Uma Infra-Estrutura para Migração de Objetos CORBA Implementados em Java

Helves Humberto Domingues

Dissertação Apresentada ao
Instituto de Matemática e Estatística da
Universidade de São Paulo
como Parte dos Requisitos
para a Obtenção do Grau de
Mestre em Ciência da Computação

Área de Concentração: Sistemas Distribuídos
Orientador: Prof. Dr. Francisco Reverbel

São Paulo — Dezembro de 2001

Uma Infra-Estrutura para Migração de Objetos CORBA Implementados em Java

Este exemplar corresponde à redação
final da dissertação devidamente corrigida
e defendida por Helves Humberto Domingues
e aprovada pela comissão julgadora.

São Paulo, 18 de dezembro de 2001.

Banca examinadora:

- Prof. Dr. Francisco Reverbel (orientador) - IME - USP
- Profa. Dra. Graça Bressan - Escola Politécnica da USP
- Prof. Dr. Alfredo Goldman - IME - USP

Agradecimentos

- Ao Prof. Reverbel, pelo tempo que dedicou na minha orientação. Pela paciência e compreensão das minhas deficiências, e além de tudo isto, por tudo que me ensinou.
- À Profa. Graça Bressan e ao Prof. Alfredo Goldman, por aceitar fazer parte da banca examinadora e pelas sugestões apontadas após a exposição da dissertação.
- Ao Prof. Merklen, cuja participação foi imprescindível no processo que me permitiu a continuação do programa de mestrado.
- Ao Wellington Brigante, que, como meu chefe imediato, compreendeu a necessidade de obtenção deste grau e concedeu me o tempo necessário para a conclusão.
- À minha querida Katia, minha esposa, por todas suas privações, aconselhamentos, incentivos, que foram cruciais para que eu consiguisse iniciar e terminar o mestrado.
- A minha mãe, Zeferina Eva Lima Domingues, que sempre motivou os meus estudos, pela sua força de enfrentar grandes desafios, inspirando-me a vencer esta etapa.
- Aos meus irmãos, João Batista, Moisés, Augusto, Simone, Valéria e Elaine, meus irmãos, meus amigos, pelo incentivo, dicas e apoio para a conclusão do mestrado.

Resumo

Esta dissertação propõe uma infra-estrutura para migração de objetos CORBA implementados em Java. Dois objetivos nortearam o projeto dessa infra-estrutura. Um deles foi possibilitar a migração de objetos individuais, isto é, a migração de somente um ou de alguns dos objetos residentes num servidor. O segundo objetivo foi prover transparência de migração, ou seja, preservar a validade das referências para os objetos que migraram e permitir que clientes continuem usando tais referências para invocar métodos dos objetos mesmo durante o processo de migração.

A infra-estrutura de migração consiste num conjunto de servidores CORBA que fornecem contextos de execução para objetos móveis. Cada um desses servidores de mobilidade funciona como hospedeiro para um conjunto de objetos CORBA implementados em Java. Estes objetos podem migrar de um servidor de mobilidade para outro. Os servidores de mobilidade são genéricos, podendo hospedar objetos com diferentes interfaces IDL, desde que implementados seguindo certas regras. Um servidor de mobilidade não precisa conhecer, em tempo de sua compilação, nem as interfaces IDL nem as classes Java correspondentes aos objetos móveis que ele abrigará.

O uso de CORBA e Java foi motivado tanto por sua relevância e aceitação quanto por algumas de suas características. A transparência de localização e o mecanismo de *location forward* oferecidos por CORBA foram cruciais para este trabalho. A independência de plataforma, a mobilidade de código e as facilidades para serialização de objetos oferecidas por Java foram igualmente importantes.

Abstract

This dissertation proposes an infrastructure to support migration of CORBA objects implemented in Java. Two goals drove the design of this infrastructure. One of them was to allow the migration of individual objects, that is, the migration of just one or some of the objects hosted by a server process. The second goal was to provide migration transparency, which means preserving the validity of references for objects that have migrated. Clients may use such references to perform method invocations even during the migration.

The proposed infrastructure consists af a set of CORBA servers that provide execution contexts for movable objects. Each such mobility server acts as a container for a set of CORBA objects implemented in Java. These objects can migrate from one mobility server to another. The moblity servers are generic, in the sense that they can host objects with different IDL interfaces, provided that these objects have been implemented following some rules. A mobility server does not need to know, at compile time, neither the IDL interfaces nor the Java classes of the movable objects it will eventually host.

The usage of CORBA and Java was motivated for their relevance and industry support, as well for some of their features. CORBA provides location transparency and a location forward mechanism. Both were crucial for this work. Java offers platform independence, code mobility and object serialization facilities, features that were equally important for us.

Sumário

Lista de Figuras

Lista de Tabelas

Introdução

Os objetos em uma linguagem de programação têm várias características importantes como identidade, estado e comportamento. Todas elas fornecem recursos para modelar uma aplicação, pois aproximam os objetos físicos, como os cartões magnéticos, cheques e pessoas, aos objetos em uma linguagem de programação, objetos lógicos. Quanto mais semelhanças existirem entre os objetos lógicos e os objetos físicos, mais fácil será a modelagem da aplicação.

As pessoas se deslocam, os cartões magnéticos e os cheques são deslocados de um lado para outro e, deste modo, espera-se que os objetos lógicos que os representam também possam se deslocar ou ser deslocados. O ato de deslocar ou ser deslocado está sendo chamado neste texto de migração. No dicionário Aurélio a palavra "migrar" tem a seguinte definição : "Mudar periodicamente, ou passar de uma região para outra, de um país para outro".

Em 1989, o *Object Management Group (OMG)* foi formado para promover o desenvolvimento de aplicações distribuídas portáveis, em sistemas heterogêneos, através da padronização de serviços e interfaces. Utilizando conceitos da orientação de objetos, as aplicações distribuídas criam o conceito de objetos distribuídos, isto é, os objetos que uma aplicação utiliza não estão mais implementados somente na própria aplicação, mas podem estar em uma outra aplicação e, possivelmente, em uma outra máquina.

As primeiras especificações chaves do OMG foram a *Object Management Architecture (OMA)* [16] e a especificação *Common Object Request Broker Achitecture (CORBA)* [18]. A especificação OMA é composta por dois modelos relacionados, "Object Model" e "Reference Model". O "Object Model" define como são os objetos distribuídos em um ambiente heterogêneo, enquanto o "Reference Model" caracteriza as interações entre estes objetos. Mediando essas interações aparece, no "Reference Model", o componente *Object Request Broker (ORB)*. A especificação CORBA padroniza os ORBs e define uma *Interface Defi-*

nition Language (IDL) para descrever a interface de um objeto, de forma independente de plataformas e linguagens de programação.

Dentre as várias linguagens de programação orientadas a objetos, Java [7] se destaca pelo excelente suporte para mobilidade de código. Os "applets" [3] são uma evidência disso. A máquina virtual Java, que é uma camada acima da máquina real, fornece a infra-estrutura necessária para que um objeto Java possa existir em qualquer tipo de plataforma. A independência de plataforma e o suporte para mobilidade de código em um ambiente heterogêneo, proporcionadas pela linguagem Java, facilitam e incentivam a migração de objetos.

Objetivos deste Trabalho

Este documento propõe uma infra-estrutura para migração de objetos CORBA implementados em Java. Esta infra-estrutura foi concebida tendo em vista os seguintes objetivos:

Migração de objetos individuais Nosso interesse é a migração de objetos individuais, isto é, a migração de um ou de alguns dos objetos implementados por um servidor. O problema de migração de servidores (com a conseqüente migração "em massa" de todos objetos num servidor) é adequadamente resolvido pelos repositórios de implementações integrados a muitos produtos CORBA.

Transparência de migração A migração de um objeto CORBA para outro servidor deve ser transparente para os clientes do objeto, preservando a validade de suas referências. A infra-estrutura proposta assegura que todas as referências para um objeto CORBA permaneçam válidas após a migração.

Motivação

Uma infra-estrutura para migração de objetos é importante em várias áreas de aplicação. Algumas dessas áreas são:

Gerenciamento dinâmico de redes Cada nó da rede poderia ter um servidor que recebesse objetos, também chamado de servidor elástico [6]. Um administrador da rede poderia solicitar a migração de um objeto para um certo nó. Chegando lá, o objeto

poderia executar todas as instruções disponíveis localmente para o gerenciamento de rede.

Gerenciamento de workflow Um workflow pressupõe a passagem de uma entidade, possivelmente representado por um objeto, de uma pessoa para outra ou de um departamento para outro. Esta passagem poderia ser uma migração do objeto para o servidor mais próximo de uma pessoa ou departamento, permitindo a visualização e o trabalho com o objeto migrado independentemente do funcionamento do servidor de origem. A OMG já tem uma especificação sobre gerenciamento de workflow [17].

Balanceamento de carga Uma aplicação administradora poderia perceber que existem muitos objetos em um certo servidor e migrar alguns objetos para um outro servidor com poucos objetos. Esta aplicação ficaria mais simples se todos os objetos gastassem uma quantidade similar de recursos do servidor. Caso contrário, a decisão de qual objeto migrar pode ser complexa. De qualquer modo, a migração de objetos pode ser utilizado para o balanceamento de carga de servidores. A OMG não tem ainda uma especificação para este assunto, mas já tem uma RFP [19].

Organização do Trabalho

O primeiro e o segundo capítulos são os pré-requisitos para o entendimento deste trabalho. Nestes dois capítulos são abordados os assuntos mais importantes de CORBA e Java. Deste modo, foram escolhidos somente os assuntos que serão utilizados posteriormente.

O terceiro capítulo descreve alguns trabalhos relacionados a migração de objetos e a especificação Life Cycle. Os trabalhos foram escolhidos porque utilizavam CORBA ou porque foram escritos em Java. Estes trabalhos e a especificação serviram como base conceitual para o desenvolvimento da infra-estrutura proposta.

O quarto capítulo fornece o primeiro contato com a infra-estrutura. Este capítulo apresentará uma visão geral, onde será possível visualizar todas as interfaces envolvidas e quais componentes existem para que a infra-estrutura funcione. Também é descrito como um desenvolvedor poderá implementar um objeto móvel. A preocupação será de explicar a infra-estrutura na visão do seu principal usuário: o desenvolvedor de objetos móveis.

O quinto capítulo descreve detalhadamente como foi implementada a infra-estrutura, mostrando as estruturas de dados envolvidas, o protocolo de migração e os trechos mais

relevantes do código implementado. O resultado real deste capítulo é a implementação que servirá tanto como exemplo de uma realização da arquitetura proposta neste trabalho, quanto como validação das decisões tomadas.

O sexto capítulo descreve o ambiente de desenvolvimento, os arquivos-fonte de nosso protótipo de infra-estrutura e os testes com ele realizados, além de relacionar algumas melhorias que poderiam ser incorporadas ao protótipo.

O sétimo capítulo traz nossas considerações finais. Este capítulo compara com outros sistemas a infra-estrutura aqui proposta, ressalta a contribuição principal deste trabalho e, finalmente, relaciona ítens para investigação futura.

Capítulo 1

CORBA

A especificação *Common Object Request Broker Architecture (CORBA)* é a resposta do *Object Management Group (OMG)* às necessidades de interoperabilidade entre os vários tipos de hardware e software que proliferaram rapidamente nos últimos anos. Deste modo, CORBA permite que as aplicações se comuniquem uma com a outra, independentemente da plataforma (hardware e software) que elas utilizam.

A primeira versão de CORBA foi publicada pelo OMG em 1991. Esta versão introduziu a *Interface Definition Language* (IDL), definiu o *Object Request Broker* (ORB) e especificou a forma de interação entre aplicações clientes e objetos existentes em aplicações servidoras. Clientes e objetos interagem através de chamada remota de métodos, essencialmente uma extensão orientada a objetos do mecanismo de chamada remota de procedimentos (RPC) tradicionalmente usado em sistemas distribuídos. Versões seguintes da especificação CORBA definiram questões não contempladas pela versão inicial, tais como a padronização de protocolos e de um formato para transporte e armazenamento de referências para objetos, mapeamentos de IDL para diversas linguagens de programação e problemas de portabilidade de servidores. CORBA 2.0, publicada em 1995, especificou o protocolo GIOP/IIOP, um formato de uma *object reference* interoperável (IOR), e o mapeamento de IDL para C++. CORBA 2.2, publicada em 1998, resolveu o problema da portabilidade de servidores com a padronização do *Portable Object Adapter* (POA).

Abordaremos, nas próximas seções, somente os aspectos de CORBA indispensáveis para a compreensão deste trabalho. Para um estudo completo, sugerimos a leitura de [8].

1.1 ORB - Object Request Broker

O ORB provê um mecanismo para as aplicações clientes enviarem requisições a objetos remotos, existentes em aplicações servidoras. Este mecanismo tem as seguintes características:

Independência de plataforma. As aplicações, cliente e servidora, podem residir em máquinas com arquiteturas e/ou sistemas operacionais diferentes.

Independência de linguagem. As aplicações podem ser implementadas em diferentes linguagens de programação.

Transparência de localização. A aplicação cliente não precisa conhecer a localização do objeto remoto, apenas deve ter uma referência para o objeto CORBA para gerar uma requisição, pois o ORB saberá encaminhá-la para o destino correto.

O ORB, com as características apresentadas acima, permite, por exemplo, que duas aplicações se comuniquem, mesmo que uma seja executada numa máquina Sun e a outra num Macintosh, uma seja escrita em Java e outra em C++, uma não conheça o endereço IP da máquina na qual a outra roda, nem tampouco a porta TCP usada pela outra aplicação.

Estas características são bastante interessantes para trabalhos de pesquisa na área de mobilidade de objetos, pois estes enfrentam também os problemas de plataforma, linguagem e localização. A localização é um problema particularmente importante, pois um objeto que migra com muita freqüência pode gerar grande dificuldade para as suas aplicações clientes, caso estas tenham de se preocupar com a localização do objeto.

A figura 1.1 mostra o principal objetivo de um ORB: *o transporte de uma requisição proveniente de uma aplicação cliente para a implementação do objeto CORBA na aplicação servidora.*

Os objetos que residem numa aplicação servidora e recebem invocações remotas de métodos têm suas interfaces definidas em IDL, uma linguagem puramente declarativa. O objetivo desta linguagem é descrever a interface de um objeto de forma independente de plataforma e de linguagem de programação, bem como permitir a geração automática de código. Uma parte do código gerado automaticamente por um compilador IDL será utilizada pela aplicação cliente e outra parte pela aplicação servidora.

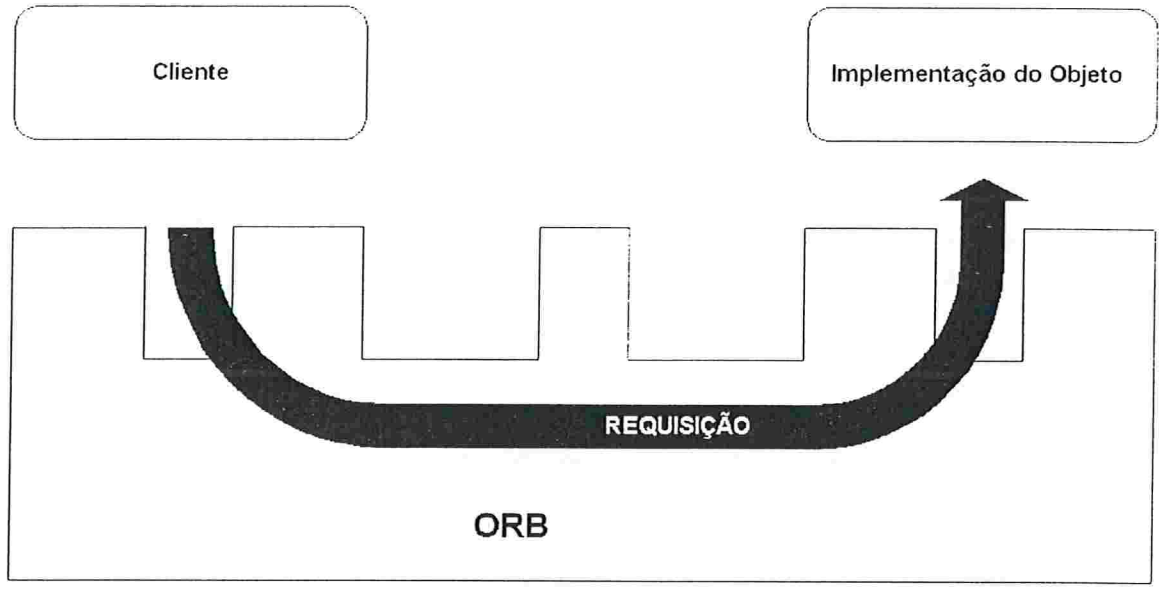

Figura 1.1: Requisição enviada por meio do ORB

1.2 Principais Elementos de CORBA

Os principais elementos de CORBA, mostrados na figura 1.2, são descritos a seguir.

Stub IDL. Este elemento, gerado automaticamente a partir de uma interface definida em IDL, é incorporado à aplicação cliente e tem a função de representar um objeto CORBA. A aplicação cliente, quando aciona um método do objeto CORBA, está, na verdade, acionando um método do *Stub IDL*, o qual constrói uma mensagem de requisição e a envia para a aplicação servidora, por meio do núcleo do ORB. O stub inclui na mensagem de requisição os argumentos do método, codificados num formato neutro (independente de linguagem de programação e arquitetura de hardware).

Interface do ORB. Esta interface disponibiliza operações que podem ser chamadas tanto por aplicações clientes como por aplicações servidoras.

Adaptador de objetos. Este é o elemento responsável pela geração de referências para objetos e pelo direcionamento das requisições recebidas por um servidor. Cada uma dessas requisições passa pelo adaptador de objetos, que as direciona para a implementação do objeto alvo da requisição.

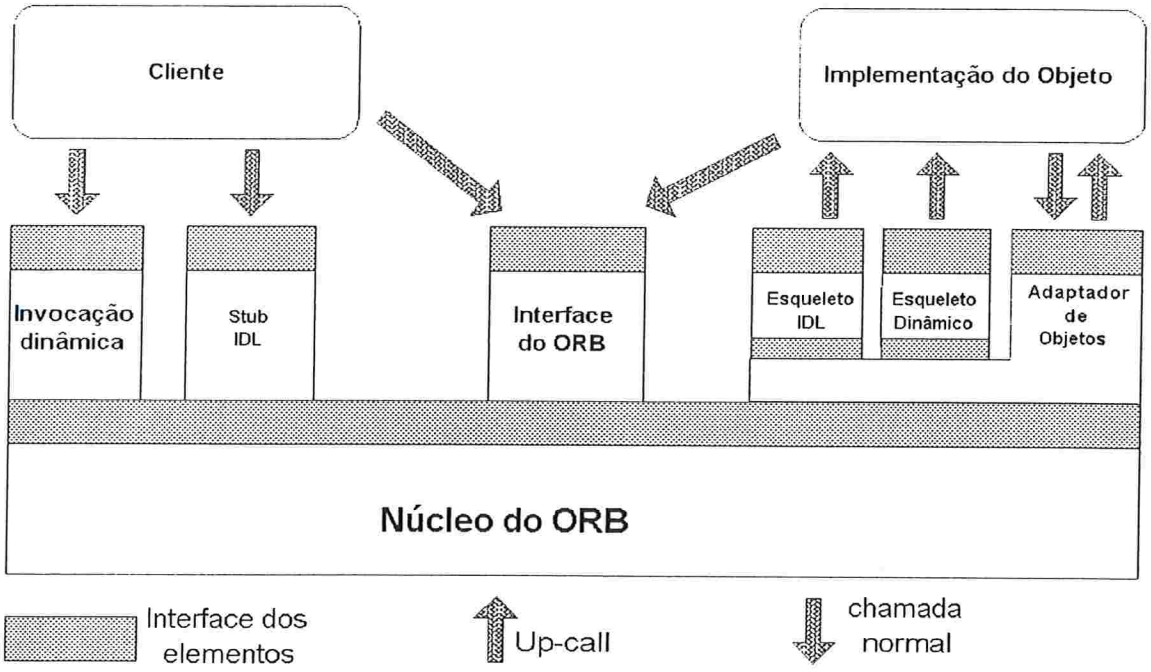

Figura 1.2: Principais elementos de CORBA

Esqueleto IDL. Este elemento é também gerado automaticamente a partir de uma interface definida em IDL. Ele recebe do adaptador de objetos mensagens de requisição e as converte em invocações de métodos sobre objetos da aplicação servidora. Os resultados de cada invocação de método são empacotados pelo esqueleto IDL numa mensagem de resposta que, com a ajuda do núcleo do ORB, chega a quem fez a requisição.

Interface de Invocação Dinâmica. Esta interface permite que um cliente invoque métodos de objetos cujas interfaces ele não conhecia em tempo de compilação. Este elemento é útil para "clientes genéricos", tais como depuradores e *object browsers.*

Esqueleto Dinâmico. Este elemento permite que um servidor implemente objetos cujas interfaces ele não conheceu em tempo de compilação. Seu propósito é permitir a construção de *gateways*, interligando um ambiente CORBA a outro ambiente de objetos distribuídos, como DCOM.

Por ser um elemento especialmente importante para este trabalho, o adaptador de objetos será descrito detalhadamente em seções futuras. O esqueleto dinâmico e a interface de

invocação dinâmica não foram empregados neste trabalho.

1.3 Fluxo de Requisições em CORBA

Quando um cliente utiliza um objeto CORBA, os elementos acima descritos funcionam em conjunto da seguinte maneira:

1. O cliente pode chamar um método de um objeto usando a Interface de Invocação Dinâmica ou usando um *Stub IDL*. Em ambos os casos, uma requisição é direcionada para a parte do núcleo do ORB incorporada ao cliente.
2. O núcleo do ORB cliente transmite a mensagem de requisição para o núcleo do ORB incorporado à aplicação servidora.
3. O núcleo do ORB do servidor direciona a requisição para o adaptador de objetos.
4. O adaptador de objetos repassa a requisição para o esqueleto IDL[1].
5. O esqueleto IDL aciona o método do objeto que implementa a interface CORBA.
6. O esqueleto IDL transforma o resultado do método acionado em uma mensagem de resposta, que fará o caminho contrário da mensagem de requisição, até chegar ao cliente.

1.4 Interoperabilidade: IOR, GIOP e IIOP

Antes da versão 2.0, a maior crítica a CORBA era em relação aos problemas de interoperabilidade dos ORBs, ou seja, uma implementação de ORB não se comunicava com uma outra. Existiam dois grandes problemas, um deles era a falta de uma especificação do protocolo de comunicação. Assim, cada ORB desenvolveu o seu protocolo ou usou outro já pronto, desenvolvido pela área de sistemas distribuídos, provocando problemas de comunicação entre eles. Na especificação CORBA 2.0, respondendo às críticas, criou-se o *General Inter-ORB Protocol (GIOP)*, que descreve uma arquitetura de interoperabilidade de ORBs. GIOP é um protocolo abstrato que especifica um conjunto padrão de mensagens

[1] Aqui estamos supondo que o servidor usa esqueletos IDL. O caso (bem menos comum) de um servidor que usa o esqueleto dinâmico é análogo.

e a sua sintaxe de transferências sobre um canal genérico de transporte orientado à conexão. O protocolo concreto, sobre o TCP/IP, é o *Internet Inter-ORB Protocol (IIOP)*, que detalha como o GIOP funciona com o TCP/IP.

O outro grande problema de interoperabilidade dos ORBs era a falta da padronização do formato das referências de objetos CORBA que são estruturas de dados contendo informações necessárias aos ORBs, para que eles estabeleçam uma comunicação entre clientes e os objetos CORBA. O padrão definido foi o *Interoperable Object Reference (IOR)*, que identifica um ou mais protocolos GIOP e, para cada um, contém informações específicas. No caso do IIOP, uma IOR contém, dentre outras informações, o nome do host, a porta TCP/IP e uma *object key* que identifica o objeto CORBA, dentre outros, na aplicação servidora.

1.4.1 IOR - Referências CORBA

Uma referência a um objeto CORBA é análoga a uma referência a um objeto Java, mas a maior diferença é a localização do objeto CORBA que, diferente do objeto Java, pode estar em outro processo e, possivelmente, em outra máquina. Além desta diferença, podemos destacar as seguintes características importantes das referências CORBA:

- As referências podem ser persistentes.

  Clientes e servidores podem converter uma referência em uma string e gravar no disco. Algum tempo mais tarde, aquela string pode voltar a ser uma referência e denotar o mesmo objeto CORBA.

