UNIVERSIDADE ESTADUAL DE CAMPINAS

FACULDADE DE ENGENHARIA ELÉTRICA E DE COMPUTAÇÃO

Serviços de gerenciamento ODP utilizando a arquitetura CORBA

Douglas Evangelista de Araújo
Orientador: **Prof. Dr. Eleri Cardozo**

Dissertação de Mestrado apresentada à Faculdade de Engenharia Elétrica como parte dos requisitos exigidos para obtenção do título de Mestre em Engenharia Elétrica, Área de concentração: Automação Industrial

Campinas - SP Brasil
1996

FICHA CATALOGRÁFICA ELABORADA PELA
BIBLIOTECA DA ÁREA DE ENGENHARIA - BAE - UNICAMP

Ar15s

Araújo, Douglas Evangelista de
Serviços de gerenciamento ODP utilizando a arquitetura CORBA / Douglas Evangelista de Araújo.-- Campinas, SP: [s.n.], 1996.

Orientador: Eleri Cardozo.
Dissertação (mestrado) - Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica e de Computação.

1. Processamento eletrônico de dados - Processamento distribuído. 2. Programação orientada a objetos (Computação). 3. Redes de computação. I. Cardozo, Eleri. II. Universidade Estadual de Campinas. Faculdade de Engenharia Elétrica e de Computação. III. Título.

Aos meus pais, Francisco e Darcy, e aos meus irmãos, José Carlos e Emerson, que sempre acreditaram e torceram por mim.

Agradecimentos

À minha família, mais uma vez e sempre, por tudo que fizeram e que ainda farão por mim.

Ao prof. Eleri pela orientação e o incentivo à realização do trabalho.

Aos meus amigos de república, pelo companheirismo, pela tolerância e principalmente, pelos momentos de alegria.

Aos amigos do LCA, com quem o convívio é sempre um aprendizado.

Aos velhos amigos de Uberlândia, que nunca esqueceram de mim, mesmo quando ausente.

Aos novos amigos de Campinas, que me apoiaram nessa jornada.

Ao CNPq pelo apoio financeiro.

" E agora José?
A luz apagou, o
vento parou e o
povo sumiu. E agora
José? "

Carlos Drumond de Andrade.

UNIVERSIDADE ESTADUAL DE CAMPINAS

FACULDADE DE ENGENHARIA ELÉTRICA E DE COMPUTAÇÃO

Serviços de gerenciamento ODP utilizando a arquitetura CORBA

Douglas Evangelista de Araújo
Orientador: **Prof. Dr. Eleri Cardozo**

Dissertação de Mestrado

Campinas - SP Brasil
1996

Sumário

Lista de figuras

Lista de abreviações

CORBA - Common Object Request Broker;

gcl - gerente de cluster;

gcp - gerente de cápsula;

Gdbm - GNU database manager;

gob - gerente de objeto;

IDL - Interface Definition Language;

IPC - Inter Process Call;

ISO - International Standard Organization;

OBE - Objeto Básico de Engenharia;

ODP - Open Distributed Processing;

OMG - Object Management Group;

OMT - Object Modeling Technique;

ORB - Object Request Broker;

RM-ODP - Reference Model for Open Distributed Processing;

sbd - servidor de base de dados;

str - servidor de transações;

Windows-NT - Windows - New Technology;

Resumo

Araújo, D. E., Serviços de gerenciamento ODP utilizando a arquitetura CORBA. Campinas: DCA/FEEC/UNICAMP, 1996. (Dissertação de Mestrado)

O crescente desenvolvimento tecnológico e o aumento da demanda por sistemas e aplicações distribuídas levaram à definição de um padrão para processamento distribuído aberto, RM-ODP (*Reference Model for Open Distributed Processing*), e ao surgimento de plataformas comerciais capazes de integrar aplicações que estão espalhadas em um ambiente heterogêneo.
O trabalho representa uma proposta para implementação de alguns dos serviços básicos do RM-ODP utilizando uma arquitetura de integração baseada na especificação CORBA (*Common Object Request Broker Architecture*). Entre os serviços oferecidos estão o suporte e o gerenciamento de objetos, *cluster*, cápsula e nó.
A principal contribuição do trabalho está na identificação de alguns requisitos tecnológicos necessários à construção de sistemas distribuídos abertos e heterogêneos, e no mapeamento em termos de implementação dos conceitos propostos pelo modelo ODP do ponto de vista computacional para o de engenharia.

Palavras chaves: Processamento eletrônico de dados - Processamento distribuído. Programação orientada a objetos (Computação). Redes de computação.

Abstract

Araújo, D. E., ODP management services using the CORBA architecture. Campinas: DCA/FEEC/UNICAMP, 1996. (Masters)

The increasing technological development and the demand for distributed systems and applications resulted in the definition of ODP (Open Distributed Processing) standard and and architectures able to integrate applications in a heterogeneous environment.
This work aims to provide some basic services described in the ODP standard using a distributed computing plataform based on the CORBA specification. Management services of the level of objects, cluster, capsule and node are offered.
The main contribution of this work is the identification of important technological requirements to construct open distributed systems, as well as mapping of some concepts from the ODP computational viewpoint to engineering viewpoint.

keywords: Distributed processing. Object-oriented programming. Computer networks.

Capítulo 1

Introdução

1.1 Motivação

Com o crescente desenvolvimento tecnológico dos processadores, interconexões e redes locais, aliado à demanda dos usuários por recursos mais rápidos e seguros, a computação distribuída vem tornando-se cada vez mais uma alternativa aos tradicionais modelos centralizados.

Entre os benefícios que podem ser alcançados em um ambiente distribuído, estão um aumento da confiabilidade e da tolerância a falhas, uma maior proximidade entre usuários e recursos oferecidos, compartilhamento, integração e particionamento de recursos entre diferentes sistemas, localizações e domínios de gerenciamento, melhoria de desempenho como resultado do processamento paralelo da informação, redução do custo na adição de novos módulos ao sistema e gerenciamento descentralizado dos recursos. Contudo, para que essas vantagens se tornem realidade, é fundamental que exista transparência. Os usuários não podem perder tempo em localizar recursos distribuídos, nem os desenvolvedores devem necessitar codificar em seus aplicativos as localizações dos recursos na rede.

Um outro estímulo para a construção de sistemas distribuídos foi a definição de um padrão estabelecido pela ISO para processamento distribuído e o surgimento de plataformas comerciais capazes de integrar aplicações que estão espalhadas em um ambiente heterogêneo.

Sendo assim, a principal motivação para este trabalho é analisar, segundo o padrão mencionado, os requisitos para a construção de sistemas distribuídos, propondo sugestões para preenchê-los.

1.2 Objetivos

Quando se fala em padronização para o processamento distribuído aberto, deseja-se permitir que os benefícios da distribuição da informação sejam atingidos em um ambiente de recursos heterogêneos e múltiplos domínios oganizacionais. Assim, torna-se necessário prover componentes e funções que acomodem as dificuldades de se projetar e desenvolver sistemas distribuídos.

Tais sistemas possuem algumas características inerentes:

- Componentes remotos: componentes podem estar espalhados geograficamente; interações podem ser locais ou remotas.
- Concorrência: qualquer componente pode executar em paralelo com outros componentes.
- Ausência de um estado global: o estado global não pode ser determinado precisamente.
- Falhas parciais: qualquer componente pode falhar independente de outro componente.
- Modo assíncrono: atividades de comunicação e processamento não são dirigidas por um relógio global.
- Heterogeneidade: Não existe garantia que componentes são construídos usando a mesma tecnologia e o conjunto das várias tecnologias certamente mudará com o tempo. A heterogeneidade aparece em muitos lugares: hardware, sistemas operacionais, redes e protocolos de comunicação, linguagens de programação, aplicações etc.
- Autonomia: um sistema distribuído pode se espalhar sobre várias autoridades de gerenciamento ou controle. O grau de autonomia especifica a extensão para qual recursos de processamento e dispositivos associados (impressoras, dispositivos de armazenamento, display gráficos, dispositivos de áudio etc) estão sob o controle de entidades separadas.
- Evolução: durante o seu período de trabalho, um sistema distribuído aberto geralmente tem que enfrentar várias mudanças que são motivadas pelo progresso tecnológico, que permitem um desempenho melhor a um custo menor, por decisões estratégicas sobre novos objetivos e por novos tipos de aplicações.
- Mobilidade: as fontes de informação, nós de processamento e usuários podem ser fisicamente móveis. Programas e dados podem também ser movidos entre nós, por exemplo, para otimizar o desempenho.

Para lidar com essas características citadas anteriormente, o sistema distribuído deve dar suporte as seguintes propriedades:

- Abertura: permitir portabilidade (isto é, executar diferentes componentes sobre diferentes nós de processamento sem modificação), interatividade (ou seja, suportar interações entre componentes que possivelmente estão em sistemas diferentes), e inserção e remoção de componentes a qualquer instante de tempo.
- Integração: incorporar vários sistemas e recursos num todo sem que isso represente um custo alto para o desenvolvimento, o que incluiria integração de sistemas com arquiteturas diferentes, recursos e desempenho diferentes (ajudando assim a tratar a heterogeneidade).
- Flexibilidade: suportar a evolução do sistema. Um sistema distribuído aberto deve ser capaz de tratar mudanças em tempo de execução, por exemplo, deve ser capaz de ser reconfigurado dinamicamente para acomodar fatores externos. Esta propriedade ajuda a lidar com a mobilidade.
- Modularidade: permitir que partes de um sistema sejam autônomas, mas interrelacionadas, fornecendo assim, a base para a flexibilidade.

- Federação: combinar sistemas de diferentes domínios técnicos ou administrativos para atingir um objetivo comum.
- Gerenciamento: permitir monitoramento, controle e gerenciamento de recursos do sistema a fim de suportar configuração e políticas de qualidade de serviço.
- Provisão de qualidade de serviço: assegurar um conjunto de requisitos de qualidade no desempenho do sistema. A propriedade cobre disponibilidade e confiabilidade de recursos remotos e interações, e tolerância a falhas.
- Segurança: assegura que as facilidades do sistema e dados estão protegidos contra acesso não autorizado.
- Transparência: mascarar os detalhes e as diferenças das aplicações nos mecanismos usados para superar problemas causados pela distribuição.

Entretanto, o próprio padrão estabelecido pela ISO reconhece que não é simples suportar uma infra-estrutura que atenda a todas as propriedades exigidas por um sistema distribuído, e coloca que é necessário selecionar os componentes que forneçam os requisitos para tipos particulares de aplicação, descrever estes componentes, e mostrar como eles se combinam. Paralelamente, o desenvolvimento de plataformas (ambientes para construção e integração de aplicações distribuídas) que deixam transparentes questões como comunicação e localização de recursos, abre novas perspectivas na busca de soluções para os problemas que ocorrem num ambiente distribuído e heterogêneo. Logo, o objetivo deste trabalho é, utilizando as ferramentas disponíveis, implementar funcionalidades que ajudem a encontrar tais soluções e que garantam algumas das propriedades citadas anteriormente.

1.3 Contribuições

O trabalho apresenta uma descrição de alguns pontos relevantes do modelo ODP (Open Distributed Processing) e mostra aspectos importantes relacionados com o funcionamento de uma plataforma comercial para construção de aplicações distribuídas. Contudo, a principal contribuição está na identificação de requisitos tecnológicos necessários para se mapear as abstrações de alguns conceitos do modelo proposto pela ISO para o mundo real.

1.4 Apresentação

Este trabalho está estruturado em seis capítulos. O primeiro capítulo é esta introdução. O capítulo 2 corresponde a uma revisão da literatura, apresentando conceitos importantes do ODP necessários na construção de aplicações distribuídas, e uma descrição da especificação CORBA (Common Object Request Broker Architecture).
O capítulo 3 mostra uma proposta para implementação das funções de gerenciamento definidas no ODP utilizando uma implementação CORBA.

O capítulo 4 aborda de maneira mais aprofundada aspectos ligados à implementação em si dos serviços de gerenciamento e das funções de suporte construídos, descreve as limitações encontradas nas ferramentas utilizadas e apresenta as soluções práticas do ponto de vista computacional adotadas.
O capítulo 5 descreve uma aplicação utilizando os serviços construídos.
O capítulo 6 apresenta as conclusões sobre o trabalho, comentando as dificuldades encontradas e sugerindo alguns trabalhos futuros.
O Apêndice A descreve todas as interfaces IDL utilizadas na implementação.

Capítulo 2

Revisão da literatura

2.1 Processamento distribuído aberto - ODP (Open Distributed Processing)

O modelo de referência do processamento distribuído aberto da ISO (RM-ODP) consiste de:

- um modelo de referência [RM-ODPa]: contém conceitos chaves e algumas linhas gerais da arquitetura ODP. Explica também como o modelo deve ser entendido e aplicado pelos usuários. Esta parte não é normalizada;
- um modelo descritivo [RM-ODPb]: apresenta a definição de conceitos, estruturação e notação para uma descrição normalizada de sistemas de processamento distribuído. Este é apenas um nível de detalhe para dar suporte ao modelo prescritivo e para estabelecer requisitos para novas técnicas de especificação.
- modelo prescritivo [RM-ODPc]: mostra as especificações das características que qualificam um sistema de processamento distribuído como aberto. Nele estão os requisitos do padrão ODP. Esta parte é normalizada.
- semântica arquitetônica [RM-ODPd]: refere-se à formalização do conceito de modelamento ODP definido no modelo descritivo. A formalização é atingida interpretando cada conceito em termos de construção de diferentes técnicas de descrição formal padronizadas. Esta parte é normalizada.

2.1.1 Pontos de vista

A especificação completa de qualquer sistema distribuído não trivial envolve uma quantidade muito grande de informação. Ao invés de lidar com toda a complexidade de um sistema distribuído, pode-se analisá-lo sob pontos de vista diferentes, cada qual

refletindo um conjunto de atributos distintos e representando um nível de abstração diferente do sistema original, sem a necessidade de um grande modelo que descreva todos estes níveis em detalhes [Gunji95].

No ODP, uma separação de contextos é estabelecida pela identificação de cinco pontos de vista, cada um associado com uma linguagem que expressa conceitos e regras relevantes aos contextos.

Os pontos de vista definidos no modelo ODP (RM-ODP) são:

- ponto de vista empresarial, que enfoca as atividades de negócios do sistema especificado;
- ponto de vista de informação, que descreve a informação a ser armazenada e processada no sistema;
- ponto de vista computacional, que contém a descrição do sistema como um conjunto de objetos que se interagem através de suas interfaces, possibilitando a distribuição do sistema;
- ponto de vista de engenharia, que define os mecanismos de suporte a distribuição do sistema;
- ponto de vista tecnológico, que detalha os componentes com os quais o sistema distribuído é construído.

A figura 2-1 mostra como os pontos de vista do RM-ODP devem ser utilizados nas diversas fases do ciclo de vida de um sistema ODP:

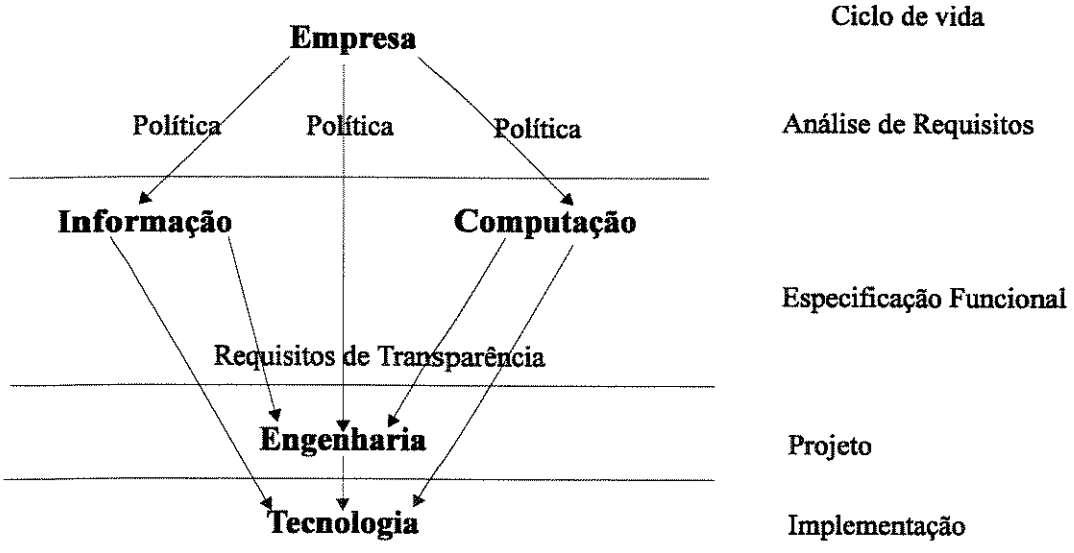

Figura 2-1 Uso de pontos de vista no ciclo de vida

2.1.2 Transparências de distribuição do ODP

Transparência é um importante requisito para o usuário final em sistemas distribuídos. O RM-ODP define um conjunto de transparências de distribuição necessárias para se implementar um sistema transparente do ponto de vista do usuário.

Transparência de acesso
Mascara diferenças na representação de dados e nos mecanismos de invocação para habilitar interação entre objetos.

Transparência de falha
Mascara a falha e a possível recuperação de outros objetos, permitindo tolerância a falhas.

Transparência de locação ou localização
Mascara a localização do objeto que está sendo acessado.

Transparência de migração
Mascara a capacidade de mudar a localização de um objeto. Freqüentemente utilizada para atingir balanceamento de carga e para reduzir latência.

Transparência de relocação
Esconde de um objeto computacional o fato de que as interfaces mudaram sua localização.

Transparência de replicação
Esconde o uso de um grupo de objetos compatíveis para suportar uma interface. É freqüentemente utilizada para garantir desempenho e disponibilidade.

Transparência de persistência
Esconde de um objeto a desativação e a reativação de outros objetos.

Transparência de transação
Esconde de um usuário a existência de concorrência no sistema. Mascara os detalhes de sincronização e os mecanismos de ordenação que garantem que o estado do sistema seja consistente.

2.1.3 O Modelo Prescritivo

O modelo prescritivo define linguagens para cada ponto de vista e então identifica as funções chaves relacionadas com as linguagens. As definições são expressas em termos presentes no modelo descritivo [RM-ODPb]. As informações acerca da linguagem de engenharia serão discutidas de forma mais completa por ser de maior interesse para o propósito do trabalho.

2.1.3.1 Linguagem Empresarial

A linguagem empresarial introduz os conceitos básicos necessários para representar um sistema ODP no contexto da organização que ele opera. O propósito de uma especificação empresarial é expressar os objetivos, as restrições e as políticas ao sistema de interesse [Griethuysen92, Domville92].

2.1.3.2 Linguagem de Informação

A linguagem de informação pode ser representada em termos de objetos de informação e seus relacionamentos, onde objetos de informação são abstrações de entidades que ocorrem no mundo real, no sistema ODP, ou em outros pontos de vista. Contém conceitos que habilitam a especificação do significado da informação manipulada e armazenada dentro de um sistema ODP.

2.1.3.3 Linguagem Computacional

A linguagem computacional define a decomposição funcional de um sistema ODP em objetos que interagem através de suas interfaces.

A especificação computacional provê a base para decisões sobre como distribuir os trabalhos a serem feitos, já que as interfaces podem ser desenvolvidas independentemente, assumindo mecanismos de comunicações definidos na especificação de engenharia.

A linguagem computacional está associada com um modelo de objetos que define a forma da interface que um objeto pode ter, como construir uma interface e as formas de interação que podem envolvê-la.

2.1.3.4 Linguagem de Engenharia

A linguagem de engenharia focaliza o modo como a interação de objetos é atingida e os recursos necessários para isso. Assim, na linguagem de engenharia o objetivo principal é suportar as interações individuais entre objetos computacionais, que sob o ponto de vista de engenharia, podem ser vistos como objetos básicos de engenharia.

Além dos conceitos do ITU-TVRec. X.902 - ISO/IEC 10746-2 [RM-ODPb], são definidos também na linguagem de engenharia:

- objeto básico de engenharia: um objeto de engenharia que exige o suporte de uma infra-estrutura distribuída;

- domínio de gerenciamento de referência de interface de engenharia: um conjunto de nós formando um domínio de nomes a ser utilizado pelas referências de interfaces de engenharia;
- política de gerenciamento de referência de interface de engenharia: um conjunto de permissões e proibições que governam a federação de domínios de gerenciamento de referência de interface de engenharia;
- *template*: estrutura utilizada para descrever objetos.
- *template* de *cluster*: um *template* de objeto para uma configuração de objetos e qualquer atividade exigida para instanciá-los e estabelecer *bindings*;
- *checkpoint*: um template de objeto derivado do estado e da estrutura de um objeto de engenharia que pode ser usado para instanciar outro objeto de engenharia, consistente com estado do objeto original no momento em que foi requerido o *checkpointing*;
- *checkpointing*: criação de um *checkpoint*, que só pode ocorrer se o objeto de engenharia envolvido satisfaz uma pré-condição estabelecida na política de *checkpointing*;
- *cluster checkpoint*: um *template* contendo *checkpoints* de objetos básicos de engenharia num *cluster*;
- desativação: *checkpointing* seguido da eliminação de um *cluster*;
- *cloning*: instanciação do *cluster* a partir do *checkpoint*;
- recuperação: *cloning* de um *cluster* após a falha ou eliminação de um *cluster*;
- reativação: *cloning* de um *cluster* após sua desativação;
- migração: move um *cluster* para uma cápsula diferente;

Regras de Estruturação

Uma especificação de engenharia [RM-ODPa, RM-ODPc] define uma infra-estrutura necessária para suportar distribuição funcional de um sistema ODP através da:

- identificação das funções ODP exigidas para gerenciar distribuição física, comunicação, processamento e armazenamento;
- identificação dos papéis de diferentes objetos de engenharia suportando as funções ODP (por exemplo, o núcleo).

Além disso, uma especificação de engenharia é expressa em termos:

- da configuração de objetos de engenharia, *clusters*, cápsulas e nós;
- das atividades que ocorrem nos objetos de engenharia;
- das interações que ocorrem entre os objetos de engenharia.

A linguagem de engenharia possui as seguintes regras:

- regras de canal;
- regras de referência de interfaces;
- regras de *bindings* distribuídos [RM-ODPa,RM-ODPc];
- regras de relocação;
- regras de *cluster*;
- regras de cápsula;
- regras de falha.

Regras de canal

Um canal suporta transparência de distribuição na interação entre objetos de engenharia. Isto inclui:

- interação entre um objeto cliente e um objeto servidor;
- um grupo de objetos interagindo com outro grupo de objetos;
- um fluxo de dados entre múltiplos objetos produtores para múltiplos objetos consumidores.

