Universidade Estadual de Campinas

DCA Faculdade de Engenharia Elétrica
Departamento de Engenharia de Computação
e Automação Industrial

Plataforma Multiware: Suporte a Objetos em Tempo de Execução

Tese apresentada à Faculdade de Engenharia Elétrica da Universidade Estadual de Campinas, como parte dos requisitos exigidos para a obtenção do título de

Mestre em Engenharia Elétrica

por

Tetsu Gunji

Engenheiro de Computação - UNICAMP/SP

Prof. Dr. Eleri Cardozo

Orientador - FEE/UNICAMP/SP

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

G955p

Gunji, Tetsu
Plataforma multiware: serviço de suporte a objetos em tempo de execução. / Tetsu Gunji.--Campinas, SP: [s.n.], 1995.

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

1. Sistemas operacionais distribuídos. 2. UNIX (Sistema operacional de computador). 3. Programação orientada a objetos. I. Cardozo, Eleri. II. Universidade Estadual de Campinas. Faculdade de Engenharia Elétrica. III. Título.

Universidade Estadual de Campinas

DCA **Faculdade de Engenharia Elétrica**
Departamento de Engenharia de Computação e Automação Industrial

Tese : **Plataforma Multiware: Suporte a Objetos em Tempo de Execução**

Autor : **Tetsu Gunji**

Orientador : **Prof. Dr. Eleri Cardozo**

Aprovada em 25 de agosto de 1995 pela banca examinadora

Prof. Dr. Eleri Cardozo (Presidente)

Prof. Dr. Felipe Afonso de Almeida IEC/ITA

Prof. Dr. Maurício Magalhães Ferreira FEE/UNICAMP

Agradecimentos

- Ao prof. Eleri pela orientação tanto em aspectos teóricos quanto na forma de implementação do trabalho;
- Aos meus amigos de república Douglas, Chang, Neil, Cláudio, Marcone e Arturo pelo companheirismo;
- Ao Maezi pelo material bibliográfico;
- Ao Cassius Di Cianni pela entrevista à respeito do projeto A-HAND;
- Aos profs. Tarcísio, Gilmar e Bottura que me incentivaram a fazer o mestrado;
- A CAPES pelo financiamento da bolsa de mestrado.

Conteúdo

Lista de Figuras

Abreviações

- **A-HAND** Ambiente de desenvolvimento de software baseado em Hierarquias de Abstração em Níveis Diferenciados;
- **API** Application Programming Interface;
- **ATM** Asynchronous Transmission Mode;
- **B-ISDN** Broudband - Integrated Services Digital Network;
- **CD-ROM** Compact Disk - Read Only Memory;
- **CLOS** Common Lisp Object System;
- **CORBA** Common Object Request Broker Architecture;
- **CSCW** Computer Supported Cooperative Work;
- **DAI** Distributed Artificial Intelligence;
- **DCE** Distributed Computing Environment;
- **DQDB** Dual Queue on a Dual Bus;
- **FAPESP** Fundação de Amparo à Pesquisa do Estado de São Paulo;
- **FDDI** Fiber Distributed Data Interface;
- **GDBM** GNU Data Base Manager;
- **ISO** International Standard Organization;
- **ODP** Open Distributed Processing;
- **OMT** Object Modelling Technique;
- **ORB** Object Request Broker;
- **PC** Personal Computer;
- **RM-ODP** Reference Model for Open Distributed Processing;
- **RPC** Remote Procedure Call;
- **SQL** Structured Query Language;
- **UCP** Unidade Central de Processamento;
- **VLSI** Very Large Scale Integration;

- **WINDOWS-NT** Windows - New Technology
- **XDR** eXternal Data Representation.

Resumo

O rápido crescimento do processamento distribuído tem levado à necessidade de uma padronização do processamento distribuído aberto (ODP: *Open Distributed Processing*). O modelo de referência do ODP propõe uma arquitetura que dá suporte à distribuição, interconexão e portabilidade.

No modelo ODP a unidade básica de processamento é o objeto. Um objeto é composto de seus atributos (estado) e de suas interfaces (métodos). Os objetos podem ser agrupados formando um *cluster* e os *clusters* por sua vez também podem ser agrupados novamente formando uma cápsula. A agregação dos objetos formando um *cluster*, e dos *clusters* formando uma cápsula tem como propósito facilitar o gerenciamento dos objetos. Os objetos podem interagir entre si através de regras bem definidas no modelo.

A plataforma *multiware* é um modelo para o suporte ao ODP. Este modelo se presta a identificar os vários componentes e serviços de uma plataforma de suporte ao ODP.

A proposta deste trabalho é prover um dos serviços da plataforma *multiware* que é o suporte a objetos em tempo de execução, ou seja, a criação de objetos (com seus atributos e interfaces), *clusters*, cápsula, além dos mecanismos de comunicação entre objetos.

Capítulo 1

Introdução

1.1 Motivação

Até recentemente não havia a necessidade e nem tampouco a tecnologia para construir sistemas distribuídos [1]. Quando os computadores começaram a tornar-se comuns na década de 60, dado o seu custo, era muito importante que o limitado poder de processamento disponível não fosse desperdiçado. Sistemas de processamento em *batch* maximizavam a execução de tarefas, porém havia o tedioso custo de preparação das tarefas e o tempo de resposta era muito grande. Com o passar dos anos as taxas de custo/desempenho para sistemas foi tornando-se cada vez mais favorável. Sistemas de *timesharing* interativo foram tomando o lugar dos sistemas de processamento em *batch*. Aplicações mais complexas foram construídas, por exemplo, utilizando vários computadores *mainframe* para o processamento simultâneo de muitas transações de um banco de dados a uma grande velocidade. Tais sistemas foram bem sucedidos e os computadores passaram a ser utilizados em funções de negócios, pesquisa e governo.

Estes complexos sistemas foram se tornando interdependentes, e muitas vezes era necessário uní-los para que uma tarefa fosse executada. Porém a união de tais sistemas, muitas vezes era quase impossível, pois a maioria deles não havia sido projetada para interoperar com outros sistemas. Mesmo uma transferência de dados *off-line* com fita magnética era complicada, pois cada fabricante fornecia um formato de arquivo diferente para seus produtos. Nesta época, surgiu o microprocessador e houve uma sensível melhora na taxa custo/desempenho para a computação.

Atualmente, o poder do computador é barato e onipresente. Porém os avanços no *software* não tem acompanhado a evolução do *hardware*. Mesmo com o poder dos mais avançados sistemas, aplicações de grande porte são de implementação ainda custosa. Muitas vezes os sistemas atuais são até mais difíceis de serem construídos, porque a variedade de novos produtos e subsistemas que os compõem vem crescendo a cada dia. Este atraso na tecnologia do *software* ameaça limitar o poder computacional que este novo *hardware* pode proporcionar. Neste ambiente surgiram os dois principais estímulos para o interesse corrente em sistemas distribuídos: mudanças tecnológicas e necessidades dos usuários [52].

- mudanças tecnológicas: o crescimento na microeletrônica, VLSI, etc, tem mudado a razão preço/desempenho a favor de múltiplos processadores de desempenho mode-

[1] Sistemas distribuídos são sistemas cujos componentes executam em diferentes espaços de endereçamento e atuam de forma cooperativa para atingir o objetivo para o qual o ele foi construído

rado ao invés de um único processador de alto desempenho. Também, a interconexão e o custo de comunicação têm decrescido dramaticamente nos últimos anos. Barramento, redes de troca de pacotes, redes locais, etc, são agora confiáveis e disponíveis a baixo custo.

- necessidades do usuário: o crescimento geral no uso da computação tem levado a uma grande demanda por recursos mais sofisticados (mais rápidos, extensíveis, confiáveis etc). Ao mesmo tempo há uma pressão por soluções mais econômicas que reduzam operações desnecessárias e proporcionem poder computacional onde ele é necessário. Em muitos casos é desejado um gerenciamento descentralizado.

A principal motivação deste trabalho está na necessidade de se investigar técnicas que diminuam o custo de desenvolvimento de sistemas distribuídos.

1.2 Objetivos

Os sistemas de computação distribuída apresentam muitas vantagens sobre os modelos centralizados. Os benefícios que uma organização pode obter com sistemas distribuídos são [23]:

- disponibilidade, confiabilidade e tolerância a falhas;
- superar a separação entre usuários e recursos computacionais - por exemplo, PCs acessando recursos remotos, sistemas de informação corporativa distribuída e diagnóstico remoto e manutenção;
- compartilhamento, integração e particionamento de recursos entre diferentes sistemas, localizações e domínios de gerenciamento em resposta às necessidades do usuário (fig. 1.1);
- melhoria no desempenho como resultado da operação paralela de diferentes aplicações ou de partes de uma mesma aplicação;
- contenção de custo e risco devido à permissão de adição de novos módulos para um sistema, balanceamento de carga dentro de um sistema, e reconfiguração dinâmica para reduzir certos tipos de custos (por exemplo, custo de comunicação) [30, 16, 53];
- ambiente de gerenciamento descentralizado em que os donos dos recursos podem gerenciar seus recursos.

Para que estes benefícios sejam possíveis é necessário que a tecnologia de processamento distribuído dê suporte a:

- integração de sistemas e recursos;
- provisão de modularidade;
- provisão de sistemas de segurança;
- manutenção da integridade de dados e consistência;

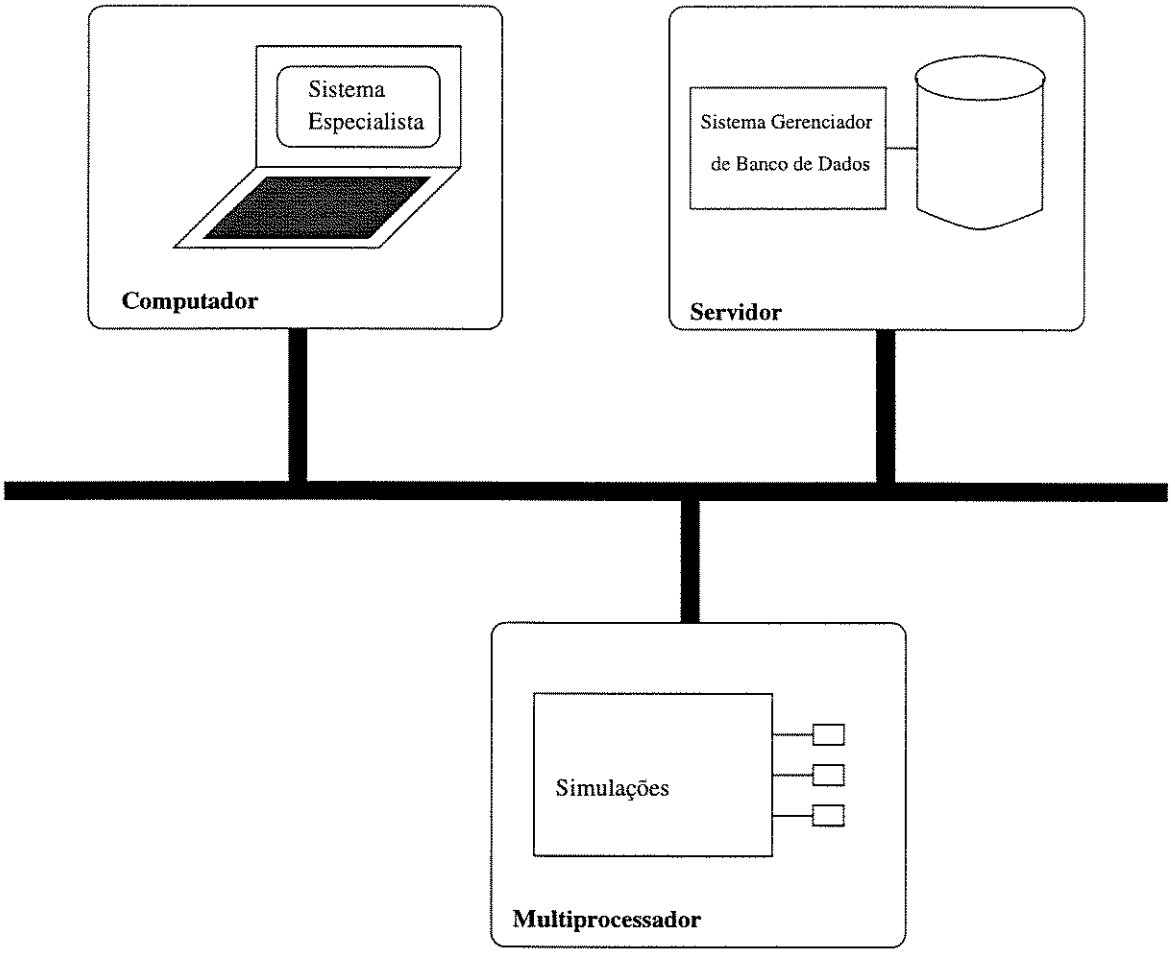


Figura 1.1: Integração de sistemas

- provisão de auditoria;
- provisão de ferramentas de engenharia de software para o suporte ao desenvolvimento de aplicações;
- provisão de mascaramento de detalhes técnicos de distribuição pelos programas de aplicação cobrindo:
  - transparência de acesso: a propriedade de esconder de um usuário os detalhes dos mecanismos de acesso para um dado objeto, incluindo detalhes da representação de dados e os mecanismos de invocação ;
  - transparência de localização: a propriedade de esconder de um usuário onde um objeto acessado está localizado;
  - transparência de falha: a propriedade de esconder de um usuário os efeitos dos erros nos componentes do sistema e nas comunicações. Para que este tipo de transparência possa ser implementado o sistema deve prover características tais como disponibilidade, confiabilidade, tolerância a falhas e manutenção da integridade e consistência dos dados;
  - transparência de concorrência (ou transação): a propriedade de esconder de um usuário a existência de concorrência no sistema. A preocupação aqui é mascarar

os detalhes de sincronização e os mecanismos de ordenação que garantem que o estado do sistema fique sempre consistente;

- transparência de migração: a propriedade de esconder de um usuário que o objeto sendo acessado teve sua localização modificada. Este conceito é relacionado com o conceito de transparência de localização;
- transparência de recursos: a propriedade de esconder de um usuário os mecanismos que gerenciam a alocação de recursos do sistema;
- transparência de federação: a propriedade de esconder de um usuário os limites do domínio administrativo e/ou tecnológico. A transparência de federação provê a integração de sistemas e recursos, cobrindo, por exemplo, integração de sistemas de arquiteturas diferentes, e de sistemas com recursos e desempenhos diferentes; ele provê também a modularidade;
- transparência de grupo: a propriedade de esconder de um usuário o uso de um grupo de objetos básicos de engenharia para dar suporte a uma única interface computacional. Isto permite a replicação de objetos pela provisão de disponibilidade, confiabilidade, tolerância a falhas, melhoria no desempenho e modularidade que permite crescimento incremental sem impactos sobre as aplicações já existentes.

Embora alguns sistemas distribuídos tenham resolvido um ou mais desses problemas, nenhuma solução de *software* conseguiu até agora unificar todos eles. Porém o desenvolvimento de técnicas práticas para resolver estes problemas tem permitido a construção de tais sistemas. A mais importante técnica é a abstração [50, 49, 4, 59, 23], que nos leva em direção a sistemas orientados/baseados em objetos.

1.3 Contribuições

Este trabalho mostra a importância de uma definição de um modelo de referência para o processamento distribuído aberto. Faz-se uma comparação de um sistema desenvolvido segundo o modelo de referência RM-ODP com outros sistemas distribuídos similares destacando os pontos positivos e negativos do ODP. Mostra-se resumidamente o que é o modelo de referência ODP, dando-se destaque ao modelo prescritivo do mesmo e a importância dos pontos de vista do ODP. As dificuldades e soluções de implementação de um sistema de suporte a objetos em tempo de execução aderente ao modelo ODP é a principal contribuição deste trabalho.

1.4 Apresentação

Este trabalho está dividido em oito capítulos.

O capítulo 2 apresenta uma comparação entre sistemas distribuídos baseado/orientados a objetos segundo os critérios descritos por Chin [10].

O capítulo 3 apresenta resumidamente o que é um sistema ODP, a organização dos documentos do modelo de referência ODP, as diferentes abstrações pelas quais o sistema pode ser visto e, por fim, o modelo prescritivo do modelo de referência ODP.

O capítulo 4 descreve a plataforma *multiware* que é um modelo para o suporte ao ODP, descrevendo mais detalhadamente a camada *middleware* da plataforma e apresentando nesta camada a funcionalidade do gerenciamento de objetos. Depois apresenta as restrições de implementação da camada *middleware* e por fim os componentes da mesma.

O capítulo 5 apresenta os serviços que dão suporte em tempo de execução pelo sistema desenvolvido e a descrição dos quatro principais módulos funcionais que são: escalonador, interface com a API, gerenciador de objetos e gerenciador de *binding*.

O capítulo 6 apresenta os aspectos mais relevantes a respeito da implementação do sistema de suporte a objetos em tempo de execução.

O capítulo 7 apresenta exemplos de utilização do sistema desenvolvido.

O capítulo 8 apresenta os pontos positivos e negativos do ODP de acordo com os critérios descritos por Chin [10], cita a importância dos pontos de vista do ODP, a dificuldade geral do trabalho e por fim sugestões para futuros trabalhos.

Capítulo 2

Sistemas Distribuídos Orientados/Baseados em Objetos

2.1 Introdução

Conforme citado no capítulo anterior a seguir será apresentado uma comparação entre sistemas distribuídos baseado/orientado a objetos.

A base destes sistemas é a abstração da programação através de um modelo de computação chamado modelo de objetos. O conceito fundamental deste modelo é o objeto, uma entidade que encapsula alguns dados ou informações num estado, e um conjunto de operações que manipulam estes dados. Em um sistema baseado/orientado a objetos, os dados e as operações são indivisíveis. O estado de um objeto não pode ser modificado, ou mesmo visto, exceto através de invocações das operações definidas no objeto.

A classe é o modelo através da qual os objetos são definidos e criados. Todo objeto é uma instância de uma classe. Classes tipicamente existem em tempo de compilação; objetos existem em tempo de execução.

Um sistema de programação distribuída orientado/baseado em objetos permite que os objetos sejam instanciados em um ambiente distribuído.

Estes tipos de sistemas devem ter as seguintes características :

- distribuição: o sistema executa numa rede de computadores independentes e heterogenêos;

- transparência: o sistema esconde o ambiente distribuído e outros detalhes desnecessários do usuário. Por exemplo, um sistema pode prover a característica da transparência de localização e com isto o usuário não precisa se preocupar com a localização física de um objeto para fazer uma invocação sobre ele;

- integridade dos dados: o sistema assegura que um objeto persistente está sempre em um estado consistente antes de uma invocação ser executada;

- tolerância a falhas: a falha de um computador ou um objeto representa apenas uma falha parcial do sistema; a perda é restrita ao computador ou ao objeto. O restante do sistema continua processando, talvez com a inconveniência de uma perda de desempenho;

- disponibilidade: o sistema deve assegurar que todos os objetos permanecem disponíveis com uma alta probabilidade, independente de falhas em computadores. Se um serviço tornar-se inacessível, o sistema pode se auto-inicializar novamente e restaurar o serviço;

- recuperabilidade: se um computador falhar, o sistema automaticamente recupera os objetos persistentes que residem sobre ele;

- autonomia dos objetos: o sistema permite que o dono de um objeto especifique os clientes autorizados a operar sobre o objeto;

- concorrência na programação: o sistema permite que os objetos de um programa possam ser atribuídos a múltiplos processadores para que eles possam ser executados concorrentemente;

- concorrência nos objetos: um objeto servidor pode servir múltiplas invocações de clientes concorrentemente;

- melhoria de desempenho: um programa bem projetado pode tipicamente executar mais rapidamente num sistema distribuído do que num sistema convencional.

2.2 Trabalhos Correlatos

Os sistemas distribuídos baseados/orientados a objetos devem prover um ambiente de computação descentralizado ou distribuído com as funcionalidades dadas pela figura 2.1.

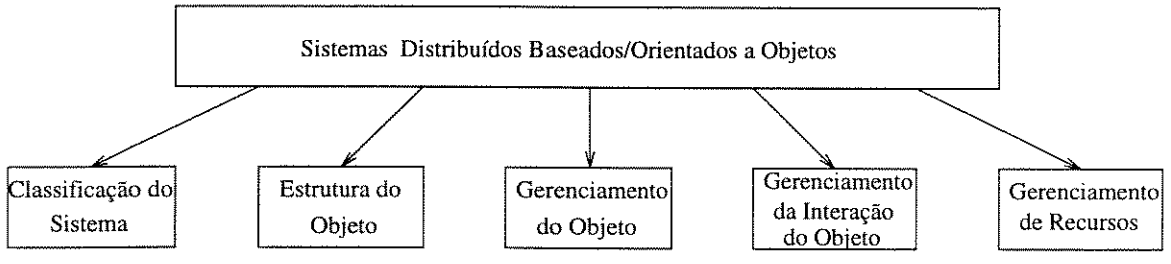


Figura 2.1: Classificação em categorias

A seguir comparamos o modelo de referência ODP (Open Distributed Processing) proposta pela ISO (International Standard Organization) [23, 24, 25, 22], o sistema CRONUS [3], o projeto do A-HAND (Ambiente de desenvolvimento de software baseado em Hierarquias de Abstração em Níveis Diferenciados) [15, 14]. Os tópicos citados anteriormente serão discutidos nas subseções subsequentes. Em cada subseção é dado uma breve explicação teórica acerca do assunto, um comentário a respeito de cada sistema em relação ao assunto abordado [1] e uma figura ilustrativa com a classificação de todos os sistemas de acordo com as definicões discutidas.

[1] somente para ODP, CRONUS e A-HAND, pois o restante se encontra em [10]

2.2.1 Classificação do Sistema

Um sistema de programação é dito baseado em objetos se ele dá suporte a objetos e orientado a objeto se ele dá suporte ao conceito de herança. A herança é um mecanismo que permite que novas classes possam ser desenvolvidas a partir de classes já existentes simplesmente especificando como as novas classes diferem das originais. Uma classe pode herdar as operações e o estado de uma classe base também denominada superclasse (herança simples) ou de várias classes base (herança múltipla).

Visão Geral da Classificação para alguns Sistemas

ODP

O modelo de referência ODP-ISO sugere um sistema orientado a objetos. As classes podem ser organizadas numa hierarquia de herança de acordo com o relacionamento classe derivada/classe base. Uma classe derivada pode ter mais de uma classe base (herança múltipla) e também pode proibir a supressão de propriedades da classe base (herança rigorosa).

CRONUS

O sistema CRONUS é orientado a objetos. O modelo de herança do CRONUS é rigorosamente hierárquico. No CRONUS, uma subclasse só pode ser derivada de uma única superclasse (herança simples).

A-HAND

O sistema do projeto A-HAND também é orientado a objeto, apresentando herança múltipla e uma característica adicional que é o polimorfismo (mesmo nome de operação com características diferentes em classes distintas) [50, 49, 4].

A figura 2.2 resume a classificação para alguns sistemas.

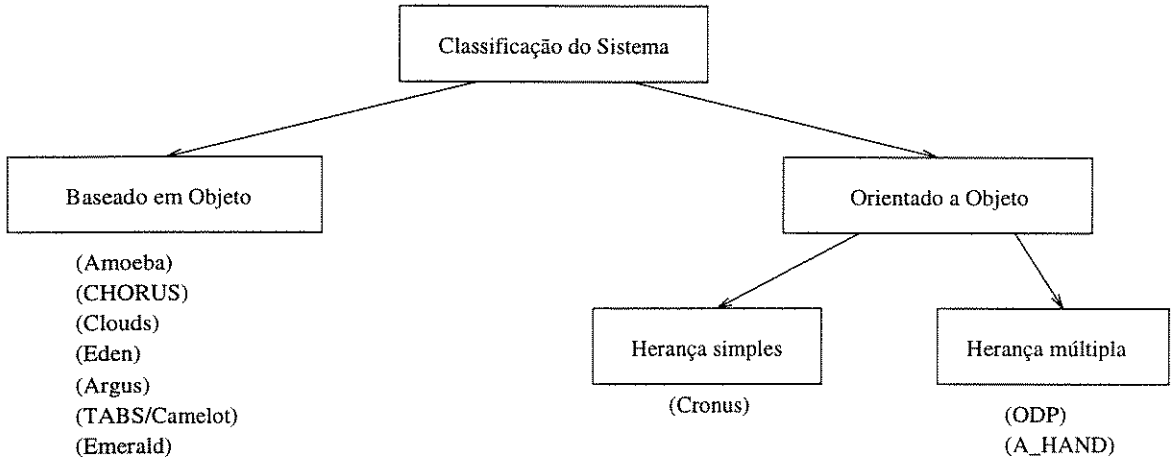


Figura 2.2: Classificação do sistema

2.2.2 Estrutura do Objeto

A estrutura de um objeto é caracterizada pela sua granularidade e pela sua composição. A estrutura dos objetos que é suportada pelo sistema influencia no desenvolvimento geral

de um projeto.

Granularidade

O tamanho relativo (memória requerida), o *overhead* [2] e a quantidade de processamento executada por um objeto caracterizam sua granularidade que podem ser classificados da seguinte forma:

- **objetos de grão grosso:** tipicamente possuem seus próprios espaços de endereçamento. Estes objetos são caracterizados por seu grande tamanho, relativamente ao grande número de instruções que eles executam para atender uma invocação e as poucas interações que eles têm com outros objetos. Alguns exemplos de objetos de grão grosso são os principais componentes de um programa, um arquivo, um banco de dados mono-usuário, um processo pesado [54, 60, 2], etc;
- **objetos de grão médio:** estes objetos podem residir no espaço de endereçamento de um objeto de grão grosso. Podem ser criados e mantidos sem muito custo porque eles são menores em tamanho e em escopo que os objetos de grão grosso. Exemplos de objetos de grão médio são estruturas de dados como listas ligadas, filas, os componentes de um banco de dados multiusuário, um processo leve(*thread*) [60, 58], etc;
- **objetos de grão fino:** estes objetos podem residir no espaço de endereçamento de um objeto de grão médio ou grosso. Estes objetos são caracterizados pelo pequeno número de instruções que eles executam e pelo grande número de interações que eles têm com outros objetos. Alguns exemplos de objetos de grão fino são tipos de dados tais como booleanos, inteiros e reais.

Quanto à granularidade os sistemas foram classificados em três tipos:

- sistemas de grão grosso;
- sistemas de grão grosso e médio;
- sistemas de grão grosso, médio e fino.

Composição

O relacionamento entre os processos e os objetos do sistema caracterizam a composição dos objetos que podem ser classificados da seguinte forma:

- **modelo de objetos passivos:** os processos e os objetos são entidades completamente separadas. Um processo não é limitado ou restrito a um único objeto, ao invés disso, um único processo é usado para executar todas as operações requeridas para satisfazer uma ação (fig. 2.3);
- **modelo de objetos ativos:** vários processos são criados e atribuídos para cada objeto para manipular a invocação dos seus pedidos. Cada processo é limitado e restrito a um objeto em particular para o qual é criado. Nesta abordagem, múltiplos processos podem ser envolvidos na execução de uma única ação (fig. 2.4). Este modelo ainda pode ser subclassificado em duas formas:

[2] quantidade de recursos que o sistema necessita para se autogerir

- **estático:** um número fixo de processos servidores é criado para cada objeto que é criado ou ativado. Neste esquema os pedidos podem ficar aguardando numa fila de espera se os processos servidores estiverem todos ocupados;
- **dinâmico:** processos servidores são criados dinamicamente para um objeto quando este os requisita. Os pedidos nunca vão para uma fila de espera.

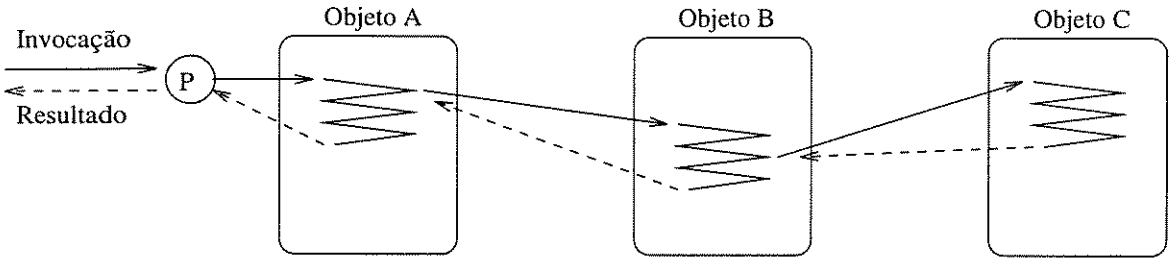


Figura 2.3: Executando uma ação num modelo de objetos passivos

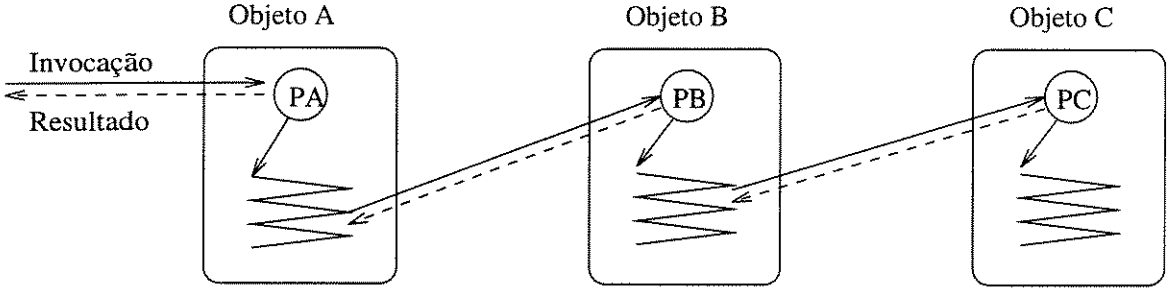


Figura 2.4: Executando uma ação num modelo de objetos ativos

Visão Geral da Classificação da Estrutura dos Objetos para alguns Sistemas

ODP

No modelo de referência ODP-ISO todo objeto tem um *behaviour* (que é tipicamente implementado como um processo leve) ativado durante a sua criação o que o caracteriza como um sistema de objetos ativos e também como um sistema de, no mínimo grão médio (existe um relativo *overhead* para a criação do mais simples objeto no ODP e a sua manutenção no sistema) que pode se tornar de grão grosso dependendo da complexidade do objeto. No ODP novos objetos são criados dinamicamente quando é requisitado tal pedido.

CRONUS

No sistema CRONUS os objetos são processos do sistema operacional nativo. Os parâmetros da chamada de um pedido são empacotados por um processo cliente e enviados para um servidor, onde os parâmetros são despachados para a rotina de processamento da operação.

A-HAND

No projeto A-HAND uma única *thread* (processo leve) pode conter vários objetos. Um objeto pode ser uma entidade tão simples que pode ter um único tipo básico como

atributo e nenhum método. Uma mensagem de requisição recebida numa porta de serviço de um servidor pode ser tratada de duas formas:

- o pedido é mapeado em uma exceção, cujo tratador é um *dispatcher* que cria um processo leve específico para atender essa mensagem; esse processo leve termina sua execução após completar o serviço pedido e enviar a resposta de volta ao cliente;

- o pedido é tratado diretamente pelo servidor, não é criado nenhum processo para tratá-lo. Se durante a execução de um serviço outros chegarem estes são enfileirados e só após o término total de um serviço é que outro pedido pode ser atendido.

A figura 2.5 resume a classificação da estrutura dos objetos para alguns sistemas.