- As referências podem ser interoperáveis.

  CORBA especifica um formato padrão para as referências, possibilitando um ORB usar uma referência criada por outro ORB. Por esta razão, o padrão é chamado de referência de objeto interoperável ou, em inglês, *Interoperable Object Reference (IOR)*.

- As referências são opacas.

  As referências contêm várias partes padronizadas, que são iguais para todos os ORBs, mas também contêm informações proprietárias, específicas para cada ORB. Para permitir que os códigos-fonte do cliente e do servidor sejam portáveis entre diferentes ORBs, estes códigos não acessam diretamente as informações contidas nas

referências, mas apenas através de uma interface padronizada. Este encapsulamento da referência a um objeto é um aspecto chave em CORBA.

Conteúdo de Uma Referência

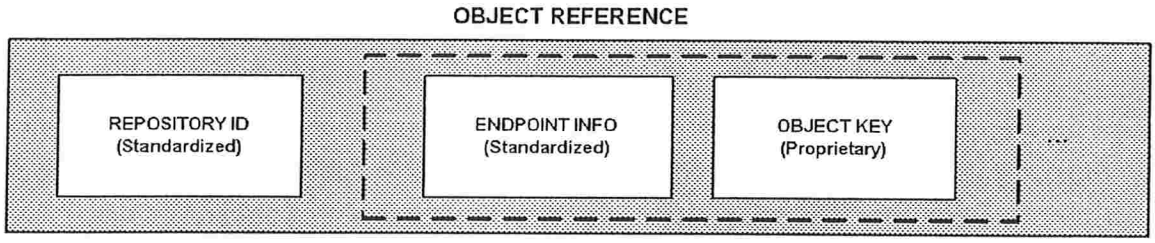

Figura 1.3: Principais elementos de uma IOR

Uma referência é basicamente composta por três partes, conforme podemos visualizar na figura 1.3. As duas últimas partes, informações de localização (*endpoint info*) e *object key*, podem aparecer mais de uma vez dentro de uma IOR. Nesta situação, a IOR é classificada como tendo múltiplos perfis. A seguir detalhamos cada parte de uma IOR.

1. Identificador do repositório de interfaces (*repository id*)

   É uma string identificadora de uma interface, possivelmente publicada no repositório de interfaces. Por meio da string e do repositório, é possível obter todas as informações da interface.

2. Informações de localização (*endpoint info*)

   Esta parte contém todas informações necessárias para que o ORB estabeleça uma conexão com o servidor, onde reside o objeto CORBA. No caso do IIOP, temos o endereço IP e a porta TCP/IP.

3. *object key*

   Esta informação é proprietária, isto é, cada ORB utiliza esta parte para colocar informações no formato que deseja. É interessante notar que, apesar de existir esta parte com informações específicas de cada ORB, isto não acarreta nenhum problema de interoperabilidade porque somente o ORB que criou a referência precisa interpretar o conteúdo da *object key* nela contida. Entretanto, todo ORB permite que as

aplicações servidoras especifiquem, no momento da criação de uma referência, um *object id* que será embutido no campo *object key* dessa referência.

Toda requisição CORBA carrega a *object key* do objeto alvo. A cada requisição, o ORB do lado do servidor examina a *object key* e extrai dela o *object id*, permitindo que a aplicação servidora tenha acesso a este id. A possibilidade de se definir um *object id* no momento da criação de uma referência CORBA para um dado objeto e de se obter de volta o mesmo id a cada requisição para esse objeto é um recurso importante e que foi crucial para este trabalho.

1.4.2 Mensagens GIOP

Atualmente existem três versões de GIOP: 1.0, 1.1 e 1.2. Em cada versão existem algumas particularidades que não serão abordadas. O importante para este trabalho será o seu funcionamento básico e algumas de suas mensagens mais importantes.

Dentre as 8 mensagens GIOP existentes, como pode ser visto na tabela 1.1, duas são as mais importantes: "Request" e "Reply", pois implementam a comunicação básica entre o cliente e o servidor. As outras se preocupam com exceções, otimizações e com o gerenciamento da comunicação.

Tipo da mensagem	**Originador**
Request	Cliente
Reply	Servidor
CancelRequest	Cliente
LocateRequest	Cliente
LocateReply	Servidor
CloseConnection	Servidor
MessageError	Cliente ou Servidor
Fragment	Cliente ou Servidor

Tabela 1.1: Mensagens GIOP

A mensagem "Request" tem sempre a origem no cliente e o destino no servidor. Ela é usada para invocar uma operação, cujos parâmetros necessários estão na própria mensagem. O "Reply" é sempre uma mensagem do servidor para o cliente em resposta a uma

mensagem "Request". O conteúdo do "Reply" é o resultado de uma operação, contendo o valor de retorno, os parâmetros de saída e os parâmetros alterados, mas se alguma exceção ocorrer no servidor, a mensagem "Reply" conterá somente as informações desta exceção.

O funcionamento básico de troca destas mensagens é o seguinte: O cliente abre a conexão com o servidor, que por sua vez a aceita. Para invocar uma operação em um objeto CORBA, residente no servidor, o cliente envia a mensagem "Request" na conexão aberta e fica esperando uma resposta. O servidor responde com a mensagem "Reply" e o cliente, recebendo esta mensagem, pode fechar a conexão.

O formato simplificado destas mensagens, em pseudo IDL, é apresentado na figura 1.4.

As estruturas apresentadas na figura 1.4 estão incompletas, foram colocados somente os campos mais importantes. Todas as mensagens GIOP são definidas como uma seqüência de estruturas, sendo algumas delas descritas na figura 1.4. Assim, quando um cliente envia uma mensagem "Request", temos a seguinte seqüência de estruturas presentes na mensagem:

`MessageHeader` Esta estrutura inclui o campo chamado `message_type`, cujo valor é, neste exemplo, `Request`; e outro que contém o tamanho total da mensagem.

`RequestHeader` O primeiro campo é a identificação da requisição, `request_id`, para que o cliente saiba a qual requisição corresponde uma resposta. O segundo campo é o `object_key` que identifica o objeto alvo da requisição. Por último, dentre os campos apresentados da estrutura, temos o nome da operação a ser executada no objeto CORBA, ou simplesmente, operação requisitada.

`Request Body` Contém todos os parâmetros da operação requisitada. O formato desta estrutura não aparece na figura 1.4 porque ele é variável, pois depende da interface do objeto CORBA e da operação requisitada.

Após a mensagem descrita acima, o cliente receberá uma mensagem "Reply" contendo as seguintes estruturas:

```
//------------------------------------
//--- Mensagens GIOP -----------------
//------------------------------------
module GIOP {

  //--- Tipos de mensagens
  enum MsgType {
    Request, Reply, CancelRequest, LocateRequest, LocateReply,
    CloseConnection, MessagemError, Fragment
  }

  //--- Status da resposta
  enum ReplyStatusType {
    NO_EXCEPTION, USER_EXCEPTION, SYSTEM_EXCEPTION, LOCATION_FORWARD
  }

  //--- Header presente em todas as mensagens.
  struct MessageHeader {
        ...
        MsgType message_type;
        octect  message_size;
        ...
  };

  //--- Header presente nas mensagens Request
  struct RequestHeader {
        ...
        unsigned long     request_id;
        sequence<octect>  object_key;
        string            operation;
        ...
  }

  //--- Header presente nas mensagens Reply
  struct ReplyHeader {
        ...
        unsigned long   request_id;
        ReplyStatusType reply_status;
        ...
  }
  ...
}
```

Figura 1.4: Cabeçalhos das mensagens do GIOP.

`MessageHeader` O campo `message_type` terá o valor `Reply`.

`ReplyHeader` O campo `request_id` terá o mesmo valor da mensagem "Request" correspondente. O `reply_status` pode ter três tipos de valores :

1. Não ocorreu nenhuma exceção, `NO_EXCEPTION`, por isso o corpo da resposta, `Reply Body`, conterá o resultado da operação requisitada.
2. Ocorreu alguma exceção, `USER_EXCEPTION` ou `SYSTEM_EXCEPTION`, deste modo o corpo da resposta terá uma descrição da exceção ocorrida.
3. O objeto CORBA não é implementado pelo servidor que recebeu a requisição e uma nova referência para este objeto é enviada no corpo da mensagem. O cliente deve reenviar a requisição para o servidor especificado pela nova referência.

`Reply Body` O formato desta estrutura não aparece na figura 1.4 porque ela é variável. Dependendo do campo `reply_status`, esta parte da mensagem "Reply" conterá os resultados de uma operação bem sucedida, dependentes da interface do objeto CORBA e da operação requisitada, ou as informações sobre uma exceção ocorrida ou, caso o `reply_status` seja `LOCATION_FORWARD`, uma nova referência.

Uma característica do GIOP que é fundamental para este trabalho é o "Reply" com `LOCATION_FORWARD`. Esta mensagem diz para o cliente: *Você enviou a requisição para o local errado, não posso lhe ajudar, mas sugiro a você tentar novamente com esta nova referência.* Será exatamente isso que ocorrerá quando um cliente enviar uma requisição para o servidor e o objeto já não estiver lá, devido a uma migração.

1.5 OA - Object Adapter

1.5.1 Conceitos

O *Object Adapter (OA)*, um conceito independente de CORBA, é um objeto que converte a interface esperada pelos clientes de um dado objeto para a verdadeira interface deste objeto. Em outras palavras, o OA é um objeto intermediário entre um objeto que faz uma requisição e outro que a recebe, tendo como objetivo esconder a verdadeira interface oferecida pelo objeto que recebe a requisição. Um objeto CORBA tem como interface esperada, aquela definida pela IDL e tem como a verdadeira interface, a do objeto que o implementa. CORBA, escondendo a verdadeira interface, ganha independência da linguagem de programação, utilizada para implementar o objeto. Para entendermos os objetivos do OA em CORBA, expostos na próxima seção, apresentamos os seguintes conceitos:

Objeto CORBA Entidade "virtual" capaz de ser localizada por um ORB e ter invocação remota de métodos direcionadas a ela. Um objeto CORBA é identificado, localizado e endereçado por sua referência de objeto.

Servente Entidade da linguagem de programação que existe no contexto de um servidor e implementa um objeto CORBA. Em linguagens não orientadas a objetos, como C e COBOL, um servente é implementado por uma coleção de funções que manipulam dados, que representam o estado de um objeto CORBA. Em linguagens orientadas a objeto, como C++ e Java, um servente é um objeto com os métodos necessários.

Esqueleto Entidade da linguagem de programação na qual o servidor foi implementado. O esqueleto conecta um servente a um OA, permitindo que o OA despache requisições para o servente. Em C, um esqueleto é uma coleção de ponteiros para funções específicas do servente. Em C++ e Java, um esqueleto é uma classe base a partir da qual a classe do servente deriva.

Object id Identificador usado para nomear um objeto no escopo de seu OA. Identificadores de objetos não têm garantia de unicidade global nem são, necessariamente, únicos dentro de um processo servidor. A única restrição é que sejam únicos dentro do OA, onde foram criados ou registrados. Este é o mesmo *object id* que fica embutido no campo *object key* da referência para o objeto (vide seção 1.4.1).

Ativação/ Encarnação Ato de associar um servente a um objeto CORBA pré-existente, através do *object id*, permitindo que o servente receba requisições. É importante notar que uma ativação não implica em criação de objetos CORBA, pois um objeto CORBA não pode ser ativado se não foi criado ainda.

Desativação/ Eterização Ato de dissociar um objeto CORBA de um servente. É o processo contrário da ativação, sendo importante notar que desativar um objeto não implica destruí-lo, pois após a desativação um objeto CORBA deixa de atender requisições, mas pode ser ativado novamente.

Mapa dos objetos ativos É a tabela mantida por um OA, que mapeia objetos CORBA ativos para seus serventes associados. Os objetos CORBA que estão ativos são nomeados na tabela pelo *object id*. Usar esta tabela do OA é a maneira mais fácil de fazer a associação de servente a um objeto CORBA; em outras palavras, ativar um objeto CORBA.

Referência persistente a objeto CORBA É a referência cujo tempo de vida é independente do tempo de vida do servidor que a criou. Sendo assim, o servidor pode terminar e retornar sua execução, mantendo a referência persistente válida.

Referência transiente a objeto CORBA É a referência cujo tempo de vida é dependente do tempo de vida do servidor que a criou, ou seja, o encerramento de uma execução do servidor provoca a invalidez de todas as referências transientes por ele criadas.

1.5.2 Objetivos do OA

Pode-se resumir os objetivos do OA em três ítens :

1. Criar referências de objetos CORBA para permitir que clientes façam requisições a eles;
2. Garantir que cada objeto alvo de uma requisição seja encarnado por um servente;
3. Encaminhar cada requisição que chegar ao ORB (existente no lado servidor) para o servente alvo.

1.5.3 Problemas de Mobilidade e o Object Adapter

Os objetivos do OA e as definições, citadas na seção anterior, começam a levantar uma série de dúvidas e problemas em relação à mobilidade em CORBA. A seguir, apresentaremos as mais importantes:

- O objeto CORBA é *virtual*, pois é o servente que contém código e estado. Assim, deve ficar claro que é o servente o objeto móvel que será migrado. Apesar disto, o *object id* é uma identificação do objeto CORBA e não do servente e, para dificultar a migração, não temos a garantia da unicidade global desta identificação. Isto significa que pode existir duas aplicações servidoras, cada uma com serventes distintos e respondendo requisições de objetos CORBA, também distintos, mas com *object ids* iguais. Caso isto ocorra, quando um servente migrar para a outra aplicação servidora, o Object Adapter não saberá direcionar as requisições para os serventes corretos, pois os dois representam objetos CORBA com o mesmo *object id.*

- Para migrar um servente será necessário desativá-lo em um servidor (aplicação servidora) e ativá-lo em outro. No entanto, entre a desativação e a ativação, o que acontece com as requisições que chegarem?

- Depois da migração do servente, o que acontece com os clientes que possuem uma referência CORBA (transiente ou persistente) utilizada quando o servente não tinha migrado? Esta referência será válida? Se não for válida, o núcleo do ORB que está no cliente, resolve este problema de forma transparente para a aplicação?

- O esqueleto também será migrado? Ele tem estado?

- O Mapa de Objetos Ativos é apenas uma forma de um OA manter a associação entre um servente e um objeto CORBA, mas será necessário manipular este mapa durante a migração do servente, para desativá-lo em um servidor e ativá-lo em outro. Será que o OA oferece uma interface ou uma API para controlar o conteúdo deste mapa?

Retornaremos a esses problemas na seção 5.9 do capítulo 5.

1.6 POA - Portable Object Adapter

O Object Adapter utilizado neste trabalho e o atualmente especificado pelo OMG é o *Portable Object Adapter (POA)*. Ele oferece vários recursos de configuração e modo de funcionamento, trazendo vários benefícios em relação ao *Basic Object Adapter (BOA)*, que é o Object Adapter existente anteriormente. Os objetivos do POA são aqueles especificados para OA, acrescentado-se a portabilidade. Isto significa que, utilizando o POA para escrever um servidor, será possível o seu funcionamento em qualquer ORB, sem nenhuma alteração.

1.6.1 Principais Elementos

Um POA é um elemento intermediário existente entre o ORB e o servente, existindo somente no contexto de um servidor. Cada POA fornece um espaço de nomes para os *object ids* e para outros POAs, chamados POAs filhos. Isto significa que *object ids* com os mesmos valores, e POAs filhos, com nomes iguais, são diferentes se foram criados a partir de POAs distintos. Encadeados, os POAs, formam uma hierarquia de espaços de nomes para objetos CORBA de um servidor.

Os principais elementos do POA, conforme a figura 1.5, são os seguintes:

Root POA É o POA criado pelo ORB, oferecendo à aplicação um POA inicial. O Root POA pode ser suficiente para as necessidades da aplicação ou podem ser criados novos POAs a partir dele. Todas as aplicações pegam uma referência do Root POA com a seguinte instrução:

```
org.omg.PortableServer.POA rootPOA =
 org.omg.PortableServer.POAHelper.narrow(
   orb.resolve_initial_references("RootPOA"));
```

Default Servant É um objeto associado a um POA que, conseqüentemente, direciona todas as suas requisições para este objeto. Este é um recurso interessante, pois podemos ter somente um servente respondendo as requisições para vários objetos CORBA.

Adapter Activator Um *Adapter Activator* é um objeto que o desenvolvedor de aplicação pode associar a um POA, que irá invocá-lo quando tratar de uma requisição para

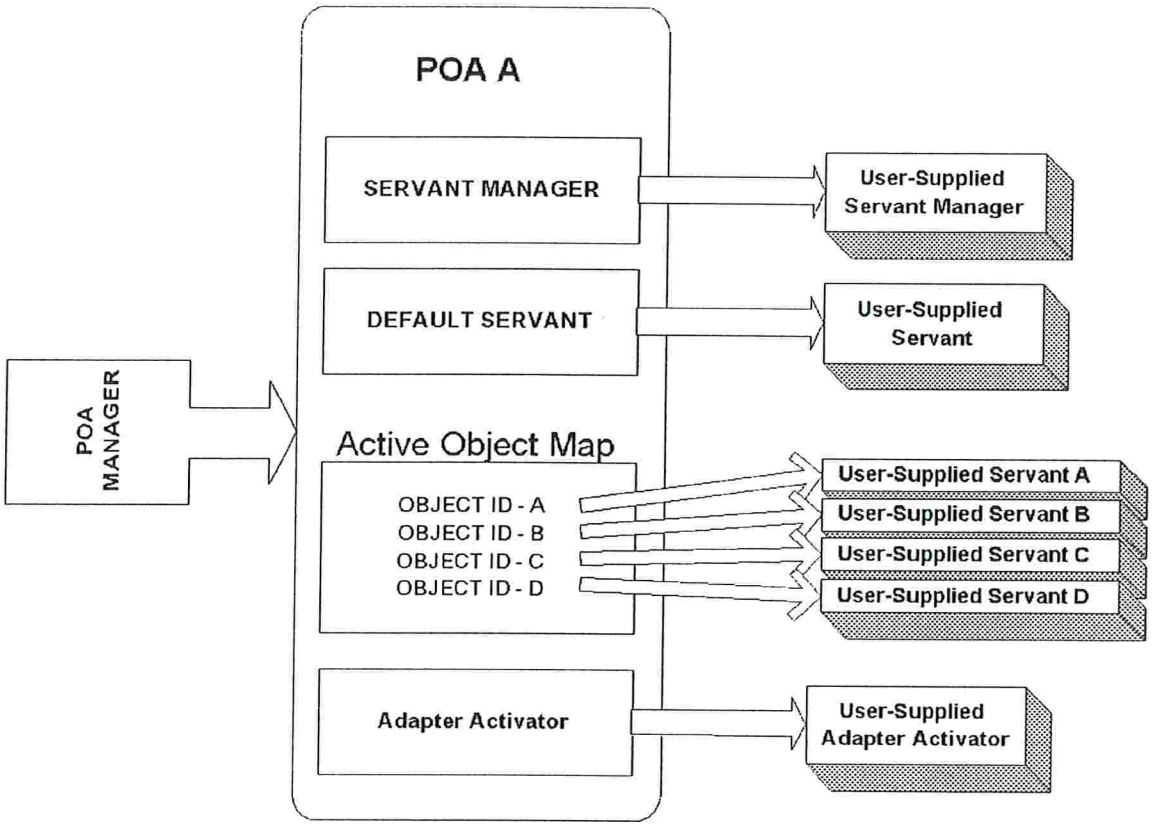

Figura 1.5: Principais elementos do POA

um objeto CORBA que existe em um POA filho, ainda não instanciado. Assim, o *Adapter Activator* pode instanciar o POA filho sob demanda.

Active Object Map Um *Active Object Map*, ou simplesmente, mapa de objetos ativos, armazena o relacionamento de *object id* com serventes.

POA Manager Um *POA Manager* é um objeto que controla o estado de processamento de um ou mais POAs. Usando suas operações, um desenvolvedor pode alterar o fluxo de requisições de um certo POA. Esta alteração pode provocar, por exemplo, um enfileiramento das requisições ou o seu descarte.

Servant Manager O *Servant Manager* é um objeto que pode ser associado a um POA, que irá invocá-lo para ativar e desativar serventes sob demanda. Ele tem a responsabilidade de gerenciar a associação de um objeto a um particular servente, determinando se um objeto existe ou não.

Policy Políticas são objetos que descrevem como um POA deve funcionar. Alguns dos elementos citados acima (Default Servant, Active Object Map, Servant Manager) podem existir ou não em um certo POA, dependendo das políticas especificadas na criação do POA.

Os três últimos elementos do POA (POA Manager, Servant Manager e Policy) são importantes para o entendimento deste trabalho, assim, nas próximas seções estes elementos serão detalhados. Após este detalhamento, serão expostos alguns exemplos para nos familiarizarmos com o POA.

1.6.2 POA Manager

O POA Manager tem quatro estados possíveis, conforme a figura 1.6, que são os seguintes: *active*, *inactive*, *holding* e *discarding*. O estado de processamento determina a situação dos POAs associados a este POA Manager e como são processadas as requisições que chegarem.

Active State Quando o POA Manager se encontra neste estado, todos os POAs associados irão receber requisições.

Discarding State Neste estado todas as requisições serão descartadas e o cliente receberá a exceção *TRANSIENT*, para que seja avisado que a requisição não foi processada.

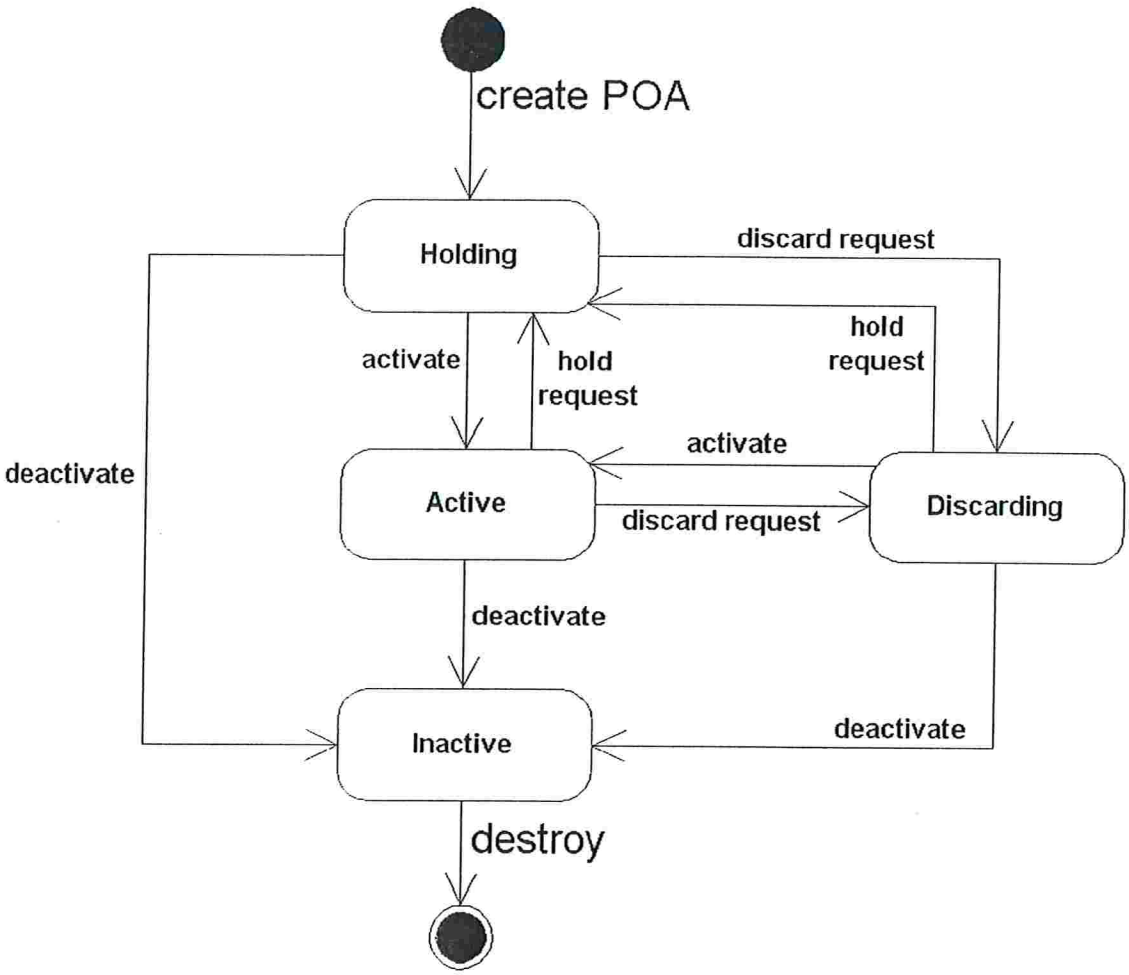

Figura 1.6: Estados do POA

Hoding State Quando se encontra neste estado, o POA Manager irá enfileirar todas as requisições. O limite deste enfileiramento será dependente da implementação do ORB.