A figura 2-2 abaixo ilustra um exemplo de canal:

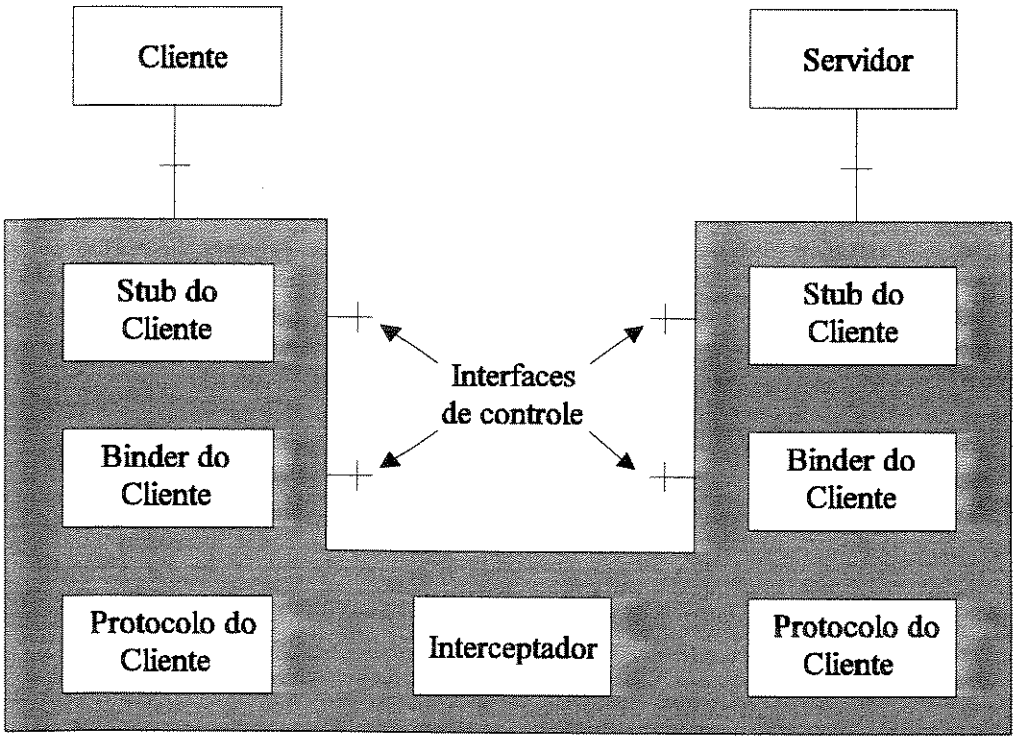

Figura 2-2 Um exemplo de um canal cliente/servidor básico

Um canal é uma configuração de objetos, *stubs*, *binders*, protocolos e interceptadores conectados a um conjunto de objetos de engenharia. Se for criado um canal no mesmo domínio de comunicação, não existe a necessidade de um interceptador.

Stubs

Objetos básicos de engenharia que interagem via canais estão ligados localmente a *stubs*. Num canal, *stubs* suportam a conversão de dados durante a iteração. *Stubs* podem controlar e manter registros (para segurança). Os *stubs* podem interagir com outros objetos de engenharia fora do canal, caso seja necessário.

Um *stub* tem uma interface de apresentação para interação com um objeto de engenharia e uma interface de controle para gerenciamento de qualidade de serviço.

Binders

Os *binders* num canal gerenciam a integridade fim-a-fim e a qualidade de serviço do canal. Devem prover transparência de relocação quando for necessário, monitoração de falhas de comunicação e reparar conexões quebradas.

Um *binder* tem interfaces de controle que possibilitam mudanças na configuração do canal e remoção do mesmo.

Protocolos

Protocolos provêem funções de comunicação. Podem interagir com objetos de engenharia fora do canal se necessário. Possuem uma interface para interação com um binder e uma interface para interação com outros objetos protocolo.

Quando objetos protocolo estão em domínios de comunicação diferentes, eles requerem um interceptador para que possam interagir entre si.

Interceptadores

Os interceptadores são responsáveis pela verificação e pelas transformações durante uma interação entre domínios diferentes; logo, eles necessitam ter acesso aos tipos das interfaces dos objetos de engenharia que estão unidos pelo canal em que o interceptador está localizado.

Regras de referências de interfaces

A referência de uma interface de engenharia identifica sem ambigüidades no espaço e no tempo as interfaces dos objetos de engenharia [RM-ODPc].

Uma referência de uma interface de engenharia contém dados que possibilitam a determinação de:

- um *template* adequado de um objeto canal para ser selecionado e instanciado para a interface;
- a localização no espaço e no tempo de algum ponto de comunicação para a interface.

Regras de relocação

Objetos de engenharia podem ser relocados como um resultado de:

- uma reativação e desativação;
- *checkpoint* e *recover*;
- uma migração;
- funções de gerenciamento do domínio de comunicações (mudando o identificador da interface de comunicação).

Regras de *cluster*

Um *cluster* contém um conjunto básico de objetos de engenharia associados com um gerente de *cluster*. Cada membro de um *cluster* pode ter uma interface suportando a função de gerenciamento de objeto. Cada interface de gerenciamento está ligada ao gerente de *cluster*. Um objeto básico de engenharia num *cluster* está sempre ligado ao seu núcleo numa interface que provê funções de gerenciamento de nó, e ao seu gerente de *cluster*. Além disso, um objeto num cluster pode estar ligado a outros objetos no mesmo *cluster*, ou em outros clusters. Cada gerente de *cluster* na cápsula está ligado ao gerente de cápsula. Esta estrutura está ilustrada na figura 2-3:

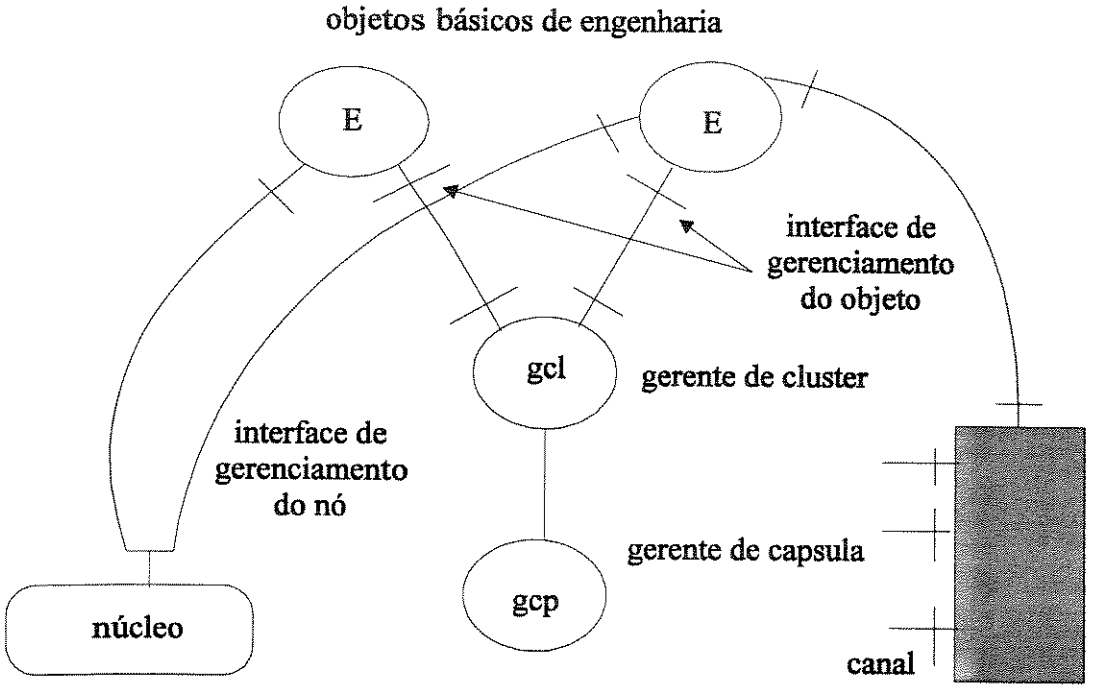

Figura 2-3 Exemplo de uma estrutura suportando um objeto básico de engenharia

Um *cluster* está sempre contido numa cápsula simples, sendo responsável pela sua segurança, embora possa ser auxiliado por funções de segurança. Sua instanciação é feita pelo gerente de cápsula.

O gerente de *cluster* fornece as funções de gerenciamento, incorporando a política de gerenciamento para os objetos de engenharia no seu *cluster* associado. Esta política pode levar o gerente de *cluster* a interagir com outras funções ODP para completar atividades de gerenciamento.

.

Regras de cápsula

Uma cápsula consiste de:

- *cluster(s)*;
- gerentes de *cluster*, um para cada *cluster* na cápsula;
- um gerente de cápsula, que está ligado aos gerentes de cluster na cápsula.

A instanciação de uma cápsula é desempenhada pelo núcleo, utilizando um template de cápsula que especifica a configuração inicial dos objetos na cápsula.

O gerente de cápsula gerencia a política de clusters na cápsula. Esta política pode levar o gerente de cápsula a interagir com outras funções para completar as atividades de gerenciamento.

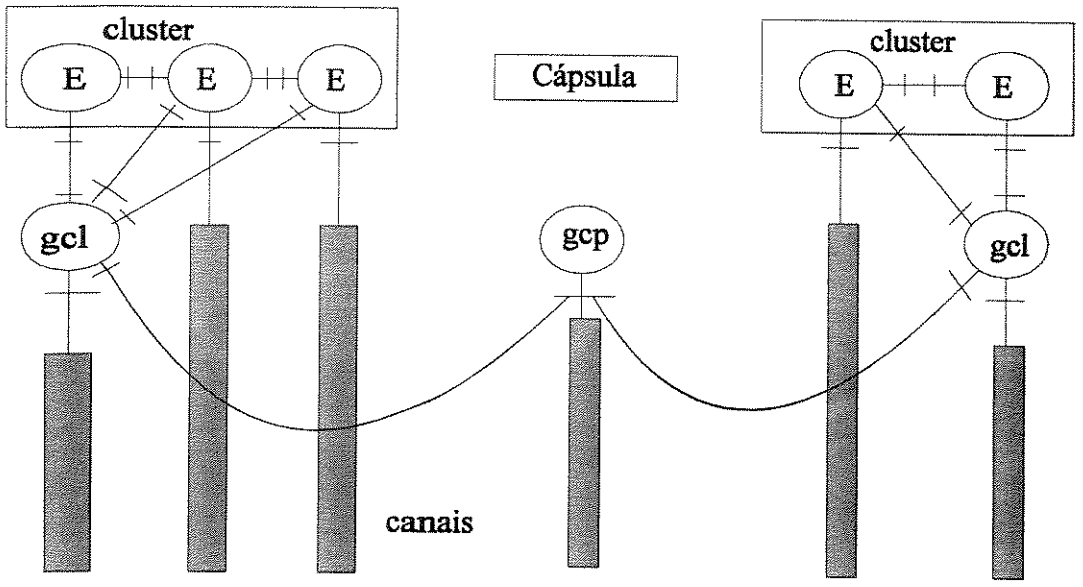

Figura 2-4 Exemplo da estrutura de uma cápsula

Regras de nó

Um nó consiste de um núcleo e um conjunto de cápsulas. Todos os objetos num nó compartilham as mesmas funções de processamento, armazenamento e comunicação.

A estrutura de um nó é ilustrada na figura 2-5.

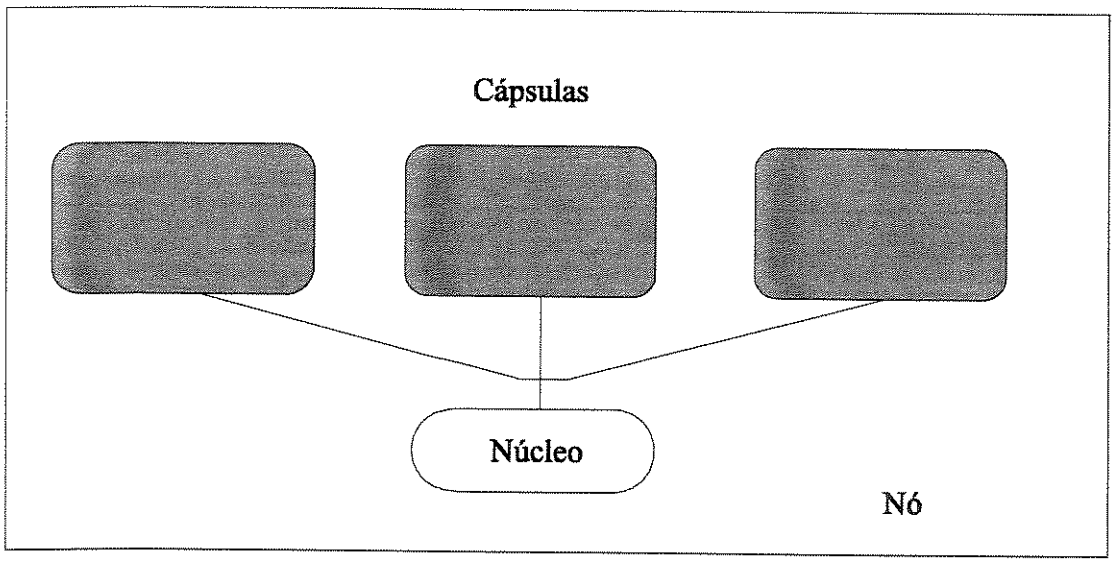

Figura 2-5 Exemplo da estrutura de um nó

O núcleo incorpora a política de gerenciamento do nó provendo uma interface de gerenciamento separada para cada cápsula.

Regras de gerenciamento de aplicação

O gerenciamento do ciclo de vida (criação, migração, desativação, reativação, *checkpointing*, falha, recuperação e remoção) dos *clusters* é coordenado por políticas de gerenciamento.

Um conjunto de *clusters* gerenciados forma um domínio de gerenciamento de aplicação. Políticas de gerenciamento de aplicação podem ser implementadas pelas funções de gerenciamento específicas da aplicação.

Regras de falha

Uma falha pode estar localizada nos *clusters*, cápsulas ou nós.

- uma falha localizada num *cluster* pode ser detectada pelo seu gerente de *cluster*;
- uma falha localizada numa cápsula pode ser detectada pelo seu gerente de cápsula;
- uma falha de um nó pode ser detectada por objetos protocolo de outros nós, aos quais ele se interconecta;
- uma falha de um domínio de comunicação pode ser detectada por objetos protocolo em outros domínios de comunicação aos quais ele se interconecta.

2.1.3.5 Linguagem Tecnológica

Descreve a implementação do sistema ODP em termos de uma configuração de objetos representando os componentes de hardware e o software da implementação e mostrando o custo e disponibilidade destes componentes.
O ponto de vista tecnológico fornece uma ligação entre o conjunto de especificações de pontos de vista e a implementação real.

2.1.3.6 Mapeamento da Linguagem de Computação em Linguagem de Engenharia

O objeto computacional pode ser suportado por funções de engenharia, ou seja, pode-se fazer um mapeamento de um objeto computacional em funções de engenharia.
O refinamento de *templates* computacionais em *templates* de engenharia (formando um *cluster*), corresponde a noção de compilar programas para produzir um código objeto [Gunji95].
O conceito de cápsula corresponde a noção de espaço de endereçamento ou a um processo na maioria dos sistemas operacionais.
As seguintes correspondências têm sido identificadas entre objetos computacionais e objetos básicos de engenharia:

- um objeto computacional com uma interface corresponde a um conjunto de objetos básicos de engenharia pertencentes a um único *cluster* ou apenas a um objeto básico de engenharia (figura 2-6);
- um objeto computacional pode ter várias interfaces que podem ser executadas numa mesma estação de trabalho ou em múltiplas estações (figura 2-7).

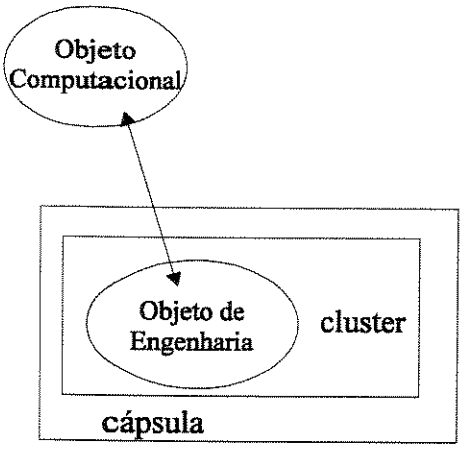

Figura 2-6 Correspondência 1:1 entre as duas linguagens

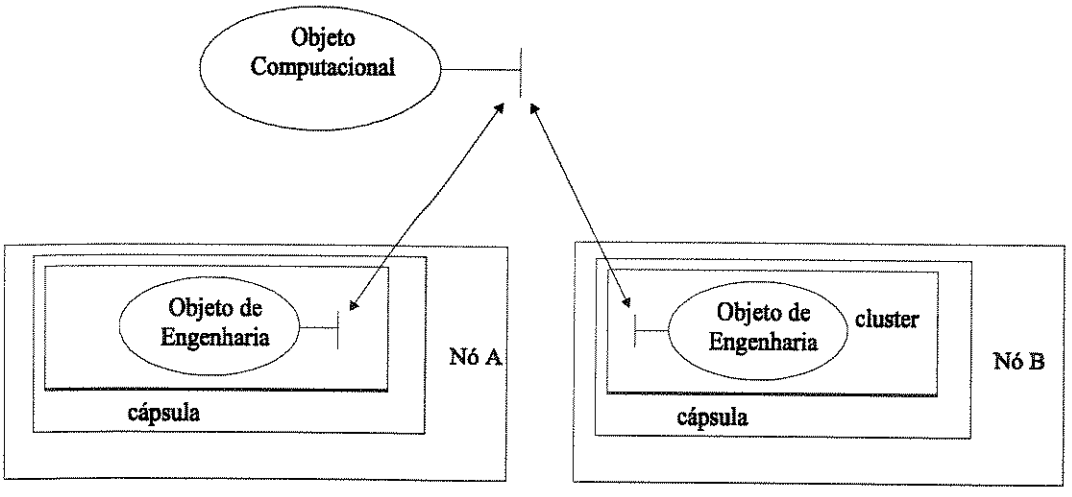

Figura 2-7 Correspondência n:1

2.1.4 Funções de gerenciamento ODP

2.1.4.1 Função de gerenciamento de nó

Controla processamento, armazenamento e comunicação dentro do nó, sendo fornecida por cada núcleo numa ou mais interfaces de gerenciamento de nó.
A função de gerenciamento de nó:

- gerencia *threads*;

- acessa *clocks* e gerencia *timers*;
- cria canais e localiza interfaces;
- instancia e remove cápsulas.

Gerenciamento de *threads*

Provê funções para criação e sincronização de *threads* dentro de uma cápsula.

Acesso a *clock* e gerenciamento de *timer*

Provê funções para determinar a hora corrente num domínio de gerenciamento específico, ativar, monitorar e cancelar *timers*.

Criação de canal e localização de interface

Provê funções para:

- habilitar *binding* entre um objeto de engenharia dentro de uma cápsula e uma instância da função de *trading* [RM-ODPc];
- colocar uma interface de engenharia disponível para *binding* com outros objetos em outras cápsulas;
- estabelecer um *binding* entre um objeto de engenharia dentro de uma cápsula e um conjunto de outros objetos de engenharia (identificados por referências de interface de engenharia);
- determinar o tipo de canal e a interface de comunicação envolvida, a partir de uma referência de interface específica.

Instanciação do *template* de cápsula e remoção da cápsula

A instanciação do template de cápsula consiste de:

- alocação de funções de comunicação, armazenamento e processamento para uma nova cápsula no mesmo nó, bem como da interface de gerenciamento do nó fornecida pelo núcleo;
- criação do gerente de cápsula para a nova cápsula;
- criação de uma interface de gerenciamento no novo gerente de cápsula;
- produção de um identificador para a interface de gerenciamento de cápsula;
- criação de uma interface de controle de cápsula para a nova cápsula no núcleo;
- produção de um identificador para a interface de controle da cápsula.

A remoção de uma cápsula remove todos os objetos na cápsula.

2.1.4.2 Funções de gerenciamento de objeto

Quando um objeto pertence a um *cluster* que pode ser desativado, sofrer um *checkpoint* ou migração, o objeto deve ter uma interface de gerenciamento que forneça uma ou mais das seguintes funções:

- *checkpoint* de objeto;
- remoção de objeto.

Interfaces de gerenciamento de objetos diferentes podem prover funções diferentes. *Checkpoint* de objeto será utilizado durante o *checkpoint* de *cluster*.

Quando um objeto é removido, *stubs*, *binders*, protocolos e interceptadores ligados ao objeto podem ser removidos.
A função de gerenciamento do objeto é usada pela função de gerenciamento do *cluster*.

2.1.4.3 Funções de gerenciamento de *cluster*

O gerente do *cluster* através da interface de gerenciamento do *cluster* pode fornecer uma ou mais das seguintes funções:

- modificação da política de gerenciamento do *cluster*;
- desativação do *cluster*;
- *checkpoint* do *cluster*;
- substituição do *cluster*, com um novo *cluster* sendo instanciado a partir de um *checkpoint*;
- migração do *cluster*;
- remoção do *cluster*.

A interface de gerenciamento pode variar de um *cluster* para outro.

Checkpoint do *cluster*

Contém a informação necessária para restabelecer um *cluster* e inclui os seguintes elementos:

- *checkpoint* dos objetos no *cluster*;
- a configuração dos objetos no *cluster*;
- informação suficiente para restabelecer canais que envolvam objetos no *cluster*.

Remoção, desativação e falha do *cluster*

Numa operação de remoção todos os objetos do *cluster* (inclusive *stubs* e *binders*) e o gerente de *cluster* são removidos. A desativação é coordenada pelas funções de desativação e reativação, produzindo um *checkpoint* e então removendo o *cluster* e sua estrutura.
Uma falha no *cluster* pode causar remoção de todos os objetos e em alguns casos, da estrutura de suporte ao *cluster*.

Reativação e recuperação do *cluster*

Um *cluster* desativado pode ser reativado a partir de um de seus *checkpoints*. Reativação é necessariamente uma função de gerenciamento de cápsula, porque o gerente de *cluster* é removido como parte da desativação do *cluster*. Um *cluster* pode ser recuperado também a partir de um de seus *checkpoints*.

Migração do *cluster*

Consiste de uma cópia do *cluster* de um local para o outro, com remoção do *cluster* original. É coordenada pela função de migração.

2.1.4.4 Funções de gerenciamento de cápsula

O gerente de cápsula através da interface de gerenciamento de cápsula, provê uma ou mais das seguintes funções:

- instanciação (dentro da cápsula) de um template de *cluster*, incluindo reativação e recuperação;
- desativação da cápsula, incluindo a desativação de todos os *clusters* na cápsula (utilizando as funções de gerenciamento de *cluster*);
- *checkpoint* da cápsula envolvendo *checkpoint* de todos os *clusters* na cápsula;
- remoção da cápsula, removendo todos os *clusters* associados a ela e o gerente de cápsula.

Instanciação do *template* de *cluster*

A instanciação do template de *cluster* consiste dos seguintes passos:

- instanciação de um *cluster* a partir de um *template* de *cluster*;
- introdução de um gerente de *cluster* para o novo *cluster*;
- criação de um identificador para a interface de gerenciamento do gerente de *cluster*;
- *binding* para outros objetos de acordo com as regras da linguagem de engenharia e informação de *binding* no *template* de *cluster*.