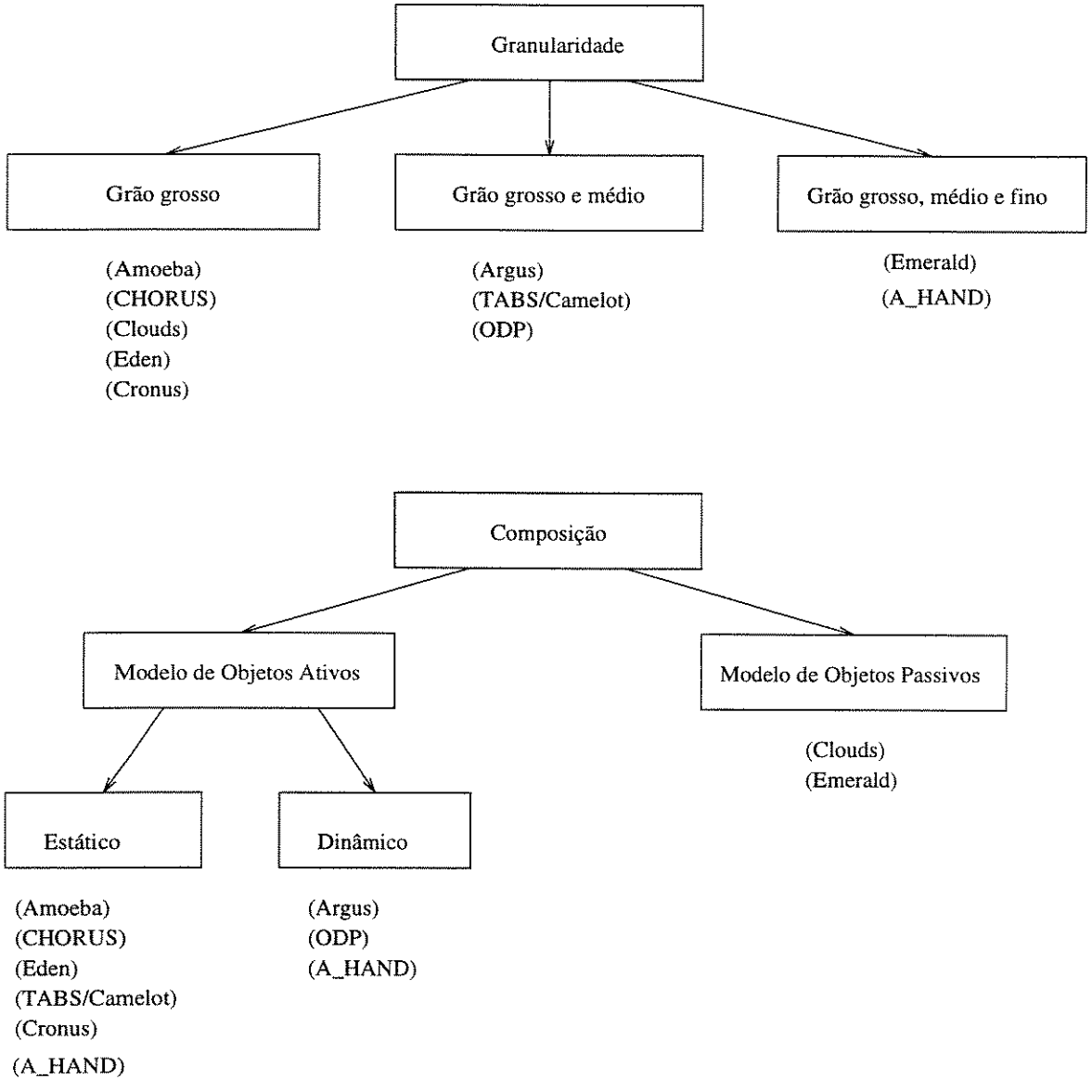


Figura 2.5: Classificação da estrutura dos objetos

2.2.3 Gerenciamento de Objetos

Os objetos são os recursos fundamentais de um sistema baseado/orientado a objetos, portanto o gerenciamento é uma função essencial desses tipos de sistema. Os mecanismos de gerenciamento de objetos envolvem o gerenciamento de ações, a sincronização, a segurança e a confiabilidade dos objetos.

Gerenciamento de Ações

Uma importante função do sistema é gerenciar as atividades das ações. As ações tem as seguintes propriedades:

- serialização: ações concorrentes são escalonadas de tal forma que são executadas sequencialmente em alguma ordem;
- atomicidade: uma ação é completada totalmente ou abortada;
- persistência: os efeitos de uma ação que é completada com sucessso nunca é perdido, exceto se ocorrer uma falha catastrófica.

Tipicamente, uma ação é executada com sucesso apenas quando todas as invocações associadas são completadas. Uma ação que é executada com sucesso é dita compromissada, e uma ação que falha é dita abortada. Se uma ação foi compromissada, todas as modificações ocorridas são registradas permanentemente numa memória não volátil. Se uma ação foi abortada, todas as modificações executadas são desfeitas.

O procedimento de compromissamento assegura que todas as modificações feitas nos objetos por uma ação são permanentes ou desfeitas. Existem vários procedimentos de compromissamento para ambientes distribuídos [41, 43, 46]. O compromissamento de uma ação pode ser de responsabilidade do sistema ou do usuário. Existem três esquemas:

- **esquema de pedido:** neste esquema, o usuário é responsável por decidir quando iniciar o procedimento de compromissamento;
- **esquema de transição:** quando todas as invocações associadas com uma ação são executadas com sucesso, o sistema executa o procedimento de compromissamento para fazer as modificações permanentes;
- **esquema de transação aninhada:** neste esquema, uma ação num nível mais alto cria e gerencia múltiplas sub-ações autonômas num nível mais baixo. A falha de uma sub-ação não necessariamante força a ação de nível mais alto a falhar também. A ação de nível mais alto pode manipular a falha da sub-ação da maneira que desejar (por exemplo, executando a sub-ação novamente).

Sincronização

É uma importante função que deve garantir que atividades de múltiplas ações que invocam o mesmo objeto não conflitam ou interfiram entre si. Existem muitos esquemas de sincronização, porém a maioria deles podem ser classificada de duas formas:

- **sincronização pessimista:** o sistema executa os passos adequados para prevenir a ocorrência de conflitos. Uma ação que invoca um objeto é temporariamente suspensa se irá interferir com outra ação que está correntemente sendo servida pelo objeto. *Locks* de leitura/escrita são os mecanismos mais comuns utilizados por esquema de sincronização pessimista, entretanto, *timestamps*, semáforos, e monitores são também utilizados;

- **sincronização otimista:** o sistema não toma o cuidado para prevenir conflitos enquanto as invocações estão sendo processadas. Ao invés disso, antes de uma ação ser compromissada, é testado e observado se não há conflitos com outras mudanças feitas por outra ação que já foi compromissada. Isto é, o sistema determina se o dado que foi modificado por uma ação é ainda atual ou se já foi alterado por uma ação que já foi compromissada. Se o dado ainda é atual a operação é compromissada. Mas se existe um dado mais recente a operação é abortada.

Segurança

A segurança é especialmente importante num sistema multiusuário, onde são atribuídos diferentes níveis de segurança aos usuários para operar sobre os diferentes conjuntos de objetos. Podemos dividir os esquemas de segurança em dois tipos:

- **esquema de capacidade:** é o mecanismo de segurança mais comum. A capacidade é uma chave consistindo de dois campos: o campo nome e campo com os direitos de acesso. O campo nome especifica um objeto em particular; o campo direitos de acesso indica as operações específicas de um objeto que podem ser invocadas. Cada capacidade tem exatamente um objeto, mas um objeto pode ter múltiplas capacidades. Isto possibilita o dono de um objeto variar os direitos de acesso para diferentes clientes. Este esquema é muito utilizado na segurança de arquivos [60, 51];

- **procedimentos de controle:** neste esquema, todo objeto tem um procedimento especial através da qual todos os pedidos de invocação devem passar primeiro. Estes procedimentos verificam as autorizações dos clientes que estão fazendo um pedido. O esquema de procedimento de controle é mais flexível pois pode suportar qualquer tipo de esquema de segurança. Se um objeto não é importante, um esquema mínimo de segurança pode ser utilizado; por outro lado, se o acesso a um objeto é restrito, esquemas mais sofisticados podem ser implementados.

Confiabilidade dos Objetos

Um sistema deve ser capaz de detectar e recuperar objetos. Há dois métodos gerais para prover confiabilidade dos objetos. Um método é recuperar uma falha num objeto o mais rápido possível, limitando a quantidade de tempo que este permanece indisponível. Outro método alternativo é replicar os objetos em vários computadores.

Recuperação de Objetos

Existem dois esquemas para recuperação de objetos:

- *roll-back*: o estado do objeto em falha é restaurado para o seu último estado consistente que foi registrado numa mémoria não volátil por um procedimento de compromissamento. Todos os processos e invocações que estavam em progresso quando a falha ocorreu são perdidos;

- *roll-forward*: o objeto que falhou e todos os demais objetos que estavam interagindo com ele são restaurados para seus últimos estados consistentes que foram registrados por um procedimento de compromissamento. Todos os processos e invocações são reinicializados e executados por completo. A recuperação *roll-forward* faz parecer que uma falha não ocorreu.

Replicação de Objetos

O esquema de replicação permite que cópias de um objeto existam sobre múltiplos computadores. A falha de um computador apenas resulta na indisponibilidade de réplicas que residem sobre esse computador; uma réplica que reside sobre outro computador pode ser capaz de continuar servindo o pedido de invocação. Há três esquemas de replicação de objetos:

- **objetos imutáveis:** este é o esquema mais simples que só permite que objetos imutáveis sejam replicados. Estes objetos só podem ser examinados pelos clientes, mas seus estados não podem ser modificados;

- **cópia primária:** neste esquema, uma réplica do objeto é designada como a cópia primária, as demais cópias são mantidas sobre outros computadores como cópias secundárias. Pedidos de leitura podem ser manipulados por qualquer réplica. Porém, pedidos de escrita só podem ser servidas pela cópia primária, que então propaga as modificações para cada cópia secundária;

- **cópias iguais:** neste esquema, todas as réplicas são consideradas iguais. Pedidos de leitura/escrita podem ser servidas por qualquer réplica. Entretanto, a cooperação de algumas ou todas as cópias são requeridas de forma a processar um pedido [27, 1, 34]

Visão Geral de Gerenciamento de Objetos para alguns Sistemas

ODP

O modelo de referência do ODP-ISO diz que o sistema deve dar suporte ao esquema de transações aninhadas. Existe uma entidade chamada gerenciador de transações que coordena as ações de compromissamento. Se alguma falha ocorrer, um protocolo de recuperação pode ser inicializado por este gerenciador.

O modelo apresenta um gerenciador de concorrência que assegura que invocações sobre um objeto compartilhado sejam isoladas e serializadas, ou seja somente uma ação de invocação do objeto é executada a cada vez, enquanto as outras invocações permanecem bloqueadas.

Quanto ao aspecto de segurança, um requerente de invocação faz um pedido de acesso a um objeto para o responsável por manter o objeto através de um *dispatcher* para um conjunto particular de operações; o *dispatcher* é o responsável por habilitar o conjunto de operações que foi pedido.

O *dispatcher* faz o pedido para um agente responsável pelo controle de acesso ao objeto (não é o mesmo que mantém objeto). A decisão de acesso permitido ou negado é retornado ao *dispatcher.*

O *dispatcher* passará a decisão para o requerente da invocação e o responsável por manter o objeto. O requerente pede acesso ao agente responsável por manter o objeto. E este permite ou não o acesso de acordo com as informações que este recebeu do *dispatcher*, conforme ilustrado na figura 2.6.

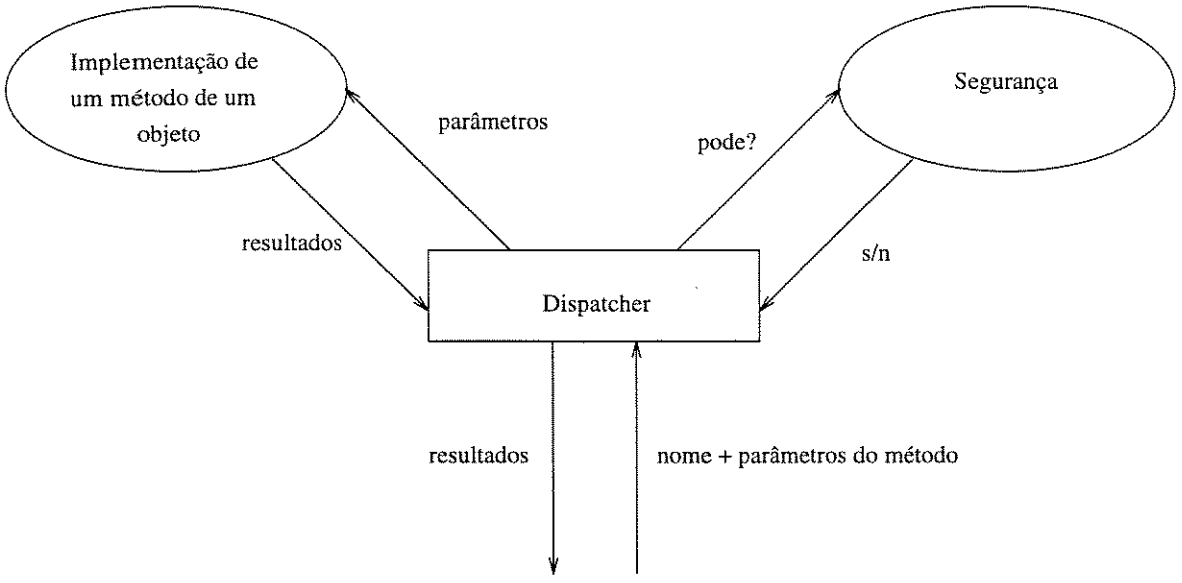


Figura 2.6: Segurança de objetos

A recuperação de objetos através de um *rool-back* é possível pois existe o mecanismo de *checkpoint*. O *checkpoint* pode ser combinado com uma invocação de operação atômica - um reinício de um serviço atômico poderia retornar a um estado anterior consistente qualquer transação em progresso antes do reinício.

Outra forma para conseguir transparência de falha é através da replicação de um objeto e formação de réplicas dentro de um grupo (grupo é um conjunto de uma ou mais interfaces que podem ser invocadas como se fossem uma única interface).

CRONUS

O sistema CRONUS provê seu próprio processamento de transações através de leituras e atualizações num padrão SQL. O usuário é o responsável por decidir quando iniciar o procedimento de compromissamento.

O sistema não gerencia diretamente os problemas de sincronização apesar de fornecer um mecanismo de semáforo pelo qual o usuário pode construir abstrações através do modelo de objetos, que escondam os detalhes de implementação no mecanismo de sincronização.

Cada objeto tem associado a si uma lista de controle de acesso especificando aquelas entidades que têm permissão para manipular o objeto. O controle de acesso fornece vários privilégios para os "principais" e os grupos do CRONUS, habilitando-os a fazer operações de invocação sobre os objetos. O "principal" é uma abstração que representa o usuário humano. Um grupo é um conjunto de "principais". A lista de controle de acesso consiste de uma lista de "principais" (e/ou grupos), e os direitos que eles possuem para o objeto.

O sistema de processamento de transações provê um mecanismo de *roll-back* num padrão SQL.

Os objetos no CRONUS são replicados utilizando uma variação de cópias iguais que é conhecido como votação com versões (assim chamada pois combina dois mecanismos conhecidos como vetor de versão e votação [27]).

A-HAND

O projeto do A-HAND não dá suporte a transações, portanto é o usuário que é res-

ponsável por executar o compromissamento de alguma ação.

O sistema não dá suporte direto à sincronização apesar de fornecer um mecanismo de entrada condicional em uma região crítica com a qual a sincronização pode ser implementada.

Os objetos estão protegidos indiretamente pelo sistema de arquivos do sistema (todo objeto está dentro de um arquivo). Além disso a porta de acesso dos objetos servidores podem ter acesso restrito por razões de segurança.

O sistema não apresenta suporte direto para a replicação de objetos.

A figura 2.7 resume o gerenciamento de objetos para alguns sistemas.

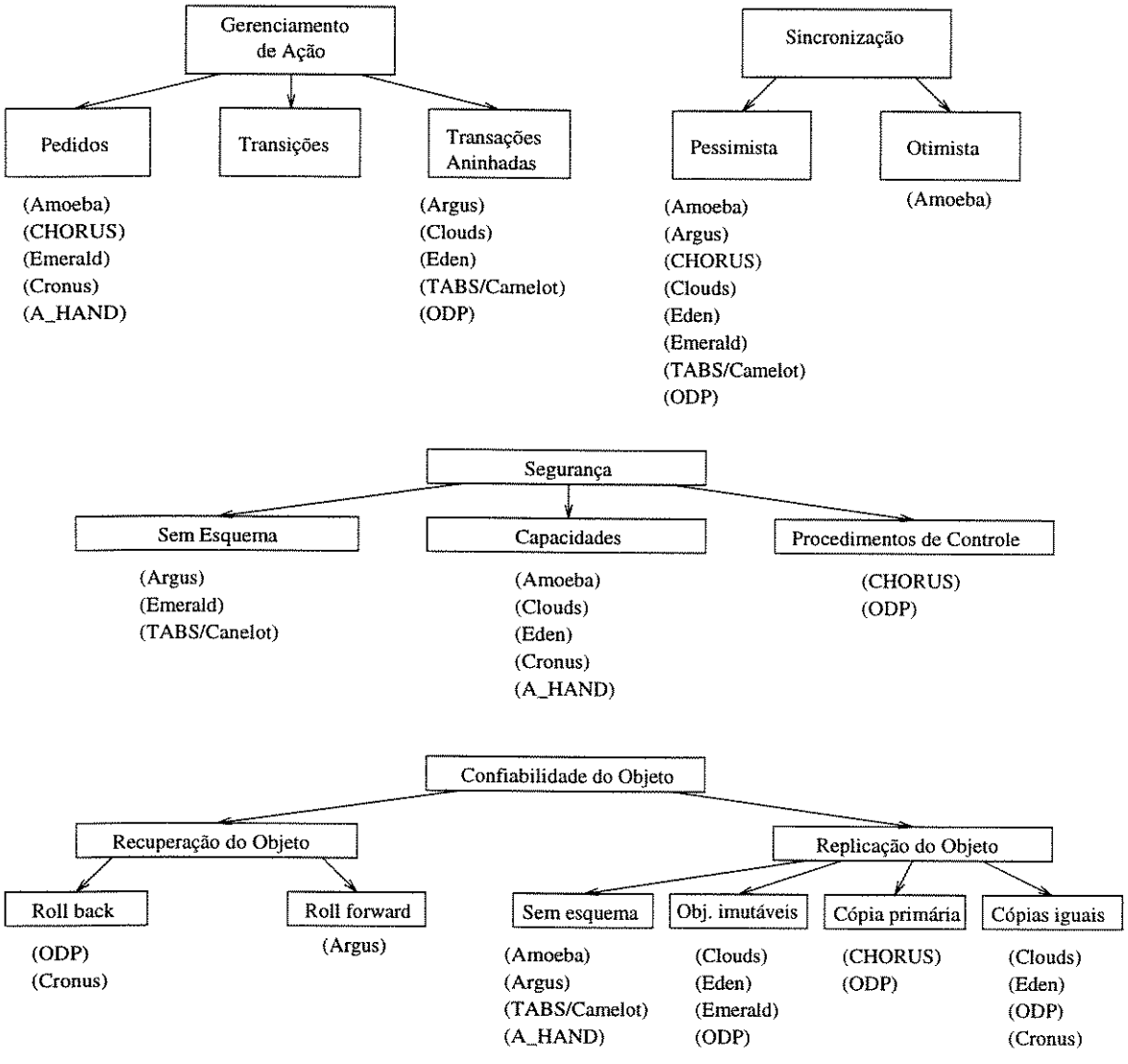


Figura 2.7: Gerenciamento de objetos

2.2.4 Gerenciamento da Interação de Objetos

Uma das funções de sistema de programação distribuída orientada/baseado em objetos é gerenciar a invocação entre objetos cooperantes. O sistema deve ser capaz de localizar

um determinado objeto, manipular as interações entre os objetos e detectar falhas de invocação.

Localização de um Objeto

O sistema deve prover a propriedade de transparência de localização. Os métodos mais comuns são:

- **esquema de codificação da localização no identicador no objeto:** quando uma invocação é feita, o sistema simplesmente examina o campo apropriado do identificador do objeto de forma que consegue determinar o computador sobre o qual o objeto reside;
- **esquema do servidor de nomes distribuído:** neste esquema, o sistema cria um grupo de objetos servidores de nome que podem ser mantidos sobre um determinado número de computadores, e não necessariamente todas;
- **esquema** *cache/broadcast*: uma *cache* é mantida sobre cada computador que registra a última localização conhecida de um determinado número de objetos remotos recentemente referenciados. Quando um cliente faz uma invocação remota, a *cache* é inicialmente examinada para determinar se há uma referência de localização para o objeto invocado. Se a localização é encontrada, o pedido de invocação é enviada para o local encontrado, mas caso a localização do objeto não esteja mais registrada na *cache* ou a informação dela encontra-se desatualizada, uma mensagem é difundida para a rede pedindo a localização do objeto.

Manipulação da Invocação a Nível de Sistema

Quando um cliente faz uma invocação sobre um objeto, o sistema é responsável por executar os passos necessários para entregar o pedido para o objeto servidor especificado e retornar o resultado para o cliente. Existem duas formas utilizadas:

- **troca de mensagens:** este esquema é tipicamente utilizado num modelo de objetos ativos. Quando um cliente faz uma invocação sobre um objeto, os parâmetros de invocação são empacotados dentro da mensagem do pedido. Esta mensagem é então enviada para o processo servidor. O processo servidor aceita a mensagem, desempacota os parâmetros, e executa a operação especificada. Quando a operação é completada, o resultado é empacotado dentro de uma mensagem de resposta, que é então enviada de volta para o cliente;
- **invocação direta:** este esquema é tipicamente utilizado num modelo de objetos passivos. Num modelo de objetos passivos, um único processo é responsável por todas as operações associadas com uma ação. Uma invocação sobre um objeto local é similar a uma chamada de procedimento, e uma invocação sobre um objeto remoto é similar a uma chamada de procedimento remoto.

Detectação de Falhas de Invocação

Uma falha pode ser classificada como existente ou transiente. Uma falha existente é definida como uma falha que ocorre antes de uma invocação ser iniciada. Uma falha

transiente, por outro lado, é uma falha que ocorre enquanto uma invocação está sendo executada. Os métodos utilizados para detectar estes tipos de falha são:

- *time-out*: o cliente/servidor fica esperando por um determinado limite de tempo. Se este limite estourar o cliente/servidor conclui que ocorreu uma falha;
- **objeto de investigação** (*Probe*)**:** é criado um objeto responsável por identificar falhas no cliente/servidor;
- **relatório de** *status*: o servidor mantém uma estrutura de dados para auxiliar a detecção e destruição de processos orfãos. Tais processos surgem por uma falha no cliente, uma transação abortada, uma falha num computador, ou uma partição da rede. Os processos orfãos desperdiçam recursos do sistema dado que eles podem manter *locks* diminuindo a concorrência do sistema. Consequentemente é desejável que eles sejam eliminados o mais breve possível.

Visão Geral do Gerenciamento da Interação de Objetos para alguns Sistemas

ODP

A localização de um objeto se dá através de uma entidade chamada *trader* [22], onde estão registrados os serviços que são fornecidos pelos servidores. Um *trader* pode interagir com outros *traders* de outras redes para localizar determinado serviço. No modelo ODP também existe o objeto relocador que é utilizado quando os objetos servidores mudam de localização.

Os objetos que estão num mesmo *cluster* (*cluster* é composto de vários objetos que são salvos, recuperados, migrados, desativados e destruídos em conjunto) se comunicam diretamente através de suas interfaces. Quando os objetos estão em *clusters* diferentes existe a necessidade da criação de um canal de comunicação.

Toda comunicação num canal (incluindo a invocação de um objeto) é monitorada por objetos denominados *binders*. Esta entidade pode iniciar, reativar, recuperar ou migrar um objeto em uma comunicação, inclusive ativando o objeto servidor. O *binder* detecta erros na configuração de canais, incluindo falhas no gerenciamento da rede.

A falha numa comunicação direta é muito difícil de ocorrer e não existe nenhum esquema de detecção previsto pelo modelo.

CRONUS

No Cronus existe um objeto chamado Localizador que é o responsável pela localização de objetos no sistema. Inicialmente este objeto consulta a *cache* e verifica se não há informações mantidas localmente sobre o serviço, caso não haja, o Localizador contacta os *clusters* (conjunto de *hosts* com uma única unidade administrativa) locais em busca do serviço. Todos os *clusters* que fornecem o serviço retornam uma indicação positiva para o Localizador. Se a informação ainda não tiver sido encontrada são contactados os *clusters* remotos.

A invocação de uma operação sobre um objeto é implementado no CRONUS como uma mensagem do processo cliente para um gerenciador de objetos e este para o objeto servidor que executa a operação, envia a resposta para o gerenciador que por sua vez envia de volta para o cliente.

Um simples esquema de *time-out* é usado pelos clientes para detectar falhas de invocação.

A-HAND

O sistema do projeto A-HAND utiliza um servidor de nomes responsável pela identificação e localização dos objetos.

O sistema utiliza um mecanismo de troca de mensagens utilizando portas de comunicação no esquema cliente-servidor. As portas podem ser de entrada, de saída, ou de entrada e saída simultaneamente, conforme a direção das mensagens a elas enviadas. As portas de dados simples transmitem sequências de caracteres sem uma estrutura associada; já as portas tipadas têm um tipo associado, e as operações de entrada e saída são efetuadas sobre valores desse tipo.

Um simples esquema de *time-out* é usado pelos clientes para detectar falhas de invocação. Quando os servidores são criados pode-se estipular um parâmetro que determina quanto tempo ele pode ficar inativo. Passado este tempo sem receber nenhum pedido o servidor se auto-destrói.

A figura 2.8 resume o gerenciamento da interação de objetos para alguns sistemas.

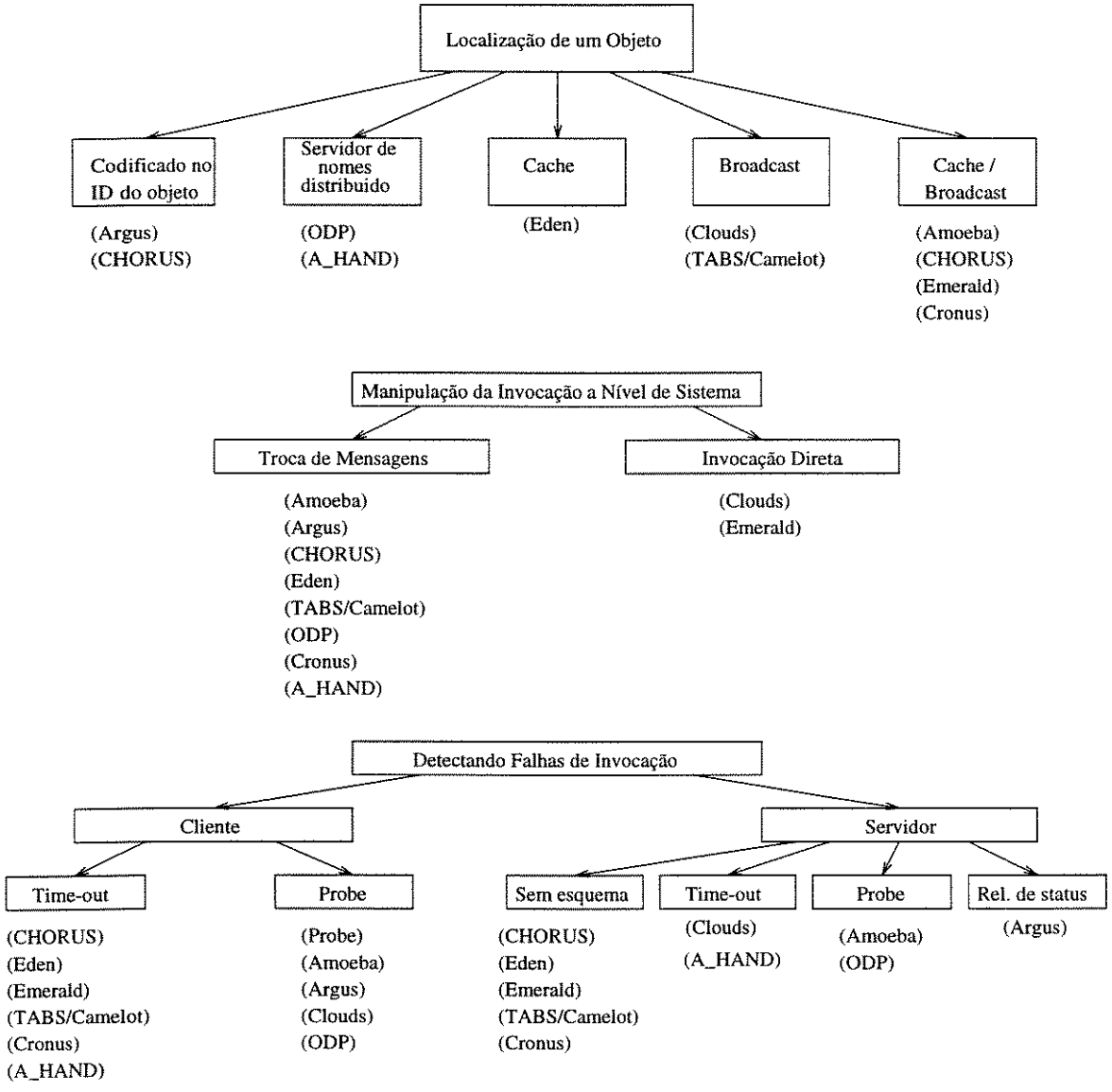


Figura 2.8: Gerenciamento da interação de objetos

2.2.5 Gerenciamento de Recursos

Um sistema de programação distribuída orientado/baseado em objetos deve prover mecanismos para gerenciar os recursos físicos, que incluem memória, disco, processadores e estações de trabalho da rede.

Armazenamento em Memória/Disco

Os objetos podem ser representados na memória através de duas formas:

- **versão única:** neste esquema apenas uma única versão de cada objeto é mantido em memória. Todas as ações que invocam um objeto modificam a mesma versão. Este esquema é utilizado tipicamente quando o sistema adota uma sincronização pessimista;
- **versão múltipla:** neste esquema múltiplas versões de cada objeto são criados e mantidos em memória. Toda ação que invoca um objeto é atribuído a sua própria versão do objeto sobre a qual a modificação é executada. Isto possibilta que múltiplas ações sejam invocadas sobre o mesmo objeto simultâneamente e assegura que eles não irão interferir entre si. Este esquema é utilizado tipicamente quando o sistema adota uma sincronização otimista.

O sistema deve registrar informações suficientes em disco para que um objeto persistente possa ser restaurado para um estado consistente. O sistema deve registrar o estado de um objeto toda vez que uma transação compromissada alterar o estado do objeto (esquema de *checkpoint*); ou pode simplesmente registrar as mudanças relativas para o objeto desde algum estado previamente registrado (esquema de *log*).

Existem duas formas de *checkpoint*:

- *checkpoint* **puro:** apenas uma última cópia do estado do objeto é mantido;
- **história de** *checkpoints*: os estados dos objetos são representados no disco como uma coleção ordenada de *checkpoints*.