Inactive State Este estado tipicamente precede a destruição do POA. Como pode ser visto na figura 1.6, deste estado não há transição possível para outros. Todas as requisições serão rejeitadas nesta situação.

Uma deficiência do POA Manager se manifesta quando um dos seus POAs tem muitos serventes ativos. O estado de "holding" enfileira todas as requisições para todos os serventes. Não é possível enfileirar as requisições somente para um servente. Se fosse possível, seria muito interessante para a implementação da infra-estrutura proposta neste trabalho.

1.6.3 Servant Manager

Um Servant Manager é um objeto registrado no POA como um objeto do tipo *CallBack*, isto é, um objeto que é chamado pelo POA quando necessário.

Para implementar este objeto precisamos conhecer as interfaces envolvidas: `ServantManager`, `ServantActivator` e `ServantLocator`. Na figura 1.7, temos a definição destas interfaces.

A figura 1.7 mostra-nos que a interface ServantManager é vazia. Seu único propósito é ser uma superinterface comum para as interfaces `ServantActivator` e `ServantLocator`. Esta superinterface é o tipo do parâmetro recebido pela operação que associa um objeto `ServantActivator` ou um objeto `ServantLocator` a um POA.

A interface `ServantActivator` não foi utilizada neste trabalho, deste modo não serão detalhados os seus métodos. Uma visão simplificada desta interface nos mostra que seus métodos são usados em conjunto com a tabela de serventes ativos. Quando é necessário inserir um elemento na tabela, é chamado o método `incarnate`, e quando é necessário retirar, é chamado o método `etherealize`.

A interface `ServantLocator`, muito útil para este trabalho, é usada quando o POA não tem uma tabela de serventes ativos. Para cada requisição que chegar a esse POA ocorrerão chamadas a métodos do seu `ServantLocator`. Temos os seguintes passos:

1. Uma requisição chega em um POA com um objeto `ServantLocator`;

```
interface ServantManager{};

interface ServantActivator: ServantManager{
          Servant incarnate(in ObjectId oid,
                            in POA      adapter)
                  raise (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)
                  raise (ForwardRequest);

          void postinvoke(in ObjectId          oid,
                          in POA               adapter,
                          in CORBA:identifier  operation,
                          in Cookie            the_cookie,
                          in Servant           the_servant);
};
```

Figura 1.7: Definição das interfaces ServantManager, ServantActivator e ServantLocator

2. O POA executa o método `preinvoke` do `ServantLocator`;

3. O método executado retorna um servente;

4. O POA direciona a requisição para o servente retornado;

5. A requisição é processada e o servente volta o controle para o POA;

6. O POA executa o método `postinvoke` do `ServantLocator`;

7. O POA envia para o cliente o resultado do processamento da requisição.

Existem dois pontos relevantes nesta interface. O primeiro ponto é a informação dada ao objeto `ServantLocator` de qual operação será ou foi executada no servente. Esta

informação, passada no terceiro parâmetro dos métodos `preinvoke` e `postinvoke`, será útil quando o `ServantLocator` tiver que fazer algo especial para uma determinada operação, como será o caso na implementação da migração.

O segundo ponto importante é que o método `preinvoke` pode lançar a exceção `ForwardRequest`. Lançada esta exceção, o POA encaminhará aos clientes uma resposta com status `LOCATION_FORWARD` e não mais executará o método `postinvoke`. A única maneira de uma aplicação enviar uma resposta `LOCATION_FORWARD` é implementando a interface `ServantActivator` ou a `ServantLocator`.

Em um dos exemplos de utilização do POA, será mostrado como definir um `ServantLocator` para um POA.

1.6.4 Policy

Políticas, como já mencionado, são objetos que definem como será o funcionamento de um POA. No momento da criação do POA é passada uma lista de objetos, políticas previamente criadas e, conseqüentemente, o seu funcionamento está determinado.

Existem 7 tipos de políticas. Seis delas têm 2 valores possíveis e uma tem 3 valores possíveis, resultando em 192 combinações de políticas do POA, como pode ser visto na tabela 1.2. Na realidade, existem combinações de políticas não válidas, diminuindo o número de comportamentos diferentes, mas de qualquer maneira é um número grande de possibilidades. Assim, detalharemos somente quatro políticas: "Assignment policy", "Lifespan policy", "Servant Retention policy" e "Request Processing policy"; e no final será exposta uma combinação de valores destas políticas, interessante para este trabalho.

Id Assignment Policy

O *object id*, como já descrito, está em todas as referências como parte da *object key*, mas quem define o seu valor? Temos duas opções:

`USER_ID` As referências têm o *object id* definido pela aplicação, que o informa no momento da criação da referência.
`SYSTEM_ID` O *object id* é criado e atribuído a uma referência pelo POA, considerado neste contexto como o sistema.

Política	**Valores possíveis**
Thread	`ORB_CTRL_MODEL` (*default*)
	`SINGLE_THREAD_MODEL`
Lifespan	`PERSISTENT`
	`TRANSIENT` (*default*)
Object Id Uniqueness	`UNIQUE_ID` (*default*)
	`MULTIPLE_ID`
Id Assignment	`USER_ID`
	`SYSTEM_ID` (*default*)
Servant Retention	`NON_RETAIN`
	`RETAIN` (*default*)
Request Processing	`USE_SERVANT_MANAGER`
	`USE_ACTIVE_OBJECT_MAP_ONLY` (*default*)
	`USE_DEFAULT_SERVANT`
Implicit Activation	`IMPLICIT_ACTIVATION`
	`NO_IMPLICIT_ACTIVATION` (*default*)

Tabela 1.2: Política existentes

Lifespan Policy

Para se terem as referências transientes e persistentes, será necessário escolher o valor correspondente nesta política.

`TRANSIENT` As referências ficam inválidas quando o processo servidor terminar a sua execução. Se o cliente utilizar uma referência inválida, receberá uma exceção chamada `OBJECT_NOT_EXIST`, caso o servidor, que criou a referência, reinicie a sua execução.
`PERSISTENT` A referência se mantém válida entre execuções do servidor CORBA.

Servant Retention Policy

A utilização do mapa de objetos ativos é controlada por esta política. Temos abaixo as seguintes opções:

`RETAIN` O POA mantém no mapa de objetos ativos os serventes ativos.
`NON_RETAIN` O mapa de objetos ativos não será utilizado.

Request Processing policy

Esta política em combinação com a política anterior, "Servant Retention", cria várias opções para o POA direcionar requisições ao servente correto, provendo uma grande flexibilidade para este mapeamento. Abaixo, temos as opções possíveis desta política.

`USE_ACTIVE_OBJECT_MAP_ONLY` O POA utilizará somente o mapa de objetos ativos para direcionar requisições aos serventes.
`USE_DEFAULT_SERVANT` O POA direcionará todas as requisições para o servente default que deverá processar a requisição.
`USE_SERVANT_MANAGER` O POA, quando chegar uma requisição, chamará o gerenciador de serventes para que ele devolva um servente que será o alvo da requisição.

Uma Combinação das Políticas Citadas

Como já foi mencionado, temos uma grande variedade de combinações de políticas válidas, mas para este trabalho será detalhada somente uma combinação que é exposta na tabela 1.3.

Política	**Valor**
Id Assignment	`USER_ID`
Lifespan	`PERSISTENT`
Servant Retention	`NON_RETAIN`
Request Processing	`USE_SERVANT_MANAGER`

Tabela 1.3: Políticas e valores escolhidos

Um POA criado com estas políticas se comportará da seguinte maneira:

1. O *object id* será definido pela aplicação;
2. As referências criadas, a partir deste POA, serão persistentes;
3. Não será utilizado o mapa de objetos ativos;
4. Será usado o "Servant Manager" que, a cada requisição que chegar neste POA, devolverá um servente válido para essa requisição.

O motivo da escolha desta combinação de políticas será explicitado no capítulo 4. Neste momento, é importante compreender como POA se comportará com estas políticas.

Na próxima seção veremos alguns exemplos de utilização do POA que irão mostrar a criação de um POA com essas políticas.

1.7 Exemplos de Utilização do POA

Para escrever um servidor, utilizando o POA, será necessário escrever muitas ou poucas linhas de código, dependendo do comportamento esperado do servidor. O código mais simples, utilizando todas as características básicas ou implícitas, está apresentado na figura 1.8.

Para implementar a migração de objetos CORBA será necessário explorar alguns outros recursos do POA, não utilizados na figura 1.8, como a *criação de POA*, *ativação de objetos*

```
public class Server
{
    public static void main( String[] args )
    {
      try {

        // --> Obtém referencia do ORB
        org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args, null);

        // --> Obtém referencia do RootPoa
        org.omg.PortableServer.POA rootPOA =
            org.omg.PortableServer.POAHelper.narrow(
                orb.resolve_initial_references("RootPOA"));

        // --> Ativa o RootPOA
        rootPOA.the_POAManager().activate();

        // --> Obtém a referencia do objeto CORBA e ativa o servente.
        org.omg.CORBA.Object o = rootPOA.servant_to_reference(
                                    new <Nome da classe Servente>);

        // --> Escreve a referencia em um arquivo ou publica em um
        //     serviço de nome
        ...

        // --> Ativa o ORB.
        orb.run();

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

Figura 1.8: Exemplo de utilização do POA

CORBA e *criação de referência CORBA*. O restante desta seção descreve estes recursos. Existem outras formas de utilização ou recursos do POA que não serão expostos neste trabalho, por não serem necessários para a migração de objetos.

1.7.1 Criação do POA

Criando um novo POA a partir do *Root POA*, o desenvolvedor da aplicação pode escolher as políticas deste POA e atribuir a ele um *Servant Manager*, um *Adapter Activator* e um *Default Servant*. O código da figura 1.9 exemplifica esta situação.

As duas primeiras políticas da figura 1.9 descrevem como serão as referências criadas a partir daquele POA. As duas últimas descrevem como funcionará a ativação de objetos. O `poaPersistente` será utilizado como exemplo nos próximos ítens, para descrever como serão as referências criadas a partir dele e como será o processo de ativação de desativação de objetos CORBA, tomando como referência o código apresentado na figura 1.9.

1.7.2 Utilização de um Servant Manager

Genericamente, um objeto CORBA é considerado como ativo, caso exista, no mapa de objetos ativos deste POA, uma associação `<object id, Servente>`. Graças a essa associação, todas as requisições para um objeto CORBA ativo serão direcionadas ao servente correspondente. Uma requisição destinada a um objeto não ativo, isto é, um objeto para o qual não exista uma associação no mapa de objetos ativos, faz o POA tentar efetuar a ativação do objeto. Nesta situação, o comportamento do POA é dependente de suas políticas, que determinam uma das seguintes alternativas:

- O POA invoca um *Servant Manager*, que devolve um servente para que o POA encaminhe a requisição. Dependendo das políticas, o *object id* presente na requisição e o servente devolvido no processo, podem entrar ou não no mapa de objetos ativos e, caso entrem, o *Servant Manager* não será mais utilizado para este *object id*;
- O POA tem associado um *Default Servant* para o qual toda requisição é direcionada.

No caso do `poaPersistente`, criado na figura 1.9, o mapa de objetos ativos não é utilizado devido à política `ServantRetentionPolicyValue.NON_RETAIN`. É portanto necessário escolher uma das duas alternativas apresentadas acima. A escolha feita é definida pela política `RequestProcessingPolicyValue.USE_SERVANT_MANAGER`.

```
...
// --> Obtem o Root Poa
org.omg.CORBA.Object obj =
orb.resolve_initial_references("RootPOA");
POA RootPoa = POAHelper.narrow(obj);

// --> Define as políticas do POA que será criado
Policy[] policiesPOA = new Policy[4];

policiesPOA[0] = RootPoa.create_id_assignment_policy(
                        IdAssignmentPolicyValue.USER_ID);

policiesPOA[1] = RootPoa.create_lifespan_policy(
                        LifespanPolicyValue.PERSISTENT);

policiesPOA[2] = RootPoa.create_servant_retention_policy(
                  ServantRetentionPolicyValue.NON_RETAIN);

policiesPOA[3] = RootPoa.create_request_processing_policy(
        RequestProcessingPolicyValue.USE_SERVANT_MANAGER);

// --> Obtem o POA Manager do Root POA para ser
// --> reutilizado no novo POA
POAManager poa_mgr = RootPoa.the_POAManager();

// --> Cria o novo poa com o nome 'Persistent', com o
// --> POA Manager do Root POA e com as políticas definidas
// --> previamente.
POA poaPersistente = RootPoa.create_POA("Persistent",
                                poa_mgr,policiesPOA);
...
```

Figura 1.9: Criação de um POA

Nesta situação, um objeto é considerado ativo, dependendo da resposta do *Servant Manager*. Se ele devolve um servente, o objeto CORBA é considerado ativo, caso contrário, inativo, provocando alguma exceção para o cliente.

Devido a escolha das políticas de ativação de objetos CORBA para o `poaPersistente`, é necessário definir um *Servant Manager* para esse POA. A figura 1.10 mostra como se faz isto.

```
...
// --> Cria um Servant Locator
ServantLocatorImpl sl = new ServantLocatorImpl(...);

// --> Obtem uma referencia do objeto sl
org.omg.CORBA.Object servantLocatorRef =
                      RootPoa.servant_to_reference(sl);

// --> Configura o Servant Manager do poaPersistente
poaPersistente.set_servant_manager(
              ServantLocatorHelper.narrow(servantLocatorRef));
...
```

Figura 1.10: Configuração do Servant Locator para o `poaPersistente`

A classe `ServantLocatorImpl`, da figura 1.10, herda a classe `ServantLocator` que por sua vez, é um `ServantManager`.

O objeto `sl`, criado e atribuído ao `poaPersistente`, irá receber todas as requisições direcionadas a este POA. Normalmente, um objeto deste tipo mantém uma tabela de serventes, contém uma lógica para buscar um elemento desta tabela e retornar uma referência. Deste modo, o objeto `sl` tem todo o controle sobre a ativação de objetos CORBA.

1.7.3 Criação de Referência CORBA

Referências a objetos CORBA são criadas no servidor em um certo POA. Uma vez criada, a referência pode ser exportada, transmitida ou publicada, permitindo a utilização do objeto CORBA pelos seus clientes. O *object id* e o nome do POA são as duas mais importantes informações para a criação de referências. O *object id* pode ser gerado pelo POA ou pela aplicação, é a política `IdAssignmentPolicy` que define como será gerado. Para o `poaPersistente` foram utilizadas duas políticas que determinam o funcionamento das referências. As duas políticas são as seguintes:

- `IdAssignmentPolicyValue.USER_ID`. Esta política determina que, para cada referência criada com esse POA, a aplicação deve fornecer o *object id*.

- `LifespanPolicyValue.PERSISTENT`. Esta política especifica que as referências serão persistentes.

O código da figura 1.11 mostra a criação de uma referência com o `poaPersistente`.

```
...
// --> Define o repository Id
String repositoryId = "IDL:grid:1.0";

// --> Cria um array de byte contendo o object id
byte [] objectId = "Objeto1".getBytes();

// --> Cria a referencia utilizando o POA criado previamente
org.omg.CORBA.Object ref  =
                    poaPersistente.create_reference_with_id(
                              objectId,
                              repositoryId);
...
```

Figura 1.11: Criação de referência por meio do objeto `poaPersistente`

A informação *Repository ID*, necessária para a criação de referência, identifica a interface do objeto CORBA no repositório de interfaces, permitindo a utilização do repositório por clientes baseados na interface de invocação dinâmica.

Na referência `ref`, o *object id* será `"Objeto1"`, que estará em todas as requisições realizadas com esta referência.

Capítulo 2

Java

Java é uma linguagem de programação relativamente nova, porém muitas das suas características não são novas. Os projetistas da linguagem tomaram emprestadas várias idéias de linguagens já existentes, como C++, Smalltalk e Lisp.

A linguagem Java foi projetada para ser robusta e segura. Desta forma ela poderia ser usada para que pequenos "hosted programs", chamados também de "applets", fossem escritos e executados com segurança por "hosting programs", como "web browsers". Por esse motivo, Java é uma linguagem natural para se trabalhar com migração de objetos.

Neste capítulo veremos algumas características da linguagem Java e depois daremos ênfase aos assuntos importantes para a implementação da infra-estrutura de migração.

2.1 Características Gerais

Algumas das características gerais da linguagem Java são as seguintes:

Java é uma linguagem simples. Como sua sintaxe é muito parecida com C++, é fácil para um programador C++ entender programas em Java. Mas a similaridade entre estas duas linguagens fica na sintaxe. Java não tem muitas das características problemáticas existentes em C++. Se examinarmos a linguagem Java, veremos que ela está mais próxima do Smalltalk.

Java é uma linguagem "tipada" estaticamente. Isto significa que o compilador Java pode realizar checagem de tipos e garantir o cumprimento de certas regras de uso.

Mas a linguagem Java também é "tipada" em tempo de execução, pois o sistema de execução de código Java mantém sob controle todos os objetos existentes, associando um tipo a cada objeto. Por exemplo, o "cast" de um certo tipo para outro é verificado em tempo de execução. A verificação de um tipo, em tempo de execução, também torna possível o uso de um objeto carregado dinamicamente, completamente novo para a aplicação, sem prejudicar a segurança de tipos.

Java é uma linguagem que realiza "late-binding" como Smalltalk. A ligação entre a invocação de um método e a sua definição é normalmente feita em tempo de execução. Esta característica é essencial para uma linguagem orientada a objetos, pois uma subclasse pode sobrecarregar métodos da sua superclasse e, apenas em tempo de execução, pode-se determinar qual método deve ser invocado. Se, entretanto, o compilador descobrir que um método não pode ser sobrecarregado por uma subclasse, a ligação da invocação do método a sua definição ocorrerá em tempo de compilação, denominada "early binding". Neste caso haverá um ganho de desempenho na execução do código gerado.

Java fica responsável pelo gerenciamento de memória da aplicação. A linguagem possui alocação dinâmica de objetos e tem a responsabilidade de reciclar o espaço alocado, quando for seguro fazer isto. Esta técnica é denominada coleta de lixo ("garbage collection").

Java permite múltiplos "threads" de execução. Existem mecanismos na linguagem para sincronização, espera explícita e sinalização entre "threads".

Java utiliza herança simples de classes. Além disto, provê uma construção chamada "interface", que especifica o comportamento de um objeto sem definir a sua implementação. Java permite a herança múltipla de interfaces, que provê muitos dos benefícios da herança múltipla de classes, sem os problemas associados a esta característica.

Java facilita a mobilidade de código. Esta é uma de suas principais vantagens para este trabalho. Java tem vários recursos para facilitar a mobilidade, como a variável de ambiente `CLASSPATH`, o "tag" `CODEBASE` em páginas HTML e o objeto `ClassLoader`, sendo o último o recurso mais flexível.

Java possui classes para realizar a reflexão. Isto permite a um programa Java examinar-se e manipular suas propriedades internas. Por exemplo, é possível que uma classe Java obtenha os nomes de todos os seus métodos e mostre estas informações para o usuário.

Java tem o recurso de serializar objetos. Este recurso permite que um programa escreva um objeto com todo o seu estado, em uma seqüência de bytes. Depois, é possível, a partir da seqüência de bytes, recuperar o objeto escrito.

Os recursos de mobilidade de código, realização de reflexão e serialização são os mais importantes para este trabalho. Por isso, cada um deles será detalhado nas próximas seções.

2.2 Mobilidade de Código

Quando não existe mobilidade de código, antes de executar um programa é necessário copiá-lo, muitas vezes manualmente, para uma máquina e, somente lá, localmente, será possível executá-lo. Em Java isto é diferente, pois será necessário existir, em uma certa máquina, o *Java Run-time Environment (JRE)*. Com este ambiente básico, será possível executar programas, carregados automaticamente, que existam em uma máquina remota.

A carga automática de programas, mais especificamente de classes Java, é de responsabilidade do objeto "carregador de classes", que chamaremos de class loader. Dado um nome de uma classe, este objeto é responsável em realizar as seguintes tarefas:

1. Encontrar a seqüência de bytes da classe informada;
2. Ler a seqüência de bytes;
3. Transformar os bytes lidos em uma classe na máquina virtual Java;
4. Devolver a classe obtida no ítem anterior.

Todas as tarefas do *system class loader* ocorrem, implicitamente, na maioria dos programas Java. Vejamos o exemplo da figura 2.1.

No exemplo mostrado na figura 2.1, quando é executado `java Exemplo`, estamos colocando para executar a máquina virtual do Java, passando como parâmetro o nome de uma classe, `"Exemplo"`. Todos esperam que o método `main` seja executado, mas para que isto

```
----------------------------------------
----- Conteúdo do diretório corrente -----
----------------------------------------
./Teste.class
./Exemplo.class

----------------------------------------
----- Conteúdo do CLASSPATH --------------
----------------------------------------
CLASSPATH=.

----------------------------------------
----- Código do Exemplo.java  ------------
----------------------------------------
class Exemplo {

   public static void main( String[] args )

      Teste t1 = new Teste();
      t1.run();
   }
}

----------------------------------------
----- Comando executado no diretório  ----
----  corrente.                         ----
----------------------------------------
java Exemplo
```

Figura 2.1: Exemplo 1 do funcionamento do class loader

ocorra, é necessário que a máquina virtual solicite ao *System Class Loader*, aquele que lê a variável `CLASSPATH`, que carregue a classe `Exemplo`. Vejamos como isto ocorre, revendo as tarefas de um objeto do tipo class loader:

Encontrar a seqüência de bytes da classe informada. O class loader utilizado irá procurar o arquivo `Exemplo.class` nos diretórios especificados na variável `CLASSPATH`. Como o conteúdo desta variável é `''.''` (diretório corrente), temos esta primeira tarefa terminada.

Ler a seqüência de bytes. Nesta tarefa o objeto class loader irá abrir o arquivo `Exemplo.class` e ler todo o seu conteúdo para um vetor de bytes.

Transformar os bytes lidos em uma classe na máquina virtual Java. O class loader transforma um vetor de bytes em um objeto do tipo `Class`.

Devolver a classe obtida no ítem anterior. Devolve à máquina virtual o objeto criado no ítem anterior.

Além das tarefas acima, também foi carregada a classe `Teste`, que é referenciada dentro da classe `Exemplo`, utilizando também o class loader inicial.

Observaremos o exemplo da figura 2.2, onde a carga de uma classe é feita explicitamente e por um class loader diferente.

O class loader `URLClassLoader`, existente no pacote `java.net`, é um class loader especial que não procura as classes utilizando a variável `CLASSPATH`, mas utilizando URLs passadas como parâmetro ao seu construtor. Este class loader já está pronto e foi utilizado no exemplo da figura 2.2. É possível criar novos tipos de class loaders, codificando uma classe que estenda a classe `ClassLoader`. Isto não foi feito porque neste trabalho não houve necessidade, pois a classe `URLClassLoader` satisfez plenamente as exigências da implementação da infra-estrutura de migração.

No exemplo da figura 2.2 deve-se tomar duas precauções. Primeiro, não podemos colocar no código a seguinte instrução: `Teste1 t1 = (Teste1)b.newInstance();` ou algo que referencia a classe `Teste1`. Se isto ocorrer, no momento em que o class loader inicial estiver carregando a classe `Exemplo2`, haverá uma tentativa de carregar a classe `Teste1`, provocando um erro porque o arquivo `Teste1.class` não está no diretório corrente, mas no diretório referenciado pela URL `http://servidor/classes`.

```
----------------------------------------
----- Conteúdo do diretório corrente -----
----------------------------------------
./Exemplo2.class

----------------------------------------
----- Conteúdo do diretório representado -
----- pela URL: http://servidor/classes/ -
----------------------------------------
Teste1.class

----------------------------------------
----- Conteúdo do CLASSPATH --------------
----------------------------------------
CLASSPATH=.