Um *template* de *cluster* pode conter informação específica a um domínio. Se o *template* for instanciado em outro domínio, esta informação deve ser transformada.

Remoção da cápsula

Provê remoção do gerente de cápsula, podendo resultar na remoção de *stubs*, *binders*, protocolos, interceptadores e outros objetos associados na cápsula.

2.2 Arquitetura CORBA (Common Object Request Broker Architecture)

2.2.1 Introdução

A arquitetura CORBA [CORBA95] é uma tentativa de satisfazer as metas do consórcio OMG [Vinoski93], um grupo de empresas dedicado a produzir especificações de ambientes orientados a objetos e que tem como finalidade permitir a integração de vários sistemas de objetos provendo mecanismos pelos quais esses objetos transparentemente solicitam serviços e recebem respostas.
Um objeto é uma entidade identificável, encapsulada, que provê um ou mais serviços pedidos por um cliente. Sendo assim, cada objeto possui uma interface, que é uma descrição de um conjunto de possíveis operações que um cliente pode pedir sobre um objeto. Essa interface é especificada em IDL (Linguagem de Definição de Interface), uma linguagem que define os tipos de objetos de acordo com as operações que podem ser executadas sobre eles e os parâmetros para essas operações. Um objeto possui também uma identificação própria e única, chamada de referência do objeto.
A arquitetura também provê interoperabilidade entre aplicações sobre diferentes máquinas em ambientes distribuídos heterogêneos [Konstantas93], interconectando múltiplos sistemas de objetos.

2.2.2 A estrutura de um núcleo ORB (Object Request Broker)

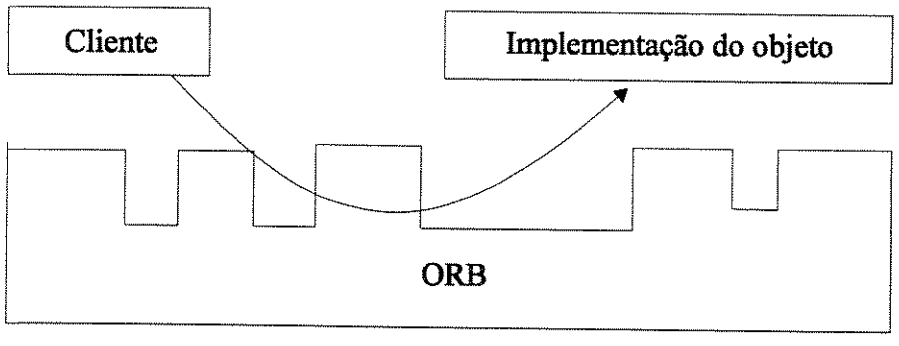

Figura 2-8 Invocação de um pedido via ORB

A figura 2-8 mostra um pedido sendo enviado por um cliente para uma implementação de objeto. O cliente é uma entidade que deseja executar uma operação sobre um objeto e a implementação do objeto constitui no código (métodos) e dados necessários para se criar um objeto que forneça um conjunto de serviços desejados.
O ORB é responsável por localizar a implementação do objeto, pela preparação da implementação do objeto para receber o pedido e pela comunicação de dados do pedido. A interface que o cliente vê é completamente independente de onde o objeto

servidor está localizado, da linguagem de programação que ele está implementado ou qualquer outro aspecto que não esteja refletido na interface do objeto.

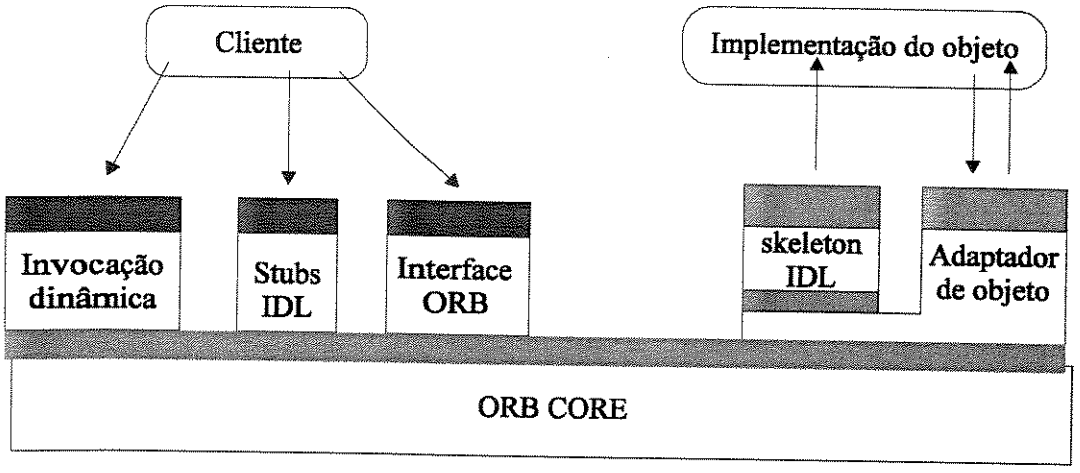

Figura 2-9 Estrutura de um ORB simples

A figura 2-9 mostra a estrutura de um ORB simples. As interfaces do ORB são mostradas pelas caixas sombreadas, e as setas indicam se o ORB é chamado ou executa uma chamada *up-call* através da interface.
Para se fazer um pedido, o cliente pode usar uma interface de invocação estática, (especificada em IDL e específica a um objeto provedor de serviço), ou dinâmica (interfaces contidas num repositório de interfaces, que permite referenciar objetos em tempo de execução). O cliente pode também diretamente interagir com o ORB para algumas funções.
O ORB localiza o código de implementação apropriado, transmite parâmetros e transfere controle para a implementação do objeto através de um *skeleton* IDL (interface para os métodos do objeto). A implementação do objeto pode chamar o ORB via adaptador de objetos enquanto processando um pedido. O adaptador de objetos é responsável pela comunicação entre a implementação do objeto e os serviços fornecidos pelo ORB.
Os *skeletons* são específicos para a interface ORB e o adaptador de objetos. Executando o pedido, a implementação do objeto pode obter alguns serviços do ORB através do adaptador de objetos. Quando o pedido está completo, valores de controle e saída são retornados para o cliente.
A implementação do objeto pode escolher qual adaptador de objeto a usar. Esta decisão é baseada no tipo de serviço que a implementação do objeto exige.
A figura 2-10 mostra como a informação de interface e implementação é feita disponível para clientes e implementações de objetos. A interface é definida em IDL e as definições IDL são usadas para gerar *stubs* para clientes e os *skeletons* para implementação de objetos.
A informação da implementação do objeto é fornecida na hora da instalação e é armazenada no Repositório de Implementação para uso durante a entrega do pedido.

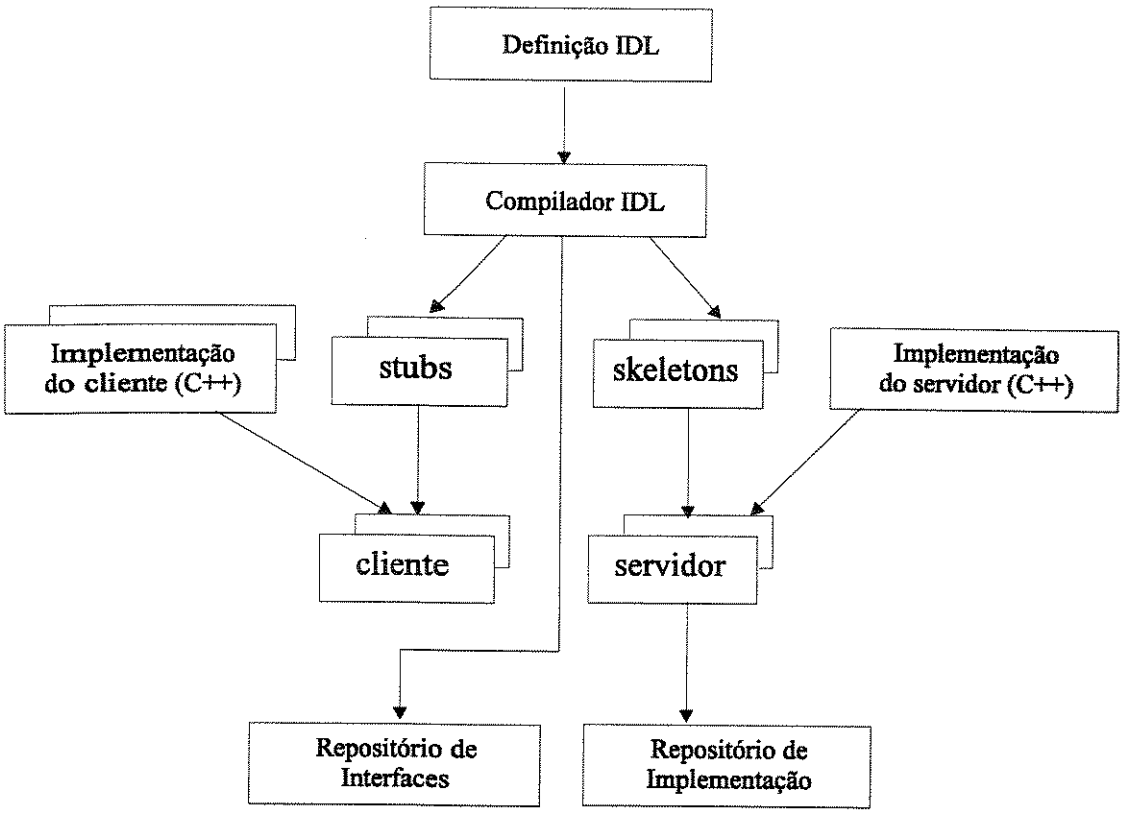

Figura 2-10 Desenvolvimento em CORBA

2.2.3 Definição dos componentes da arquitetura CORBA

2.2.3.1 Object Request Broker (ORB)

Na arquitetura CORBA não se exige que o ORB seja implementado como um simples componente, mas que seja definido pelas suas interfaces. Qualquer implementação ORB que provê uma interface apropriada é aceitável. Esta interface pode ser organizada em três categorias:

1. aquelas operações que são as mesmas para todas implementações ORB;
2. aquelas operações que são específicas a tipos particulares de objetos;
3. aquelas operações que são específicas a estilos particulares de implementações de objetos.

Diferentes ORBs podem fazer escolhas de implementação diferentes, e, juntos com os compiladores IDL, repositórios e vários adaptadores de objetos, fornecer um conjunto de serviços para clientes e implementações de objetos, que tem diferentes propriedades e qualidades.

Podem existir múltiplas implementações ORB com diferentes representações para referências de objetos e diferentes meios de executar invocações. É possível a um cliente ter acesso simultaneamente a duas referências de objetos gerenciadas por diferentes implementações ORB. Quando se pretende trabalhar com dois ORBs juntos, por exemplo, esses devem ser capazes de distinguirem suas referências de objetos. Esta distinção não deve ser responsabilidade do cliente.
A parte do ORB que provê a representação básica de objetos e a comunicação de pedidos é chamada de ORB Core. A arquitetura CORBA é projetada para suportar diferentes mecanismos de objetos, e faz isso estruturando o ORB com componentes acima do ORB Core, que provê interfaces que podem mascarar as diferenças entre ORB Cores.

2.2.3.2 Cliente

Um cliente de um objeto tem acesso à sua referência e invoca operações sobre ele. Um cliente conhece somente a estrutura lógica do objeto, de acordo com sua interface e sabe do seu comportamento através das invocações. Embora geralmente se considere que um cliente seja um programa ou um processo iniciando pedidos sobre um objeto, é importante reconhecer que qualquer coisa pode ser um cliente relativo a um objeto particular. Por exemplo, a implementação de um objeto pode ser um cliente de outros objetos.
Os clientes geralmente vêem objetos e interfaces ORB através da perspectiva de um mapeamento de linguagem, trazendo o ORB até o nível do programador. São portáteis e devem ser capazes de trabalhar sem alterações sobre qualquer ORB que suporte o mapeamento de linguagem desejado com qualquer instância de objeto que implementa a interface desejada. Os clientes não têm conhecimento da implementação do objeto, de qual adaptador de objeto é usado pela implementação, ou qual ORB é usado para acessá-lo.

2.2.3.3 Implementações de objetos

Uma implementação de objeto provê a semântica do objeto, normalmente definindo dados para a instância do objeto e código para métodos do objeto. Freqüentemente a implementação usará outros objetos ou software adicional para implementar o comportamento do objeto. Em alguns casos, a função primária do objeto é ter efeitos colaterais sobre entidades que não são objetos.
Uma variedade de implementações de objetos pode ser suportada, incluindo servidores separados, bibliotecas, um programa por método, uma aplicação encapsulada, uma base de dados orientada a objetos etc. Através do uso de adaptadores de objetos adicionais, é possível suportar virtualmente qualquer estilo de implementação de objeto.
Geralmente, a implementação do objeto não depende do ORB ou como o cliente invoca o pedido. Implementações de objetos podem selecionar interfaces para serviços dependentes de ORB pela escolha do adaptador de objeto. Implementações

de objetos são portáteis através de qualquer ORB que suporte o mapeamento de linguagem desejada e implemente o adaptador de objeto desejado.

2.2.3.4 Referências de objetos

Uma referência de objeto é a informação necessária para especificar um objeto dentro de um ORB. Tanto clientes quanto implementações de objetos possuem pouca noção das referências de objetos de acordo com o mapeamento da linguagem.
Duas implementações ORB podem diferir na escolha das representações da referência de objeto.
A representação de uma referência de objeto manipulada pelo cliente é somente válida para o tempo de vida do cliente.
Todos os ORBs devem prover o mesmo mapeamento de linguagem para uma referência de objeto, dada uma linguagem de programação particular. Isto permite a um programa escrito em uma linguagem particular acessar referências de objetos independentes do ORB.
Existe também uma referência de objeto distinta, garantidamente diferente de todas referências de objetos, que denota "nenhum objeto".

2.2.3.5 Linguagem de definição de interface (IDL)

A linguagem de definição de interface define os tipos de objetos especificando suas interfaces. Uma interface consiste de um conjunto de operações nomeadas e os parâmetros para aquelas operações. Embora a IDL forneça o *framework* conceitual para descrição de objetos manipulados pelo ORB, não é necessário existir um código fonte IDL disponível ao ORB. Tão logo a informação equivalente esteja disponível na forma de rotinas *stub* ou um repositório de interface em tempo de execução, um ORB particular pode funcionar corretamente.
IDL é o meio pelo qual uma implementação de objeto específica informa seus clientes potenciais quais operações estão disponíveis e como elas devem ser invocadas. Das definições IDL, é possível mapear objetos CORBA dentro de linguagens de programação particular ou sistemas de objetos.

2.2.3.6 Mapeamento da linguagem de programação

Diferentes linguagens de programação, orientadas ou não a objetos, podem preferir acessar objetos da arquitetura CORBA por diferentes modos. Para linguagens orientadas a objeto pode ser desejável ver objetos CORBA como objetos da linguagem de programação. Mesmo para linguagens não orientadas a objeto, é uma boa idéia esconder a representação exata ORB da referência de objeto, nomes de método etc. O mapeamento da linguagem inclui a definição de tipos de dados específicos e interfaces para acessar objetos através do ORB. Ele inclui a estrutura da

interface *stub* para o cliente, a interface de invocação dinâmica, o *skeleton* de implementação, os adaptadores de objetos e a interface direta do ORB.
O mapeamento da linguagem também define a interação entre invocações de objetos e as *threads* de controle no cliente ou na implementação. O mapeamento mais comum provê chamadas síncronas nas quais a rotina retorna quando a operação completa. Mapeamentos adicionais podem ser fornecidos para permitir uma chamada ser iniciada e o controle retornado para o programa. Em tais casos, rotinas adicionais específicas da linguagem devem ser fornecidas para sincronizar as *threads* de controle do programa com a invocação do objeto.

2.2.3.7 *Stubs* do cliente

Para um mapeamento de linguagem particular, existirá uma interface de programação dos *stubs* para cada tipo de interface. Os *stubs* fazem chamadas sobre o resto do ORB (otimizadas para o ORB *Core* específico). Se mais de um ORB está disponível, podem existir diferentes *stubs* correspondentes aos diferentes ORBs. Nesse caso, é necessário que o ORB e o mapeamento de linguagem cooperem para associar os *stubs* corretos com a referência de objeto específica.

2.2.3.8 Interface de invocação dinâmica

É uma interface que permite a construção dinâmica de invocações de objetos. Uma chamada de uma rotina *stub* é específica a uma operação particular sobre um objeto particular. No modo de invocação dinâmica um cliente pode especificar o objeto a ser invocado, a operação a ser executada, e o conjunto de parâmetros para a operação através de uma chamada ou uma seqüência de chamadas. O código cliente deve fornecer informação sobre a operação a ser executada e os tipos de parâmetros que estão sendo passados (obtendo isso talvez de um repositório de interface ou de outra fonte). A natureza da interface de invocação dinâmica pode variar substancialmente de um mapeamento de linguagem de programação para outro.

2.2.3.9 *Skeleton* de implementação

Para um mapeamento de linguagem particular, e possivelmente dependendo do adaptador de objeto, existirá uma interface *skeleton* para os métodos que implementam cada tipo de objeto. A interface será geralmente uma interface *up-call*.
A existência de um *skeleton* não implica na existência de um correspondente *stub* do cliente (clientes podem também fazer pedidos via a interface de invocação dinâmica).
É possível também escrever um adaptador de objeto que não use *skeletons* para invocar métodos de implementação.

2.2.3.10 Adaptador de objeto

Um adaptador de objeto é a maneira pela qual uma implementação de objeto acessa serviços fornecidos pelo ORB. Serviços fornecidos pelo ORB através de um adaptador de objetos freqüentemente incluem: geração e interpretação de referências de objetos, invocação de método, segurança de interações, ativação e desativação de objeto e implementação, mapeamento de referências de objetos para implementações e registro de implementações. Diferentes ORBs proverão diferentes níveis de serviço e diferentes ambientes operacionais podem prover algumas propriedades implicitamente, e exigirem que outras sejam adicionadas ao adaptador de objetos. Por exemplo, é comum que implementações de objetos queiram armazenar certos valores na referência de objeto para facilitar a identificação do objeto na invocação. Se o adaptador de objeto permite que a implementação especifique tais valores quando um novo objeto é criado, é possível armazenar esses valores na referência de objetos para aqueles ORBs que permitam tal operação. Se o ORB *Core* não provê esta característica, o adaptador de objeto poderia registrar o valor no seu próprio armazenamento e fornecê-lo para a implementação sobre uma invocação. Com adaptadores de objetos, é possível para uma implementação de objetos ter acesso a um serviço se ele está ou não implementado no ORB *Core* - se o ORB *Core* provê o serviço, o adaptador simplesmente fornece uma interface para ele; se não, o adaptador deve implementá-lo no topo do ORB Core. Cada instância de um adaptador particular deve prover a mesma interface e serviço para todos ORBs sobre os quais ele está implementado. Entretanto, não é necessário que todos adaptadores de objetos forneçam a mesma interface ou funcionalidade.

2.2.3.11 Interface ORB

A interface ORB é a mesma para todos os ORBs e não depende da interface do objeto ou do adaptador de objeto. Devido à maioria das funcionalidades do ORB serem fornecidas através do adaptador de objetos, dos *stubs*, do *skeleton*, ou da interface de invocação dinâmica, existem poucas operações que são comuns a todos objetos. Estas operações são úteis tanto para clientes como para implementações de objetos.

2.2.3.12 Repositório de interface

É um serviço que provê objetos persistentes que representam a informação IDL numa forma disponível em tempo de execução. A informação do repositório de interface pode ser usada pelo ORB para executar pedidos. Usando esta informação, é possível também um programa encontrar um objeto cuja interface não era conhecida quando o programa foi compilado, e ainda, ser capaz de determinar quais operações são válidas sobre o objeto e fazer uma invocação sobre ele.
O repositório de interface é um lugar comum para armazenar informação adicional associada com as interfaces dos objetos ORB.

2.2.3.13 Repositório de implementação

Contém a informação que permite o ORB localizar e ativar implementações de objetos. Embora a maioria da informação seja específica a um ORB ou a um ambiente operacional, o repositório de implementação também registra dados sobre a instalação de implementações e controle de políticas relacionadas com a ativação e execução de implementações de objetos, controle administrativo, alocação de recursos, segurança etc.

2.2.4 Alguns exemplos de ORBs

2.2.4.1 ORB residente no cliente e na implementação

Se existe um mecanismo de comunicação adequado, um ORB pode ser implementado em rotinas residentes nos clientes e implementações. Os *stubs* no cliente ou usam um mecanismo de comunicação (IPC) transparente de locação ou diretamente acessam um serviço de locação para estabelecer a comunicação com as implementações. O código ligado com a implementação é responsável por estabelecer as bases de dados apropriadas para uso pelos clientes.

2.2.4.2 ORB baseado em servidor

Para centralizar o gerenciamento do ORB, todos os clientes e implementações podem se comunicar com um ou mais servidores, que têm o trabalho de rotear os pedidos dos clientes para as implementações.

2.2.4.3 ORB baseado em sistema

Para garantir segurança, robustez, e performance, o ORB poderia ser fornecido como um serviço básico do sistema operacional. Se o sistema operacional tem acesso a localização e a estrutura dos clientes e implementações, é possível implementar várias otimizações, como por exemplo, evitar formatação dos dados quando executando uma chamada local.

2.3 Trabalhos correlatos

Os trabalhos correlatos descritos nesta seção apresentam como característica comum em relação à implementação realizada, o fato de que ambos procuram fornecer uma estrutura que dê suporte suporte a conceitos e serviços básicos do modelo ODP. A principal diferença entre os trabalhos realizados e o que está sendo proposto está no mecanismo utilizado na comunicação dos objetos.
Sendo assim, uma proposta para modelagem das funções de gerenciamento para processamento distribuído aberto [Garcia94] é mostrada na figura 2-11. Considera-se aqui um núcleo por máquina, representado por um servidor que irá prover as funções de gerenciamento de nó, e associados ao núcleo um conjunto de servidores (gerentes de cápsulas) e associado a cada um destes um outro conjunto de servidores (gerente de clusters). A comunicação entre os diversos servidores e destes com os objetos básicos de engenharia se dá através de um mecanismo baseado em RPC. Como interface com o programador, é oferecida uma biblioteca com um conjunto de chamadas para acesso às funções dos servidores.