Existem duas formas de *log*:

- *redo log*: neste esquema são registrados todas as modificações ocorridas desde um *checkpoint* base. Periodicamente um novo *checkpoint* base é registrado para cada objeto, e todas as entradas do antigo *log* são eliminadas. Só existe registro quando uma ação for compromissada;
- *commit log*: neste esquema todos os estados de todos os objetos modificados durante uma ação que foi ou está em processo de compromissamento é registrado. Se uma ação é abortada, uma entrada abortada é escrita no *log*.

Balanceamento de Carga

O principal objetivo do balanceamento de carga é maximizar a taxa de resposta do sistema. Isto pode ser obtido de duas formas: primeiro, os objetos podem ser atribuídos a processadores que estão mais ociosos para que os objetos possam trabalhar concorrentemente; segundo, objetos que interagem frequentemente poderiam ser atribuídos ao mesmo ou em processadores próximos para reduzir seus custos de comunicação. Existem duas formas possíveis de se obter o balanceamento de carga:

- **escalonamento de objetos:** um objeto é geralmente atribuído para o processador sobre o qual este é criado. Entretanto, o sistema pode permitir que ele seja criado sobre um processador remoto. Uma exceção são os objetos imóveis onde a localização dos objetos estão codificados dentro de seus identificadores. O esquema de escalonamento de objetos pode ser explícito ou implícito. No esquema explícito, o usuário é responsável por especificar o processador para o qual o objeto será atribuído. No esquema implícito, o sistema é responsável por determinar onde os objetos serão atribuídos;

- **migração de objetos:** um esquema mais sofisticado de escalonamento que permite que objetos movam ou migrem de um processador para outro a qualquer instante, em alguns casos mesmo enquanto estão processando um serviço.

Visão Geral Gerenciamento de Recursos para alguns Sistemas

ODP

O modelo ODP representa cada objeto em memória usando uma abordagem de versão única. Objetos são representados em disco utilizando o esquema de *checkpoint* puro.

O ODP apresenta a migração de *clusters*, que são formados por objetos.

CRONUS

O sistema CRONUS representa cada objeto em memória usando uma abordagem de versão única. Não existe um esquema definido para a representação em disco.

O CRONUS dá suporte a migração de objetos.

A-HAND

O projeto A-HAND representa cada objeto em memória usando uma abordagem de versão única. Não existe um esquema definido para a representação em disco. Futuramente com a inclusão da migração de objetos um esquema de *checkpoint* será implementado.

O A-HAND não dá suporte a um esquema de escalonamento de objetos. É utilizado o escalonador do sistema operacional nativo. É fornecido um mecanismo de criação remota de objetos.

A figura 2.9 resume o gerenciamento de recursos para alguns sistemas.

No capítulo seguinte veremos o modelo de referência ODP.

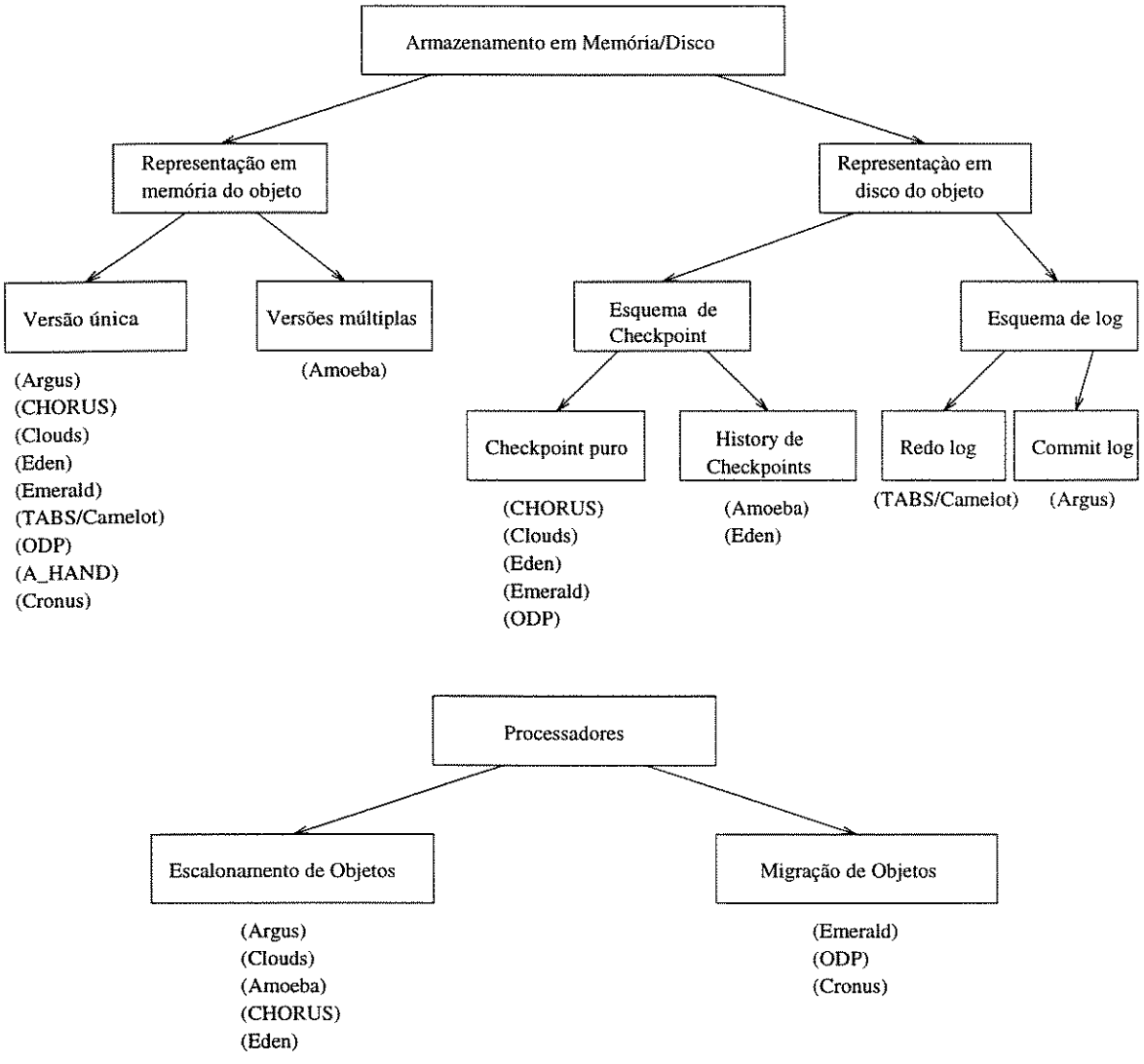


Figura 2.9: Gerenciamento de recursos

Capítulo 3

Processamento Distribuído Aberto - ODP

3.1 Introdução

Conforme citado no capítulo anterior a seguir será descrito o modelo de referência ODP.

3.2 Modelo de Referência ODP

O rápido crescimento do processamento distribuído tem levado a necessidade de uma padronização [20, 39, 44, 5]. O modelo de refêrencia ODP (Open Distributed Processing) cria uma arquitetura que dá suporte à distribuição, interconexão e portabilidade.

O Modelo Básico de Referência do Processamento Distribuído Aberto (RM-ODP), é baseado nos conceitos derivados das pesquisas e desenvolvimentos na aréa de processamento distribuído.

O RM-ODP consiste de:

- visão geral e guia de uso do modelo de referência [23]: contém a motivação do ODP dando o escopo, justificação e explicação dos conceitos chaves, e algumas linhas gerais da arquitetura ODP. Ele explica como o RM-ODP deve ser entendido e aplicado pelos usuários. Esta parte não é normalizada;

- modelo descritivo [24]: contém 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 requerimentos para novas técnicas de especificação. Esta parte é normalizada;

- modelo prescritivo [25]: contém as especificações das características que qualificam um sistema de processamento distribuído como aberto. Nele estão os requerimentos do modelo de referência ODP. Esta parte é normalizada;

- semântica arquitetônica [26]: contém a formalização do conceito de modelamento ODP definido no modelo descritivo. A formalização é atingida interpretando cada conceito em termos da construção de diferentes técnicas de descrição formal padronizadas [55]. Esta parte é normalizada.

3.3 Sistemas ODP

O resultado do modelo de referência ODP possibilita a implementação de sistemas ODP [12] que podem operar consistemente e confiavelmente enquanto está permitindo a distribuição de objetivos, recursos e atividades. Tais sistemas utilizam componentes que podem ser:

- heterogêneos;
- capazes de operação concorrente [36, 64];
- fisicamente e/ou logicamente distribuídos;
- portavéis;
- capazes de cooperar com outro sistema.

Neste contexto, heterogeneidade se refere a:

- equipamentos;
- sistemas operacionais;
- computação (linguagens, banco de dados, etc);
- autoridade (grupos de usuários com diferentes formas de acesso);
- aplicação (integração de planejamento e produção).

3.4 Organização do Modelo de Referência ODP

O modelo de referência ODP pode ser dividido em categorias. As categorias mais gerais contém as mais específicas. As categorias são:

- o modelo geral de referência (RM-ODP) que define os conceitos e a identificação de funções comuns;
- modelos de referência específicos que cobrem tipos especiais de organizações usando os conceitos e funções do RM-ODP, e definindo detalhes conceituais e funções específicas adicionais, por exemplo *Telecommunication Information Networking Architecture*;
- padrões para a realização de funções comuns e genéricas para um espectro amplo de aplicações, por exemplo *Trader* [22] ou uma linguagem de definição de interface;
- padrões para a realização de funções específicas de aplicações particulares, por exemplo interfaces de chamadas de conexões telefônicas.

O RM-ODP estabelece um estilo e uma base para o projeto e implementação de sistemas, permitindo o uso de componentes, metódos de projeto e representação que já são comuns. O RM-ODP deve:

- prover um modelo consistente, pemitindo que a consistência possa ser mantida entre múltiplos padrões desenvolvidos separadamente;
- definir as áreas onde o modelo de referência ODP é necessário, o relacionamento entre os padrões ODP e os outros padrões;
- descrever um conjunto básico de ferramentas a serem usadas na definição do modelo de referência ODP;
- prover um conjunto consistente de conceitos e terminologias comuns.

3.5 Pontos de Vista do ODP

Ao invés de lidar com toda a complexidade de um sistema distribuído, pode-se ver o sistema de pontos de vista diferentes, cada qual refletindo um conjunto de atributos diferentes. Cada ponto de vista representa um nível de abstração diferente do sistema distribuído original, sem a necessidade de criar um grande modelo que descreva todos estes níveis em detalhes [39].

Especificando um sistema utilizando diferentes pontos de vista permite-se que um sistema grande e complexo possa ser separado em partes gerenciavéis, cada um focalizando assuntos relevantes para membros diferentes da equipe de desenvolvimento [44].

Informalmente, um ponto de vista leva a uma representação do sistema com ênfase sobre um determinado assunto. Mais formalmente, o resultado de uma representação é uma abstração do sistema; isto é, a descrição que reconhece algumas distinções (aquelas relevantes para o nível em questão) e ignora outras (aquelas que não são relevantes para o nível em questão).

O modelo de referência ODP reconhece cinco pontos de vista:

- ponto de vista empresarial, que enfoca as políticas de negócios, políticas de gerenciamento e as regras do usuário humano em relação ao sistema e o ambiente na qual eles interagem: o uso da palavra empresa aqui não implica numa limitação para uma única organização; o modelo construído pode descrever muito bem a interação de um grande número de organizações distintas. Esta visão procura caracterizar os requisitos empresariais;
- ponto de vista de informação, que enfoca o modelamento da informação, provendo uma visão comum consistente cobrindo as fontes da informação, os destinos da informação e o fluxo de informações entre eles. Modelam-se as estruturas e o fluxo de informação que devem governar a sua manipulação;
- ponto de vista computacional, que enfoca os algoritmos e estruturas de dados e com a definição dos requisitos de transparência e distribuição;
- ponto de vista de engenharia, que enfoca os mecanismos de distribuição e a provisão de vários tipos de transparência necessários para dar suporte a distribuição;
- ponto de vista tecnológico, que enfoca os detalhes de componentes e *links* com o qual um sistema distribuído é construído. Deve mostrar qual o *software* e o *hardware* que devem ser utilizados para construir o sistema.

A figura 3.1 mostra como os pontos de vista do RM-ODP devem ser utilizados nas diversas fases do ciclo de vida de um sistema ODP. Por exemplo o ponto de vista empresarial leva à análise dos requisitos e políticas do sistema; os pontos de vista de informação e computacional fornecem documentos válidos para a fase de especificação funcional, por exemplo, especificações dos requisitos da informação (sua organização, aquisição, processamento, armazenamento e representação no sistema ODP) e os modelos de interação (acesso a todas as interfaces) e de construção (infraestrutura distribuída para programação de funções). O ponto de vista de engenharia permite a especificação das funções de processamento, armazenamento e comunicação necessárias à implementação do sistema: em particular deverão considerar-se aqui todos os requisitos de transparência de distribuição, de portabilidade e de interconectividade. Finalmente, através do ponto de vista tecnológico, deverão ser formulados requisitos de implementação, identificadas tecnologias a serem usadas, casos de testes, etc [63].

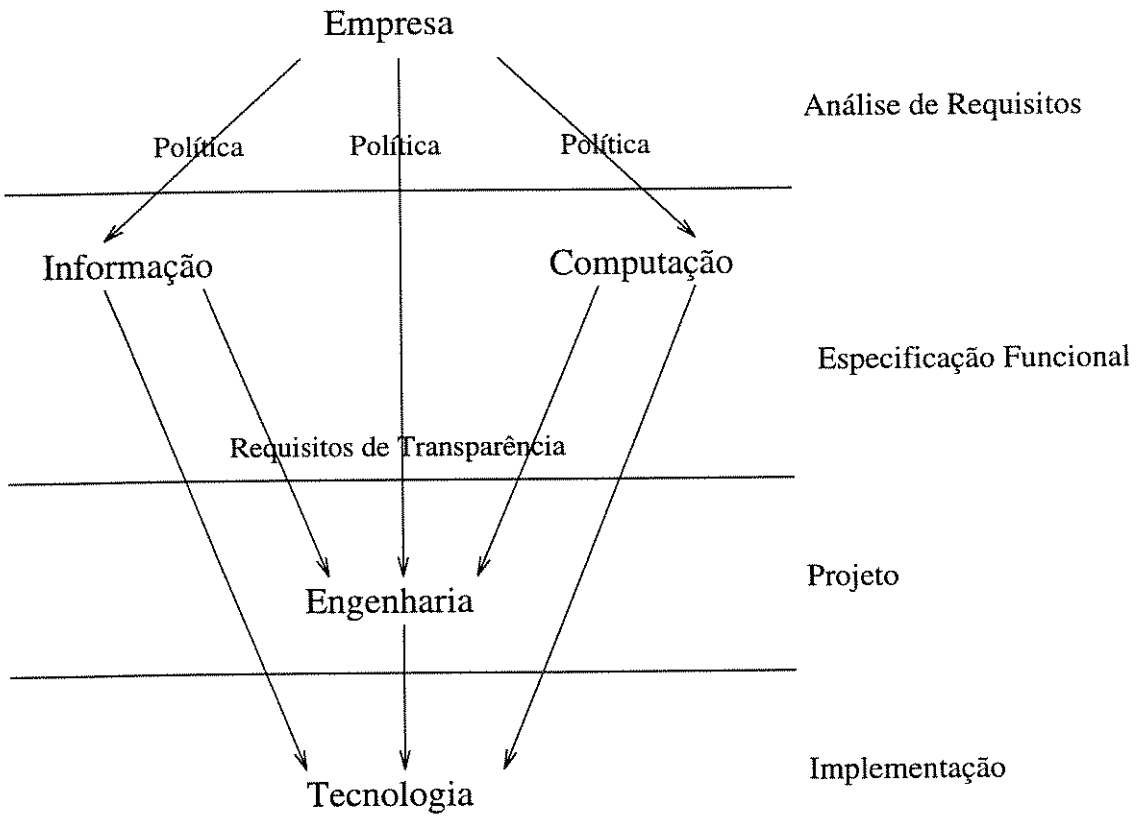


Figura 3.1: Uso de pontos de vista no ciclo de vida

3.6 O Modelo Prescritivo

O modelo prescritivo define linguagens para cada ponto de vista e então identifica as funções chave relacionadas com estas linguagens. As definições são expressas em termos presentes no modelo descritivo [24]. As informações acerca das linguagens empresarial, de informação, computacional e tecnológica foram resumidas enquanto que a linguagem de engenharia foi descrita de uma forma mais completa.

3.6.1 Linguagem Empresarial

A linguagem empresarial está preocupada com políticas, regras organizacionais e restrições do sistema. Ela deve suportar as decisões e objetivos organizacionais da qual muitas escolhas de projeto irão depender [18, 13]. Por exemplo, políticas de segurança estabelecidas pelo ponto de vista empresarial determinarão muitas das subsequentes escolhas dos mecanismos de segurança.

A linguagem empresarial é expressa em termos da identificação de um conjunto de atividades chaves e as regras envolvidas para executá-las. Estes conceitos são usados para forçar as ações disponíveis em uma dada situação e para expressar obrigações que devem ser cumpridas.

3.6.2 Linguagem de Informação

A linguagem de informação é expressa em termos dos objetos abstratos que representam a informação manipulada pela empresa. Nesta linguagem, a distribuição do processamento ainda não é visível.

As técnicas aplicadas são provalvemente familiares aos projetistas de sistemas de informação, mas um cuidado particular deve ser tomado para assegurar que a decisão tomada não prejudique na preparação para a distribuição.

3.6.3 Linguagem Computacional

O ponto de vista computacional lida com o particionamento lógico das funções, quebrando o problema em fluxos de invocação e provedores de serviço. Aqui as regras de interação entre clientes e servidores tornam-se importantes [31].

Existem dois principais aspectos relacionados com a linguagem computacional. O primeiro é o modelo de interação, que introduz os conceitos de invocação e anúncio para representar interações com ou sem resposta. Este modelo estabelece a interpretação da parametrização e mostra a semântica de uma falha para uma interação. Ele define os requerimentos sobre um tipo de interface do sistema e as regras de *matching* entre interfaces que devem ser safisteitas antes que uma ligação entre eles possa ser estabelecida [32].

A segunda principal parte da linguagem computacional é relacionada com a construção da configuração de objetos [42, 61], e o suporte para a criação de complexas redes de objetos interagentes, estipulando as regras que governam seu estabelecimento e sua desconexão.

Portanto a linguagem computacional define o meio pela qual os objetos podem interagir e estabelece os mecanismos pelos quais eles podem ser criados e depois destruídos, provendo os meios pelas quais configurações de sistemas complexos são construídos. Estas definições formam a base para a identificação das funções genéricas necessárias para dar suporte a especificação computacional, tal como a noção de um "fabricante" de objetos ou de um *trader*[22] para negociar e buscar serviços.

3.6.4 Linguagem de Engenharia

A linguagem de engenharia é um conjunto de conceitos, regras e estruturas para a especificação de um sistema ODP sobre o ponto de vista de engenharia.

Uma especificação de engenharia é utilizada para:

- descrever a organização de uma forma abstrata possibilitando a execução das funções de um sistema ODP;
- identificar as abstrações necessárias para gerenciar recursos físicos locais e distribuídos do sistema;
- identificar e definir os papéis dos diferentes objetos que dão suporte as funções ODP (por exemplo, funções de transparência);
- identificar os pontos em comum entre os diferentes objetos.

Conceitos Chave

A linguagem de engenharia contém os conceitos do ITU-T Rec.X.902 — ISO/IEC 10746-2 [24] e os que estão definidos abaixo.

- núcleo: um objeto que coordena processamento, armazenagem e funções de comunicação para uso por outros objetos de engenharia dentro do nó da qual ele pertence. Um sistema operacional (o *kernel* é um exemplo de um núcleo);
- cápsula: uma configuração de objetos formando uma unidade para o propósito de processamento e armazenagem. Um processo de um sistema operacional é um exemplo de uma cápsula;
- gerenciador da cápsula: um objeto que gerencia os objetos numa cápsula;
- *cluster*: uma configuração de objetos básicos de engenharia formando uma unidade para o propósito de desativação, *checkpoint* [11], recuperação, reativação e migração;
- gerenciador do *cluster*: um objeto que gerencia os objetos num *cluster*;
- *cluster checkpoint* : a criação de uma cópia do *cluster* quando o *cluster* se encontra num estado consistente;
- *checkpointing*: a operação de criar um *cluster checkpoint*;
- desativação: *checkpointing* seguido de uma eliminação de um *cluster*;
- clonagem: estabelecimento de uma cópia de um *cluster* pela instanciação de um *cluster checkpoint*;
- recuperação: clonagem de um *cluster* após a falha ou eliminação de um *cluster*;
- reativação: clonagem de um *cluster* após sua desativação (previsto);
- objeto básico de engenharia: um objeto de engenharia que requer o suporte de uma infraestrutura distribuída;
- *stub*: um objeto que provê funções de conversão (por exemplo, conversão/desconversão para uma forma canônica de representação de dados para dar suporte ao acesso transparente na interação entre objetos de engenharia);
- *binder*: um objeto que mantém a comunicação entre os objetos de engenharia;

- interceptador: um objeto que está presente na fronteira entre dois domínios que:
  - executa checagens ou monitora políticas para permitir a interação entre domínios diferentes;
  - executa transformações para mascarar as diferenças na interpretação de dados nos domínios.

  Um *gateway* é um exemplo de um interceptador;
- *protocol*: um objeto que comunica com outro objeto *protocol* para permitir a interação entre objetos de engenharia (possivelmente em outros *clusters*, cápsulas ou nós);
- canal: uma configuração de objetos *stub, binder, protocol* através da qual uma interação pode ocorrer;
- interface de comunicação: uma interface de um objeto *protocol* que está ligada a uma interface de um objeto interceptador ou de um outro objeto *protocol*
- identificador de interface de engenharia: um identificador para uma interface de objeto de engenharia [47] que está ligada a um canal;
- referência da interface de engenharia: um identificador para uma interface de objeto de engenharia que está disponível para o estabelecimento de uma comunicação [1];
- domínio administrativo: um conjunto de objetos cuja segurança, auditoria, monitoramento e outras funções de controle são controladas por uma administração comum;
- domínio da comunicação: um conjunto de objetos *protocol* capazes de se interagirem;
- nó: uma configuração de objetos formando uma unidade para o propósito de localização no espaço, e que incorpora um conjunto de funções de processamento, armazenagem e comunicação. Um computador e os *softwares* nele presentes correspondem a um nó;
- migração: mover um *cluster* para uma cápsula diferente.

Mapeamento de Linguagem de Computação em Linguagem de Engenharia

O objeto computacional pode ser suportado por funções de engenharia. Isto implica que pode-se fazer um mapeamento de um objeto computacional em funções de engenharia. Portanto para obter-se uma especificação de engenharia utiliza-se dos conceitos e regras de engenharia. O diagrama da figura 3.2 ilustra o mapeamento de um objeto computacional em uma configuração de objetos de engenharia, em termos de núcleo, cápsula e cluster.

O refinamento de *templates* computacionais [48, 31] em *templates* de engenharia (formando um *cluster*) corresponde a noção de compilar programas para produzir código objeto.

[1] Obs: Uma referência da interface de engenharia é apenas utilizada para o estabelecimento de uma comunicação, e é distinta do identificador de interface de engenharia que é utilizada para o propósito de interação.

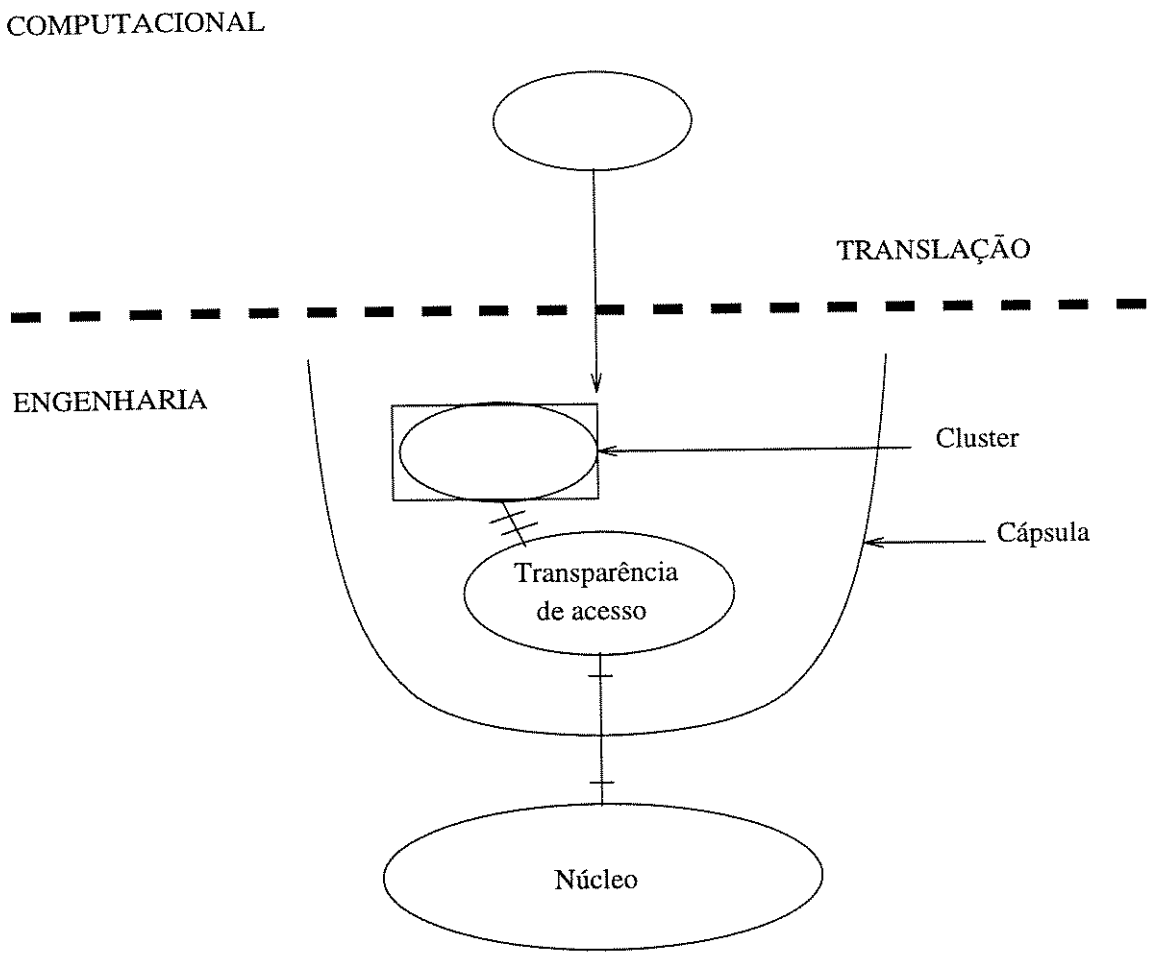


Figura 3.2: Translação da linguagem computacional para a de engenharia

O refinamento de *templates* de engenharia em *templates* de *clusters* (formando uma cápsula) corresponde a noção de unir módulos para formar um programa executável.

O conceito de cápsula corresponde a noção de espaço de endereçamento ou 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 todos num único *cluster* ou apenas a um objeto básico de engenharia;
- um objeto computacional com várias interfaces podem ser executadas numa mesma estação de trabalho ou em múltiplas estações de trabalho.

Transparência

Na passagem da linguagem computacional para a linguagem de engenharia, a preocupação da especificação de estruturas computacionais e declarações das propriedades necessárias para a interação entre objetos é agora relacionada aos mecanismos de engenharia capazes de assegurar estas propriedades. Isto leva a identificação de uma série de transparências (conforme definidas na página 4).

O mecanismo de transparência situa-se entre o usuário e o sistema e possibilita que o usuário peça alguma requisição ao sistema não se preocupando com o comportamento do sistema para atender seu pedido.

O conceito de transparência têm muitas aplicações além do modelamento de um sistema ODP. A transparência é o resultado normal do processo de abstração que é utilizado para especificar qualquer grande sistema, sem necessariamente implicar em ser distribuído. No ODP os requerimentos estão limitados a garantias de várias formas de transparência de distribuição.

As transparências definidas na linguagem de engenharia são:

- transparência de acesso;
- transparência de localização;
- transparência de migração;
- transparência de concorrência (ou transação);
- transparência de recursos;
- transparência de falha;
- transparência de grupo;
- transparência de federação.

A figura 3.3 mostra o meio pela qual as linguagens computacional e de engenharia estão relacionadas pela introdução de transparências.

Estruturação de Engenharia

Uma especificação de engenharia define a infraestrutura necessária para dar suporte as funções de um sistema ODP, identificando:

- as funções ODP necessárias para o gerenciar distribuição física, comunicação, processamento e armazenamento;
- os pápeis dos diferentes objetos que dão suporte as funções ODP (por exemplo o núcleo);
- as ligações entre os diferentes objetos.

Uma especificação de engenharia é expressa em termos:

- da configuração dos objetos de engenharia;
- das atividades que ocorrem dentro dos objetos [29];
- da interação que ocorrem entre os objetos [38, 45, 37].

Uma especificação de engenharia deve seguir regras da linguagem de engenharia. Estas regras são:

- regras de canal;

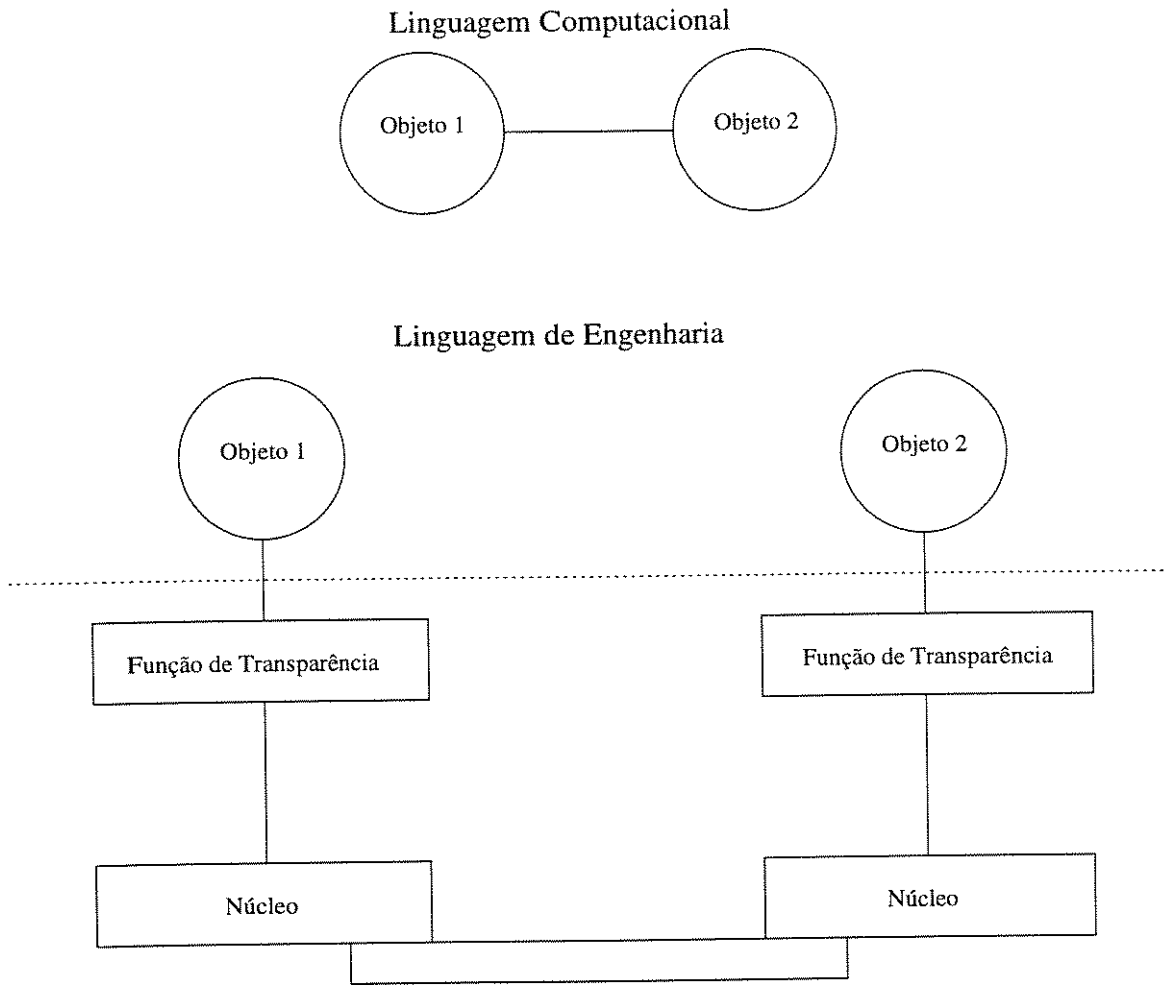


Figura 3.3: Visão computacional e de engenharia da interação de objetos

- regras de estabelecimento de um canal;
- regras de referência de interfaces;
- regras de *cluster*;
- regras de cápsula;
- regras de nó;
- regras de falha.

