSharedMarker(in string serverName,in string markerName);
    void addMethod(in string serverName,in string methodName, in string newCommand);
    void removeMethod(in string serverName,in string methodName);
    void newDirectory(in string dirName);
    void deleteDirectory(in string dirName,in boolean deleteChildren);
    void changeOwnerServer(in string new_owner,in string serverName);
    void addInvokeRights(in string userGroup,in string serverName);
    public void registerPersistentServer(String serverName,int serverPid,
        org.omg.CORBA.StringHolder codeProtocol,
        org.omg.CORBA.StringHolder commsProtocol,
        org.omg.CORBA.StringHolder commsPort);
    void removeInvokeRights(in string userGroup,in string serverName);
    void addLaunchRights(in string userGroup,in string serverName);
    void removeLaunchRights(in string userGroup,in string serverName);
    void addInvokeRightsDir(in string userGroup,in string dirName);
    void removeInvokeRightsDir(in string userGroup,in string dirName);
    void addLaunchRightsDir(in string userGroup,in string dirName);
    void removeLaunchRightsDir(in string userGroup,in string dirName);
};
```

Índice Remissivo

Life is an energy process.
Like every energy-process, it is in principle irreversible
and is therefore directed towards a goal.
That goal is a state of rest. [...]
The end of every process is its goal.
Carl Jung

A

AC 15, 23
Admin 53
Adaptabilidade 91
Adaptador de Objetos 125
Agência 5, 5, 32, 55, 58, 82, 88
Agência Orientada a Serviços 34
Agente 1, 25, 32, 34, 49, 51
Agentes em_largura 98
Agentes em_profundidade 98
Agente Estacionario 25
Agente Gerenciamento 102
Agente Móvel 5, 8, 9, 21, 24, 25
Agente Móvel CORBA 49
Aglets 2, 19, 24
Aleatório 19
Ambiente computacional 15
Armazena 71
Arquitetura 1, 31, 32, 97
Assincrono 16
Ativação 39
Autônoma 41
Autonomia 20
Autoridade 27
Autoridade de Agentes 25
Autorização 38
Avaliação Remota 8, 9

B

Balanceamento 19, 88
Base de Código 27
Benchmarks 48, 80, 82, 83, 84
Bind 58, 67, 68, 72, 80, 106
Bind_context 72
Bind_new_context 72, 73
Bloqueia 71
Busca 68
Byte-code 23, 87

C

C++ 60, 77, 86, 123
Cache 37, 63, 70
Callback 19, 59
Características Gerais de Agentes 18
Catálogo 32, 36, 37, 38, 40, 52, 58
CfMAF 65, 135
Chorus 2
Classes 85
Clear 70
Ciclo de Vida 35
Cliente 1
Cliente-Servidor 8
Clonagem Remota 16
COBOL 123
COD 8, 10, 13, 14, 15, 21
Codebase 27
Código Enviado 16
Código Móvel 2, 8, 23
Código Puxado 16
Código Sob Demanda 8, 9
Comparações 13, 86
Componentes 1, 31, 34
Comunicação 27
Comunicação-remota 21
Concorrente 6
Configuração 20
Contabilidade 97
Contains 70
Containskey 70
Contribuições 2
COOL 2
CORBA 2, 19, 24, 31, 34, 49, 51, 55, 56, 63, 77, 92, 123
CS 10, 12, 14, 17, 21

D

DCOM 25
Desabilita_ajuste 69
Desabilita_migração 69
Desempenho 47, 97
Desserialização 27
Desregistra 71
Destroy 73
Dinâmica 20
Diretórios 51
Disponibilidade 1, 31, 35, 38, 48, 55, 62, 63, 68, 88, 92, 94
Disponibilidade de um Agente 36
Disponibilidade de uma Agência 35

Distribuição 11
Domínio 32, 38, 51, 62
Domínios Distribuídos Dinâmicos 38
DynamicPropEval 53

E
Escolhe 69
Esquemas de gerenciamento 93
Experimentação 105
Export 38
Esqueleto 124
Espaço de Nomes 52
Estado 38
Estado de Agente 7, 25
Estado de Execução de Agentes 25
