Arquiteturas de Componentização de Servidores como Integradoras de Bancos de Dados Distribuídos

Myrthes Cavalcante de Aguiar

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 — Abril de 2002

Arquiteturas de Componentização de Servidores como Integradoras de Bancos de Dados Distribuídos

Este exemplar corresponde à redação
final da dissertação devidamente corrigida
e defendida por Myrthes Cavalcante de Aguiar
e aprovada pela comissão julgadora.

São Paulo, 10 de abril 2002.

Banca examinadora:

- Prof. Dr. Francisco Reverbel (orientador) — IME-USP
- Prof. Dr. Marcos Ribeiro Pereira Barretto — Escola Politécnica da USP
- Prof. Dr. João Eduardo Ferreira — IME-USP

Agradecimentos

Ao meu orientador, Prof. Dr. Francisco Reverbel, que me acolheu como aluna oferecendo não simplesmente um trabalho norteado, onde as diretrizes são pré-definidas, mas uma verdadeira parceria na pesquisa por tecnologias de interesse comum. Agradeço imensamente sua generosidade em compartilhar conhecimentos, discutir idéias e soluções, e sua dedicação ao trabalho de formar seu aluno.

Aos meus pais que incentivaram incondicionalmente todas as minhas iniciativas, depositando em mim a confiança necessária para enfrentar qualquer dificuldade, e me dando o prazer de compartilhar com eles a alegria de minhas conquistas.

A minha querida tia Zilce (*in mem.*), minha segunda mãe, que esteve presente em todas as minhas conquistas e que sempre me ofertou todo suporte necessário para que pudesse atingir meus objetivos.

Ao meu marido Natan e ao meu filho Daniel, que souberam generosamente ceder espaço em nossa vida em comum gerando as condições necessárias para que eu tivesse tranquilidade para concluir este trabalho.

Resumo

Esta dissertação apresenta uma proposta para a integração de bancos de dados distribuídos através da utilização de arquiteturas de componentização de servidores. Foram analisadas as arquiteturas *Enterprise JavaBeans* (EJB), especificada pela Sun Microsystems e *CORBA Component Model* (CCM), especificada pelo OMG. Tais arquiteturas oferecem uma infraestrutura capaz de gerenciar automaticamente distribuição de objetos, persistência, transações e segurança.

Este trabalho propõe uma solução simplificada para o problema de integração de bancos de dados distribuídos utilizando as funcionalidades destas arquiteturas, sem abordar aspectos de otimização de consultas distribuídas. A solução apresentada restringe-se à integração de bancos de dados relativos a um mesmo tipo de aplicação e onde estão armazenadas informações referentes a um mesmo domínio. O objetivo é disponibilizar uma visão comum de um subconjunto pré-estabelecido das informações distribuídas por vários bancos de dados, e não uma mesclagem completa de todos os esquemas participantes.

Com a finalidade de viabilizar a análise da solução proposta, foi desenvolvida, como estudo de caso, uma aplicação de controle de estoque distribuído. A arquitetura escolhida para o desenvolvimento desta aplicação foi a EJB, pelo grau de evolução em que ela se encontra e pela disponibilidade de diversos servidores que implementam a especificação definida pela Sun Microsystems.

Abstract

This dissertation presents a proposal for integrating distributed databases using server component architectures. The Enterprise JavaBeans architecture (EJB), specified by Sun Microsystems, and the CORBA Component Model architecture (CCM), specified by the OMG were analyzed. Such architectures offer an infrastructure that can automatically manage object distribution, persistence, transactions, and security.

This work suggests a simplified solution for the integration problem of distributed databases using the features of such architectures, without addressing issues related to distributed query optimization. The solution presented is restricted to the integration of data bases for applications of the same kind, in which are stored information relative to the same application domain. The goal is to present a common vision of a predefined subset of information distributed across a number of databases and not an entire merge of all participating schemas.

In order to make possible an analysis of the suggested solution, an application for distributed inventory control was developed as a proof of concept. The EJB architecture was chosen for the development of this application, due to its present degree of evolution and due to the availability of several servers that implement the specification defined by Sun Microsystems.

Sumário

Lista de Figuras

Introdução

Uma característica freqüentemente encontrada nos sistemas de informação atualmente é a distribuição de dados entre repositórios autônomos e heterogêneos. Plataformas de gerenciamento de objetos distribuídos têm sido utilizadas para construir sistemas que trabalham com bases de dados distribuídas e heterogêneas. As capacidades e funcionalidades de tais plataformas devem ser levadas em conta no projeto de arquitetura de sistemas multibase de dados [41].

A fim de construir aplicações distribuídas escaláveis, desenvolvedores precisam integrar suas lógicas negociais numa arquitetura que inclui, no mínimo, serviços de transações, persistência, eventos e nomes. Além disso, precisam ser capazes de ajustar suas aplicações e possuir modelos flexíveis de implantação. Modelar, projetar e implantar tais aplicações é uma tarefa complexa.

Devido ao esforço necessário, cada vez maior, para se desenvolver desde o início e manter uma plataforma de gerenciamento de objetos distribuídos, a demanda de *middleware* para *Distributed Object Computing* (DOC), tal como *Common Object Request Broker Architecture* (CORBA) [22] do OMG[1], vem aumentando. CORBA permite aos clientes invocar operações em objetos distribuídos sem se preocupar com a localização do objeto, a linguagem de programação, a plataforma de sistema operacional, os protocolos de comunicação e o hardware utilizados. Outros padrões de *middleware* para objetos distribuídos disponíveis são:

- *Microsoft Distributed Component Object Model* (DCOM) [5]
- *Java Remote Method Invocation* (RMI) [38]

[1] *Object Management Group*: consórcio formado por aproximadamente 800 empresas que produz e mantém um conjunto de especificações que suportam projetos de desenvolvimento de aplicações e sistemas de *software* distribuídos e heterogêneos, desde a análise e projeto até a codificação, implantação, *runtime* e manutenção.

O uso de CORBA como uma infraestrutura flexível para aplicações distribuídas do tipo cliente/servidor cresceu rapidamente durante os cinco últimos anos [30]. Historicamente, a especificação CORBA tem se concentrado na definição de interfaces, as quais definem contratos entre clientes e servidores. Uma interface define como clientes vêem e utilizam serviços de objetos fornecidos por um servidor. Este modelo oferece como vantagem a transparência de localização, porém não padroniza as interações entre serventes e ORBs (*Object Request Brokers*) [36]. Os desenvolvedores continuam responsáveis pela definição de como os serventes são implementados e como interagem com o ORB. Isto resulta em implementações *ad-hoc* que aumentam a complexidade das atualizações de *software*, reduzindo a reusabilidade e flexibilidade de sistemas baseados em CORBA. Observações similares valem para os outros padrões de *middleware* para objetos distribuídos, Java RMI e DCOM.

Para suprir essas limitações, foram definidas arquiteturas de componentização de servidores[2], como:

- *CORBA Component Model* (CCM) [37]
- *Enterprise JavaBeans* (EJB) [21]
- *Microsoft Transaction Server* (MTS) [13, 20]

Cada uma dessas arquiteturas é fundamentada num padrão de *middleware* para objetos distribuídos e define um "servidor de aplicação" que disponibiliza um *container* no qual são implantados componentes. O desenvolvedor de aplicação não precisa mais escrever um servidor completo. Ele desenvolve e interliga componentes, eventualmente reutilizando componentes pré-existentes, deixando a cargo do servidor de aplicação questões como gerenciamento de *threads*, transações e segurança, entre outras. As interações entre os componentes e o *container* são padronizadas pela arquitetura de componentização de servidores.

O modelo de componentes *Enterprise JavaBeans*, baseado em Java RMI, fornece uma abstração para monitores de transação de componentes que representam a convergência

[2]Neste trabalho usamos o neologismo "componentização" e optamos pela denominação "arquiteturas de componentização de servidores" ao invés de "arquiteturas de servidores baseados em componentes". Empregaremos ainda dois outros neologismos de uso comum na área: serializável (*serializable*) e negocial (em "método negocial", do inglês *business method*). Além disso, manteremos em inglês certos termos técnicos para os quais não há tradução consagrada: *middleware*, *container*, *thread*, *handle* e *home*, dentre outros.

entre duas tecnologias: a empregada nos monitores de processamento de transações tradicionais e a tecnologia de objetos distribuídos.

O OMG definiu o CCM, um modelo que, embora baseado em CORBA, foi fortemente influenciado pelo EJB. O CCM pode ser visto como uma generalização das primeiras versões do EJB. Por se tratar de um padrão bastante recente, só agora começam a surgir suas primeiras implementações, ainda incompletas.

A solução MTS, proposta pela Microsoft, usa um modelo de componentes servidores e um serviço distribuído de componentes baseado em DCOM. Apesar das facilidades que oferece aos desenvolvedores, o MTS é uma solução proprietária restrita à ambientes Microsoft e por este motivo não será abordado neste trabalho.

As tecnologias de componentização de servidores permitem criar objetos que correspondem a dados armazenados em bancos de dados. Num cenário que envolva bancos de dados distribuídos e heterogêneos pode-se, com estas tecnologias, definir uma visão global, através de interfaces, de modo a possibilitar uma integração entre os vários bancos de dados distribuídos. O acesso a objetos da base de dados através de interfaces não requer conhecimento do esquema da base de dados: alterações no esquema são transparentes aos clientes. Interfaces podem ser definidas para expor somente ítens de dados que alguns clientes tem permissão para ler e atualizar. Interfaces podem desempenhar um papel análogo às visões relacionais, tanto em relação a independência dos dados, como em relação a autorização [25].

Tanto CCM como EJB permitem que referências a objetos sejam convertidas para um formato adequado para armazenamento persistente[3]. Os objetos implementados por um servidor podem conter referências persistentes para objetos implementados por outros servidores. Isto sugere a construção de um banco de dados distribuído e heterogêneo, cujos sistemas participantes são interconectados através de *middleware* de componentização de servidores.

O objetivo deste trabalho é estudar o uso das arquiteturas EJB e CCM como integradoras de bancos de dados distribuídos e heterogêneos, enfocando os aspectos de gerenciamento de persistência realizado pelo *container* do componente. Embora a questão da integração de bancos de dados distribuídos e heterogêneos tenha sido bastante estudada [1, 6, 7, 18, 26, 27, 32, 33, 40, 42], ela envolve uma série de dificuldades práticas que

[3]No caso do CCM pode-se converter *Interoperable Object References* (IORs) para *strings*. No caso do EJB pode-se empregar *entity handles*, que são serializáveis.

tornam não trivial a aplicação das soluções propostas na literatura. Mostraremos que arquiteturas como EJB e CCM facilitam o processo de integração de dados distribuídos em dois momentos:

- automatizando a geração do código de objetos que envelopam dados dos diferentes repositórios;
- liberando o desenvolvedor de tarefas relativas ao gerenciamento de transações, persistência, distribuição, segurança e ciclo de vida.

Usamos como estudo de caso uma "aplicação exemplo" implementada e testada com *middleware* EJB. Embora nossa abordagem seja também aplicável ao CCM, a inexistência de implementações do CCM inviabilizou a realização de trabalhos práticos com esta arquitetura.

Esta dissertação está organizada da seguinte maneira:

- O capítulo 1 apresenta a arquitetura *Enterprise JavaBeans*. A implementação de componentes EJB é exemplificada por meio de uma aplicação simples de controle de estoque.
- No capítulo 2 é apresentada uma visão geral da arquitetura de componentes CORBA proposta pelo OMG.
- O capítulo 3 descreve o problema da integração de bancos de dados distribuídos e heterogêneos.
- O capítulo 4 propõe que esse problema seja abordado com o uso de arquiteturas de componentização de servidores.
- O capítulo 5 apresenta, como estudo de caso, uma aplicação de controle de estoque distribuído. A aplicação exemplo do capítulo 1 é retomada, sendo agora levados em conta os aspectos de distribuição.
- O capítulo 6 descreve os trabalhos relacionados que encontramos na literatura.
- O capítulo 7 traz nossas considerações finais. São relatadas as dificuldades que encontramos, nosso trabalho é comparado com outros relacionados e, finalmente, são apresentadas algumas linhas para investigação futura.

Capítulo 1

Enterprise JavaBeans

1.1 Visão Geral

A arquitetura *Enterprise JavaBeans* (EJB) [29] é uma arquitetura de componentes para o desenvolvimento e implantação de aplicações distribuídas orientadas a objeto.

A tecnologia de Monitores de Transações de Componentes [21] é o sustentáculo da arquitetura EJB. O termo Monitor de Transações de Componentes (*Component Transaction Monitor*, ou CTM) descreve os mais sofisticados servidores de aplicação de objetos distribuídos [31]. CTMs evoluíram como um híbrido entre os monitores de processamento de transações tradicionais [9], como o CICS da IBM e o Tuxedo da BEA, e as tecnologias de *Object Request Brokers*. Eles fornecem uma infraestrutura que pode automaticamente gerenciar transações, distribuição de objetos, concorrência, segurança e recursos. São capazes de lidar com ampla população de usuários e com trabalhos de missão crítica.

Enterprise JavaBeans é um modelo padrão de componentes servidores para CTMs, os quais são baseados em tecnologias de objetos distribuídos. Por sua vez, sistemas de objetos distribuídos são um fundamento natural para arquiteturas de três ou mais camadas (*three-tier* ou *n-tier*). A figura 1.1 mostra uma arquitetura *three-tier*: a lógica de apresentação reside no cliente (1ª camada), a lógica negocial na camada do meio (*middle tier*), e outros recursos, como bases de dados, residem na 3ª camada (*backend*).

A arquitetura EJB suporta implicitamente os seguintes serviços:

- **Ciclo de Vida** — Alocação de processo, gerenciamento de *threads*, ativação ou destruição de objeto

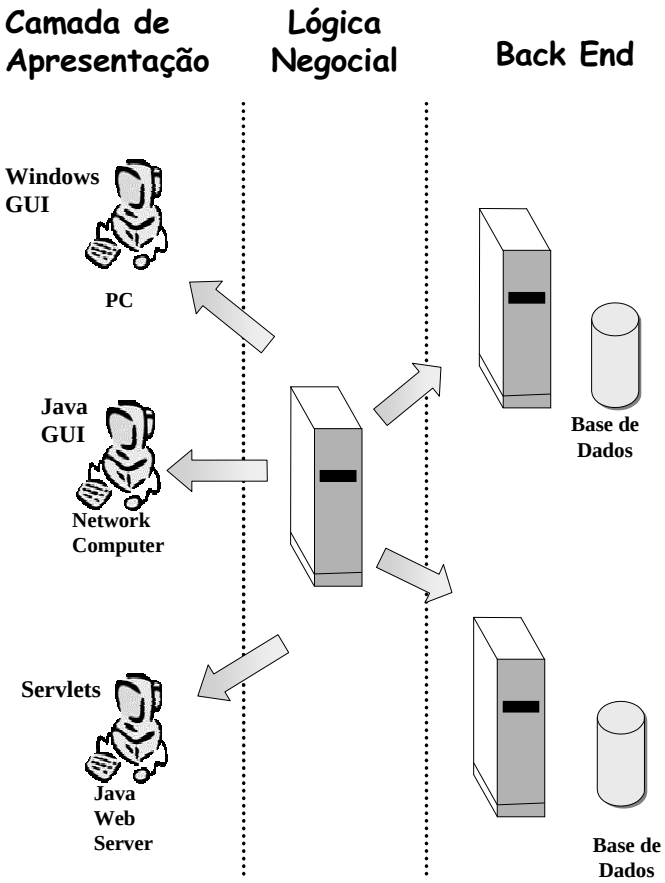

Figura 1.1: Arquitetura Three-Tier.

- **Gerenciamento de estado** — Salvar e restaurar o estado conversacional do objeto entre as chamadas de métodos
- **Segurança** — Autenticação de usuários e checagem de níveis de acesso
- **Transações** — Demarcação, gerenciamento do início, desenrolar, efetivação (*commit*) e cancelamento (*rollback*) de transações
- **Persistência** — Armazenar e recuperar objetos persistentes de uma base de dados

A arquitetura EJB estabelece papéis e prevê especificações para:

- Servidor EJB

- *Container* EJB
- Componente EJB

1.2 Papéis EJB

A arquitetura EJB define seis papéis distintos no ciclo de vida do desenvolvimento e da implantação. São eles:

- **Fornecedor de componente EJB** — É o produtor de componentes. Gera uma arquivo ejb-jar contendo um ou mais componentes. É responsável pela definição das interfaces *home* e *remote* do componente e pelas classes Java que implementam os seus métodos negociais. É também responsável pelo descritor de implantação, que inclui informações estruturais do componente, como tipo e nome do componente, e declara todas as dependências externas, como tipos e nomes de recursos que o componente usa.

  O fornecedor de um componente é tipicamente um especialista do domínio da aplicação e não necessariamente um especialista em programação em nível de sistema. Ele deixa a cargo do *container* o gerenciamento de serviços como transações, segurança e concorrência.

- **Montador da aplicação** — Combina os componentes em unidades de aplicação implantáveis. Recebe os arquivos ejb-jar produzidos pelo fornecedor de componentes e gera um ou mais arquivos ejb-jar contendo os componentes com suas instruções de montagem da aplicação. A instrução da montagem da aplicação é inserida nos descritores de implantação. O montador da aplicação pode também combinar componentes EJB com outros tipos de componentes de aplicação como Java ServerPages para compor uma aplicação.

  O montador de uma aplicação é tipicamente um especialista de domínio que compõe aplicações que usam componentes EJB. Trabalha com o descritor de implantação e, embora deva conhecer a funcionalidade fornecida pelas interfaces do componente, não precisa conhecer suas implementações.

- **Implantador** — A partir de um ou mais arquivos ejb-jar produzidos pelo fornecedor de componente ou pelo montador da aplicação, implanta num ambiente operacio-

nal os componentes contidos nestes arquivos. Este ambiente operacional inclui um servidor EJB e um *container* EJB.

O implantador deve resolver todas as dependências externas declaradas pelo fornecedor de componente e deve seguir as instruções de montagem da aplicação definidas pelo montador. Ele é um especialista num ambiente operacional específico.

O processo de implantação é realizado tipicamente em dois estágios:

- Geração das classes e interfaces adicionais que habilitam o *container* a gerenciar componentes EJB em tempo de execução;
- Execução da instalação dos componentes e das classes e interfaces adicionais no *container* EJB.

- **Fornecedor do servidor EJB** — É um especialista na área de gerenciamento de transações distribuídas, objetos distribuídos e outros serviços de sistema. O típico fornecedor de servidor EJB é uma empresa fabricante de algum sistema operacional, *middleware* ou banco de dados.

- **Fornecedor do *container* EJB** — É responsável por fornecer as ferramentas necessárias para a implantação de componentes EJB, além de um ambiente de execução para as instâncias dos componentes.

  Da perspectiva do componente, o *container* é parte do ambiente operacional. O ambiente de execução do *container* fornece gerenciamento de transações e segurança, possibilita o acesso aos componentes por clientes locais e remotos e faz gerenciamento escalável de recursos, entre outros serviços geralmente requeridos como parte de uma plataforma servidora.

  O foco do fornecedor de *container* é o desenvolvimento de um *container* seguro, escalável e com apoio para transações, que esteja integrado com um servidor EJB. O fornecedor de *container* isola o componente EJB das especificidades do servidor EJB através do fornecimento de uma API padrão entre o *container* e o componente. Esta API é o contrato de especificação do componente EJB.

- **Administrador do sistema** — É responsável pela configuração e administração da computação dos componentes e da infraestrutura de rede que inclui o servidor e *container* EJB.

Cada um desses papéis pode ser desempenhado por um participante diferente, ou em alguns cenários um único participante poderá desempenhar vários papéis. Por exemplo, o fornecedor do servidor EJB tipicamente também é o fornecedor do *container* EJB. Um único programador pode exercer os papéis de fornecedor de componente e montador da aplicação.

1.3 Servidor EJB

É um servidor de aplicação genérico que fornece um ambiente de apoio à arquitetura EJB.

Fornece um ou mais *container*s para os componentes EJB nele implantados. É responsável pelo gerenciamento e coordenação da alocação de recursos como *threads*, processos, memória e conexões com bancos de dados, bem como por fornecer acesso a um conjunto padrão de serviços, tais como transações, serviços de nomes, de diretório, de segurança e de persistência.