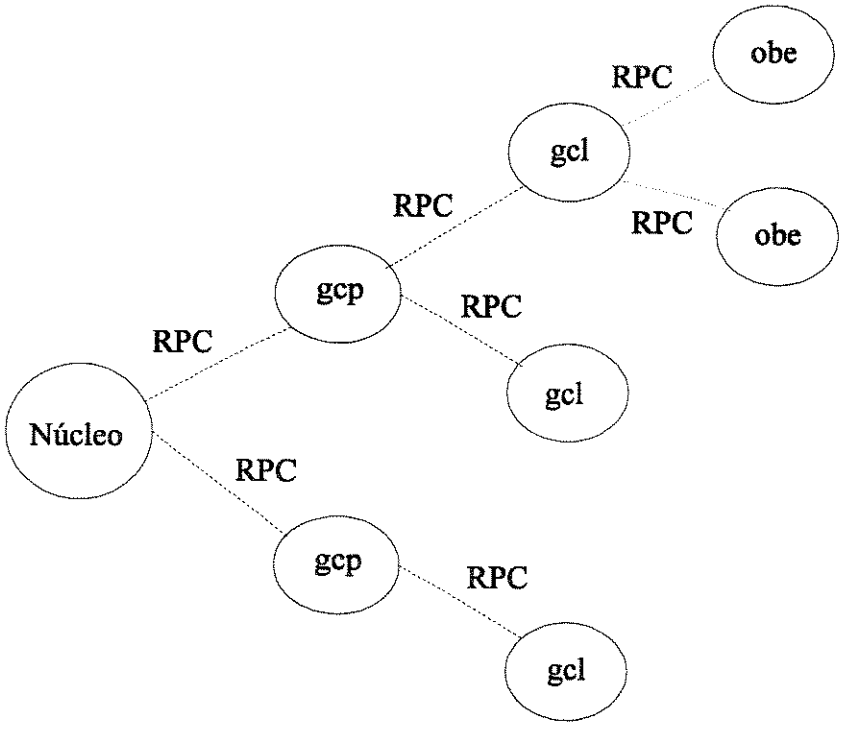

Figura 2-11 Um protótipo para processamento distribuído aberto

Uma outra proposta de suporte para processamento distribuído seguindo o padrão ODP é apresentada na figura 2-12:

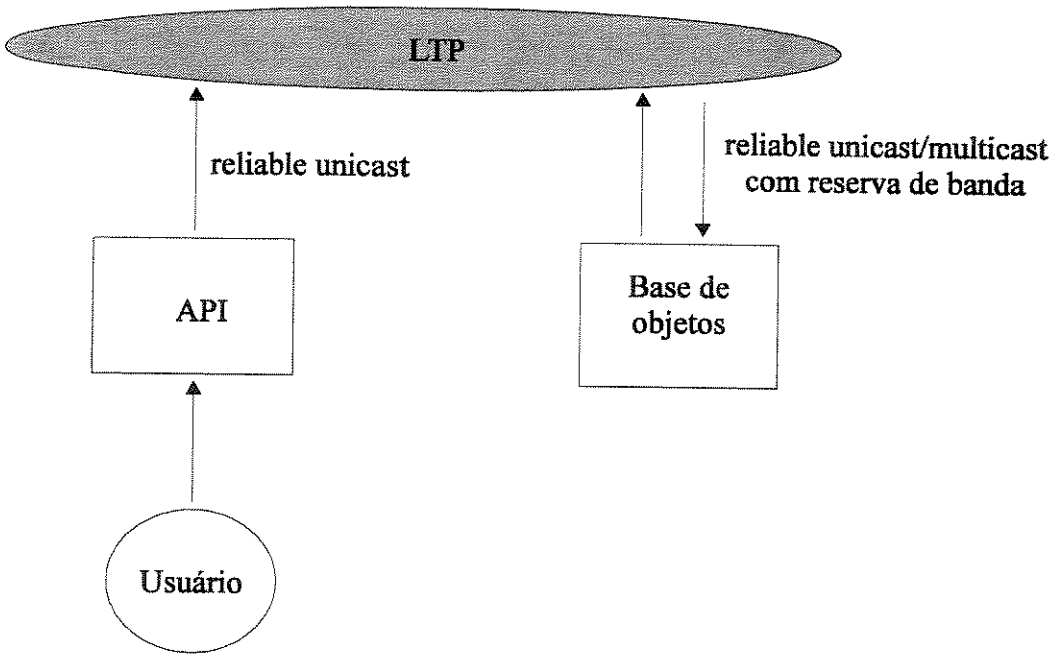

Figura 2-12 Uma estrutura de suporte ao modelo ODP

As aplicações do usuário fazem acesso uniforme aos serviços oferecidos pela base de objetos via uma interface de programação (API) [Maezi95]. Esta API é formada por um conjunto de classes de objetos denominados *templates*, que armazenam a informação necessária para instanciar um objeto. A base de objetos [Gunji95] provê a infra-estrutura necessária à existência e à interação entre objetos em tempo de execução, provendo mecanismos para gerenciamento das funções ODP a nível de cápsula, *cluster* e objeto, e para criação de canais ligando uma interface de um objeto cliente a uma interface de um objeto servidor. A comunicação entre a API e a base de objetos é feita através de *reliable unicast*, fornecido por um protocolo de comunicação confiável, LTP [Conceição95], que é capaz de prover também serviços de *reliable multicast* e comunicação não confiável com reserva de banda. O LTP também é utilizado pela base de objetos no suporte às funções de distribuição.

Capítulo 3

Proposta da implementação das funções de gerenciamento ODP utilizando uma arquitetura CORBA

3.1 Introdução

Este capítulo apresenta uma proposta para a implementação das principais funções de gerenciamento definidas pelo modelo ODP, utilizando uma plataforma baseada na arquitetura CORBA, descrita no capítulo anterior. Estas funções de gerenciamento se constituem de alguns dos requisitos necessários para a construção de aplicações distribuídas segundo o modelo citado.

Para a realização do trabalho, algumas decisões de projeto foram tomadas face às limitações dos recursos disponíveis.

Primeiro, foi utilizada uma plataforma compatível com a especificação CORBA sobre o sistema AIX da IBM. A mesma implementação poderia ter sido desenvolvida em ambientes Solaris e Windows NT.

Segundo, como a plataforma CORBA usada não apresentava um suporte a serviços de persistência de objetos, tornou-se necessário buscar uma base de dados externa. Embora o RM-ODP não determine na linguagem tecnológica como deve ser a base capaz de prover tais serviços, o ideal seria que essa fosse orientada a objetos, facilitando a abstração dos conceitos ODP, e que provesse um serviço de transações. Considerando ainda que não se dispunha de uma base com essas características, decidiu-se então utilizar uma que fosse de domínio público e que tivesse algumas propriedades importantes no âmbito do trabalho. Desenvolveu-se também um servidor de transações para garantir a consistência dos dados armazenados na base usada.

Algumas outras dificuldades relacionadas com o funcionamento das ferramentas empregadas foram encontradas, e serão comentadas mais detalhadamente no próximo capítulo.

3.2 Serviços oferecidos

Conforme proposto pelo modelo ODP, a linguagem de engenharia define mecanismos e funções para suportar uma interação distribuída entre objetos num sistema ODP. No entanto, para que esses objetos cooperem entre si, alcançando transparência na distribuição, o modelo propõe estruturas como nó, cápsula, *cluster*, e funções que podem ser combinadas para formar especificações capazes de orientar a construção de aplicações num ambiente aberto e heterogêneo. Essas funções dizem respeito a:

- gerenciamento;
- coordenação;
- repositório;
- segurança.

Dentro desse contexto, decidiu-se nesta etapa do trabalho pela implementação das chamadas funções de gerenciamento a nível de nó, cápsula, *cluster* e objeto, por serem consideradas serviços básicos num sistema caracterizado como ODP. Sendo assim, os serviços oferecidos em tempo de execução possibilitam:

- instanciação e gerenciamento de objetos;
- *clustering* (agregação) de objetos e gerenciamento de *cluster*;
- encapsulamento dos *clusters* e gerenciamento de cápsula;
- gerenciamento de nó.

3.2.1 Instanciação e gerenciamento de objetos

O conceito de objetos aqui está relacionado com a noção de objetos definida pela arquitetura CORBA [CORBA95], ou seja, um objeto é uma entidade encapsulada, com identificador único que provê um ou mais serviços. Estes serviços são descritos pelas interfaces dos objetos.
Na arquitetura CORBA, cada objeto está associado a uma interface especificada em IDL.

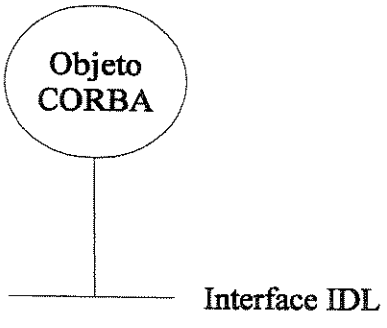

Figura 3-1 Um objeto na implementação

Entretanto, o modelo ODP define que um objeto pode possuir múltiplas interfaces. Sendo assim, para manter a compatibilidade com o padrão ODP, do ponto de vista de engenharia, um objeto básico na proposta é representado:

- por um objeto de gerenciamento (gob), que possui uma interface IDL com a descrição das funções de gerenciamento, representando a interface de gerenciamento do objeto;

- por objetos de aplicação com interfaces definidas pelo usuário representando as múltiplas interfaces que podem ser adicionadas a um objeto básico de engenharia (OBE).

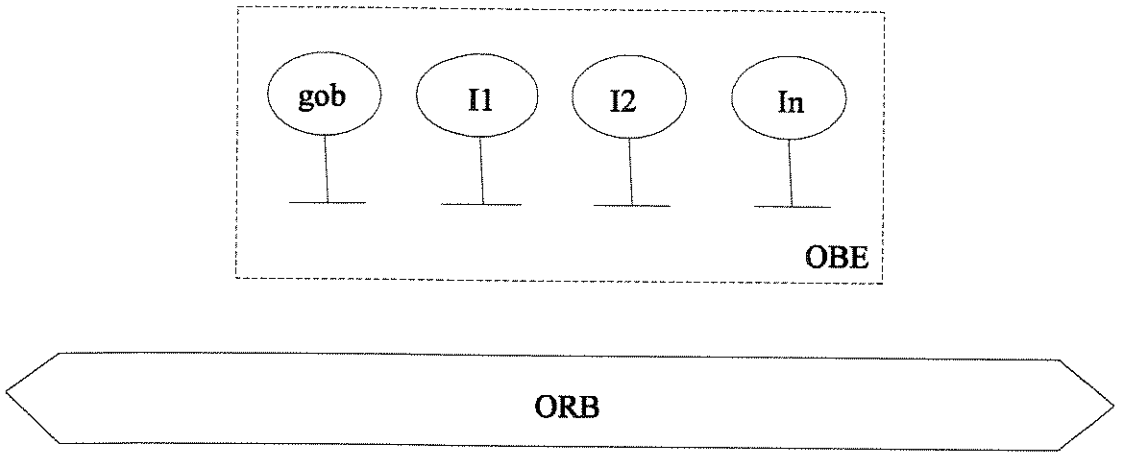

Figura 3-2 Um OBE na implementação

Não foi abordada a interação de objetos através de canais.
Os objetos aqui são gerenciados por processos chamados processos servidores. Num ambiente multi-*thread*, cada objeto é representado por *threads* em execução dentro de um processo maior que seria a cápsula. O gerenciamento dessas *threads* é de responsabilidade do implementador.
O estado de um objeto básico é então determinado pelos atributos do objeto de gerenciamento e dos objetos de aplicação definidos pelo usuário.
No ODP, um objeto é instanciado a partir de seu *template*, que é uma estrutura que descreve seus parâmetros, operações e seu comportamento, além de informações relativas às interfaces associadas a ele. Embora o ODP defina as funcionalidades do *template*, ele não determina como implementá-lo. A instanciação do *template* depende da especificação da linguagem usada e pode envolver outros *templates* ou interfaces existentes [RM-ODPb].
No trabalho realizado, um objeto é instanciado a partir da invocação remota ou local de um de seus métodos (ou operações, que representam os serviços fornecidos por ele) contidos na sua interface. O conceito de *template* é mapeado para uma *string* que

contém basicamente a informação necessária para identificar um objeto e executar algumas de suas funções de gerenciamento.

Figura 3-3 Estrutura do template na implementação

nome do objeto: identifica o objeto no sistema;

|: caracter delimitador da informação;

t1: indicador temporal

Esse indicador temporal será utilizado nas operações de *checkpoint*, recuperação, desativação, reativação e migração. Por exemplo, um objeto pode receber várias operações de *checkpoint* em instantes diferentes (t1, t2, t3 ...tn) durante o seu ciclo de vida, e depois ser removido. Caso seja necessário realizar uma reativação desse objeto, será necessário indicar a partir de qual instante se deve recuperar os dados referentes ao estado do objeto, ou seja, o objeto a ser restabelecido numa reativação (tn) será idêntico ao objeto que existia naquele instante (tn) quando foi realizado o *checkpoint*.

A nível de objeto foram desenvolvidas as seguintes funções de gerenciamento:

- addInterface: adiciona uma interface de aplicação ao gerente de objeto;
- getInterface: retorna uma lista de ponteiros para os identificadores das interfaces de aplicação associadas ao gerente de objeto;
- deleteInterface: remove a estrutura da interface de aplicação associada ao gerente de objeto;
- Delete: remove toda a estrutura relacionada a um objeto básico de engenharia (gerente do objeto e as interfaces de aplicação);
- *checkpoint*: realiza o *checkpoint* no gerente do objeto e chama o *checkpoint* sobre todas as interfaces associadas a ele;
- *recover*: responsável pela recuperação da estrutura do objeto básico de engenharia a partir de um determinado *checkpoint*.

3.2.2 *Clustering* (agregação) de objetos e gerenciamento de *cluster*

Clustering representa a agregação de objetos básicos (objetos de gerenciamento e objetos de aplicação) para o propósito de *checkpointing*, remoção, desativação, reativação, recuperação e migração. Estas funções de gerenciamento podem ser acessadas a partir da interface do objeto que gerencia o *cluster* (gcl), e são aplicadas a todos objetos associados a esse gerente, ou seja, a todos os objetos pertencentes ao *cluster*.

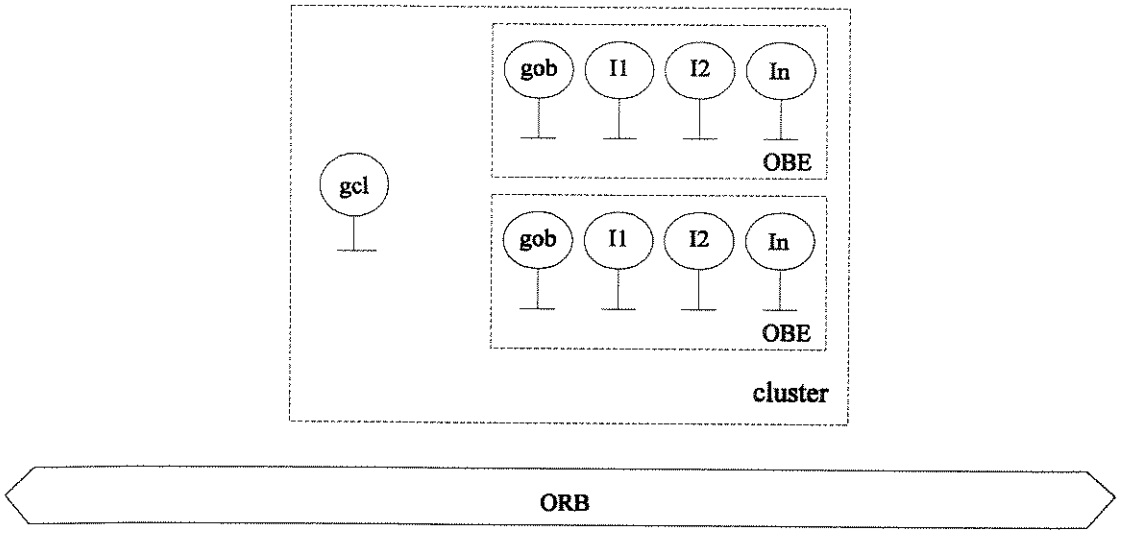

Figura 3-4 Ilustração de um cluster na implementação

O gerente de *cluster* (gcl) provê os seguintes métodos:

- makeObject: responsável pela criação de um gerente de objeto (gob);
- getObjects: retorna uma lista de ponteiros dos gerentes de objetos associados ao *cluster*;
- Delete: remove toda estrutura associada ao cluster;
- checkpoint: realiza o checkpoint sobre todos os objetos pertencentes ao *cluster*;
- recover: responsável pela recuperação da estrutura do *cluster* a partir de um determinado *checkpoint*;
- deactivate: realiza a desativação do *cluster*;
- migrate: promove a migração de um *cluster* para uma nova cápsula.

Em relação ao que propõe o ODP, não foi implementada uma função para monitorar a política de gerenciamento do *cluster*.

3.2.3 Encapsulamento dos *clusters* e gerenciamento de cápsula

Para o propósito de encapsulamento, os *clusters* podem estar associados a um objeto chamado gerente de cápsula (gcp), configurando assim uma cápsula. O gerente de cápsula será responsável por fornecer uma interface de gerenciamento a todos os *clusters* da cápsula, ou seja, qualquer operação de gerenciamento sobre a cápsula é refletida sobre todos os seus *clusters* e consequentemente, sobre seus objetos.

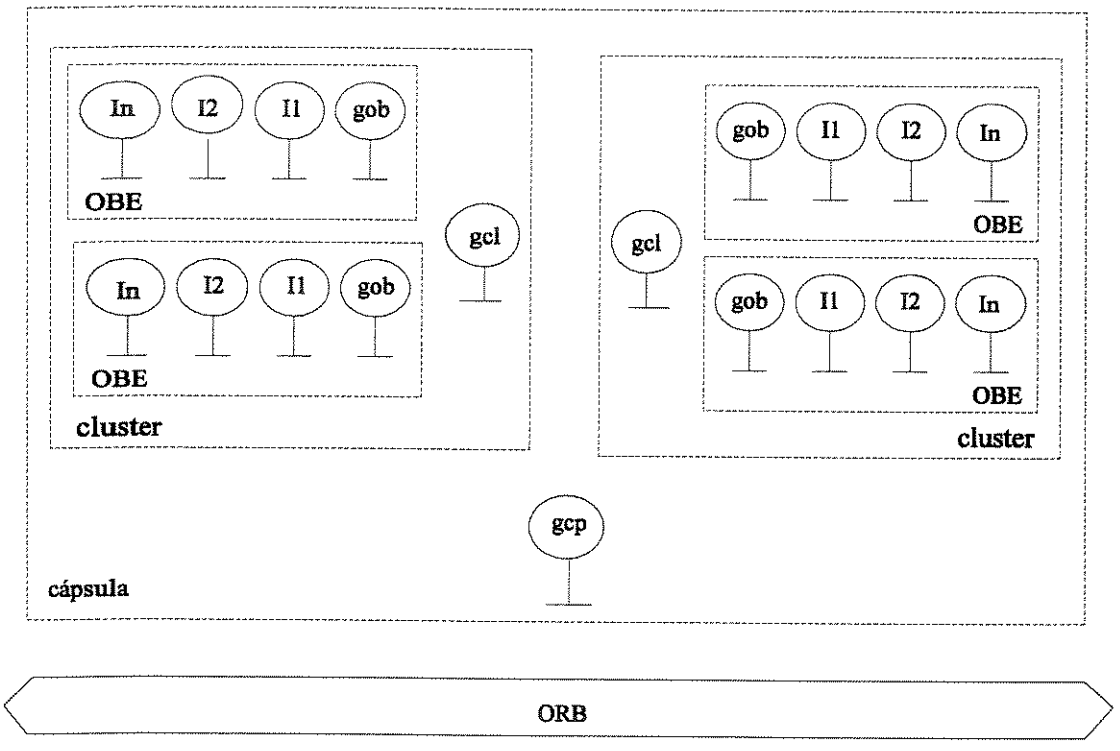

Figura 3-5 Ilustração de uma cápsula na implementação

As funções de gerenciamento presentes no gerente de cápsula (gcp) são:

- makeCluster: realiza a criação de um gerente de *cluster*;
- getClusters: retorna uma lista de ponteiros para os gerentes de *clusters* associados à cápsula;
- reactivate: responsável pela reativação de um *cluster* após sua desativação.

Não foram implementadas a nível de cápsula as funções de desativação e *checkpoint*.

3.2.4 Gerenciamento de nó

Enquanto no ODP o gerenciamento de nó é fornecido pelo núcleo, e controla processamento, armazenamento e comunicação, na implementação ele é provido por um objeto gerente de nó (gno) que tem como funcionalidades básicas:

- makeCapsule: cria um gerente de cápsula;
- getCapsules: retorna uma lista de ponteiros dos gerentes de cápsulas contidos num nó.

A comunicação na implementação é de responsabilidade do ORB, e as funções de armazenamento e processamento são desempenhadas por um servidor de base de dados (sbd) e um de transações (str).

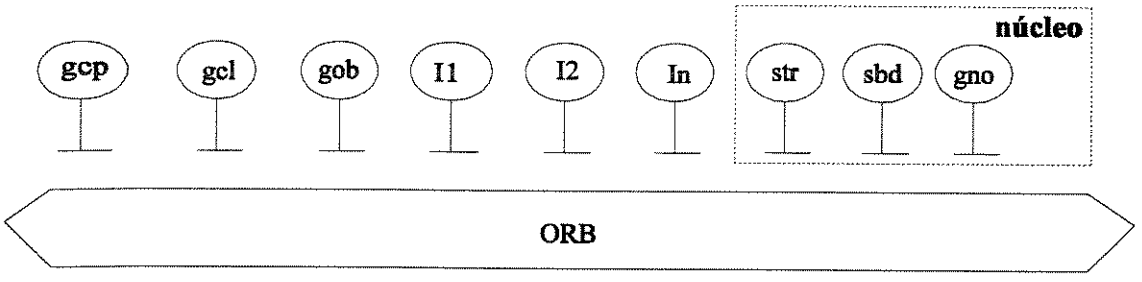

Figura 3-6 Estrutura de um nó na implementação

Não foram implementadas funções de gerenciamento de *threads*.

3.3 Descrição do sistema

Todas as funções de gerenciamento são acessadas através de interfaces especificadas em IDL. Cada interface é mapeada em um objeto que contém a implementação dos métodos (ou funções) nela descritos.
O sistema provê ainda um servidor de base de dados (sbd) e um de transações (str), que são utilizados por algumas funções de gerenciamento (*checkpoint*, *recover*), e que podem ser utilizados pelos usuários no desenvolvimento de suas interfaces de aplicação.
No entanto, algumas funcionalidades do sistema são de responsabilidade do usuário. Sabendo que as interfaces de aplicação serão definidas em tempo de execução e que será utilizado o processo de invocação estática [CORBA95], o usuário terá que fornecer ao sistema a informação necessária para identificar essas interfaces,

complementando na implementação do gerente de objeto, as funções addInterface, deleteInterface, checkpoint e recover.
Os procedimentos a nível de modificação de código a serem seguidos pelo usuário estão descritos no Apêndice A.

3.4 Transparência do sistema

Com relação ao ODP, o sistema oferece transparência de acesso, persistência, localização e transação.
As transparências de acesso e locação é alcançada através das interfaces IDL e do modo como a arquitetura CORBA trata o processo de invocação de objetos, mascarando detalhes da representação dos dados, mecanismos utilizados na instanciação e localização dos objetos.

A transparência de persistência é atingida através das funções de gerenciamento que esconde os recursos usados nos processos de desativação e reativação.

A transparência de transação é proporcionada pelo servidor de transações (str) que mascara mecanismos de sincronização e ordenação utilizados.