----------------------------------------
----- Código do Exemplo2.java  -----------
----------------------------------------
import java.net.*;

class Exemplo2 {

   public static void main( String[] args ) {

    try {
      URL url1 = new URL("http://servidor/classes/");
      URL [] urls = {url1};
      ClassLoader urlCL = new URLClassLoader(urls);
      Class  c = urlCL.loadClass("Teste1");
      Object obj = c.newInstance();
    }
    catch(Exception e){
        System.out.println(e.toString());
    }
   }
}
```

Figura 2.2: Exemplo 2 do funcionamento do class loader

A segunda precaução é que não poderá existir o arquivo `Teste1.class` no diretório corrente, senão este será utilizado, mesmo que exista outro arquivo com o mesmo nome no servidor HTTP. A instrução `loadClass("Teste1")` irá primeiramente passar esta requisição para o objeto class loader pai, que é o system class loader, e depois, não tendo encontrado a classe, irá procurá-la nas URLs informadas.

Devido ao primeiro problema, não podemos simplesmente executar o método `run` existente na classe `Teste1`, porque o método `newInstance` retorna o objeto `obj` da classe `Object`, que não tem o método `run`. Nesta situação, só será possível executar o método se conhecermos reflexão, que é o assunto da próxima seção.

2.3 Reflexão

A reflexão possibilita que o código Java examine e trabalhe com classes e objetos Java que não são conhecidos durante a programação, mas somente em tempo de execução. Para começar esclarecer esta definição vamos examinar o código apresentado na figura 2.3.

No código apresentado na figura 2.3 foram utilizados para a reflexão, somente as classes `Class` e `Method`, mas existem várias outras, como por exemplo: `Field` para acessar campos de uma classe e `Constructor` para acessar os construtores.

A classe `Class` é onde começa a reflexão, sem um objeto desta classe não é possível utilizar as classes do pacote `java.lang.reflect`, onde estão as outras classes citadas. Vejamos abaixo dois métodos importantes desta classe:

`Method getDeclaredMethod(String name, Class[] parameterTypes)`- Retorna o objeto do tipo `Method` que reflete o método especificado nos parâmetros. Se a classe não tem o método especificado é gerada a exceção: `NoSuchMethodException`.

`Method[] getDeclaredMethods()` - Retorna um array com todos os métodos da classe.

Esses métodos retornam objetos do tipo `Method`. Alguns métodos importantes deste objeto são descritos abaixo:

`String toString()`- Retorna uma string, descrevendo o objeto alvo da chamada.

`Object invoke(Object obj, Object[] args)`- Executa o método no objeto `obj` com os parâmetros `args`. O método acionado é aquele representado pela instância da classe `Method` que é alvo da chamada `invoke`.

```
----------------------------------------
----- Código do Exemplo3.java  ----------
----------------------------------------
import java.net.*;
import java.lang.reflect.*;

class Exemplo3 {

   public static void main( String[] args ) {

    try {

      //-- Carrega a classe Teste1 ----------------
      URL url1 = new URL("http://servidor/classes/");
      URL [] urls = {url1};
      ClassLoader urlCL = new URLClassLoader(urls);
      Class c = urlCL.loadClass("Teste1");

      //-- Examina a classe carregada -------------
      Method m[] = c.getDeclaredMethods();
      for (int i =0; i < m.length; i++)
          System.out.println(m[i].toString());

      //-- Trabalha com a classe Teste1 -----------
      Method run = c.getDeclaredMethod("run",null);
      Object obj = c.newInstance();
      Object returned = run.invoke(obj,null);

    }
    catch(Exception e){
        System.out.println(e.toString());
    }
   }
}
```

Figura 2.3: Exemplo de reflexão em Java

A classe `Teste1` tem o código apresentado na figura 2.4.

```
public class Teste1 {

    public Teste1(){
     System.out.println("criado um objeto Teste1");
    }

    public void run() {
     System.out.println("executou run");
    }
}
```

Figura 2.4: Código da classe "Teste1"

A execução do `Exemplo3`, exposto na figura 2.3, devido ao código da classe `Teste1`, gerará o resultado da figura 2.5.

```
public void Teste1.run()
criado um objeto Teste1
executou run
```

Figura 2.5: Resultado de execução do "Exemplo3"

Na seção anterior vimos como realizar a carga de uma classe dinamicamente. O exemplo apresentado nesta seção expande o da seção anterior, mostrando como se pode usar reflexão para chamar métodos de classes carregadas dinamicamente.

2.4 Serialização

A serialização permite que um programa escreva um objeto, com todo o seu estado, em uma seqüência de bytes. Depois é possível, a partir dessa seqüência, recuperar o objeto escrito.

A seqüência de bytes, que representa um objeto Java serializado, pode ser armazenada no sistema de arquivos ou em um banco de dados. Este recurso é útil para todas as aplicações cujos objetos devem manter estado entre uma execução e outra.

Neste trabalho, usaremos a serialização para deslocar um objeto de uma aplicação para outra. Será serializado um objeto, resultando em uma seqüência de bytes, que será depois

transferida para uma outra aplicação. Chegando lá, esta seqüência de bytes será desserializada, recuperando-se o objeto com o mesmo estado que tinha antes da serialização.

Um objeto Java, para suportar esse recurso, deve implementar a interface `Serializable` ou a `Externalizable`. A interface `Externalizable`, raramente utilizada, contém dois métodos: `readExternal` e `writeExternal`, que devem ser implementados para personalizar o mecanismo de serialização. A interface `Serializable`, que é uma interface vazia, na maioria dos casos é, suficiente para se utilizar o recurso de serialização.

```
-------------------------------------------
-- Escrevendo objetos Java em um arquivo.  --
-------------------------------------------
import java.io.*;
import java.util.*;

public class Serializacao {

   public static void main(String[] args) {
    try {

     FileOutputStream arquivo = new FileOutputStream("/tmp/tmp.txt");
     ObjectOutputStream saida = new ObjectOutputStream(arquivo);
     saida.writeObject("Today");
     saida.writeObject(new Date());
     saida.flush();

    } catch (Exception e) {
       System.out.println(e.toString());
    }
   }
}
```

Figura 2.6: Exemplo de serialização

Para mostrar como funciona a serialização, consideremos o exemplo da figura 2.6. Neste exemplo, o principal objeto é o objeto `saida`, que é da classe `ObjectOutputStream`, que contém métodos para escrever objetos Java no arquivo informado como parâmetro do seu construtor.

Após a execução da classe `Serializacao`, temos um arquivo chamado `tmp.txt` que contém dois objetos, o primeiro do tipo `String`, cujo valor é `"Today"`, e o segundo do tipo `Date`, cujo valor é a data corrente.

```
-------------------------------------------
-- Lendo objetos Java de um arquivo.      --
-------------------------------------------
public class Desserializacao {

 public static void main(String[] args) {
   try {

    //-- Abre o arquivo para desserializar os objetos
    FileInputStream arquivo = new FileInputStream("/tmp/tmp.txt");
    ObjectInputStream entrada = new ObjectInputStream(arquivo);

    //-- Desserializa os objetos do arquivo aberto
    String today = (String) entrada.readObject();
    Date date = (Date) entrada.readObject();

    //-- Imprime os objetos desserializado
    System.out.println(today);
    System.out.println(date.toString());

   } catch (Exception e) {
      System.out.println(e.toString());
   }
  }
}
```

Figura 2.7: Exemplo de desserialização

O arquivo gerado, `tmp.txt`, será o arquivo utilizado no exemplo da figura 2.7 para demonstrar a desserialização. Neste exemplo, o principal objeto é o objeto `entrada`, que é da classe `ObjectInputStream`, que contém métodos para ler objetos Java do arquivo informado como parâmetro do construtor.

Após a execução da classe `Desserializacao`, temos impresso a string `"Today"` e o valor do objeto da classe `Today`, no momento em que foi realizada a serialização.

Na implementação da infra-estrutura, não foram utilizados arquivos para serializar objetos, mas arrays de bytes que ficam em memória. Para se trabalhar com um array de bytes é necessário:

- Fornecer um objeto da classe `ByteArrayOutputStream` ao construtor da classe `ObjectOutputStream`;
- Fornecer um objeto da classe `ByteArrayInputStream` ao construtor da classe `ObjectInputStream`.

Capítulo 3

Trabalhos Relacionados

A área de mobilidade de objetos ou agentes tem sido foco de intensa atividade de pesquisa. Os trabalhos nesta área são geralmente identificados como *Sistemas de Agentes Móveis*, pois em muitos casos fornecem uma implementação como proposta de infra-estrutura de mobilidade de agentes, objetos ou código. Dentre outros podemos citar: Jumping Beans [22], Aglets [10], Voyager [5], Telescript [26], Agent TCL [9], Concordia [25], Mole [1], Ara [20], MOA [12] e MASIF [11].

Este capítulo aborda os sistemas *Jumping Beans*, *Voyager* e *Aglets*. Escolhemos os dois primeiros porque usam CORBA e o último por ser em Java e bastante conhecido. Ao final do capítulo, apresentaremos a especificação do OMG chamada Life Cycle. Essa especificação aborda vários assuntos, incluindo uma proposta de mobilidade de objetos CORBA.

3.1 Jumping Beans

A empresa *Ad Astra Engineering, Inc.*, fundada em 1993, desenvolveu o sistema Jumping Beans [22] em 1998, tendo como foco a mobilidade.

3.1.1 Descrição

Segundo a documentação deste sistema, "Jumping Beans é um framework Java para construir aplicações móveis. Uma aplicação móvel começa a sua execução em uma máquina, podendo se mover para uma outra máquina, mesmo que nunca tenha sido instalada na

mesma. Jumping Beans oferece aos desenvolvedores um modo fácil de criar aplicações móveis que serão seguras e robustas."

Este framework é composto, basicamente, de uma API e de um servidor que centraliza todos os deslocamentos da aplicação móvel. Chamaremos esse servidor de Servidor Jumping Beans. A API contém dois elementos principais, sendo um deles uma classe chamada `Agency`, que cria um contexto de execução para as aplicações móveis. O outro elemento é uma interface chamada `MobileApp`, que deve ser implementada por uma classe Java, cujas instâncias serão aplicações móveis.

Assim, é possível ter um entendimento básico do que é o Jumping Beans, entendendo as funções de três entidades principais : o Servidor Jumping Beans, a Agência e uma aplicação móvel. Uma aplicação móvel simples roda num contexto de execução fornecido por alguma agência e deve simplesmente implementar a interface `MobileApp`. Uma agência, para criar um contexto de execução, pede várias informações da aplicação móvel. Dentre estas, as mais importantes são as seguintes: o nome da aplicação móvel, o nome da classe e uma lista de parâmetros. Após o contexto de execução ter sido criado, a aplicação móvel já estará em execução e poderá se deslocar para uma outra agência, passando pelo Servidor Jumping Beans.

O contexto de execução é um objeto importante para o funcionamento do Jumping Beans, pois é ele que tem o método `DispatchTo`, cuja definição temos na figura 3.3. Assim, qualquer objeto que consegue obter o contexto de execução de uma aplicação móvel, pode solicitar a ele o deslocamento da mesma para uma outra agência. Deste modo, a própria aplicação móvel pode também solicitar o seu deslocamento.

Não é possível saber exatamente como é o protocolo entre as agências e o servidor Jumping Beans, pois o seu código fonte não está disponível. Podemos somente ter uma visão de usuário desta infra-estrutura, ou seja, o que o Jumping Beans nos oferece para que possamos fazer uma aplicação móvel.

3.1.2 Utilização

O sistema oferece o Servidor Jumping Beans, um produto fechado e pronto para ser utilizado, e mais uma biblioteca de classes. O usuário fica responsável pela implementação das Agências (utilizando para tanto a classe `Agency` fornecida com o sistema) e das aplicações móveis propriamente ditas.

Uma agência pode ser criada com o código da figura 3.1, que mostra somente os parâmetros mais importantes. Após a execução desse código, temos uma agência criada recebendo requisições na porta 8000 e registrada no Servidor Jumping Beans "Pizza" com o nome de "Pan".

```
agency = new com.JumpingBeans.core.api.Agency(
             "Pan",      // Nome da agencia
              ...
              8000,        // Porta da agência
              "Pizza",   // Server name
              8001,        // Porta do Servidor
              ...):
```

Figura 3.1: Criação de uma agência

Alguns dos métodos importantes de um objeto do tipo Agency são os seguintes:

```
MobileAppContext      createMobileApp(String mobileAppName,}
                                      String className,
                                      Object[] args,...)

MobileAppID[]         listAllMobileApps()

AgencyID[]            listAllAgenciesInDomain(boolean listAvailable)

AgencyID              getAgencyID (java.lang.String agencyName)
```

Figura 3.2: Alguns métodos da classe Agency

Com uma agência é possível criar uma aplicação móvel efetuando uma chamada ao método createMobileApp que devolve um objeto do tipo MobileAppContext, cujos métodos mais importantes aparecem na figura 3.3.

```
void dispatchTo(AgencyID destinationAgencyID)
void destroy()
void deactivate(int duration)
```

Figura 3.3: Alguns métodos da classe MobileAppContext

O método dispatchTo espera um objeto do tipo AgencyID que identifica unicamente

uma agência. Este objeto pode ser obtido, por exemplo, através do método `getAgencyID`, citado na figura 3.2.

O método `destroy` finaliza a execução da aplicação móvel e por último, o `deactivate` serializa a aplicação móvel, armazenando-a de forma persistente pelo tempo especificado.

A utilização do Jumping Beans envolve os seguintes passos:

- Criar uma agência (`new Agency`);
- Criar uma aplicação móvel (`createMobileApps`);
- Descobrir a identificação da agência destino (`getAgencyID`);
- Solicitar a migração ao contexto da aplicação (`dispatchTo`).

3.2 Aglets

A IBM desenvolveu o sistema Aglets [10] e disponibilizou o "Aglets Software Development Kit" (ASDK), cuja versão utilizada neste trabalho foi a "1.1b2". Além disso, publica uma especificação do sistema Aglets e disponibiliza um "Agent Server" chamado Tahiti. O kit e a especificação detalham o funcionamento dos Aglets e contêm exemplos de como construí-los e utilizá-los.

3.2.1 Descrição

Segundo a documentação do sistema, "Aglets são objetos Java que podem se mover de uma máquina para outra na internet. Um Aglet que é executado em uma máquina pode paralisar sua execução, transferir-se para uma máquina remota e lá continuar sua execução. Quando um aglet se move, ele carrega junto seu código e também seu estado."

Para uma aplicação ser um aglet, basta que estenda a classe `com.ibm.Aglet`. Para exemplificar, a classe apresentada na figura 3.4 pode ser considerada um aglet.

A classe `Aglet` tem muitos métodos e os mais importantes podem ser vistos na figura 3.5.

O método `clone` cria uma cópia do aglet alvo desta requisição; o `deactivate` serializa e armazena-o por um determinado tempo. O método `dispatch` envia o aglet para o contexto identificado pelo objeto do tipo `Ticket` e o método `dispose` o destrói.

```
package trial;
import com.ibm.aglet.*;

public class MyAglet extends Aglet {
 public void run() {
  System.out.println("Hello, world!");
 }
}
```

Figura 3.4: Exemplo de Aglet

```
public final java.lang.Object clone()

public final void deactivate(long duration)

public final void dispatch(Ticket ticket)

public final void dispose()

public void run()
```

Figura 3.5: Alguns métodos da class `Aglet`

Dentre os métodos da figura 3.5, o único que o aglet pode e deve redefinir é o método `run`, pois os restantes são métodos finais. O método `run` é o ponto de entrada na execução de um aglet.

Para criar um aglet, precisamos de um contexto de execução. Alguns de seus métodos estão na figura 3.6.

```
void start()

AgletProxy createAglet(java.net.URL codeBase, java.lang.String code,..)

AgletProxy retractAglet(java.net.URL url, AgletID aid)

void shutdown()
```

Figura 3.6: Alguns métodos da classe `AgletContext`

O método `start` inicia um contexto e método `shutdown` o finaliza. Cada chamada a `createAglet` cria um aglet no contexto, assim, podemos ter vários aglets em um único contexto. O método `retractAglet` é um método curioso, porque traz de volta um aglet que foi enviado para um outro contexto de execução.

Na figura 3.7 temos algumas operações descritas na figura 3.5 e na figura 3.6. A operação `deactivate` é aquela que coloca um aglet em uma mídia persistente, que na figura, está determinada como "secondary storage".

3.2.2 Utilização

Esta subseção apresenta um dos vários exemplos fornecidos com o ASDK. O objetivo deste exemplo é vigiar um certo arquivo e quando este for alterado, emitir uma notificação. O programa consiste em dois aglets. Um é estacionário, chamado `Watcher`, o outro é móvel, chamado `WatcherNotifier`. Esse último é enviado pelo primeiro, para uma localidade remota, com um nome de arquivo. Quando o arquivo é alterado, o `WatcherNotifier` envia uma mensagem para o aglet `Watcher` que mostra uma notificação ao usuário na sua tela principal. A notificação é composta da palavra "UPDATE", mais o nome do arquivo, data e o horário da alteração. O aglet `WatcherNotifier` fica no Servidor Aglet remoto por um determinado tempo e depois finaliza sua execução. Na figura 3.8 podemos ver o aglet `Watcher` executando e também algumas notificações.

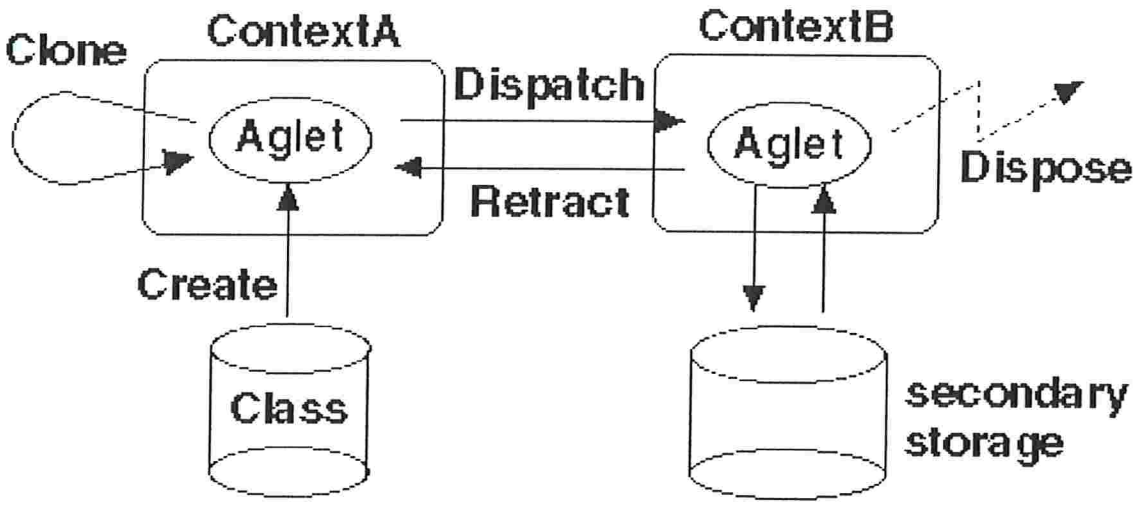

Figura 3.7: Operações possíveis em um aglet

Algumas partes do código do aglet `Watcher` serão apresentadas para esclarecer alguns pontos. Na figura 3.9, podemos ver como é criado o aglet `WatcherNotifier`.

No pacote Java `com.ibm.agletx.patterns` existem várias classes que modelam padrões de funcionamento de aglets. Além do padrão utilizado neste modelo, o `Notifier`, existem outros padrões, como o `Meeting` e o `Messenger`. O padrão `Notifier` obriga o aglet a implementar o método `doCheck` que será executado com uma freqüência determinada pelo parâmetro `interval`. É neste método que o aglet fica vigiando qualquer alteração do arquivo passado como parâmetro, enviando uma mensagem para o aglet `Watcher` a cada alteração.

Aglets comunicam-se trocando objetos do tipo mensagem (`Message`), composto de um objeto do tipo `String`, que especifica o tipo da mensagem, e de um outro objeto arbitrário. Quando um aglet envia uma mensagem para um outro, esta chega ao destino através do método `handleMessage`. Na figura 3.10 temos o código de recebimento de mensagem do aglet `Watcher`, para tratar as notificações enviadas pelo aglet `WatcherNotifier`.

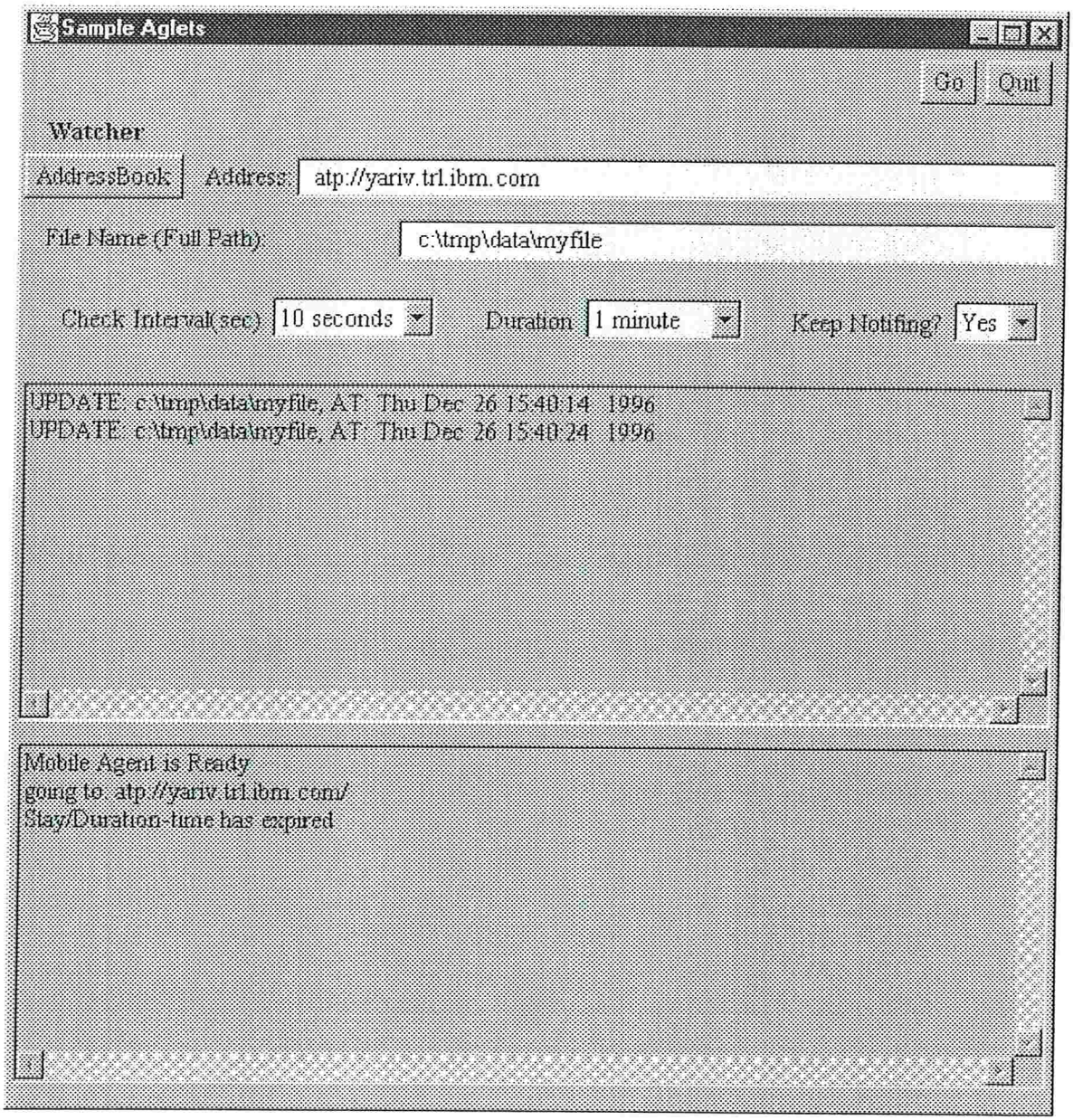

Figura 3.8: Execução do aglet "Watcher"

```
//----------------------------
//--- Método da classe Watcher
//----------------------------
protected void createSlave(Vector destinations, Object obj) {
  try {

  Notifier.create(null, // url da classe, para o caso
                        // de carga remota.
      "examples.patterns.WatcherNotifier", // nome da classe
      getAgletContext(),        // contexto onde será criado
      this,                     // aglet master
      "atp://localhost:9000",   // destino
      interval,                 // intervalo entre as checagens
      duration,                 // tempo de vida deste aglet
      stay,                     // se o aglet será destruído após
                                // a notificação
      path);                    // parâmetro do aglet

  } catch (Exception e) {
    ....
  }

}
```

Figura 3.9: Classe `Watcher` parte 1

```
//----------------------------
//--- Método da classe Watcher
//----------------------------
public boolean handleMessage(Message msg) {

  if (msg.sameKind("notification"))
    //--- Mostra a mensagem na janela principal
    message((Arguments)(msg.getArg()));
  else
    super.handleMessage(msg);

  return true;
}
```

Figura 3.10: Classe `Watcher` parte 2

3.3 Voyager

O Voyager [5], comercializado pela ObjectSpace, é um ORB que se diferencia dos demais por oferecer funcionalidades que facilitam a mobilidade de objetos e a criação de agentes móveis.

3.3.1 Descrição

O sistema Voyager é composto de um servidor, utilitários e um conjunto de pacotes Java. A quantidade de funcionalidades é muito grande, inviabilizando uma explicação detalhada de todas elas nesta seção. Iremos nos limitar às funcionalidades relativas a mobilidade, mas antes abordaremos alguns assuntos básicos sobre Voyager.

As aplicações desenvolvidas com o Voyager devem executar o comando `Voyager.startup` no início e `Voyager.shutdown` no final do seu código. O método `startup` pode receber um parâmetro que é uma URL. Caso ele seja chamado com esse parâmetro, a aplicação se torna um servidor que pode receber requisições a objetos. Caso contrário, a aplicação é uma aplicação cliente que não pode receber requisições, mas enviá-las apenas para um servidor.

3.3.2 Utilização

Este é o mais simples dos exemplos fornecidos com o ORB Voyager 3.2. É dada uma classe Java `Stockmarket`, que implementa a interface `IStockmarket` da figura 3.11. A figura 3.12 mostra a aplicação `Basics`, que utiliza um objeto da classe `Stockmarket`.