1.4 *Container* EJB

O *container* EJB (figura 1.2) fornece processo ou *thread* para a execução de componentes EJB. É responsável por:

- Fornecer contexto de execução e contexto transacional aos componentes
- Registrar o componente no serviço de nomes e diretórios
- Fornecer interface remota do componente, criar e destruir instâncias
- Gerenciar transações, estado e persistência

1.5 Componente EJB

Interface *home* — A interface *home* define os métodos de ciclo de vida do componente: métodos para criação de novas instâncias do componente, remoção de instâncias do componente e busca de instâncias. Por meio desta interface, clientes vêem um componente EJB como uma coleção homogênea de instâncias, com operações para inclusão, remoção e busca de instância. A interface *home* estende `javax.ejb.EJBHome`,

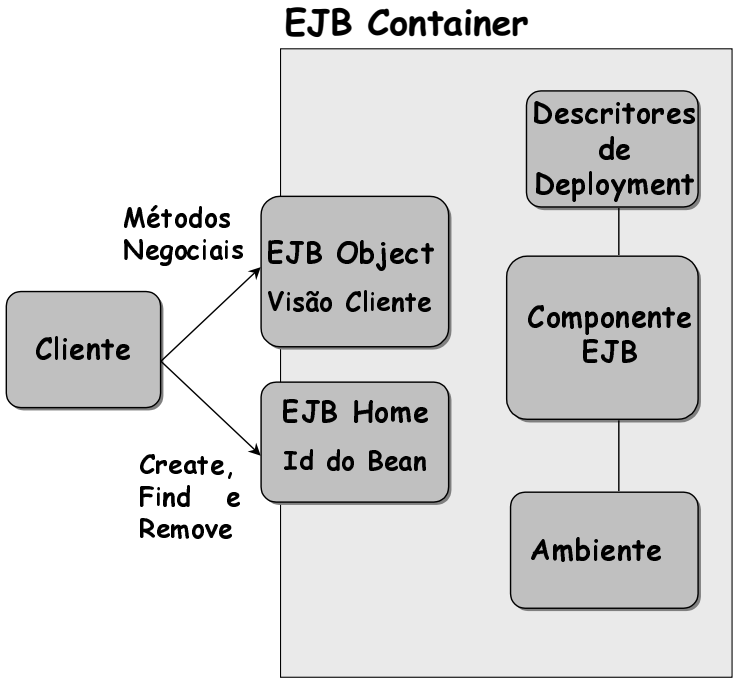

Figura 1.2: *Container* Enterprise JavaBeans.

que por sua vez estende `java.rmi.Remote`. A figura 1.3 mostra a interface `EJBHome` do pacote `javax.ejb`. Este pacote Java contém todas as interfaces e classes padronizadas pela especificação EJB.

Interface *remote* — A interface *remote* define os métodos negociais do componente. Ela estende `java.ejb.EJBObject`, que por sua vez estende `java.rmi.Remote`. A interface *remote* representa a visão que o cliente terá de uma instância do componente EJB. Expõe as interfaces relacionadas à aplicação, mas não as usadas pelo *container* para gerenciar e controlar instâncias do componente. A figura 1.4 mostra a interface `EJBObject`.

Classe do componente — Esta classe implementa os métodos negociais do componente. Normalmente, a classe do componente não implementa as interfaces *home* e *remote*. Entretanto, deve possuir métodos de mesmo nome dos definidos na interface *remote*, e métodos correspondendo a alguns dos métodos da interface *home*. Os clientes de um componente nunca chamam diretamente métodos da classe do componente. Toda interação entre um cliente e o componente é feita por meio dos métodos das interfaces *home* e *remote*.

```
package javax.ejb;

import java.rmi.RemoteException;

public interface EJBHome extends java.rmi.Remote {

    //Obtém a interface MetaData para o componente.
    public EJBMetaData getEJBMetaData() throws RemoteException;

    //Obtém um handle para o objeto home remoto.
    public HomeHandle getHomeHandle() throws RemoteException;

    //Remove um objeto EJB identificado por sua chave primária.
    public void remove(Object primaryKey) throws RemoteException;

    //Remove um objeto EJB identificado por seu handle.
    public void remove(Handle handle) throws RemoteException;

}
```

Figura 1.3: A interface `EJBHome`.

Além dos métodos da interface padrão `javax.ejb.EJBHome`, uma interface *home* também inclui métodos especiais de criação e busca do componente. Os métodos `create` e `find` são específicos do componente, portanto cabe ao desenvolvedor do componente declará-los apropriadamente na interface *home*.

Os métodos do tipo `create` lançam uma exceção `CreateException` se ocorrer algum erro durante o processo de criação. Os métodos do tipo `find` lançam uma exceção `FinderException` se o componente solicitado não puder ser localizado. Além dessas exceções, estes métodos também lançam a exceção `RemoteException`.

Componentes EJB são implantados num *container* EJB. Em tempo de implantação, o *container* automaticamente gera classes que implementam as interfaces *home* e *remote* do componente e cria uma instância da classe *home*. A classe *remote* será instanciada sempre que forem criadas instâncias do componente. As implementações das interfaces *home* e *remote* delegam à classe do componente as chamadas de métodos feitas por seus clientes.

```
package javax.ejb;

import java.rmi.RemoteException;

public interface EJBObject extends java.rmi.Remote {

    //Obtém a interface home do componente.
    public EJBHome getEJBHome() throws RemoteException;

    //Obtém um handle para o objeto remoto.
    public Handle getHandle() throws RemoteException;

    //Obtém a chave primária para o objeto.
    public Object getPrimaryKey() throws RemoteException;

    //Testa se um objeto EJB é idêntico ao objeto invocado.
    public boolean isIdentical(EJBObject ejbo) throws RemoteException;

    //Remove o objeto EJB.
    public void remove() throws RemoteException;

}
```

Figura 1.4: A interface `EJBObject`.

A figura 1.2 mostra um cliente interagindo com objetos *EJB Home* e *EJB Object*. Estes objetos são respectivamente instâncias das classes geradas pelo *container* para implementar as interfaces *home* e *remote* definidas pelo fornecedor do componente. Ao implantar um componente, o *container* automaticamente registra seu *EJB Home* num diretório, por meio da API JNDI (*Java Naming and Directory Interface*) [8]. Usando JNDI, qualquer cliente pode localizar esse *EJB Home*, para então criar ou encontrar uma instância do componente. Quando um cliente cria ou encontra uma instância do componente, o *container* retorna uma referência para o *EJB Object* correspondente.

O *container* intercepta todas as operações efetuadas sobre uma instância do componente. Quando um cliente invoca um método do componente a requisição é direcionada

para o *container*[1], que só depois de executar os serviços de gerenciamento de estado, controle de transações, segurança e persistência, delega a chamada a um método negocial da classe do componente.

A arquitetura EJB especifica dois tipos básicos de componentes: *entity bean* e *session bean*.

1.5.1 *Entity Beans*

Um típico componente do tipo entidade possui as seguintes características:

- Fornece uma visão em forma de objeto dos dados de uma base de dados
- Permite acesso compartilhado por múltiplos usuários
- Possui "vida longa" (mesmo tempo de vida do dado associado na base de dados)
- A instância de um *entity bean*, sua chave primária, e sua interface remota sobrevivem a uma eventual queda (*crash*) do *container* EJB. Se o estado de uma instância estava sendo atualizado por uma transação quando ocorreu a queda, ele é automaticamente restaurado ao estado anterior, estabelecido na última efetivação de transação. O encerramento da execução do *container* não é totalmente transparente para o cliente, ou seja, o cliente pode receber uma exceção se efetuar uma chamada de um método do *entity bean* afetado pela queda.

Os componentes do tipo entidade modelam objetos persistentes, normalmente registros em algum tipo de base de dados. Descrevem o estado e comportamento de objetos do mundo real, permitindo aos desenvolvedores encapsular os dados e as regras negociais associadas a estes dados. Isto possibilita que dados associados com conceitos sejam manipulados consistentemente e de forma segura.

Quando um *entity bean* é criado, um novo registro deve ser inserido na base de dados. A nova instância do *entity bean*, por sua vez, deverá ser associada ao novo registro inserido. As alterações no estado da instância do *entity bean* devem ser sincronizadas com os dados na base de dados.

[1]Mais precisamente: a requisição é direcionada para um "objeto intermediário" (o *EJB Home* ou o *EJB Object* da figura 1.2), que atua como preposto do *container*. O objeto intermediário foi instanciado pelo *container*, a partir de uma classe gerada por este.

Vamos utilizar como exemplo uma aplicação simplificada de controle de estoque. Nessa aplicação, cada ítem de estoque possui os seguintes atributos: código, descrição do ítem, quantidade e fornecedor do ítem. Além disso, o conjunto de ítens que forma um estoque tem um identificador de estoque. Este identificador pode, por exemplo, indicar a localização física do estoque. Todos os ítens de um mesmo estoque têm o mesmo identificador de estoque. Modelando um estoque como um componente persistente, do tipo entidade, podemos definir suas interfaces *home* e *remote* como mostrado nas figuras 1.5 e 1.6.

```
package exemplo.estoque;

import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.FinderException;

public interface EstoqueHome extends javax.ejb.EJBHome {

    public Estoque create(int codigo)
        throws CreateException, RemoteException;

    public Estoque findByPrimaryKey(EstoquePK pk)
        throws FinderException, RemoteException;

    public java.util.Collection findAll()
        throws FinderException, RemoteException;

    public java.util.Collection findByDescricao(String desc)
        throws FinderException, RemoteException;

}
```

Figura 1.5: A interface *home* do componente `Estoque`.

A interface `EstoqueHome` contém declarações de métodos para criar um ítem de estoque dado o código desse ítem, para buscar ítens do estoque pela chave primária ou pela descrição do ítem e para buscar todos os ítens do estoque. Os métodos `create` e `findByPrimaryKey` retornam objetos tipo `Estoque`, que representam ítens do estoque. Os métodos `findByDescricao` e `findAll` retornam coleções de objetos tipo `Estoque`. O pri-

meiro retorna a coleção de ítens com a descrição especificada. O segundo retorna a coleção de todos os ítens do estoque.

A interface *remote*, `Estoque`, modela um ítem do estoque e declara os métodos negociais do ítem. Como cada ítem do estoque representa um registro numa base de dados, seus métodos negociais são métodos de leitura ou escrita dos atributos do registro: o código (que não é alterável), a descrição, a quantidade e o fornecedor do ítem. A interface `Estoque` tem também um método de leitura do identificador do estoque que contém o ítem.

```
package exemplo.estoque;

import java.rmi.RemoteException;

public interface Estoque extends javax.ejb.EJBObject {

    public int getCodigo() throws RemoteException;

    public String getDescricao() throws RemoteException;

    public void setDescricao(String desc) throws RemoteException;

    public int getQuantidade() throws RemoteException;

    public void setQuantidade(int qtde) throws RemoteException;

    public String getFornecedor() throws RemoteException;

    public void setFornecedor(String forn) throws RemoteException;

    public String getIdEstoque() throws RemoteException;

}
```

Figura 1.6: A interface *remote* do componente `Estoque`.

Além de definir as interfaces *home* e *remote* e implementar a classe do componente, no caso de um *entity bean* o fornecedor do componente deve escrever também uma classe que representa a chave primária do componente. A figura 1.7 mostra a classe da chave primária do componente `Estoque`, que simplesmente contém o código de um ítem.

```
package exemplo.estoque;

public class EstoquePK implements java.io.Serializable {

    public int codigo;

    public int hashCode() {
        return codigo;
    }

    public boolean equals(Object obj) {
        if (obj instanceof EstoquePK) {
            return (codigo == ((EstoquePK)obj).codigo);
        }
        return false;
    }

    public String toString() {
       return String.valueOf(codigo);
    }

}
```

Figura 1.7: A chave primária do componente `Estoque`.

O processo de coordenar os dados representados por uma instância *entity bean* com a base de dados é denominado persistência. Existem dois tipos de *entity beans*, que se distinguem pela forma como é gerenciada a persistência:

- *Entity bean* com persistência gerenciada pelo *container*
- *Entity bean* com persistência gerenciada pelo componente

Persistência Gerenciada pelo *Container*. Neste caso a persistência é delegada ao *container* EJB. Ao implantar o *entity bean*, identifica-se quais campos serão automaticamente gerenciados pelo *container* e como serão mapeados para a base de dados. Uma vez feito isso, o *container* gera a lógica necessária para salvar o estado de uma instância do *entity bean* automaticamente.

- Vantagem — O *entity bean* pode ser definido independentemente da base de dados. O estado da instância é definido independentemente o que torna o componente mais reutilizável e flexível.
- Desvantagem — O *container* precisa oferecer ferramentas sofisticadas de mapeamento. Em alguns casos, pode ser simplesmente um mapeamento de cada campo da instância numa coluna da base de dados, ou uma serialização do componente inteiro em um arquivo. Em outros casos, o estado de alguns componentes poderá ser definido em termos de um *join* complexo numa base de dados relacional.

Persistência Gerenciada pelo Componente. Neste caso o desenvolvedor do *entity bean* deverá escrever explicitamente a lógica da persistência na classe do componente. Para isso, deverá conhecer previamente que tipo de base de dados estará sendo utilizada e como os campos de uma instância do *entity bean* serão mapeados.

- Vantagem — Fornece mais flexibilidade em como o estado é gerenciado. *Entity beans* que sejam definidos como *joins* complexos, ou como uma combinação dos dados de diferentes bases de dados, ou que tenham seus estados armazenados em sistemas legados, serão beneficiados com este tipo de persistência. Essencialmente, a persistência gerenciada pelo componente é a alternativa existente quando as ferramentas de implantação oferecidas pelo *container* forem inadequadas para realizar o mapeamento para a base de dados.
- Desvantagem — Requer muito mais trabalho para se definir o componente. É preciso entender a estrutura da base de dados e desenvolver a lógica para criar, atualizar e remover dados associados com o componente, além de explicitamente escrever o código dos métodos de busca definidos na interface *home* do componente. Outra desvantagem é que o componente fica amarrado ao tipo e estrutura da base de dados. Qualquer alteração na base de dados ou na estrutura dos dados requer alterações na definição do componente, o que pode não ser trivial.

No caso do componente `Estoque`, o mapeamento entre instâncias do componente e registros da base de dados é bastante simples. Persistência gerenciada pelo *container* é perfeitamente adequada para essa situação. Empregando persistência gerenciada pelo *container*, podemos implementar a classe do componente `Estoque` conforme mostram as figuras 1.8

e 1.9. Essa classe, que denominamos `EstoqueBean`, possui campos correspondentes aos atributos de um ítem do estoque.

```
package exemplo.estoque;

public class EstoqueBean implements javax.ejb.EntityBean {

    public int codigo;
    public String descricao;
    public int quantidade;
    public String fornecedor;

    public EstoquePK ejbCreate(int codigo) {
        this.codigo = codigo;
        return null;
    }

    public void ejbPostCreate(int codigo) { }

    public int getCodigo() {
        return codigo;
    }

    public String getDescricao() {
        return descricao;
    }

    public void setDescricao(String str) {
        descricao = str;
    }

    public int getQuantidade() {
        return quantidade;
    }

    public void setQuantidade (int qtde) {
        quantidade = qtde;
    }
```

Figura 1.8: A classe do componente `Estoque`.

```
    public String getFornecedor() {
        return fornecedor;
    }

    public void setFornecedor(String forn) {
        fornecedor = forn;
    }

    public String getIdEstoque()
       throws EJBException, RemoteException {
         try {
             Context initCtx = new InitialContext();
             Context myEnv = (Context)initCtx.lookup("java:comp/env");
             return (String)myEnv.lookup("idEstoque");
             }
             catch(javax.naming.NamingException ne) {
                  throw new EJBException(ne);
             }
    }

    public void setEntityContext(javax.ejb.EntityContext ctx) { }

    public void unsetEntityContext() { }

    public void ejbActivate() { }

    public void ejbPassivate() { }

    public void ejbLoad() { }

    public void ejbStore() { }

    public void ejbRemove() { }

}
```

Figura 1.9: A classe do componente `Estoque` (continuação).

A classe `EstoqueBean` tem métodos `ejbCreate` e `ejbPostCreate`, ambos com a mesma lista de argumentos que o método `create` da interface `EstoqueHome`. Tem também métodos correspondentes aos da interface `Estoque`, para acesso aos atributos de um ítem do estoque. Finalmente, essa classe implementa a interface `javax.ejb.EntityBean`. Ela o faz de modo trivial, pois fornece implementações com o corpo vazio para todos os sete métodos declarados na interface `javax.ejb.EntityBean`: `setEntityContext`, `unsetEntityContext`, `ejbActivate`, `ejbPassivate`, `ejbLoad`, `ejbStore` e `ejbRemove`.

Em geral a interface *home* de um componente pode ter vários métodos `create`. A classe do componente deve ter um par de métodos `ejbCreate` e `ejbPostCreate` para cada método `create` da interface *home*. No caso de um *entity bean* com persistência gerenciada pelo componente, cada método `ejbCreate` deve inserir um novo registro na base de dados e retornar a chave primária desse registro. No caso de um *entity bean* com persistência gerenciada pelo *container*, como é o caso de nosso componente `Estoque`, o método `ejbCreate` simplesmente inicializa os campos do componente e retorna `null`. A inserção de um novo registro na base de dados fica a cargo do *container*.

Na classe `EstoqueBean` não aparecem implementações dos métodos `find<...>` declarados na interface `EstoqueHome`. *Entity beans* com persistência gerenciada pelo *container* não fornecem implementações desses métodos na classe do componente, pois tais implementações são automaticamente geradas pelo *container*. Já no caso de um *entity bean* com persistência gerenciada pelo componente, a classe do componente deve ter um método `ejbFind<...>` para cada método `find<...>` declarado na interface *home*.

O componente estoque utiliza uma variável de ambiente para identificar a coleção de ítens de estoque que está representando. A esta variável, denominada `idEstoque`, deverá ser atribuído um valor em tempo de implantação. Para obter o valor dessa entrada do ambiente, o método `getIdEstoque` utiliza o pacote `javax.naming`, que é parte da API JNDI. As entradas do ambiente de um componente ficam acessíveis a este via JNDI, no subcontexto de nomes cujo nome (relativo ao contexto de nomes inicial do componente) é `"java:comp/env"`.

Por meio dos sete métodos da interface `javax.ejb.EntityBean`, o *container* efetua *callbacks* para notificar a implementação do componente da ocorrência de certos eventos no decorrer da vida de uma instância de *entity bean*. O *container* chama esses métodos quando determinadas ações sobre a instância estão prestes a ser tomadas ou acabaram de ser tomadas. Exemplificando: o processo que garante que um registro da base de da-

dos e uma instância de um componente do tipo entidade são equivalentes é denominado sincronização. Eventos de sincronização entre o estado da instância do componente e a base de dados ocasionam chamadas aos métodos `ejbLoad` e `ejbStore`. Na persistência gerenciada pelo *container* os campos do componente são automaticamente sincronizados com a base de dados. Por esse motivo os métodos `ejbLoad` e `ejbStore` da classe `EstoqueBean` têm o corpo vazio. Nos casos em que uma lógica mais sofisticada for necessária, como por exemplo quando se deseja reformatar ou comprimir a informação, pode-se efetuar estas ações nos métodos `ejbLoad` e `ejbStore`. O método `ejbLoad` é chamado logo após o *container* atualizar os campos do componente de acordo com os dados da base de dados. O método `ejbStore` é chamado imediatamente antes do *container* ler os campos do componente para armazená-los na base de dados.

Examinando a implementação do componente `Estoque`, nota-se a total ausência de comandos que lidem com a base de dados. Todo o código necessário para interagir com a base de dados e garantir a persistência dos campos de uma instância do componente será gerado automaticamente pelo *container*. Dessa forma, o desenvolvedor de um componente do tipo entidade com persistência gerenciada pelo *container* pode se concentrar na definição e na implementação dos métodos negociais específicos do seu componente. O resultado é uma implementação simples e limpa.

1.5.2 *Session Beans*

Um típico componente do tipo sessão possui as seguintes características:

- É utilizado por um único cliente
- Não representa diretamente dados compartilhados mantidos na base de dados
- Pode atualizar dados compartilhados mantidos numa base de dados
- Possui vida relativamente curta
- É removido quando ocorre uma queda do *container* EJB. O cliente tem de obter um novo objeto sessão para continuar seu trabalho.

Ao contrário dos *entity beans*, os *session beans* não representam dados numa base de dados, embora possam consultar e atualizar bases de dados. Eles são úteis para descrever e

gerenciar interações entre componentes *entity bean* e para implementar tarefas específicas, tipicamente fluxos de trabalho.

Existem dois tipos de *session bean*:

- **Sem estado** — É uma coleção de serviços relacionados, cada um representado por um método. Entre uma invocação de método e outra, uma instância do componente não mantém nenhum estado que seja ao mesmo tempo específico da instância e vinculado ao cliente desta instância. Ao se invocar um método num *session bean* sem estado, este executa o método e retorna o resultado sem levar em consideração requisições anteriores ou futuras.

- **Com estado** — É dedicado a um cliente pelo tempo de vida da instância do componente, comportando-se como um agente do cliente. Pode consultar e atualizar dados numa base de dados, mas não pode representá-los como os *entity beans*. *Session beans* com estado mantêm estado conversacional, ou seja, as variáveis da instância podem guardar dados relativos ao cliente entre as chamadas de métodos. Isto possibilita uma interdependência entre os métodos, de modo que alterações feitas no estado do componente possam afetar o resultado das chamadas dos métodos subseqüentes.

Retomando nossa aplicação de controle de estoque, vamos criar um componente do tipo sessão, denominado `GerenciadorDeEstoque`, com a finalidade de gerenciar as interações entre o *entity bean* `Estoque` e as aplicações clientes. A figura 1.10 mostra a interface *home* do componente `GerenciadorDeEstoque`. A ausência de métodos `find<...>` é um aspecto que distingue essa interface da interface *home* do componente `Estoque`. Ao contrário

```
package exemplo.gerenciador;

public interface GerenciadorDeEstoqueHome extends javax.ejb.EJBHome {

    public GerenciadorDeEstoque create()
        throws javax.ejb.CreateException, java.rmi.RemoteException;

}
```

Figura 1.10: A interface *home* do componente `GerenciadorDeEstoque`.

dos componentes tipo entidade, componentes tipo sessão não têm chave primária nem suportam métodos `find<...>`.