Regras de canal

Um canal é responsável pela transparência 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 e objetos consumidores.

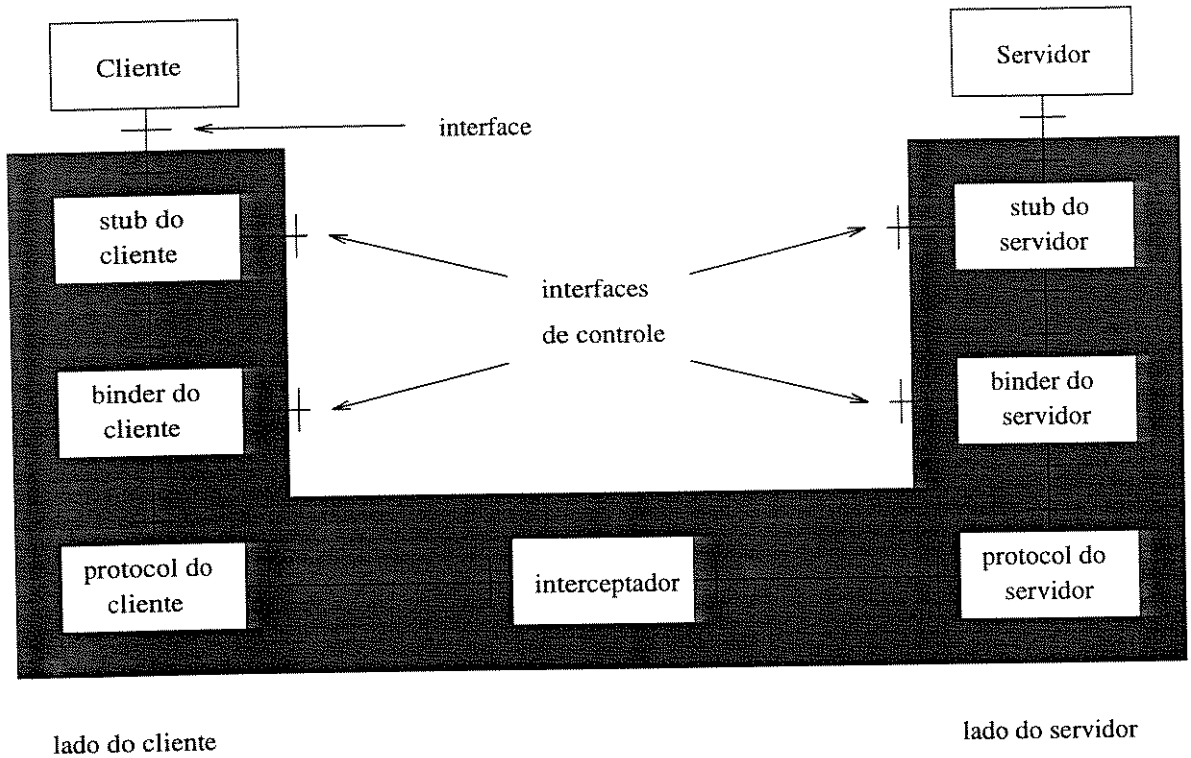


Figura 3.4: Exemplo de um simples canal cliente/servidor

Um exemplo de canal está ilustrada na figura 3.4.

Um canal é uma configuração de objetos *stubs*, *binders* e *protocols* 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

Os objetos de engenharia que estão interagindo são ligados a *stubs*. Os *stubs* são responsáveis pela conversão/desconversão de dados durante a interação. Os *stubs* podem controlar e manter registros (por exemplo, para segurança e auditoria). Eles podem interagir com outros objetos fora do canal (por exemplo, uma função de segurança) caso seja necessário.

Um *stub* tem:

- uma interface de apresentação para interação com um objeto de engenharia;
- uma interface de controle para gerenciamento da qualidade de serviço [9, 33].

Binders

Os *binders* em um canal gerenciam a integridade fim-a-fim e a qualidade de serviço do canal. Os *binders* devem prover transparência de relocação quando for necessário, monitoração de falhas de comunicação e reparar conexão quebradas.

Um *binder* tem interfaces de controle que possibilitam:

- mudanças na configuração do canal;
- destruição de todo, ou parte, de um canal.

Protocols

Os objetos *protocol* provêem funções de comunicação. Eles podem interagir com objetos fora do canal para obter as informações adicionais. Quando os objetos *protocol* estão em domínios de comunicação diferentes eles requerem um interceptador para que possam se interagir entre si.

Interceptadores

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

Regras de estabelecimento de um canal

O núcleo estabelece canais configurando adequadamente objetos *stubs, binders, protocols* e interceptadores. Os canais são estabelecidos em estágios (ilustrado na figura 3.5):

1. primeiro, um objeto (E1) inicia a configuração de um canal interagindo com seu núcleo:
   - a interação é parametrizada pelo tipo do canal, um papel (define a qualidade do serviço) e uma interface de um objeto (E2) para ser unida ao canal;
   - o núcleo une a interface de E2 com a interface de apresentação de um *stub* (S1), o *stub* com um *binder* (B1) e o *binder* com um *protocol* (P1);
   - o tipo do *stub, binder* e *protocol* são selecionados e determinados pelos parâmetros tipo do canal e papel;
   - o resultado da interação é uma referência de uma interface de engenharia (I) para comunicação com outros objetos e a união das interfaces de controle do *binder* (B1) e *stub* (S1) com o objeto (E1).
2. segundo, a referência de interface (I) é comunicada para outro objeto de engenharia (E3), possivelmente em outro *cluster*, cápsula ou nó;
3. terceiro, o objeto (E3) interage com seu núcleo para ligar o canal:
   - a interação é parametrizada pelo tipo do canal, um papel e uma interface de um objeto (E4) para ser unida ao canal;
   - o núcleo une a interface de E4 com a interface de apresentação de um *stub* (S2), o *stub* com um *binder* (B2) e o *binder* com um *protocol* (P2) e, se for necessário, o *protocol* com um interceptador;
   - o tipo do *stub, binder, protocol* e se necessário um interceptador são selecionados e determinados pelos parâmetros tipo do canal e papel;
   - é unido o protocol (P1) com o protocol (P2) através de um interceptador se o canal o utilizar;
   - o núcleo une as interfaces de controle do *binder* (B2) e *stub* (S2) com o objeto (E3).

Regras de referências de interface

A referência de uma interface de engenharia identifica sem ambiguidades no espaço e no tempo as interfaces dos objetos de engenharia. Os objetos de engenharia podem ser realocados porém isto não invalida as referências de interface de engenharia desses objetos.

Os *binders* são responsáveis por manter o mapeamento das referências de interface de engenharia para identificadores de interface de comunicação quando os objetos são relocados, em cooperação com a função de relocação.

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 instaciado, para a interface;
- a localização no espaço e no tempo de alguma interface de comunicação, para a interface.

Regras de cluster

Um *cluster* contém um conjunto de objetos básicos de engenharia, associados a um gerenciador de *cluster*. Cada membro de um *cluster* tem uma interface que dá suporte a função de gerenciamento do objeto. Cada interface de gerenciamento de objeto deve ser associado ao gerenciador de *cluster*. Um *cluster* é sempre contido numa única cápsula.

O gerenciador de *cluster* é responsável por gerenciar a política para os objetos no *cluster*.

Um *cluster* é responsável por sua própria segurança, mas pode ser assistido por funções de segurança. Ele também é responsável pelo gerenciamento das referências de interface de engenharia para as interfaces dos objetos no *cluster*, mas pode ser assistido pela função de gerenciamento de referências de interface de engenharia.

Um objeto básico de engenharia num *cluster* é sempre unida a:

- seu núcleo, através de uma interface provida pela função de gerenciamento do nó;
- seu gerenciador de *cluster*.

Todas as outras interfaces de um objeto básico de engenharia são:

- ligadas a outro objeto de engenharia no mesmo *cluster*, ou
- ligadas a um canal.

O gerenciador de *cluster* é associado ao gerenciador de cápsula.

A figura 3.6 ilustra esta estrutura.

A instanciação de um *cluster* (incluindo a clonagem de um *cluster*) é desempenhada por um gerenciador de cápsula.

Um *template* de *cluster* (incluido *checkpoints* de um *cluster*) especifica a configuração dos objetos no *cluster* e qualquer atividade necessária para instanciá-los e estabelecer suas ligações. Se o *template* é um *checkpoint* de um *cluster*, a instanciação (isto é, clonagem) possibita que o novo *cluster* aja como um substituto do *cluster* original da qual o *template* do *cluster* foi derivado. Neste caso todas as ligações que eram mantidas pelo *cluster* original devem ser reestabelecidas.

Regras de cápsula

A estrutura de uma cápsula é ilustrada na figura 3.7. Uma cápsula consiste de:

- *cluster(s)*;
- gerenciadores de *cluster*, um para cada *cluster* na cápsula;
- objetos *stubs, binders* e *protocols* para cada canal que é ligado a uma interface de um objeto básico de engenharia dentro de alguns dos *clusters*;
- um gerenciador de cápsula na qual é ligado cada gerenciador de *cluster* presente na cápsula.

O gerenciador de cápsula é responsável por gerenciar a política para os *clusters* na cápsula. Ele possui uma interface que provê a função de gerenciamento da cápsula.

A cápsula é sempre contida dentro de um único nó. A instanciação de uma cápsula é desempenhada pelo núcleo utilizando um *template* de uma cápsula que especifica a configuração inicial dos objetos na cápsula. A estrutura de engenharia no sistema ODP deve prover mecanismos para a interação entre *clusters*, gerenciadores de cápsula e o núcleo.

Regras de nó

Um nó consiste de um núcleo e um conjunto de cápsulas. Todos os objetos num nó comportam as mesmas funções de processamento, armazenagem e comunicação. A estrutura de um nó está ilustrada na figura 3.8.

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

O procedimento de instanciação de um nó está fora do modelo ODP, mas deve resultar em:

- introdução de um núcleo ao nó e associado com ele as funções de processamento, armazenagem e comunicação, incluindo funções que possibilitam a ligação entre interfaces;
- introdução de alguma função de negociação necessária para o processo de instanciação;
- alguns canais necessários como parte da configuração inicial do nó (por exemplo, para objetos de suporte tais como um realocador);
- um conjunto de objetos *protocol* que determina o conjunto inicial de domínios de comunicação na qual o nó pertence.

Regras de falha

Uma falha pode ser localizada num *cluster*, numa cápsula ou num nó.

- uma falha localizada num *cluster* pode ser detectada pelo seu gerenciador de *cluster*;
- uma falha localizada num cápsula pode ser detectada pelo seu gerenciador de cápsula;
- uma falha de um nó pode ser detectada pelos objetos *protocol* dos outros nós da qual ele está interconectado.

3.6.5 Linguagem Tecnológica

A linguagem tecnológica mostra as possíveis soluções técnicas para os requerimentos da linguagem de engenharia, mostrando o custo e a disponibilidade de *hardware* e/ou *software* que o satisfaçam. O modelo ainda apresenta poucas regras aplicadas a linguagem tecnológica. Trabalhos sobre a linguagem tecnológica ainda estão em andamento.

No capítulo seguinte veremos a descrição da plataforma *multiware*, na qual este trabalho se insere.

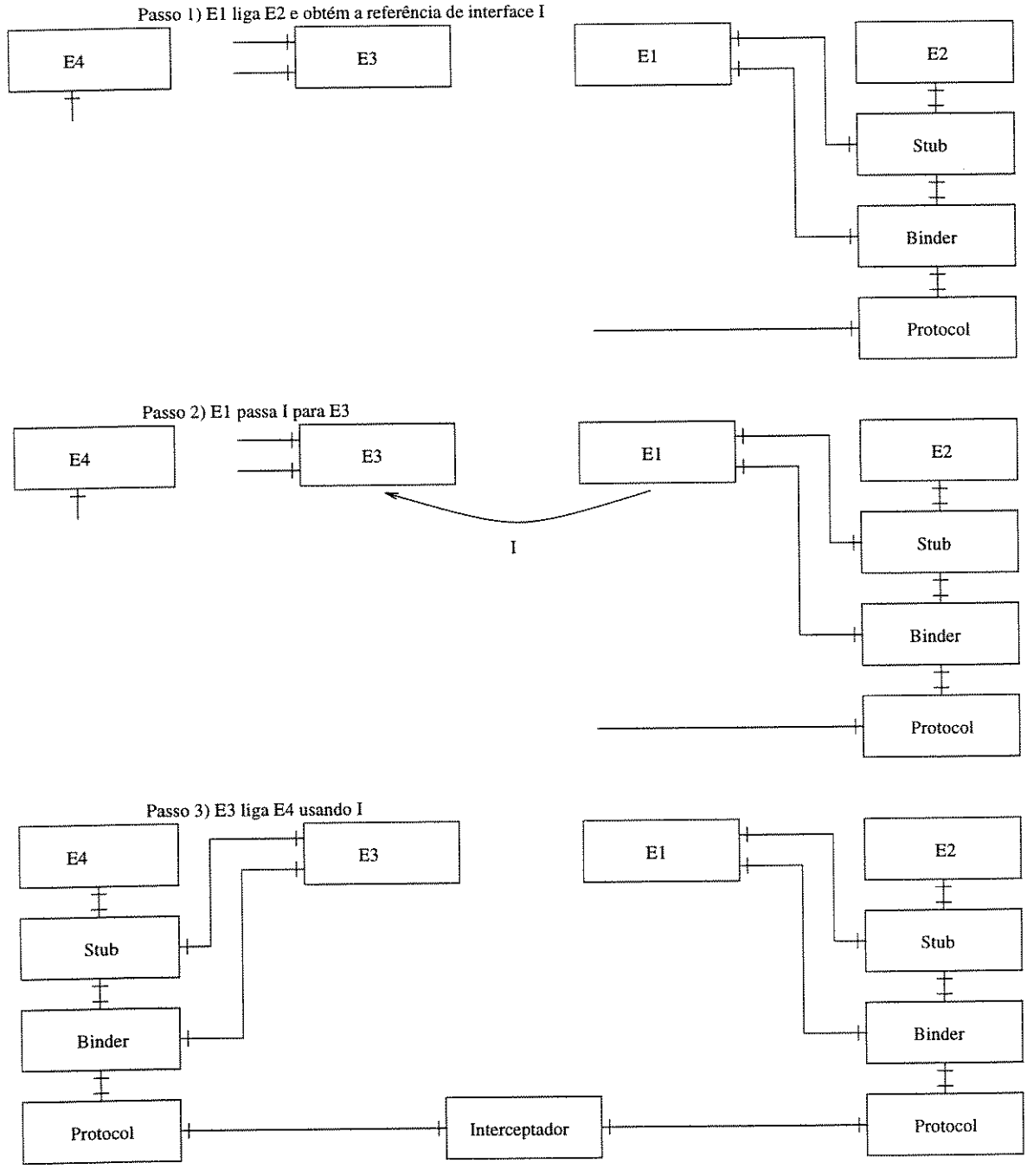


Figura 3.5: Ilustração das regras de estabelecimento de um canal

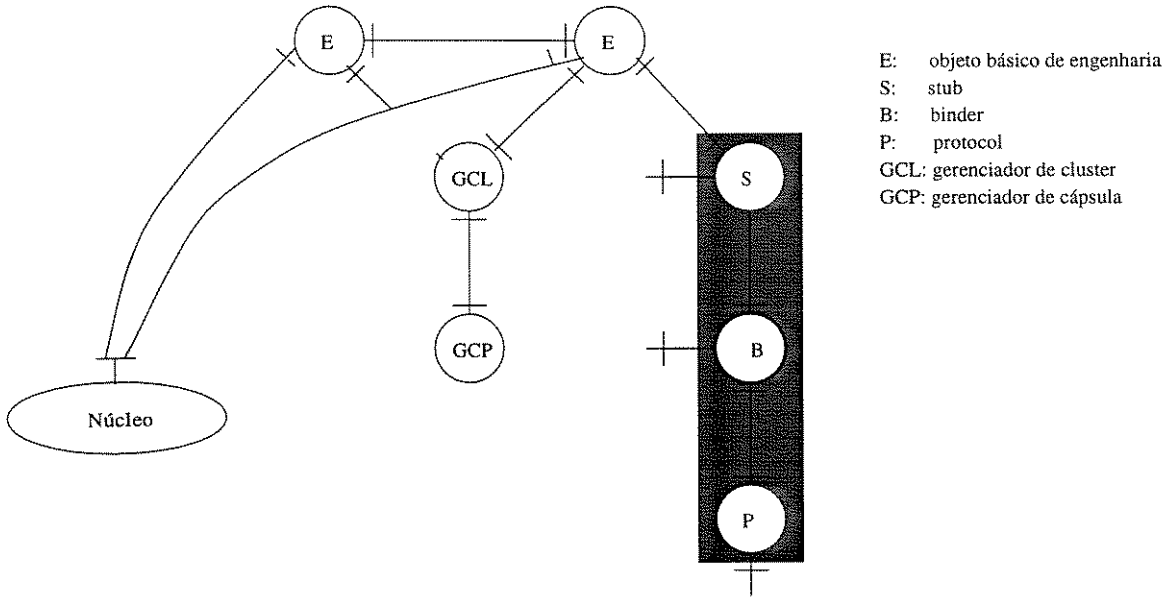
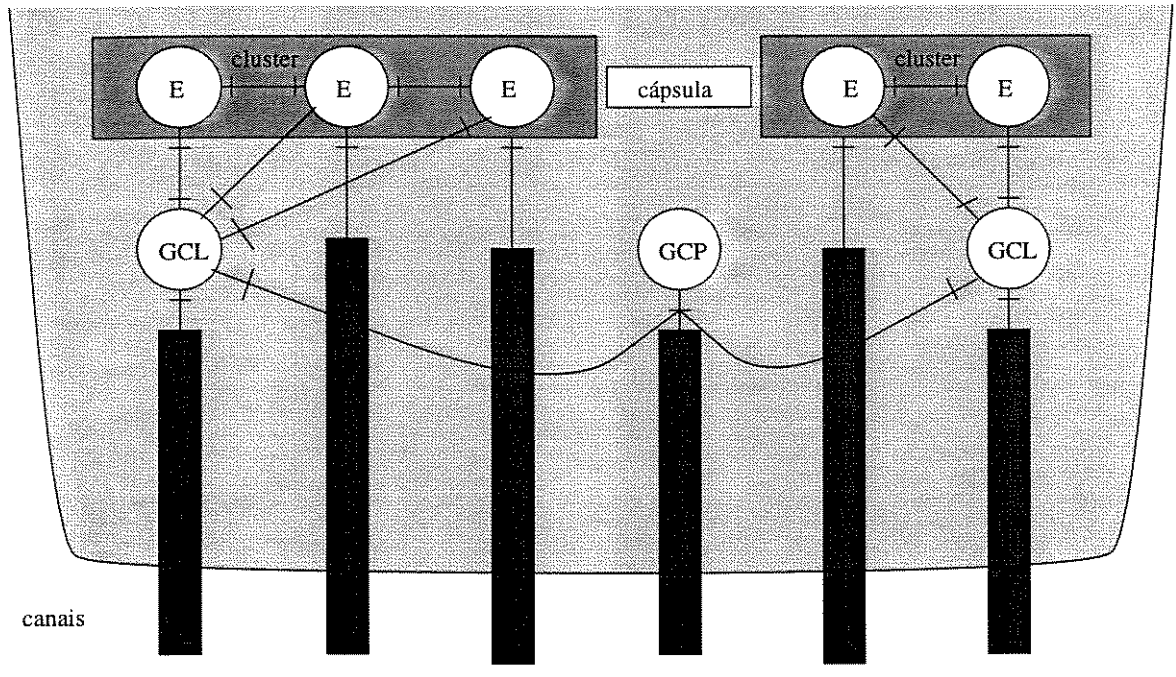


Figura 3.6: Exemplo de uma estrutura que dá suporte a objetos básicos de engenharia



E: objeto básico de engenharia
GCL: gerenciador de cluster
GCP: gerenciador de cápsula

Figura 3.7: Exemplo de uma estrutura de uma cápsula

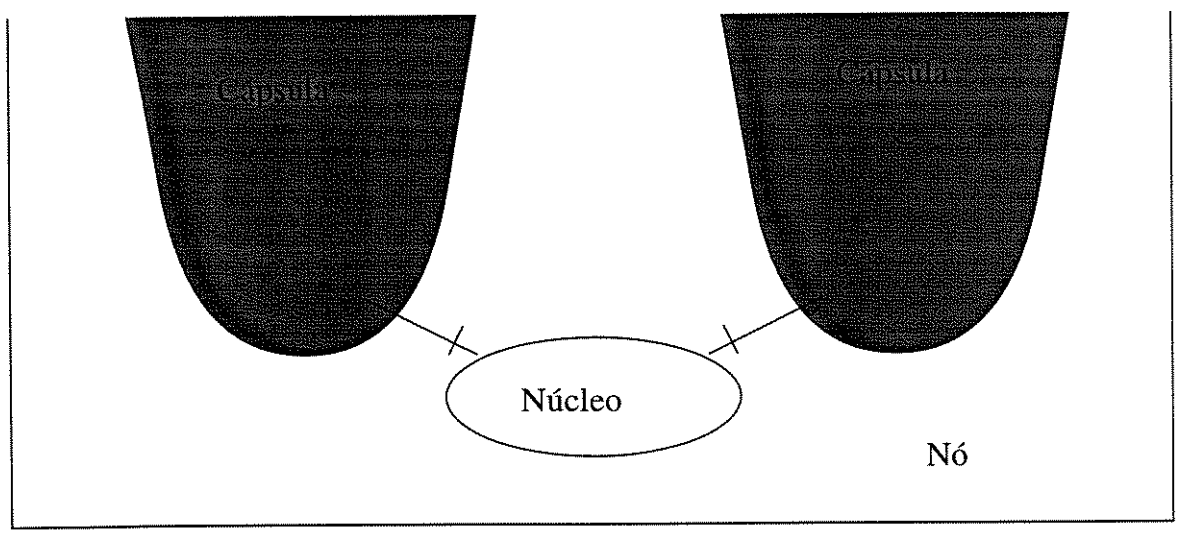


Figura 3.8: Exemplo de uma estrutura de um nó

Capítulo 4

Plataforma Multiware

4.1 Introdução

Conforme citado no capítulo anterior a seguir será feita a descrição da plataforma *multiware*.

No ambito do projeto temático financiado pela FAPESP [8] a plataforma *multiware* é um modelo para o suporte ao processamento distribuído aberto (ODP: Open Distributed Processing). Este modelo de referência se presta a identificar os vários componentes e serviços de uma plataforma de suporte ao processamento distribuído aberto. A plataforma *multiware* significa o uso de [7]:

- múltiplas plataformas de *hardware* (RISC, CISC, etc.);
- múltiplos sistemas operacionais (UNIX, Windows NT, OS/2, etc.);
- múltiplas linguagens de programação (C++, CLOS, etc.);
- múltiplas formas de informação (dados, vídeo, voz, etc.);
- múltiplas infraestruturas de comunicação (Ethernet, Token Ring, DQDB, FDDI, ATM, B-ISDN, etc.);

Em termos funcionais, a plataforma *multiware* foi dividida em quatro camadas, conforme a fig. 4.1. Uma breve descrição das quatro camadas é fornecida abaixo. A seguir apresentamos uma descrição mais detalhada da camada *middleware* onde este trabalho está focalizado.

- **camada de processamento local:** esta camada é constituído pelo processador com o seu software básico como sistema operacional (podendo ser baseado em *microkernel*), sistema gráfico nativo de apresentação, protocolos de comunicação, etc. Esta camada não provê suporte ao processamento distribuído.
- **camada** *middleware*: esta camada provê facilidades de processamento distribuído para as camadas superiores. Alguns serviços oferecidos por esta camada são: serviço de nomes, gerenciamento de recursos distribuídos e replicados, processamento de transações, transparência de acesso e localização, dentre outros. A camada pode utilizar os recursos de plataformas comerciais, base de dados comerciais e não comerciais e de outros produtos comerciais ou não comerciais, mas deve mascarar o uso

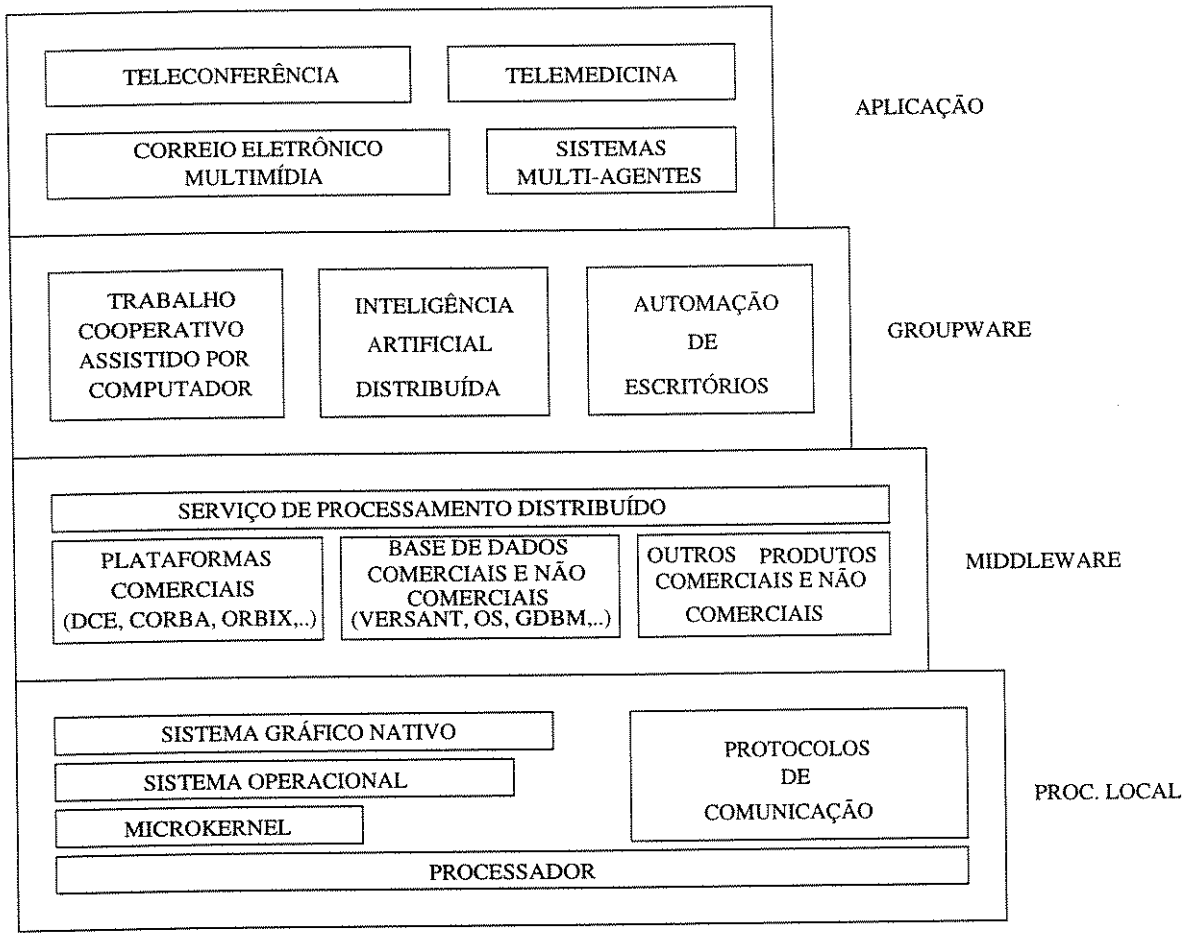


Figura 4.1: Estruturação de uma plataforma *multiware*

desses recursos adequando-os ao modelo de referência ODP. Esta camada deve dar suporte ao paradigma de programação orientada a objetos e a criação das abstrações do sistema ODP (objetos, *clusters*, cápsulas).

- **camada** *groupware*: esta camada provê as funcionalidades requeridas por diferentes classes de aplicações, como trabalho cooperativo suportado por computador (CSCW: *Computer Supported Cooperative Work*), inteligência artificial distribuída (DAI: *Distributed Artificial Intelligence*), automação de escritórios, etc. Serviços típicos oferecidos por esta camada são: gerenciamanto de diálogos, protocolos de interação, manipulação de documentos multímidia, dentre outros.

- **camada de aplicação:** esta camada reune aplicações específicas que se utilizam dos serviços da camada *groupware* oferecidos para a sua respectiva classe. Exemplos típicos: teleconferência e telemedicina (da classe CSCW); sistemas multi-agentes (da classe DAI); correio eletrônico multimídia (da classe automação de escritórios).

4.2 Camada Middleware

A camada *middleware* deve ser estruturada levando-se em conta a tecnologia de orientação a objetos. Aplicando-se esta tecnologia, as camadas superiores podem acessar os serviços

de processamento distribuído através de uma interface uniforme. Além disso, esta tecnologia provê mais facilidades para a abstração de um projeto. A camada *middleware* dá suporte ao processamento distribuído aberto aderente ao modelo de referência ODP, provendo serviços que permitem as seguintes funcionalidades presentes no modelo de referência ODP:

- **gerenciamento de objetos:** o gerenciamento de objetos é feito de uma forma hierárquica no ODP: os objetos são agrupados em *clusters*, os *clusters* são agrupados em cápsulas e por fim as cápsulas residem num nó. Cada nível hierárquico apresenta um gerenciador (Explicado mais detalhadamente na próxima seção). As funções de gerenciamento que devem ser suportadas pela camada *middleware* são: *checkpoint*, recuperação, destruição e criação, dentre outro;.