3.5 Modelo de objetos

A figura 3-7 representa o modelo de objetos proposto pela implementação na notação OMT (Object Modeling Technique) [Rumbaugh91].

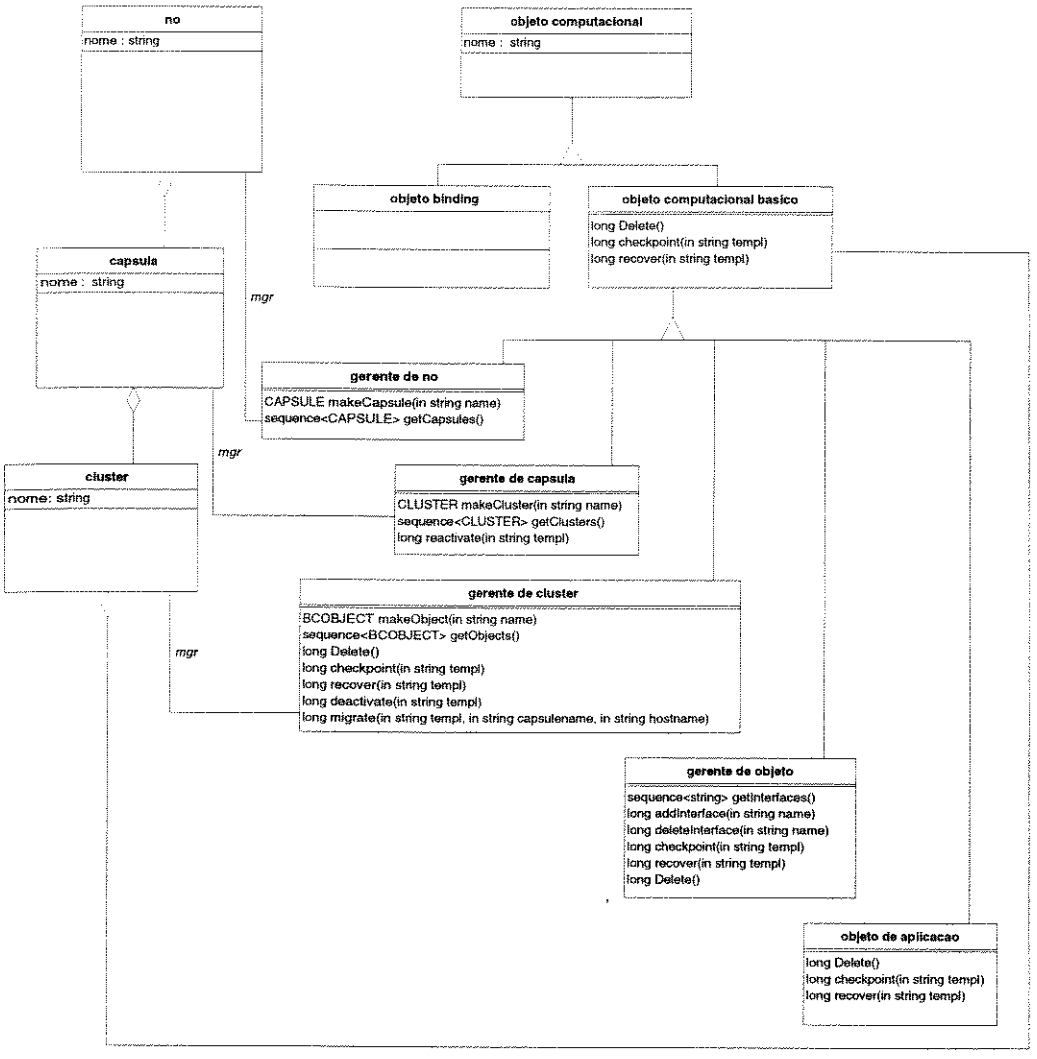

Figura 3-7 Modelo de objetos da implementação na notação OMT

Capítulo 4

Implementação

4.1 Introdução

A seguir serão apresentados os aspectos considerados na implementação das funções de gerenciamento ODP utilizando uma plataforma baseada na arquitetura CORBA.
É importante ressaltar que o trabalho apresenta algumas restrições diante da proposta apresentada, tendo em vista as dificuldades impostas pelos recursos utilizados. Por exemplo, a plataforma Orbix [Orbix95a,b,c] utilizada não executa num ambiente multi-*thread*, nem provê um serviço de persistência a nível de objeto. Sendo assim, as funções de gerenciamento que poderiam estar dentro de *threads*, são executadas por processos pesados do UNIX. Foi utilizado uma base de dados externa, no caso o Gdbm[Gdbm94], para a implementação de algumas das funções de núcleo relacionadas com o armazenamento persistente do estado dos objetos. Desenvolveu-se um protocolo de transações capaz de garantir consistência para esta base de dados.
Todavia, apesar das limitações encontradas, o trabalho procurou manter uma compatibilidade entre o padrão ODP e a arquitetura CORBA, justificando assim sua viabilidade.
Inicialmente, serão mostrados tópicos importantes da principal ferramenta utilizada, o Orbix. Posteriormente, serão apresentadas as definições das interfaces de gerenciamento de nó, cápsula, *cluster* e objeto, com a descrição de suas funcionalidades, e outros aspectos relevantes considerados durante a implementação.

4.2 Orbix

4.2.1 Introdução

O Orbix é um ambiente poderoso para construção e integração de aplicações baseado na arquitetura CORBA. O Orbix reduz o custo de desenvolvimento de aplicações distribuídas e de integração dos componentes existentes. Isso permite um sistema ser construído como um conjunto de objetos que interagem, cada um com uma interface bem definida, facilitando a comunicação entre eles.

Um sistema de computação distribuída é um conjunto de componentes de software executando em vários computadores conectados em rede. Estes componentes podem ser padrão, freqüentemente compartilhados por várias aplicações; uma aplicação pode também ser constituída por um número de componentes específicos. Um exemplo simples consiste de um processo que interage com um número de componentes compartilhados.
Em Orbix, os componentes de um programa distribuído são objetos. Cada objeto está associado com um processo servidor - um servidor gerencia um conjunto de objetos com interfaces iguais ou diferentes. Cada objeto Orbix internamente tem um único identificador no sistema, que pode ser utilizado para localizá-lo dentro do sistema distribuído. Cada servidor pode ter qualquer número de clientes que comunicam com seus objetos.
Em um programa convencional, não distribuído, o mecanismo usual para conectar diferentes módulos de um programa é uma chamada de função ou procedimento. Como este mecanismo é bem entendido, é razoável de se esperar que qualquer facilidade para aplicações de programação distribuída permita que chamadas de função ou procedimento continuem a ser usadas para conectar componentes. Entretanto, para um programa distribuído, os vários componentes podem ser remotos. Logo, deve ser possível fazer chamadas de função e procedimentos remotos, nas quais o chamador está (possivelmente) em uma máquina separada do componente chamado.
O Orbix estende este modelo permitindo que objetos em um sistema distribuído possam ser usados como se fossem locais ao chamador. O paradigma de orientação a objetos é adotado, de forma que esses são componentes distribuídos.
Um programa C++ executando sobre o Orbix pode invocar funções sobre um objeto em qualquer nó num sistema distribuído. Isso é atingido com um alto grau de transparência porque um programador C++ pode fazer uma chamada de função para invocar um objeto remoto.

4.2.2 Componentes do Orbix

4.2.2.1 Interfaces

Em um programa distribuído, é possível implementar diferentes objetos em diferentes linguagens de programação. Para facilitar a interação entre tais objetos, é comum abstrair a funcionalidade fornecida por cada objeto definindo sua interface. A linguagem de definição de interfaces, IDL, é uma linguagem padrão definida pela OMG para construir tais interfaces.
Definida a interface de um objeto em IDL, o próximo passo é implementar o objeto usando uma linguagem de programação. A linguagem utilizada para implementar um objeto pode diferir daquela utilizada para implementar um cliente usando aquele objeto.

4.2.2.2 Clientes e servidores

Um sistema de software distribuído consiste de objetos executando dentro de clientes e servidores. Servidores provêem objetos para serem usados por clientes e por outros servidores. Quando um objeto chama uma operação sobre outro, o primeiro é definido como objeto cliente, e o outro como objeto servidor.
Um cliente pode conter objetos, e qualquer um desses objetos pode ser conhecido por outros clientes e servidores, e consequentemente usado por eles.

4.2.2.3 Classes IDL C++

Uma interface IDL consiste de especificações de operações e atributos. O compilador IDL pode ser usado para produzir uma classe C++ correspondente a cada interface IDL. Isso deve ser feito antes que a interface seja implementada ou utilizada pelo código C++. Na terminologia Orbix, tais classes C++ são chamadas classes IDL C++. Uma classe IDL C++ lista as funções que clientes da interface podem usar, e estas funções devem ser definidas em C++ pelo implementador da interface.
Uma vez que a interface é implementada, qualquer número de objetos pode ser criado para aderir à interface. Seus objetos, embora objetos C++, são especiais porque eles podem ser usados de qualquer nó em um ambiente distribuído.

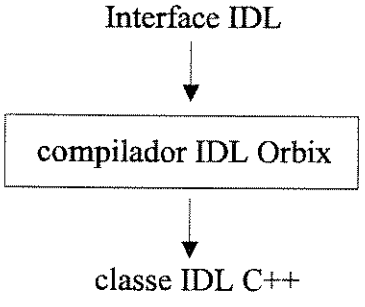

Figura 4-1 Compilação de uma interface IDL

4.2.2.4 Ponto de vista do cliente

Um ponteiro para uma classe IDL C++ pode denotar um objeto que é local (está no mesmo espaço de endereçamento de memória) ou remoto (está num espaço de endereçamento diferente - num nó diferente ou no mesmo nó). Quando o objeto é remoto, o ponteiro referirá um objeto *proxy*. Um objeto *proxy* é uma representação local de um objeto remoto, que implementa o suporte necessário para enviar pedidos para o objeto remoto. Todos pedidos feitos pelo cliente serão direcionados automaticamente pelo *proxy* para o objeto remoto. Objetos *proxy* são assim usados para suportar clientes de um objeto remoto, e o programador de um cliente não necessita estar ciente da sua existência.

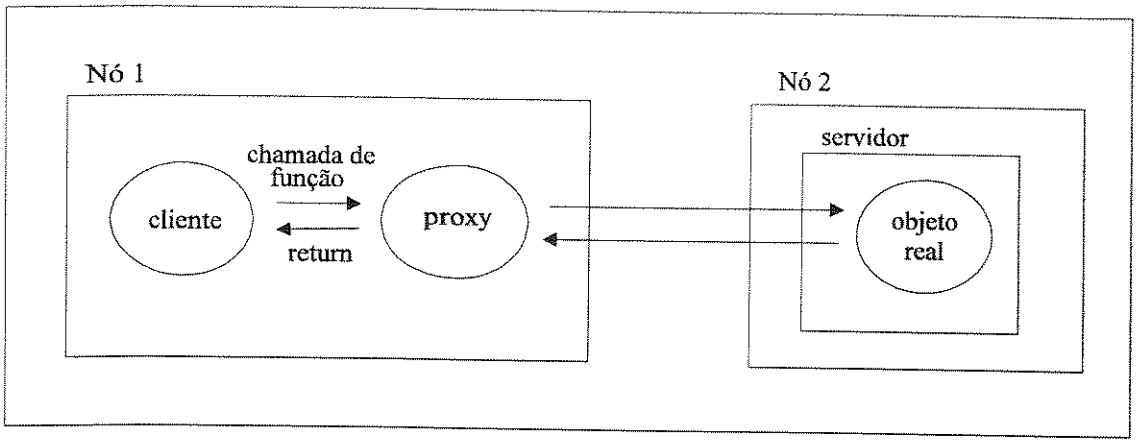

Figura 4-2 Chamada remota via *proxy*

4.2.2.5 Implementação de uma interface

Cada interface IDL deve ser implementada por uma classe C++. Esta classe C++ deve implementar no mínimo cada uma das funções que correspondem às operações e atributos IDL. Instâncias de tal classe correspondem a objetos Orbix. A classe de implementação é diferente da classe IDL C++, que é automaticamente gerada da especificação da interface IDL pelo compilador IDL, pois ela pode prover mais funcionalidades.

4.2.2.6 Servidores e o repositório de implementação

Todo servidor tem um nome que é único dentro do nó onde executa, mas o mesmo nome de servidor pode ocorrer em diferentes nós. Um servidor pode consistir de um ou mais processos.
Os objetos que são gerenciados por um servidor não necessitam ser da mesma classe de implementação, ou da mesma classe IDL C++. Assim, um servidor pode suportar diferentes interfaces IDL.
O Orbix provê também um repositório de implementação, que mantém um mapeamento do nome do servidor para o nome do arquivo do código executável que implementa o servidor. O programador de um servidor deve registrar seu código no repositório de implementação. Consequentemente, este código será automaticamente ativado (se já não o está) pelo Orbix quando uma invocação de uma função é feita para qualquer objeto cujo identificador nomeia aquele servidor em particular. Servidores diferentes podem usar o mesmo código executável.
Existem também vários mecanismos ou modos de ativação de servidores, dando ao programador controle sobre como os servidores são implementados como processos pelo sistema operacional. O modo de um servidor é especificado quando ele está sendo registrado, e pode ser:

- modo de ativação compartilhado: neste modo, todos os objetos com o mesmo nome do servidor em uma máquina, são gerenciados pelo mesmo processo naquela máquina. Este é o modo mais comum. Se o processo já está ativado quando uma invocação de função chega para um de seus objetos, então o Orbix roteará a invocação para aquele processo; caso contrário, o Orbix ativará o processo (usando o repositório de implementação para determinar qual arquivo executável utilizar).

- modo de ativação não compartilhado: neste modo, objetos individuais de um servidor são registrados no repositório de implementação. Quando cada objeto é invocado, um processo individual é criado para executar o código executável para aquele objeto particular - um processo é criado para cada objeto registrado ativo. Cada objeto gerenciado por um servidor pode ser registrado com um arquivo executável diferente.

- modo de ativação por método: neste modo, nomes dos métodos são registrados no repositório de implementação. Chamadas inter-processo podem ser feitas para estas operações - e cada invocação resultará na criação de um processo individual. Um processo é criado para manipular cada chamada de operação individual, e é destruído uma vez que a operação foi completada.

Para cada um desses modos, um servidor pode também ser ativado num modo de ativação por cliente ou modo de ativação múltiplos clientes:

- por cliente: neste modo, ativações do mesmo servidor por diferentes usuários finais resultarão em diferentes processos a serem criados para cada usuário.

- múltiplos clientes: neste modo, ativações do mesmo servidor por diferentes usuários finais compartilharão o mesmo processo de acordo com modo de ativação fundamental selecionado.

Embora todo servidor deva ser registrado num repositório de implementação, objetos individuais não necessitam ser registrados. Somente aqueles objetos para os quais o Orbix deve ativar um processo (se um processo apropriado ainda não está executando) devem ser registrados. Além disso, se um servidor é registrado num modo compartilhado, então ele pode ser ativado manualmente (ou por algum mecanismo externo ao Orbix, por exemplo um *debugger*), antes que invocações sejam feitas sobre seus objetos.
Por *default*, cada processo tem uma simples *thread* que manipula os pedidos, isto é, o Orbix assegura que somente um pedido está ativo em um determinado momento no processo.

4.2.2.7 Repositório de interface

O repositório de interface provê informação em tempo de execução sobre a interface e definições de tipo. Dada uma referência para um objeto, a interface do objeto e toda a informação sobre ela pode ser determinada em tempo de execução, pela chamada de funções definidas pelo repositório de interface.

4.3 Decisões de implementação

4.3.1 Apresentação do sistema

Numa visão macro do sistema, cada objeto gerente é construído de forma a armazenar informações necessárias para identificar os objetos a ele associados e com isso, executar as funções de gerenciamento.

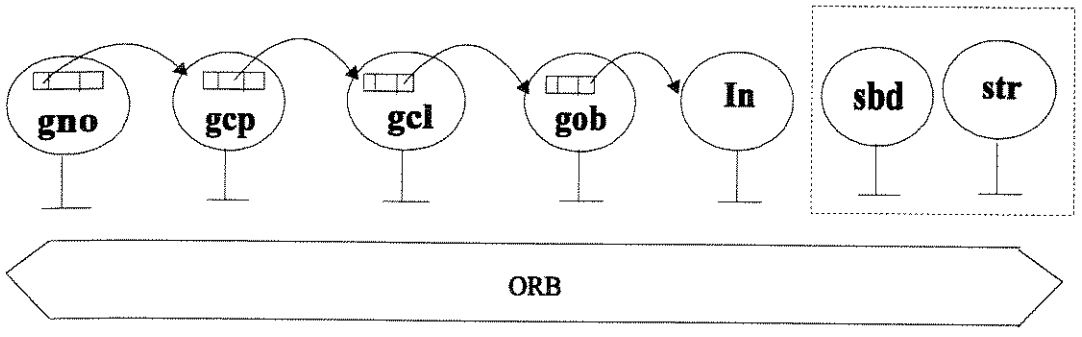

Figura 4-3 Uma visão macro do sistema

Por exemplo, o gerente de cápsula mantém a variável lista_de_clusters, que identifica uma lista que contém os ponteiros dos gerentes de *clusters* associados a ele, bem como seus identificadores.
O gerente de *cluster* mantém a variável lista_de_objetos, que identifica uma lista que contém os ponteiros dos gerentes de objetos associados e seus identificadores.
O gerente de objeto possui a variável lista_de_interfaces, que aponta para uma lista dos identificadores (nomes) das interfaces (ou objetos de aplicação) associadas a ele.

O gerente de nó também possui uma lista de todos os ponteiros e identificadores dos gerentes de cápsula presentes no nó (lista_de_cápsulas).

Todos esses objetos gerentes utilizam as funções de suporte para armazenamento de dados, do servidor de base de dados (sbd), e as funções para processamento de transações do servidor de transações (str).

4.3.2 Identificação dos objetos

O Orbix associa internamente um identificador único a cada objeto. Entretanto, ele também fornece um serviço de nomeação através da função _marker(char *), que permite que o objeto seja identificado por uma *string*. Sendo assim, na implementação quando um servidor recebe uma invocação de uma aplicação cliente para criar um objeto gerente, é passado também uma *string* representando o nome (que deve ser único) do objeto, através do qual ele poderá ser acessado pelos usuários. A responsabilidade pela unicidade dos nomes atribuídos aos objetos gerentes é toda do cliente.

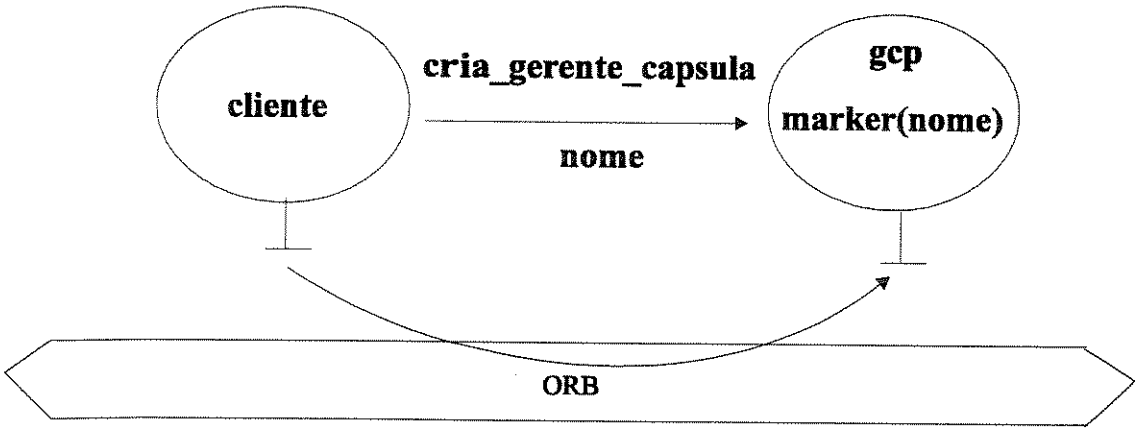

Figura 4-4 Nomeação dos objetos

4.3.3 Realização do *checkpoint*

O *checkpoint* é responsável por armazenar de uma forma persistente o estado do objeto. Uma operação de *checkpoint* no *cluster* é uma ação recursiva, pois reflete sobre o gerente do *cluster* e todos os objetos associados a ele.

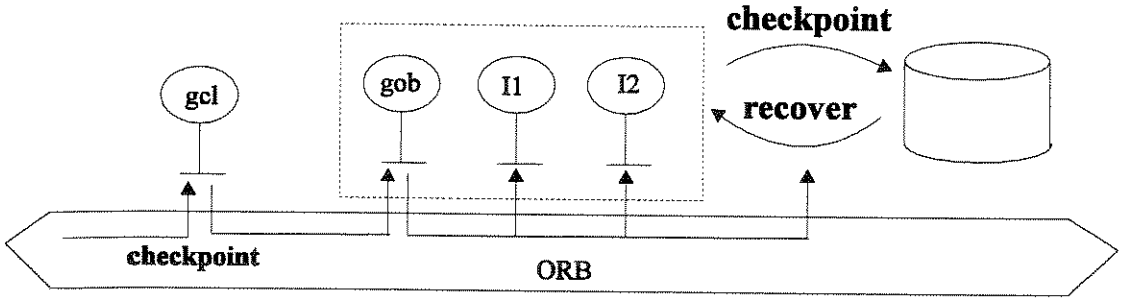

Figura 4-5 *Checkpoint* de um *cluster*

No *checkpoint* do gerente de *cluster* são armazenados as seguintes variáveis de estado:

- lista_de_objetos: contém os identificadores (nomes) dos gerentes de objetos;
- nobjetos: indica o número de gerentes de objetos presentes no *cluster*;
- hostname: determina o nó em que se encontra o gerente do *cluster*.

No *checkpoint* do gerente de objeto são armazenados:

- lista_de_interfaces: contém os nomes das interfaces (objetos de aplicação) associadas ao objeto;
- ninterfaces: indica o número de interfaces ligadas ao objeto;
- hostname: determina o nó em que se encontra o gerente do objeto.

Com relação às interfaces (ou objetos) de aplicação, fica sob responsabilidade do usuário a implementação da operação de *checkpoint*, bem como a definição das variáveis de estado a serem armazenadas.
Um objeto na implementação pode sofrer vários *checkpoints* durante o seu ciclo de vida. O instante de realização do *checkpoint* é uma informação que pode ser usada para recuperar o estado específico de um objeto naquele determinado momento. É o que ocorre com a utilização da estrutura do *template* apresentada no capítulo anterior ("nomeIt1"). O *template* funciona como uma chave de busca no arquivo de dados quando, por exemplo, uma operação de recuperação é requerida. O objeto recuperado, identificado pelo campo "nome", representa o mesmo objeto existente no instante "t1" da realização do *checkpoint*.
Toda operação de recuperação, seja a nível de gerente de *cluster*, gerente de objeto ou interface de aplicação, deve restabelecer o estado existente do objeto no instante do *checkpoint*.
A figura 4-6 ilustra a estrutura de dados armazenada numa operação de *checkpoint* de um objeto gerente.