As declarações dos métodos negociais do `GerenciadorDeEstoque` estão na interface *remote* desse componente, apresentada na figura 1.11. Para proporcionar funcionalidades

```
package exemplo.gerenciador;

import java.rmi.RemoteException;
import javax.ejb.FinderException;

public interface GerenciadorDeEstoque extends javax.ejb.EJBObject {

    public ItemEstoque criaItemEstoque(int cod, String desc,
                                       int qtde, String forn)
        throws javax.ejb.CreateException, RemoteException;

    public ItemEstoque consultaItemEstoque(int cod)
        throws FinderException, RemoteException;

    public void removeItemEstoque(int cod)
        throws FinderException, javax.ejb.RemoveException, RemoteException;

    public void incrementaEstoque(int cod, int qtde)
        throws FinderException, RemoteException;

    public void liberaEstoque(int cod, int qtde)
        throws FinderException, RemoteException;

    public ItemEstoque[] listaEstoque()
        throws RemoteException;

    public String[] listaFornecedoresDeItem(String desc)
        throws RemoteException;

    public String [] listaFornecedores()
        throws RemoteException;

}
```

Figura 1.11: A interface *remote* do componente `GerenciadorDeEstoque`.

básicas de gerenciamento de estoque, incluímos nessa interface métodos que permitem:

1. Criar ítem de estoque
2. Obter informações sobre determinado ítem de estoque (operação de consulta)
3. Remover ítem de estoque
4. Incrementar a quantidade de um ítem de estoque
5. Liberar (decrementar) a quantidade de um ítem de estoque
6. Listar todos os ítens da base de estoque
7. Listar fornecedores de determinado ítem de estoque
8. Listar todos os fornecedores da base de estoque

Com o objetivo de repassar a seus clientes as informações sobre um certo ítem do estoque, o componente `GerenciadorDeEstoque` utiliza a classe auxiliar mostrada na figura 1.12.

```
package exemplo.gerenciador;

public class ItemEstoque implements java.io.Serializable {

    public int codigo;
    public String descricao;
    public int quantidade;
    public String fornecedor;
    public String estoque;

    public ItemEstoque(int codigo, String descricao, int quantidade,
                       String fornecedor, String estoque) {
        this.codigo = codigo;
        this.descricao = descricao;
        this.quantidade = quantidade;
        this.fornecedor = fornecedor;
        this.estoque = estoque;
    }
    public String toString() {
         return ("\ncod: " + codigo + "\ndescr: " + descricao
                 + "\nquant: " + quantidade + "\nforn: " + fornecedor
                 + "\nestoque: " + estoque);
    }
}
```

Figura 1.12: Classe auxiliar utilizada pelo componente `GerenciadorDeEstoque`.

As figuras 1.13 e 1.14 mostram a classe do componente `GerenciadorDeEstoque`, denominada `GerenciadorDeEstoqueBean`. Para evitar redundância, omitimos as implementações dos últimos quatro métodos negociais da interface `GerenciadorDeEstoque`.

```
package exemplo.gerenciador;

import java.rmi.RemoteException;
import javax.ejb.FinderException;
import javax.ejb.CreateException;
import javax.ejb.RemoveException;

public class GerenciadorDeEstoqueBean implements javax.ejb.SessionBean {

    public EstoqueHome home;

    public void ejbCreate() throws CreateException {
        try {
            javax.naming.Context jndiContext =
                            new javax.naming.InitialContext();
            Object obj = jndiContext.lookup(
                            "java:comp/env/ejb/refComponenteEstoque");
            home = (EstoqueHome)javax.rmi.PortableRemoteObject.narrow(
                            obj, EstoqueHome.class);
        }
        catch (javax.naming.NamingException ne) {
            throw new CreateException("Estoque não encontrado");
        }
    }

    public ItemEstoque criaItemEstoque(int cod, String desc,
                                       int qtde, String forn)
                    throws CreateException, RemoteException {
        Estoque e = home.create(cod);
        e.setDescricao(desc);
        e.setQuantidade(qtde);
        e.setFornecedor(forn);
        return new ItemEstoque(cod, desc, qtde, forn, e.getIdEstoque());
    }
```

Figura 1.13: A classe do componente `GerenciadorDeEstoque`.

```
    public ItemEstoque consultaItemEstoque(int cod)
                    throws FinderException, RemoteException {
        EstoquePK pk = new EstoquePK();
        pk.codigo = cod;
        Estoque e = home.findByPrimaryKey(pk);
        return new ItemEstoque(cod, e.getDescricao(), e.getQuantidade(),
                               e.getFornecedor(), e.getIdEstoque());
    }

    public void removeItemEstoque(int cod)
                    throws FinderException,
                           RemoveException, RemoteException {
        EstoquePK pk = new EstoquePK();
        pk.codigo = cod;
        home.findByPrimaryKey(pk).remove();
    }

    public void incrementaEstoque(int cod, int qtde)
                    throws FinderException, RemoteException {
        EstoquePK pk = new EstoquePK();
        pk.codigo = cod;
        Estoque e = home.findByPrimaryKey(pk);
        e.setQuantidade(e.getQuantidade() + qtde);
    }

    ... // omitidas as implementações dos demais métodos negociais

    public void setSessionContext(javax.ejb.SessionContext cntx) { }

    public void ejbActivate() { }

    public void ejbPassivate() { }

    public void ejbRemove() { }

}
```

Figura 1.14: A classe do componente `GerenciadorDeEstoque` (continuação).

A classe GerenciadorDeEstoqueBean tem um método ejbCreate com a mesma lista de argumentos que o método create da interface GerenciadorDeEstoqueHome. Tem também métodos negociais correspondentes aos da interface GerenciadorDeEstoque. Finalmente, ela implementa de forma trivial a interface javax.ejb.SessionBean. A classe GerenciadorDeEstoqueBean fornece implementações com o corpo vazio para os quatro métodos dessa interface: setSessionContext, ejbActivate, ejbPassivate e ejbRemove. As "obrigações contratuais" de um componente tipo sessão são menores que as de um componente tipo entidade, pois a interface javax.ejb.SessionBean possui menos métodos que a javax.ejb.EntityBean e, além disso, a classe de um componente tipo sessão não precisa ter um método ejbPostCreate.

Um GerenciadorDeEstoque está associado a um estoque. A classe do componente GerenciadorDeEstoque possui um campo home, que referencia a interface EstoqueHome do estoque gerenciado. Por ocasião da criação de uma instância esse campo recebe o valor da entrada "ejb/refComponenteEstoque", que o implantador deve ter definido no ambiente do componente GerenciadorDeEstoque. O emprego de uma entrada no ambiente permite que o vínculo entre um GerenciadorDeEstoque e o seu Estoque seja estabelecido somente em tempo de implantação.

A exemplo das implementações de métodos negociais apresentadas nas figuras 1.13 e 1.14, as que foram omitidas usam o campo home para interagir com a interface *home* do componente Estoque associado ao GerenciadorDeEstoque. Cabe ressaltar que embora a classe do componente GerenciadorDeEstoque tenha uma variável de instância (o campo home), esse componente é um *session bean* sem estado. A distinção entre os *session beans* sem estado e os com estado é baseada na existência de estado vinculado a um cliente específico. O valor do campo home de uma instância do componente GerenciadorDeEstoque não está associado a nenhum cliente. De fato, esse valor é o mesmo para todas as instâncias de um componente implantado.

1.6 A Visão do Cliente

Para um cliente, um objeto do tipo sessão é um objeto não persistente que implementa alguma lógica negocial rodando num servidor. Um *entity bean* é um componente que fornece uma visão orientada a objeto de entidades armazenadas em alguma unidade de armazenamento persistente, tal como um banco de dados, ou de entidades persistentes

implementadas por uma aplicação corporativa já existente.

Um cliente utiliza um componente do tipo sessão ou um componente do tipo entidade através das interfaces *home* e *remote* do componente. Essas interfaces constituem a visão do cliente. O *container* gera ou fornece classes que implementam as interfaces *home* e *remote* dos componentes nele implantados. Embora a visão do cliente de um componente implantado seja de fato realizada por classes geradas ou fornecidas pelo *container*, estas são transparentes para o cliente.

JNDI permite que uma aplicação cliente visualize o servidor EJB como uma estrutura (possivelmente hierárquica) de contextos de nomes. Depois que a aplicação cliente usa JNDI para localizar e obter uma referência para a interface *home* do componente EJB, ela utiliza a interface *home* para obter uma referência para a interface *remote* de uma instância do componente e, a partir desse momento, invocar métodos negociais sobre essa instância.

1.7 Gerenciamento de Transações

Suporte a transações é especialmente importante num ambiente distribuído, já que agentes interagindo via rede podem perder o contato entre si, ou um agente pode sofrer uma queda durante uma série de interações em que está engajado com outro agente.

O *container* EJB possui um papel importante no gerenciamento de transações, já que é o responsável tanto pela geração de transações para as interações do clientes com os componentes, quanto pela detecção de transações requisitadas pelos clientes. A arquitetura EJB recai na *Java Transaction API* (JTA) [8] para o gerenciamento de transações distribuídas. JTA define as interfaces necessárias para a interação com um servidor de transações.

No contexto de um componente EJB, os limites de uma transação podem ser definidos pelo cliente, pelo *container* ou pelo próprio componente. Em todos os casos, o *container* decide como lidar com o contexto da transação a cada vez que um método remoto do componente for invocado. Durante o tempo de vida do componente, o *container* decide se executa os métodos negociais do componente dentro da transação do cliente, ou dentro de uma transação que o *container* define, ou se permite que o próprio componente gerencie os limites da transação.

Em tempo de implantação de um componente, pode-se selecionar um dos seguintes valores para o seu atributo de suporte a transações:

- `TX_NOT_SUPPORTED` — O componente não suporta transações, logo seus métodos devem ser invocados sem um contexto de transação. Se o cliente iniciou uma transação, o *container* a suspende antes de chamar o método requisitado pelo cliente. Após o término da execução do método o *container* restabelece a transação do cliente.

- `TX_SUPPORTED` — O componente suporta transações iniciadas pelo cliente. Se o cliente chama um método no componente estando dentro de uma transação, o contexto desta transação do cliente é passado ao componente, caso contrário o método é executado sem o contexto transacional.

- `TX_REQUIRED` — O componente requer que todas as chamadas feitas a seus métodos sejam executadas dentro do contexto de uma transação. Se o cliente chamar um método dentro de uma transação, o contexto desta transação é repassado ao componente. Caso contrário, o *container* cria uma nova transação para execução do método chamado. Esta transação será efetivada após o término de execução do método, mas antes do retorno dos resultados ao cliente.

- `TX_REQUIRES_NEW` — O componente requer que todas as chamadas feitas a seus métodos sejam executadas dentro de uma transação nova. O *container* automaticamente inicia uma nova transação antes de chamar o método referenciado. Essa transação será efetivada após o término de execução do método, mas antes do retorno dos resultados ao cliente. Se o cliente invocar um método do componente dentro de uma transação, o *container* a suspende e cria uma nova transação para execução do método chamado. A transação do cliente só é restaurada após a efetivação da nova transação criada.

- `TX_MANDATORY` — O cliente só poderá invocar um método do componente após iniciar uma transação, caso contrário o *container* lançará uma exceção.

- `TX_BEAN_MANAGED` — O componente gerencia todos os limites de suas transações. Se o cliente invocar um método do componente dentro de uma transação, esta transação é suspensa pelo tempo de execução do método chamado. Os métodos negociais do componente serão executados dentro de uma transação somente se o componente explicitamente criar uma transação para cada chamada recebida.

1.8 O Descritor de Implantação

Existem dois tipos de informação no descritor de implantação:

Informações estruturais do componente — descrevem a estrutura e declaram as dependências externas do componente. O produtor de um arquivo ejb-jar deve obrigatoriamente fornecer informações estruturais no descritor de implantação do componente.

Informações de montagem da aplicação — descrevem como o(s) componente(s) contido(s) num arquivo ejb-jar se relacionam de modo a formar uma unidade de implantação de aplicação. O produtor de um arquivo ejb-jar pode opcionalmente incluir informações de montagem.

O descritor de implantação é um arquivo XML [39] escrito de acordo com uma *Document Type Definition* (DTD) padronizada pela especificação EJB [28, 29]. Essa DTD define a sintaxe de um descritor de implantação e os elementos XML que podem aparecer nele. Além de respeitar as regras sintáticas de sua DTD, o conteúdo de um descritor de implantação deve estar também em conformidade com uma série de regras semânticas, especificadas nos comentários dessa DTD [28, 29]. Um descritor de implantação deverá referenciar sua DTD por meio de uma declaração como a apresentada na figura 1.15, onde `V` e `R` representam os números da versão e da revisão da especificação EJB que padronizou a DTD.

```
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//
  DTD Enterprise JavaBeans V.R//EN"
  "http://java.sun.com/j2ee/dtds/ejb-jar_V_R.dtd">
```

Figura 1.15: Referência para a DTD de um descritor de implantação.

O fornecedor do componente deve utilizar o elemento `enterprise-beans` para listar todos os componentes no arquivo ejb-jar. As seguintes informações devem ser fornecidas para cada componente:

- Nome do componente — Deverá ser atribuído um nome lógico a cada componente no arquivo ejb-jar. Não existe relação entre este nome e o nome JNDI que o implantador irá atribuir ao componente. O fornecedor do componente especifica o nome do componente no elemento `ejb-name`.

- Tipo do componente — Os tipos de componentes possíveis são entidade e sessão. O fornecedor do componente deve usar o elemento `entity` ou `session` para especificar o tipo do componente.

- Interface *home* do componente — O fornecedor do componente deve especificar o nome completo da interface *home* do componente no elemento `home`.

- Interface *remote* do componente — O fornecedor do componente deve especificar o nome completo da interface *remote* do componente no elemento `remote`.

- Classe do componente — O fornecedor do componente deve especificar o nome completo da classe Java que implementa os métodos negociais do componente. Isto deve ser especificado o elemento `ejb-class`.

- Tipo de sessão — Se o componente for do tipo sessão, o fornecedor do componente deverá usar o elemento `session-type` para declarar se o componente é com ou sem estado.

- Tipo de demarcação de transação do componente — Se o componente for do tipo sessão, o fornecedor do componente deve usar o elemento `transaction-type` para declarar se a demarcação será executada pelo componente ou pelo *container*.

- Gerenciamento de persistência do componente — Se o componente for do tipo entidade, o fornecedor do componente deve usar o elemento `persistence-type` para declarar se o gerenciamento de persistência será efetuado pelo componente ou pelo *container*.

- Classe da chave primária do componente — Se o componente for do tipo entidade, o fornecedor do componente pode usar o elemento `prim-key-class` para especificar o nome completo da classe da chave primária do componente. A especificação da classe da chave primária é obrigatória no caso de um componente do tipo entidade com persistência gerenciada pelo componente e é opcional no caso de um componente do tipo entidade com persistência gerenciada pelo *container*.

- Campos gerenciados pelo *container* — Se o componente for do tipo entidade com gerenciamento de persistência feito pelo *container*, o fornecedor do componente deve especificar os campos persistentes por meio de elementos `cmp-field` (*container-managed persistent field*).

- Entradas do ambiente — O fornecedor do componente deve declarar as entradas do ambiente por meio de elementos `env-entry`.
- Referências a fábricas de recursos — O fornecedor do componente deve declarar as referências para fábricas de recursos por meio de elementos `resource-ref`.
- Referências EJB — O fornecedor do componente deve empregar elementos `ejb-ref` para declarar as entradas do ambiente que referenciam interfaces *home* de outros componentes.
- Referências a papéis de segurança — O fornecedor do componente deve declarar as referências a papéis de segurança usando elementos `security-role-ref`.

Voltando à nossa aplicação exemplo: as figuras 1.16, 1.17 e 1.18 mostram o descritor de implantação do *entity bean* `Estoque`.

```
<?xml version="1.0"?>

<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems,
 Inc.//DTD Enterprise JavaBeans 1.1//EN"
 "http://java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd">

<ejb-jar>

 <enterprise-beans>

   <entity>
      <description>
            Entity bean que representa uma unidade de estoque.
      </description>
      <ejb-name>ComponenteEstoque</ejb-name>
      <home>exemplo.estoque.EstoqueHome</home>
      <remote>exemplo.estoque.Estoque</remote>
      <ejb-class>exemplo.estoque.EstoqueBean</ejb-class>

      <persistence-type>Container</persistence-type>
```

Figura 1.16: O descritor de implantação do componente `Estoque`.

```
     <prim-key-class>exemplo.estoque.EstoquePK</prim-key-class>
     <reentrant>False</reentrant>

     <cmp-field><field-name>codigo</field-name></cmp-field>
     <cmp-field><field-name>descricao</field-name></cmp-field>
     <cmp-field><field-name>quantidade</field-name></cmp-field>
     <cmp-field><field-name>fornecedor</field-name></cmp-field>

     <env-entry>
        <description>
               Identificador do Estoque
        </description>
        <env-entry-name>idEstoque</env-entry-name>
        <env-entry-type>java.lang.String</env-entry-type>
        <env-entry-value>Estoque Numero 1</env-entry-value>
     </env-entry>
  </entity>

</enterprise-beans>

<assembly-descriptor>

  <security-role>
     <description>
        Permissão de acesso total ao componente estoque.
     </description>
    <role-name>everyone</role-name>
  </security-role>

  <method-permission>
    <role-name>everyone</role-name>
    <method>
        <ejb-name>ComponenteEstoque</ejb-name>
        <method-name>*</method-name>
    </method>
  </method-permission>
```

Figura 1.17: O descritor de implantação do componente `Estoque` (continuação).

```
   <container-transaction>
     <method>
        <ejb-name>ComponenteEstoque</ejb-name>
        <method-name>*</method-name>
     </method>
     <trans-attribute>Required</trans-attribute>
   </container-transaction>

 </assembly-descriptor>

</ejb-jar>
```

Figura 1.18: O descritor de implantação do componente `Estoque` (continuação).

As figuras 1.19, 1.20 mostram o descritor de implantação do utilizado para o *session bean* `GerenciadorDeEstoque`.

```
<?xml version="1.0"?>

<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems,
 Inc.//DTD Enterprise JavaBeans 1.1//EN"
 "http://java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd">

<ejb-jar>

 <enterprise-beans>

  <session>
   <ejb-name>ComponenteGerenciadorDeEstoque</ejb-name>
   <home>exemplo.gerenciador.GerenciadorDeEstoqueHome</home>
   <remote>exemplo.gerenciador.GerenciadorDeEstoque</remote>
   <ejb-class>exemplo.gerenciador.GerenciadorDeEstoqueBean</ejb-class>
   <session-type>Stateless</session-type>
   <transaction-type>Container</transaction-type>
```

Figura 1.19: O descritor de implantação do componente `GerenciadorDeEstoque`.

```
  <ejb-ref>
    <ejb-ref-name>ejb/refComponenteEstoque</ejb-ref-name>
    <ejb-ref-type>Entity</ejb-ref-type>
    <home>exemplo.estoque.EstoqueHome</home>
    <remote>exemplo.estoque.Estoque</remote>
  </ejb-ref>
 </session>

</enterprise-beans>

<assembly-descriptor>

  <security-role>
     <description>
        Permissão de acesso total ao componente gerenciador de estoque.
     </description>
    <role-name>everyone</role-name>
  </security-role>

  <method-permission>
    <role-name>everyone</role-name>
    <method>
        <ejb-name>ComponenteGerenciadorDeEstoque</ejb-name>
        <method-name>*</method-name>
    </method>
  </method-permission>

  <container-transaction>
    <method>
       <ejb-name>ComponenteGerenciadorDeEstoque</ejb-name>
       <method-name>*</method-name>
    </method>
    <trans-attribute>Required</trans-attribute>
  </container-transaction>
 </assembly-descriptor>

</ejb-jar>
```

Figura 1.20: O descritor de implantação do componente `GerenciadorDeEstoque` (cont.).

Quando esses componentes forem implantados num *container*, as informações contidas nos descritores de implantação deverão ser complementadas com dados adicionais. Como a especificação EJB não padroniza o modo de se implantar componentes num *container*, o implantador deve usar ferramentas e/ou convenções específicas do *container* para personalizar elementos do descritor de implantação e/ou fornecer quaisquer dados adicionais necessários em tempo de implantação. No caso do componente `Estoque`, os elementos a serem personalizados incluem o valor de uma entrada do ambiente (`"idEstoque"`). Um dado adicional que será necessário fornecer é a identificação da base de dados na qual os registros de estoque serão mantidos. Caso se trate de uma base de dados relacional, o implantador deverá especificar o mapeamento entre os campos do componente e as colunas de uma tabela. No caso do componente `GerenciadorDeEstoque`, o implantador deverá inserir uma referência EJB (`"ejb/refComponenteEstoque"`) no ambiente do componente.

1.9 A Versão 2.0 da Especificação EJB

As seções precedentes deste capítulo descreveram características presentes na arquitetura EJB desde a versão 1.1 da especificação EJB [29]. Esta seção fornece uma visão geral das novas características recentemente introduzidas pela versão 2.0 da especificação EJB [28]. Os ítens relacionados a seguir descrevem sucintamente as novas funcionalidades:

- Foram introduzidos componentes dirigidos por mensagens (*message-driven*) e foi especificada a integração com *Java Message Service* (JMS) [8], que fornece uma API baseada em mensagens para comunicação entre processos Java. Um componente dirigido por mensagens é um componente sem estado que é chamado pelo *container* como resultado da chegada de uma mensagem JMS. O objetivo deste modelo dirigido por mensagens é permitir que componentes invocáveis assincronamente para processar mensagens JMS sejam desenvolvidos de modo simples e rápido.