Estratégia 39
Estratégia de seleção 23
Ethernet 78
ExportAg 75

F
Filtros 101
Firewall 27
Flexibilidade 20
Fluxograma 50
FOKUS 87
Fragmento 16

G
Gerenciamento 34, 47, 89, 91
Gerenciamento de Código e Estado 16
Gerente de Ativação 34, 56, 60, 88, 163
Get 70
GMD 87
Grasshopper 24, 63, 56, 97
Grupos 34

I
IDL 55, 66, 123, 159
IIOP 61, 123, 125
Implementação 63, 100
ImportAg 75
Inicialização 51
Invocação Dinâmica 125
Interações 1
Interface 38, 82
interface ORB 65
Inicialização 22
Interconexão 27
Internet 60
IOR 51
Itinerário 71
IT_daemon 56, 61, 81, 163

J
Java 21, 23, 55, 60, 78, 86, 123, 130
Java/RMI 24
JavaIDL 21, 64, 123, 100
javaORBs 123
JDMK 96
JIT 87
JVM 23, 86

K
keys 70

L
Link 53
List 73
Localidade 28
Localização 23, 31, 40, 88
Localização de Agentes 26
Locator 61, 80
Lookup 53
Lugar 26

M
MA 8, 10, 12, 17, 21
MAFAgentSystem 56, 139
MAFFinder 56, 151
MAGNA 24, 97
Marshaling 79
MASIF 5, 24, 27, 55, 56, 97, 135
Mecanismo de Mobilidade 16
Medidas 84
Métricas 47
Middleware 19, 34, 92
Migração 2, 16, 21, 77
Migração Transparente 59
Migração Transparente em Sistemas de Agentes 39
Mobilidade 1, 20, 32, 35, 39, 49, 58, 62, 63, 71, 99, 100
Mobilidade Explícita 1,19, 25, 41, 63
Mobilidade Forte 15, 16, 17
Mobilidade Fraca 15, 16, 17
Mobilidade Implícita 1, 19, 25, 41, 63
ModifyAg 75
MomentA 97
Monitoração 105, 97
Motivação 2
Móvel 80
Multi-agentes 1, 19

N
Não Móvel 17
Nomes 1, 36, 40, 63, 72, 104, 130
Nomes de Agentes 25
New_context 73

O
Object_to_string 67
ODP 3, 38, 51
Odyssey 24
Oferta de Disponibilidade 47
OMG 97
ORB 32, 56, 82
Orbix 60, 77, 82
Orbixd 60, 61
Orbixdj 65
OrbixWeb 55, 60, 63, 64, 100, 123

P
Padrão 38
Paradigmas 13, 17, 21
Paradigmas de Código Móvel sobre CORBA 21
Perfil 26
Persistência 35
Place 26

Plataforma de Agente 63
POA 61, 63, 65
Portabilidade 65
PortableServer 159, 65
Proativa 16
Profundidade 32
Propriedades Dinâmicas 49
Prototipagem 77
Proxy 7, 37, 38, 53
Put 70

Q
QoS 36, 80

R
Razão 71
Reativa 16
Rebind 74
Rebind_context 73
Recebe 72
Rechamadas 19, 23, 59
Referências de objetos 61
Região 26, 57, 62
Region_registration 38
Register 38
Registra 71
Registrados 38
Registro 39
Registro Remoto 34
Remove 70
Repositório de Estado 53
Repositório de Implementação 34, 53, 125, 77, 82, 88
Repositório de Interface 53
Repositórios 53
Requisições 19, 23
Resolve 74
REV 8, 10, 12, 14, 17, 21
RMI 21
RMI/IDL 21
RPC 25

S
SCM 15
Segurança 20, 47
Seleção 23, 63, 69
Sequencial 6
Seqüências 78
Serialização 27
Serialização 26
Servant 65, 125
ServantActivator 61, 65
ServantLocator 61, 65
ServantManager 65
Serviços 31, 33
Serviço de Armazenamento 51
Serviço de Catálogo 52
Serviço de Nomes 25, 51
Serviços disponíveis 32
Serviços não-disponívei 32
Servidor 1
Síncrono 16
Sistema de Agente 5, 55
Sistemas de código móvel 15
Sítios 1
Smalltalk 123
Solaris 55
Sun Microsystems 23
Suporte a Mobilidade 49, 50
SunOS 79
Stub 124
String_to_object 67

T
Tamanho do estado do agente 42
Tcl 25
Tecnologias 15, 17
Telescript 24
Tipo de Sistema de Agentes 26
Tolerância a falhas 23
Trader 1, 32, 34, 52, 63, 75, 87, 104
TraderComponents 53
Tráfego 7, 11, 41
Trajetória 7, 32
Transação 34
Transparência 39, 58, 63, 67, 80, 99
Transparente 1, 31

U
UE 15
UML 66, 159
Unbind 74
Unidades de execução 15

V
Visibroker 64, 87
Voyager 24

Símbolos
_bind 80