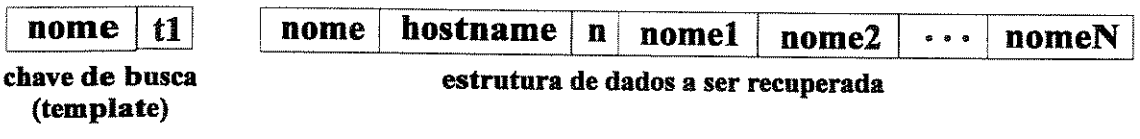

Figura 4-6 Estrutura de dados utilizada no *checkpoint* e na recuperação

Os parâmetros da estrutura são descritos abaixo:

- nome: identifica o objeto;

- t1: indicador temporal;
- hostname: nó onde se encontra o objeto gerente;
- contém o número de objetos associados ao objeto gerente;
- nome1 ... nomeN: identifica os nomes dos objetos associados ao objeto gerente.

4.3.4 Atualizações

Como os gerentes de *cluster* e de objeto podem sofrer uma operação de remoção, as listas mantidas pelo gerente de cápsula (lista_de_clusters) e pelo gerente de *cluster* (lista_de_objetos) devem ser cuidadosamente atualizadas. Essa atualização é feita através das funções updateCapsule e updateCluster, presentes nas interfaces de gerenciamento.
Quando um gerente de *cluster* recebe uma invocação de remoção através da operação Delete, antes que seja retornado um indicador de sucesso ou falha, é realizada uma chamada para o gerente de cápsula invocando a função updateCapsule, que irá retirar da lista de *clusters* (lista_de_clusters) o identificador do gerente de *cluster*. O mesmo ocorrerá com o gerente de objeto que deverá chamar a função updateCluster no gerente de *cluster* para atualizar a lista de objetos (lista_de_objetos).
A figura 4-7 mostra um exemplo do processo de atualização da lista de *clusters* mantida por um gerente de cápsula.

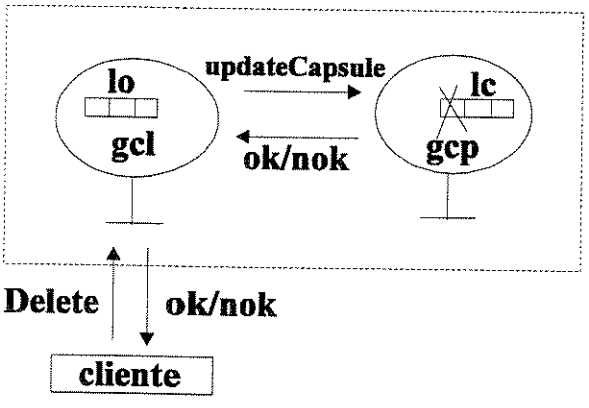

Figura 4-7 Atualização da lista de *clusters*

Na figura 4-7, as variáveis lo e lc indicam:

- lo: indica lista de objetos;
- lc: indica lista de *clusters*.

A figura 4-8 representa um diagrama de eventos da atualização da lista de *clusters*.

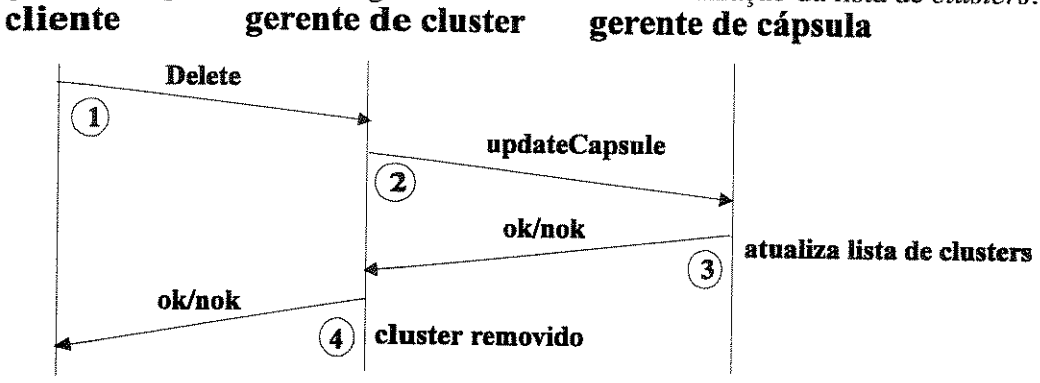

Figura 4-8 Diagrama de eventos da atualização da lista de *clusters*

4.3.5 Recompilação do sistema pelo usuário

Se uma aplicação cliente deseja utilizar os serviços de uma determinada interface e não possui uma referência para o objeto que a implementa, a maneira de consegui-la é através da função _bind() do Orbix, que tem como alguns de seus parâmetros o nome do objeto a ser instanciado, o nome do servidor que gerencia o objeto e o nome do nó onde o servidor será disparado. O nome do objeto, conforme já mencionado, o identifica e deve ser único no sistema. O valor retornado é um ponteiro para o objeto instanciado.

A função _bind() representa uma forma de invocação estática dos serviços da interface. Entretanto, para utilizá-la, o cliente deve ter acesso as *stubs* IDL gerados a partir da compilação da interface. Como algumas interfaces podem ser adicionadas ao repositório em *runtime* (interfaces de aplicação definidas pelo usuário), torna-se impossível invocá-las estaticamente sem uma nova recompilação do sistema.

A solução para o problema apresentado anteriormente seria o processo de invocação dinâmica, que permitiria a requisição de pedidos para interfaces definidas em *runtime*. No entanto, como esse processo de invocação necessita da referência do objeto que implementa a interface, e esse objeto pode vir a ser removido como resultado de uma operação de desativação, essa alternativa não é viável.

Sendo assim, optou-se pela utilização de invocações estáticas, ficando sob responsabilidade do usuário a recompilação do sistema, para que ocorra o acesso às interfaces definidas em *runtime*. Para que um objeto gerente possa operar sobre as interfaces de aplicação (ou objetos de aplicação) definidas pelo usuário, ele precisa conter a informação necessária (stubs IDL) para ativá-las, e para chamar as funções de gerenciamento (Delete, checkpoint, recover) que devem estar definidas nestas interfaces, conforme proposto no modelo computacional apresentado no capítulo anterior. Logo, o usuário deverá modificar a nível de código, as funções definidas no gerente de objeto (addInterface, deleteInterface, checkpoint, recover), acrescentando as chamadas _bind() para as interfaces e as invocações para as funções de gerenciamento citadas.

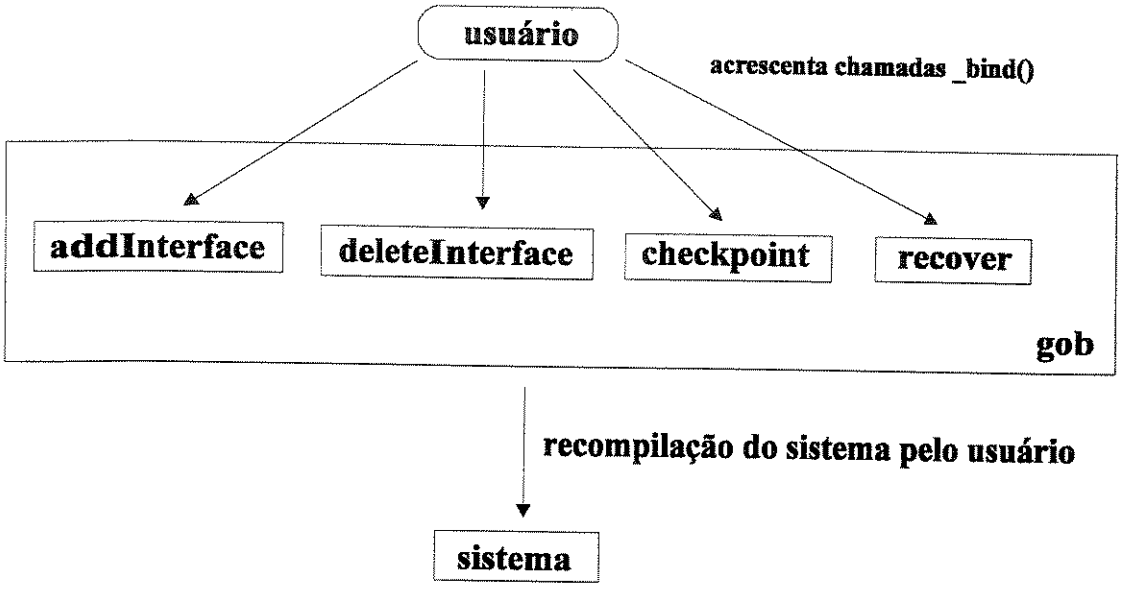

Figura 4-9 Funções que deverão ser modificadas em *runtime* pelo usuário

A seguir serão apresentados alguns diagramas de eventos ilustrando como as modificações introduzidas pelo usuário influenciam na execução das funções addInterface, deleteInterface, checkpoint e recover.

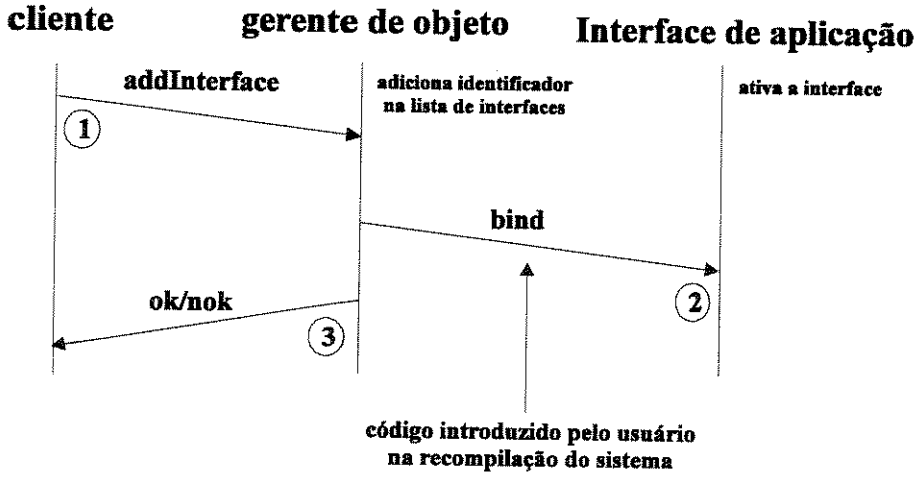

Figura 4-10 Diagrama de eventos envolvendo a função addInterface

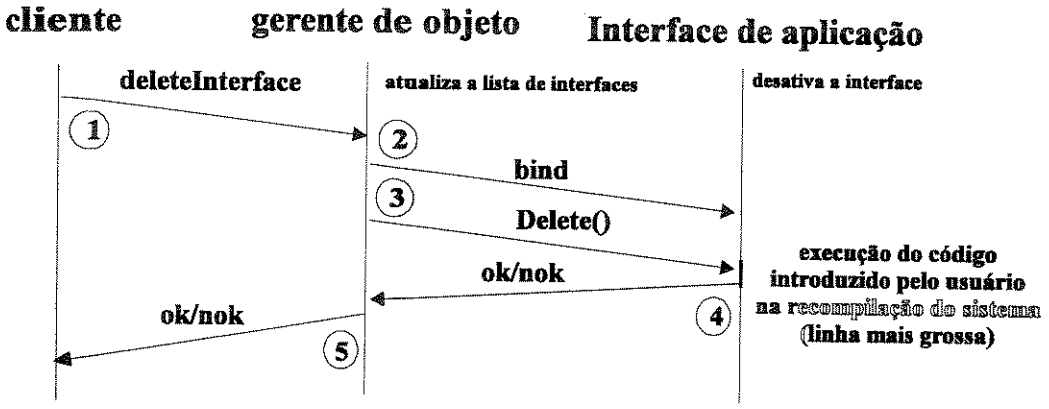

Figura 4-11 Diagrama de eventos envolvendo a função deleteInterface

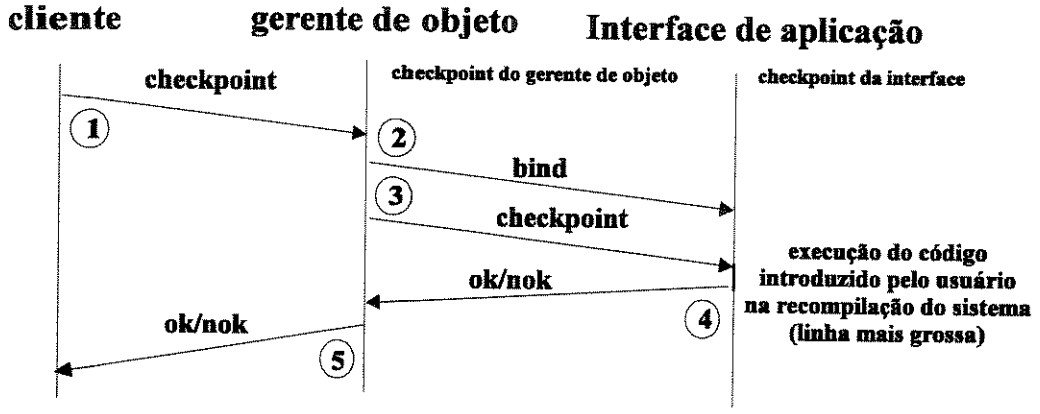

Figura 4-12 Diagrama de eventos envolvendo a função checkpoint

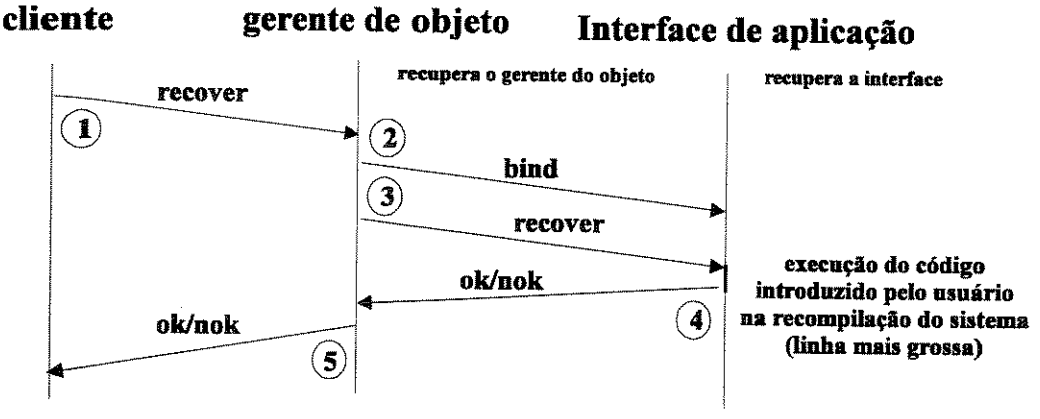

Figura 4-13 Diagrama de eventos envolvendo a função recover

4.4 Interfaces de gerenciamento

A seguir, serão apresentadas as interfaces de gerenciamento e uma descrição das suas funções, bem como um mapeamento para C++ ilustrando como essas funções podem ser acessadas pelas aplicações clientes.
As definições IDL das interfaces de gerenciamento também podem ser encontradas no Apêndice A.

4.4.1 Interface do gerente de nó

A interface IDL de gerenciamento que um objeto gerente de nó possui é da seguinte forma:

```
interface NO {

        CAPSULE makeCapsule(in string name);
        sequence<CAPSULE> getCapsules();

};
```

onde:

- CAPSULE makeCapsule(in string name): é responsável pela criação de um gerente de cápsula, ou seja, um objeto com a interface de gerenciamento de cápsula. O parâmetro de entrada nomeia o objeto e funciona como seu identificador único. O valor retornado representa uma referência para o objeto instanciado e a partir dele pode-se acessar todos os métodos descritos na sua interface.

Em C++:

```
NO *gno;
CAPSULE *gcp;
char *capname = "GCP";

gno = NO::_bind(":nodeserver","");  // obtém ponteiro para o gerente de nó
gcp = gno->makeCapsule(capname); // obtém ponteiro para o gerente de cápsula
```

- sequence<CAPSULE> getCapsules(): retorna um vetor de ponteiros para gerentes de cápsula, ou seja, mantém uma lista de todos os gerentes de cápsulas associadas ao nó.

Em C++:

```
_IDL_SEQUENCE_CAPSULE lista_de_capsulas = gno->getCapsules();
```

4.4.2 Interface do gerente de cápsula

Cada gerente de cápsula possui uma interface do tipo:

```
interface CAPSULE {

        CLUSTER makeCluster(in string name, in string capsulename);
        sequence<CLUSTER> getClusters();
        long reactivate(in string templ);
};
```

onde:

- CLUSTER makeCluster(in string name, in capsulename): é responsável pela criação de um gerente de *cluster*, ou seja, um objeto com a interface de gerenciamento de *cluster*. O parâmetro de entrada name, funciona como identificador único do gerente de *cluster*. O parâmetro capsulename identifica o gerente de cápsula, e associa o *cluster* a uma determinada cápsula. O valor retornado representa uma referência para um gerente de *cluster* instanciado e a partir dele pode-se acessar todos os métodos descritos na sua interface.

Em C++:

```
CLUSTER *gcl;
char *cluname = "GCL";
gcl = gcp->makeCluster(cluname,capname);
```

- sequence<CLUSTER> getClusters(): retorna um vetor de ponteiros para gerentes de *cluster*, ou seja, mantém uma lista de todos os gerentes de *clusters* associados à cápsula.

Em C++:

```
_IDL_SEQUENCE_CLUSTER lista_de_clusters = gcp->getClusters();
```

- long reactivate(in string templ): responsável pela reativação de um *cluster*, após sua desativação. É uma função desempenhada pelo gerente de cápsula, tendo em vista que numa eventual desativação do *cluster* o seu gerente será removido. Dessa forma, será necessário aqui a criação de um novo gerente de *cluster*. O parâmetro de entrada contém o identificador do gerente de *cluster* e indica a partir de qual *checkpoint* se deve fazer a sua reativação. O valor retornado representa se a operação teve sucesso (1) ou não (0).

Em C++:

```
long t1 = 9999;
char clutempl[64];
sprintf(clutempl, "%s%s",cluname,"lt1");
gcp->reactivate(clutempl);
```

4.4.3 Interface do gerente de *cluster*

Cada gerente de *cluster* possui uma interface do tipo:

```
interface CLUSTER {

        BCOBJECT makeObject(in string name, in string clustername);
        sequence<BCOBJECT> getObjects();
        long Delete();
        long checkpoint(in string templ);
        long recover(in string templ);
        long deactivate(in string templ);
        long migrate(in string templ, in string hostname);

};
```

onde:

- BCOBJECT makeObject(in string name, in string clustername): é responsável pela criação de um gerente de objeto, ou seja, um objeto com a interface de gerenciamento de um objeto básico de engenharia. O parâmetro de entrada name, funciona como identificador único do gerente do objeto. O parâmetro clustername identifica o gerente de *cluster* e associa o objeto básico a um determinado *cluster*. O valor retornado representa uma referência para um gerente de objeto instanciado e a partir dele pode-se acessar todos os objetos de aplicação definidos pelo usuário, e associados a ele.

Em C++:

```
char *objname = "GOB";
BCOBJECT *gob;
gob = gcl->makeObject(objname,cluname);
```

- sequence<BCOBJECT> getObjects(): retorna um vetor de ponteiros para gerentes de objeto, ou seja, mantém uma lista de todos os gerentes de objetos associados ao *cluster*.

Em C++:

```
_IDL_SEQUENCE_BCOBJECT lista_de_objetos = gob->getObjects();
```

- long Delete(): remove toda a estrutura de suporte ao *cluster*, o que na implementação realizada corresponde a remoção do gerente de *cluster*, e de todos os objetos (de gerenciamento e aplicação) associados a ele. A remoção implica também na atualização da lista de *clusters* mantida pela cápsula. O valor retornado representa se a operação teve sucesso (1) ou não (0).

Em C++:

```
if (gcl->Delete())
        cout << "Cluster deletado sem problemas " ;
```

- long checkpoint(in string templ): é responsável pelo serviço de persistência do *cluster*, ou seja, pelo armazenamento em disco do estado de todos os objetos associados a ele. Para isso é utilizado o servidor de base de dados (sbd). O parâmetro de entrada contém o identificador do gerente de *cluster* que vai receber o *checkpointing* e vai servir como chave de busca no arquivo de dados. Contém também o instante do *checkpoint* realizado. A partir do gerente de *cluster*, é propagado o *checkpoint* para os outros objetos. O valor retornado representa se a operação teve sucesso (1) ou não (0). Se um *checkpoint* sobre algum dos objetos falhar, a operação sobre o *cluster* será abortada.

```
if (gcl->checkpoint(clutempl))
        cout << "Cluster checkpointed";
```

- long recover(in string templ): responsável pela recuperação em disco do estado do gerente de *cluster* e de todos os objetos associados a ele. Aqui também é utilizado o sbd. Inicialmente, quando um gerente de *cluster* recebe uma invocação de um *recover*, todos os objetos ligados a ele são removidos. O parâmetro de entrada serve como chave de busca para o gerente de *cluster* no arquivo de dados, e indica a partir de qual *checkpoint* deve-se fazer a recuperação. Após a recuperação do gerente de *cluster*, é propagado um *recover* para os outros objetos pertencentes ao *cluster* no instante que foi realizado o *checkpoint*. O valor retornado representa se a operação teve sucesso (1) ou não (0). Se um *recover* sobre algum dos objetos falhar, a operação sobre o *cluster* será abortada. Vale ressaltar também que no caso da remoção do gerente de *cluster*, a recuperação do *cluster* se faz através do gerente de cápsula, invocando uma operação de reativação do *cluster*.

Em C++:

```
if (gcl->recover(clutempl))
        cout << "Cluster recovered";
```

- long deactivate(in string templ): responsável pela desativação do *cluster*, o que equivale a realização de um *checkpoint* no *cluster*, seguido de sua remoção. O parâmetro de entrada é utilizado na operação de *checkpoint*. O valor retornado representa se a operação teve sucesso (1) ou não (0).

Em C++:

```
if (gcl->deactivate(clutempl))
        cout << "Cluster deactivated";
```

- long migrate(in string templ, in string capsulename, in string hostname): promove a migração de um *cluster* para uma nova cápsula, podendo a cápsula estar localizada localmente (no mesmo nó) ou remotamente. Entretanto, para a realização da migração, considera-se que o *cluster* deve estar desativado, ou seja, do ponto de vista da implementação, somente os *clusters* desativados são passíveis de sofrerem migração. O primeiro parâmetro de entrada corresponde as informações necessárias para se realizar a desativação do *cluster*. O segundo parâmetro indica o nome da cápsula para a qual o *cluster* vai ser migrado e o terceiro identifica o nome da máquina onde está localizada a cápsula. O valor retornado representa se a operação teve sucesso (1) ou não (0).