- Foi revisada a persistência gerenciada pelo *container* para componentes do tipo entidade e foi adicionado suporte para relacionamentos gerenciados pelo *container*. Novos contratos foram especificados para componentes do tipo entidade com persistência gerenciada pelo *container*, com o objetivo de eliminar limitações da abordagem baseada em campos utilizada nas verões anteriores do EJB. Os novos mecanismos de persistência gerenciada pelo *container* foram adicionados para fornecer as seguintes funcionalidades:

  - Suportar relacionamentos gerenciados pelo *container* entre componentes do tipo entidade
  - Fornecer a base para uma sintaxe de consulta portável para busca

- Foram definidas interfaces *local home* e *local* (análogas às interfaces *home* e *remote*) para componentes do tipo sessão e do tipo entidade. O novo par de interfaces tem o objetivo de suportar acessos mais "leves" por parte de componentes que são clientes locais. Interfaces locais permitem que componentes do tipo sessão e do tipo entidade sejam fortemente conectados a seus clientes e que sejam utilizados sem o custo tipicamente associado com chamadas remotas de métodos. Interfaces locais fornecem também o fundamento para relacionamentos gerenciados pelo *container* entre componentes do tipo entidade com persistência gerenciada pelo *container*.

- Foi definida uma sintaxe declarativa para a especificar métodos de consulta a componentes do tipo entidade com persistência gerenciada pelo *container*. Essa sintaxe permite especificar métodos *finder* e *select*, cujas implementações serão fornecidas pelo *container*. A linguagem resultante, EJB QL (*Enterprise JavaBeans Query Language*), fornece navegação através da rede de *entity beans* definida pelos relacionamentos gerenciados pelo *container*.

- Foram adicionados métodos *select*, para uso interno a componentes do tipo entidade com persistência gerenciada pelo *container*. Estes métodos permitem o uso de consultas EJB QL para selecionar valores ou componentes relacionados.

- Foi acrescentado suporte, na interface *home*, para métodos adicionais que implementem lógica negocial independente de uma instância específica de um componente do tipo entidade. Tais métodos tem propósito de desempenhar, na arquitetura EJB, um papel análogo ao dos métodos de classe das linguagens orientadas a objetos.

- Foi adicionada funcionalidade de segurança de identidade do tipo *run as*, que permite especificação declarativa do principal a ser usado como identidade de um componente.

- Foi definido um protocolo de interoperabilidade baseado em CORBA/IIOP para permitir que chamadas remotas a componentes do tipo sessão e do tipo entidade sejam efetuadas por componentes J2EE implantados em produtos de fabricantes diversos, bem como por clientes CORBA quaisquer, possivelmente não escritos em Java.

Há três tipos de componentes: *session beans*, *entity beans* e *message-driven beans*.

***Session beans* e *entity beans*:** Têm características similares às especificadas na versão 1.1

***Message-driven beans*:** Um componente do tipo *message-driven* possui as seguintes características:

- Executa após a recepção de uma mensagem de um cliente
- Pode ser ciente de transações
- Não representa diretamente dados compartilhados numa base de dados, embora possa consultá-los e atualizá-los
- Não possui estado
- É removido quando ocorre uma queda do *container*. O *container* precisa reconstruí-lo para que o tratamento das mensagens prossiga.

1.9.1 As Visões do Cliente: Local e Remota

O cliente de um componente do tipo sessão ou do tipo entidade pode ser local ou remoto. A inclusão de interfaces para acesso local na versão 2.0 da especificação EJB ocasionou uma mudança de nomenclatura. Assim como um componente EJB 1.x, um componente EJB 2.0 exporta um par de interfaces, que podem agora ser específicas para acesso remoto ou para acesso local. Por esse motivo, a interface que era denominada *remote* em EJB 1.1 passou a ser denominada *component* em EJB 2.0.

A visão remota do cliente pode ser usada para acessar o componente tanto por um cliente local quanto por um cliente remoto. Os objetos que implementam as interfaces para a visão remota do cliente são objetos remotos Java e são acessíveis pelo cliente através das APIs Java para chamadas remotas (RMI). Por outro lado, a visão local do cliente somente pode ser usada para acessar um componente local. Componentes locais executam na mesma JVM que o cliente. A visão local do cliente é baseada no uso de interfaces padrão Java.

O fornecedor do componente e o fornecedor do *container* cooperam para criar uma visão do cliente, seja ela local ou remota. Tanto a visão local como a visão remota incluem os seguintes ítens:

- **Interface *home*** — Contém os métodos que clientes usam para criar, remover e buscar objetos EJB de um mesmo tipo. Componentes que fornecem a seus clientes uma visão remota têm uma interface *home* remota, a qual estende a interface `javax.ejb.EJBHome`. Componentes que fornecem a seus clientes uma visão local têm uma interface *local home*, a qual estende a interface `javax.ejb.EJBLocalHome`.

- **Interface *component*** — Um objeto EJB é acessível através de sua interface *component*. Esta interface possui os métodos negociais que podem ser invocados por clientes. A interface *component* pode ser uma interface remota ou uma interface local. Uma interface remota estende a interface `javax.ejb.EJBObject`, que contém operações de criação de *handle* persistente para o objeto EJB e operações relacionadas com a identidade desse objeto. Uma interface *local* estende a interface `javax.ejb.EJBLocalObject`, que contém operações relacionadas com a identidade do objeto EJB (vide ítem seguinte).

- **Identidade do objeto** — Cada objeto EJB possui uma identidade única dentro de sua *home*. Para componentes do tipo sessão, o *container* é responsável por gerar um novo identificador único para cada objeto do tipo sessão. Este identificador não é exposto ao cliente. Entretanto, um cliente pode testar se duas referências a objetos identificam um mesmo objeto do tipo sessão. Para componentes do tipo entidade, o fornecedor do componente é o responsável por fornecer uma chave primária em tempo de criação do objeto do tipo entidade. O *container* usa a chave primária para identificar o objeto do tipo entidade. Um cliente pode obter a chave primária de um objeto do tipo entidade por meio da interface `javax.ejb.EJBObject` ou da `javax.ejb.EJBLocalObject`. O cliente também poderá testar se duas referências a objetos identificam o mesmo objeto do tipo entidade.

Clientes Remotos. Um cliente remoto acessa um componente do tipo sessão ou do tipo entidade através das interfaces *remote* e *remote home*, que fornecem a visão remota do cliente. Esta visão é independente de localização. Para acessar um componente, um cliente remoto rodando na mesma JVM que o componente usa a mesma API que um cliente rodando numa JVM diferente, possivelmente em outra máquina.

As interfaces *remote* e *remote home* são interfaces Java RMI. O *container* fornece classes que implementam estas interfaces. Os objetos que implementam as interfaces *remote* e *remote home* são objetos Java, e são acessíveis por um cliente através das APIs Java RMI.

Os argumentos e resultados dos métodos das interfaces e *remote home* são passados por valor.

Um cliente remoto pode ser outro componente implantado no mesmo *container* ou não. Pode também ser um programa Java qualquer: aplicação, *applet* ou *servlet*. A visão remota do cliente pode ainda ser mapeada para ambientes de clientes não Java, tais como clientes CORBA não escritos na linguagem Java.

Clientes Locais. Componentes do tipo sessão e do tipo entidade podem possuir clientes locais. Um cliente local e o componente do tipo sessão ou do tipo entidade que lhe fornece uma visão local têm de rodar na mesma JVM. Ao contrário da visão remota do cliente, a visão local do cliente de um componente não é independente de localização. O acesso a um componente através da visão local requer a coexistência, na mesma JVM, do componente e de seu cliente local. Notar que esse cliente local pode ser outro componente.

Um cliente local acessa um componente do tipo sessão ou do tipo entidade através da interface *local* do componente e da interface *local home*. O *container* fornece as classes que implementam a interface *local home* e a interface *local* do componente. Os objetos que implementam estas interfaces são objetos Java locais.

Os argumentos e resultados dos métodos das interfaces *local home* e *local* são passados por referência. Componentes que fornecem visão local a seus clientes devem portanto ser codificados assumindo que o estado de qualquer objeto Java passado como argumento ou como resultado é compartilhado entre o invocador e o invocado.

1.9.2 Relacionamentos Gerenciados pelo *Container*

Podem ser definidos relacionamentos persistentes, gerenciados pelo *container*, entre componentes tipo entidade que forneçam visões locais e residam na mesma JVM. Um *container* EJB 2.0 pode gerenciar relacionamentos persistentes entre componentes tipo entidade nele implantados, mas não pode gerenciar relacionamentos envolvendo componentes implantados em outro *container*.

Um fornecedor de componentes que desenvolve um conjunto de componentes do tipo entidade para uma dada aplicação pode especificar relacionamentos entre esses componentes. Para tanto, ele projeta um esquema abstrato de persistência associado a cada componente do tipo entidade. Esse esquema define os campos e os relacionamentos persistentes

que deverão ser gerenciados pelo *container*, bem como os métodos para acessá-los. Em tempo de execução, uma instância do componente tipo entidade acessa seus campos e relacionamentos gerenciados pelo *container* por meio dos métodos de seu esquema abstrato de persistência.

O esquema abstrato de persistência é especificado no descritor de implantação produzido pelo fornecedor do componente. O implantador do componente determinará como os campos e relacionamentos persistentes gerenciados pelo *container* serão mapeados para a base de dados e gerará as classes e interfaces que permitirão ao *container* efetuar, em tempo de execução, o gerenciamento da persistência desse campos e relacionamentos.

O descritor de implantação descreve relacionamentos lógicos entre componentes do tipo entidade. Ele não fornece um mecanismo para especificar como o esquema abstrato de persistência deverá ser mapeado para a base de dados. Isto é responsabilidade do implantador do componente.

O fornecedor do componente deverá especificar, no descritor de implantação:

- os campos persistentes, por meio de elementos `cmp-field`;
- os relacionamentos, que poderão ser um-para-um, um-para-muitos ou muitos-para-muitos, por meio de elementos `relationships`;
- os campos de relacionamentos, através de elementos `cmr-fields` (*container-managed relationship fields*).

Um *entity bean* acessa os *entity beans* com ele relacionados através dos métodos acessores para seus campos de relacionamento. Tais métodos são especificados nos elementos `cmr-fields` do descritor de implantação. Os relacionamentos são definidos em termos das interfaces locais dos componentes relacionados. A visão que um componente do tipo entidade apresenta aos componentes relacionados com ele é definida por suas interfaces *home* e *component* locais. Portanto um componente do tipo entidade poderá ser o alvo de um relacionamento de outro componente do tipo entidade somente se possuir interfaces locais.

Embora o contrato para componentes do tipo entidade com persistência gerenciada pelo *container* tenha sido alterado substancialmente na especificação EJB 2.0, *entity beans* que usam a especificação EJB 1.1 são suportados em *containers* EJB 2.0.

Capítulo 2

O Modelo de Componentes CORBA

2.1 CORBA

O *Object Management Group* (OMG) vem promovendo um conjunto de padrões para desenvolvimento de aplicações orientadas a objetos em ambientes distribuídos. Esses padrões são baseados numa arquitetura conhecida como *Object Management Architecture* (OMA) [10, 34], que une as tecnologias de computação distribuída e de orientação a objetos. Um elemento fundamental da OMA é o *Object Request Broker* (ORB), responsável pela comunicação entre clientes e objetos num ambiente distribuído. A arquitetura do ORB é padronizada pela especificação CORBA (*Common Object Request Broker Architecture*) [11, 34]. CORBA define o mecanismo de invocação remota de métodos que fornece a base para a OMA.

O ORB é o responsável pelo encaminhamento de requisições dos clientes para os objetos e de respostas dos objetos para os clientes. Nenhuma das partes envolvidas numa invocação remota de método — nem o cliente nem o objeto alvo da requisição — precisa tomar conhecimento da localização da outra parte (transparência de localização). A interação entre essas partes ocorre independentemente da plataforma de hardware e do sistema operacional de cada parte, bem como das linguagens de programação em elas foram escritas (interoperabilidade).

As interfaces dos objetos CORBA são especificadas numa *Interface Definition Language* (IDL) puramente declarativa, que provê interoperabilidade entre linguagens de programação. Para que um objeto possa receber requisições através do ORB, esse objeto deve ter uma interface definida em IDL. Se o objeto tiver tal interface, seus serviços poderão ser

requisitados por clientes escritos numa linguagem diferente da que foi usada para implementar o objeto.

Clientes e objetos CORBA não são implementados em IDL, que não é uma linguagem de programação, e sim em linguagens de programação para as quais há mapeamentos de IDL. O OMG definiu mapeamentos de IDL para diversas linguagens de programação, como C, C++, Java, Smalltalk, COBOL, Ada, LISP e Python.

A figura 2.1 mostra as partes da arquitetura CORBA. O *stub* e o esqueleto IDL nela representados são gerados mecanicamente por compiladores IDL, a partir da definição da interface IDL de um objeto. O *stub IDL* é agregado aos clientes do objeto e o esqueleto IDL é incorporado ao servidor que implementa o objeto.

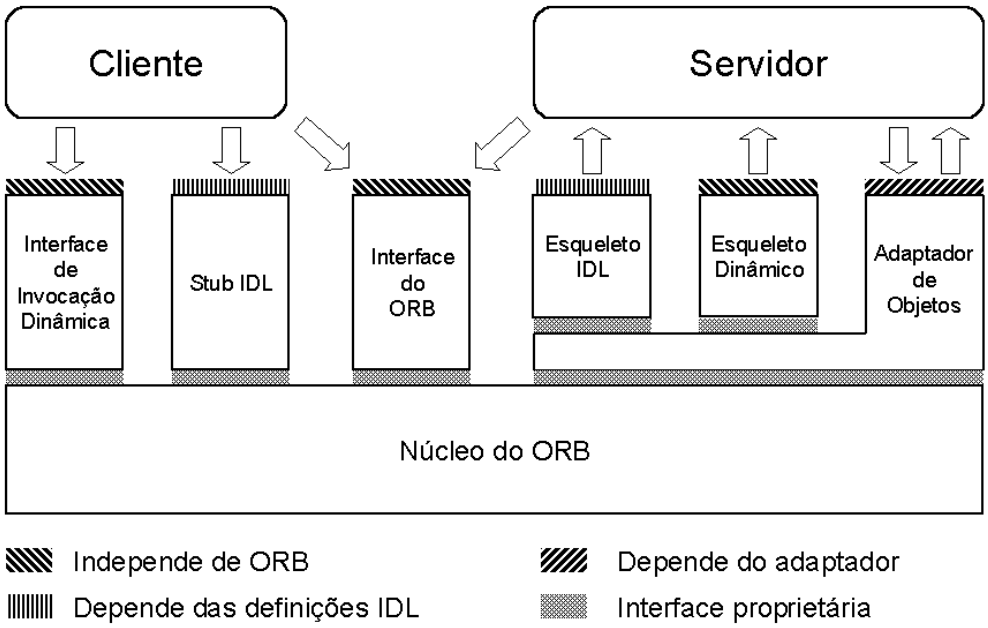

Figura 2.1: Principais elementos de CORBA.

Clientes parecem invocar diretamente métodos de objetos remotos, mas de fato invocam métodos de *stubs* locais. A invocação de um método de um *stub* causa o envio de uma mensagem de requisição para o servidor que implementa o objeto e faz o *stub* aguardar uma mensagem de resposta. O núcleo do ORB encaminha a requisição para o servidor alvo, onde ela passa por um adaptador de objetos e chega a um esqueleto IDL. Este chama um método do servente que encarna o objeto alvo e envia os resultados dessa chamada numa mensagem de resposta, a qual é encaminhada pelo núcleo do ORB até o cliente. O *stub* que aguardava esta resposta extrai dela os resultados da invocação remota de método

e os retorna para o código de aplicação.

Embora sejam perfeitamente adequados para a maioria das situações, *stubs* e esqueletos requerem conhecimento prévio, em tempo de compilação IDL, das interfaces dos objetos. Para os casos relativamente raros em que esse requisito não pode ser satisfeito, CORBA oferece as interfaces de invocação dinâmica (DII) e de esqueleto dinâmico (DSI). Por meio da DII, um "cliente dinâmico" pode invocar métodos de objetos que ele encontrou em tempo de execução, sem ter conhecimento prévio das interfaces desses objetos. Empregando a DSI, um "servidor dinâmico" pode implementar objetos especificados em tempo de execução, sem nenhum conhecimento prévio das interfaces desses objetos.

Clientes dinâmicos tipicamente utilizam o repositório de interfaces, um elemento de CORBA não representado na figura 2.1. Consultando esse repositório, que armazena definições de interfaces IDL, clientes dinâmicos podem obter informações sobre as interfaces de objetos encontrados em tempo de execução.

A figura 2.1 mostra ainda a interface do ORB, cujas operações podem ser chamadas localmente por clientes ou servidores, e um adaptador de objetos, que só existe do lado do servidor. O *Portable Object Adapter* (POA) é o adaptador de objetos padronizado desde a versão 2.2 da especificação CORBA. Ele é responsável pela criação de referências para objetos CORBA, pela ativação de objetos e pelo direcionamento de cada requisição para o servente que implementa o objeto alvo.

O modo de operação de um POA é determinado por um conjunto de políticas (*policies*) especificadas na criação do POA. Selecionando a política adequada pode-se, por exemplo, escolher se as referências para objetos CORBA criadas pelo POA serão válidas somente enquanto estiver ativo o servidor que implementa os objetos (referências transientes), ou se elas permanecerão válidas mesmo depois que esse servidor encerrar sua execução (referências persistentes).

Um cliente que deseje invocar métodos de um objeto CORBA precisa de uma referência para o objeto. Em CORBA, uma *object reference* identifica univocamente o objeto e encapsula todas as informações necessárias para o ORB encaminhar uma requisição a seu destino. CORBA padroniza um formato para transporte e armazenamento de referências, denominado *Interoperable Object Reference* (IOR). Esse formato é usado sempre que uma referência para um objeto CORBA for convertida para cadeia de caracteres, possivelmente para armazenamento num arquivo ou banco de dados, ou quando ela for transmitida como argumento ou resultado de uma invocação remota de método.

2.2 Visão Geral do CCM

O modelo de componentes CORBA (*CORBA Component Model* — CCM) [3] é o *framework* que suporta a definição, geração de código, empacotamento, montagem e implantação de componentes CORBA no contexto da OMA.

As versões iniciais de CORBA preocuparam-se principalmente com a descrição de interfaces. CORBA 2.2, que padronizou o POA, começou a dar mais atenção às implementações dos objetos. A especificação do POA definiu como os serventes que implementam objetos CORBA são construídos e gerenciados pelo ORB. O CCM dá o próximo passo: enriquece IDL com construções para definição de componentes CORBA e introduz uma nova linguagem, a *Component Implementation Definition Language* (CIDL), para descrever a implementação de componentes num nível de detalhamento que permita a geração, a montagem e a implantação de componentes num servidor.

O CCM estende o modelo tradicional de objetos CORBA definindo funcionalidades e serviços que permitem aos desenvolvedores de aplicação implementar, gerenciar, configurar e implantar componentes num ambiente padrão. Tais componentes podem fazer uso dos serviços CORBA integrados ao ambiente padrão oferecido pelo CCM, que incluem os serviços de persistência, segurança, transações e eventos.

O modelo de componentes CORBA abrange as seguintes partes integradas e inter-relacionadas:

- Modelo abstrato de componentes
- *Framework* para implementação de componentes
- Modelo de programação do *container* de componentes
- Arquitetura do *container* de componentes
- Integração com persistência, transações e eventos
- Empacotamento e implantação de componentes
- Modelo de metadados de componentes

2.3 Modelo Abstrato de Componentes

Componente (*component*) é um novo metatipo básico em CORBA. Este metatipo estende e especializa o meta-tipo *object*. Tipos de componentes são definidos em IDL, por meio de construções adicionadas à essa linguagem especificamente para suportar componentes. Um componente é denotado pela referência do componente, a qual é representada por uma referência para um objeto CORBA "normal". A interface deste objeto depende do tipo do componente, em IDL, e é denominada interface equivalente do componente. Componentes são instanciados e "vivem" num *container* para componentes CORBA.

A arquitetura especifica dois níveis de componentes: básico e estendido. Embora ambos sejam gerenciados por *homes* análogas às interfaces `EJBHome` dos componentes EJB, os dois níveis diferem nas funcionalidades que suportam.

O nível básico essencialmente fornece um mecanismo simples de componentização de objetos CORBA, sem adicionar muito ao modelo de programação de CORBA. Componentes no nível básico são implantados em *containers* básicos. Este tipo de componente assemelha-se bastante ao especificado pela arquitetura EJB.

O nível estendido fornece um conjunto de funcionalidades bem mais rico. Componentes deste nível suportam interfaces pré-definidas chamadas portas, que no CCM são usadas para facilitar as interações entre componentes e as interações dos componentes com o seu ambiente. Os vários tipos de portas incluem facetas, receptáculos, fontes e sorvedouros (*sinks*) de eventos. A figura 2.2 ilustra estes tipos de portas.

Facetas. São interfaces com nomes distintos que fornecem as funcionalidades da aplicação aos clientes. Facetas permitem que componentes CORBA suportem interfaces não relacionadas [36], de modo bastante similar aos componentes do *Microsoft Component Object Model* (COM) [4, 5]. Cada faceta incorpora uma visão do componente e corresponde a um papel no qual o cliente pode atuar.

A implementação de uma faceta é completamente encapsulada pelo componente e não é visível ao cliente. Um cliente pode navegar da referência do componente para uma faceta, ou de uma faceta para a referência do componente. Componentes básicos não suportam facetas.