- **transparência:** o modelo de referência ODP prevê várias formas de transparência para o usuário, tais como localização, acesso, replicação, concorrência, migração, recursos, federação e grupo. A camada *middleware* é responsável pela criação destas transparências;

- **interação entre objetos:** no modelo de referência ODP a interação entre objetos se dá diretamente através de suas interfaces se os objetos estão localizados num mesmo *cluster* e caso contrário existe a necessidade da criação de uma conexão chamada canal. Um canal no modelo de referência ODP pode simplesmente enviar um sinal de um objeto a outro como também pode transportar um fluxo contínuo de dados, tendo suas grandezas monitoradas (taxa de transmissão, atraso, etc) permitindo eventualmente que um fluxo de um canal seja sincronizado com o de outro canal (verificando a perda de sincronismo e tomando as medidas adequadas para corrigi-lo). Um objeto pode requerer uma interação multi-ponto (de-um-para-muitos). A funcionalidade de interação é considerada uma das mais importantes em qualquer plataforma para construção de sistemas distribuídos;

- **negociação** (*trading*)**:** a negociação de serviços no ODP é feito com a intermediação de uma entidade chamada *trader* [22]. Quando um objeto servidor deseja exportar suas funcionalidades este deve enviar para o *trader* todas as características dos serviços que provê, enquanto que um objeto cliente que deseja um serviço, contacta o *trader*, e solicita dele o nome (ou identificador) de um objeto servidor que ofereça o serviço com as caractéristicas desejadas (ou pelo menos próximas da desejada). A camada *middleware* é responsável pela implementação da entidade chamada *trader*;

- **segurança:** funcionalidades tais como: auditoria, autenticação, integridade, confidencialidade dentre outras devem ser suportadas pela camada *middleware* como forma de proteger os objetos;

- **serviços de repositório:** provê meios de armazenamento e persistência de objetos. A camada *middleware* deve prover os mecanismos para o gerenciamento do repositório de dados;

- **serviços de transação:** provê meios para que os objetos permaneçam sempre num estado consistente quando estão participando de uma transação. A camada *middleware* deve prover funções de transação tais como início, cancelamento, compromissamento e bloqueio, dentre outras.

4.2.1 Gerenciamento de Objetos

Como citado anteriormente os objetos são agrupados em um *cluster*, os *cluster* em cápsulas e por fim as cápsulas estão presentes num nó. Cada um destes agrupamentos possui um gerenciador diferente. As funcionalidades destes gerenciadores estão citadas abaixo:

Funções de gerenciamento a nível de nó

O nó é gerenciado pelo seu núcleo. As funções de gerenciamento de nó (acessados através da interface de gerenciamento do núcleo) são:

- gerenciar as *threads* de uma cápsula. As funções de gerenciamento de *threads* são: criar, suspender e sincronizar, dentre outras;
- acessar o relógio (obter o tempo corrente) e gerenciar os *timers* (iniciar, monitorar e cancelar *timers*);
- criar canais e localizar interfaces para *binding* entre objetos;
- instanciar cápsulas através de templates de cápsulas.

Funções de gerenciamento a nível de cápsula

As cápsulas contém um objeto especial, o gerenciador de cápsula, que contém uma interface de acesso as funções de gerenciamento da cápsula. O comportamento de um gerenciador de cápsula é guiado pela política de gerenciamento desta cápsula pelo seu núcleo. As funções de gerenciamento presentes no modelo de referência ODP são:

- instanciar, recuperar (após uma deleção ou falha), reativar (após uma desativação) os *clusters* através de seus templates;
- remover (sem *checkpoint*) a cápsula (removendo todos os *clusters* presentes na cápsula);
- *checkpoint* da cápsula (*checkpoint* em todos os *clusters* presentes na cápsula);
- *desativar* (*checkpoint* seguido de uma remoção) a cápsula (desativando todos os *clusters* presentes na cápsula).

Um gerenciador de cápsula pode implementar parcialmente estas funções de gerenciamento.

Funções de gerenciamento a nível de *cluster*

Um *cluster* é um agrupamento de objetos. Um desses objetos é responsável pelas funções de gerenciamento do *cluster*. Estas funções são acessadas através de uma interface de gerenciamento do objeto responsável pelo gerenciamento do *cluster*. Todos os gerenciadores de *clusters* numa cápsula estão unidos ao seu respectivo gerenciador de cápsula. O comportamento de um gerenciador de *cluster* é guiado pela política de gerenciamento deste *cluster* pelo seu gerenciador de cápsula. As funções de gerenciamento presentes no modelo de referência ODP são:

- modificar a política de gerenciamento do *cluster*;

- desativar (*checkpoint* seguido de uma remoção) o *cluster* (desativando todos os objetos presentes no *cluster*);
- *checkpoint* do *cluster* (*checkpoint* em todos os objetos presentes no *cluster*);
- reativar (após uma desativação) o *cluster* (recuperando todos os objetos presentes no *cluster*);
- migrar o *cluster* para outra cápsula (desativando e reativando todos os objetos presentes no *cluster* em outra cápsula);
- remover (sem *checkpoint*) o *cluster* (removendo todos os objetos presentes no *cluster*).

Um gerenciador de *cluster* não necessariamente precisa implementar todas estas funções de gerenciamento.

Funções de gerenciamento a nível de objeto

Os objetos são unidades computacionais acessadas através de suas interfaces. Cada objeto possui uma interface exclusiva para o propósito de gerenciamento. Todas as interfaces de gerenciamento dos objetos estão unidas ao seu respectivo gerenciador de *cluster*. As funções de gerenciamento de objetos são somente duas:

- *checkpoint* do objeto (salvando todas as informações necessárias para incorporação do *checkpoint*);
- remoção do objeto.

Uma interface de gerenciamento de objetos não necessariamente precisa implementar ambas as funções de gerenciamento

A figura 4.2 mostra a estrutura para computação distribuída do modelo de referência ODP.

4.3 Restrições de Implementação da Camada Middleware

Foi citado anteriormente todas as funcionalidades que uma camada *middleware* deveria ter para estar totalmente condizente com o modelo de referência ODP. Porém nesta implementação inicial uma série de funcionalidades não serão implementadas. Estas "simplificações" estão citadas abaixo:

- **gerenciamento de objetos:** as seguintes funções não serão implementadas em cada hierarquia de gerenciamento:
  - a nível de nó:
    - gerenciamento das *threads* de uma cápsula;
    - acesso a um relógio e *timers*.
  - a nível de *cluster*:

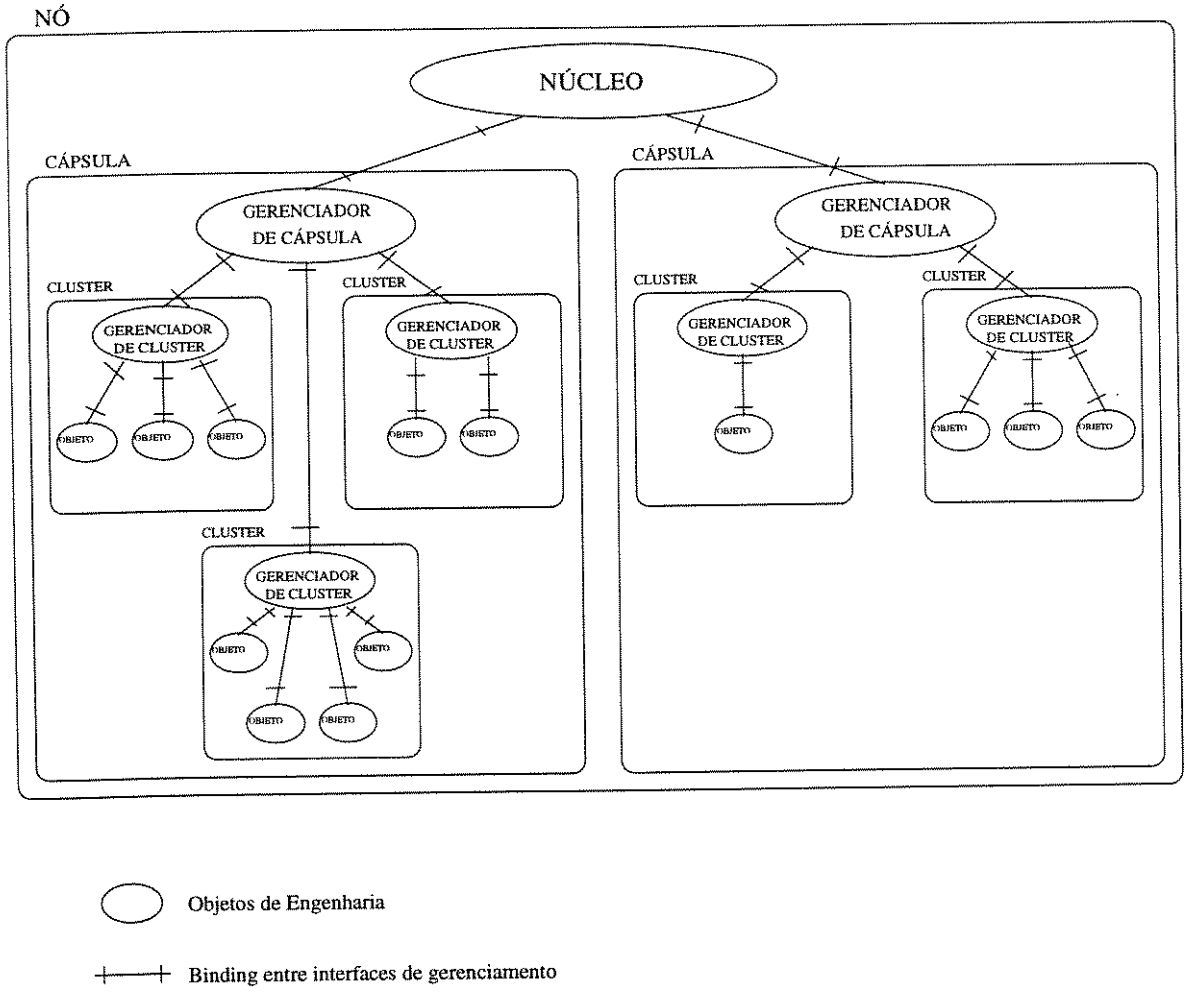


Figura 4.2: Estrutura ODP para computação distribuída

  - * modificar a política de gerenciamento do *cluster*;
  - * migrar o *cluster* para outra cápsula.
- a nível de objeto:
  - * o *checkpoint* do objeto não guarda informações acerca das comunicações que o objeto está fazendo no momento do *checkpoint*. Só será salvo o estado do objeto e as interfaces do objeto.

- **transparência:** somente serão implementadas as transparências de localização e acesso;
- **interação entre objetos:** somente pode ser criado canais ponto-a-ponto e a única grandeza que pode ser monitorada é a taxa de transmissão;
- **negociação** (*trading*)**:** a função de negociação não está presente. A única negociação que existe no sistema diz respeito a largura de banda e ao atraso numa comunição com reserva de banda, porém estes recursos são "negociados" diretamente com o protocolo de comunicação sem o intermédio de um *trader*;
- **segurança:** não foi implementada nenhuma função de segurança. A única segurança é aquela provida pelo próprio sistema operacional nativo;

- **serviços de repositório:** um conjunto mínimo de funcionalidades estará disponível, suportado pelo gbdm [17];

- **serviços de transação:** um conjunto básico de transações será implementado. Implementações mais complexas poderão ser feitas utilizando-se do conjunto básico de operações de transações. Não foi implementado nenhum objeto servidor de transações que monitora completamente uma transação, como no modelo de referência ODP.

4.3.1 Componentes da Camada Middleware

A camada *middleware* é composta atualmente por 4 componentes que têm por responsabilidade prover os recursos para desenvolver as funcionalidades ODP. Os componentes desta camada provêem os seguintes recursos:

- serviços de núcleo;

- serviços de comunicação;

- serviços de objetos;

- interface de programação (API: *Application Programming Interface*).

A ligação entre estes componentes está mostrada na figura 4.3. A seguir será dada uma breve explicação das funcionalidades de cada componente.

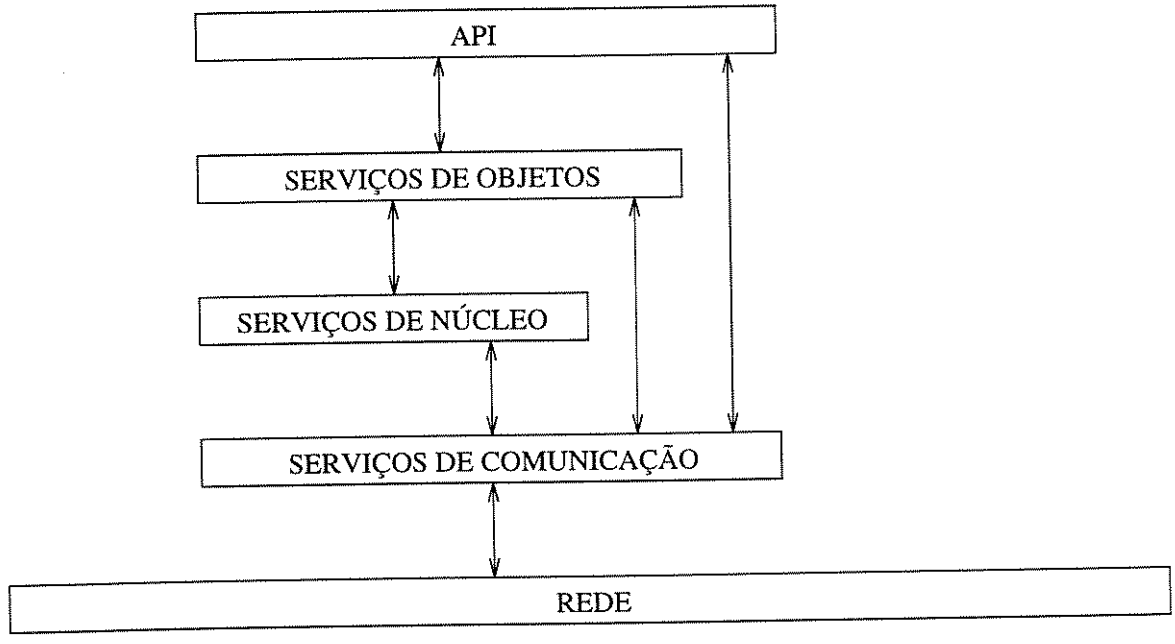


Figura 4.3: Arquitetura *middleware*

Serviços de Núcleo

Os serviços de núcleo são constituidos dos serviços básicos necessários ao processamento distribuído do ODP. Foi implementado um servidor chamado núcleo que fornecerá tais serviços [6]. Este núcleo não é o do modelo de referência ODP. Os serviços básicos implementados pelo núcleo são:

- geração de números globais a uma máquina, globais a uma rede de computadores e contadores para identificação de grupos;
- processamento de transações;
- base de dados elementar;
- mapeamento de *ports* (*portmapping*).

Serviços de Comunicação

Os serviços de comunicação são essencias para o processamento distribuído. São providos três categorias de serviços [35]:

- serviços de comunicação confiável ponto-a-ponto (*reliable unicast*). Nesta categoria de serviço é garantida a entrega de uma mensagem ao seu destino;
- serviços de difusão confiável de mensagens para um grupo de nodos numa rede local (*reliable multicast*);
- serviço de comunicação não confiável de entrega de mensagens, mas com uma qualidade de serviço de reserva de banda e retardo máximo.

Serviços de Objetos

Os serviços de objetos provêem a infraestrutura necessária para a existência dos modelos de processamento distribuído de acordo com o ODP. Os objetos possuem propriedades tais como instanciação e encapsulamento que são providos pelo serviço de objetos. Além disso ele provê os recursos necessários ao gerenciamento de objetos a nível de cápsula, *cluster* e objeto, provendo facilidades para o agrupamento de objetos em *cluster* e de *clusters* em cápsula. Estes serviços estão disponíveis numa base de objetos.

Outro serviço muito importante que é fornecido suporta a interação entre dois objetos que estão em *clusters* diferentes, provendo a criação e monitoração de um canal ligando uma interface de um objeto cliente a uma interface de um objeto servidor.

4.3.2 Interface de Programação

A API fornece um acesso uniforme aos serviços de objetos. A camada *groupware* acessa estes serviços via API. A API consiste de uma hierarquia de classes especificadas segundo a metodologia OMT (*Object Modeling Technique*)[50, 49] e disponíveis na linguagem de programação C++[56]. Uma outra função da API é a criação dos *templates* do modelo de referência ODP [40].

No capítulo seguinte veremos a base de objetos que é responsável pelo serviço de suporte de objetos em tempo de execução e se constitui na principal contribuição deste trabalho.

Capítulo 5

Plataforma ODP: Serviços de Objetos

5.1 Introdução

A base de objetos, como citado no capítulo anterior, é responsável por fornecer os serviços de objetos, dando suporte em tempo de execução para a criação de aplicações que sigam o modelo ODP. Em particular a base de objetos foi projetada para dar suporte a aplicações multimídia, o que exigiu que fosse desenvolvida de tal forma que permitisse o escalomento e a sincronização em tempo-real. Uma descrição detalhada de todos os serviços oferecidos pela base de objetos se encontra no apêndice **A**.

5.2 Suporte em Tempo de Execução

O suporte em tempo de execução da plataforma ODP permite:

- instanciação de objetos;
- agregação de objetos (*clustering*);
- gerenciamento à nível de objeto, *cluster* e cápsula;
- ligação (*binding*) de objetos;
- escalonamento de tempo-real;
- sincronização de tempo-real.

5.2.1 Instanciação de Objetos

Objetos são instanciados através de *templates*. Um objeto é composto de seu estado e de suas interfaces que estão especificados no *template*. Os objetos fazem uso dos recursos alocados para a cápsula a qual pertencem: as interfaces dos objetos são *threads* em execução dentro da cápsula e os estados dos objetos ocupam espaço em memória alocado dinamicamente. O estado de um objeto é determinado pelo conteúdo de seus atributos. Esta implementação permite aos objetos processar serviços em paralelo (por exemplo, objetos podem ter mais de uma interface ativa ao mesmo tempo). Entretanto, o controle de

concorrência quando múltiplas interfaces estão acessando o estado de um mesmo objeto ao mesmo tempo é de responsabilidade do implementador.

No *template* de um objeto está definido um *behaviour*. O *behaviour* armazena código escrito pelo implementador do objeto e é executado tão logo um objeto é instanciado. Tipicamente tal código inicializa o estado do objeto, adiciona e retira interfaces (além daquelas que estão presentes no *template* do objeto e que já são ativadas automaticamente durante o processso de instanciação do objeto). Nos *templates* de objetos também estão especificados os recursos demandados pelos objetos. Tais recursos definem um contrato que é negociado com a intermediação do sistema de suporte a execução de forma que assegure adequadamente os recursos que um objeto necessita para executar sua tarefa. Por exemplo, considere um objeto que coleta e comprime quadros de uma câmera de vídeo e os envia através de uma rede. Tais objetos requerem dois tipos de recursos: acesso periódico da UCP para o processamento da imagem, e largura de banda da rede para transmissão da imagem. Os recursos especificados no *template* devem ser assegurados durante a instanciação do objeto. Se os recursos não forem assegurados, a instanciação do objeto é abortada. Neste exemplo, a UCP é negociada com o escalonador e a largura de banda da rede com o protocolo de reserva de banda [35].

5.2.2 Agregação de Objetos (Clustering)

Os objetos são agregados para o propósito de desativação, remoção, *checkpointing*, reativação, recuperação e migração. A esta agregação de objetos dá-se o nome de *cluster*. Uma migração corresponde a desativação seguida de uma reativação do *cluster* em outra cápsula. *Checkpointing* é o ato de salvar um estado consistente dos objetos, que consiste em salvar os valores dos atributos do objeto e as interfaces ativas do objeto no momento da operação. (Nesta implementação não serão salvos as informações referentes aos canais entre objetos.) A recuperação é a operação de voltar ao seu último *checkpointing*. A desativação corresponde a um *checkpointing* seguido de uma remoção. Por fim, a reativação é a recuperação dos objetos desativados. A princípio não será implementada a operação de migração de um *cluster*.

Os objetos que fazem parte de um *cluster* estão definidos no *template* de um *cluster*. Quando é feito uma operação num *cluster*, todos os objetos do *cluster* serão afetados. Por exemplo, uma operação de desativação de um determinado *cluster*, desativa cada objeto deste *cluster* que sua vez terá todas as suas interfaces desativadas. O sistema de suporte de execução portanto deve ter conhecimento de todos os objetos e interfaces presentes num determinado *cluster*. Todo *cluster* possui um gerenciador de *cluster* que comanda a execução das operações de desativação, remoção, *checkpointing*, reativação, recuperação e migração do *cluster*. Um *cluster* também possui um *behaviour* e um contrato.

5.2.3 Gerenciamento de Objetos

A plataforma provê objetos padronizados para gerenciamento de cápsula e *cluster*. Tais objetos são compostos de interfaces que implementam as funções de gerenciamento (desativação, remoção, *checkpointing*, reativação, recuperação e migração) encontradas no modelo de referência ODP. O usuário pode implementar suas próprias interfaces de gerenciamento acessando algumas primitivas de baixo nível que serão fornecidas pela plataforma (por exemplo, pode-se fazer um *checkpoint* somente de um objeto específico de um

cluster e não de todos os objetos do *cluster*). Entretanto, nos parece que as funções de gerenciamento do ODP são suficientes para uma grande faixa de aplicações.

A plataforma fornece *templates* capazes de gerar objetos que agem como gerenciadores de cápsulas e gerenciadores de *clusters*. As funções de gerenciamento são executadas evocando as operações nas respectivas interfaces de gerenciamento de um objeto.

5.2.4 Binding de Objetos

Os objetos num mesmo *cluster* podem interagir sem a ajuda de um canal. A interação de objetos dentro de um mesmo *cluster* é feita enviando-se os dados diretamente para a interface destino de um objeto. Neste caso, não existe a necessidade da criação de objetos específicos para comunicação (um do lado do emissor e outro do lado do receptor) que constituem um canal. Mais detalhes destes objetos de comunicação serão supridos a seguir.

Se dois objetos localizados em *clusters* diferentes desejam se comunicar faz-se necessário a criação de um canal de comunicação que é composto de dois objetos específicos para comunicação: um do lado emissor e outro do lado receptor. Um objeto de comunicação é composto de uma interface denominada *stub*, uma interface denominada *binder* e uma interface denominada *protocol*. O modelo de referência ODP diz que o *stub*, o *binder* e o *protocol* são objetos, porém neste caso foi feita uma simplificação do modelo para facilitar a implementação.

O sistema de suporte em tempo de execução deve prover mecanismos para interação com estes objetos de comunicação oferecendo primitivas do tipo *submit* e *deliver* pela qual o implementador da aplicação possa enviar e receber dados de um canal. As interfaces de cada objeto do canal se comunicam entre si enviado dados diretamente umas para as outras. A comunicação fim-a-fim via rede se dá através das duas interfaces *protocols* do canal, conforme ilustrado na figura 5.1

O *stub* é uma interface de um objeto de comunicação que tem como função prover conversão/desconversão de dados para acesso transparente na interação de interfaces de objetos que tiverem representação de dados diferentes.

O *binder* é uma interface de um objeto de comunicação que tem como função monitorar o canal. Atualmente a única atribuição do *binder* é atualizar periodicamente a informação da taxa de transmissão de dados que passa através do canal. Com isso é possível detectar a degradação de qualidade de serviço.

O protocol é uma interface de um objeto de comunicação que tem como função enviar (receber) dados para (de) outra interface protocol definida num objeto de comunicação localizado em outro *cluster*. A comunicação faz uso dos protocolos descritos no *reliable unicast* e de reserva de banda [35]. Esta interface é responsável pela comunicação fim-a-fim.

A plataforma proverá *stubs*, *binders* e protocols padronizados já definidos em *templates* (que são instanciados durante a criação de canais). A comunicação multi-ponto não será implementada como já foi dito anteriormente.

5.2.5 Escalonamento em Tempo-Real

O escalonamento é executado em dois níveis. Escalonamento a nível de nodo que é de responsabilidade do sistema operacional do nodo. Escalonadores de nodo executam

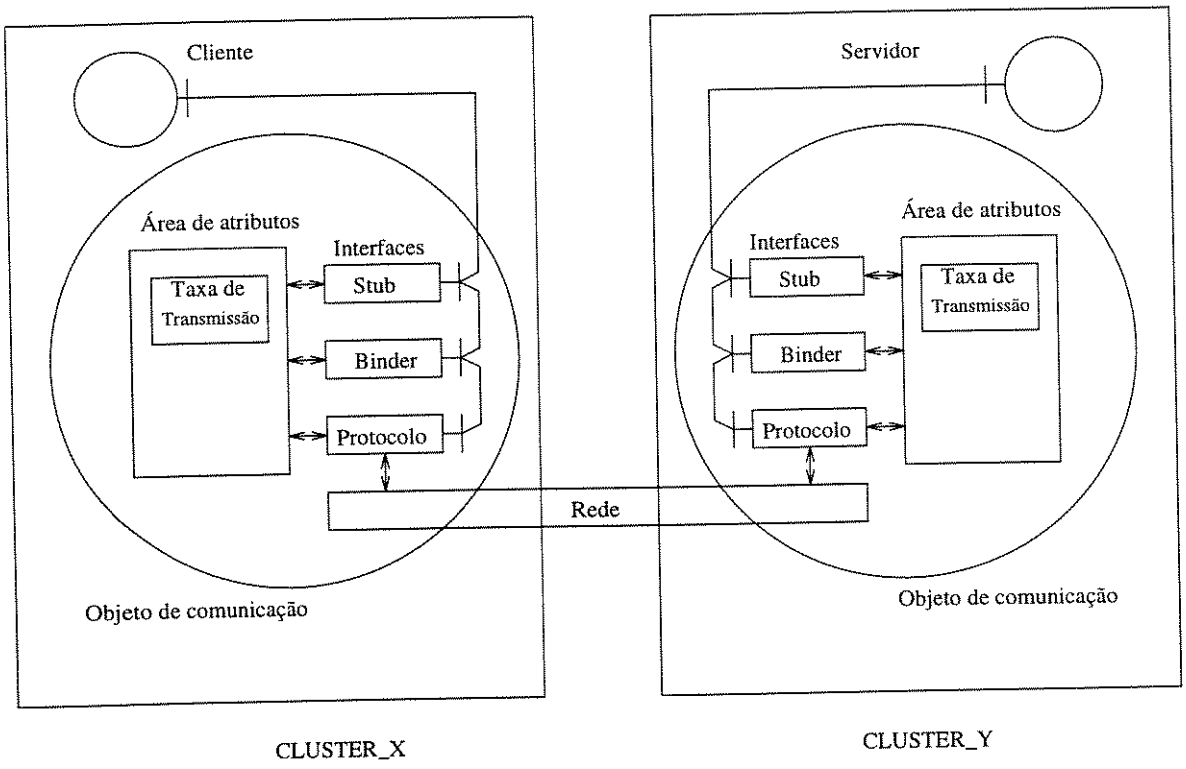


Figura 5.1: Comunicação entre objetos através de um canal

escalonamento de cápsulas (implementados como processos pesados). No segundo nível, cada cápsula tem um escalonador que é responsável pelo escalonamento de interfaces (*threads*) mantidas por objetos dentro da cápsula. O escalonamento a nível de nodo utiliza um esquema baseado em prioridade que é encontrado atualmente em muitos sistemas operacionais. Quanto menor o número de cápsulas num nodo melhor será o escalonamento a nível de nodo em termos de previsibilidade. De fato, em aplicações multimídia é esperado definir o menor número possível de cápsulas por nodo (idealmente somente uma).

Algumas interfaces podem ter requerimentos de tempo-real que devem ser satisfeitos pelo sistema de execução. Estes requerimentos estão presentes no *template* da interface, mais especificamente no seu contrato. O requerimento de tempo-real mais comum é a necessidade para execução periódica. No caso mais simples, uma *thread* torna-se eleita para executar sua tarefa no começo de cada período de tempo e deve executar antes que o período expire (o tempo de pronto coincide com o início do período e o *deadline* coincide com o fim do período). *Threads* com requerimentos periódicos são escalonadas de acordo com a política *earliest-deadline* [28]. Esta política de escalonamento prioriza a *thread* cujo *deadline* está mais próximo do final do período.

Embora a política *earliest-deadline* seja fácil de ser implementada e possua propriedades de tempo-real interessantes, acreditamos que escalonadores mais especializados a nível de cápsula devem ser implementados, principalmente para aplicações multimídia.

5.2.6 Sincronização em Tempo-Real

Em aplicações multimídia distribuídas as imagens em movimento, os sons e o texto devem ser sincronizados em tempo-real. Num computador *stand-alone* onde os dados se encontram todos armazenados num CD-ROM a sincronização é muito mais simples pois

os diversos tipos de fluxos se encontram juntos e a medida o programa lê os dados multimídia este só verifica em que dispositivo ele deve enviá-lo (áudio ou vídeo). Em aplicações multimídia distribuída os diversos tipos de dados são transmitidos por canais distintos para que o sistema tenha um maior controle sobre eles. Ou seja é necessário haver um protocolo que garanta reserva de banda suficiente para cada tipo de fluxo. Ademais os fluxos devem ser sincronizados e os problemas de perdas de dados durante a transmissão devem ser contornados de alguma forma. O trabalho de sincronização em tempo-real está sendo desenvolvido em paralelo com este projeto [62].

5.3 Implementação do Sistema de Suporte em Tempo de Execução

O sistema de suporte de execução para objetos distribuídos foi dividido em quatro módulos funcionais que são o Escalonador, o Gerenciador de Objetos (que inclui também *clusters* e interfaces), o Gerenciador de *Binding* e a Interface com a API, conforme a figura 5.2.

Os módulos funcionais e/ou *threads* que são criados durante a instanciação de objetos se comunicam externamente com a API e com os protocolos de comunicação. Eles ainda fazem acesso ao repositório de *templates* e também as primitivas do núcleo.

Todos os módulos funcionais são *threads* e portanto estão no mesmo espaço de endereçamento. No projeto destes módulos funcionais tomou-se o cuidado para que o código fosse reentrante e com isto evitar-se inconsistência dos dados. Porém, quando as *threads* são implementadas pelo usuário, fica a cargo deste a responsabilidade de fazer com que o código seja reentrante. O sistema de suporte de execução corresponde a um processo pesado ou a uma cápsula de acordo com a definição ODP. A seguir é dada a funcionalidade de cada módulo.

5.3.1 Escalonador

Este é o escalonador a nível de cápsula citado anteriormente. O escalonador é a *thread* responsável pelo escalonamento de todas as *threads* que serão ativadas na cápsula, inclusive dos outros módulos funcionais. Devido a este fato, o escalonador deve ser a *thread* de mais alta prioridade da cápsula. O escalonador executa sua tarefa (escolhe qual a *thread* que será colocada em execução) e logo depois se auto bloqueia por um determinado período de tempo. Neste tempo em que o escalonador está bloqueado a *thread* escolhida é executada. Expirado o intervalo de tempo em que o escalonador permanece bloqueado, este é novamente ativado e escolhe uma *thread* para ser colocada em execução (nada impede que seja a mesma *thread* que já estava executando). Este ciclo permanece indefinidamente. O escalonador receberá pedidos de escalonamento de *threads* que são criadas pelos objetos que são instanciados pelo Gerenciador de Objetos e pelo Gerenciador de *Binding*. Nesta implementação não foi desenvolvido um escalonador *earliest-deadline* em tempo-real.