```
package examples.stockmarket;
public interface IStockmarket
 {
 int quote( String symbol );
 int buy( int shares, String symbol );
 int sell( int shares, String symbol );
 void news( String announcement );
 }
```

Figura 3.11: Interface IStockmarket

```
package examples.basics;
import com.objectspace.voyager.*;

class Basics {

  public static void main( String[] args ) {
   ...
   //--- Inicia a aplicação como uma aplicação cliente
   Voyager.startup();

   //--- Cria um objeto Stockmarket remoto, isto é, no
   //--  servidor Voyager que está executando na máquina
   //--  localhost e escutando na porta 8000
   final String classname = "examples.stockmarket.Stockmarket";

   IStockmarket market = (IStockmarket)
            Factory.create( classname, "//localhost:8000");

   //--- Aciona o método news no objeto remoto.
   market.news("Sun releases Java");

   //--- Aciona o método quote no objeto remoto
   System.out.println("sun share price =" +
                  market.quote("SUN") );
   ...
   //--- Termina a aplicação cliente.
   Voyager.shutdown();
  }
}
```

Figura 3.12: A aplicação `Basics` Voyager

Antes de executar a aplicação `Basics` é necessário rodar o servidor `voyager`, com o seguinte comando:

```
> voyager 8000
```

A execução da aplicação `Basics` produz a seguinte saída:

```
> java examples.basics.Basics
> sun share price = 103
```

Apesar da simplicidade deste exemplo, é possível perceber algumas características do Voyager, como o método estático `create` da classe `Factory`. Este método recebe um ou dois parâmetros. Se receber um parâmetro, o objeto é criado localmente. Caso contrário, o segundo parâmetro é a localização onde será criado o objeto, que pode ser o servidor `voyager` ou uma outra aplicação servidora. Neste caso, o objeto retornado pelo método `create` é, na realidade, um proxy para o objeto remoto.

3.3.3 Movendo um Objeto

Para mover um objeto deve-se, em primeiro lugar, passá-lo como parâmetro numa chamada ao método `Mobility.of(Object obj)`. Este objeto tem que ser serializável. O método estático `of` da classe `Mobility` retorna um objeto do tipo `IMobility`, que tem os seguintes métodos :

`moveTo(String url)` Move o objeto para a url especificada;

`moveTo (Object obj)` Move o objeto para o programa que contém o objeto especificado.

Por exemplo, o código da figura 3.13 cria um objeto do tipo `StockMarket` em `"//dallas:8000"` e depois o move para `"//tokyo:9000"`.

O método `moveTo()` causa uma série de eventos que iremos detalhar, porque tem muitos passos em comum com a infra-estrutura que iremos apresentar no capítulo 4. Os eventos são os seguintes:

1. As mensagens para o objeto que irá deslocar são processadas. Qualquer outra mensagem que chegar para o objeto após o método `moveTo()` são enfileiradas;

```
//--- Cria o objeto market em
//--- dallas:8000
IStockMarket market = (IStockMarket)
                  Factory.create("StockMarket",
                                 "//dallas:8000");

//--- Envia uma requisição para o local inicial
market.news("at first location");

//--- Obtem a faceta IMobility do objeto market
IMobility mobility = Mobility.of( market );

//--- Move o objeto para um novo local
mobility.moveTo("//tokyo:9000");

//--- Envia uma segunda requisição para o novo local
market.news("at second location");
```

Figura 3.13: Exemplo 2 Voyager

2. O objeto é copiado para o novo local, usando serialização Java. Uma exceção é disparada quando qualquer parte do objeto não é serializada ou quando um erro da rede ocorre;

3. O novo endereço do objeto e todas suas partes não transientes ficam em cache na antiga localidade;

4. O objeto antigo é destruído;

5. As mensagens enfileiradas para o objeto movido e todas as outras mensagens que chegaram ao local antigo do objeto são tratadas da seguinte maneira:

   (a) Uma exceção especial é gerada. Esta exceção contém o novo local do objeto.

   (b) A exceção é tratada pelo proxy. O proxy reconecta-se com o novo endereço e reenvia a mensagem;

6. O método `moveTo()` retorna à aplicação depois do objeto mover-se com sucesso, caso contrário é gerada uma exceção.

3.4 Life Cycle

A especificação Life Cycle [8] diz como os objetos podem ser criados, destruídos, copiados ou movidos. Apresentaremos nas subseções seguintes o que ela propõe para cada uma destas operações.

A especificação definiu basicamente três interfaces, denominadas `FactoryFinder`, `LifeCycleObject` e `GenericFactory`, que são apresentadas na figura 3.14. Ela recomenda que os métodos de gerenciamento de ciclo de vida sejam oferecidos pelos próprios objetos de aplicação, os quais devem implementar a interface `LifeCycleObject`. Para tais objetos, a interface `LifeCycleObject` é geralmente herdada pela "interface principal" do objeto.

Ao final da seção, abordaremos as críticas ou dúvidas referentes à especificação Life Cycle que encontramos na literatura [8].

3.4.1 Criação de Objetos

A especificação Life Cycle recomenda que as aplicações CORBA usem o padrão fábrica ("Factory") para criar objetos. Uma fábrica é um objeto CORBA que oferece uma ou mais operações para criar outros objetos.

Para criar um novo objeto, o cliente invoca uma operação na fábrica. A implementação da operação cria um novo objeto CORBA e retorna uma referência ao novo objeto para o cliente.

A criação de objetos é muito dependente ao tipo do objeto que está sendo criado. Deste modo, o cliente passará para a fábrica, dependendo do tipo de objeto, um conjunto diferente de parâmetros. A especificação Life Cycle definiu a interface de uma fábrica genérica para atender esta necessidade. Na figura 3.14, podemos ver a interface `GenericFactory` proposta.

Na interface proposta temos dois métodos. O primeiro, `supports`, recebe uma chave e retorna um `boolean` que informa se é possível criar um objeto do tipo representado pela chave. O segundo, `create_object`, recebe dois parâmetros. Um deles identifica o objeto a ser criado e o outro é uma seqüência de pares com nome e valor. O segundo parâmetro serve para comunicar à fábrica todas as informações necessárias para a criação do objeto.

```
module CosLifeCycle
{
        typedef CosNaming::Name Key;
        typedef Object _Factory;
        typedef sequence <_Factory> Factories;
        typedef struct NVP {
                CosNaming::Istring  name;
                any     value;
        } NameValuePair;

        typedef sequence <NameValuePair> Criteria;
        ...

        interface FactoryFinder {
              Factories find_factories(in Key factory_key)
                        raises(NoFactory);
        };

        interface LifeCycleObject {
             LifeCycleObject copy(in FactoryFinder there,
                                  in Criteria      the_criteria)
                             raises(NoFactory, NotCopyable,
                                    InvalidCriteria,CannotMeetCriteria);

             void move(in FactoryFinder there,
                       in Criteria      the_criteria)
                  raises(NoFactory, NotMovable,
                         InvalidCriteria, CannotMeetCriteria);

             void remove()
                  raises(NotRemovable);
        };

        interface GenericFactory {
             boolean supports(in Key k);

             Object  create_object(in Key      k,
                                   in Criteria the_criteria)
                     raises (NoFactory, InvalidCriteria,
                             CannotMeetCriteria);
        };
};
```

Figura 3.14: Interfaces do Life Cycle

3.4.2 Destruição de Objetos

Para destruir um objeto, o cliente invoca a operação `remove` no objeto. Depois desta operação, qualquer cliente que for usar o objeto destruído receberá a exceção `OBJECT_NOT_EXIST`. Deste modo, o que é destruído é o objeto CORBA e não somente o servente deste objeto.

A implementação desta operação no servente dependerá das políticas escolhidas para o POA. O caso mais simples ocorre quando o POA usa o Servant Locator, que mantém uma tabela de serventes ativos. Neste caso, a implementação do `remove` deverá apenas retirar uma entrada dessa tabela. Quando o POA não utilizar o Servant Locator, a implementação do objeto irá interagir com o POA para se desativar.

Observando as interfaces do Life Cycle, pode-se pensar que a operação `remove` deveria estar na interface `GenericFactory`, pois a fábrica que criou um objeto deveria saber destruí-lo. Uma análise mais aprofundada mostra que colocar o método `remove` na fábrica não é uma boa escolha. O problema reside na necessidade de se saber qual foi a fábrica que criou um certo objeto.

3.4.3 Cópia de Objetos

Conceitualmente, a operação `copy` é muito similar a operação `create_object` da fábrica, pois ambas as operações criam um novo objeto. A diferença é que na operação `copy` o estado inicial do objeto não é passado como parâmetro, mas obtido do objeto alvo da operação.

A implementação do `copy` irá receber como parâmetro um objeto do tipo `FactoryFinder`. Com esse objeto, a implementação poderá invocar a operação `find_factories` e receber uma lista de fábricas. Desta lista, a implementação irá escolher uma fábrica e invocará nela a operação `create_object`. Após estes principais procedimentos para a implementação da operação `copy` terminará a sua execução, retornando uma referência do objeto novo.

3.4.4 Migração de Objetos

A operação `move` tem a sua assinatura muito similar a operação `copy`. A única diferença é que o retorno do método `copy` é a referência do objeto `LifeCycleObject` e o retorno do

`move` é `void`.

O objetivo da operação `move` é mover um objeto de uma localização para outra, sem invalidar as referências do objeto movido. O ato de um objeto se mover é também chamado de migração.

A implementação, possivelmente, irá invocar o método `create_object` no novo local e depois, invocar o seu próprio método `remove`, acrescentando o cuidado para preservar a suas referências.

3.4.5 Críticas ao Life Cycle

A especificação Life Cycle foi criticada em vários de seus aspectos, como os tipos escolhidos nas interfaces, a quantidade de métodos de cada interface e a necessidade de se implementar a interface `FactoryFinder`. As críticas ou dúvidas mais importantes para este trabalho, apresentadas a seguir, dizem respeito ao método `move`:

1. *No modelo de objetos CORBA a transparência de localização é um ponto chave. É escondida do cliente a localização do objeto. Clientes lidam somente com referências de objetos CORBA, as quais encapsulam informações de localização e de identificação local (dentro de um servidor) dos objetos. Assim, o cliente não consegue inspecionar a referência e descobrir onde está o objeto. Com estas observações, o que significaria "lá" (there) em uma solicitação de migração?*

   Neste trabalho definimos "lá" com uma referência CORBA. Isto significa que "lá' é onde reside a implementação do objeto CORBA representado pela referência. Deste modo, é preservada a transparência de localização.

2. *O servidor para onde foi deslocado o objeto pode utilizar um protocolo diferente do servidor de origem. Assim, o cliente que utilizava o objeto antes da migração deverá conhecer o novo protocolo para continuar utilizando o objeto. Deste modo, para o cliente não existe a transparência de protocolo requerida em CORBA.*

   Assumimos neste trabalho que o único protocolo utilizado é o GIOP/IIOP. Na situação atual, isto não é uma limitação, porque este protocolo tem grande aceitação e utilização.

3. *O objeto que se move pode ter o seu estado persistente em um banco de dados. A*

menos que os servidores origem e destino compartilhem o mesmo banco de dados, o trabalho para implementar o `move` *será difícil.*

A infra-estrutura proposta se preocupa somente com os objetos transientes. Em trabalhos futuros e complementares serão abordados os objetos persistentes.

4. *Devido a transparência de linguagem e implementação, os servidores origem e destino podem ter diferentes arquiteturas de hardware. Como a implementação de objeto pode funcionar nesta situação?*

   Neste trabalho usamos somente Java nas implementações, que resolve este problema com a JVM. Com outras linguagens este problema persiste.

5. *Como as referências continuam válidas após uma migração?*

   A infra-estrutura proposta neste trabalho emprega o mecanismo `LOCATION_FORWARD` para resolver o problema de manter as referências válidas após a migração de um objeto. Os detalhes serão apresentados no capítulo 5.

Capítulo 4

A Infra-Estrutura de Migração

A infra-estrutura de migração proposta neste trabalho consiste num conjunto de servidores CORBA que fornecem contextos de execução para objetos móveis. Estes servidores serão denominados servidores de mobilidade. Cada um deles funciona como hospedeiro para um conjunto de objetos CORBA implementados em Java. Tais objetos podem migrar de um servidor de mobilidade para outro. Os servidores de mobilidade são genéricos, ou seja, podem hospedar objetos móveis com diferentes interfaces IDL, desde que estes objetos sejam implementados seguindo algumas regras. Um servidor de mobilidade não precisará conhecer, em tempo de sua compilação, nem as interfaces IDL nem as classes Java correspondentes aos objetos móveis que ele abrigará. Desta forma, um processo servidor de mobilidade (uma execução do servidor) poderá, inclusive, hospedar objetos cujas interfaces IDL e classes Java foram definidas depois que esse processo começou sua execução.

Este capítulo dará uma visão geral da infra-estrutura de migração e descreverá como ela deve ser utilizada por um implementador de objetos móveis.

4.1 Visão Geral

A figura 4.1 mostra a arquitetura de migração e seus componentes. Observamos dois tipos de aplicação: aplicação cliente e servidor de mobilidade (SM). O primeiro tipo usa os objetos móveis por meio de chamadas CORBA, enviando e recebendo mensagens IIOP. O segundo tipo de aplicação tem como função principal oferecer um contexto de execução para os objetos móveis. As responsabilidades desse contexto incluem a criação e a migração de objetos móveis.

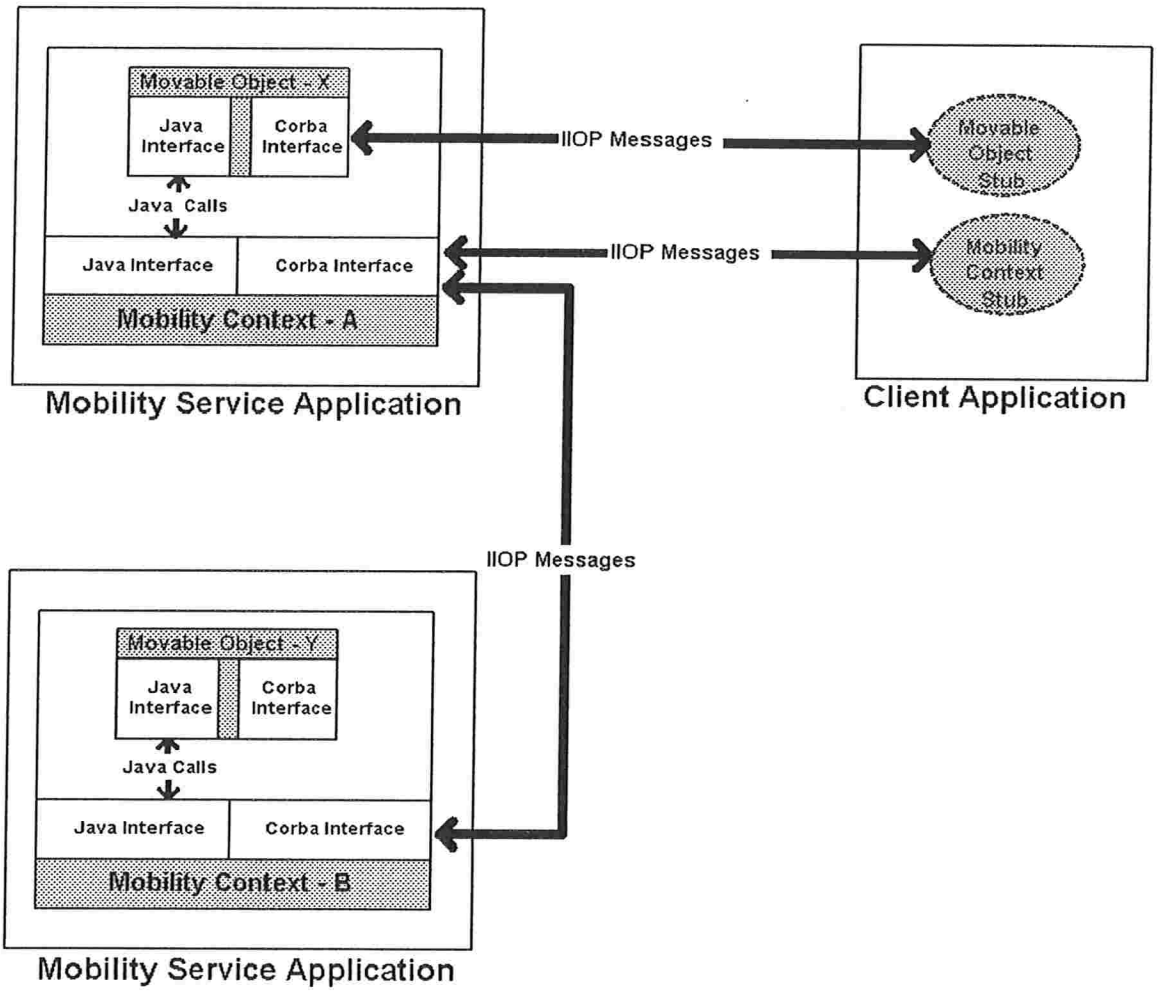

Figura 4.1: Infra-estrutura de migração

O contexto de mobilidade e o objeto móvel implementam, cada um, duas interfaces: uma CORBA e outra Java. A interface CORBA do objeto móvel é utilizada pela aplicação cliente. Já um objeto móvel e seu contexto de mobilidade comunicam-se por meio das suas interfaces Java. Finalmente, a interface CORBA do contexto de mobilidade permite a criação de objetos móveis pelas aplicações clientes e é utilizada para a comunicação entre os servidores de mobilidade. Deste modo, no projeto da infra-estrutura foram escolhidas interfaces Java para a comunicação local e interfaces CORBA para a comunicação remota.

O restante desta seção apresentará de forma resumida todas as interfaces da infra-estrutura: as interfaces CORBA e, depois, as interfaces Java.

As interfaces CORBA são apresentadas na figura 4.2, onde podemos ver o módulo `MobilityService` que agrupou as duas interfaces CORBA existentes na infra-estrutura: `MobilityContext` e `Movable`. A primeira, `MobilityContext`, é a interface CORBA do contexto de mobilidade e apresenta somente três métodos. Um deles, `createMovableObject`, é utilizado por uma aplicação cliente para criar um objeto móvel. Os outros, `receiveMovableObject` e `updateObjectLocation`, são usados na migração de um objeto, ocasião em que ocorrem chamadas inter-servidores. A segunda interface, `Movable`, é empregada por aplicações clientes para solicitar a migração do objeto alvo. Seu único método, `move`, tem semântica síncrona: no retorno de uma chamada bem sucedida a `move` o objeto alvo já está no novo contexto. Essa migração não invalida as referências CORBA para o objeto móvel.

Para completar a apresentação de todas as interfaces, temos as interfaces Java da figura 4.3, empregadas na comunicação entre o contexto de mobilidade e o objeto móvel. O contexto de mobilidade executa métodos da interface `JMovable` do objeto móvel que, por sua vez, executa métodos da interface `JMobilityContext` do contexto de mobilidade.

As observações abaixo sumarizam a arquitetura de migração:

- Os contextos de mobilidade se comunicam por meio de chamadas CORBA e recebem das aplicações cliente as chamadas CORBA solicitando a criação de objetos móveis.
- Os objetos móveis e seu contexto de mobilidade se comunicam por meio de chamadas locais de métodos Java, em um certo servidor de mobilidade.
- As aplicações cliente se comunicam com os objetos móveis por meio de chamadas CORBA.

```
module MobilityService {

    typedef sequence<octet> SerializedServant;
    typedef string ObjectId;
    typedef string MobilityContextRef;
    typedef string RepositoryId;

    // Definições de exceções
    ...

    interface Movable;

    //-------------------------------------------------//
    //--- Interface CORBA do contexto de mobilidade ---//
    //-------------------------------------------------//
    interface MobilityContext {

        Movable createMovableObject(in ObjectId initialId,
                                    in RepositoryId rid,
                                    in string className)
                raises (CannotCreate);

        Movable receiveMovableObject(in SerializedServant incoming,
                                     in MobilityContextRef homeContext,
                                     in ObjectId id,
                                     in ObjectId intialId,
                                     in RepositoryId rid)
                raises (CannotReceive);

        void updateObjectLocation(in ObjectId initialId,
                                  in Movable currentRef)
             raises (CannotUpdate);
    };
    //-----------------------------------------//
    //--- Interface CORBA de um objeto Móvel ---//
    //-----------------------------------------//
    interface Movable {

        void move(in MobilityContextRef there)
             raises (Busy, RemoteSystemException, Error);
    };
};
```

Figura 4.2: Interfaces CORBA da infra-estrutura

```
//-----------------------------------------------//
//--- Interface Java do contexto de Mobilidade ---//
//--- Arquivo: JMobilityContext.java           ---//
//-----------------------------------------------//
package MobilityService;

import org.omg.PortableServer.Servant;

public interface JMobilityContext {

   public void delegateMove(MobilityContext there, Servant theServant)
               throws Busy, MobilityServiceNotActive, Error;

   public void asynchronousMove(MobilityContext there, Servant theServant);

   public Movable servantToReference(Servant theServant)
                  throws ServantNotActive;

   public void deactivate(Servant theServant)
               throws ServantNotActive;

}

//-----------------------------------------------//
//--- Interface Java do objeto Móvel ------------//
//--- Arquivo: JMovable.java         ------------//
//-----------------------------------------------//
public interface JMovable extends Serializable {

   public void onCreation(JMobilityContext homeLocation););

   public void onDeparture();

   public void onArrival(JMobilityContext newLocation);
}
```

Figura 4.3: Interfaces Java da infra-estrutura

4.2 Definição de Objeto Móvel

Neste trabalho adotaremos a seguinte definição de objeto móvel:

Objeto Móvel (OM) - É servente de objeto CORBA que implementa a interface Java `JMovable` e a interface CORBA `Movable`.

Objetos móveis só podem ser instanciados em servidores de mobilidade, mediante chamadas a métodos disponibilizados pela infra-estrutura de migração.

4.3 Interface JMovable

Nesta seção apresentamos a interface `JMovable`, descrevendo os seus métodos e parâmetros.

4.3.1 onCreation

```
public void onCreation(JMobilityContext homeLocation);
```

Descrição

Este método é acionado imediatamente após a criação do objeto móvel. Pode ser utilizado pelo objeto móvel para iniciar alguma variável. O objeto móvel recebe como parâmetro uma referência para o contexto de mobilidade onde foi criado.

Parâmetros

`homeLocation`: É uma referência para o contexto de mobilidade. O objeto móvel recém-criado deve armazenar esta referência para uso posterior.

4.3.2 onDeparture

```
public void onDeparture();
```

Descrição

Antes de ocorrer a migração, este método é executado. Deste modo, o objeto móvel é avisado de que sofrerá uma migração.

Parâmetros

Nenhum.

4.3.3 onArrival

```
public void onArrival(JMobilityContext newLocation);
```

Descrição

Após a migração, já no novo contexto de mobilidade, este método é executado no objeto móvel para avisá-lo de que acabou a migração.

Parâmetros

`newLocation`: É uma referência para o novo contexto de mobilidade. O objeto móvel deve armazenar esta referência para uso posterior.

4.4 Implementação de um Objeto Móvel

Para implementar um objeto móvel é necessário codificar um servente e implementar as interfaces `JMovable` e `Movable`. O objeto móvel deve ser serializável, pois a interface `JMovable` herda a interface `Serializable`.

A interface CORBA `Movable`, apresentada na figura 4.2, é a interface mínima de um objeto móvel. Ela contém um método importante da infra-estrutura: `move`. Para que o desenvolvedor acrescente outros métodos à interface do objeto móvel, será necessário que ele defina uma outra interface com todos os métodos que desejar e que esta última herde a interface `Movable`. Consideremos como exemplo a interface CORBA `Grid` apresentada na figura 4.4.

A figura 4.5 apresenta um exemplo simples de implementação de objeto móvel com a interface CORBA `Grid` da figura 4.4.