Receptáculos. São usados para especificar as conexões entre componentes e objetos. Além disso, os receptáculos fornecem uma maneira genérica de se conectar o componente a certos tipos de objetos. Podem ser de dois tipos:

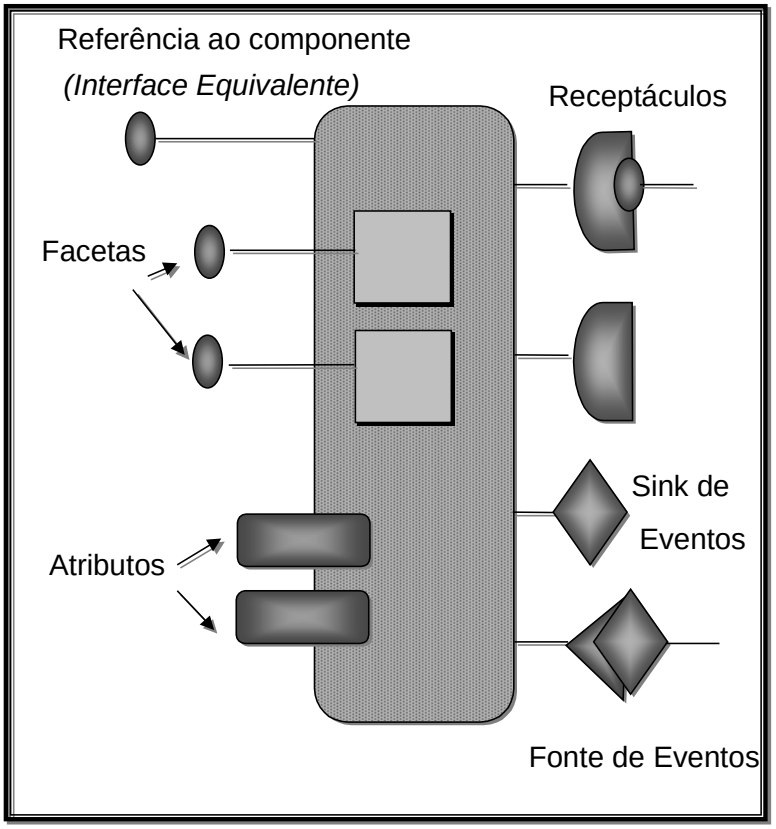

Figura 2.2: Modelo Abstrato de Componente.

Simplex — Gerenciam uma única referência a objeto

Multiplex — Gerenciam múltiplas referências

Fontes e *sinks* de eventos. Componentes podem interagir monitorando o estado de outros componentes e reagindo quando certas alterações de estado ocorrerem. No CCM as fontes e os sorvedouros de eventos pressupõe o emprego de um mecanismo de entrega de eventos que ofereça um subconjunto das interfaces e das características do serviço de notificações CORBA [23], embora o uso deste serviço não seja requerido e os desenvolvedores possam optar por implementar seus próprios mecanismos de notificação. A conexão de várias partes de um sistema por meio de um mecanismo de notificação é um caso de uso importante em sistemas de tempo real orientados a

eventos [12, 24].

As seguintes características são comuns aos componentes básicos e aos estendidos:

***Home*.** Está associada a um componente e gerencia as instâncias desse componente, dentro do escopo do seu *container*. Conceitualmente, a *home* é um gerenciador da extensão (conjunto de instâncias) do tipo definido pelo componente. Ela fornece operações para gerenciamento do ciclo de vida do componente e da associação entre uma chave primária e uma instância do componente.

Chave primária. Pode ser associada a uma instância de um componente, por meio da *home* do componente. No CCM, uma chave primária é um dado, exposto aos clientes do componente, que pode ser usado para identificar instâncias do componente e obter referências para elas.

Atributos e configuração. Os atributos de um componente são valores nomeados, expostos através de métodos acessores e mutadores. Podem ser usados por ferramentas de configuração para estabelecer valores de configuração de um componente. Um componente pode usar atributos para representar estados que dizem respeito ao componente como um todo. Atributos de componente diferem de atributos de interface, pois estes descrevem estados que dizem respeito a interfaces individuais. Os atributos de componente fornecem um mecanismo padrão para se especificar valores iniciais de estados do componente, com o intuito de configurá-lo.

A figura 2.3 ilustra a utilização de um componente CORBA para modelar um distribuidor de bebidas [19]. Os serviços desse componente hipotético são acessíveis por meio da referência do componente ou por meio de referências para facetas do componente. Cada faceta expõe determinado subconjunto das funcionalidades do componente, subconjunto esse voltado para certa categoria de aplicações clientes. A "referência principal" da figura 2.3, sob a legenda "Referência ao Distribuidor", é a referência do componente. Ela referencia um objeto CORBA com a interface equivalente do componente. A figura 2.3 mostra também referências para três facetas: a primeira é utilizada por aplicações que representam os clientes da máquina distribuidora de bebidas, a segunda é utilizada por aplicações que representam os fornecedores de bebidas e a terceira por aplicações de manutenção, executadas pelo reparador do sistema.

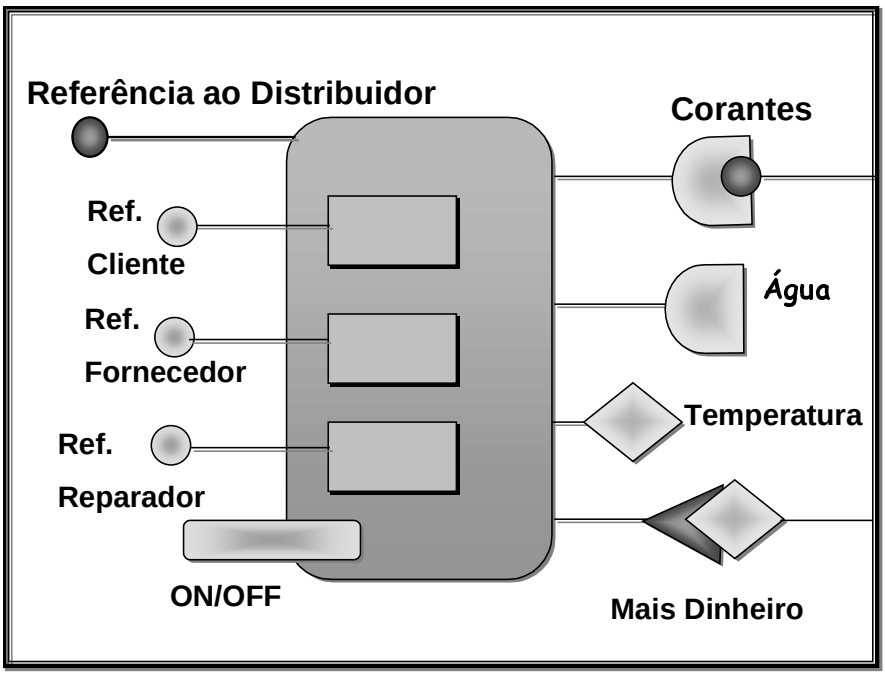

Figura 2.3: Exemplo de componente CCM.

O componente da figura 2.3 tem um atributo de configuração, dois receptáculos, um *event sink* e uma fonte de eventos. O atributo permite que se configure o estado do distribuidor de bebidas: ligado ou desligado. Através dos receptáculos, o distribuidor interage com outros componentes para solicitar fornecimento de corantes e de água. Por meio do *event sink*, ele recebe notificações sobre a temperatura das bebidas. Exemplificando: uma dessas notificações poderia ser gerada quando a temperatura ultrapassar os limites aceitáveis para conservação e/ou comercialização de certa bebida. Por meio da fonte de eventos, o distribuidor gera notificações para os responsáveis pela coleta de dinheiro. Uma dessas notificações poderia ser gerada quando a quantia armazenada na máquina distribuidora exceder determinado valor.

2.4 *Framework* para Implementação de Componentes

Este *framework* define o modelo de programação para a construção de implementações de componentes. Ele é baseado numa linguagem declarativa, a CIDL (*Component Implementation Definition Language*) [37]. A especificação CCM introduziu essa linguagem para descrever implementações de componentes e de *homes* de componentes, bem como esta-

dos abstratos de componentes. A especificação do estado abstrato de um componente tem o objetivo de possibilitar o gerenciamento automático do estado persistente do componente.

Como representado na figura 2.4, compiladores CIDL usam descrições CIDL para gerar esqueletos de implementação com boa parte da funcionalidade básica dos componentes: gerenciamento de ciclo de vida das instâncias, navegação entre facetas, gerenciamento da identidade, ativação e gerenciamento de estado.

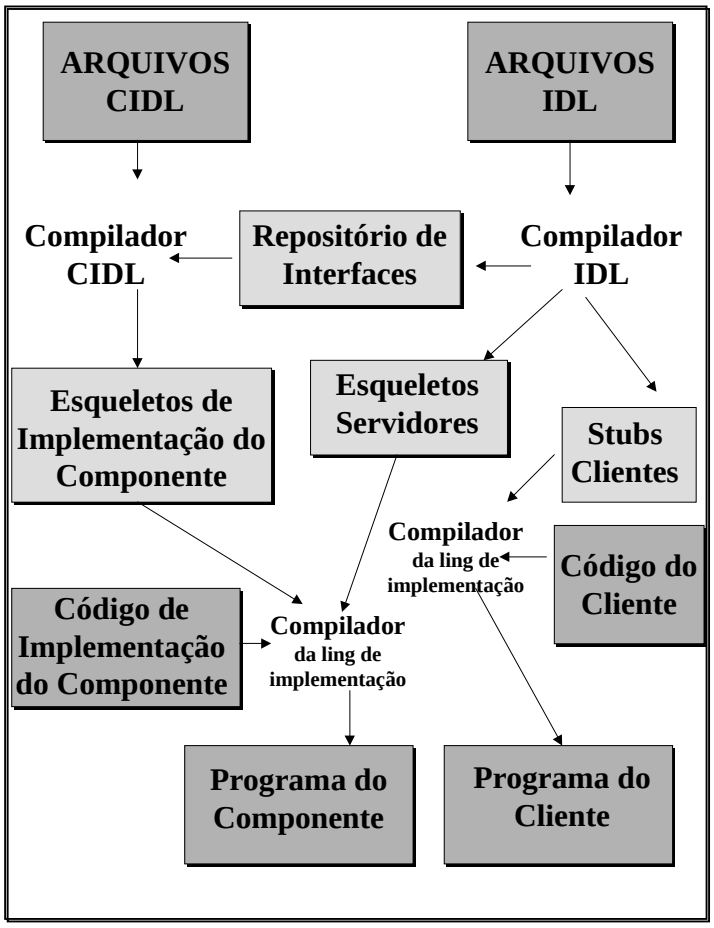

Figura 2.4: Usando IDL e CIDL para implementação do componente.

2.5 Modelo de Programação do *Container*

O *container* é o ambiente de execução do componente CORBA no servidor. O modelo de programação do *container* define um conjunto de APIs e padrões de interação que simplificam a tarefa de desenvolver e configurar aplicações CORBA. Um *container* encapsula a implementação de um componente e usa essas APIs para fornecer um ambiente de execução ao componente. Este ambiente de execução tipicamente oferece as seguintes funcionalidades:

- Ativar e desativar implementações de componentes para preservar recursos limitados de sistema, como memória principal
- Fornecer uma camada de adaptação para os quatro serviços mais comuns: transações, persistência, segurança e notificações.
- Fornecer camadas de adaptação para *callbacks* que o *container* e o ORB usam para notificar o componente de certos eventos, tais como recepção de mensagens enviadas pelos serviços de transações e notificações
- Selecionar políticas do POA para determinar como serão criadas referências de componentes

A figura 2.5 representa a arquitetura do modelo de programação do *container*. Nela aparecem dois conjuntos de interfaces: as interfaces externas do componente, acessíveis aos clientes do componente, e as interfaces envolvidas na interação entre o componente e o *container*, usadas pelo desenvolvedor do componente.

- ***External API*** — É um grupo de interfaces empregado pelos clientes do componente. Ele inclui a interface *home*, cujo papel é análogo ao da interface `EJBHome` de um componente EJB, e mais um conjunto de interfaces com papéis análogos ao da interface `EJBObject` de um componente EJB: a interface equivalente e as facetas do componente.

  A API externa de um componente pode ser de um dos seguintes tipos: sem chave primária e com chave primária. No primeiro caso, a API externa do componente é análoga à de um componente EJB do tipo sessão. No segundo caso, a API externa do componente é análoga à de um componente EJB do tipo entidade.

- ***Container API*** — Inclui as interfaces internas do *container*, que o componente pode invocar para acessar serviços fornecidos pelo *container*, assim como as interfaces *callback* que o *container* pode invocar no componente. Através das interfaces internas o *container* permite que os componentes nele contidos acessem o POA e os serviços CORBA do ambiente CCM.

  Há dois tipos de *container API*, que se diferenciam pelo tipo das referências para componentes que suportam. *Containers* com API tipo sessão suportam referências transientes. *Containers* com API tipo entidade suportam referências persistentes.

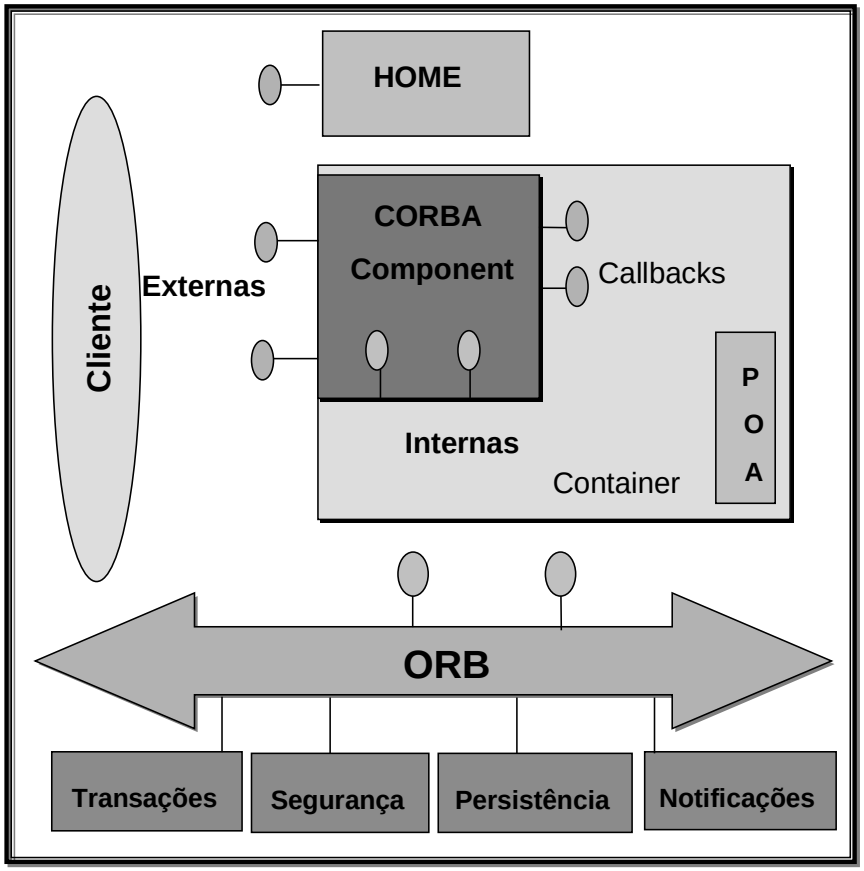

Figura 2.5: Modelo de Programação do *Container*.

Além das APIs externa e do *container*, o modelo de programação do *container* especifica modelos de utilização de CORBA e categorias de componente. O modelo de utilização CORBA define as interações entre o *container* e o ambiente CORBA. Ele é controlado por políticas que selecionam um dentre os possíveis padrões de interação com o POA e os serviços CORBA. O CCM define três modelos de utilização de CORBA:

- **Sem estado** — Emprega referências transientes e registra com o POA um servente que pode implementar todos os objetos de um certo tipo.
- **Conversacional** — Emprega referências transientes e registra com o POA um servente para cada objeto.
- **Durável** — Emprega referências persistentes e registra com o POA um servente para cada objeto.

Uma combinação pré-definida do tipo da API externa com o tipo da API do *container* e com o modelo de utilização de CORBA determina uma categoria de componente, conforme mostra a tabela 2.1.

Tipo da API externa	**Tipo da API do *container***	**Modelo de utilização de CORBA**	**Categoria do componente**
sem chave primária	sessão	sem estado	serviço
sem chave primária	sessão	conversacional	sessão
sem chave primária	entidade	durável	processo
com chave primária	entidade	durável	entidade

Tabela 2.1: Categorias de componente.

As possíveis categorias de componente são:

- **Serviço** — Não possui estado nem identidade, mas possui comportamento. É útil para a modelagem de coisas que fornecem somente a execução independente de uma operação. Exemplo: transação CICS
- **Sessão** — Possui comportamento, estado transiente, e identidade não persistente. Serve para modelar coisas que requerem a manutenção de um estado transiente en-

quanto estiverem sendo utilizadas por um cliente, mas não requerem armazenamento persistente desse estado. Exemplo: um iterador

- **Processo** — Possui comportamento (provavelmente transacional), estado persistente, não visível ao cliente, e identidade persistente, que só será visível ao cliente através de operações explícitas definidas pelo desenvolvedor. Exemplo: proposta de empréstimo

- **Entidade** — Possui comportamento (provavelmente transacional), estado persistente, e identidade visível ao cliente através de chave primária. É útil para modelagem de coisas que representam entidades negociais. Exemplos: clientes, contas

2.6 Arquitetura do *Container*

Esta arquitetura suporta sete categorias distintas de *container*:

- **Serviço** — Gerencia componentes da categoria serviço, sem estado

- **Sessão** — Gerencia componentes da categoria sessão, com estado transiente

- **Processo** — Gerencia componentes da categoria processo, que possuem estado persistente e encapsulam o acesso a dados no servidor

- **Entidade** — Gerencia o componentes da categoria entidade, que possuem estado persistente e permitem que a responsabilidade pelo acesso a seu estado seja compartilhada entre o cliente e o servidor

- **Sessão EJB** — Gerencia *EJB session beans*

- **Entidade EJB** — Gerencia *EJB entity beans*

- **Vazia** — Esta categoria de *container* não fornece gerenciamento automático de componentes, mas apenas disponibiliza às implementações de componentes as interfaces padronizadas por CORBA 3.0 [35].

2.7 Integração com Persistência, Transações e Eventos

Persistência. É suportada pelos *containers* das categorias processo, entidade e entidade EJB. Duas formas de persistência são suportadas por cada categoria:

- Persistência gerenciada pelo *container* — É automaticamente gerado código para carregar e armazenar o estado do componente em tempos determinados, bem como para implementar as operações de fábrica e de busca declaradas na interface *home* do componente.
- Persistência gerenciada pelo componente — O desenvolvedor do componente tem de escrever todo esse código.

Transações. Componentes podem gerenciar suas transações ou delegar este gerenciamento ao *container*.

- Transações gerenciadas pelo *container* — As políticas de transação definidas no descritor do componente (vide seção 2.8) são usadas pelo *container* para fazer as chamadas apropriadas ao serviço de transações CORBA.
- Transações gerenciadas pelo componente — O componente é responsável pela demarcação das transações, mediante chamadas à interface `UserTransaction` do *container* ou ao serviço de transações CORBA.

Eventos. CCM define um modelo simples que suporta dois tipos de publicação de eventos:

- Publicação num canal dedicado
- Publicação num canal compartilhado

O *container* é responsável pelo mapeamento das semânticas desses tipos de publicação de eventos para as interfaces do serviço de notificações CORBA e pela entrega dos eventos através de um canal de notificação.

2.8 Empacotamento e Implantação de Componentes

É usado um vocabulário XML (*Extensible Markup Language*) [39] para descrever os pacotes de *software* e suas dependências. Componentes são empacotados em arquivos zip, cada um deles contendo descritores XML que descrevem seus conteúdos.

Um pacote de componente é o veículo usado para implantação de uma só implementação de componente. Um pacote de montagem de componentes é o veículo usado para implantação de um conjunto de implementações de componentes inter-relacionados. É um padrão para instanciar um conjunto de componentes e *homes*.

Os pacotes de componentes e montagem são implantados em *hosts* numa rede usando uma aplicação ou ferramenta de implantação que instala e ativa as *homes* e as instâncias dos componentes. Essa ferramenta recebe como entrada as informações do descritor do componente ou da montagem, mais informações adicionalmente fornecidas pelo usuário responsável pela implantação. Durante o processo de implantação, com auxílio da ferramenta de implantação, é possível configurar as propriedades de um componente e conectá-lo a outros componentes através de interfaces e portas de eventos.

2.9 Modelo de Metadados de Componentes

Tendo enriquecido a linguagem IDL com construções para definição de componentes, o CCM teve de estender também o repositório de interfaces de CORBA, para que o modelo de informações desse repositório continuasse a ser isomorfo à IDL. O modelo de metadados de componentes, um conjunto de extensões ao modelo de informações do repositório de interfaces, permite o armazenamento de definições de componentes no repositório.

Capítulo 3

O Problema da Integração de BDs Distribuídos

A integração de bancos de dados heterogêneos distribuídos é o processo pelo qual as informações de cada banco de dados participante podem ser conceitualmente integradas para formarem uma multibase de dados [42]. É o processo de se projetar um esquema conceitual global que define a visão corporativa de toda a base de dados. Essa fase de projeto é feita de baixo para cima, ou seja, os bancos de dados individuais já existem com suas bases de dados já estabelecidas, então a definição do esquema conceitual global envolve integrar estas bases isoladas formando uma multibase de dados.