5.3.2 Gerenciador de Objetos

O Gerenciador de Objetos é responsável por criar, destruir, ler atributos, modificar atributos, efetuar *checkpointing*, desativar e reativar um objeto. Deve também adicionar, remover e listar as interfaces de um objeto. A nível de *cluster* é responsável por criar,

destruir, adicionar objetos, retirar objetos, acessar objetos, desativar, reativar e proceder *checkpointing* de um *cluster*. Por fim a nível de interface o Gerenciador de Objetos é responsável por criar, destruir, enviar dados e receber dados. O Gerenciador de Objetos mantém uma lista com todas as interfaces, objetos e *clusters* da cápsula. Toda interface, objeto e *cluster* possui um identificador global que o identifica univocamente em toda a plataforma.

5.3.3 Gerenciador de Binding

O Gerenciador de *Binding* é responsável por criar, destruir, enviar dados e receber dados através de um objeto de comunicacão além de ativar um objeto de comunicação que contém as interfaces *stub*, *binder* e *protocol*. Durante a criação do canal no lado cliente é solicitado a criação da outra metade do canal do lado do servidor. Existe registrado no núcleo o nodo e a porta de comunicação de uma cápsula na qual a outra metade do canal deve ser ativada. Resumidamente, a formação de um canal de comunicação envolve três fases bem distintas que são:

1. formação de um lado do canal (ativação de um objeto de comunicação) feito por um Gerenciador de *Binding* (lado do cliente, emissor);

2. durante a ativação do objeto de comunicação do lado do cliente é feita uma consulta ao núcleo (serviço de mapeamento) que indicará em qual cápsula a outra metade do canal deve ser criada (existe a indicação do nodo e da porta de comunicação desta cápsula);

3. formação do outro lado do canal (ativação de um objeto de comunicação) feito por um Gerenciador de *Binding* remoto (lado do servidor, receptor).

Duas interfaces de objetos podem pertencer a uma mesma cápsula (dois objetos que pertecem a *clusters* diferentes, porém na mesma cápsula) e nesse caso o Gerenciador de *Binding* envolvido do lado do cliente e do servidor serão os mesmos. O Gerenciador de *Binding* após a criação de um canal, ficará responsável por enviar e receber dados através do objeto de comunicação do canal. Deve também oferecer meios de monitorar (observar a taxa de transmissão de dados) e destruir um canal (destruindo os objetos de comunicação que compõem o lado do cliente e o lado do servidor). O Gerenciador de *Binding* mantém uma lista com todos os objeto de comunicação da cápsula. Todo objeto de comunicação possui um identificador global que o identifica univocamente.

Sequência de Eventos para a Criação de um Canal

Os eventos disparados para a criação de um canal são mostrados na figura 5.3. Cada evento é descrito a seguir:

1. um cliente pede a criação de um canal de comunicação para a API;

2. a API envia o pedido de criação de canal para o sistema de suporte em tempo de execução no nodo onde o cliente executa;

3. o sistema de suporte em tempo de execução do lado do cliente envia um pedido de criação da outra metade do canal para uma determinada cápsula. A localização desta cápsula é obtida utilizando os serviços de mapeamento do núcleo que indica qual o nodo e a porta de comunicação pela qual podem ser feitos pedidos para esta cápsula;

4. o sistema de suporte de execução no nodo onde a outra cápsula executa cria a outra metade do canal (ativa um objeto de comunicação) e o *behaviour* da interface *protocol* registra o nome do serviço com o respectivo nodo e porta de comunicação pela qual uma outra interface *protocol* possa se conectar;

5. a interface *protocol* do lado cliente consulta o serviço de mapeamento do núcleo e se conecta com a interface *protocol* do lado servidor;

6. o servidor executa uma chamada *accept* e aguarda o identificador do objeto de comunicação do lado do servidor da API;

7. a API do lado do servidor envia o pedido de espera de um identificador do objeto de comunicação para o sistema de suporte em tempo de execução do nodo onde ele executa;

8. o sistema de suporte em tempo de execução do lado do servidor envia o identificador do objeto de comunicação do seu lado para o lado do cliente;

9. a API do lado do servidor recebe o identificador do objeto de comunicação do seu lado;

10. o servidor recebe o identificador do objeto de comunicação do seu lado;

11. a API do lado do cliente recebe o identificador do objeto de comunicação do seu lado;

12. o cliente recebe o identificador do objeto de comunicação do seu lado.

Sequência de Eventos para a Destruição de um Canal

Os eventos disparados para a destruição de um canal de comunicação são mostrados na figura 5.4. Cada evento é descrito a seguir:

1. um cliente pede a destruição de um canal de comunicação para a API;

2. a API envia o pedido de destruição de canal para o sistema de suporte em tempo de execução no nodo onde o cliente executa;

3. o Gerenciador de *Binding* do lado cliente destrói o objeto de comunicação do seu lado do canal e também envia um pedido de destruição do objeto de comunicação do outro lado do canal;

4. o Gerenciador de *Binding* do lado cliente recebe a notificação que o outro extremo do canal foi destruído;

5. após receber as notificações de que os objetos de comunicação do lado do cliente e do lado do servidor estão destruídos, uma resposta de que a operação foi concluída é enviada para a API;

6. a API notifica para a aplicação que o canal foi destruído.

5.3.4 Interface com a API

Esta interface é responsável por receber pedidos das APIs. A Interface com a API ao receber um pedido, enviará este pedido para o Gerenciador de Objetos ou o Gerenciador de *Binding* para que um deles possa processá-lo. Esta interface é implementada acima do protocolo de *reliable unicast*.

5.3.5 Diagrama Funcional do Sistema de Suporte em Tempo de Execução

A Interface com a API do sistema de suporte em tempo de execução recebe as chamadas do tipo RPC das APIs com seus respectivos parâmetros. A interface deve identificar o endereço de retorno (que neste caso é o nome do nodo e o número de uma porta de comunicação) que está presente nos parâmetros da chamada.

A Interface com a API não executa nenhum serviço, tendo apenas a responsabilidade de despachar o pedido para o módulo adequado.

Os módulos para os quais a Interface com a API envia os pedidos são o Gerenciador de Objetos e o Gerenciador de *Binding* que mantém uma tabela com os serviços solicitados a eles que estão em execução. Os serviços são executados como *threads*. A figura 5.5 mostra a interação da API, da Interface com a API, do Gerenciador de Objetos e do Gerenciador de *Binding*.

Funcionamento do Gerenciador de Objetos

O Gerenciador de Objetos é responsável por disparar *threads* que possuam relação com interfaces, objetos e *clusters*. Este gerenciador mantém tabelas internas com a identificação de todos os objetos do sistema que pertençam às classes OBJETO, *CLUSTER* e INTERFACE. O Gerenciador de Objetos executará alguns métodos das classes OBJETO, *CLUSTER* e INTERFACE como se fossem *threads*. O Gerenciador de Objetos receberá as requisições da API que deverão ter como parâmetro o identificador do objeto na qual a operação será executada (A única exceção é no momento da criação dos objetos, quando o identificador do objeto é retornado). Como o identificador do objeto é passado como parâmetro, é possível verificar se o objeto existe na tabela interna do Gerenciador de Objetos. Caso não exista a operação é imediatamente cancelada. Esta estruturação permite que mais de uma API solicite serviços simultaneamente. Neste caso o pedido do segundo serviço não ficará bloqueado aguardando o término do primeiro. O Gerenciador de Objetos pode manter uma tabela com o serviço e a identificação da *thread* que está executando o pedido pois desta forma pode-se cancelar o serviço, destruindo-se a *thread* responsável pelo seu processamento. Se as *threads* retornam algum valor, o valor é enviado para um determinado nodo e para uma determinada porta de comunicação que foram passados à *thread* no momento em que o serviço foi solicitado.

Funcionamento do Gerenciador de Binding

O Gerenciador de *Binding* mantém tabelas dos objetos de comunicação criados por clientes e servidores. Quando a API pede um serviço de *Binding* é criada uma tabela com o identificador do objeto de comunicação e a cada identificador está associado uma interface *stub*, uma interface *binder* e uma interface *protocol* do lado cliente. A interface *protocol* cria uma requisição para um servidor. Do lado do servidor o comando *AcceptBind* aceita os pedidos de conexão vindos do lado cliente. É criada também uma tabela com o identificador do objeto de comunicação e a cada identificador está associado à interfaces *stub*, *binder* e *protocol*. Do lado servidor o Gerenciador de *Binding* deve manter uma tabela de *Bindings* pendentes. Podem ocorrer dois casos:

1. existem requisições de *Binding* pendentes - neste caso o comando *AcceptBind* é executado imediamente, o identificador do objeto de comunicação é capturado e o *Binding* pendente é retirado da lista. O identificador do objeto de comunicação é retornado imediamente para um servidor causando o desbloqueio do cliente que executou a operação de *Binding*;

2. não há *Binding* pendente - neste caso o comando *AcceptBind* bloqueia o servidor até que surja algum pedido de *Binding*.

No capítulo seguinte veremos os principais aspectos de implementação da base de objetos.

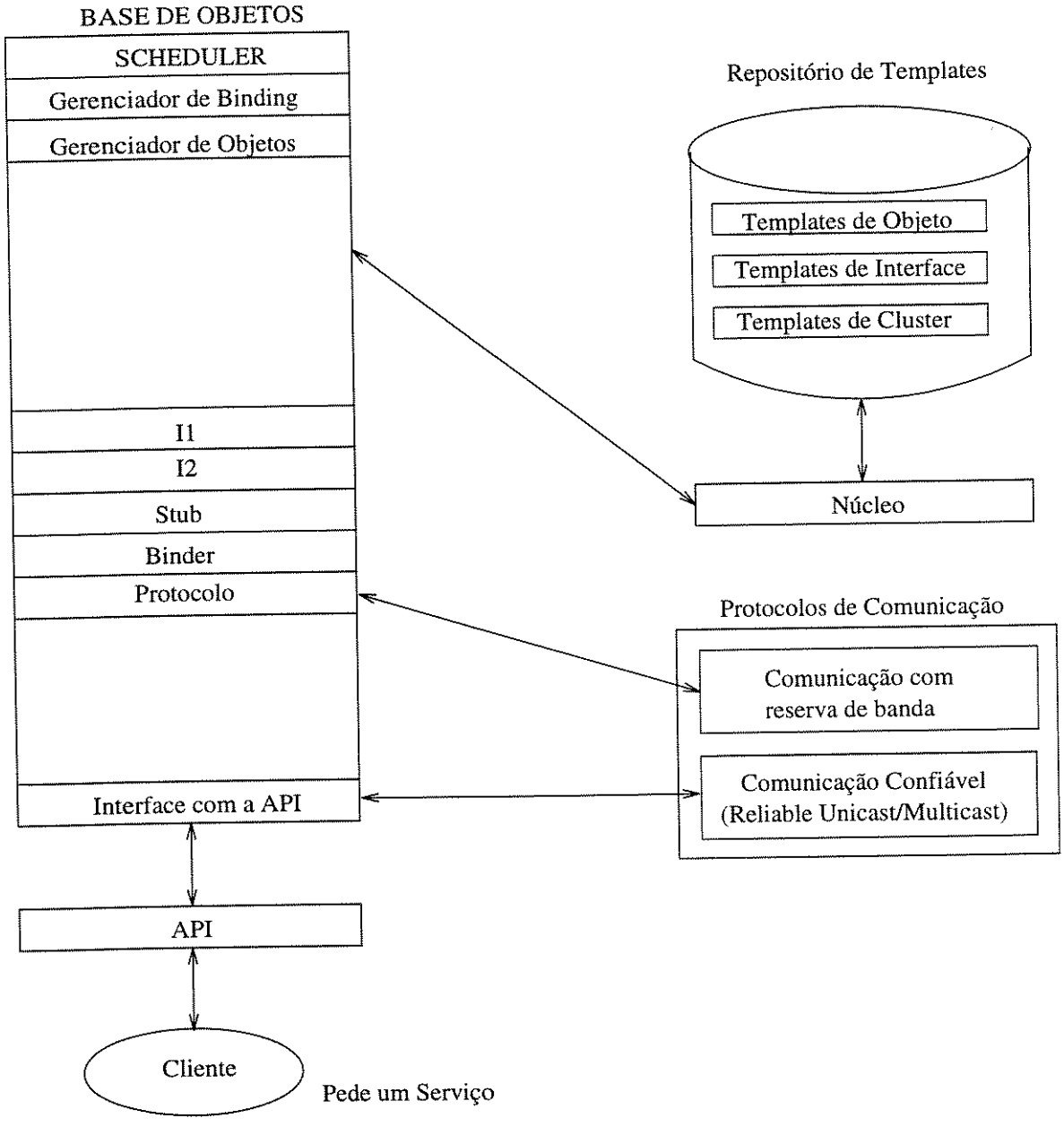


Figura 5.2: Componentes do sistema de suporte à execução da plataforma ODP

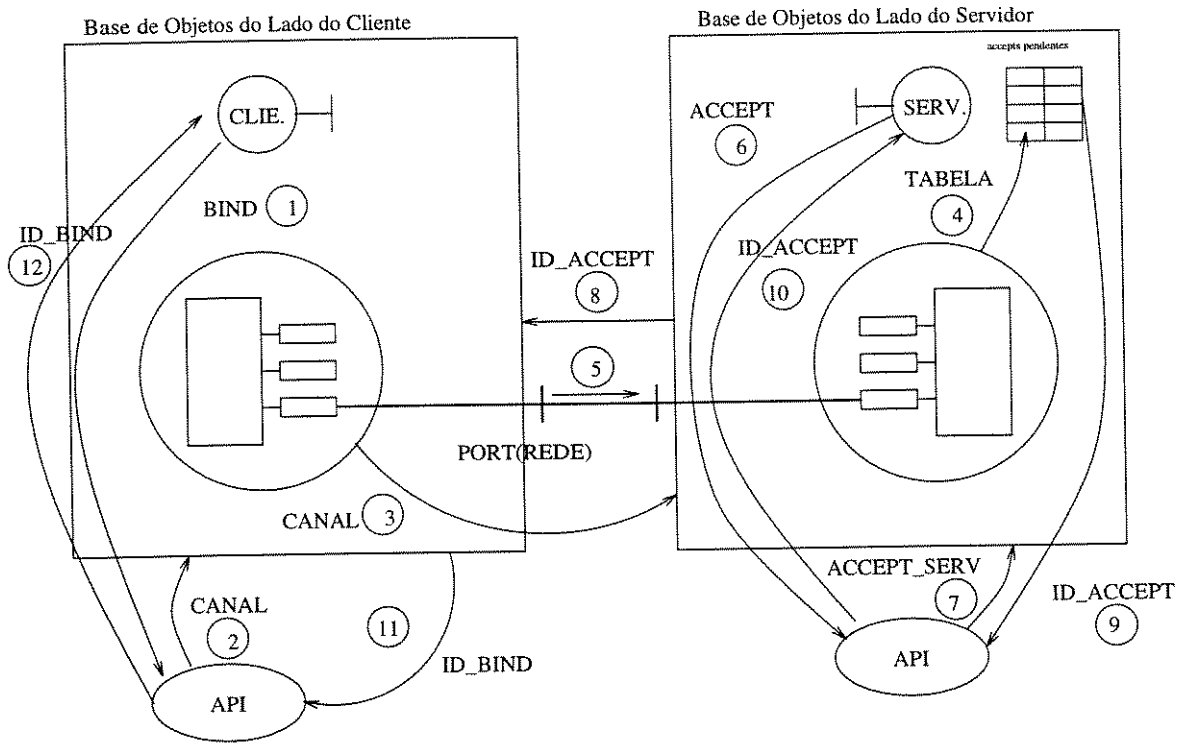


Figura 5.3: Sequência de eventos para o estabelecimento de um canal

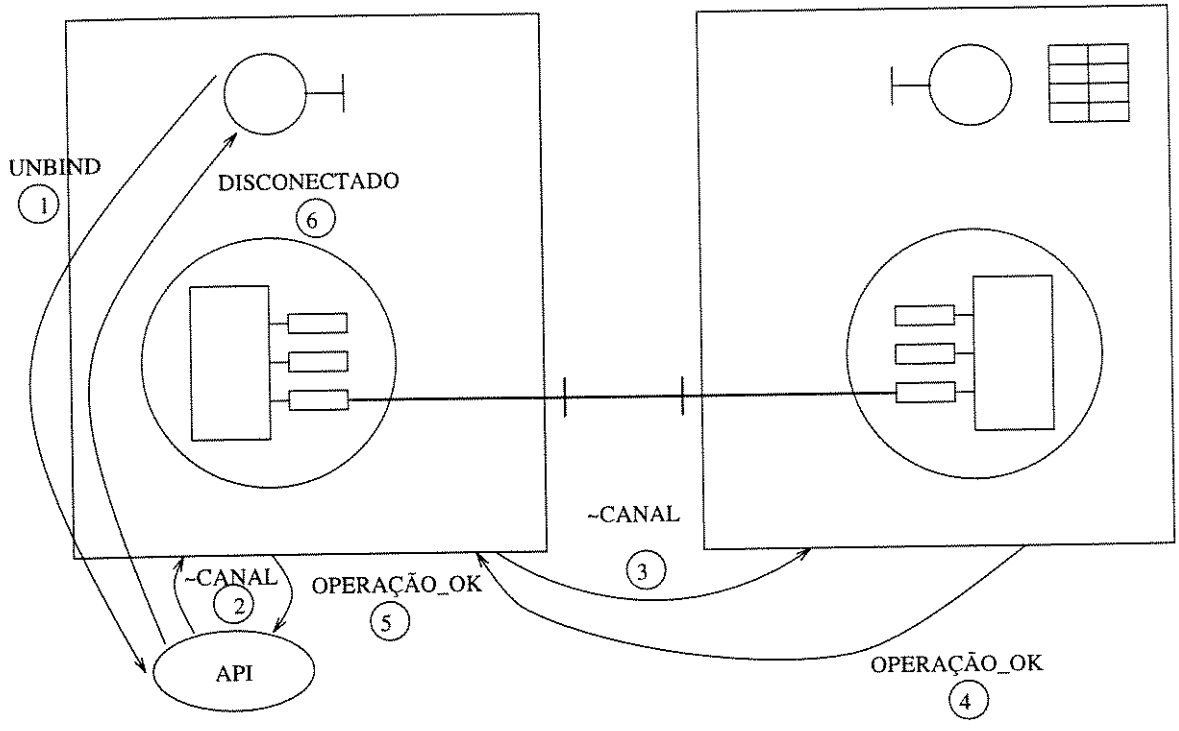


Figura 5.4: Sequência de eventos para a destruição de um canal

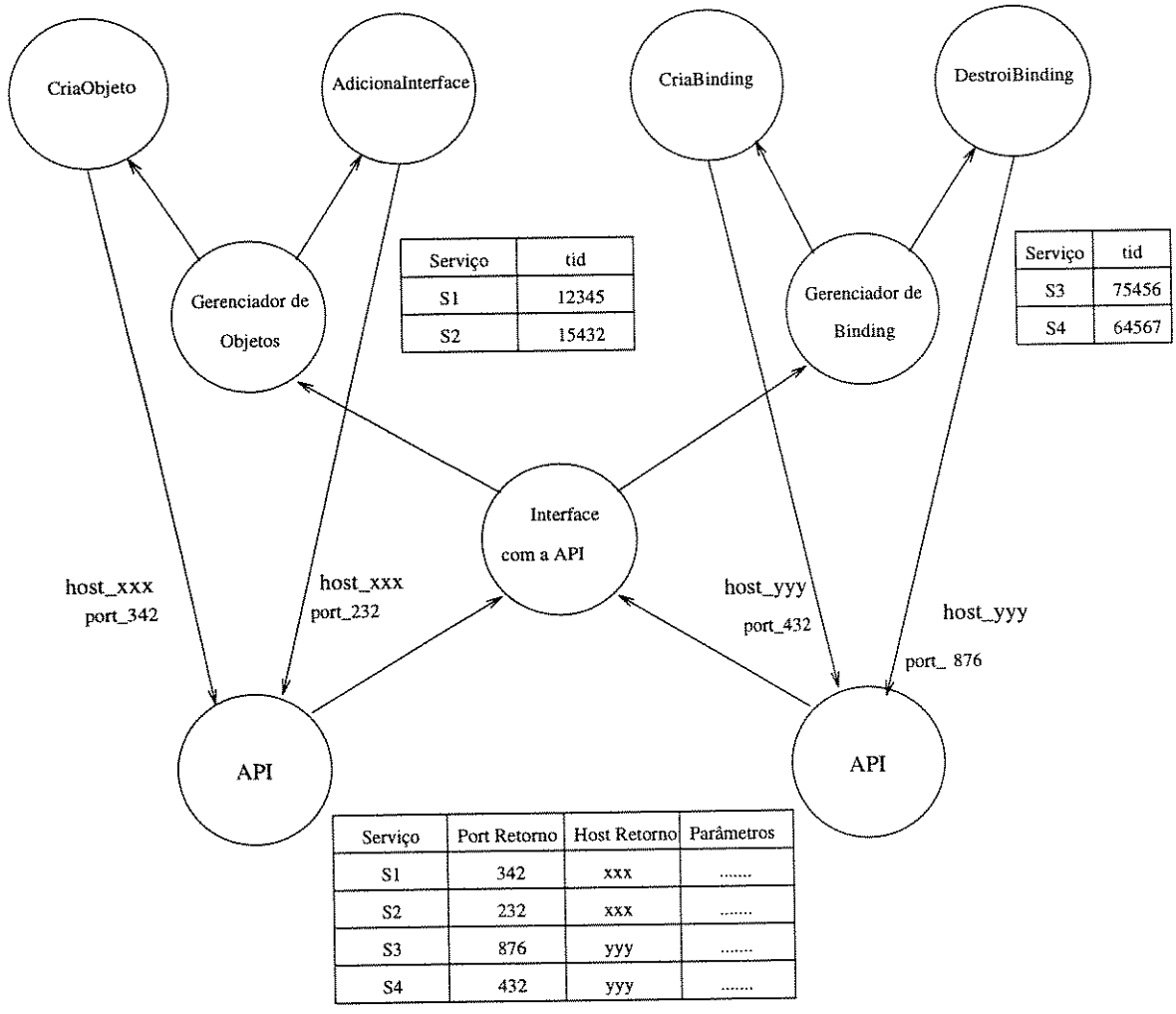


Figura 5.5: Interação entre API, interface com a API, e gerenciadores de objetos e *binding*.

Capítulo 6

Implementação

6.1 Introdução

Conforme citado no capítulo anterior agora serão apresentados os aspectos mais relevantes a respeito da implementação da base de objetos.

O serviço de objetos (ou base de objetos) é suportado basicamente por quatro classes de objetos que foram denominadas de INTERFACE, OBJETO, *CLUSTER* e CANAL. Os relacionamentos, atributos e os métodos de cada classe são mostrados na figura 6.1. A figura mostra uma notação utilizada pela técnica de modelamento de objetos (OMT: *Object Modeling Technique*) proposta por Rumbaugh [50, 49]. As classes INTERFACE, OBJETO e *CLUSTER* serão suportadas pelo gerenciador de objetos enquanto a classe CANAL será suportada pelo gerenciador de *binding*. O algoritmo de cada método e como os métodos são chamados através de *threads* são mostrados em detalhes num relatório interno [19].

6.2 Vetor de Clusters, Objetos, Interfaces e Canais

Os vetores de *clusters*, objetos, interfaces e canais armazenam os objetos da Base de Objetos que implementam estas abstrações do modelo de referência ODP. Cada vetor desses é mantido pela Base de Objetos. A seguir será mostrado a estrutura de dados de cada elemento de um vetor de *clusters*, a forma da declaração deste vetor e as estruturas auxiliares que foram utilizadas para a manipulação deste vetor.

Cada elemento de um vetor de *clusters* é uma estrutura de dados declarada da seguinte forma:

```
typedef struct elem_cluster{
LOCK locking;
char *host;
PORT porta;
void *parametros;
CLUSTER *obj_cluster;
SEMAPHORE *s;
} ELEM_CLUSTER;
```

onde:

- locking: indica se este elemento do vetor de clusters está sendo ocupado ou não (campo `obj_cluster` está endereçando ou não algum objeto que implementa um *cluster*). Assume os valores FREE ou BUSY;
- host: endereça o nome do *host* da qual veio um pedido de um serviço que faz com que seja executado algum método do objeto que implementa o *cluster* e para a qual deve ser enviado uma resposta após a execução do serviço (método do objeto que implementa *cluster*);
- porta: porta de comunicação para a qual deve ser enviado uma resposta após ter sido executado algum serviço (método do objeto que implementa *cluster*);
- parametros: apontador para os parâmetros do serviço (parâmetros de algum método do objeto que implementa *cluster*);
- `obj_cluster`: apontador para o objeto que implementa as abstrações de um *cluster* do modelo de referência ODP;
- s: apontador para um semáforo que protege este elemento do vetor de *clusters*. Basicamente é utilizado para acesso concorrente ao campo locking, o que impede que o mesmo `obj_cluster` seja utilizado para endereçar dois objetos diferentes.

O vetor de *clusters* é declarado de seguinte forma:

```
ELEM_CLUSTER v_cluster[DIM_LIST_CLUSTERS];
```

A cada vez que um *cluster* é criado é buscado uma posição que esteja livre no vetor de *cluster* (campo locking com valor FREE). A busca é circular, ou seja chegando se ao final do vetor começa a busca novamente de seu início, se uma posição livre ainda não foi achada. Antes de analisar esta variável é necessário verificar se o semáforo associado com o elemento desta posição do vetor de *cluster* permite o acesso. A busca da posição livre no vetor de *cluster* é feita com o auxílio de uma variável global (`ult_cluster`) que mostra qual a última posição do vetor de *clusters* que foi utilizada. Esta variável também está protegida por um semáforo. Com isto tenta-se acessar a próxima posição após `ult_cluster` e verifica-se se esta posição está livre. A variável `ult_cluster` se mostra muito útil se considerarmos que cada *cluster* tenha uma média de vida semelhante (ou seja os *clusters* que foram criados primeiro serão destruídos primeiro também) com isto sempre a posição seguinte do vetor após `ult_cluster` estará provavelmente livre. No caso em que cada *cluster* tenha uma média de vida totalmente diferente um do outro esta variável não auxiliará muito na busca de uma posição livre.

Todo objeto *cluster* possui um identificador global que é gerado pelo núcleo e este número é de conhecimento de quem necessita operar sobre este objeto. O identificador global é uma associação um-para-um com a posição do elemento do vetor de *clusters* na qual está localizado o objeto que implementa o *cluster*. Toda requisição para este *cluster* terá como parâmetro o seu identificador global, ficando a cargo da Base de Objetos (mais especificamente do gerenciador de objetos) buscar a posição no vetor de *clusters* na qual este *cluster* se encontra. De forma a agilizar este processo de busca foi criada uma tabela de *hashing* fazendo a associação do identificador global do *cluster* com a posição no vetor de *clusters*. O acesso a esta tabela de *hashing* também foi protegido por um semáforo. A

função de *hashing* utilizada foi o resto da divisão do identificador global do *cluster* pelo tamanho da tabela de *hashing*. Uma boa função de *hashing* dá um bom espalhamento na tabela de *hashing* e a função escolhida têm esta propriedade.

Os vetores de objetos, interfaces e canais possuem estruturas semelhantes. O gerenciador de objetos manipula os vetores de *clusters*, objetos e interfaces enquanto que o gerenciador de *bindings* manipula o vetor de canais.

6.3 Listas Associando Identificador de uma Thread com Identificador Global de Cluster, Objeto, Interface ou Canal

Nestas listas cada elemento é composto de um identificador de uma *thread* e um identificador global de *cluster*, objeto, interface ou canal e servem para que o *behaviour* de cada uma dessas abstrações do modelo de referência ODP possa obter sua própria identificação. O *behaviour* é sempre executado como uma *thread*. No momento em que um *cluster*, objeto, interface ou canal é instanciado, seus respectivos *behaviours* são colocados em execução. Após o início da execução é retornado um identificador de *thread* que é armazenado juntamente com o identificador global de *cluster*, objeto, interface ou canal na sua respectiva lista. O *behaviour* de um *cluster*, objeto, interface ou canal que desejar fazer uma operação sobre si mesmo deve obter o seu identificador de *thread* e com ele solicitar o identificador global de *cluster*, objeto, interface ou canal e de posse de um destes identificadores fazer uma requisição de serviço. Para acessar os identificadores globais foram implementadas as quatro seguintes funções:

```
unsigned long InWhichClusterIam(THREAD);
unsigned long InWhichObjectIam(THREAD);
unsigned long InWhichInterfaceIam(THREAD);
unsigned long InWhichChannelIam(THREAD);
```

Estas listas são implementadas globalmente na base de objetos e estas funções fazem a busca diretamente nestas listas. A execução dos serviços também poderia ser feita a princípio diretamente, pois as listas de *clusters*, objetos, interfaces e canais citados na seção anterior se encontram no mesmo espaço de endereçamento aonde está sendo executado o *behaviour*, mas por questões de compatibilidade de chamadas foi decidido que elas seriam feitas com a intermediação da API. Todas as listas são protegidas por semáforos. As listas que contém o identificador de uma *thread* com o identificador global de *cluster*, objeto, interface ou canal são:

```
genList L_THREAD_CLUSTERS;
genList L_THREAD_OBJETOS;
genList L_THREAD_INTERFACES;
genList L_THREAD_CANAIS;
```

6.4 Semáforos

O núcleo fornece uma classe semáforo binário que deve ser instanciado passando como parâmetro um identificador de semáforo. Para que dois ou mais processos tenham acesso

a uma aréa de exclusão mútua comum a ambos basta que utilizem o mesmo identificador de semáforo. Este identificador geralmente é definido estaticamente num programa. Porém, quando duas ou mais *threads* de um mesmo processo necessitam acesso a uma área de exclusão mútua comum apenas no mesmo espaço de endereçamento do processo, tal esquema não pode ser utilizado pois uma nova instância do mesmo processo pode ser ativado criando as mesmas instâncias de *threads* utilizando os mesmos semáforos. O semáforo utilizado por uma *thread* de um processo pode bloquear o espaço de endereçamento de outra *thread* de outro processo e tal fato não pode ocorrer. Para resolver tal problema foi adotado a seguinte solução:

- cada região crítica tem associado um número local ao *host*, que é gerado pelo núcleo durante a fase de inicialização da base de objetos. Este número fica armazenado numa variável global ao processo;
- toda vez que uma *thread* deseja acessar uma região crítica, a mesma instancia um semáforo com o número local (identificador) do semáforo associado com esta região e o utiliza para entrar ou aguardar o acesso a região crítica.

Com isto, o acesso a uma região crítica semelhante mas de processos diferentes, terão um número local diferente e não ocorrerá conflitos. Este semáforo só é utilizado para processos numa mesma máquina, não é um semáforo distribuído.

6.5 Behaviours

O *behaviour* define as ações que são executadas por alguma abstração do modelo de referência ODP (cápsula, *cluster*, objeto, interface e canal). Estas ações dependem da implementação de cada sistema para uma uma aplicação em particular. O *behaviour* portanto é escrito pelo usuário da aplicação. No sistema o *behaviour* nada mais é do que uma função que é executada como uma *thread*. Uma limitação de executar uma função como uma *thread* é o fato dela só poder aceitar um único parâmetro inteiro. A definição da função deve ser colocada no arquivo `ob_lib1.cc` junto com a diretiva `extern` e logo depois como sendo um dos parâmetros da função `defina_thread` que deve ser um dos comandos da função `ob_inicio()`, função esta que é ativada durante a inicialização da Base de Objetos. O arquivo `ob_lib1.o` (gerado após a compilação do arquivo fonte `ob_lib1.cc`) juntamente com o(s) arquivo(s) objeto que contém as definições de todos os *behaviours* que a Base de Objetos pode colocar em execução devem ser unidos ao sistema durante a fase de ligação dos programas.