```
interface Grid: MobilityService::Movable {
        void set(in short x, in short y, in short value);
        short get(in short x, in short y);
}
```

Figura 4.4: Interface CORBA Grid

Uma instância da classe `GridImpl` satisfaz nossa definição de objeto móvel pois, além de ser um servente, implementa a interface Java `JMovable` e a interface CORBA `Movable`

Eis um resumo das etapas efetuadas quando um cliente solicita a migração de um objeto `Grid`:

1. O cliente obtém uma referência CORBA do contexto de mobilidade destino;
2. O cliente obtém uma referência CORBA para o objeto CORBA `Grid`;
3. O cliente aciona o método `move` do objeto `Grid`, passando a referência do contexto de mobilidade destino;
4. O ORB leva a requisição até o servente `GridImpl`;
5. O método `move` do objeto móvel (`GridImpl`) é acionado pelo POA;
6. O objeto móvel aciona o método `delegateMove` do contexto de mobilidade local, passando a referência recebida como parâmetro: contexto de mobilidade destino;
7. O contexto de mobilidade local migra o objeto móvel para o destino desejado.

Finalizamos esta seção sumarizando as atividades necessárias para implementar um objeto móvel:

1. Na interface CORBA do objeto móvel declaramos a herança da interface `Movable`;
2. Na declaração da classe do servente, é necessário declarar a implementação da interface `JMovable`;
3. Declarar na classe servente um campo do tipo `JMobilityContext` para referenciar o contexto de mobilidade local;
4. Implementar os métodos `onCreation`, `onDeparture`, `onArrival` e `move`.

```
public class GridImpl extends GridPOA
                      implements JMovable
{
    private short height = 31;
    private short width = 14;
    private short [][] mygrid;
    private JMobilityContext location;

    //-----------------------------------------
    public GridImpl()
    {
         mygrid = new short[height][width];
    }
    //-----------------------------------------
    public short get(short n, short m)
    {
         return mygrid[n][m];
    }
    //-----------------------------------------
    public void set(short n, short m, short value)
    {
         if( ( n <= height ) && ( m <= width ) )
              mygrid[n][m] = value;
    }
    //-----------------------------------------
    public void onCreation(JMobilityContext homeLocation) {
         location = homeLocation;
    }

    //-----------------------------------------
    public void onDeparture() {
         //-- Nothing
    }
    //-----------------------------------------
    public void onArrival(JMobilityContext newLocation) {
         location = newLocation;
    }
    //-----------------------------------------
    public void move(MobilityService.MobilityContext there){
         location.delegateMove(there,this);
    }
}
```

Figura 4.5: Implementação do objeto móvel `Grid`

4.5 Utilização de um Objeto Móvel

Uma vez disponibilizadas as classes que implementam certo tipo de objetos móveis, no nosso exemplo objetos do tipo `Grid`, aplicações cliente poderão criar tais objetos. Para criar um objeto móvel, uma aplicação cliente deve chamar o método `createMovableObject` do contexto de mobilidade onde deseja que o objeto móvel seja criado. É necessário que as classes que implementam o objeto estejam disponíveis (possivelmente por carga remota) para o servidor de mobilidade no qual ocorre a criação do objeto. O método `createMovableObject` retorna uma referência CORBA para o objeto móvel recém-criado.

Após a criação de um objeto `Grid`, a aplicação cliente que o criou já pode utilizá-lo. A utilização consiste em realizar chamadas CORBA ao objeto, requisitando a execução dos métodos `set`, `get` ou `move`. Todas as chamadas CORBA citadas podem ser vistas na figura 4.6.

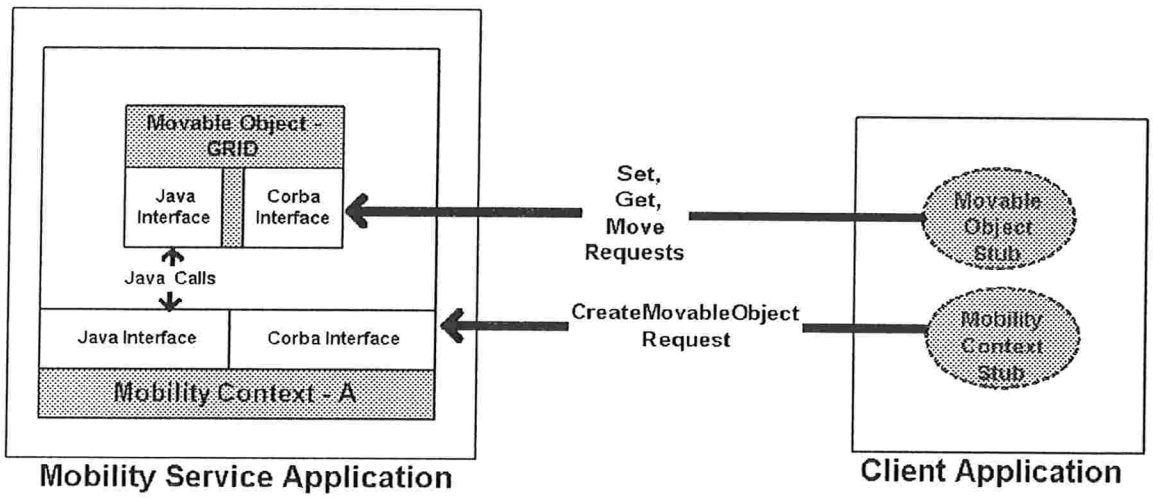

Figura 4.6: Utilização da infra-estrutura

Caso a aplicação cliente chame o método `move` sobre o objeto `Grid` temos que, após o retorno deste método, o objeto móvel estará no servidor de mobilidade destino. Para concretizar a migração, o contexto de mobilidade origem chama o método `receiveMovableObject` sobre o contexto de mobilidade destino e o método `updateObjectLocation` no contexto de mobilidade onde foi criado o objeto móvel, também chamado de contexto home. No nosso exemplo, o contexto origem e o contexto home são coincidentes, porque esta é a primeira solicitação de migração. Na figura 4.7, temos

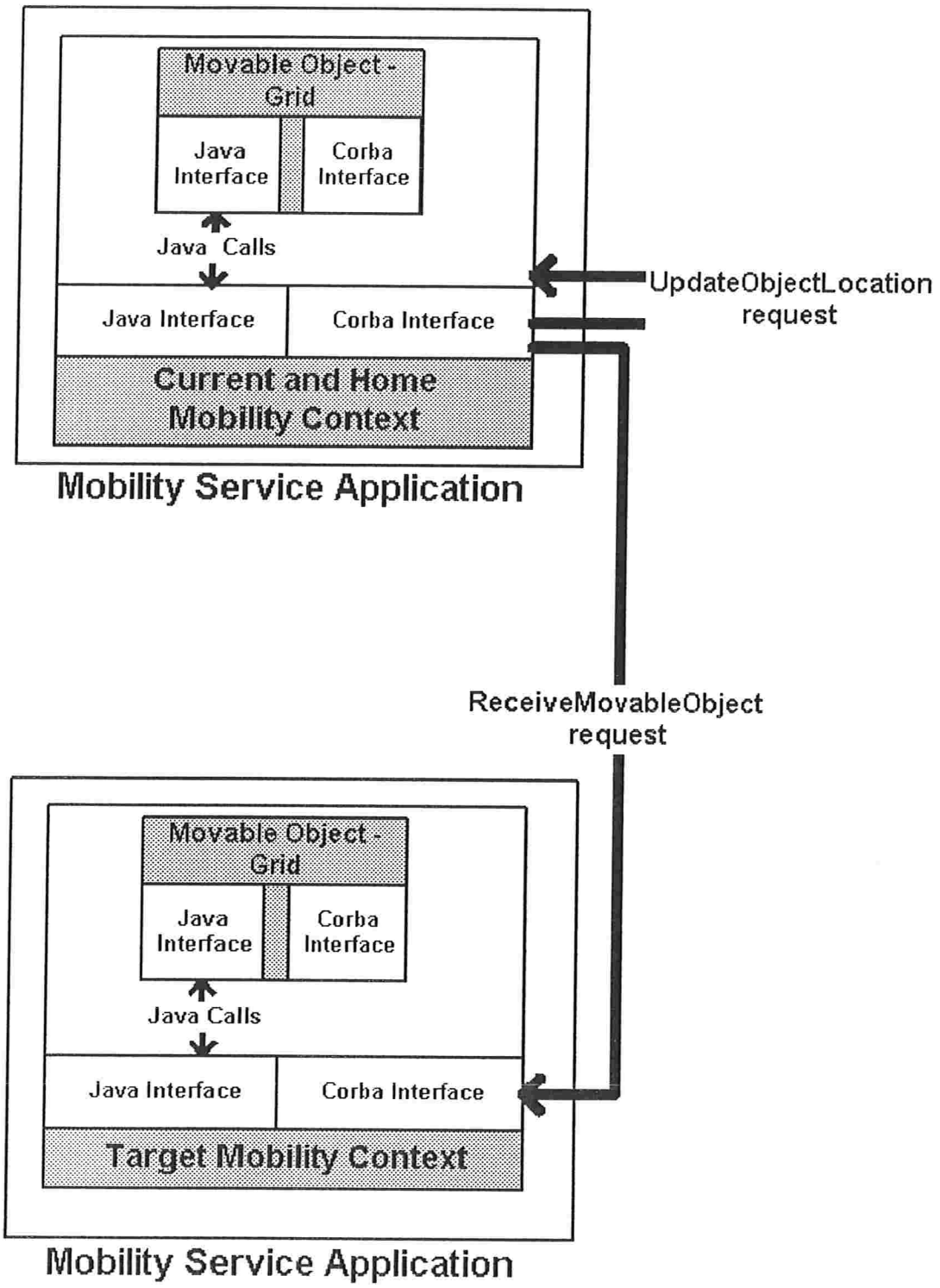

Figura 4.7: Visão geral da comunicação dos servidores de mobilidade

uma visão geral da comunicação entre servidores de mobilidade para realizar uma primeira migração. No próximo capítulo veremos em detalhe como essa infra-estrutura funciona.

Capítulo 5

A Implementação da Infra-Estrutura de Migração

O conjunto dos servidores de mobilidade tem o mesmo código, as mesmas funcionalidades e as mesmas tabelas. O que diferencia cada um desses servidores é o conteúdo das suas tabelas e os objetos móveis nele residentes num certo instante. Para um certo objeto móvel todos os servidores de mobilidades são aparentemente iguais, mas na verdade existe um servidor de mobilidade que é especial para o objeto: aquele no qual o objeto móvel foi criado. Chamaremos este servidor de *servidor de mobilidade home* ou simplesmente de *servidor home*, ficando subtendido que o nome completo seria *servidor de mobilidade home do objeto móvel X*.

Além de abrigar um conjunto de objetos móveis, cada servidor de mobilidade tem seu "objeto principal", um contexto de mobilidade, o qual é responsável pelo processo de migração de objetos. Por um lado, o contexto de mobilidade é um objeto que oferece a interface Java `JMobilityContext`, utilizada por objetos móveis locais. Por outro lado, é um objeto que oferece a interface CORBA `MobilityContext`, mediante a qual os servidores de mobilidade interagem uns com os outros. Esta interação se dá por meio de um protocolo de migração, no qual reside a lógica central da implementação da infra-estrutura.

Antes de apresentar o protocolo de migração, precisamos definir as estruturas de dados utilizadas e detalhar as interfaces Java e CORBA do contexto de mobilidade.

5.1 Estruturas de Dados

Cada servidor de mobilidade (SM) possui duas tabelas: a *Tabela de Serventes Ativos (TSA)* e a *Tabela de Objetos Móveis (TOM)*. As figuras 5.1 e 5.2 apresentam as definições dessas tabelas.

Colunas	**Definição**
`id`	Identificação do OM utilizada em todos os SMs
`objectIdInicial`	Identificação inicial do OM
`estado`	Situação do OM em SM
`servente`	Referência Java do OM
`repositoryId`	Identificação do tipo da interface CORBA do OM
`CORBARefHome`	Referência CORBA do SM onde o OM foi criado (Home)
`contador`	Contador de requisições CORBA em processamento

Figura 5.1: TSA - Tabela de Serventes Ativos

Colunas	**Definição**
`objectId`	Identificação inicial do OM
`reference`	Referência CORBA atual do OM
`referenciaInicial`	Referência CORBA inicial do OM

Figura 5.2: TOM - Tabela de Objetos Móveis

Estas são as duas únicas tabelas usadas na implementação. Todas as informações usadas pela infra-estrutura estão contidas na TSA ou na TOM de algum SM. Cada etapa importante da migração de um objeto provoca a alteração ou a inclusão de linhas nestas tabelas. A seguir, será exposto o significado de cada coluna e os seus possíveis valores.

5.1.1 Tabela de Serventes Ativos

A TSA contém os serventes ativos num SM e guarda outras informações utilizadas pelo protocolo de migração. As colunas desta tabela são:

`id` **(coluna chave)** Esta coluna contém a referência CORBA para o OM convertida para string.

`objectIdInicial` Identificação inicial do OM. Esta identificação é fornecida pelo cliente que solicitou a criação de um Objeto Móvel. Ela é usada na criação de referências CORBA para um OM, quando é necessário informar o *object id.*

`estado` É o estado do OM. Os possíveis estados serão os seguintes:

1. Instanciado;
2. Irá migrar;
3. Migrado.

Conforme o andamento da migração esta coluna será atualizada, permitindo que se acompanhe o processo de migração e que se verifique se há alguma situação anormal.

`servente` **(coluna chave)** Referência Java do OM. O `ServantLocator` utilizado no servidor de mobilidade retorna esta referência ao POA, para que este encaminhe um requisição ao OM. Esta coluna é chave, devido à necessidade de se efetuar "buscas por servente" na TSA.

`repositoryId` É a identificação do tipo da interface CORBA do OM. Esta identificação é utilizada, juntamente com o `objectIdInicial`, na criação da referência CORBA do OM.

`CORBARefHome` É a referência CORBA do contexto de mobilidade que reside no servidor de mobilidade onde foi criado o OM indicado pela coluna `id`. Em outras palavras, é a referência CORBA do servidor home do OM.

`contador` Este contador indica quantas requisições CORBA estão sendo tratadas no momento. Quando o objeto móvel não estiver processando nenhuma requisição CORBA, este contador será zero.

5.1.2 Tabela de Objetos Móveis

A TOM de um SM mantém informações sobre todos os OMs nele criados. Em outras palavras, para cada OM existirá uma única entrada na TOM de seu SM home. As colunas desta tabela são:

`ObjectId` **(coluna chave)** Identificação de um OM. Este valor é escolhido pela aplicação que solicitou a criação do objeto móvel. É a chave da tabela, identificando unicamente um OM no seu SM home.

`reference` Referência CORBA atual do OM. É gerada uma nova referência CORBA para um OM sempre que ele se desloca para outro servidor de mobilidade. Nesta coluna é armazenada a mais recente dessas referências

`referenciaInicial` É a primeira referência CORBA do OM. É aquela que foi gerada pelo servidor home e retornada para o cliente que solicitou a criação de OM.

A TSA é implementada por uma classe Java desenvolvida neste trabalho como classe utilitária. Esta classe contém basicamente os métodos `put`, `get` e `remove`. O uso de duas `Hashtables` é necessário porque uma `Hashtable` espera somente uma chave de busca, mas a tabela TSA tem duas chaves. Deste modo, quando se executa `put` e `remove`, estas operações estão na realidade sendo efetuadas em duas `Hashtables`, de modo que estas estruturas de dados fiquem sempre sincronizadas. No caso do `get`, existem duas possibilidades: uma passando um servente e outra informando o `id`, pois ambos os parâmetros são chaves da TSA.

A classe utilitária que implementa a TSA faz todo o controle de concorrência necessário para suportar multi-threading, isto é, todos os seus métodos são `synchronized`.

5.2 Interface Java do Contexto de Mobilidade

O cérebro da infra-estrutura é o contexto de mobilidade. Este é o mais complexo e importante objeto para o entendimento da implementação da infra-estrutura. O contexto de Mobiliade tem duas interfaces: a interface Java `JMobilityContext`(figura 4.3), cujos métodos são descritos nesta seção, e a interface CORBA `MobilityContext`, cujos métodos serão descritos na próxima seção.

5.2.1 delegateMove

```
public void delegateMove(MobilityContext there,
                         Servant theServant)
            throws Busy, RemoteSystemException, Error;
```

Descrição

Este método é usado pelo objeto móvel para repassar ao contexto de mobilidade uma requisição de migração. O objeto móvel recebe requisições de migração por meio do método `move` da interface `Movable`, e as repassa a seu contexto de mobilidade através deste método. Chamadas a `delegateMove` só podem ser feitas pela implementação do método `move` da interface `Movable`.

Parâmetros

`there`: Este parâmetro é uma referência CORBA do contexto de mobilidade destino.

`theServant`: É a referência Java do servente que está chamando este método. Na prática, o objeto móvel passará neste parâmetro a referência `this`.

Exceções

`Busy`: Esta exceção pode ocorrer caso o POA esteja configurado para o modo multi-threaded. Nesta situação, enquanto uma requisição está sendo processada, poderá chegar uma outra requisição para que o objeto se desloque. O contexto de mobilidade esperará um tempo pré-determinado até que todas as requisições em processamento terminem. Após este tempo, caso exista alguma requisição em processamento, a infra-estrutura disparará a exceção `Busy`, relatando o problema.

`RemoteSystemException`: Esta exceção ocorre caso o servidor de mobilidade destino ou o servidor home não estejam ativos.

`Error`: Qualquer outra situação de erro encontrada pela infra-estrutura irá resultar na exceção `Error`.

5.2.2 asynchronousMove

```
public void asynchronousMove(MobilityContext there,
                             Servant theServant);
```

Descrição

Inicia uma migração assíncrona do servente `theServant`. Uma chamada a este método apenas registra a migração desse objeto como pendente e retorna sem efetuá-la. A migração propriamente dita ocorrerá em algum momento posterior ao retorno da chamada.

O método `asynchronousMove` pode ser usado, por exemplo, por um objeto móvel que migra por iniciativa própria. O método `delegateMove` não poderia ser usado para essa finalidade, pois só pode ser chamado pela implementação do método `move` da interface CORBA `Movable`.

Parâmetros

`there`: Este parâmetro é uma referência CORBA do contexto de mobilidade destino.

`theServant`: É a referência Java do servente que está chamando este método. Na prática, o objeto móvel passará neste parâmetro a referência `this`.

5.2.3 servantToReference

```
public Movable servantToReference(Servant theServant)
                throws ServantNotActive;
```

Descrição

Este método serve para o objeto móvel obter a sua referência CORBA. A referência obtida é a primeira referência CORBA criada para o objeto móvel.

Parâmetros

`theServant`: É a referência Java do servente que está chamando este método. Na prática, o objeto móvel passará neste parâmetro a referência `this`.

Exceções

`ServantNotActive`: Esta exceção é disparada caso `theServant` não esteja ativo.

5.2.4 deactivate

```
public void deactivate(Servant theServant)
            throws ServantNotActive;
```

Descrição

Este método desativa o servente representado pelo parâmetro. Após a execução deste método, nenhuma requisição CORBA para o objeto móvel desativado será processada.

Parâmetros

`theServant`: É a referência Java do servente que está chamando este método. Na prática, o objeto móvel irá passar neste parâmetro a referência `this`.

Exceções

`ServantNotActive`: Caso a referência, passada como parâmetro, não esteja mais ativa ou incorreta, esta exceção é disparada.

5.3 Interface CORBA do Contexto de Mobilidade

A interface CORBA `MobilityContext`, apresentada na figura 4.2, é usada por outros contextos de mobilidade e por clientes CORBA. Um de seus métodos, `createMovableObject`, pode ser chamado por qualquer cliente CORBA. Este método não é chamado por outros contextos de mobilidade. Os outros dois métodos desta interface, `receiveMovableObject` e `updateObjectLocation`, são de uso exclusivo dos contextos de mobilidade, que os chamam para implementar o protocolo de migração.

Esta seção descreve cada um dos métodos presentes na interface CORBA dos contextos de mobilidade. Após estas descrições, poderemos apresentar o protocolo de migração.

5.3.1 createMovableObject

```
Movable createMovableObject(in ObjectId initialId,
                            in CORBA::RepositoryId rid,
                            in string className);
```

Descrição

Quando os servidores de mobilidade iniciam a sua execução, não existe nenhum objeto móvel. Só existe a infra-estrutura necessária para realizar migrações. Para criar objetos móveis temos este método, `createMovableObject`, cujos parâmetros são necessários para se instanciar um objeto e para se criar uma referência CORBA do OM. Esta referência é o retorno deste método (objeto do tipo `Movable`). Logo após a instanciação do objeto móvel, este método chamará o método `onCreation` do OM.

Parâmetros

`initialId`: Identificação inicial do OM. Esta identificação é escolhida pelo usuário da infra-estrutura.

`rid`: Identificação do repositório de interface do OM. Este parâmetro, juntamente com o parâmetro anterior, contém as informações necessárias para se criar uma referência CORBA.

`className`: Nome da classe da qual se instanciará um objeto móvel. Deve-se passar o nome de todas os pacotes Java que antecedem o nome da classe. Por exemplo: `grid.GridImpl`, para a classe `GridImpl` do pacote `grid`.

5.3.2 receiveMovableObject

```
Movable receiveMovableObject(in SerializedServant incoming,
                             in MobilityContext homeContext,
                             in ObjectId id,
                             in ObjectId intialId,
                             in CORBA::RepositoryId rid);
```

Descrição

Este método é chamado em um contexto de mobilidade para que ele receba um servente. O servente recebido como parâmetro é o objeto móvel que está migrando. Deste modo, o contexto de mobilidade que recebe esta requisição, na verdade, estará recebendo um objeto móvel. Uma outra maneira de nomear este método seria: *receba o objeto móvel passado como parâmetro e o instancie*. Além de instanciar um objeto móvel, este método cria uma referência CORBA para o objeto e a retorna para o chamador.

Parâmetros

`incoming`: Este parâmetro é o objeto móvel serializado. Portanto ele contém o estado do objeto móvel, bem como informações sobre a classe deste objeto.

`homeContext`: É a referência CORBA para o contexto de mobilidade home do OM.

`id`: É a identificação do OM. Este valor pode ser usado como chave na tabela TSA.

`intialId`: É a primeira identificação do OM, aquela que o contexto de mobilidade home usa para identificar um OM. Assim, este parâmetro será útil quando houver necessidade de se comunicar com o contexto de mobilidade home.

`rid` Este parâmetro identifica o tipo do objeto CORBA e só é usado, neste trabalho, na criação de referências CORBA.

5.3.3 updateObjectLocation