Em C++:

```
char *capid = "GCP2";
char *hostname = "trancoso.dca.fee.unicamp.br";
if (gcp->migrate(clutempl,capid,hostname))
        cout << "Cluster migrated";
```

4.4.4 Interface de gerenciamento de objeto

Cada gerente de objeto possui uma interface do tipo:

```
interface BCOBJECT {

        long addInterface(in string name);
        sequence<string> getInterfaces();
        long deleteInterface(in string name);
        long Delete();
        long  checkpoint (in string templ);
        long  recover(in string templ);

};
```

onde:

- long addInterface(in string name): adiciona uma interface a um objeto de gerenciamento. O conceito de interface na implementação está associado aos

objetos de aplicação definidos pelo usuário. Dessa forma, as interfaces (ou objetos) de aplicação estão relacionadas a um objeto gerente, cuja finalidade é prover a essas interfaces (ou objetos), as funções de gerenciamento ODP. O parâmetro de entrada identifica a interface (ou objeto) de aplicação. O valor retornado representa se a operação teve sucesso (1) ou não (0).

Em C++:

Supondo que o usuário tenha definido uma interface chamada teste:

```
char *internname = "teste";
gob->addInterface(internname);
```

- sequence<string> getInterfaces(): retorna um vetor de ponteiros para os identificadores das interfaces (ou objetos) associadas ao gerente de objeto. Esses identificadores são utilizados no processo de instanciação das interfaces (ou objetos) de aplicação.

Em C++:

```
_IDL_SEQUENCE_string lista_de_interfaces = gob->getInterfaces();
```

- long deleteInterface(in string name): remove a estrutura da interface (ou objeto) de aplicação definida pelo usuário. O parâmetro de entrada representa o identificador da interface a ser removida. O valor retornado representa se a operação teve sucesso (1) ou não (0).

Em C++:

```
gob->deleteInterface(internname);
```

- long Delete(): remove toda a estrutura relacionada a um objeto básico de engenharia, ou seja, o gerente de objeto e as interfaces (ou objetos) associadas a ele. Esta operação resulta na atualização da lista de objetos mantida pelo gerente de *cluster*. O valor retornado representa se a operação teve sucesso (1) ou não (0). A operação deve estar contida nas interfaces de aplicação do usuário.

Em C++:

```
gob->Delete();
```

- long checkpoint (in string templ): realiza o *checkpoint* no gerente do objeto e chama o *checkpoint* sobre todas as interfaces associadas a ele, utilizando o sbd. O parâmetro de entrada contém o identificador do gerente de objeto que vai receber o *checkpoint* e vai servir como chave de busca no arquivo de dados. Contém também o instante do *checkpoint* realizado. O valor retornado representa se a operação teve sucesso (1) ou não (0). Se um *checkpoint* sobre alguma das

interfaces falhar, a operação sobre o objeto será abortada. A operação deve estar contida nas interfaces de aplicação do usuário.

Em C++:

```
long t1 = 9999;
char *objname = "OBJ";
char objtempl[64];
sprintf(objtempl,"%s%s",objname,"lt1");
if (gob->checkpoint(objtempl))
        cout << "Object checkpointed";
```

- long recover(in string templ): responsável pela recuperação em disco do estado do gerente de objeto e de todas as interfaces associadas a ele. Aqui também é utilizado o sbd. Inicialmente, quando um gerente de objeto recebe uma invocação de um *recover*, todos as interfaces ligadas a ele são removidas. O parâmetro de entrada serve como chave de busca para o gerente de objeto no arquivo de dados, e indica a partir de qual *checkpoint* deve-se fazer a recuperação. Após a recuperação do gerente de objeto, é propagado um *recover* para as interfaces que haviam sido adicionadas ao gerente até o momento do *checkpoint*. O valor retornado representa se a operação teve sucesso (1) ou não (0). Se um *recover* sobre alguma das interfaces falhar, toda a operação será abortada. A operação deve estar contida nas interfaces de aplicação do usuário.

Em C++:

```
if (gob->recover(objtempl))
        cout << Object recovered";
```

4.4.5 Classes de implementação

Para cada interface, o compilador IDL gera uma classe IDL C++ correspondente, constituindo parte da informação (stubs IDL) usada no processo de invocação dos objetos. Para cada classe IDL C++, existe uma classe de implementação no servidor provendo o código que implementa os serviços descritos na interface e outras funcionalidades não vistas pelo usuário. Por exemplo, a interface BCOBJECT gera uma classe IDL C++ BCOBJECT, que é mapeada pela classe de implementação BCOBJECT_i, cuja estrutura é apresentada a seguir:

```
class BCOBJECT_i {
public:
        _IDL_SEQUENCE_string *lista_de_interfaces;
        char *oid;
        char *hostname;
        char *clustername;
        long ninterfaces;
```

```
        BCOBJECT_i();
        ~BCOBJECT_i();

        void adiciona_interfaces(char *);

        virtual void clustermgr(const char *clustermgr);
        virtual char *clustermgr();
        virtual void objid(const char *objid);
        virtual char *objid();

        virtual _IDL_SEQUENCE_string getInterfaces();
        virtual long addInterface (const char *name);
        virtual long deleteInterface(const char *name);
        virtual long checkpoint(const char *templ);
        virtual long recover(const char *templ);
        virtual long Delete();

};
```

As variáveis de estado, o construtor, destrutor, e a função adiciona_interfaces não são acessadas pelo usuário via interface IDL.
As funções clustermgr e objid representam os atributos da interface BCOBJECT, e, embora possam ser acessadas pelo usuário via interface IDL, na implementação são utilizadas somente pelas funções de gerenciamento.

A interface CLUSTER tem seus serviços implementados pela classe CLUSTER_i.

```
class CLUSTER_i {
public:
        _IDL_SEQUENCE_BCOBJECT *lista_de_objetos;
        char *cid;
        char *hostname;
        char *capsulename;
        long nobjetos;

        CLUSTER_i();
        ~CLUSTER_i();

        void adiciona_objeto(BCOBJECT *);

        virtual void capsulemgr(const char *capsulemgr);
        virtual char *capsulemgr();
        virtual void cluid(const char*cluid);
        virtual char *cluid();
```

```
        virtual long updateCluster(const char *objname);
        virtual BCOBJECT *makeObject(const char *name, const char *clustername);
        virtual _IDL_SEQUENCE_BCOBJECT getObjects();
        virtual long Delete();
        virtual long checkpoint(const char *templ);
        virtual long recover(const char *templ);
        virtual long deactivate(const char *templ);
        virtual long migrate(const char *templ);

};
```

As variáveis de estado, o construtor, destrutor, e a função adiciona_objeto não são acessadas pelo usuário via interface IDL.
As funções capsulemgr e cluid representam os atributos da interface CLUSTER, e são utilizadas na implementação somente pelas funções de gerenciamento.
A função updateCluster é utilizada pelo gerente de objeto para atualizar a lista de objetos mantida pelo *cluster*.

A classe CAPSULA_i implementa a interface CAPSULA.

```
class CAPSULE_i {
public:
        char *idcap;
        _IDL_SEQUENCE_CLUSTER *lista_de_clusters;
        char *hostname;
        long nclusters;


        CAPSULE_i();
        ~CAPSULE_i();

        void adiciona_cluster(CLUSTER *);

        virtual long updateCapsule(const char *clustername);

        virtual void capid(const char capid);
        virtual char *capid();

        virtual CAPSULE *create_capsule(const char *name);
        virtual CLUSTER *makeCluster(const char *name, const char *capsulename);
        virtual _IDL_SEQUENCE_CLUSTER getClusters();
        virtual long reactivate(const char *templ);

};
```

As variáveis de estado, o construtor, destrutor, e a função adiciona_cluster não são acessadas pelo usuário via interface IDL.

A função capid representa o atributo da interface CAPSULE, e não é utilizada pelo usuário.
A função create_capsule é utilizada pelo objeto gerente de nó.
A função updateCapsule é usada pelo objeto gerente de *cluster* para atualizar a lista de *clusters* mantida pelo gerente de cápsula.

A classe NODE_i corresponde à interface NODE.

```
class NODE_i {
public:
        _IDL_SEQUENCE_CAPSULE *lista_de_capsulas;
        char *hostname;
        long ncapsulas;

        NODE_i();
        ~NODE_i();

        void adiciona_capsula(CAPSULE *);

        virtual CAPSULE *makeCapsule(const char *name);
        virtual _IDL_SEQUENCE_CAPSULE getCapsules();

};
```

As variáveis de estado, o construtor, destrutor, e a função adiciona_capsula não são acessadas pelo usuário via interface IDL.

4.4.6 Servidores

Todos os servidores estão registrados no modo *default*, ou seja, compartilhados, e para múltiplos clientes. A ativação é feita a partir da invocação de um método de algum dos objetos por ele gerenciados. Todos os servidores possuem um *timeout* infinito, o que significa que eles estão permanentemente esperando a requisição de um pedido por algum cliente.

4.4.7 Interfaces do usuário (objetos de aplicação)

As interfaces do usuário são as interfaces de aplicação definidas em tempo de execução. A implementação dessas interfaces é de responsabilidade do usuário. A identificação é feita de forma que o nome do objeto coincida com o nome da interface que ele implementa. Devem estar especificadas nestas interfaces as operações de *checkpoint*, *recover* e *Delete*. Essas operações são chamadas através do gerente de objeto associado a interface.

4.4.8 Funções do núcleo

Na implementação, as funções de núcleo propostas pelo ODP são desempenhadas parte pelo gerente de nó, e parte por dois servidores Orbix, o gnuserver e o ptserver, responsáveis respectivamente pelo armazenamento de dados e gerenciamento de transações. Esses servidores gerenciam objetos com interfaces especificadas em IDL, que descrevem os serviços que podem ser acessados. Ambos servidores também utilizam um servidor de números globais, o nglserver.

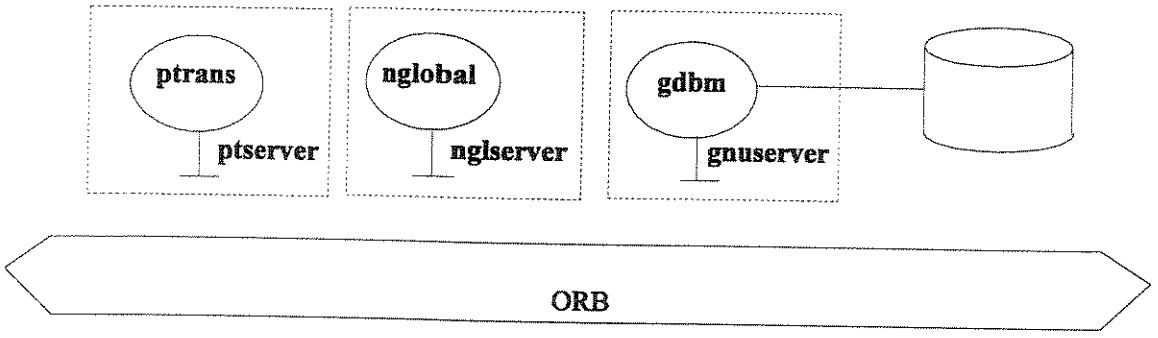

Figura 4-14 Objetos e servidores das funções de núcleo

O servidor gnuserver, que é único, utiliza uma biblioteca de funções da base de dados Gdbm, e armazena uma chave de busca, única, e o dado associado a ela. Uma característica importante aqui é a possibilidade de se ter o mesmo arquivo aberto por múltiplas aplicações. Também é permitida a abertura de vários arquivos, sendo que o ponteiro para cada um deles está associado a um número global na rede, que serve como sua chave de busca numa tabela *hash*. A localização do gnuserver é determinada pelo arquivo Orbix.hosts. Este arquivo contém os nomes das máquinas (*hosts*) onde o servidor está disponível.
As principais funções do Gdbm estão descritas na interface:

```
interface gdbm {

        unsigned long gdbmx_open(in string name, in long block_size, in long flags, in
long mode);
        void gdbmx_close(in unsigned long dbf);
        long gdbmx_store(in unsigned long dbf, in datam key, in datam content, in
long flag);
        datam gdbmx_fetch(in unsigned long dbf, in datam key);
        datam gdbmx_delete(in unsigned long dbf, in datam key);
        datam gdbmx_firstkey(in unsigned long dbf);
        long gdbmx_nextkey(in unsigned long dbf, in datam key);

};
```

onde:

- gdbmx_open: inicializa o sistema gdbm. O parâmetro name representa o nome da base de dados a ser aberta; block_size é usado durante a inicialização para determinar o tamanho do bloco de dados a ser transferido do disco para memória. A variável flags indica se o usuário quer simplesmente ler a base de dados (GDMB_READER); se o usuário quer acesso exclusivo de leitura e escrita (GDBM_WRITER); ou se o usuário deseja acesso de leitura e escrita (GDBM_WRCREAT) e, caso a base de dados não exista, uma nova deve ser criada. A variável mode determina o modo de permissões da base de dados. O valor retornado funciona como uma chave de busca numa tabela *hash*, onde está armazenado o ponteiro da base de dados criada.

- gdbmx_close: fecha a base de dados. O parâmetro dbf corresponde ao valor retornado na função gdbmx_open.

- gdbmx_store: armazena a informação key/content, onde key funciona como uma chave de busca, que deve ser única, e content corresponde ao conteúdo do dado armazenado. O parâmetro dbf identifica o arquivo onde a informação será guardada. Flag indica a ação a ser tomada quando a chave já está na base. Se flag está com o valor de GDMB_REPLACE, significa que dado mais antigo deve ser substituído pelo mais novo. Se flag tem o valor de GDBM_INSERT, ocorrerá um erro e nenhuma ação será tomada se a chave já existe. O valor retornado indica sucesso (0) ou não da operação(-1 ou 1).

O tipo datam possui a seguinte estrutura:

```
struct datam {

        char *dptr;   // ponteiro para os bytes a serem armazenados
        long dsize;   // tamanho da sequência de bytes armazenada

};
```

- gdbmx_fetch: a partir de uma chave de busca (key), retorna a informação associada (content), armazenada na base de dados (dbf).

- gdbmx_delete: remove a informação associada à chave key. O valor retornado indica sucesso na operação (0) ou se o dado não estava presente na base de dados (-1).

As duas próximas funções permitem o acesso de todos os dados na base de dados.

- gdbmx_firstkey: retorna a primeira chave armazenada na base de dados.

- gdbmx_nextkey: a partir de uma dada chave (key), ele retorna a próxima chave armazenada.

O servidor ptserver é responsável pelo processamento de transações. Ele é baseado nos protocolos *two-phase locking protocol* e *two-phase commit protocol* [Moss85], e tem a finalidade de garantir a consistência dos dados num ambiente distribuído. Um exemplo é que todo acesso a uma base de dados distribuída deve ser realizado dentro de uma transação, que possui um identificador único no sistema. Para cada nó na rede envolvido na transação é disparado um servidor ptserver.
Os serviços que implementam o protocolo de transações são definidos na interface ptrans,

```
interface ptrans {

        unsigned long begin (in timevalix timeout, in tranid tid);
        long addLock(in tranid tid, in nclLockx nlock, in timevalix timeout);
        long getState(in unsigned long nroglobal);
        long abort(in tranid tid);
        long removeLock(in tranid tid, in nclLockx nclock);
        long prepare(in tranid tid);
        long commit(in tranid tid);

};
```

onde:

- begin: é responsável por criar uma transação. O parâmetro *timeout* indica o prazo que a transação tem para terminar. A variável tid contém o identificador da transação e *host* onde ela foi criada. O valor retornado indica que a transação foi iniciada (NCL_STARTED) ou que algum problema ocorreu e a operação não teve sucesso (0).

Os tipos timevalix e tranid apresentam as seguintes estruturas:

```
struct timevalix {

        long t1;  // componente do timeout da transação
        long t 2; // componente do timeout da transação
        char *servername; // contém o nome do servidor onde foi criada a transação

};

struct tranid {

        long tid;  // número global que identifica a transação
        char *hostname; // contém o nome do servidor onde foi criada a transação

};
```

- long addLock: adiciona um *lock* na transação. O parâmetro tid identifica a transação que vai ter o *lock* adicionado na lista de *locks*, nlock contém a

O servidor ptserver é responsável pelo processamento de transações. Ele é baseado nos protocolos *two-phase locking protocol* e *two-phase commit protocol* [Moss85], e tem a finalidade de garantir a consistência dos dados num ambiente distribuído. Um exemplo é que todo acesso a uma base de dados distribuída deve ser realizado dentro de uma transação, que possui um identificador único no sistema. Para cada nó na rede envolvido na transação é disparado um servidor ptserver.
Os serviços que implementam o protocolo de transações são definidos na interface ptrans,

```
interface ptrans {

        unsigned long begin (in timevalix timeout, in tranid tid);
        long addLock(in tranid tid, in nclLockx nlock, in timevalix timeout);
        long getState(in unsigned long nroglobal);
        long abort(in tranid tid);
        long removeLock(in tranid tid, in nclLockx nclock);
        long prepare(in tranid tid);
        long commit(in tranid tid);

};
```

onde:

- begin: é responsável por criar uma transação. O parâmetro *timeout* indica o prazo que a transação tem para terminar. A variável tid contém o identificador da transação e *host* onde ela foi criada. O valor retornado indica que a transação foi iniciada (NCL_STARTED) ou que algum problema ocorreu e a operação não teve sucesso (0).

Os tipos timevalix e tranid apresentam as seguintes estruturas:

```
struct timevalix {

        long t1; // componente do timeout da transação
        long t 2; // componente do timeout da transação
        char *servername; // contém o nome do servidor onde foi criada a transação

};

struct tranid {

        long tid; // número global que identifica a transação
        char *hostname; // contém o nome do servidor onde foi criada a transação

};
```

- long addLock: adiciona um *lock* na transação. O parâmetro tid identifica a transação que vai ter o *lock* adicionado na lista de *locks*, nlock contém a

informação do objeto a receber o *lock* e o tipo de *lock*, e o *timeout* indica o tempo que a transação tem para terminar. O valor retornado indica sucesso (1) ou não da operação (0).

A estrutura nclLockx é do tipo:

```
nclLockx {

        long lockMode; // indica o modo do lock (READ ou WRITE)
        Objectx target;  // contém informações do objeto que vai receber o lock

};
```

onde:

```
Objectx {

        string name; // nome do objeto
        string hostname; // host onde se encontra o objeto
        long type; // tipo do objeto

};
```

getState: recupera o estado corrente da transação. O parâmetro de entrada equivale ao número global da transação e o valor retornado corresponde a uma das constantes definidas no arquivo nucstd.h, representando o estado da transação. Essas constantes podem ser:

- NCL_STARTED: indica que a transação tem iniciado;
- NCL_LOCK: indica que a transação adquiriu um *lock*;
- NCL_PREPARED: indica que a transação está preparada para receber um *commit*;
- NCL_COMMIT: indica que a transação sofreu um *commit*;
- NCL_ABORT: indica que a transação foi abortada.

abort: termina a transação identificada por tid. O valor retornado indica sucesso (1) ou não (0).

removeLock: remove o *lock* da lista de *locks* da transação tid. O valor retornado indica sucesso (1) ou não (0).

prepare: coloca uma transação tid no estado NCL_PREPARED para receber um *commit*. O valor retornado indica sucesso (1) ou não (0).

commit: realiza um *commit* na transação tid. O valor retornado indica sucesso (1) ou não (0).

O servidor nglserver gerencia um objeto cuja função é prover um número global na rede.

```
interface nglobal {

        long gera_nroglobal();

};
```

onde:

- gera_nroglobal: retorna um número global.

4.4.9 *Template*

O *template*, conforme definido pelo ODP, é uma estrutura que contém informações suficientes para instanciar um objeto, e para realizar operações sobre o mesmo. Na implementação, conforme citado anteriormente, o *template* é uma *string* que contém o identificador do objeto e o instante da realização do *checkpoint* sobre o objeto. Entretanto, essa estrutura pode ser modificada de forma a conter dados mais abrangentes sobre o objeto, que vão desde sua criação até o suporte de canais, o que não foi considerado neste trabalho.

4.4.10 Linguagem C++

A linguagem ANSI C++ [Stroustrup94] foi escolhida para a realização do projeto basicamente por ser:

- portátil;
- orientada a objetos.

Essas características, além de facilitarem a aplicação de conceitos como heterogeneidade e autonomia [Nicol93, Schmidt95], constituem requisitos importantes para a utilização do Orbix.

Capítulo 5

Exemplo de utilização

5.1 Introdução

A seguir será descrito um exemplo de uma aplicação utilizando os conceitos de gerenciamento e transparência propostos nesta implementação.

5.2 Descrição da aplicação

Seja uma aplicação onde estão disponíveis três recursos (figura 5-1):

- R1: servidor de transações;
- R2: servidor de base dados;
- R3: gerador de números globais.

Esses recursos representam os serviços oferecidos por um núcleo de um sistema ODP, e estão agrupados em um *cluster* para fins de gerenciamento. A agregação dos recursos (*clustering*) tem por objetivo facilitar operações de gerenciamento como por exemplo o *checkpoint*, que deve ser aplicado quando se deseja realizar uma retroação (*rollback*) no processador de transações. Uma outra operação que se pode aplicar ao *cluster* é a migração.
Para cada recurso está associado um objeto de gerenciamento que por sua vez está associado ao gerente do *cluster*.

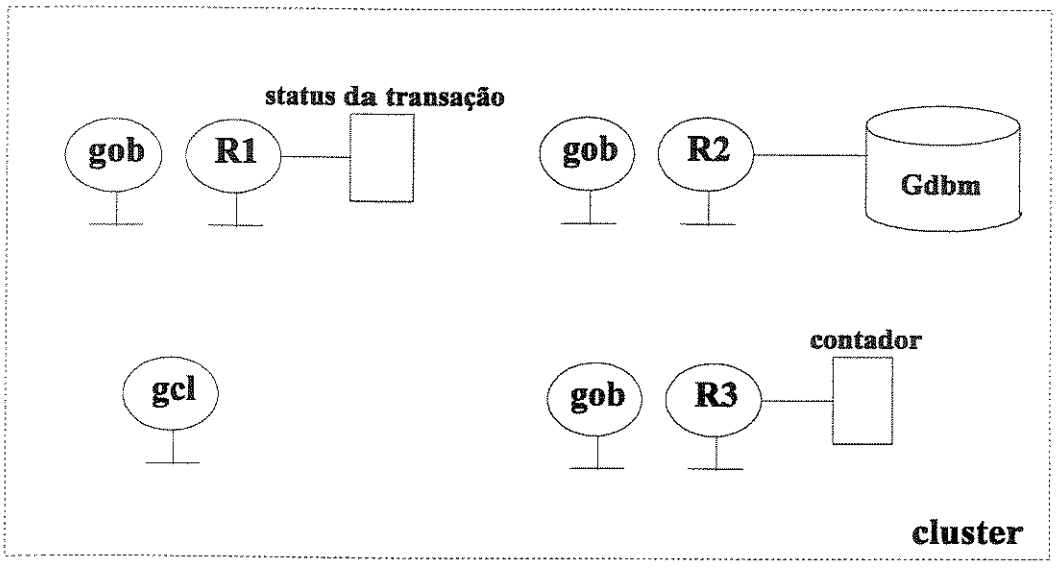

Figura 5-1 Exemplo de uma aplicação

Uma aplicação cliente que necessite utilizar o processador de transações (R1) inicialmente realiza uma invocação de um *begin* (figura 5-2). Essa invocação implica no acesso ao gerador de números globais (R3), que irá fornecer o identificador da transação. Toda chamada *begin* resulta também numa invocação ao gerente de *cluster* da função *checkpoint*. O gerente de *cluster* propaga então o *checkpoint* para todos os gerentes de objetos associados. Assim, caso a transação seja abortada num processo de aquisição de *lock*, seu estado inicial poderá ser recuperado e a aplicação cliente continuará a operar sobre ela.