A integração da base de dados poderá ocorrer em duas fases (figura 3.1):

- Tradução
- Integração

3.1 Tradução

Na primeira fase os esquemas das bases de dados componentes são traduzidos para uma representação intermediária comum. Esta representação intermediária deverá ser suficientemente expressiva a ponto de incorporar os conceitos disponíveis em todas as bases de dados que posteriormente deverão ser integradas. O modelo orientado a objeto é geralmente visto como o mais apropriado para este propósito devido as suas características de

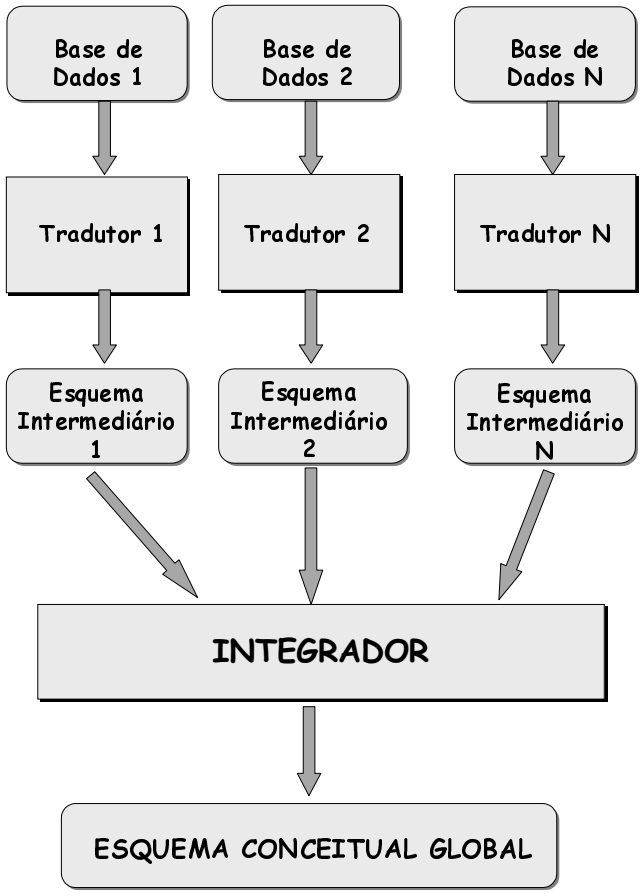

Figura 3.1: Processo de Integração das Bases de Dados.

encapsulamento e abstração, habilitando o desenvolvimento de invólucros que encapsulam um repositório particular e fornecendo uma interface comum ao resto do sistema. O encapsulamento permite que as diferenças de interface e implementação dos Bancos de Dados componentes sejam escondidas, e a abstração (especialização e generalização) que é modelada por subtipos e supertipos permite a criação de tipos que abstraiam as similaridades de entidades armazenadas em diferentes bases de dados. O uso do modelo orientado a objeto engloba o conceito de tradução.

A fase de tradução apenas se faz necessária quando as bases de dados componentes forem heterogêneas e cada esquema local tiver sido definido usando um modelo de dados diferente.

O esquema de tradução é a tarefa de mapear um esquema já existente em outro intermediário. Isto requer a especificação do modelo de dados final integrante da definição do esquema conceitual global. Para isso equivalências entre os conceitos do esquema a ser mapeado e o modelo alvo deverão ser estabelecidas.

3.2 Integração

Na segunda fase, integração, cada esquema intermediário será integrado num esquema conceitual global. A integração é o processo de identificar os componentes de uma base de dados que são relacionados um ao outro, selecionando a melhor representação para o esquema conceitual global, integrando os componentes de cada esquema intermediário. A integração envolve duas sub-tarefas: homogeneização e mesclagem [42].

3.2.1 Homogeneização

Durante esta fase, tanto os problemas de heterogeneidade estrutural como os de semântica são resolvidos.

A heterogeneidade semântica refere-se a diferenças entre as bases de dados no que diz respeito a significado, interpretação e o objetivo de uso dos dados. Os aspectos mais importantes da heterogeneidade semântica acabam se revelando como conflitos de nomes. O problema fundamental quanto a nomes diz respeito ao sinônimo e ao homônimo. Duas entidades idênticas que possuem nomes diferentes são sinônimos, e duas entidades diferentes que possuem nomes idênticos são homônimos. Existem inúmeros métodos para se lidar com conflitos de nomes. Um deles é resolver homônimos prefixando os termos pelo nome do esquema ou modelo. Não é possível resolver sinônimos da mesma forma. A abordagem usual é o uso de ontologias. Uma ontologia é específica a um domínio de aplicação particular e define os termos junto com as semânticas aceitáveis neste domínio. Se todo esquema de base de dados para este domínio usar uma ontologia comum, os conflitos de nome serão naturalmente resolvidos.

Conflitos estruturais ocorrem de quatro maneiras [1]:

Conflitos de tipo — ocorrem quando um mesmo objeto é representado por um atributo num esquema e por uma entidade no outro.

Conflitos de dependência — ocorrem quando modos diferentes de relacionamentos

(um-para-um, muitos-para-muitos) são usados para representar a mesma coisa nos diferentes esquemas.

Conflitos de chave — ocorrem quando diferentes candidatos a chave estão disponíveis e diferentes chaves primárias são selecionadas nos diferentes esquemas.

Conflitos de comportamento — estão diretamente ligados ao mecanismo de modelagem, por exemplo, a remoção de um último ítem de uma base de dados poderá implicar na remoção de toda a entidade que o contém.

A determinação de sinônimos e homônimos assim como a identificação de conflitos estruturais, requer especificação do relacionamento entre os esquemas intermediários. Dois esquemas podem ser relacionados de quatro maneiras:

- Podem ser idênticos
- Um pode ser um subconjunto do outro
- Alguns componentes de um esquema podem ocorrer no outro
- Podem ser completamente diferentes

A determinação do tipo de relacionamento é essencial no projeto de um esquema conceitual global. Porém a identificação desses relacionamentos não pode ser totalmente executada sintaticamente; as semânticas de cada esquema devem ser consideradas. Portanto, a homogeneização requer uma significativa intervenção humana, já que o conhecimento semântico de todos os esquemas intermediários é requerido.

3.2.2 Mesclagem

Todos os esquemas são mesclados num único esquema de base de dados e depois reestruturados para criar o melhor esquema integrado. A mesclagem requer que a informação pertinente aos esquemas participantes seja retida no esquema integrado.

Três dimensões de mesclagem e reestruturação podem ser definidas [1]:

- Completeza
- Minimalidade
- Inteligibilidade

A mesclagem é **completa** se toda a informação de todos os esquemas estiver integrada no esquema comum. Para atingir uma mesclagem completa pode se usar uma técnica que descreve uma entidade em termos de outra, chamada *subsetting*. Os conceitos de generalização e especialização são casos especiais de *subsetting*.

Uma mesclagem é **não mínima** quando informações redundantes de relacionamento são retidas num esquema integrado por causa de uma falha na detecção de conteúdo onde parte de um esquema intermediário pode estar incluído dentro de outro. Esquemas não mínimos podem também resultar do processo de tradução devido a produção de um esquema intermediário que por si só não é mínimo.

Inteligibilidade — é a dimensão final para se determinar o melhor esquema. Uma vez que todos os elementos foram mesclados, a reestruturação pode facilitar um esquema inteligível.

Capítulo 4

O Uso de Arquiteturas de Componentização de Servidores

4.1 Viabilidade

O objetivo deste trabalho é o de estudar a viabilidade do uso dessas arquiteturas como integradores de Bancos de Dados distribuídos e heterogêneos relativos a um mesmo tipo de aplicação, onde as informações contidas nas diversas bases referem-se a um mesmo domínio.

Pretende-se disponibilizar uma visão comum que abranja um conjunto de informações pré-definido e não uma mesclagem completa de todos os esquemas de todas as bases envolvidas. Desta forma, o problema de integração a ser enfocado pelo estudo se restringirá à tradução e definição da visão comum a todas as bases envolvidas.

Numa implementação de um sistema multibase de dados o problema principal a se considerar é a heterogeneidade que existe basicamente em quatro níveis [6, 7, 18]:

- Nível de plataforma — sistemas de bases de dados residem em hardwares diferentes e utilizam sistemas operacionais diversos
- Nível de comunicação — utilização de diferentes protocolos de comunicação
- Nível do sistema de gerenciamento da base de dados — os dados são gerenciados por uma variedade de sistemas gerenciadores de bases de dados baseados em diferentes modelos de dados e linguagens (relacional, orientado a objeto, ...)

- Nível semântico — bases de dados diferentes projetadas independentemente geram conflitos de semântica

O uso de uma arquitetura de componentização de servidores como EJB ou CCM pretende eliminar os problemas nesses vários níveis, uma vez que :

1. Equaliza as plataformas envolvidas
   - **No caso EJB** — através do uso de JVMs (*Java Virtual Machines*) nos diversos *hosts*
   - **No caso CCM** — a arquitetura CORBA fornece transparência de localização e interoperabilidade entre plataformas, que permitem a um cliente acessar um objeto através de sua interface definida em IDL independente do ambiente de software e hardware onde reside o objeto
2. Unifica os protocolos de comunicação
   - **No caso EJB** — é utilizado um dos dois protocolos previstos pela especificação EJB: o *Java Remote Method Protocol* (JRMP) ou o *Internet Inter-ORB Protocol* (IIOP)
   - **No caso CCM** — a comunicação entre objetos em diferentes máquinas utiliza o protocolo IIOP
3. Define uma visão comum das bases envolvidas
   - **No caso EJB** — através do uso de componentes do *entity beans* com persistência gerenciada pelo *container*
   - **No caso CCM** — com componentes CCM CORBA do tipo entidade

   Os dois últimos níveis podem ser atingidos com o mapeamento feito para a base de dados através do uso de persistência gerenciada pelo *container*, ou em casos mais complexos, com a persistência gerenciada pelo próprio componente.

Estas arquiteturas fornecem uma infraestrutura de serviços necessários para o desenvolvimento de um sistema distribuído tais como: gerenciamento de transações, segurança, gerenciamento de estado, ciclo de vida e persistência.

A arquitetura a ser utilizada neste trabalho é a EJB pelo fato da existência de mais de uma implementação que viabilizarão o estudo. Até o momento ainda não existe implementação de servidores que suportem a arquitetura CCM.

4.2 Análise de Soluções

Considerando o problema de integração de bases de dados distribuídas, podemos encontrar registros numa base contendo referências para outros registros em outras bases de toda a rede distribuída. No caso do CCM, estas referências poderiam ser IORs armazenadas como *strings*. No caso do EJB, seriam *handles* serializados de componentes residentes em outros servidores. A situação em que as bases de dados contêm referências remotas é apresentada na figura 4.1.

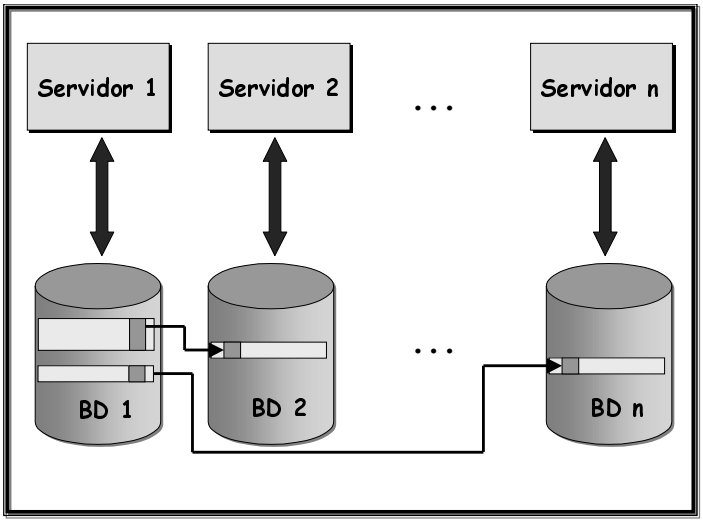

Figura 4.1: Bases contendo referências remotas.

Entretanto o formato dessas referências é integralmente dependente do fornecedor do ORB/CCM ou do fornecedor do servidor EJB. Esta solução, portanto, gera uma rede de interconexões arbitrárias envolvendo referências armazenadas nos diversos bancos de dados, acarretando uma situação de difícil e árduo gerenciamento.

Analisando o processo de alteração ou atualização dos softwares envolvidos, em que as IORs ou *handles* para os objetos em outras bases sofram modificação estrutural, observa-

mos que as atualizações que deverão ser efetuadas por toda a rede distribuída tornam-se, sem dúvida, tarefas bastante complexas e demoradas.

Como exemplo que pode ilustrar a utilização de uma arquitetura desse tipo, podemos citar uma aplicação que envolve componentes que modelam departamentos, funcionários e projetos. É natural pensar que o gerenciamento de projetos possa promover a alocação de funcionários de diversos departamentos distribuídos pela rede. Sendo assim, poderemos encontrar na base de dados projetos que contenham referências diretas para registros de funcionários que estão armazenados em outros servidores remotos.

Uma solução mais adequada nos leva a desejar que um servidor integrante da rede só possa acessar outro servidor remoto através de uma referência *home* de um componente que se encontra implantado neste servidor remoto. Portanto, acessos diretos aos dados armazenados em outro servidor não são permitidos nesta solução.

Para o exemplo dos projetos que foi citado acima, no caso de um projeto referenciar um funcionário remoto, ao invés de utilizar uma referência direta ao funcionário, o servidor deverá obter a referência *home* para o componente que modela funcionários no servidor remoto. De posse da referência *home* poderá utilizar os métodos de busca oferecidos por esta interface para localizar o funcionário específico. Só neste momento será obtida a referência para o funcionário.

Uma alternativa para minimizar o problema decorrente de atualizações de referências, seria o uso de um catálogo central de *homes* para os componentes que estão distribuídos pelas bases envolvidas(figura 4.2).

Para esta solução, em caso de necessidade de atualização de referências, os servidores geradores das referências a serem atualizadas terão apenas que efetuar novos registros no catálogo central de *homes*. Dessa forma, as atualizações das referências serão refletidas para toda a rede distribuída.

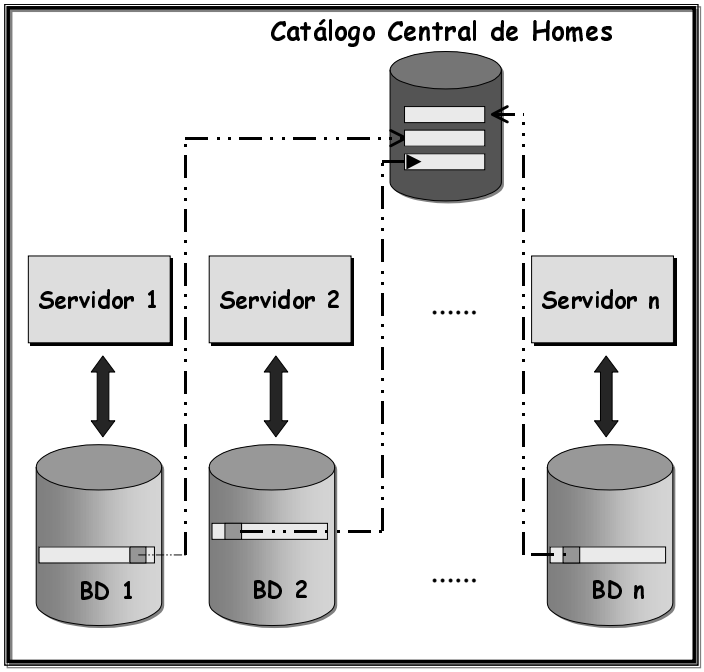

Figura 4.2: Uso de um Catálogo Central de *Homes*.

Capítulo 5

Aplicação: Controle de Estoque Distribuído

A aplicação escolhida para o estudo considera vários servidores EJB distribuídos, cada qual utilizando um Banco de Dados próprio, onde estão armazenados dados do estoque local já existente obedecendo sua própria definição de base de dados. Cada um dos servidores componentes é um gerenciador do estoque local.

Para criar a visão comum serão definidos componentes EJB, a serem implantados nos diversos servidores integrantes, que fornecerão a visão única da base de dados global e implementarão o gerenciador de estoque local. Cada ítem de estoque possuirá os seguintes atributos:

- Código do Ítem
- Descrição
- Quantidade
- Fornecedor

O tratamento da base de dados local estará encapsulado considerando as particularidades do Banco de Dados utilizado.

O sistema deverá ser capaz de executar uma consulta enviada por uma aplicação cliente, considerando a multibase de dados distribuída como sendo única, transparecendo ao cliente um processamento de consulta numa base de dados centralizada.

5.1 Alternativas Consideradas

5.1.1 Central Única de Atendimento

O uso de uma central de atendimento centraliza as requisições enviadas pelas aplicações clientes. O servidor que representa a Central de Atendimento manterá uma base de dados contendo referências RMI para todos os servidores componentes do sistema de estoque. Seu papel é o de centralizar as requisições enviadas pelos clientes e disparar as consultas necessárias aos diversos componentes. Desta forma, os controladores de estoques locais não são habilitados a atender requisições de aplicações clientes.

A Central de Atendimento deverá oferecer as seguintes funcionalidades:

- Inclusão de Novo Controlador de Estoque
- Exclusão de Controlador de Estoque
- Consulta Lista de Ítens de Estoque
- Liberação de Ítem de Estoque
- Inclusão de Ítem de Estoque
- Criação de Ítem de Estoque
- Consulta a Lista de Fornecedores Locais

Cada componente deverá por sua vez oferecer as seguintes funcionalidades:

- Consulta Lista de Ítens de Estoque
- Liberação de Ítem de Estoque
- Inclusão de Ítem de Estoque (aumenta a quantidade)
- Criação de Ítem de Estoque
- Consulta a Lista de Fornecedores Locais

Esta alternativa de usar um componente como Central de Atendimento tem como desvantagem a centralização das conexões, podendo ocasionar uma impossibilidade geral de atendimento de requisições se o servidor sair do ar, ou se estiver indisponível.

5.1.2 Central de Atendimento e Unidades Capazes de Receber Requisições

Uma alternativa para solucionar o problema da alternativa anterior seria a de permitir que cada um dos sistemas componentes possa receber requisições de aplicações clientes.

Se a requisição recebida não puder ser resolvida localmente, ela deverá ser repassada a central de atendimento, que se incumbirá de atender a requisição.

Se não houver um meio de conexão entre os diversos servidores participantes e o servidor central diferente do usado pela aplicação cliente, o mesmo problema de indisponibilidade da central aparece.

Se cada servidor componente mantiver uma lista das referências RMI para outros servidores, não seria preciso repassar uma requisição a Central de Atendimento quando não pudesse ser atendida localmente. O servidor componente pode disparar consultas para os diversos servidores integrantes da lista de posse do componente receptor da requisição. Pode-se optar por uma política de refrescamento da lista periodicamente (diária, semanal, . . .)

5.1.3 Unidades Capazes de Receber Requisições Assumindo Papel de Central

Uma variação da alternativa anterior usando um algoritmo de eleição de líder para estabelecer qual servidor será a Central em caso de indisponibilidade do servidor que exercia este papel.

5.1.4 Unidades Capazes de Receber Requisições — Sem Central de Atendimento

Permitindo a cada servidor componente a capacidade de atender e disparar requisições, uma vez que possua a lista de servidores integrantes, parece ser desnecessário o uso de uma Central de Atendimento. Para descartá-la seria preciso criar mecanismos de divulgação/registro de novos servidores que venham a integrar o sistema, bem como da exclusão.

Esta abordagem parece sobrecarregar cada um dos sistemas gerenciadores de estoque integrantes, pois adicionamos atribuições além do controle local. Além disso, surge a

questão de como um novo servidor obtém a lista de todos os outros integrantes do sistema para que possa se registrar em cada um deles.

5.1.5 Servidor de Diretórios Central

A idéia de se manter uma unidade central volta a ser útil. Cada novo integrante precisaria apenas se conectar ao componente central para se registrar. Este poderia forçar um refrescamento das listas mantidas pelos outros integrantes, ou a própria política de periodicidade de refrescamento poderia ser o bastante. Esta unidade central seria um Servidor de Diretórios Central, contendo as referências RMI para todos os demais estoques integrantes.

Ao invés de utilizarmos uma política de refrescamento, que poderia ser de difícil calibragem quanto a periodicidade a ser adotada, uma opção seria criar mais um componente persistente que represente o catálogo de referências remotas registradas no servidor de diretórios. A cada ativação desse componente, obteríamos uma imagem atualizada de todo o catálogo de referências remotas.

5.2 Arquitetura da Aplicação

5.2.1 Modelagem

A alternativa escolhida para implementação da aplicação de controle de estoque distribuído, que serve como estudo de caso dessa dissertação, foi a que considera o uso de um Servidor de Diretórios Central.

A aplicação foi modelada de modo a apresentar três componentes:

- Estoque (`Estoque`) — representa cada unidade de distribuição do sistema de estoque. A interface *home* do `Estoque` modela o conjunto de ítens de estoques armazenados numa mesma base de dados.
- Gerenciador de Estoque (`GerenciadorDeEstoque`) — responsável pelo gerenciamento das interações entre cada unidade distribuída de estoque e a aplicação cliente
- Lista de Estoques (`ListaDeEstoques`) — representando o catálogo de *homes* de cada componente registrado no servidor de diretórios global

Em cada servidor integrante do sistema distribuído de controle de estoque, são implantados esses três componentes.

A figura 5.1 mostra o diagrama representando a modelagem utilizada para aplicação. Nesta figura a linha tracejada representa a associação entre o componente entity e a base de dados.