```
void updateObjectLocation(in ObjectId initialId,
                          in Movable currentRef);
```

Descrição

Embora as migrações de um OM sejam transparentes para os clientes deste objeto, a cada migração o servidor de mobilidade destino cria uma nova referência para o OM. O contexto de mobilidade que recebe uma chamada deste método está sendo informado da nova referência de um certo OM.

Parâmetros

`initialId`: É a identificação inicial do objeto móvel. Com este parâmetro, o contexto de mobilidade sabe de qual OM é a referência CORBA informada no segundo parâmetro.

`currentRef`: Nova referência CORBA do OM.

5.4 Protocolo de Migração

O protocolo de migração especifica a seqüência de ações disparada por uma requisição de migração de um OM. Algumas destas ações são tomadas por iniciativa do método `delegateMove` do contexto de mobilidade, outras são efetuadas por um `ServantLocator` associado ao contexto de mobilidade. Esta série de ações, que culmina com a migração do objeto de um SM para outro, provoca alterações nas estruturas de dados de três servidores de mobilidade: o SM origem, o SM destino e o SM home do OM.

Os eventos do protocolo podem ser visualizados na figura 5.3. Nesta figura, as setas tracejadas representam retornos de métodos. A seguir, detalhamos cada evento que compõe o protocolo de migração.

1. *Um cliente chama o método* `move` *(da interface CORBA* `Movable`*) de um objeto móvel.*

   O cliente começa, nesta descrição do protocolo, a migração do objeto móvel.

2. *O POA chama o método* `preinvoke` *do* `ServantLocator` *do servidor de mobilidade origem.*

   O `ServantLocator` busca na TSA o objeto móvel alvo da requisição. Caso não o encontre ou caso o status deste objeto seja diferente de "Instanciado", ele lança uma exceção `ForwardRequest`. Caso encontre o objeto alvo da requisição, caso o status deste objeto seja "Instanciado" e caso a operação requisitada seja `move`, o `ServantLocator` incrementa o contador da TSA correspondente a este objeto móvel, atualiza o seu status para "Irá migrar" e retorna a referência do objeto móvel.

3. *O POA chama o método* `move` *do objeto móvel.*

   O POA efetua esta chamada utilizando a referência do objeto móvel previamente retornada pelo `ServantLocator`.

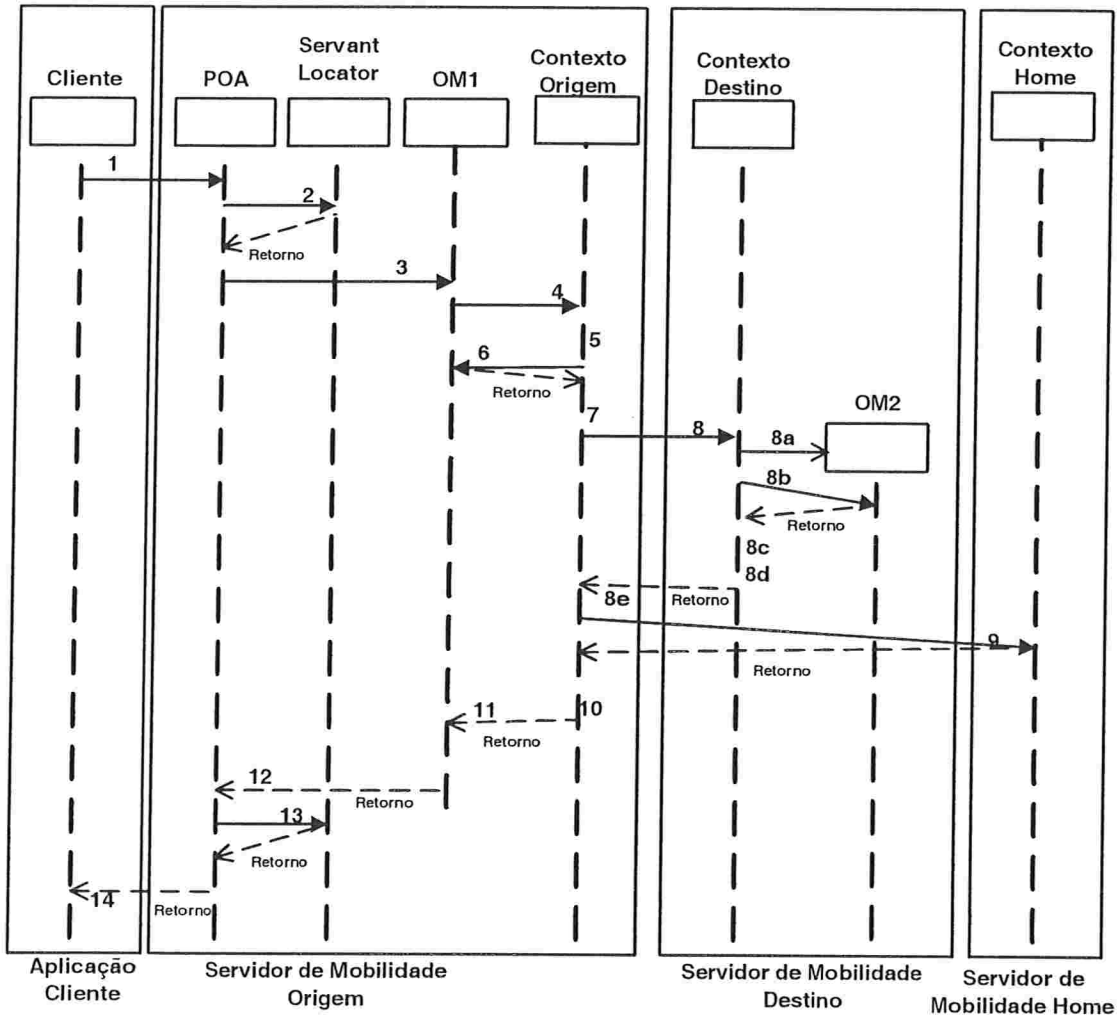

Figura 5.3: O Protocolo de migração

4. *O OM chama o método* `delegateMove` *(da interface Java* `JMobilityContext`*) no contexto de mobilidade.*

   Para poder efetuar esta chamada, o OM precisa possuir uma referência Java para o seu contexto de mobilidade. Ele deve implementar os métodos `onCreation` e `onArrival` de modo a manter essa referência sempre atualizada.

5. *O contexto verifica se pode prosseguir a migração, senão retorna a exceção* `Busy`.

   Por meio do contador da TSA, o contexto de mobilidade verifica se existe alguma outra requisição CORBA em processamento no objeto móvel. Se existir, aguarda o término deste processamento por no máximo n milisegundos e depois prossegue a migração. Caso o processamento de uma requisição concorrente não termine nesse prazo, o contexto de mobilidade retorna a exceção `Busy`. O valor de n é um parâmetro de configuração do servidor de mobilidade.

6. *O contexto de mobilidade chama o método* `onDeparture` *(da interface* `JMovable`*) do OM.*

   Este método é chamado com o objetivo de informar o OM que uma migração será realizada e que ele está prestes a ser serializado.

7. *O contexto de mobilidade serializa o OM.*

   Esta operação é possível dois motivos: (1) o OM implementa a interface `JMovable`, que estende a interface `Serializable`, e (2) o contexto de mobilidade recebeu, como parâmetro do método `delegateMove`, uma referência Java para o OM.

8. *O contexto de mobilidade chama o método* `receiveMovableObject` *(da interface CORBA* `MobilityContext`*) no servidor de mobilidade destino.*

   O contexto de mobilidade alvo da requisição `receiveMovableObject` é especificado pela referência CORBA passada ao método `delegateMove`, no parâmetro `there`.

   Na chamada a `receiveMovableObject`, os parâmetros `homeContext`, `id`, `intialId` e `rid` são obtidos da TSA. O parâmetro `incoming` é o OM serializado produzido na etapa anterior.

   O contexto de mobilidade que receber esta requisição no servidor de mobilidade destino cumprirá as seguintes tarefas:

(a) Desserializará o OM (um servente CORBA);

(b) Chamará o método `onArrival` (da interface Java `JMovable`) do servente;

(c) Ativará o servente e colocará a sua referência na TSA;

(d) Criará uma referência CORBA com os parâmetros recebidos (Repository Id, Object Id);

(e) Retornará a referência CORBA criada.

9. *O contexto de mobilidade chama o método* `updateObjectLocation` *(da interface CORBA* `MobilityContext`*) no contexto de mobilidade home do OM.*

   É possível chamar o método `updateObjectLocation` porque temos a nova referência CORBA do OM, aquela devolvida pelo método `receiveMovableObject`, e também a referência do contexto de mobilidade home do OM, que se encontra na TSA.

   O contexto de mobilidade home irá atualizar a sua TOM com a nova referência do OM.

10. *O contexto de mobilidade atualiza o status do objeto móvel.*

    O contexto de mobilidade atualiza na TSA a linha correspondente ao OM. Nesta atualização coloca o status do objeto móvel como "Migrado".

11. *O contexto de mobilidade origem retorna o controle para o OM.*

    Neste ponto as tarefas do contexto de mobilidade origem se encerram e a chamada ao método `delegateMove` retorna.

12. *O OM termina a execução do método* `move`.

    Neste ponto o controle volta para o POA.

13. *O POA chama o método* `postinvoke` *do* `ServantLocator`.

    O `ServantLocator` é chamado para completar o protocolo de migração. Ele verifica o status do objeto móvel. Caso seja "Migrado", o `ServantLocator` retira o objeto móvel da TSA. Se for qualquer outro valor, escreve "Instanciado" no status e decrementa o contador correspondente na TSA.

14. *O POA retorna a resposta do* `move` *para o cliente.*

    O POA termina o seu trabalho e envia a mensagem de resposta para o cliente que fez a requisição.

Ainda há uma questão a ser esclarecida: como os clientes CORBA que utilizavam o OM que migrou poderão continuar funcionando como se o OM nunca tivesse migrado? A resposta a esta questão virá com o detalhamento das referências CORBA do OM, assunto da próxima seção.

5.5 Referências CORBA de um OM

A primeira referência CORBA de um OM é gerada na criação do objeto, pelo método `createMovableObject` da interface CORBA `MobilityContext`. No que segue esta referência é denominada *referência inicial* do OM. Ela fica armazenada na TOM do SM home do objeto móvel, na coluna `referenciaInicial`, bem como na TSA do SM atual do objeto móvel, na coluna `id`. No caso de um OM recém criado num dado servidor, a referência inicial aparece tanto na TOM como na TSA desse servidor (que é ao mesmo tempo o SM atual e o SM home do OM).

Quando o OM for migrar, o contexto de mobilidade origem efetuará uma chamada ao método `receiveMovableObject` do contexto de mobilidade destino. O contexto de mobilidade origem passará, no parâmetro `id` desse método, o valor da coluna `id` da TSA. Deste modo, a referência retornada pelo método `receiveMovableObject` conterá a referência inicial.

Esse esquema garante que todas as referências criadas para um mesmo OM, excluindo-se a inicial, têm em seu campo object id a referência inicial. Em outras palavras, todas as requisições para um certo OM, carregam a referência inicial do OM, independentemente do servidor que abriga este OM no momento. Se a requisição for efetuada empregando-se a referência inicial, esta aparecerá no campo alvo da requisição. Se a requisição for feita empregando-se outra referência, a referência inicial aparecerá no campo object id da referência contida no campo alvo da requisição.

Referências contendo outras referências são um recurso chave para o funcionamento da infra-estrutura. O motivo da sua utilização será esclarecido na próxima seção.

5.6 O Objeto `ServantLocator`

Se um POA estiver configurado para usar um `ServantLocator`, todas as requisições que chegarem a esse POA causarão chamadas aos métodos `preinvoke` e `postinvoke` do seu

ServantLocator. Este é o caso do POA que gerencia os objetos móveis num servidor de mobilidade. Seu ServantLocator é uma instância da classe ServantLocatorImpl, que estende a classe ServantLocatorPOA do ambiente padrão CORBA e foi escrita especificamente para a infra-estrutura de migração.

A classe ServantLocatorImpl é muito importante, pois todas as requisições para objetos passam por um ServantLocator. Por isso ela merece uma apresentação detalhada. A figura 5.4 mostra um versão simplificada do seu código. Nela aparecem as três situações que um ServantLocator poderá encontrar no processamento de uma requisição para algum OM. Os três casos são:

1 - A referência para o OM está na TSA. Esta é a situação mais comum: o OM alvo da requisição está ativo neste servidor de mobilidade, isto é, está ativo no mesmo servidor de mobilidade que o objeto ServantLocator. Neste caso, o método preinvoke simplesmente pega a referência Java para o OM, a qual se encontra na TSA, e a retorna ao POA, para que este complete o processamento da requisição.

2 - A referência para o OM não está na TSA, mas está na TOM. Esta situação ocorre quando o OM alvo da requisição não se encontra ativo no servidor de mobilidade do ServantLocator (não aparece na TSA) e, além disso, quando esse servidor é o SM home do OM alvo. Neste caso, o método preinvoke pega a referência CORBA atual do OM, existente na TOM e mantida atualizada pelo protocolo de migração, e lança uma exceção ForwardRequest com essa referência. Isso faz com que uma resposta com status LOCATION_FORWARD, contendo essa referência, seja enviada ao cliente. Recebendo esta resposta, a biblioteca do ORB existente no cliente reenvia a requisição, agora utilizando como alvo a referência contida na resposta LOCATION_FORWARD, para um servidor de mobilidade que se espera estar na situação 1 (terá em sua TSA a referência para o OM).

3 - A referência para o OM não está na TSA e nem na TOM. Esta situação ocorre se o OM alvo da requisição não estiver ativo no SM do ServantLocator e se este SM não for o servidor home do OM alvo. Isto pode acontecer quando o OM sair do seu servidor home, passar por algum servidor intermediário e depois for para outro SM. Neste caso o servidor intermediário não é o servidor home do objeto móvel e nem abriga este objeto, mas pode ainda receber requisições para ele. Para entendermos como se resolve esta situação, é necessário lembrarmos que toda requisição destinada a um OM e enviada a um servidor de mobilidade diferente do servidor home

```
public class ServantLocatorImpl extends ServantLocatorPOA {

 ORB _orb;
 TSA _tsa;
 TOM _tom;

 public ServantLocatorImpl(TSA tsa,TOM tom,ORB orb) {
   _tsa = tsa; _tom = tom;  _orb = orb;
 }

 public Servant preinvoke(byte[] oid, POA adapter, String operation,
                          CookieHolder cookie)
        throws ForwardRequest {

  //-- O objeto móvel está na TSA ?
  String oidStr = new String(oid);
  TSAObject tsaObj = (TSAObject) _tsa.get(oidStr);

  if ( tsaObj == null ) {
   //-- Não. Então, o objeto móvel foi criado aqui ?
   TOMObject tomObj = (TOMObject) _tom.get(oidStr);

   if ( tomObj == null ) {
    //-- Não. Então, o object Id do objeto móvel é a referência inicial.
    org.omg.CORBA.Object om = _orb.string_to_object(oidStr);

    //-- Assim, enviaremos um Location Forward com a referência inicial.
    throw new ForwardRequest(om);
   }
   else {
    //-- Sim. Deste modo, temos a referência atual do OM na TOM.
    org.omg.CORBA.Object omAtual= _orb.string_to_object(tomObj.reference);

    //-- Assim, basta que enviemos um Location Forward para
    //-- a referência atual do objeto móvel
    throw new ForwardRequest(omAtual);
   }
  }
  else {
   //-- Sim. Assim, basta que retornemos a referência do OM.
   return tsaObj.servente;
  }
 }
}
```

Figura 5.4: Implementação do ServantLocator

desse OM contém sempre, no campo object id de sua referência alvo, a referência inicial do OM. Basta, portanto, que o método `preinvoke` extraia da requisição a referência inicial e lance uma exceção `ForwardRequest` com essa referência. Isto causa o envio, ao cliente, de uma resposta com status `LOCATION_FORWARD` contendo a referência inicial. Recebendo esta resposta, a biblioteca do ORB do cliente reenvia a requisição, agora tendo como alvo a referência inicial. O servidor que receber esta requisição certamente estará na situação 1 (é o servidor home do OM e contém o OM) ou na situação 2 (é o servidor home do OM e não contém o OM).

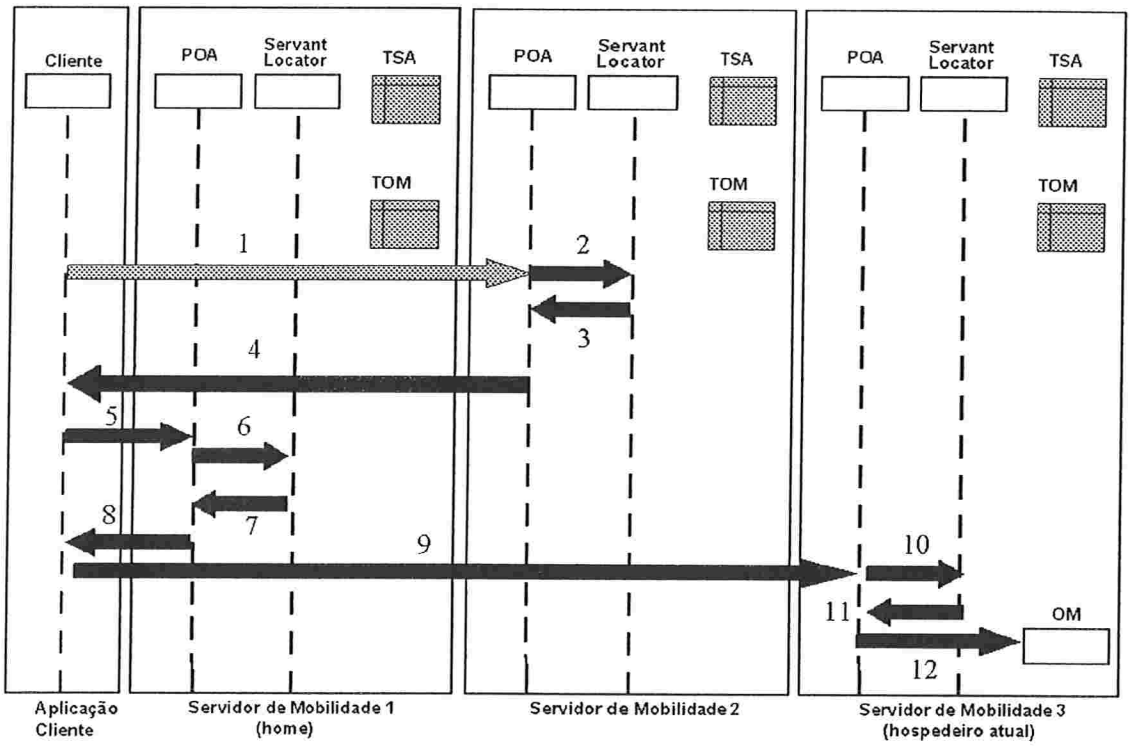

Figura 5.5: Encaminhamento de uma requisição para um OM

A figura 5.5 mostra todos os passos de uma requisição para um OM que não está no home nem no servidor de mobilidade que recebeu a requisição. Ela ilustra o encaminhamento de uma requisição para um OM que foi criado no servidor de mobilidade 1, passou pelo servidor de mobilidade 2 e se encontra atualmente no servidor de mobilidade 3. A requisição é inicialmente enviada ao servidor de mobilidade 2. Este caso, que corresponde ao ítem 3 do parágrafo anterior, "A referência para o OM não está na TSA e nem na TOM",

será detalhado a seguir:

1. *A requisição chega no servidor de mobilidade 2.*

   O POA recebendo a requisição irá passá-la para o Servant Locator para que este devolva um servente ou lance uma exceção `ForwardRequest` com uma nova referência.

2. *A requisição chega no Servant Locator.*

   O Servant Locator irá procurar na TSA e na TOM o servente alvo da requisição. Neste caso, ele não encontra o servente em nenhuma das tabelas.

3. *O Servant Locator lança uma exceção* `ForwardRequest` *para o POA.*

   É importante ressaltar que o Servant Locator coloca a referência inicial do Objeto Móvel na exceção lançada.

4. *O POA devolve uma resposta com o status* `LOCATION_FORWARD` *para o cliente.*

   Esta resposta contém a referência inicial do Objeto Móvel.

5. *O stub do cliente reenvia a requisição utilizando a referência inicial.*

   A referência inicial direciona a requisição para o servidor de mobilidade home do Objeto Móvel.

6. *A requisição chega no Servant Locator do servidor home.*

   O Servant Locator irá procurar na TSA e na TOM o servente alvo da requisição. Neste caso, ele não encontra na TSA, mas encontra na TOM a referência atual do Objeto Móvel.

7. *O Servant Locator lança uma exceção* `ForwardRequest` *para o POA.*

   Esta exceção contém a referência atual do Objeto Móvel.

8. *O POA devolve uma resposta com o status* `LOCATION_FORWARD` *para o cliente.*

   Esta resposta contém a referência atual do Objeto Móvel.

9. *O stub do cliente reenvia a requisição utilizando a referência atual.*

   A referência atual direciona a requisição para o servidor de mobilidade onde o Objeto Móvel reside.

10. *A requisição chega no Servant Locator.*

    O Servant Locator irá procurar na TSA e na TOM o servente alvo da requisição. Neste caso, ele encontra o servente na TSA.

11. *O Servant Locator devolve o servente do Objeto Móvel para o POA.*

    O servente encontrado na TSA é devolvido para o POA.

12. *O POA aciona o método requisitado no Objeto Móvel.*

    Este é o último passo para que a requisição seja processada pelo Objeto Móvel.

É importante ressaltar que as informações sobre um OM não ficam distribuídas por todos os servidores de mobilidade ao longo do caminho percorrido pelo OM. Somente o SM home e o SM atual de um OM (os quais podem ser coincidentes) têm informações sobre esse OM. Em outras palavras, a migração de objetos "não deixa rastros" e é escalável com respeito ao número de migrações efetuadas. Se considerarmos apenas o consumo de recursos mantidos pela infra-estrutura de migração, cada OM ocupa somente duas entradas em tabelas (uma entrada da TOM de seu SM home e outra na TSA de seu SM atual), independentemente de quantas vezes ele já migrou ou venha a migrar.

Cabe ainda observar que, em parte, a simplicidade do código da infra-estrutura de migração é conseqüência da divisão de responsabilidades entre ela e os clientes CORBA. Estes, embora não façam parte da infra-estrutura, participam do protocolo de migração aceitando respostas com status `LOCATION_FORWARD`. A participação dos clientes CORBA não requer nenhum esforço de programação, pois as todas as ações que um cliente CORBA toma ao receber uma resposta com status `LOCATION_FORWARD` são efetuadas no nível da biblioteca do ORB, sem envolvimento algum do código de aplicação.

Além das ações apresentadas na figura 5.4, o `ServantLocator` atualiza o valor do campo `contador` da entrada da TSA referente ao objeto móvel alvo da requisição. O `ServantLocator` incrementa o valor desse campo nas chamadas ao método `preinvoke` e o decrementa nas chamadas ao método `postinvoke`. Isto ocorre a cada requisição para um objeto móvel, independentemente dela ser uma requisição de migração ou uma chamada CORBA a outro método qualquer. O valor desse contador permite que o contexto de mobilidade verifique se um dado OM se encontra ocupado processando requisições, e portanto não pode ser migrado já. O contexto de mobilidade lançará a exceção `Busy` se o OM alvo de uma solicitação de migração estiver ocupado e permanecer assim por mais de n milisegundos, onde n é um parâmetro de configuração do servidor de mobilidade.

5.7 Políticas do POA

Cada servidor de mobilidade cria um POA para uso exclusivo dos objetos móveis. Em nossa implementação, este POA é criado com as políticas apresentadas na tabela 5.1. Esta combinação de políticas não é necessariamente a única escolha possível, mas atende plenamente às necessidades da infra-estrutura de migração.

Política	**Valor escolhido**
id assignment policy	`USER_ID`
lifespan policy	`PERSISTENT`
servant retention policy	`NON_RETAIN`
request processing policy	`USE_SERVANT_MANAGER`

Tabela 5.1: Políticas escolhidas

O valor `USER_ID` para a política *id assignment* nos permite definir a identificação do objeto. Esta recurso é usado na criação de um objeto móvel, pois o campo id da referência inicial do objeto é especificado pelo usuário, bem como a cada migração do objeto, pois todas as referências geradas nas sucessivas migrações do objeto devem conter a referência inicial em seu campo id.

A combinação de `PERSISTENT`, `NON_RETAIN` e `USER_SERVANT_MANAGER` é crucial para nossa implementação, pois esta combinação de valores de políticas permite a utilização de um `ServantLocator`.

5.8 Localização e Carga das Classes dos Objetos Móveis

Embora não consideremos este tópico tão crucial quanto alguns dos que acabamos de abordar (o protocolo de migração, o uso de referências dentro de referências e a implementação do `ServantLocator`), temos de esclarecer como a infra-estrutura de mobilidade encontra e carrega o código das classes dos objetos móveis.

A serialização de um objeto Java inclui tanto o estado do objeto como algumas informações sobre a classe do objeto (o nome completo da classe e o seu identificador de versão), mas não inclui os bytecodes dessa classe. Esses bytecodes precisam ser encontrados e carregados sempre que o primeiro objeto móvel de uma certa classe for instanciado em um

servidor de mobilidade. Essa necessidade pode surgir tanto na criação de um objeto móvel nesse servidor como na migração de um objeto móvel para esse servidor.

Colocar previamente as classes (os arquivos `.class`) dos objetos móveis em todas as máquinas que rodam servidores de mobilidade não é factível, pois estas classes podem ser criadas depois que os servidores começaram a rodar. Além disso, é de se esperar que o conjunto de servidores de mobilidade em execução varie ao longo do tempo.

Também não é desejável supor que todos os servidores de mobilidade compartilhem um sistema de arquivos e que cada um deles possa ter em seu `CLASSPATH` um ou mais diretórios compartilhados, nos quais seriam colocados todos os arquivos `.class` correspondentes às classes dos objetos móveis. Tal arranjo seria desnecessariamente restritivo.

A solução que adotamos foi utilizar a classe `URLClassLoader`, existente no JDK 1.2.2, e carregar o código das classes dos objetos móveis via HTTP. Os arquivos `.class` são disponibilizados em um diretório de um servidor HTTP. Cada servidor de mobilidade emprega um `URLClassLoader` construído com a URL desse diretório, ou seja, todos os servidores de mobilidade passam como parâmetro ao construtor da classe `URLClassLoader` a mesma URL HTTP. Esta solução resolve o problema da distribuição das classes e centraliza os arquivos `.class` num só ponto.

A utilização da classe `URLClassLoader` é simples, como podemos ver no trecho de código apresentado na figura 5.6 e empregado pelo servidor de mobilidade.

Embora o problema pareça totalmente resolvido, o emprego de uma classe do tipo `URLClassLoader` envolve duas sutilezas. A primeira é que o `URLClassloader` não será utilizado caso a classe que se deseja carregar esteja disponível no `CLASSPATH`.

A segunda sutileza envolve a serialização e a desserialização. Como já vimos, a serialização é feita por meio do método `writeObject` da classe `ObjectOutputStream` e a desserialização por meio do método `readObject` da classe `ObjectInputStream`. A implementação do método `readObject` precisa carregar a classe do objeto serializado. A carga dessa classe é feita com o class loader da classe que efetuou a chamada a `readObject`, o qual pode (a menos que se tome os devidos cuidados) ser o class loader que utiliza o `CLASSPATH`. Para evitar isso, criamos a classe `Serial` e implementamos o servidor de mobilidade de modo que toda serialização ou desserialização de um objeto móvel seja efetuada mediante chamada a um método dessa classe. A figura 5.7 mostra o código da classe `Serial`.

O servidor de mobilidade carrega a classe `Serial` com o mesmo `URLClassLoader` que