A função `defina_thread` adiciona um novo elemento numa tabela que associa o nome simbólico de uma função com o apontador de uma função. Este nome simbólico é o mesmo que está presente no campo *behaviour* dos templates de cápsula, *cluster*, objeto, interface e canal. Foi criado uma tabela de *hashing* que associa o nome simbólico junto com o índice da tabela de *threads*. A função de *hashing* utilizada foi a soma do código ASCII do primeiro e do último caracter do nome simbólico dividido pelo tamanho da tabela de *hashing*. Portanto, para executar um *behaviour* como uma *thread* basta passar o nome simbólico do *behaviour* e fazer uma busca na tabela de *thread* a partir desse nome, obtendo-se o apontador de função para o respectivo *behaviour* e passá-lo como parâmetro para uma função que execute uma função como uma *thread*.

Um exemplo de um arquivo `ob_lib1.cc` está mostrado abaixo:

```
#include ''ob_lib1.hpp''
#include ''base_objetos.hpp''

.
.
.
extern behaviour_objeto_0(int);
.
.
.
.


ob_inicio()
{
.
.
.
define_thread(''behaviour_objeto_0'',behaviour_objeto_0);
.
.
}

defina_thread(char *nome, int(*thread)(int))
{
.
.
.
.
}
```

Alguns *behaviours* já estão pré-definidos pela Base de Objetos, como por exemplo os *behaviours* dos elementos que compõem um canal de comunicação (*stub, binder* e *protocol*).

6.6 Padrão XDR

O padrão XDR(*eXternal Data Representation*)[57] permite a transmissão de dados entre diferentes tipos de máquina. A abordagem do XDR é transmitir os dados numa representação canônica. Um programa rodando em um tipo de máquina converte os dados da representado local para o formato no padrão XDR e os envia para uma máquina de outro tipo. Esta outra máquina lê os dados no formato do padrão XDR e os desconverte para os dados da sua representação local. O tempo gasto para converter e desconverter os dados do formato canônico é insignificante principalmente em aplicações em rede, onde o tempo de transmissão é muito maior. O padrão XDR foi utilizado na transmissão de pedidos de serviços da API [40] para a Base de Objetos e nos *behaviours* dos *stubs* que são os elementos responsáveis pela conversão/desconversão de dados para a transmissão

num canal do modelo de referência ODP. A especificação do padrão XDR está disponível publicamente podendo ser programado para qualquer tipo de máquina.

6.7 Threads

Threads podem ser considerados mini-processos dentro de um processo comum. Elas compartilham a CPU, podem criar *threads* filhos e ficam bloqueadas quando estão esperando por uma chamada de sistema. Enquanto uma *thread* está bloqueada, uma outra pode entrar em execução. A única diferença entre um processo comum e uma *thread*, é que esta última compartilha o mesmo espaço de endereçamento dentro de um processo. A vantagem disso é que a troca de contexto devido a uma ação de escalonamento é muito mais rápida utilizando-se *threads*. A desvantagem desse mecanismo é a necessidade de um maior controle por parte do implementador. As *threads* também permitem que sejam implementadas um mecanismo de escalonamento interno ao processo, otimizando o tempo de execução das *threads*, quando um processo está executando na CPU. Como toda cápsula, *cluster*, objeto e interface possui um *behaviour*, e cada *behaviour* possui um código que é executado logo após sua ativação optou-se por adotar o mecanismo de *threads* para execução dos *behaviours*. Os próprios serviços que são fornecidos pela Base de Objetos são executadas independentemente criando-se dinamicamente *threads* que processam esses serviços. A interface com a API, o escalonador, o gerenciador de objetos e o de *binding* também são *threads*.

6.8 Armazenamento de Templates

Os *templates* foram armazenados como se fossem registros de um banco de dados. A chave de busca é o nome do *template*. O gerenciador de banco de dados utilizado foi o GNU dbm (gdbm) [17] desenvolvido pela *Free Software Foundation*. As funções do gdbm fazem o uso de tabelas de *hashing* para uma busca eficiente dos registros.

O gdbm armazena o par chave/dado num determinado arquivo. Cada chave é única ou seja só é associado com um item de dado. A unidade básica de dados no gdbm é a estrutura:

```
typedef struct{
char *dptr;
int dsize;
}datum;
```

Esta estrutura permite um tamanho arbitrário para chaves e items de dados. Neste tipo de banco de dados não há desperdício de disco como em alguns bancos de dados tradicionais que necessitam definir o tamanho dos campos de um registro. Porém apresenta a desvantagem de deixar a cargo do implementador a serialização e a desserialização dos campos de um registro de dado.

O gbdm apresenta uma característica muito importante num banco de dados num sistema distribuído que é permitir que um mesmo arquivo dados possa ser aberto por múltiplas aplicações. Quando uma aplicação abre um banco de dados gdbm, este é designado como sendo de leitura ou escrita. A banco de dados pode ser aberto no máximo

uma vez para escrita. Entretanto, pode ser aberto simultaneamente diversas vezes para leitura. O banco de dados não pode ser aberto para escrita e leitura ao mesmo tempo.

Porém, este gerenciador de banco de dados não apresenta uma característica muito desejável que é o processamento de transações. Com um serviço de transação é possível fazer a replicação de dados. O núcleo do sistema [6] ficou responsável por fornecer esta facilidade e com isto foi necessário que o núcleo intermediasse as chamadas as funções ao banco de dados gdbm. O formato básico das chamadas ao banco de dados gdbm não foi alterado.

6.9 Linguagem C++

A linguagem ANSI C++ [56] foi escolhida para o desenvolvimento do projeto devido a sua portabilidade e a sua característica de ser uma linguagem de programação orientada a objetos, o que facilitou a transposição do projeto para a implementação, pois toda estruturação do modelo de referência ODP é baseada nos conceitos de orientação a objetos. A implementação de uma estrutura do modelo de referência ODP com uma linguagem não orientada a objetos também seria possível, porém muito difícil de ser executada.

6.10 Inicialização da Interface com a API, Gerenciador de Binding e Gerenciador de Objetos

A base de objetos é um programa executável que é disparado numa máquina pela API passando como parâmetros o nome de um *template* de cápsula e um número global fornecido pelo núcleo. Inicialmente o programa atribui o nome do *template* de cápsula a uma variável global da base de objetos, são feitos alguns procedimentos iniciais e por fim são instanciados as *threads* interface com a API, gerenciador de *binding* e gerenciador de objetos tendo como parâmetro o número global passado inicialmente ao programa.

6.10.1 Inicialização da Thread Interface com a API

A interface com a API abre uma conexão *reliable unicast* [35] pela qual são recebidos os pedidos de serviços da Base de Objetos. Logo após abrir uma conexão o identificador da porta de comunicação mais o *host* da máquina que fornece o serviço juntamente com o nome da Base de Objetos (que é o nome de uma cápsula, presente no template de cápsula e que está armazenado numa váriavel global) concatenado com a caracter '|' mais o identificador que foi passado como parâmetro são armazenados num *portmapper*. O *portmapper* diz em qual *host* e *port* pode ser obtido algum serviço. Esta entrada é aguardada pela API para fazer requisições a esta base de objetos. Depois disto é buscado no mesmo *portmapper* qual o *port* do gerenciador de binding e do gerenciador de objetos. A busca desses serviços é feita procurando no *portmapper* o nome do gerenciador '`gerenciador_binding`' (e depois '`gerenciador_objetos`') concatenado com a caracter '|' mais o identificador que foi passado como parâmetro. Isto impede que a interface com a API tente se comunicar com o gerenciador de objetos ou o gerenciador de *binding* de outra Base de Objetos. Feito isto a base de objetos aguarda a chegada dos serviços da API e os envia para o gerenciador de objetos ou o gerenciador de *binding*.

6.10.2 Inicialização da Thread Gerenciador de Binding

O gerenciador de *binding* abre uma conexão *reliable unicast* pela qual pode ser recebidos os pedidos de serviços vindos da interface com a API. Logo após abrir uma conexão o identificador da porta de comunicação mais o *host* da máquina que fornece o serviço juntamente com o nome '`gerenciador_binding`' concatenado com o caracter '|' mais o identificador (número global) que foi passado como parâmetro são armazenados num *portmapper*. Depois disso o gerenciador de *binding* fica aguardando a chegada de algum serviço da interface com a API e quando chega um pedido o mesmo é identificado, criando-se uma *thread* adequada para tratar o serviço.

6.10.3 Inicialização da Thread Gerenciador de Objetos

O gerenciador de objetos abre uma conexão *reliable unicast* pela qual pode ser recebidos os pedidos de serviços vindos da interface com a API. Logo após abrir uma conexão o identificador da porta de comunicação mais o *host* da máquina que fornece o serviço juntamente com o nome '`gerenciador_objetos`' concatenado com o caracter '|' mais o identificador (número global) que foi passado como parâmetro são armazenados num *portmapper*. Depois disso o gerenciador de objetos fica aguardando a chegada de algum serviço da interface com a API e quando chega um pedido o mesmo é identificado, criando-se uma *thread* adequada para tratar o serviço.

6.11 Criação de Canais

A criação de um canal envolve uma série de interações onde foi utilizado o mapeador de *ports* (*portmapping*) e algumas listas. Existem dois níveis diferentes de interação. Um nível da Base de Objetos e o outro correspondente aos *protocols* do canal. A figura 6.2 mostra todos os eventos que ocorrem ao longo do tempo para a criação de um canal. Toda a sequência de eventos esta detalhada nas próximas subseções.

6.11.1 Evento 1

A API do lado do cliente envia um pedido de criação de canal para a Base de Objetos do lado do cliente. Este pedido tem como um de seus pârametros o nome do serviço remoto, os nomes dos templates de canal do lado do cliente e do lado do servidor dentre outros.

6.11.2 Evento 2

A Base de Objetos do lado do cliente instância um objeto canal do lado do cliente utilizando-se dos dados presentes no template de canal do lado do cliente e obtém o *host* e o *port* da cápsula que fornece o serviço através do *portmapping* criando uma entrada numa lista (`L_OBJETO_BIND_OR_ACCEPT`), onde cada elemento desta lista é composto por:

- identificador do objeto canal do lado do cliente (`id_bind_object`);
- nome do *host* aonde está Base de Objetos que fornece o serviço;
- *port* da Base de Objetos que fornece o serviço;

- nome do serviço remoto.

Na lista acima foi implementada uma função de busca tendo como chave o identificador do objeto canal. Este elemento da lista será lido pelo componente *protocol* do lado do cliente.

6.11.3 Evento 3

A Base de Objetos do lado do cliente pede a criação de um objeto canal para a Base de Objetos do lado do servidor tendo como parâmetros o nome do serviço remoto, o nome do template de canal do lado do servidor dentre outros.

6.11.4 Evento 4

A Base de Objetos do lado do servidor instância um objeto canal do lado do servidor utilizando-se dos dados presentes no template de canal do lado do servidor criando uma entrada numa lista (`L_OBJETO_BIND_OR_ACCEPT`), onde cada elemento desta lista é composto por:

- identificador do objeto canal do lado do servidor (`id_accept_object`);
- nome do *host* aonde está Base de Objetos que fornece o serviço;
- *port* da Base de Objetos que fornece o serviço;
- nome do serviço.

Este elemento da lista será lido pelo componente *protocol* do lado do servidor.

6.11.5 Evento 5

O *protocol* do lado do servidor através do identificador do objeto canal (`id_accept_object`) (que este têm acesso), procura na lista `L_OBJETO_BIND_OR_ACCEPT` qual o *host* e o *port* da Base de Objetos que fornece o serviço, além do próprio nome do serviço (Evento 4). O *protocol* do lado do servidor, de posse desses dados, cria uma porta de conexão e insere uma entrada no *portmapping* da seguinte forma:

- nome do serviço: concatenação do caracter '*', nome do serviço, caracter '|',*host* da Base de Objetos que fornece o serviço, caracter '|' e *port* da Base de Objetos que fornece o serviço;
- *host*: *host* aonde está localizado o *protocol* do lado do servidor;
- *port*: *port* do *protocol* do lado do servidor.

Esta entrada será lida pelo *protocol* do lado do cliente.

6.11.6 Evento 6

O *protocol* do lado do cliente através do identificador do objeto canal (`id_bind_object`) (que este têm acesso), procura na lista `L_OBJETO_BIND_OR_ACCEPT` qual o *host* e o *port* da Base de Objetos que fornece o serviço, além do próprio nome do serviço (Evento 2). De posse desses parâmetros procura no *portmapping* um serviço com o seguinte formato: concatenação do caracter '*', nome do serviço, caracter '|',*host* da Base de Objetos que fornece o serviço, caracter '|' e *port* da Base de Objetos que fornece o serviço. Obtendo o correspondente *host* e *port* do *protocol* do lado do servidor (Evento 5). Com isto é direcionado um pedido de conexão para este *protocol* do lado do servidor.

6.11.7 Evento 7

O *protocol* do lado do servidor cria uma entrada numa lista com serviços pendentes (`L_SERVICOS`), onde cada elemento desta lista é composto por:

- nome do serviço;
- identificador do objeto canal do lado do servidor (`id_accept_object`).

Na lista acima foi implementada uma função de busca tendo como chave o nome do serviço. Este elemento da lista será lido pela Base de Objetos do lado do servidor.

6.11.8 Evento 8

A API do lado do servidor envia um pedido aceitando determinado serviço (accept), tendo como parâmetros o nome do serviço dentre outros.

6.11.9 Evento 9

A Base de Objetos do lado do servidor após receber um pedido de aceitação da API e verificar que esta se encontra na lista `L_SERVICOS`, este envia o seu identificador do objeto canal para o cliente (`id_accept_object`).

6.11.10 Evento 10

A Base de Objetos do lado do servidor envia o identificador do objeto canal (`id_accept_object`) criado do seu lado para a API.

6.11.11 Evento 11

A Base de Objetos do lado do cliente aguarda o identificador do objeto canal do lado do servidor (`id_accept_object`) que é uma indicação também de que o canal foi criado, só então envia o identificador do objeto canal (`id_bind_object`) criado do seu lado para a API.

6.12 Interação no Canal

Um objeto canal possui uma lista na qual este recebe os dados vindos da interface *stub* e por sua vez as interfaces também também possuem uma lista na qual recebem os dados vindos de outras interfaces. Um comando `SUBMIT` para um canal coloca uma dado diretamente na lista de dados da interface *stub* enquanto o comando `DELIVER` retira um dado da lista de dados do objeto canal. Um comando `QUEUE` executado por uma interface coloca o dado diretamente na lista de dados de outra interface. Existem duas variações do comando `QUEUE`: uma quando os dados são enviados para o *protocol* do outro lado da conexão e outro quando o dado é enviado para a lista de dados de um objeto canal. O comando `POP` por sua vez retira um dado da lista de dados de interface. Existe uma variação do comando `POP` que "retira" um dado da *interface protocol* do outro lado da conexão. A lista de dados do objeto canal foi denominada `dados_canal` e a lista de dados da interface de `dados_interface`. A fig. 6.3 mostra o fluxo de dados numa comunicação bidirecional num canal. A sequência dos `QUEUEs` e `POPs` numa interface de um canal estão implementadas no *behaviour* de cada uma dessas interfaces. Os comandos `SUBMIT` e `DELIVER` são utilizados para o envio e recebimento de informações através de um canal sendo os comandos acessíveis aos usuários do canal. Os comandos `QUEUE` e `POP` para interação entre as interfaces do canal são transparentes ao usuário final.

6.13 Destruição do Canal

A destruição de um canal sempre parte do cliente que solicitou sua criação. A Base de Objetos do lado do cliente mantém uma lista onde cada elemento armazena o identificador do objeto canal do lado do cliente (`id_bind_object`), identificador do objeto canal do lado do servidor (`id_accept_object`), o nome do *host* da Base de Objetos do lado do servidor e o *port* da Base de Objetos do lado do servidor. Quando chega um pedido de destruição de um determinado canal (identificado através do `id_bind_object`) é feita uma busca na lista e obtém-se o identificador do objeto canal lado do servidor, o *host* e o *port* da Base de Objetos do lado do servidor. A Base de Objetos do lado do cliente destrói a metade do canal de seu lado, enquanto é enviado um pedido para que a Base de Objetos do lado do servidor para que destrua a sua metade do canal.

No capítulo seguinte veremos o modo como a base de objetos deve ser ativada pela API e a descrição de 4 serviços da mesma.

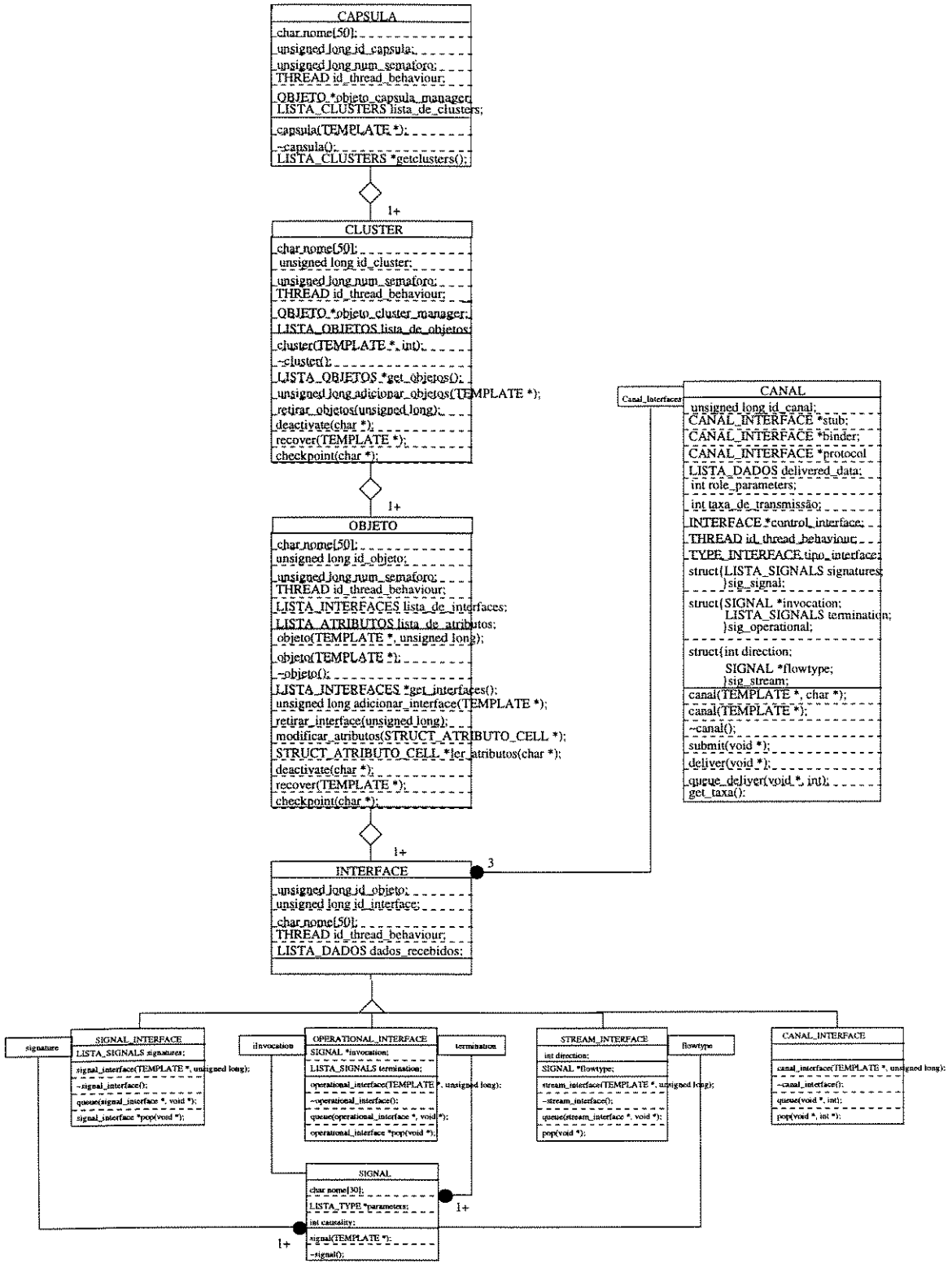


Figura 6.1: Classes da base de objetos

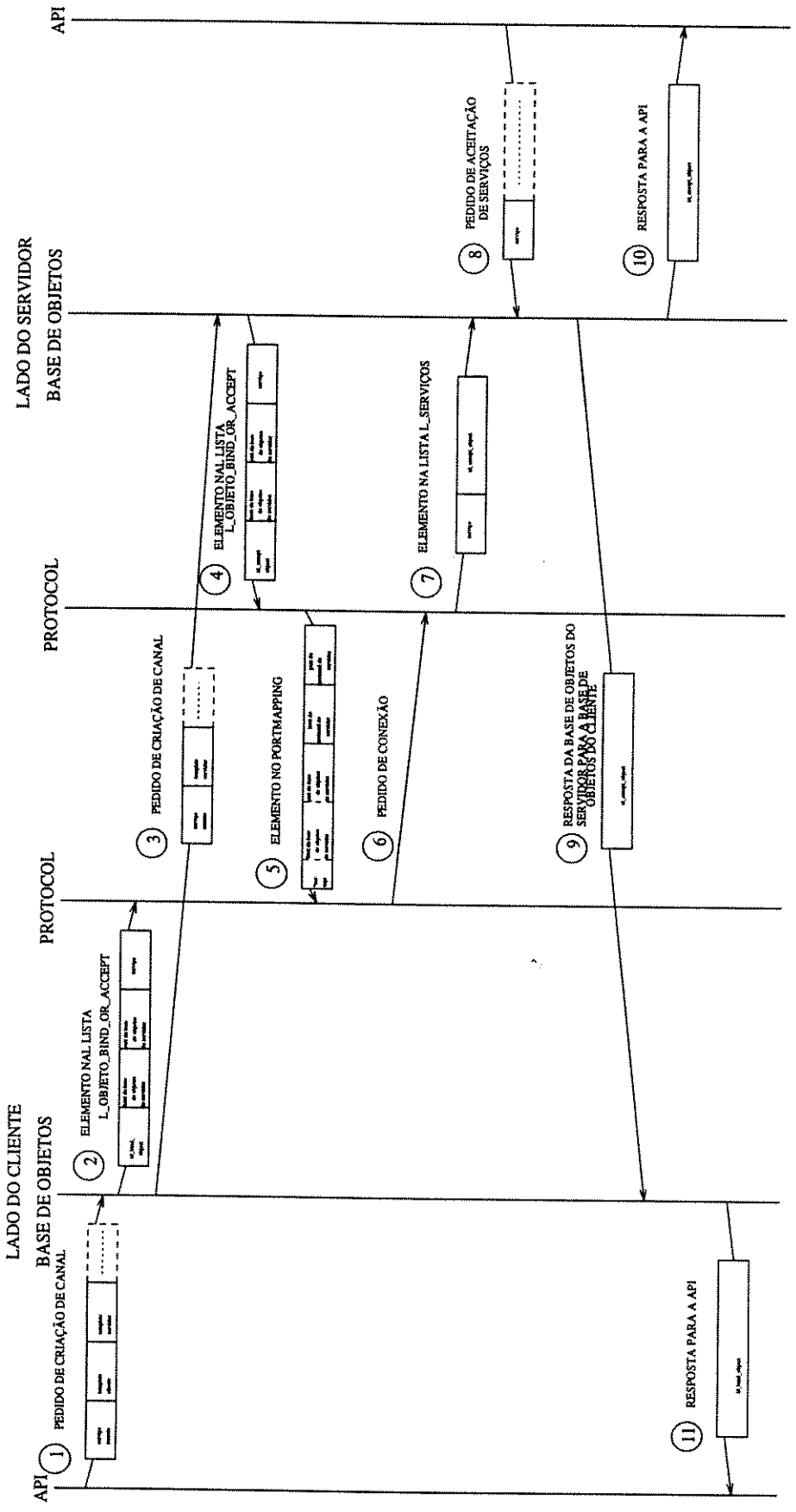


Figura 6.2: Eventos para criação de um canal

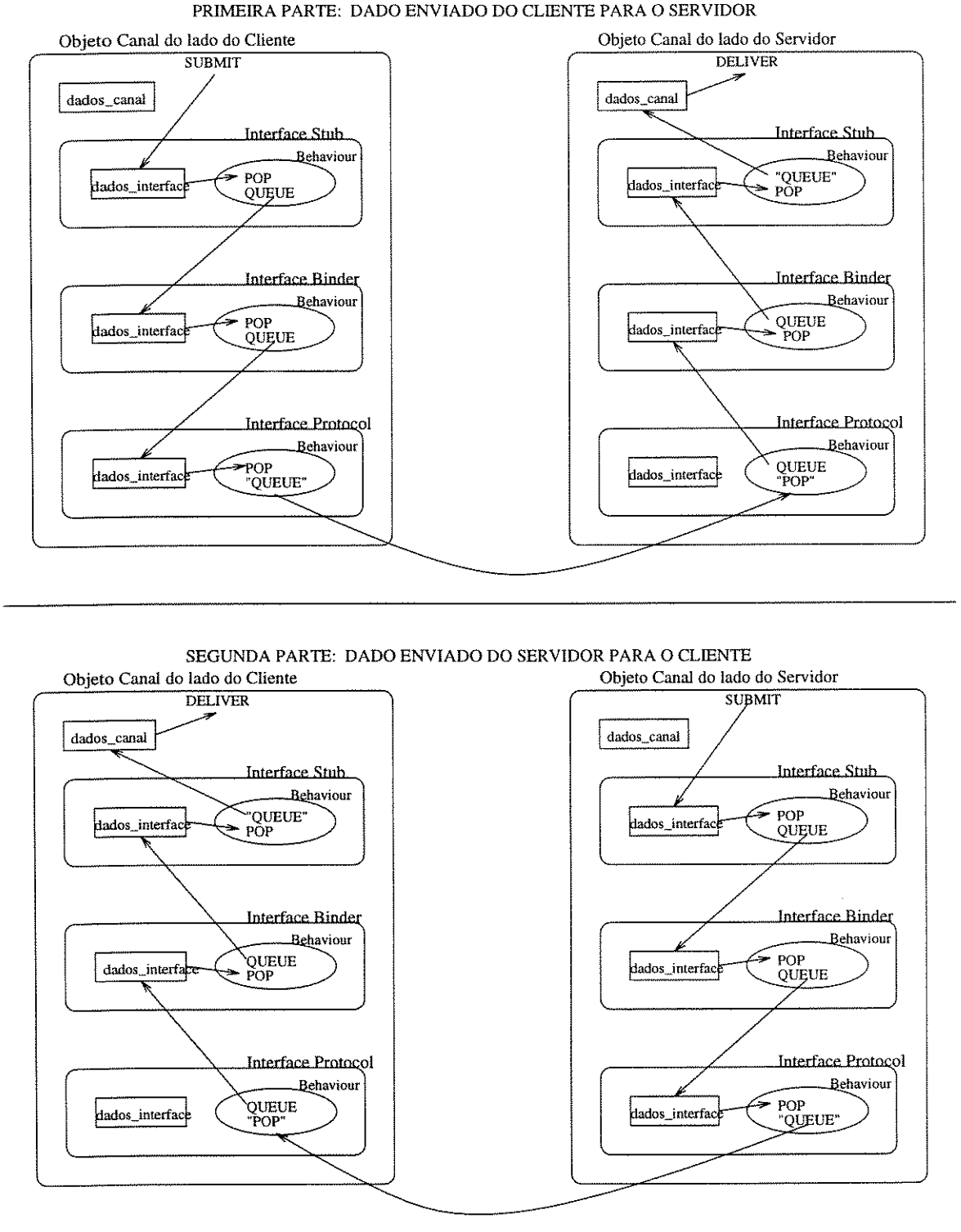


Figura 6.3: Fluxo de dados numa comunicação bidirecional

Capítulo 7

Exemplos de Utilização

7.1 Introdução

Conforme citado no capítulo anterior descreveremos o modo como a Base de Objetos deve ser ativado pela API e a descrição do que ocorre quando 4 tipos de serviços são solicitados a Base de Objetos. Uma descrição dos demais serviços se encontra em relatório interno [19].

7.2 Ativação da Base de Objetos

A Base de Objetos corresponde a abstração de uma cápsula do modelo de referência ODP. Portanto ela deve possuir um *template* e ser ativada como um processo numa determinada máquina de uma rede. A API deve ativar uma Base de Objetos numa determinada máquina passando como parâmetros o nome de uma *template* de cápsula, junto com um identificador global para esta Base de Objetos. Ao ser ativada, a Base de Objetos insere no *portmapping* o nome do serviço como sendo o nome do *template* de cápsula concatenado com o caracter '|' e um identificador global, além do nome do *host* e o *port*, como sendo o *host* e o *port* pelos quais a Base de Objetos recebe requisições de serviços da API que requisitou sua criação.

7.3 Criação de um Cluster

Para a criação de um *cluster* inicialmente o usuário deve enviar seu pedido com seus respectivos parâmetros à API. A API por sua vez converterá os parâmetros do pedido para o formato XDR numa determinada ordem e os enviará a uma determinada Base de Objetos (cápsula) na qual deseja que o *cluster* seja criado. A ordem dos parâmetros com seus respectivos tipos para a criação de um *cluster* é dada abaixo:

- identificação do pedido (int);
- *host* para envio de resposta (char *);
- *port* para envio de resposta (int);
- nome do *template* do *cluster* (char *).

Quando estes dados chegam até a *thread* interface com a API, esta desconverte do formato XDR somente a identificação do pedido e caso se trate de um pedido de criação de *cluster* o redireciona para a *thread* gerenciador de objetos. O gerenciador de objetos desconverte todos os parâmetros do formato XDR e busca uma posição livre no vetor de *clusters* da Base de Objetos verificando se o campo *locking* está livre (FREE). Achado uma posição livre o campo *locking* é alterado para BUSY. No campo *host* é colocado o nome do *host* para envio de resposta, no campo porta o *port* para envio de resposta e por fim no campo parametros o nome do *template* do *cluster*. Feito isto é criado uma *thread* especialmente para tratar este pedido. A *thread* tem como parâmetro a posição do vetor de *clusters* que foi encontrada livre. A *thread* cria uma instância do *cluster* nesta posição do vetor de *clusters* e atribui para o apontador `obj_cluster`. É retornado o identificador do *cluster* criado (convertido para o formato XDR), para o *host* e a porta especificados nesta posição do vetor de cluster (endereço de retorno para a API). Todas as operações posteriores sobre este *cluster* (*checkpoint*, desativação, recuperação, etc) devem ser feitas passando como um dos parâmetros este identificador.

7.4 Desativação de um Cluster

Para a desativação de um *cluster* inicialmente o usuário deve enviar seu pedido com seus respectivos parâmetros à API. A API por sua vez converterá os parâmetros do pedido para o formato XDR numa determinada ordem e os enviará a uma determinada Base de Objetos (cápsula) na qual deseja que o *cluster* seja desativado. A ordem dos parâmetros com seus respectivos tipos para a desativação de um *cluster* é dada abaixo:

- identificação do pedido (int);
- *host* para envio de resposta (char *);
- *port* para envio de resposta (int);
- identificador de *cluster* (unsigned long);
- nome do *template* do *cluster* (char *).