A aplicação cliente pode desejar também fazer uma migração do *cluster* para um novo *host*. Entretanto, a migração do servidor de transações (R1) é uma operação considerada crítica, tendo em vista que esse pode conter informações relacionadas com transações de outros servidores (*hosts* diferentes), o que implicará num estado de inconsistência no momento da reativação, caso alguma transação não exista mais.

Com relação ainda a uma eventual operação de migração do *cluster*, vale ressaltar que não foi implementado um serviço de transparência de locação e que, nesse caso, fica sob responsabilidade do cliente a informação da nova localização dos objetos (ou do *cluster*) no sistema.

Todo o processamento interno do *cluster* é transparente para o cliente.

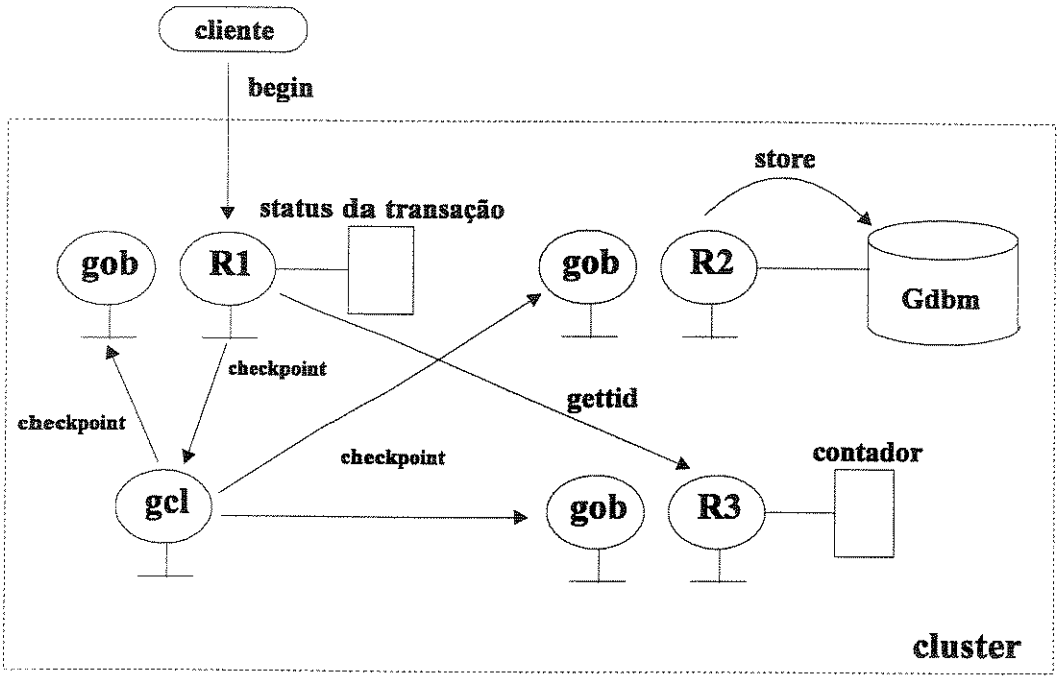

Figura 5-2 Chamada de um begin da transação

Numa operação de *checkpoint*, são armazenados as seguintes variáveis de estado na base de dados Gdbm (R2):

- no processador de transações;

- hostname: *host* do servidor;
- number: número global que identifica a transação;
- state: indica o estado da transação;
- timeout: contém o *timeout* da transação;
- lockingWD: proíbe aquisição de novos *locks*;
- lockers: aponta para a lista de *lockers* associada à transação;
- participants: aponta para a lista de *hosts* participantes da transação.

Todas essas variáveis caracterizam o *status* da transação corrente.

- no servidor de base de dados

- nfiles: número de arquivos abertos;

- filelist: aponta para uma lista de identificadores dos arquivos abertos.

- no servidor de números globais

- n: número global gerado.

Após adquirir um *lock* e realizar o acesso à base de dados, a transação libera o *lock* e vai preparar-se para o commit (figura 5-3). Novamente é necessário nova invocação de *checkpoint* para o gerente de *cluster*, a fim de que caso o processo de commit falhe, o *status* em andamento da transação (PREPARED) possa ser recuperado (figura 5-4).

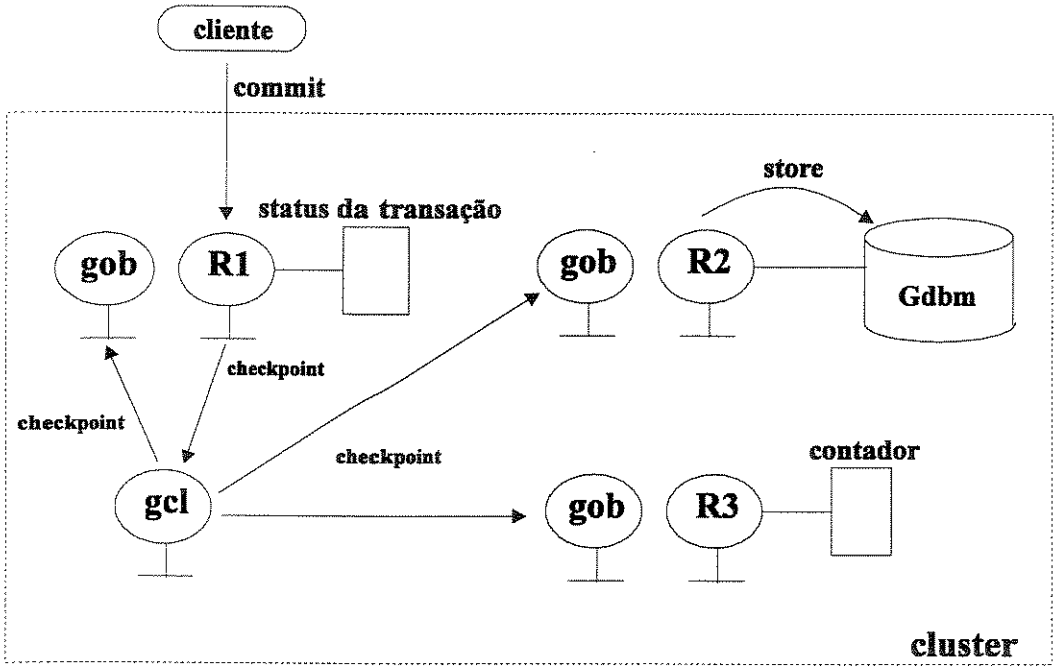

Figura 5-3 Transação preparando para receber um commit

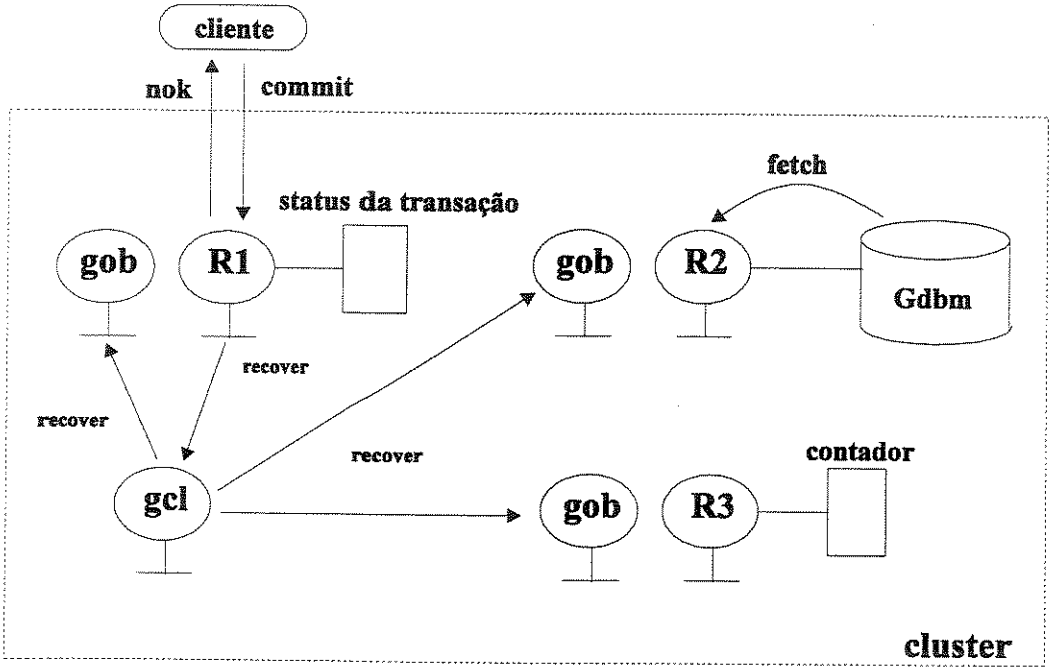

Figura 5-4 Recuperação do *cluster* numa falha do commit

5.3 Codificação da aplicação

A seguir serão apresentados alguns fragmentos de código ilustrando a aplicação e algumas modificações também a nível de código a serem seguidas pelo usuário na utilização dos objetos de gerenciamento propostos pela implementação.

5.3.1 Modificações introduzidas pelo usuário

Essas modificações correspondem às informações das interfaces definidas pelo usuário que devem ser passadas para a implementação.
Considerando que o usuário defina a interface de aplicação referente ao recurso R3:

```
interface R3 {

        long Delete();
        long checkpoint(in string templ);
        long recover(in string templ);
        long gera_nroglobal();

};
```

Essa informação será passada ao sistema da seguinte forma:

- será incluído no sistema o arquivo gerado pelo compilador IDL "R3.hh";

- na função addInterface da classe de implementação BCOBJECT_i será acrescentado:

```
if (strcmp(name,"R3") == 0) {

        R3 *r = R3::_bind(":R3server","");
        r->_release();
};
```

O parâmetro name é passado para a função addInterface e deve coincidir com o nome da interface.

- na função deleteInterface da classe BCOBJECT_i será inserido:

```
if (strcmp(name,"R3") == 0) {

        R3 *r = R3::_bind(":R3server","");
        r->Delete(r);
        r->_release();
};
```

- na função checkpoint da classe BCOBJECT_i será incluído:

```
if (strcmp(lista_de_interfaces->_buffer,"R3") == 0) {

        R3 *r = R3::_bind(":R3server","");
        r->checkpoint(intertempl);
        r->_release(0);
};
```

onde:

lista_de_interfaces->_buffer contém a lista dos identificadores das interfaces associadas ao gerente do objeto.

intertempl armazena o template utilizado nas operações de *checkpoint* e *recover* das interfaces de aplicação.

- na função recover será acrescentado:

```
if (strcmp(lista_de_interfaces->_buffer,"R3") == 0) {

        R3 *r = R3::_bind("R3server","");
        r->recover(intertempl);:
        r->_release();
};
```

Vale ressaltar que o usuário deve definir nas interfaces de aplicação as operações de Delete, checkpoint e recover, e a estrutura do template utilizada nessas duas últimas operações citadas deve ser a mesma usada pelo sistema. Além disso, após serem feitas todas as modificações, o sistema deve ser recompilado, adicionando-se a ele o stub IDL da interface (o que no exemplo equivaleria ao arquivo R3C.C).

Tais modificações também devem ser feitas com relação aos recursos R1 e R2.

5.3.2 Criação do *cluster* e instanciação dos objetos

Inicialmente serão instanciados os gerentes de nó e cápsula, a fim de que, posteriormente, possa ser criado o gerente do *cluster*.

```
NODE *gno;
CAPSULE *gcp;
CLUSTER *gcl;
BCOBJECT *gob1, *gob2, *gob3;

gno = NODE::_bind(":nodserver","");
gcp = gno->makeCapsule("gcp");
gcl = gcp->makeCluster("gcl", "gcp");
```

Após a criação do gerente de *cluster*, são criados os gerentes de objeto.

```
gob1 = gcl->makeObject("gob1", "gcl");
gob2 = gcl->makeObject("gob2","gcl");
gob3 = gcl->makeObject("gob3", "gcl");
```

Então são adicionadas aos gerentes de objetos as interfaces de aplicação (R1, R2, R3).

```
gob1->addInterface("R1");
gob2->addInterface("R2");
gob3->addInterface("R3");
```

Dessa forma os objetos de aplicação e de gerenciamento ficam dispostos de modo a formar uma agregação ou um *cluster* de objetos.

5.3.3 *Checkpoint* do *cluster*

Uma operação de *checkpoint* pode ser invocada para o gerente do *cluster* da seguinte forma:

```
if (gcl->checkpoint("gcl | t1"))
        cout << "checkpoint ok " << endl;
```

Capítulo 6

Conclusão

6.1 Introdução

Com o crescente desenvolvimento e o aumento da demanda por sistemas e aplicações distribuídos, a necessidade de se ter um padrão que possa caracterizar tais sistemas tornou-se cada vez mais evidente.
O modelo ODP surge então como uma proposta relevante que consegue abranger aspectos importantes necessários à construção de sistemas distribuídos abertos. Sistemas orientados a objetos, estruturados sob pontos de vistas diferentes, facilitando o gerenciamento e proporcionando transparência, são algumas sugestões do modelo.
O modelo ODP também pode ser aplicado às redes de informação [Stefani95] e serve como referência a especificação de arquiteturas para integração de objetos distribuídos, entre elas o TINA-C [TINAC95].
A arquitetura CORBA apresentou-se viável para construção de aplicações distribuídas, proporcionando transparência na utilização de seus recursos e facilitando a portabilidade e a reutilização de código, tendo em vista o suporte ao conceito de orientação a objetos.
A principal ferramenta utilizada no trabalho foi o Orbix, uma plataforma comercial compatível com a especificação CORBA, que procura atender os requisitos estabelecidos pela OMG.
O Orbix mostrou ser de grande valia para o trabalho, não só pelo modo transparente que trata a comunicação cliente/servidor, mas também pela facilidade que descreve as interfaces dos objetos, o que facilitou a implementação das funções de gerenciamento do ODP. Entretanto, algumas limitações presentes na versão disponível, representaram restrições para o trabalho. Por exemplo, a ausência de um ambiente multi-*thread* e interfaces do tipo *stream* [RM-ODPa] impediu que as funções de comunicação fossem incorporadas às outras já implementadas. Um outro problema foi a dificuldade de se acessar interfaces definidas em *runtime*, conforme o contexto apresentado. A falta também de um serviço de persistência de objetos determinou que se utilizasse uma base de dados externa para auxiliar no armazenamento persistente das informações contidas nos objetos.
Contudo, no âmbito geral, o trabalho conseguiu atender os seus objetivos, apresentando uma compreensão satisfatória dos conceitos propostos pelo modelo

ODP, e mapeando os mesmos para uma visão prática no que se refere ao aspecto computacional.

6.2 Implementação do trabalho

A implementação do trabalho implicou num bom conhecimento da plataforma Orbix e dos aspectos relacionados ao seu funcionamento, bem como um domínio dos conceitos de orientação a objetos e transações.
Do ponto de vista tecnológico, foi possível observar o quão difícil é mapear os conceitos da linguagem computacional para a linguagem de engenharia, dadas as limitações dos recursos disponíveis ou até mesmo a falta dos mesmos. Como exemplos das dificuldades encontradas estão a restrição da especificação CORBA em estabelecer somente uma interface para cada objeto e também a falta de um serviço de persistência de objetos, que resultaram na utilização de uma base de dados externa e na definição de novas estruturas de dados, a fim de que fosse garantida a abstração de alguns conceitos importantes do modelo ODP.

6.3 Trabalhos futuros

Numa etapa futura, as funções do núcleo do tipo armazenamento de dados e processamento de transações já estarão incorporadas numa base de dados orientada a objetos. Já está prevista para a próxima versão do Orbix, uma integração nesse sentido entre a plataforma e uma base de dados comercial, o ObjectStore [ObjectStore95], além do suporte a um ambiente multi-*thread* e de um compilador IDL para Java [Java96], de forma que se poderá utilizar objetos Java.
Uma outra proposta a ser desenvolvida refere-se à realização dos serviços de comunicação do ODP (canais). Atualmente no Orbix não é permitido uma transmissão isócrona de dados, fato que impede um tráfego do tipo *stream*, já que parâmetros relativos à qualidade de serviço não podem ser garantidos. Para contornar este problema será utilizado um outro protocolo de transporte para a transmissão efetiva dos dados, enquanto que a requisição de serviços e todas as outras funções de gerenciamento continuam a ser feitas através do núcleo ORB do Orbix.

Referências Bibliográficas

[Cockshot84] - Cockshot, W. P., Atkinson, M. O. et. al., Persistent Object Management System, *Software - Practice and Experience*, Volume 14, p. 49-71, 1984.

[Conceição95] - Conceição, L., *"Plataforma Multiware: Serviços de Comunicação"*, DCA-FEEC-UNICAMP, Dissertação de Mestrado, Setembro 1995.

[CORBA95] - *"The Common Object Request Broker: Architecture and Specification"*, Revision 2.0, July 1995.

[Domville92] - Domville, I., "The Distributed Application Environment - an Architecture Based on Enterprise Requirements", *Elsevier Science Publishers B. V. (North-Holland)*, 1992, IFIP.

[Garcia94] - Garcia, C. M., *"Uma Proposta para Modelagem de Funções de Gerenciamento para Processamento Distribuído Aberto"*, IC-UNICAMP, Dissertação de Mestrado, Dezembro 1994.

[Gdbm94] - *"GNU dbm, A Database Manager"*, Edition 1.4.1, Version 1.7.3, Free Software Foundation, 1994.

[Griethuysen92] - Griethuysen, J. J., "Enterprise Modelling - A Necessary Basis for Modern Information Systems", *Elsevier Science Publishers B. V. (North-Holland)*, 1992, IFIP.

[Gunji95] - Gunji, T., *"Plataforma Multiware: Serviços de Objetos"*, DCA-FEEC-UNICAMP, Dissertação de Mestrado, Agosto 1995.

[Java96] - Newman, A., *"Using Java"*, Que Corporation, 1996.

[Konstantas93] - Konstantas, D., "Object Oriented Interoperability", *Proceedings of the Seventh European Conference on Object Oriented Programming (ECOOP 93)*, July 1993, p. 1-23, Kaiserlautern, Germany.

[Maezi95] - Maezi, M., "*Plataforma Multiware: Interface de Programação*", DCA-FEEC-UNICAMP, Dissertação de Mestrado, Agosto 1995.

[Moss85] - Moss, J. E. B., "*Nested Transactions - An Approach to Reliable Distributed Computing*", The MIT Press series in information systems, 1985.

[Nicol93] - Nicol, J., Wilkes, T. et. al., "Object Orientation in Heterogeneous Distributed Computing Systems", *IEEE Computer*, p. 57-67, June 1993.

[ObjectStore95] - "*ObjectStore C++, API User Guide*", release 4, June 1995.

[Orbix95a] - "*Orbix distributed object technology, Programmer's Guide*", IONA Technologies Ltd., Release 1.3.1, February 1995.

[Orbix95b] - "*Orbix distributed object technology, Advanced Programmer's Guide*", IONA Technologies Ltd., Release 1.3.1, February 1995.

[Orbix95c] - "*Orbix distributed object technology, IDL and C++ Mapping*", IONA Technologies Ltd., Release 1.3.1, February 1995.

[RM-ODPa] - ISO/IEC 10746-1 RM-ODP, "*ODP Reference Model Part 1, Overview*" , June 1995.

[RM-ODPb] - ISO/IEC 10746-2 RM-ODP, "*ODP Reference Model Part 2, Foundations*" , June 1995.

[RM-ODPc] - ISO/IEC 10746-3 RM-ODP, "*ODP Reference Model Part 3, Architecture*" , June 1995.

[RM-ODPd] - ISO/IEC 10746-4 RM-ODP, "*ODP Reference Model Part 4, Architectural Semantics*" , June 1995.

[Rumbaugh91] - Rumbaugh, J., et. al., "*Object-Oriented Modeling and Design*", Prentice Hall, 1991.

[Schmidt95] - Schmidt, D., Vinoski, S., "Object Interconnections", *C++ Report magazine*, January 1995.

[Stefani95] - Stefani, J. B., "Open Distributed Processing: An Architectural Basis for Information Networks", *Computer Communications*, Volume 18, Number 11, p. 849-862, November 1995.

[Stroustrup94] - Stroustrup, B., *"The C++ programming language"*, second edition, Addison Wesley, 1994.

[TINAC95] - *"Overall Concepts and Principles of TINA"*, version: 1.0, February 1995.

[Vinoski93] - Vinoski, S., "Distributed Computing with CORBA", *C++ Report magazine*, July 1993.

Apêndice A

Interfaces IDL utilizadas na implementação

- **Interface de gerenciamento de nó**

```
interface NO {

        CAPSULE makeCapsule(in string name);
        sequence<CAPSULE> getCapsules();

};
```

- **Interface de gerenciamento de cápsula**

```
interface CAPSULE {

        CLUSTER makeCluster(in string name, in capsulename);
        sequence<CLUSTER> getClusters();
        long reactivate(in string templ);
};
```

- **Interface de gerenciamento de *cluster***

```
interface CLUSTER {

        BCOBJECT makeObject(in string name, in string clustername);
        sequence<BCOBJECT> getObjects();
        long Delete();
        long checkpoint(in string templ);
        long recover(in string templ);
        long deactivate(in string templ);
        long migrate(in string templ, in string hostname);
```

};

- **Interface de gerenciamento de objeto**

```
interface BCOBJECT {

        long addInterface(in string name);
        sequence<string> getInterfaces();
        long deleteInterface(in string name);
        long Delete();
        long  checkpoint (in string templ);
        long  recover(in string templ);

};
```

- **Interface do servidor de base de dados**

```
interface gdbm {

        unsigned long gdbmx_open(in string name, in long block_size, in long flags, in
long mode);
        void gdbmx_close(in unsigned long dbf);
        long gdbmx_store(in unsigned long dbf, in datam key, in datam content, in
long flag);
        datam gdbmx_fetch(in unsigned long dbf, in datam key);
        datam gdbmx_delete(in unsigned long dbf, in datam key);
        datam gdbmx_firstkey(in unsigned long dbf);
        long gdbmx_nextkey(in unsigned long dbf, in datam key);

};
```

- **Interface do servidor de transações**

```
interface ptrans {

        unsigned long begin (in timevalix timeout, in tranid tid);
        long addLock(in tranid tid, in nclLockx nlock, in timevalix timeout);
        long getState(in unsigned long nroglobal);
        long abort(in tranid tid);
        long removeLock(in tranid tid, in nclLockx nclock);
        long prepare(in tranid tid);
        long commit(in tranid tid);

};
```