```
....
//--- Define a URL a ser utilizada na procura pela classe.
URL url1 = new URL("http://servidorDeClasses/classes/");

//--- Define o array de URLs, por que este é o parâmetro
//--- para o construtor URLClassLoader.
URL [] urls = {url1};

//--- Cria um objeto do tipo URLClassLoader
_urlClassloader = new URLClassLoader(urls);

//--- Carrega uma classe, informada como parâmetro,
//--- através do objeto _urlClassloader
Class  classeOM = _urlClassloader.loadClass(nomeDaClasse);

//--- Instancia um objeto a partir da classe carregada.
java.lang.Object objetoOM = classeOM.newInstance();
...
```

Figura 5.6: Utilização da classe URLClassLoader

se deseja usar para carregar as classes dos objetos móveis. Assim, o método `readObject` utilizará este class loader sempre que for invocado (pelo método `deserialization` da classe `Serial`) para desserializar um objeto móvel.

5.9 Respostas aos Problemas Listados no Capítulo 1

Esta seção tem o objetivo de explicitar e sumarizar nossas respostas aos problemas levantados na seção 1.5.3 do capítulo 1. Não repetimos completamente as questões lá colocadas, mas as resumimos de modo a enfatizar as respostas.

1. *Como é resolvido o problema de identificação de objetos móveis ?*

   O valor da política "Id Assignment" escolhida é `USER_ID`. Deste modo, será a aplicação que escolherá o object id, não o POA. Na operação `createMovableObject`, é passado ao contexto de mobilidade o `intialId` que a aplicação cliente desejar. Neste momento, o contexto de mobilidade garante a unicidade local object id, pois se ocorrer duplicidade o objeto não será criado e a aplicação cliente que solicitou a criação do objeto móvel será avisada com a exceção `CannotCreate`. Com as informações dos parâmetros da operação `createMovableObject` o contexto de mobilidade cria a

```
import java.io.*;

public class Serial {

   //-----------------------------------------------------------------
   public static byte [] serialization(java.lang.Object obj){

      ByteArrayOutputStream b = new ByteArrayOutputStream();
      ObjectOutputStream s = new ObjectOutputStream(b);
      s.writeObject(obj);
      s.flush();
      return b.toByteArray();
   }
   //-----------------------------------------------------------------
   public static java.lang.Object deserialization(byte [] param){

      java.lang.Object obj = null;
      ByteArrayInputStream b = new ByteArrayInputStream(param);
      ObjectInputStream s = new ObjectInputStream(b);
      obj = s.readObject();
      return obj;
   }
}
```

Figura 5.7: Implementação da classe Serial

referência inicial do objeto móvel. Quando o objeto móvel for migrar, o servidor de mobilidade destino precisará de um object id para poder instanciar o objeto móvel. Neste momento, o servidor de mobilidade destino usa referência inicial como object id do objeto migrado. A unicidade do object id no destino é também garantida, pois a referência inicial contém o IP e a porta do servidor de mobilidade origem, além do `intialId` escolhido pela aplicação cliente que criou o objeto.

2. *Para migrar um servente é necessário desativá-lo em um servidor e ativá-lo em outro. O que acontece com as requisições que chegarem entre a desativação e a ativação?*

   Toda requisição que chegar depois que a migração foi iniciada, ou seja, depois que o estado do objeto na TSA for "Irá migrar", será respondida com uma mensagem com status `LOCATION_FORWARD`. Assim, a biblioteca do ORB no cliente reenviará a requisição para o servidor de mobilidade home, que por sua vez retornará uma nova resposta com status `LOCATION_FORWARD`, até que a requisição seja finalmente enviada ao servidor que contém o objeto móvel.

3. *O que acontece com as referências para um servente que migrou ?*

   As referências continuam válidas, pois a implementação da infra-estrutura de mobilidade resolve este problema emitindo respostas com status `LOCATION_FORWARD`, as quais são transparentemente tratadas pela biblioteca do ORB no cliente. A aplicação cliente que possuir uma referência CORBA para um objeto móvel pode continuar usando essa referência para enviar requisições para o objeto, independentemente da localização do servente.

4. *O esqueleto também será migrado? Ele tem estado?*

   Como o esqueleto não tem estado, somente o seu código será migrado. É necessário migrar o código do esqueleto porque ele é gerado a partir da interface IDL do objeto móvel, que não existe previamente nas aplicações servidoras. (Esta interface pode até ter sido definida depois que os servidores começaram a rodar).

5. *A implementação da infra-estrutura utiliza o mapa de objetos ativos do POA?*

   Não, o valor da política "Request Processing" empregado é `USE_SERVANT_MANAGER`. Quando se utiliza o mapa de objetos ativos do POA, as APIs do POA não oferecem o nível de controle (sobre o conteúdo do mapa) requerido pela infra-estrutura de migração. Por isso os servidores de mobilidade implementam suas próprias tabelas

(TSA e TOM), que armazenam todas as informações necessárias para a migração de objetos e de certa forma substituem o mapa de objetos ativos.

Capítulo 6

O Protótipo e os Testes Realizados

Este capítulo apresenta o ambiente utilizado no desenvolvimento da infra-estrutura, descreve brevemente os arquivos-fonte do protótipo, relata os resultados dos testes realizados e identifica alguns ítens que poderiam ser objeto de melhoria em versões futuras do protótipo.

6.1 Ambiente de Desenvolvimento

Um resumo do ambiente utilizado encontra-se na tabela 6.1.

Ambiente	**Utilizado**
Sistema Operacional	Linux 2.2 Distribuição Conectiva
JDK	1.2.2 da Sun (build 1.2.2-L, green threads, nojit)
ORBs	ORBacus 4.04 e JacORB 1.0 Beta 15
Servidor HTTP	Apache 1.3.9
Equipamento 1	Pentium III 450 Mhz, 64Mb, 10Gb - Desktop com Linux
Equipamento 2	Pentium III 700 Mhz, 256Mb, 15Gb - Notebook com Windows NT

Tabela 6.1: Ambiente utilizado

Todo o desenvolvimento foi realizado no desktop com Linux. O notebook foi utilizado em rede ponto a ponto com o desktop para testes dos servidores de mobilidade em sistemas operacionais diferentes. Os testes tiveram sucesso e confirmaram que a implementação não usava nada específico do sistema operacional.

O servidor HTTP Apache serviu como o repositório das classes dos objetos móveis. Todos o servidores de mobilidade carregam as classes dos objetos móveis a partir de uma certa URL.

Os ORBs utilizados foram o ORBacus e JacORB. Ambos são baseados em Java e estão disponíveis na Internet para download.

6.2 O Código da Infra-Estrutura

O código da infra-estrutura é composto basicamente de interfaces Java e CORBA, das implementações dessas interfaces, e de classes auxiliares. A relação de arquivos-fonte da infra-estrutura aparece na tabela 6.2.

Arquivos	**Conteúdo**
`Debug.java`	Classe utilitária para mostrar mensagens de depuração.
`JMobilityContext.java`	Definição da interface Java do contexto de mobilidade
`JMovable.java`	Definição da interface Java do OM
`MobilityService.idl`	Definição da interface CORBA do contexto de mobilidade
`MobilityServiceImpl.java`	Implementação do contexto de mobilidade
`Serial.java`	Classe utilitária para serialização e desserialização
`ServantLocatorImpl.java`	Implementação do objeto ServantLocator
`ServantNotActive.java`	Implementação de uma exceção Java
`MobilityServer.java`	Implementação do servidor de mobilidade
`TOMObject.java`	Classe que define um elemento da tabela TOM
`TSA.java`	Classe utilitária que implementa a tabela TSA
`TSAObject.java`	Classe que define um elemento da TSA

Tabela 6.2: Arquivos da infra-estrutura

Os arquivos mais importantes da implementação são os seguintes:

- O contexto de mobilidade, `MobilityServiceImpl.java`.
- A implementação do Servant Locator, `ServantLocatorImpl.java`.
- O servidor de mobilidade, `MobilityServer.java`.

Os demais arquivos são arquivos auxiliares ou contém definições de interfaces Java.

6.3 Testes Realizados

O código dos objetos móveis que empregamos nos testes foi adaptado do exemplo `Grid` disponibilizado por diversos ORBs. Utilizamos objetos móveis do tipo `Grid` para testar toda a funcionalidade do servidor de mobilidade.

Desenvolvemos dois clientes de objetos `Grid`. Um deles utiliza apenas os métodos da interface `Grid` original (`set` e `get`, dada uma coordenada), o outro somente chama o método `move` sobre um objeto `Grid`, fazendo este se deslocar. A tabela 6.3 mostra os arquivos-fonte dos testes.

Arquivos	**Conteúdo**
Clientes	
`Client.java`	Implementação do cliente que utiliza `set` e `get`
`ClientSM.java`	Implementação do cliente que utiliza `move`
Objeto Móvel	
`GridImpl.java`	Implementação do objeto grid
`Grid.idl`	Definição da interface CORBA do objeto grid

Tabela 6.3: Arquivos do grupo cliente e objeto móvel

O ORB utilizado em todo desenvolvimento e na maior parte dos testes foi o JacORB. Para testar a migração de objetos entre servidores de mobilidade implementados com diferentes ORBs, utilizamos também o ORBacus. Os testes com JacORB e ORBacus tiveram sucesso, isto é, conseguimos deslocar objetos móveis de um servidor de mobilidade implementado com o JacORB para outro implementado com o ORBacus, e vice-versa.

Outro teste interessante foi feito com o código dos clientes alterado para que eles ficassem em "looping". Assim, um dos clientes ficava continuamente executando chamadas `set`, com diferentes valores, seguidas de chamadas `get` para inspecionar os valores atuais no `Grid`. O segundo cliente ficava efetuando sucessivos deslocamentos do objeto `Grid` de um servidor de mobilidade para outro. Esses clientes foram executados simultaneamente e a infra-estrutura funcionou sem problemas.

6.4 Ítens a Melhorar

Apesar do protótipo já ser utilizável, identificamos alguns ítens que gostaríamos de melhorar:

Desenvolver uma interface gráfica para os servidores de mobilidade. Com uma interface gráfica o administrador do ambiente poderia facilmente visualizar e manipular cada servidor de mobilidade. Esta interface gráfica poderia apresentar dados sobre os servidores de mobilidade, tais como número de objetos residentes em cada servidor, número de objetos criados em cada servidor e nível de utilização do servidor (número de requisições CORBA recebidas por ele). Poderia também apresentar informações específicas de um certo objeto móvel, como a sua identificação, o seu servidor home e o seu horário de criação. Esta interface gráfica poderia ainda disponibilizar uma facilidade de "drag and drop" de objetos móveis, a qual efetuaria chamadas `move` sobre os objetos manipulados.

Tornar persistente a TOM. Atualmente o servidor de mobilidade não armazena de modo persistente o conteúdo dessa tabela.

Gerenciamento da configuração do class loader. Atualmente o servidor de mobilidade, ao iniciar a sua execução, se configura com uma lista das URLs onde ele deve buscar os bytecodes das classes dos objetos móveis. Assim, se for preciso alterar esta lista, temos de reiniciar todos os servidores de mobilidade. Seria interessante possibilitar que os servidores de mobilidade sejam reconfigurados sem que seja necessário reiniciá-los.

Método `move` com parâmetro de timeout. O valor do timeout de migração, atualmente um parâmetro de configuração do servidor de mobilidade, é o mesmo para todas as solicitações de migração de objetos residentes nesse servidor. Seria simples oferecer uma variação do método `move` que recebe como parâmetro o valor do timeout de migração.

Capítulo 7

Considerações Finais

As facilidades oferecidas por CORBA nos ajudaram tanto no projeto da infra-estrutura como na implementação do protótipo. No capítulo 1 descrevemos, com algum detalhe, os recursos de CORBA que utilizamos: IORs, o mecanismo de *location forward* do protocolo GIOP/IIOP, o POA, suas políticas, e o uso de um *servant locator*. O POA Manager é um recurso que descrevemos, mas não utilizamos, por não ser possível enfileirar num POA Manager somente as requisições para um dado servente. Se o POA Manager permitisse tal "enfileiramento seletivo", ele seria um recurso muito interessante para uma infra-estrutura de migração de objetos CORBA.

A utilização de Java foi também decisiva para este trabalho. Mobilidade de código, serialização e reflexão foram características do ambiente Java que usamos fortemente e com muita facilidade. A única restrição percebida foi a impossibilidade de se congelar e migrar uma thread de execução. Essa restrição não chegou a ser um empecilho para nós, pois migração de threads não era um de nossos objetivos.

Dentre os trabalhos relacionados que encontramos na literatura e sumarizamos no capítulo 3, o Voyager é o que mais se aproxima da infra-estrutura aqui proposta, por ser também baseado em CORBA. A última seção do capítulo 3 aborda a especificação Life Cycle do OMG e mostra como nosso trabalho dá respostas a algumas das críticas que essa especificação tem sofrido. Não tivemos a pretensão de responder a todas as críticas que tem sido feitas à especificação Life Cycle, mas apresentamos algumas alternativas e apontamos possíveis soluções.

O entendimento da infra-estrutura de migração deve começar pela visão do usuário que desenvolve objetos móveis e continuar pelo funcionamento interno. No capítulo 4, voltado

para o desenvolvedor de objetos móveis, descrevemos todas as interfaces que necessitam de implementação e fornecemos um roteiro de como desenvolver um objeto móvel. No capítulo 5, dedicado ao funcionamento interno da infraestrutura, descrevemos em detalhe o protocolo de migração, as estruturas de dados que ele emprega, e como ele utiliza referências, bem como a implementação do `ServantLocator` existente nos servidores de mobilidade.

7.1 Comparação com Outros Trabalhos

Esta seção apresenta uma comparação entre nossa infra-estrutura de migração e os sistemas Voyager, Aglets e Jumping Beans descritos no capítulo 3. Esses três sistemas empregam Java na sua implementação. Eles também utilizam fortemente as facilidades da linguagem Java que foram cruciais para nossa infra-estrutura: serialização, reflexão e mobilidade de código.

Uma diferença importante entre este trabalho e os sistemas Aglets e Jumping Beans é a utilização de CORBA, que padroniza a comunicação e fornece interoperabilidade entre linguagens de programação. Deste modo, nossa infra-estrutura permite o uso de aplicações clientes escritas em uma linguagem diferente de Java, as quais empregam objetos móveis implementados em Java. Essa possibilidade não existe nos sistemas Aglets e Jumping Beans.

Além disto, o Jumping Beans tem uma arquitetura centralizada, isto é, toda migração deve passar por um servidor Jumping Beans para chegar a seu destino. Isso não ocorre na infra-estrutura proposta neste trabalho. Os servidores de mobilidade desta infra-estrutura são independentes e, a cada migração, são utilizados somente os servidores origem, destino e home.

O sistema Voyager, também baseado em CORBA, oferece funcionalidade bastante semelhante à de nossa infra-estrutura. Entretanto, ele só permite migrações de objetos entre servidores implementados com o ORB Voyager. Já a infra-estrutura proposta neste trabalho permite migrar objetos entre servidores implementados com quaisquer ORBs Java, deste que esses ORBs suportem o POA. Deste modo, temos uma independência da implementação do ORB.

7.2 A Contribuição deste Trabalho

A principal contribuição deste trabalho é a apresentação de uma infra-estrutura de migração de objetos CORBA implementados em Java, além do desenvolvimento de um protótipo desta infra-estrutura. Este protótipo nos ofereceu tanto um cenário de validação dos conceitos envolvidos como uma ferramenta para experimentação com migração de objetos, com todos os seus desafios e dificuldades. Um dos testes que efetuamos com sucesso envolveu a migração de objetos entre servidores de mobilidade implementados com diferentes ORBs (JacORB e ORBacus).

Nossa infra-estrutura assegura que a migração de objetos é escalável com respeito ao número de migrações efetuadas. Para cada objeto móvel, as informações mantidas pela infra-estrutura se limitam a duas entradas em tabelas, independentemente de quantas vezes o objeto migrou e de por onde ele migrou. Uma dessas entradas fica na Tabela de Objetos Móveis do servidor no qual o objeto foi criado, a outra fica na Tabela de Serventes Ativos do servidor onde o objeto se encontra no momento. A migração de um objeto móvel "não deixa rastros", ou seja, nenhuma informação sobre o objeto é mantida nos demais servidores por onde ele passou.

Os objetivos colocados na introdução, *migração de objetos individuais* e *transparência de migração*, foram inteiramente alcançados pelo protótipo. É importante observar que a migração de objetos individuais, oferecida pela infra-estrutura que implementamos, não impede a migração de servidores CORBA, viabilizada pelos repositórios de implementações integrados a diversos produtos CORBA. A migração de objetos individuais e a migração de servidores são conceitos ortogonais. Em outras palavras: o uso de um repositório de implementações permite que nossos servidores de mobilidade sejam deslocados de uma máquina para outra, sem prejuízo da migração de objetos individuais de um processo servidor de mobilidade para outro, a qual continuará funcionando normalmente.

7.3 Trabalhos Futuros

Encerramos este trabalho relacionando alguns ítens para investigação futura:

Possibilidade dos objetos móveis utilizarem POAs com outras políticas. O servidor de mobilidade coloca todos os objetos móveis sob um POA criado com as políticas definidas na tabela 5.1 Seria interessante averiguar que outras combinações de po-

líticas poderiam ser adequadas para objetos móveis. O objetivo seria permitir que, ao solicitar a criação de um objeto móvel, o cliente possa escolher um POA com as políticas que mais lhe interessem.

Contextos de mobilidade com propriedades diferentes. Um mesmo servidor de mobilidade poderia ter contextos de mobilidade com propriedades diferentes. Assim, o cliente que solicitar a criação de um objeto móvel poderia escolher o contexto que melhor atenda às suas necessidades.

Integração da infra-estrutura com o serviço de segurança CORBA. Isto permitiria definir e implementar um esquema de controle de acesso aos objetos móveis e aos contextos de mobilidade.

Referências Bibliográficas

[1] J. Baumann, F. Hohl, K. Rothermel, and M. Straber. Mole - concept of a mobile agent system. *World Wide Web*, 1(3):123–137, September 1998.

[2] Gerald Brose. Site of jacorb. http://www.jacorb.org.

[3] Patrick Chan and Stephen E. Ingram. *Developing Professional Java Applets.* The Java Serie. Macmillan Computer Publishing, book and cd edition, June 1996.

[4] Dilma Menezes da Silva and Marco Dimas Gubitoso. Sistemas de informação distribuídos para agentes móveis. In *SEMISH 98*, 1998.

[5] G. Glass. Objectspace voyager core package technical overview. In *Mobility - Processes, Computers an Agents*, pages 611–627. Addison Wesley, 1999.

[6] Germán Goldszmidt and Yechiam Yemini. Distributed management by delegation. In *Proceedings of the 15th International Conference on Distributed Computer System*, June 1995.

[7] Mark Grand. *Java, Language Reference.* The Java Serie. O'Reill, 1997.

[8] M. Henning and S. Vinoski. *Advanced CORBA Programming with C++.* Addison-Wesley, 1999.

[9] David Kotz, Robert Gray, Saurab Nog, Daniela Rus, Sumit Chawla, and George Cybenko. Agent tcl: Targeting the needs of mobile computers. *IEEE Internet Computing*, 1(4):58–67, July/August 1997.

[10] Danny B. Lange and Mitsuru Oshima. *Programming and Deploying Mobile Agents with Java Aglets.* Addison Wesley, 1st edition, September 1998.

[11] Dejan Milojicic, Markus Breugst, Ingo Busse, John Campbell, Stefan Covaci, Barry Friedman, Kazuya Kosaka, Danny Lange, Kouichi Ono, Mitsuru Oshima, Cynthia Tham, Sankar Virdhagriswaran, and Jim White. Masif, the omg mobile agent system interoperability facility. In *Proceeding of the Second International Workshop on Mobile Agents*, pages 50–67, September 1998.

[12] Dejan S. Milojicic, William LaForge, and Deepika Chauhan. Mobile objects and agents (moa), design, implementation and lessons learned. In *Proceedings of the 4th USENIX Conference on Object Oriented Technologies (COOTS)*, pages 179–194, April 1998.

[13] Jeff Nelson. *Programming Mobile Objects with Java.* Wiley Computer Publishing, 1st edition, 1999.

[14] Object Management Group, http://cgi.omg.org/cgi-bin/doc?orbos/97-05-01. *Specification of the Portable Object Adapter (POA)*, May 1997.

[15] Object Management Group, http://cgi.omg.org/cgi-bin/doc?formal/98-07-05. *CORBAServices: Common Object Services Specification*, Jul 1998.

[16] Object Management Group, http://cgi.omg.org/cgi-bin/doc?formal/00-06-41. *Object Management Architecture (OMA) Guide*, Jun 2000.

[17] Object Management Group, http://cgi.omg.org/cgi-bin/doc?formal/2000-05-02. *Workflow Management Specification, v1.2*, May 2000.

[18] Object Management Group, http://cgi.omg.org/cgi-bin/doc?formal/2001-02-01. *CORBA/IIOP Specification, v2.4.2*, Feb 2001.

[19] Object Management Group, http://cgi.omg.org/cgi-bin/doc?orbos/01-02-04. *Load Balancing and Performance Monitoring for CORBA Application draft RPF*, Feb 2001.

[20] Holger Peine and Torsten Stolpmann. The architecture of the ara platform for mobile agents. In *Proceedings of the First International Workshop on Mobile Agent (MA'97)*, pages 50–61, Berline, Springer Verlag, April 1997.

[21] Kimmo Raatikainen. Service machine development for an open long-term mobile and fixed network environment. Technical Report telecom/98-08-08, Object Management Group, August 1998.

[22] Robert Richardson. Taking a flying leap. http://www.networkmagazine.com, December 1999.

[23] D. Schmidt and S. Vinoski. Object adapter: Concepts and terminology. *C++ Report*, 11, October 1997.

[24] Steve Vinoski. Corba: Integrating diverse applications within distributed heterogeneous environments. *IEEE*, 1996.

[25] Tom Walsh, Noemi Paciorek, and David Wong. Security and reliability in concordia. In *Proceedings of the 31st Hawaii International Conference on Systems Sciences*, volume VII, pages 44–53, January 1998.

[26] James E. White. Telescript technology: Mobile agents. *Software Agents, AAAI/MIT Press*, 1996. General Magic White Paper.