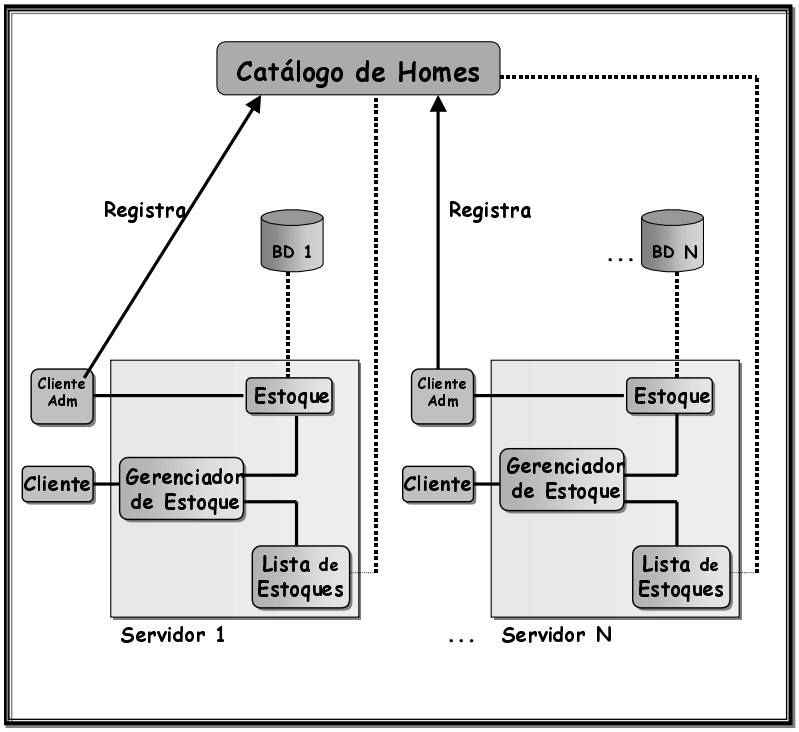

Figura 5.1: Modelagem da Aplicação.

.

5.2.2 O Componente Estoque

O componente `Estoque` modela uma unidade de estoque persistente a qual estão associados os seguintes atributos:

- `codigo` — número que identifica um ítem de estoque;
- `descricao` — descrição textual do ítem de estoque;
- `quantidade` — número que representa a quantidade disponível desse ítem de estoque;
- `fornecedor` — nome do fornecedor desse ítem de estoque.

Como este componente é do tipo Entity, ou seja persistente, seus atributos são armazenados no banco de dados utilizado pelo servidor EJB em questão. Todo o esquema de gerenciamento de persistência é realizado pelo próprio *container*.

Ao implantar o componente `Estoque` num servidor, o implantador da aplicação deverá efetuar o registro do *handle* para a interface *home* desse componente `Estoque` no Servidor de Diretórios Central. Isso pode ser feito através de um cliente administrativo que integra o arquivo ejb-jar montado pelo fornecedor do componente ou pelo montador da aplicação. Esse cliente realiza este registro a partir de uma entrada pré-definida no Servidor de Diretórios. É desta forma que uma nova unidade de estoque integrante da rede distribuída passa a ser conhecida pelas demais.

As interfaces *home* e *remote* do componente `Estoque`, bem como a classe que o implementa, já estão adequadas à solução escolhida para a aplicação e portando não necessitam sofrer alterações.

5.2.3 O Componente Lista de Estoques

Esse componente representa o catálogo de *homes* de componentes do tipo `Estoque` registradas no Servidor de Diretórios Central.

Como o Servidor de Diretórios é um tipo de banco de dados, como descrito na seção 6.3.3, esse componente foi naturalmente modelado como sendo do tipo Entity. Por tratar-se de um Servidor de Diretórios, que possui especificidades que diferem de um banco de dados padrão, a opção escolhida para gerenciamento da persistência dos dados foi a realizada pelo próprio componente. A identificação do tipo de persistência utilizada é feita no descritor de implantação, vide figuras 5.6 e 5.7.

A implementação do método `ejbLoad` se encarrega de buscar o catálogo de *homes* armazenado no Servidor de Diretórios. A sincronização com o Servidor de Diretórios utilizado fica a cargo do *container* que notifica o componente sempre que for invocado algum método negocial.

O componente `ListaDeEstoques` possui um único atributo:

- `aLista` — definida como um *array* de *homes* de componentes do tipo `Estoque`;

No momento de carga desse componente, é feita uma consulta ao Servidor de Diretórios buscando por todas as *homes* gravadas a partir de uma entrada pré-definida uti-

lizada pelo cliente administrativo no momento do registro de uma *home*. Para otimizar a consulta feita no servidor de diretórios, são buscadas as entradas que possuem o atributo `objectClass=javaSerializedObject`. Com o resultado da pesquisa o componente monta um catálogo contendo todas as *homes* registradas no Servidor de Diretórios.

A interface *home* do componente `ListaDeEstoques` está apresentada na figura 5.2.

```
package exemplo.listadeestoques;

import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.FinderException;

public interface ListaDeEstoquesHome extends javax.ejb.EJBHome {

    public ListaDeEstoques create(int id)
        throws CreateException, RemoteException;

    public ListaDeEstoques findByPrimaryKey(ListaDeEstoquesPK pk)
        throws FinderException, RemoteException;

}
```

Figura 5.2: A classe *home* do componente entity `ListaDeEstoques`.

O componente `ListaDeEstoques` oferece um método `get` que retorna a lista das *homes*, e um método `refresh` para permitir um refrescamento forçado da lista.

A interface *remote* do componente `ListaDeEstoques` está apresentada na figura 5.3.

```
package exemplo.listadeestoques;

import exemplo.estoque.EstoqueHome;

public interface ListaDeEstoques extends javax.ejb.EJBObject {

    public EstoqueHome[] get() throws java.rmi.RemoteException;

    void refresh() throws java.rmi.RemoteException;

}
```

Figura 5.3: A classe *remote* do componente entity `ListaDeEstoques`.

O método `ejbLoad`, pertencente a classe que implementa o componente denominado `ListaDeEstoques`, é o responsável por acessar o Servidor de Diretórios e construir toda a lista de *homes*. Este método está apresentado nas figuras 5.4 e 5.5.

```
public class ListaDeEstoquesBean implements javax.ejb.EntityBean {

    EstoqueHome[] aLista;
    ...
    public void ejbLoad(){
        java.util.Hashtable env = new java.util.Hashtable();
        env.put(Context.INITIAL_CONTEXT_FACTORY,
            "com.sun.jndi.ldap.LdapCtxFactory");
        env.put(Context.PROVIDER_URL,
            "ldap://localhost:389/o=EJB, dc=ime, dc=usp, dc=br");
        try {
            javax.naming.ldap.LdapContext ctx =
                new javax.naming.ldap.InitialLdapContext(env, null);
            javax.naming.NamingEnumeration ne =
                ctx.search("","objectClass=javaSerializedObject",null);
            java.util.List colDeHomesEstoquesExternos =
                new java.util.ArrayList();
            while (ne.hasMore()) {
                javax.naming.directory.SearchResult r =
                    (javax.naming.directory.SearchResult)ne.next();
                javax.naming.directory.Attributes attrs =
                    r.getAttributes();
                javax.naming.directory.Attribute attr =
                    attrs.get("javaSerializedData");
                java.io.ObjectInputStream ois =
                    new java.io.ObjectInputStream(
                       new java.io.ByteArrayInputStream(
                           (byte[]) attr.get()));
                Object obj = ois.readObject();
                if (obj instanceof javax.ejb.HomeHandle) {
                    javax.ejb.HomeHandle homeHandle =
                        (javax.ejb.HomeHandle)obj;
                    EstoqueHome home = (EstoqueHome)
                        javax.rmi.PortableRemoteObject.narrow(
                            homeHandle.getEJBHome(), EstoqueHome.class);
                    colDeHomesEstoquesExternos.add(home);
                }
            }
```

Figura 5.4: O método `ejbLoad` da classe que implementa a `ListaDeEstoques`.

```
            ne.close();
            ctx.close();
            aLista = new EstoqueHome[colDeHomesEstoquesExternos.size()];
            aLista = (EstoqueHome[])
                colDeHomesEstoquesExternos.toArray(new EstoqueHome[0]);

        } catch (Exception e) {
            throw new javax.ejb.EJBException(e);
        }
    }

 ...

}
```

Figura 5.5: O método `ejbLoad` da classe que implementa a `ListaDeEstoques` (cont.).

O descritor de implantação do componente `ListaDeEstoques` está apresentado nas figuras 5.6 e 5.7.

```
<?xml version="1.0"?>

<!DOCTYPE ejb-jar PUBLIC
 "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 1.1//EN"
 "http://java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd">

<ejb-jar>
 <enterprise-beans>
   <entity>
      <description>
            Lista de Estoques eh um entity enterprise bean que
            representa a lista de todas as homes de estoques
            registradas no LDAP.
      </description>
```

Figura 5.6: O descritor de implantação do componente `ListaDeEstoques`.

```
      <ejb-name>ListaDeEstoques</ejb-name>
      <home>exemplo.listadeestoques.ListaDeEstoquesHome</home>
      <remote>exemplo.listadeestoques.ListaDeEstoques</remote>
      <ejb-class>exemplo.listadeestoques.ListaDeEstoquesBean</ejb-class>
      <persistence-type>Bean</persistence-type>
      <prim-key-class>exemplo.listadeestoques.ListaDeEstoquesPK
                    </prim-key-class>
      <reentrant>False</reentrant>
  </entity>
</enterprise-beans>

<assembly-descriptor>
  <security-role>
     <description>
        Permissão de acesso total ao componente.
      </description>
    <role-name>everyone</role-name>
  </security-role>

  <method-permission>
    <role-name>everyone</role-name>
    <method>
        <ejb-name>ListaDeEstoques</ejb-name>
        <method-name>*</method-name>
    </method>
  </method-permission>

  <container-transaction>
    <method>
       <ejb-name>ListaDeEstoques</ejb-name>
       <method-name>*</method-name>
    </method>
    <trans-attribute>NotSupported</trans-attribute>
  </container-transaction>

 </assembly-descriptor>
</ejb-jar>
```

Figura 5.7: O descritor de implantação do componente `ListaDeEstoques`(cont.).

5.2.4 O Componente Gerenciador de Estoque

O componente `GerenciadorDeEstoque`, do tipo session, é o responsável pela interação do cliente com a base de estoques. Possui os seguintes atributos:

- *home* — contém a referência para a interface *home* do componente `Estoque` local;
- homeDaLista — contém a referência para a interface *home* do componente denominado `ListaDeEstoques`.

No momento em que o componente `GerenciadorDeEstoque` é criado, as referências para o componente `Estoque` local e para o componente `ListaDeEstoques` são obtidas permitindo que, a partir desse momento, as interações com esses componentes possam ocorrer.

Através das funcionalidades oferecidas pelo `GerenciadorDeEstoque` o cliente poderá:

1. Criar ítem de estoque — cria um novo ítem de estoque na base local;
2. Consultar ítem de estoque — consulta todos os atributos de um ítem de estoque da base local;
3. Remover ítem de estoque — remove um ítem de estoque da base local;
4. Incrementar quantidade de um ítem de estoque — adiciona unidade(s) a um ítem de estoque da base local;
5. Liberar quantidade de um ítem de estoque — decrementa unidade(s) de um ítem de estoque da base local;
6. Lista todos os ítens da base de estoque — lista todos os ítens de estoque da base local com seus atributos;
7. Listar fornecedores de determinado ítem de estoque — lista todos os fornecedores de ítens de estoque, da base local, que possuam uma determinada descrição;
8. Listar todos os fornecedores da base de estoque — lista toda a relação de fornecedores de ítens de estoque da base local;
9. Obter a identificação da base de estoque — obtém a identificação do estoque local;
10. Listar todos os ítens da base de estoque distribuída — lista todos os ítens de estoque, com seus atributos, de toda a base de estoque distribuída.

Note que o componente `GerenciadorDeEstoque` é um cliente do componente `Estoque`, para oferecer as funcionalidades de números 1 a 9, e do componente `ListaDeEstoques` para a funcionalidade de número 10.

A interface *remote* do componente `GerenciadorDeEstoque` deve ser complementada para contemplar o método que lista todos os ítens de estoque de toda a base distribuída, como apresentado na figura 5.8.

```
...

public interface GerenciadorDeEstoque extends javax.ejb.EJBObject {

    ...
    public ItemEstoque [] listaItensTodosEstoquesExternos()
       throws java.rmi.RemoteException;

}
```

Figura 5.8: Complementação da interface `Remote` do `GerenciadorDeEstoque`.

Da mesma forma, a classe que implementa o componente `GerenciadorDeEstoque` deverá ser complementada como mostram as figuras 5.9 e 5.10.

```
...

public class GerenciadorEstoqueBean implements javax.ejb.SessionBean {
    ...

    public ItemEstoque [] listaItensTodosEstoquesExternos()
        throws java.rmi.RemoteException{

        try {
            java.util.List ie = new java.util.ArrayList();

            ListaDeEstoquesPK pk = new ListaDeEstoquesPK();
            pk.codigo = 0;
            ListaDeEstoques listaDeEstoques =
                       homeDaLista.findByPrimaryKey(pk);
            EstoqueHome [] eh = listaDeEstoques.get();
```

Figura 5.9: Complementação da classe que implementa o `GerenciadorDeEstoque`.

```
            for (int j=0; j < eh.length; j++){
                EstoqueHome estHome = (EstoqueHome) eh[j];
                java.util.Collection coll = estHome.findAll();
                java.util.Iterator it = coll.iterator();
                while (it.hasNext()){
                    Estoque est = ((Estoque)it.next());
                    ie.add(new ItemEstoque(est.getCodigo(),
                                           est.getDescricao(),
                                           est.getFornecedor(),
                                           est.getIdEstoque(),
                                           est.getQuantidade()));
                }
            }
            ItemEstoque iRes[] = new ItemEstoque[ie.size()];
            iRes = (ItemEstoque[]) ie.toArray(new ItemEstoque[0]);
            return iRes;
        } catch (FinderException fe){
            throw new javax.ejb.EJBException(fe);
    }

}
```

Figura 5.10: Compl. da classe que implementa o `GerenciadorDeEstoque`(cont).

A figura 5.11 mostra o trecho contendo as entradas `ejb-ref` que permitem ao componente GerenciadorDeEstoque obter as informações necessárias para interagir com os componentes `Estoque` e `ListaDeEstoques`. O descritor de implantação do componente `GerenciadorDeEstoque` mostrado nas figuras 1.19 e 1.20, deve ser complementado para contemplar a interação com o componente `ListaDeEstoques`, e assim, ficar adequado à solução adotada para a aplicação.

5.3 Tecnologias Utilizadas

5.3.1 Servidor EJB

No início dos trabalhos foram avaliados os seguintes servidores:

- Orion Server
- Weblogic 5.0

```
...

  <ejb-ref>
    <ejb-ref-name>ejb/RefComponenteEstoque</ejb-ref-name>
    <ejb-ref-type>Entity</ejb-ref-type>
    <home>exemplo.estoque.EstoqueHome</home>
    <remote>exemplo.estoque.Estoque</remote>
  </ejb-ref>

  <ejb-ref>
    <ejb-ref-name>ejb/RefComponenteListaDeEstoques</ejb-ref-name>
    <ejb-ref-type>Entity</ejb-ref-type>
    <home>exemplo.listadeestoques.ListaDeEstoquesHome</home>
    <remote>exemplo.listadeestoques.ListaDeEstoques</remote>
  </ejb-ref>
 </session>

...
```

Figura 5.11: Complementação do descritor de implantação do `GerenciadorDeEstoque`.

- Weblogic 6.0
- JBoss 2.2

Devido ao fato de oferecer abertura dos fontes, simplicidade de uso, estabilidade da versão e clareza da documentação resolvemos utilizar o servidor JBoss. Para implementação da aplicação controle de estoque foi empregado o servidor JBoss 2.4.3.

JBoss é uma implementação da especificação EJB 1.1 (e de parte da especificação 2.0), ou seja é um servidor e *container* para *Enterprise JavaBeans*. Requer quantidade mínima de memória e espaço em disco, rodando numa máquina que possua 64Mb de RAM. A implementação de referência de servidor EJB, fornecida pela SUN com o J2SDK *Enterprise Edition* (J2EE), requer 128Mb de RAM e 31Mb de espaço em disco. O JBoss oferece um servidor de banco de dados SQL, *built-in*, para manipular componentes persistentes, que é iniciado automaticamente com o servidor JBoss.

Uma das maiores vantagens oferecidas pelo JBoss é o *hot deployment*. Isto significa que para implantar um componente no servidor basta copiar seu arquivo tipo JAR para

o diretório de implantação do servidor. Se isto for feito quando o componente já estiver implantado, JBoss automaticamente desinstala o componente implantado e implanta a nova versão.

JBoss é gratuito mesmo para uso comercial.

5.3.2 Bancos de Dados

Foram utilizados dois bancos de dados distintos a saber:

Postgresql — É um DBMS relacional orientado a objeto que suporta construções SQL como, *subselects*, transações e tipos e funções definidos pelo usuário. É um incremento no sistema de gerenciamento de base de dados POSTGRES. PostgreSQL é gratuito e toda a sua codificação está disponível.

Hypersonic — É um DBMS relacional escrito em Java, com um *driver* JDBC, suportando um subconjunto do SQL ANSI-92. Oferece um *database engine* pequeno, cerca de 100k, rápido e com opção de armazenamento de tabelas em memória ou em disco. Oferecido gratuitamente com documentação e código incluídos.

Além do uso de bancos de dados distintos, foram utilizados mapeamentos diferentes dos atributos do componente Estoque para cada uma das bases de dados envolvidas.

5.3.3 Serviço de Diretórios

Um serviço de diretórios permite a localização de usuários, recursos, serviços e informações numa rede. Pode-se distinguir um serviço de diretórios de um serviço de nomes pela habilidade nos serviços de busca e recuperação de informação nomeada. Embora possa ser considerado como um tipo de banco de dados, não possui algumas das funcionalidades típicas de um banco de dados, como transações atômicas e consultas declarativas [14]. Os acessos para leitura do diretório são tipicamente mais comuns que os de atualização.

O serviço de diretórios escolhido para utilização neste trabalho foi o LDAP, que fornece uma maneira de nomear, gerenciar e acessar coleções de pares de atributo-valor. Constitui um padrão Internet aberto e foi produzido pelo IETF, *Internet Engineering Task Force*, grupo responsável também pelo TCP/IP, SMTP, SNMP, HTTP e outros protocolos. O LDAP oferece as seguintes funcionalidades [16]:

- Acesso e Atualização — permite buscas complexas considerando porções do diretório
- Mecanismo de Autenticação — possibilita a segurança da informação através de controles de acessos
- Baseado em Modo Estensível da Informação — permite que o tipo da informação armazenada no diretório seja estendida dinamicamente.

Características do serviço de Diretórios LDAP. Do ponto de vista conceitual, o LDAP é composto pelos seguintes elementos:

1. Modelo de Informação — Todos os dados no diretório são armazenados em "entradas", sendo que cada entrada pertence a pelo menos uma "*object class*". Cada entrada possui uma coleção de atributos, que mantém as informações da entrada. As *object class* definem os atributos que uma entrada pode ou deve conter.
2. Modelo de Nomes — Especifica como a informação é organizada e referenciada num diretório LDAP. Os nomes LDAP, seqüências de atributos de entradas, dão hierárquicos.
3. Modelo Funcional — Define como clientes podem acessar, manipular e alterar as informações num diretório. O LDAP oferece as seguintes operações básica: adicionar, apagar, modificar, associar (*bind*), dissociar (*unbind*), buscar, comparar, renomear e abandonar.
4. Modelo de Segurança — Define como a informação LDAP é protegida contra acessos não autorizados.
5. Esquema LDAP — Define que dados podem ser armazenados num determinado servidor e como esses dados se relacionam com objetos do mundo real. O esquema pode usar *object classes* padronizadas ou novas *object classes* criadas para atender a requisitos específicos de uma instalação.
6. Protocolo LDAP — Especifica as interações entre clientes e servidores, e define os formatos das mensagens de requisição e resposta.
7. Interface de Programação — API que oferece um conjunto de funções e definições padronizadas. É utilizada pelos programas que acessam o diretório.

8. Formato de troca de Dados — O LDAP *Data Interchange Format* (LDIF) é um formato textual para representação de entradas e alterações nessas entradas. Facilita a importação/exportação de dados de/para diretórios X.500 ou proprietários.

Implementação Utilizada. Várias empresas oferecem produtos LDAP, incluindo Microsoft, Netscape e Novell. A *OpenLDAP Foundation* mantém e disponibiliza uma implementação *open source* do LDAP, baseada na implementação original da Universidade de Michigan. Para aplicação de estudo de caso referente a esta dissertação foi utilizado o serviço de diretórios OpenLDAP versão 2.0.11.

OpenLDAP. O projeto OpenLDAP é um esforço de colaboração para o desenvolvimento de um conjunto robusto de aplicações e ferramentas de desenvolvimento LDAP. O projeto é gerenciado por uma comunidade de voluntários que utilizam a internet para se comunicarem, planejarem e desenvolverem. O desenvolvimento do OpenLDAP prossegue acompanhando a evolução dos padrões da IETF. Inclui os seguintes módulos:

- slapd — servidor LDAP *stand-alone*;
- slurpd — servidor LDAP *stand-alone*;
- bibliotecas — implementando o protocolo LDAP;
- utilitários, ferramentas, e exemplos de clientes LDAP.

Capítulo 6

Trabalhos Relacionados

6.1 O Sistema MIND

O MIND (*METU Interoperable DBMS*) [6, 18] é um sistema *multidatabase* construído pela *Middle East Technical University* (METU) em colaboração com a Universidade de Alberta. No MIND, cada banco de dados integrante do *multidatabase* é encapsulado num objeto CORBA denominado *Local Database Agent* (LDA), cuja interface IDL representa um "banco de dados genérico". O MIND fornece implementações de LDAs para diversos bancos de dados, como Oracle, Sybase e Adabas. Cada LDA é responsável por:

- oferecer uma interface para o banco de dados local,
- representar, num modelo de dados canônico, os esquemas exportados pelo sistema de banco de dados local, e
- traduzir para a linguagem de consulta local as consultas expressas numa linguagem de consulta global.

Os LDAs são gerenciados pelo nível global do MIND, que contém um gerenciador de transações e um processador de consultas distribuídas, bem como um integrador de esquemas.

6.2 O Sistema HEROS

O HEROS (*Heterogeneous Object System*) [33] é um sistema de bancos de dados heterogêneos desenvolvido na PUC do Rio de Janeiro. Seus componentes são encapsulados por objetos CORBA. O HEROS possui um esquema global que integra os esquemas dos bancos de dados componentes e define mecanismos para gerência de transações globais.

O esquema de exportação de cada sistema integrante contém a descrição dos objetos locais que farão parte do multibanco de dados, expressa usando os conceitos do modelo de dados global. A versão do HEROS descrita em [33] faz a tradução dos modelos de dados locais para o modelo global, mas deixa a cargo do administrador da federação a detecção e a resolução manual de possíveis heterogeneidades semânticas. O sistema oferece mecanismos para tratamento de certas heterogeneidades, incluindo sinônimos, homônimos, replicação de objetos e heterogeneidade de unidade de medida.

6.3 O Sistema Garlic

O *Garlic* [26, 27], desenvolvido no *IBM Almaden Research Center*, é um sistema que oferece uma visão integrada de dados heterogêneos provenientes de múltiplas fontes, as quais podem incluir sistemas legados. A integração não requer alterações na forma de armazenamento nem tampouco nos locais onde os dados estão armazenados.

O núcleo do Garlic é um processador de consultas globais que interage com um conjunto de *wrappers* que encapsulam as fontes de dados. A arquitetura do Garlic, que aparece na figura 6.1, prevê um *wrapper* para cada fonte de dados. Um repositório de metadados contém a descrição do esquema global, mas não mantém informações sobre os recursos de processamento de consultas das várias fontes de dados.

O processador de consultas desenvolve planos para decomposição de consultas globais em consultas que possam ser interpretadas por cada repositório. Uma das responsabilidades de um *wrapper* é participar do planejamento de consultas, que tem como objetivo enumerar planos alternativos para responder a uma determinada consulta. Nessa fase, o processador de consultas identifica o maior fragmento da consulta que envolve determinada fonte de dados, e o envia ao *wrapper* dessa fonte de dados. O *wrapper* retorna uma lista (possivelmente vazia) de planos de execução que implementam total ou parcialmente o fragmento recebido, bem como estimativas do custo da execução desses planos. O processador de

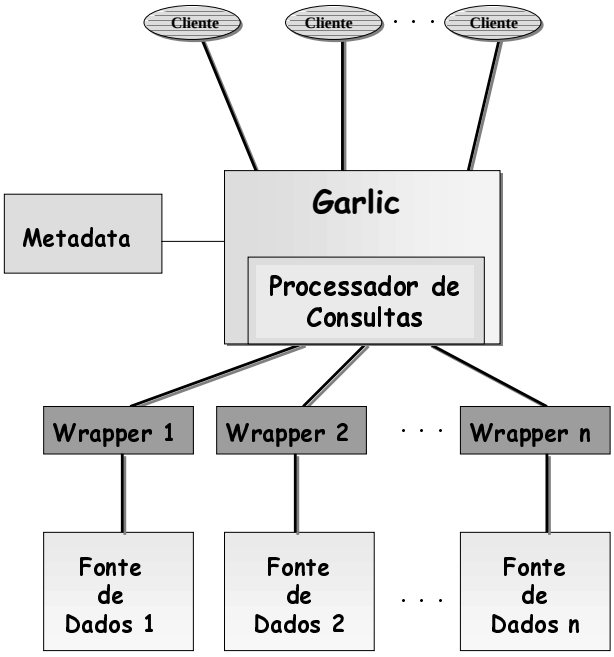

Figura 6.1: Arquitetura Garlic.

consultas inclui os planos de execução retornados pelos vários *wrappers* no conjunto de possíveis planos que ele está gerando, adicionando planos que realizam no próprio Garlic as partes da consulta que não são tratadas pelas fontes de dados. Um otimizador de consultas procura então escolher o plano de execução mais eficiente, baseado nas estimativas de custo fornecidas pelos *wrappers*.

Um *wrapper* fornece quatro serviços:

1. Modela como objetos os dados provenientes da fonte que ele encapsula;
2. Trata invocações de métodos sobre esses objetos;
3. Participa do planejamento de consultas;
4. Participa da execução das consultas.

Essa arquitetura de *wrappers* pretende oferecer:

- Baixo custo inicial de desenvolvimento — É obrigatória apenas a implementação de um pequeno conjunto de serviços.

- Capacidade de evolução — Um *wrapper*, inicialmente, poderá apenas modelar o conteúdo de um repositório como objeto. Havendo necessidade de melhorar o desempenho, as capacidades de processamento de consulta do repositório poderão ser exploradas posteriormente.
- Flexibilidade — *Wrappers* para novas fontes de dados poderão ser integrados a um *multidatabase* Garlic já existente.
- Otimização de consultas — Como participante do planejamento de consultas, um *wrapper* utilizará as informações que detém sobre as facilidades de busca e consulta oferecidas pelo repositório ao qual está associado determinando dinamicamente o quanto de uma consulta poderá ser tratada no repositório.

6.4 O Sistema DISCO

A arquitetura do sistema DISCO (*Distributed Information Search Component*) [32] aparece na figura 6.2.

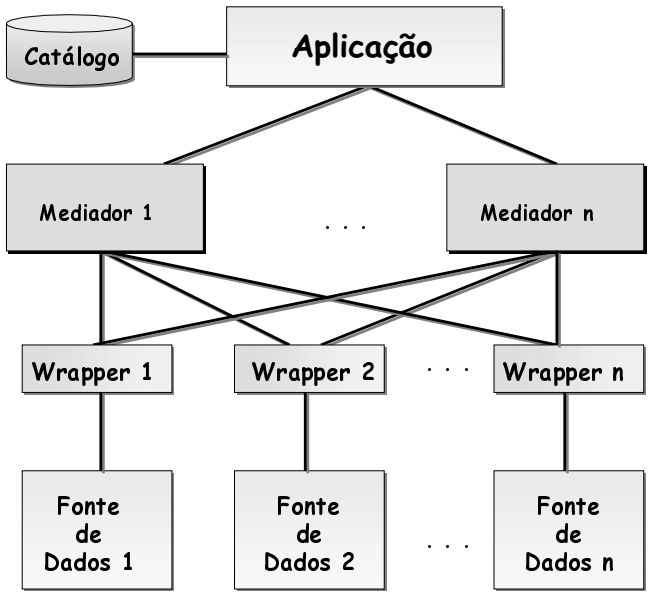

Figura 6.2: Arquitetura DISCO.

O ambiente DISCO tem os seguintes componentes:

- Aplicação — Os usuários interagem com a aplicação, que acessa uma representação uniforme das fontes de dados distribuídas.
- Catálogo — Fornece uma visão geral de todo o sistema.
- Mediadores — Oferecem como funcionalidade o acesso uniforme às diversas fontes de dados. Os mediadores recebem consultas provenientes da aplicação e as transformam em subconsultas que serão posteriormente distribuídas pelas fontes de dados através dos wrappers.
- *wrappers* — São responsáveis pela tradução das subconsultas para as linguagens relativas às fontes de dados às quais estão associados. Têm também a responsabilidade de participar da fase de planejamento de consultas, informando aos mediadores os recursos de processamento de consultas da sua fonte de dados.

A arquitetura de *wrappers* do DISCO tem pontos em comum com a do Garlic. Em ambos os sistemas os *wrappers* participam ativamente do processamento de consultas, retirando do nível superior a responsabilidade de manter informações sobre os recursos de processamento de consulta das fontes de dados. Uma diferença importante, entretanto, está no modo como o *wrapper* repassa tais informações ao nível superior. No caso do Garlic, o *wrapper* recebe uma árvore sintática representando o fragmento da consulta que envolve sua fonte de dados e retorna um conjunto de planos de execução e seus respectivos custos. Já o DISCO emprega um modelo de processamento de consultas baseado numa álgebra de operadores lógicos relacionais, tais como *select*, *project* e *scan*. O *wrapper* conhece o subconjunto desses operadores lógicos que é suportado por sua fonte de dados. Na fase de processamento de consultas ele repassa essas informações ao mediador, na forma de uma gramática que descreve as expressões lógicas suportadas.

Capítulo 7

Considerações Finais

7.1 Dificuldades Encontradas

A avaliação dos servidores EJB disponíveis foi uma tarefa que ultrapassou as expectativas estabelecidas inicialmente. Esperava-se encontrar os produtos em um grau maior de desenvolvimento.

O primeiro servidor a ser analisado foi o Orion [15], por ser de uso gratuito. Porém a documentação oferecida não acompanhava as versões liberadas do servidor, dificultando bastante a familiarização com o produto.

Após a experiência mal sucedida com o Orion, tentamos utilizar versões de experimentação (*trial versions*) do produto oferecido pela BEA Systems denominado Weblogic Server [2]. Embora a documentação existente tenha sido satisfatória, tivemos problemas para utilizar as ferramentas visuais para a implantação de componentes. As ferramentas oferecidas com as versões de experimentação pareciam não estar totalmente desenvolvidas, tendo apresentado problemas de funcionamento principalmente na geração dos arquivos descritores de implantação e mapeamento dos campos persistentes. Duas versões foram testadas, a 5.1 e a 6.0. Além dos problemas encontrados, essas versões de experimentação do Weblogic podem ser usadas gratuitamente por apenas 30 dias.

Por último foi testado o servidor JBoss [17] que nos surpreendeu pela facilidade de utilização e boa documentação. Tivemos apenas dois problemas. Um deles foi na serialização de *handle* para componente, que não armazenava a referência do servidor de origem. Desta forma, ao recuperar a referência serializada de um componente e tentar utilizá-la para in-

vocar um método negocial ocorria um erro, pois a referência não continha informações relativas a que servidor remoto tal referência pertencia. Após contato com a equipe do JBoss, foi liberada uma nova versão que solucionou o problema.

O segundo problema estava também relacionado com a utilização de referências remotas serializadas e era causado por um erro no JBoss, no código que tratava a referência. Como detínhamos os fontes, corrigimos esse erro e geramos uma versão que funcionava corretamente. Não foi necessário contatar a equipe do JBoss, pois verificamos que o código fonte da próxima versão (o qual é acessível via Internet) já continha essa correção.

A experiência com o JBoss foi bastante positiva e durante os trabalhos foi possível perceber o seu rápido desenvolvimento e alto nível de comprometimento com a especificação EJB.

7.2 Comparação com Outros Trabalhos

Embora nossa proposta de utilização de arquiteturas de componentização de servidores como integradoras de BDs distribuídos não objetive oferecer uma mesclagem completa de todos os esquemas das bases integrantes, restringindo-se a disponibilizar uma visão comum pré-definida das informações, podemos identificar similaridades com os trabalhos discutidos no capítulo 6.

Tanto o MIND como o HEROS são sistemas *multidatabase* baseados num padrão de *middleware* para objetos distribuídos (CORBA). Nossa proposta é a integração de BDs distribuídos através arquiteturas de componentização que, por sua vez, se baseiam em padrões de *middleware* para objetos distribuídos.

Os componentes tipo entidade oferecidos pelas arquiteturas de componentização de servidores têm muito em comum com os *wrappers* dos sistemas Garlic e DISCO. A semelhança com os *wrappers* do Garlic é maior, pois estes modelam como objetos os dados dos repositórios subjacentes e oferecem métodos de acesso (*get*/*set*) aos atributos dos objetos. Estes métodos de acesso são inteiramente análogos aos oferecidos por componentes tipo entidade.

No caso do Garlic, entretanto, existe a necessidade efetiva da implementação do código que encapsula o conteúdo de uma fonte de dados, associando identidade aos objetos e permitindo ao Garlic atualizar e recuperar dados do repositório. As arquiteturas de componentização de servidores oferecem a geração automática do código que gerencia a persistência

dos dados, isentando o desenvolvedor do componente da tarefa de implementar a interação com a base de dados. Evidentemente essa geração automática não é feita para toda e qualquer repositório de dados, mas apenas para aqueles suportados pelo servidor EJB ou CCM no modo *container-managed persistence*. Caso se use persistência gerenciada pelo componente, o desenvolvedor terá a possibilidade de lidar com um repositório de dados arbitrário. Neste caso, a tarefa do desenvolvedor será similar à de desenvolvimento de um *Garlic wrapper* básico, que não expõe os recursos de processamento de consultas do repositório subjacente.

Um dos pontos mais atraentes oferecidos pelas arquiteturas de componentização de servidores é, sem dúvida, a infraestrutura disponibilizada, que permite que o servidor gerencie automaticamente serviços como transações, persistência, distribuição, estado e ciclo de vida. Os sistemas descritos no capítulo 6 não fornecem todos esses serviços, sobrecarregando o desenvolvedor de aplicações com aspectos que fogem a sua área de especialidade. Por outro lado, nosso trabalho não ataca a questão da otimização de consultas envolvendo dados de vários repositórios, tratada pelos sistemas Garlic e DISCO.

7.3 Trabalhos Futuros

Hoje a utilização de arquiteturas de componentização de servidores possibilita a geração automática de código similar ao dos *Garlic wrappers* básicos. Por não atacarem a questão da otimização de consultas distribuídas, tais arquiteturas não comportam algo similar a um *Garlic wrapper* que exponha os recursos de processamento de consultas do seu repositório.

Notar que a versão 2.0 da especificação EJB possibilita a definição de relacionamentos entre componentes do tipo entidade, porém essa funcionalidade está restrita a componentes implantados em um mesmo servidor. Dessa forma, EJB 2.0 evita tratar consultas formuladas em EJB QL que envolvam dados gerenciados por múltiplos servidores EJB. Mesmo assim, é possível formular consultas EJB QL envolvendo dados de várias fontes gerenciadas pelo mesmo servidor EJB. Técnicas de otimização de consultas distribuídas análogas às utilizadas pelo Garlic poderiam ser empregadas neste caso. Esta é uma linha interessante para investigação futura.

Uma linha adicional para investigação futura é o tratamento de consultas envolvendo múltiplos servidores EJB. Isso envolveria uma extensão à arquitetura EJB. Tal extensão poderia, por exemplo, adicionar à interface `EJBHome` métodos que seriam chamados na fase

de planejamento de consultas. Uma extensão similar poderia ser proposta para o CCM.

Referências Bibliográficas

[1] C. Batini, M. Lenzirini, and S. B. Navathe. A comparative analysis of methodologies for database schema integration. *ACM Computing Surveys*, 18(4):323–364, December 1986.

[2] BEA Systems. Weblogic Application Server On-line Documentation, 2001. `http://www.weblogic.com/`.

[3] BEA Systems. *CORBA Component Model Joint Revised Submission*. Object Management Group, OMG Document orbos/99-07-01, July 1999.

[4] Don Box. *Essential COM*. Addison-Wesley, Reading, MA, 1997.

[5] David Chappell. *Unsderstanding ActiveX and OLE*. Microsoft Press, 1996.

[6] A. Dogac, C. Dengi, E. Kilic, G. Ozcan, S. Nural, C. Evrendileck, U. Halici, B. Arpinar, P. Koksal, N. Kesim, and S. Mancuhan. METU Interoperable Database System. *SGMOD Record*, 24(1):9–14, 1995.

[7] Asuman Dogac, Cevdet Dengi, and M. Tamer Özsu. Distributed object computing platforms. *Communications of the ACM*, 41(9):95–103, 1998.

[8] David Flanagan, Jim Farley, William Crawford, and Kris Magnusson. *Java Enterprise in a Nutshell — A Desktop Quick Reference*. O' Reilly & Associates, first edition, September 1999.

[9] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Francisco, 1993.

[10] Object Management Group. Discussion of the Object Management Architecture, January 1997. `ftp://ftp.omg.org/pub/docs/formal/00-06-41.pdf`.

[11] Object Management Group. The Common Object Request Broker: Architecture and Specification — Revision 2.6, December 2001.
`ftp://ftp.omg.org/pub/docs/formal/01-12-01.pdf`.

[12] T. H. Harrison, D. L. Levine, and D. C. Schmidt. The Design and Performance of a Hard Real-Time Object Event Service. In *Proceedings of the 1997 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOSPLA '97)*, Atlanta, GA, October 1997. ACM.

[13] Alex Homer and David Sussman. *Professional MTS and MSMQ programming with VB and ASP*. Wrox, 1998.

[14] Tim Howes and Mark Smith. *Programming Directory-Enabled Applications with Lightweight Directory Access Protocol*. Macmillan Technical Publishing, Indianapolis, Indiana, 1997.

[15] Ironflare AB. Orion Application Server Web Site, 2001.
`http://www.orionserver.com/`.

[16] Gustavo Scalco Isquierdo. Integração do Serviço de Diretórios LDAP com o Serviço de Nomes CORBA. Master's thesis, IME - USP, 2001.

[17] JBoss Group. JBoss Web Site, 2001. `http://www.jboss.org/`.

[18] Ebru Kilic, Gokhan Ozcan, Cevdet Dengi, Nihan Kesim, Pinar Koksal, and Asuman Dogac. Experiences in Using CORBA for a Multidatabase Implementation. In *6th International Conference and Workshop on Database and Expert Systems Applications (DEXA'95)*, pages 223–230, 1995.

[19] R. Marvie, P. Merle, J.-M. Geib, and C. Gransart. Des objets aux composants corba, première Étude et expérimentations avec corbascript. In *Proceedings of the 12th International Conference on Software and Systems Engineering and their Applications (ICSSEA'99)*, Paris, France, December 2000.

[20] Microsoft. *Microsoft Transaction Server*. `www.microsoft.com/com/tech/MTS.asp`.

[21] Richard Monson-Haefel. *Enterprise JavaBeans*. O'Reilly, 1999.

[22] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, 2.3.1 edition, 1999.

[23] Object Management Group. *Notification Service Specification*. OMG Document telecom/99-07-01 ed., July 1999.

[24] C. O'Ryan, D. C. Schmidt, and D. Levine. Applying a Scalable CORBA Events Service to Large-scale Distributed Interactive Simulations. In *Proceedings of the 5th Workshop on Object-Oriented Real-time Dependable Systems, (Monterey,CA)*. IEEE, June 1999.

[25] Francisco C. R. Reverbel and Arthur B. Maccabe. Making CORBA Objects Persistent: the Object Database Adapter Approach. In *Proceedings of the 3th Conference on Object-Oriented Technologies and Systems (COOTS'97)*, Portland, Oregon, June 1997. USENIX.

[26] Mary Tork Roth and Peter Schwarz. A Wrapper Architecture for Legacy Data Sources. Technical Report RJ10077, IBM, 1997.

[27] Mary Tork Roth and Peter Schwarz. Don't Scrap It, Wrap It! — A Wrapper Architecture for Legacy Data Sources. In *Proceedings of the 23rd VLDB Conference*, pages 266–275, Athens, Greece, 1997.

[28] Sun Microsystems. *Enterprise JavaBeans Specification*. Sun Microsystems, Version 2.0 Final Release, August 14, 2001.

[29] Sun Microsystems. *Enterprise JavaBeans Specification*. Sun Microsystems, Version 1.1 Public Draft 3, June 28, 1999.

[30] The Object Management Group. *OMG's site for CORBA amd UML Success Stories*. `http://www.corba.org/`.

[31] Anne Thomas. Enterprise JavaBeans Tecnology, Server Component Model for the Java Platform. Technical report, Patricia Seybold Group, December 1998. Prepared for Sun Microsystems, Inc.

[32] Anthony Tomasic, Louiqa Raschid, and Patrick Valduriez. Scaling Heterogeneous Databases and the Design of Disco. In *Proceedings of the 16th International Conference on Distributed Computing Systems*, pages 808–823, Hong Kong, May 1996. IEEE Computer Society.

[33] Elvira Maria Antunes Uchoa, Sérgio Lifschitz, and Rubens Nascimento Melo. HEROS: un Sistema de Bancos de Dados Heterogêneos Usando CORBA. Monografias em Ciência da Computação 42/97, Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática, 1997.

[34] Steve Vinoski. CORBA: Integrating Diverse Applications Within Heterogeneous Environments. *IEEE Communications*, 14(2), February 1997.

[35] Steve Vinoski. New Features for CORBA 3.0. *Communications of the ACM*, 41(10):44–52, October 1998.

[36] Nambor Wang, David Levine, and Doug Schmidt. *Optimizing the CORBA Component Model for High-performance and Real-Time Applications*.

[37] Nambor Wang, Douglas Schmidt, and Carlos O'Ryan. CORBA:Overview of the CORBA Component Model. submitted as a chapter to the book *Component-Based Software Engineering: Putting the Pieces Together*, edited by George Heineman and Bill Councill, to be published by Addison-Wesley, 1999.

[38] A. Wollrath, R. Riggs, and J. Waldo. A Distributed Object Model for the Java System. *USENIX Computing Systems*, vol. 9, 1996.

[39] World Wide Web Consortium. *Extensible Markup Language (XML)*. `http://www.w3.org/tr/1998/REC-xml-19980210`.

[40] L. L. Yan, M. T. Ozsu, and L. Liu. Accessing heterogeneous data through homogenization and integration mediators. In *2nd Int. Conf. on Cooperative Information Systems(CoopIS'97)*, pages 130–139, June 1997.

[41] M. T. Özsu, U. Dayal, and P. Valduriez. *Distributed Object Management*. Morgan Kaufmann, 1994.

[42] M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed DataBase Systems*. Prentice Hall, second edition, 1999.