Quando estes dados chegam até a *thread* interface com a API, esta desconverte do formato XDR somente a identificação do pedido, caso se trate de um pedido de desativação de *cluster* o redireciona para a *thread* gerenciador de objetos. O gerenciador de objetos desconverte todos os parâmetros do formato XDR e busca a posição do vetor de *clusters* correspondente ao identificador do *cluster*. Esta busca é auxiliada por uma tabela de *hashing* que associa a identificador de *cluster* com a posição do vetor. No campo *host* é colocado o nome do *host* para envio de resposta, no campo porta o *port* para envio de resposta e por fim no campo parametros o nome do *template* do *cluster* (será gerado um *checkpoint template*). Feito isto é criado uma *thread* especialmente para tratar este pedido tendo como parâmetro a posição do vetor de *clusters*. A *thread* chama o método que desativa o *cluster* da instância do objeto que implementa um *cluster* nesta posição do vetor de *clusters*. É retornado uma indicação de SUCESSO ou FRACASSO (antes convertido para o formato XDR) para o *host* e a porta especificados nesta posição do vetor de *clusters* (endereço de retorno para a API).

7.5 Criação de um Canal

Para a criação de um canal inicialmente o usuário deve enviar seu pedido com seus respectivos parâmetros à API. A API por sua vez converterá os parâmetros do pedido para o formato XDR numa determinada ordem e os enviará a uma determinada Base de Objetos (cápsula) na qual deseja que o canal seja criado. A ordem dos parâmetros com seus respectivos tipos para a criação de um canal são dados abaixo:

- identificação do pedido (int);
- *host* para envio de resposta (char *);
- *port* para envio de resposta (int);
- nome do serviço remoto (char *);
- *timeout* para criação do canal (int);
- nome do *template* do lado do cliente (char *);
- nome do *template* do lado do servidor (char *).

Quando estes dados chegam até a *thread* interface com a API, esta desconverte do formato XDR somente a identificação do pedido e caso se trate de um pedido de criação de canal o redireciona para a *thread* gerenciador de *binding*. O gerenciador de *binding* desconverte todos os parâmetros do formato XDR e busca uma posição livre no vetor de canais da Base de Objetos verificando se o campo *locking* está livre (FREE). Achado uma posição livre o campo *locking* é alterado para BUSY. No campo *host* é colocado o nome do *host* para envio de resposta, no campo porta o *port* para envio de resposta e por fim no campo parametros o nome do serviço remoto, o *timeout*, o nome do template do lado do cliente e o nome do template do lado do servidor. Feito isto é criado uma *thread* especialmente para tratar este pedido tendo como parâmetro a posição do vetor de canais que foi encontrado livre. A *thread* cria uma instância de um objeto canal nesta posição do vetor de canais e atribui para o apontador `obj_canal`. O objeto canal por sua vez instância internamente as *interfaces stub*,*binder* e *protocol*. É retornado o identificador do canal (antes convertido para o formato XDR) criado, para o *host* e a porta especificados nesta posição do vetor de canais (endereço de retorno para a API). Todas as operações posteriores sobre este canal (envio de dados, recebimento de dados, destruição, etc) devem ser feitas passando como um dos parâmetros este identificador.

7.6 Envio de Dados por um Canal do Tipo Signal

Para o envio de dados por um canal inicialmente o usuário deve enviar seu pedido com seus respectivos parâmetros à API. A API por sua vez converterá os parâmetros do pedido para o formato XDR numa determinada ordem e os enviará a uma determinada Base de Objetos (cápsula) na qual deseja que o canal seja criado. A ordem dos parâmetros com seus respectivos tipos para o envio de dados por um canal é dada abaixo:

- identificação do pedido (int);

- *host* para envio de resposta (char *);
- *port* para envio de resposta (int);
- identificador de canal (unsigned long);
- nome do sinal (char *);
- número de parâmetros (int);
- tipo do sinal (int);
- valor do sinal (char, int, float ou double);
- :
- :
- tipo do sinal (int);
- valor do sinal (char, int, float ou double).

Quando estes dados chegam até a *thread* interface com a API, esta desconverte do formato XDR somente a identificação do pedido e caso se trate de um pedido de envio de dados por um canal o redireciona para a *thread* gerenciador de *binding*. O gerenciador de *binding* desconverte todos os parâmetros do formato XDR e busca a posição do vetor de canais correspondente ao identificador do canal. Esta busca é auxiliada por uma tabela de *hashing* que associa a identificador de canal com a posição do vetor. No campo *host* é colocado o nome do *host* para envio de resposta, no campo porta o *port* para envio de resposta e por fim no campo parametros os dados que serão enviados são todos serializados. Feito isto é criado uma *thread* especialmente para tratar este pedido tendo como parâmetro a posição do vetor de canal. A *thread* chama o método que envia o dado pelo canal (`SUBMIT`) da instância do objeto que implementa um objeto canal nesta posição do vetor de canais. É retornado uma indicação de SUCESSO ou FRACASSO (antes convertido para o formato XDR) para o *host* e a porta especificados nesta posição do vetor de canal (endereço de retorno para a API).

No capítulo seguinte veremos a conclusão e os trabalhos futuros.

Capítulo 8

Conclusão

8.1 Introdução

Este capítulo apresenta a conclusão e os trabalhos futuros. Este capítulo foi dividido em 5 seções. A próxima seção destaca os pontos positivos e negativos de um sistema desenvolvido seguindo o modelo de referência ODP, segundo os critérios de comparação de sistemas distribuídos adotado por Chin [10]. A seção seguinte mostra a importância dos pontos de vista do modelo de referência ODP; a próxima seção comenta dificuldade de mapeamento de uma linguagem de engenharia para uma linguagem de tecnológica; a penúltima seção apresenta as dificuldades encontradas na implementação do trabalho e por fim a última seção propõe trabalhos futuros.

8.2 Aspectos Positivos e Negativos do modelo de referência ODP

As opiniões sobre os aspectos positivos e negativos do modelo de referência ODP são baseados na experiência de programação e estudos de problemas num ambiente distribuído do autor desta tese.

O modelo de referência ODP abrange praticamente todos os aspectos necessários para caracterizar um sistema distribuído como aberto. Em confronto com outros sistemas baseado/orientado a objetos [10], um sistema desenvolvido segundo o modelo de referência ODP se mostra superior a eles num contexto geral, segundo certos critérios de comparação adotados para este tipos de sistemas.

O modelo de referência ODP sugere um sistema orientado a objetos com herança múltipla. Este tipo de sistema permite uma grande flexibilidade na reutilização de código pois um determinado objeto pode herdar características de múltiplos objetos pré-definidos e os objetos do sistema são ativados dinamicamente o que diminui o tempo de resposta do sistema (aspectos positivos). Porém, todo objeto do sistema possui pelo menos granularidade média por mais simples que seja, pois existe um *overhead* considerável para manter este objeto (aspecto negativo).

Em relação ao gerenciamento de ações o modelo de referência ODP prevê que uma ação de nível mais alto coordena as sub-ações de nível mais baixo; o sistema de segurança pode ser implementado das mais variadas formas dependendo exclusivamente do nível de importância do objeto e, no aspecto replicação do objeto, todas as cópias são consideradas

iguais (aspectos positivos). Porém, apresenta um esquema pessimista o que diminui a concorrência dentro do sistema e a recuperação de um objeto é feita sempre através de um sistema *roll-back* onde todas as ações que estavam em progresso quando uma falha ocorreu são canceladas (aspectos negativos).

Quanto ao mecanismos de gerenciamento da interação de objetos a localização de um objeto é feita com o auxílio de uma entidade chamada *trader* que inclusive possui formas de negociação, podendo interagir com outros *traders* em busca de um determinado objeto que forneça um serviço, tudo isto de maneira transparente ao usuário. No sistema do modelo de referência ODP os objetos podem mudar de localização (migração). Quanto a manipulação de falhas de invocação, tanto o cliente quanto o servidor possuem objetos que as identificam e acionam procedimentos para tratá-las (aspectos positivos).

Por fim, quanto ao gerenciamento de recursos o modelo de referência ODP apresenta a possibilidade de migração dos objetos, balanceando melhor a carga de processamento do sistema (aspecto positivo). Porém, tanto a representação em memória quanto em disco armazena somente a última versão do objeto (aspecto negativo).

8.3 Pontos de Vista do modelo de referência ODP

Ao lidar com um grande sistema, qualquer que seja ele, é praticamente impossível manipulá-lo sem criar diferentes níveis de abstração para tratar a complexidade do problema. O modelo de referência ODP prevê este tipo de abordagem que constitui em separar um sistema em partes gerenciáveis, hierarquicamente separadas, cada qual focalizando um ponto de vista diferente. O modelo ODP pode com isto ser utilizado e compreendido por toda uma hieraquia de desenvolvimento, envolvendo desde aspectos gerais definidos pelos gerentes de um projeto, até as ferramentas básicas que serão utilizadas para a implementação do sistema, ou seja, ele é diferente dos modelos tradicionais de referência que só são compreendidos por analistas e implementadores que estejam intimamente ligados ao projeto. O modelo mostra desde como um sistema deve ser interpretado do ponto de vista empresarial até o ponto de vista tecnológico, passando pelos pontos de vista de informação, computacional e de engenharia. Uma das contribuições deste trabalho foi mostrar alguns requisitos necessários do ponto de vista tecnológico (que ainda não estavam bem definidos no início deste projeto pelo modelo de referência ODP) para a implementação de certos aspectos do ponto de vista de engenharia, mostrando que são possíveis de serem desenvolvidos com as ferramentas disponíveis atualmente. O ponto de vista de computação é mapeado no ponto de vista de engenharia.

Uma linguagem computacional é utilizada para especificar os requisitos do ponto de vista computacional, assim como a linguagem de engenharia e a tecnológica para os requisitos de ponto de vista de engenharia e tecnológica respectivamente. A linguagem computacional deve ser utilizada pelos usuários que estão implementando algum sistema sobre uma plataforma ODP. As linguagens de engenharia e a tecnológica devem ser transparentes para estes usuários.

Este trabalho foi focalizado no ponto de vista de engenharia do modelo de referência ODP.

8.4 Mapeamento da Linguagem de Engenharia para Tecnológica

Este trabalho mostrou que é possível implementar as abstrações do modelo de referência ODP da linguagem de engenharia (cápsula, *cluster*, objeto, *interface* e canal). Um dos aspectos mais difíceis do trabalho, e onde foi concentrado grande parte dos esforços foi justamente na compreensão da linguagem de engenharia do modelo de referência ODP, só depois disso é que foi definido as ferramentas e a forma de implementá-las. Algumas decisões tecnológicas como a utilização da linguagem C++ para o desenvolvimento do sistema, o padrão XDR (que possui um especificação bem detalhada) para a implementação dos objetos *stubs* de um canal e a execução dos *behaviours* de cápsula, *cluster*, objeto, *interface* e canal como *threads* e toda cápsula como um processo pesado poderiam ter sido sugeridas na linguagem tecnológica do modelo de referência ODP. Mesmo o banco de dados gdbm da GNU poderia ter sido sugerido na linguagem tecnológica, para o caso de não se ter um gerenciador de base de dados mais sofisticado, pois trata-se de um software de domínio público, cujo código fonte está escrito em linguagem C, podendo ser compilado em várias plataformas diferentes.

8.5 Implementação do Trabalho

A implementação de uma infraestrutura para o suporte a objetos ativos em tempo de execução exigiu um bom domínio de conceitos de sistemas operacionais tais como *threads*, formas de aumentar a concorrência, utilização de semáforos, protocolos de comunicação, dentre outros assuntos. Sem este trabalho ficaria a cargo do usuário o estudo dos conceitos mencionados acima e a aplicação deles a própria implementação do sistema. Para o usuário o trabalho diminui o custo de desenvolvimento de sistemas distribuídos.

8.6 Trabalhos Futuros

Atualmente a API está se comunicando diretamente com o provedor do serviço de objetos, tendo ela que saber anteriormente a localização deste provedor. Numa próxima etapa haverá a intermediação da plataforma ORBIX [21]. Esta plataforma ficará responsável pela ligação da API com o provedor de serviço (Base de Objetos).

Sincronização de fluxos de dados multímidia, negociação de serviços, desenvolvimento de um escalonador mais sofisticado são outras características que poderão ser adicionadas ao sistema.

Aspectos mais sofisticados como migração de objetos e segurança, dentre outros, necessitam de um estudo mais aprofundado mas também são possíveis de serem incorporados à plataforma.

Apêndice A

Serviços Oferecidos a API pela Base de Objetos

A base de facilidades para objetos distribuídos oferece uma série de serviços para a API. Estes serviços estão relacionados a Cluster, Objeto, Interface e Canal. A seguir indicaremos os serviços oferecidos e os formatos dos parâmteros que devem ser passados para a execução destes serviços e os formatos das respostas.

- **Pedidos relacionados a Cápsula**
  - obter a lista dos Clusters que compõem a Cápsula;
  - adicionar um Cluster na Cápsula;
  - retirar um Cluster da Cápsula.
- **Pedidos relacionados a um Cluster**
  - obter a lista de Objetos que compõem um Cluster;
  - adicionar um Objeto a um Cluster;
  - retirar um Objeto de um Cluster;
  - fazer checkpoint de um Cluster;
  - desativar um Cluster;
  - remover um Cluster;
  - recuperar um Cluster.
- **Pedidos relacionados a um Objeto**
  - adicionar uma Interface a um Objeto;
  - remover uma Interface de um Objeto;
  - obter a lista de Interfaces que compõem um Objeto;
  - ler um atributo de um Objeto;
  - modificar um atributo de um Objeto;
  - fazer checkpoint de um Objeto;
  - desativar um Objeto;

- remover um Objeto;
- recuper um Objeto.

- **Pedidos Relacionados a uma Interface**
  - enviar um dado para uma Interface Signal(Queue);
  - receber um dado de uma Interface Signal(Pop);
  - enviar um dado para uma Interface Operational(Queue);
  - receber um dado de uma Interface Operational(Pop);
  - enviar um dado para uma Interface Signal(Queue);
  - receber um dado de uma Interface Signal(Pop).

- **Pedidos relacionados a um Canal**
  - criar Canal pelo cliente ou consumidor (Bind);
  - aceitar pedidos de conexão do cliente ou consumidor (Accept);
  - destruição de um Canal pelo cliente ou consumidor; (Unbind);
  - enviar um dado para um Canal Signal(Submit);
  - receber um dado de um Canal Signal(Deliver);
  - enviar um dado para um Canal Operational(Submit);
  - receber um dado de um Canal Operational(Deliver);
  - enviar um dado para um Canal Stream(Submit);
  - receber um dado de um Canal Stream(Deliver).

A.1 Formatação dos pedidos e respostas relacionados a Cápsula

A.1.1 Obter a lista de Clusters que compõem a Cápsula

Parâmetros de entrada:

- identificação do pedido (int);
- nome do host para envio de resposta (char *);
- porta de comunicação para envio de resposta (int).

Parâmetro de resposta:

- número de Clusters (int);
- nome dos Clusters que compõem a Cápsula (char *).

OBS: Cada nome de um Cluster é uma composição do nome do Cluster presente num template de Cluster e o identificador deste Cluster; O separador é o caracter |;

A.1.2 Adicionar um Cluster na Cápsula

Parâmetros de entrada:

- identificação do pedido (int);
- nome do host para envio de resposta (char *);
- porta de comunicação para envio de resposta (int);
- nome de um template de um Cluster (char *).

Parâmetro de resposta:

- identificador do Cluster criado (unsigned long).

A.1.3 Retirar um Cluster da Cápsula

Parâmetros de entrada:

- identificação do pedido (int);
- nome do host para envio de resposta (char *);
- porta de comunicação para envio de resposta (int);
- identificador de um Cluster (unsigned long).

Parâmetro de resposta:

- indicação de SUCESSO ou FRACASSO (int).

A.2 Formatação dos pedidos e respostas relacionados a um Cluster

A.2.1 Obter a lista de Objetos que compõem um Cluster

Parâmetros de entrada:

- identificação do pedido (int);
- nome do host para envio de resposta (char *);
- porta de comunicação para envio de resposta (int);
- identificador de um Cluster (unsigned long).

Parâmetro de resposta:

- número de objetos (int);
- nome dos Objetos que compõem o Cluster (char *).

OBS: Cada nome de um Objeto é uma composição do nome do Objeto presente num template de Objeto e o identificador deste Objeto; O separador é o caracter |;

A.2.2 Adicionar um Objeto a um Cluster

Parâmetros de entrada:

- identificação do pedido (int);
- nome do host para envio de resposta (char *);
- porta de comunicação para envio de resposta (int);
- identificador de um Cluster (unsigned long);
- nome do template do Objeto a ser adicionado (char *).

Parâmetro de resposta:

- identificador do Objeto Adicionado (unsigned long).

A.2.3 Retirar um Objeto de um Cluster

Parâmetros de entrada:

- identificação do pedido (int);
- nome do host para envio de resposta (char *);
- porta de comunicação para envio de resposta (int);
- identificador de um Cluster (unsigned long);
- identificador do Objeto a ser retirado (unsigned long).

Parâmetro de resposta:

- indicação de SUCESSO ou FRACASSO (int).

A.2.4 Checkpoint de um Cluster

Parâmetros de entrada:

- identificação do pedido (int);
- nome do host para envio de resposta (char *);
- porta de comunicação para envio de resposta (int);
- identificador de um Cluster (unsigned long);
- nome de um template de Cluster (char *).

Parâmetro de resposta:

- indicação de SUCESSO ou FRACASSO (int).

A.2.5 Desativação de um Cluster

Parâmetros de entrada:

- identificação do pedido (int);
- nome do host para envio de resposta (char *);
- porta de comunicação para envio de resposta (int);
- identificador de um Cluster (unsigned long);
- nome de um template de Cluster (char *).

Parâmetro de resposta:

- indicação de SUCESSO ou FRACASSO (int).

A.2.6 Remover um Cluster

Parâmetros de entrada:

- identificação do pedido (int);
- nome do host para envio de resposta (char *);
- porta de comunicação para envio de resposta (int);
- identificador de um Cluster (unsigned long).

Parâmetro de resposta:

- indicação de SUCESSO ou FRACASSO (int).

A.2.7 Recuperação de um Cluster

Parâmetros de entrada:

- identificação do pedido (int);
- nome do host para envio de resposta (char *);
- porta de comunicação para envio de resposta (int);
- identificador de um Cluster (unsigned long);
- nome de um template de um Cluster (char *).

Parâmetro de resposta:

- indicação de SUCESSO ou FRACASSO (int).

A.3 Formatação dos pedidos e respostas relacionados a um Objeto

A.3.1 Adicionar uma Interface a um Objeto

Parâmetros de entrada:

- identificação do pedido (int);
- nome do host para envio de resposta (char *);
- porta de comunicação para envio de resposta (int);
- identificador de um Objeto (unsigned long);
- nome do template da Interface a ser adicionado (char *).

Parâmetro de resposta:

- identificador da Interface adicionado (unsigned long).

OBS: O primeiro valor presente num template de Interface é o tipo de sua Interface (Signal, Operational ou Stream);

A.3.2 Retirar uma Interface de um Objeto

Parâmetros de entrada:

- identificação do pedido (int);
- nome do host para envio de resposta (char *);
- porta de comunicação para envio de resposta (int);
- identificador de um Objeto (unsigned long);
- identificador da Interface a ser retirada (unsigned long).

Parâmetro de resposta:

- indicação de SUCESSO ou FRACASSO (int).

A.3.3 Obter a lista de Interfaces que compõem um Objeto

Parâmetros de entrada:

- identificação do pedido (int);
- nome do host para envio de resposta (char *);
- porta de comunicação para envio de resposta (int);
- identificador de um Objeto (unsigned long).

Parâmetro de resposta:

- número de interfaces (int);
- nome das intefaces que compõem o Objeto (char *).

OBS: Cada nome de um Interface é uma composição do nome da Interface presente num template de Interface e o identificador desta Interface; O separador é o caracter |;

A.3.4 Ler um atributo de um Objeto

Parâmetros de entrada:

- identificação do pedido (int);
- nome do host para envio de resposta (char *);
- porta de comunicação para envio de resposta (int);
- identificador de um Objeto (unsigned long);
- nome do atributo (char *);
- tipo do atributo (int).

Parâmetro de resposta:

- valor do atributo (char, int, float ou double).

A.3.5 Modificar um atributo de um Objeto

Parâmetros de entrada:

- identificação do pedido (int);
- nome do host para envio de resposta (char *);
- porta de comunicação para envio de resposta (int);
- identificador de um Objeto (unsigned long);
- nome do atributo (char *);
- tipo do atributo (int);
- novo valor do atributo (char, int, float ou double).

Parâmetro de resposta:

- indicação de SUCESSO ou FRACASSO (int).

A.3.6 Checkpoint de um Objeto

Parâmetros de entrada:

- identificação do pedido (int);
- nome do host para envio de resposta (char *);
- porta de comunicação para envio de resposta (int);
- identificador de um Objeto (unsigned long);
- nome de um template de Objeto (char *).

Parâmetro de resposta:

- indicação de SUCESSO ou FRACASSO (int).

A.3.7 Desativação de um Objeto

Parâmetros de entrada:

- identificação do pedido (int);
- nome do host para envio de resposta (char *);
- porta de comunicação para envio de resposta (int);
- identificador de um Objeto (unsigned long);
- nome de um template de Objeto (char *).

Parâmetro de resposta:

- indicação de SUCESSO ou FRACASSO (int).

A.3.8 Remover um Objeto

Parâmetros de entrada:

- identificação do pedido (int);
- nome do host para envio de resposta (char *);
- porta de comunicação para envio de resposta (int);
- identificador de um Objeto (unsigned long).

Parâmetro de resposta:

- indicação de SUCESSO ou FRACASSO (int).

A.3.9 Recuperação de um Objeto

Parâmetros de entrada:

- identificação do pedido (int);
- nome do host para envio de resposta (char *);
- porta de comunicação para envio de resposta (int);
- identificador de um Objeto (unsigned long);
- nome de um template de um Objeto (char *).

Parâmetro de resposta:

- indicação de SUCESSO ou FRACASSO (int).

A.4 Formatação dos pedidos e respostas relacionados a uma Interface

A.4.1 Enviar dados para uma Interface Signal

Parâmetros de entrada:

- identificação do pedido (int);
- nome do host para envio de resposta (char *);
- porta de comunicação para envio de resposta (int);
- identificador da Interface origem (unsigned long);
- identificador da Interface destino (unsigned long);
- formatação dos dados a serem enviados:
  - nome do sinal (char *);
  - numero de parâmetros do sinal (int);
  - tipo do parâmetro-1 (int);
  - valor do parâmetro-1 (char, int, float ou double);
  - :
  - :
  - tipo do parâmetro-n (int);
  - valor do parâmetro-n (char, int, float ou double).

Parâmetro de resposta:

- indicação de SUCESSO ou FRACASSO (int).

A.4.2 Receber dados de uma Interface Signal

Parâmetros de entrada:

- identificação do pedido (int);
- nome do host para envio de resposta (char *);
- porta de comunicação para envio de resposta (int);
- identificador da Interface local (unsigned long).

Parâmetro de resposta:

- identificador da Interface que enviou o dado (unsigned long);
- formatação dos dados recebidos:
  - nome do sinal (char *);
  - numero de parâmetros do sinal (int);
  - tipo do parâmetro-1 (int);
  - valor do parâmetro-1 (char, int, float ou double);
  - :
  - :
  - tipo do parâmetro-n (int);
  - valor do parâmetro-n (char, int, float ou double).

A.4.3 Enviar dados para uma Interface Operational

Parâmetros de entrada:

- identificação do pedido (int);
- nome do host para envio de resposta (char *);
- porta de comunicação para envio de resposta (int);
- identificador da Interface origem (unsigned long);
- identificador da Interface destino (unsigned long);
- formatação dos dados a serem enviados:
  - nome do sinal (char *);
  - tipo da operação (int);
  - numero de parâmetros do sinal (int);
  - tipo do parâmetro-1 (int);
  - valor do parâmetro-1 (char, int, float ou double);
  - :

  - :
  - tipo do parâmetro-n (int);
  - valor do parâmetro-n (char, int, float ou double).

Parâmetro de resposta:

- indicação de SUCESSO ou FRACASSO (int).

A.4.4 Receber dados de uma Interface Operational

Parâmetros de entrada:

- identificação do pedido (int);
- nome do host para envio de resposta (char *);
- porta de comunicação para envio de resposta (int);
- identificador da Interface local (unsigned long).

Parâmetro de resposta:

- identificador da Interface que enviou o dado (unsigned long);
- formatação dos dados recebidos:
  - nome do sinal (char *);
  - tipo da operação (int);
  - numero de parâmetros do sinal (int);
  - tipo do parâmetro-1 (int);
  - valor do parâmetro-1 (char, int, float ou double);
  - :
  - :
  - tipo do parâmetro-n (int);
  - valor do parâmetro-n (char, int, float ou double).

A.4.5 Enviar dados para uma Interface Stream

Parâmetros de entrada:

- identificação do pedido (int);
- nome do host para envio de resposta (char *);
- porta de comunicação para envio de resposta (int);
- identificador da Interface destino (unsigned long);
- formatação dos dados a serem enviados:

- nome do sinal (char *);
- número de sinais (int);
- numero de parâmetros do sinal (int);
- tipo do parâmetro-1 do sinal-1 (int);
- valor do parâmetro-1 do sinal-1 (char, int, float ou double);
- tipo do parâmetro-2 do sinal-1 (int);
- valor do parâmetro-2 do sinal-1 (char, int, float ou double);
- :
- :
- tipo do parâmetro-n do sinal-1 (int);
- valor do parâmetro-n do sinal-1 (char, int, float ou double);
- :
- :
- :
- tipo do parâmetro-n do sinal-m (int);
- valor do parâmetro-n do sinal-m (char, int, float ou double).

Parâmetro de resposta:

- indicação de SUCESSO ou FRACASSO (int).

A.4.6 Receber dados de uma Interface Stream

Parâmetros de entrada:

- identificação do pedido (int);
- nome do host para envio de resposta (char *);
- porta de comunicação para envio de resposta (int);
- identificador da Interface local (unsigned long).

Parâmetro de resposta:

- formatação dos dados recebidos:
  - nome do sinal (char *);
  - número de sinais (int);
  - numero de parâmetros do sinal (int);
  - tipo do parâmetro-1 do sinal-1 (int);
  - valor do parâmetro-1 do sinal-1 (char, int, float ou double);
  - tipo do parâmetro-2 do sinal-1 (int);

- valor do parâmetro-2 do sinal-1 (char, int, float ou double);
- :
- :
- tipo do parâmetro-n do sinal-1 (int);
- valor do parâmetro-n do sinal-1 (char, int, float ou double);
- :
- :
- :
- tipo do parâmetro-n do sinal-m (int);
- valor do parâmetro-n do sinal-m (char, int, float ou double).

A.5 Formatação dos pedidos e respostas relacionados a um Canal

A.5.1 Criação da metade do Canal do lado do cliente ou consumidor (Bind)

Parâmetros de entrada:

- identificação do pedido (int);
- nome do host para envio de resposta (char *);
- porta de comunicação para envio de resposta (int);
- nome de um serviço remoto (char *);
- timeout (int);
- nome do Bind template (char *);
- nome do Accept template (char *).

Parâmetro de resposta:

- identificador de um Objeto de Bind (unsigned long).

A.5.2 Criação da metade do Canal do lado do servidor ou produtor (Accept)

Parâmetros de entrada:

- identificação do pedido (int);
- nome do host para envio de resposta (char *);
- porta de comunicação para envio de resposta (int);

- nome de um serviço (char *);
- timeout (int).

Parâmetro de resposta:

- identificador de um Objeto de Accept-Bind (unsigned long).

A.5.3 Destruição do Canal pelo cliente ou consumidor (Unbind)

Parâmetros de entrada:

- identificação do pedido (int);
- nome do host para envio de resposta (char *);
- porta de comunicação para envio de resposta (int);
- identificador do Objeto de Bind (unsigned long).

Parâmetro de resposta:

- indicação de SUCESSO ou FRACASSO (int).

A.5.4 Enviar um dado para um Canal Signal (Submit)

Parâmetros de entrada:

- identificação do pedido (int);
- nome do host para envio de resposta (char *);
- porta de comunicação para envio de resposta (int);
- identificador de um objeto Bind (unsigned long);
- formatação dos dados a serem enviados:
  - nome do sinal (char *);
  - numero de parâmetros do sinal (int);
  - tipo do parâmetro-1 (int);
  - valor do parâmetro-1 (char, int, float ou double);
  - :
  - :
  - tipo do parâmetro-n (int);
  - valor do parâmetro-n (char, int, float ou double).

Parâmetro de resposta:

- indicação de SUCESSO ou FRACASSO (int).

A.5.5 Receber um dado por um Canal Signal(Deliver)

Parâmetros de entrada:

- identificação do pedido (int);
- nome do host para envio de resposta (char *);
- porta de comunicação para envio de resposta (int);
- identificador de um Objeto Accept-Bind (unsigned long).

Parâmetro de resposta:

- formatação dos dados recebidos:
  - nome do sinal (char *);
  - numero de parâmetros do sinal (int);
  - tipo do parâmetro-1 (int);
  - valor do parâmetro-1 (char, int, float ou double);
  - :
  - :
  - tipo do parâmetro-n (int);
  - valor do parâmetro-n (char, int, float ou double).

A.5.6 Enviar um dado para um Canal Operational (Submit)

Parâmetros de entrada:

- identificação do pedido (int);
- nome do host para envio de resposta (char *);
- porta de comunicação para envio de resposta (int);
- identificador de um Objeto Bind (unsigned long);
- formatação dos dados a serem enviados:
  - nome do sinal (char *);
  - tipo da operação (int);
  - numero de parâmetros do sinal (int);
  - tipo do parâmetro-1 (int);
  - valor do parâmetro-1 (char, int, float ou double);
  - :
  - :
  - tipo do parâmetro-n (int);
  - valor do parâmetro-n (char, int, float ou double).

Parâmetro de resposta:

- indicação de SUCESSO ou FRACASSO (int).

A.5.7 Receber um dado por um Canal Operational(Deliver)

Parâmetros de entrada:

- identificação do pedido (int);
- nome do host para envio de resposta (char *);
- porta de comunicação para envio de resposta (int);
- identificador de um Objeto Accept-Bind (unsigned long).

Parâmetro de resposta:

- formatação dos dados recebidos:
  - nome do sinal (char *);
  - tipo da operação (int);
  - numero de parâmetros do sinal (int);
  - tipo do parâmetro-1 (int);
  - valor do parâmetro-1 (char, int, float ou double);
  - :
  - :
  - tipo do parâmetro-n (int);
  - valor do parâmetro-n (char, int, float ou double).

A.5.8 Enviar um dado para um Canal Stream (Submit)

Parâmetros de entrada:

- identificação do pedido (int);
- nome do host para envio de resposta (char *);
- porta de comunicação para envio de resposta (int);
- identificador de um Objeto Bind (unsigned long);
- formatação dos dados a serem enviados:
  - nome do sinal (char *);
  - número de sinais (int);
  - numero de parâmetros do sinal (int);
  - tipo do parâmetro-1 do sinal-1 (int);
  - valor do parâmetro-1 do sinal-1 (char, int, float ou double);
  - tipo do parâmetro-2 do sinal-1 (int);
  - valor do parâmetro-2 do sinal-1 (char, int, float ou double);

  - :
  - :
  - tipo do parâmetro-n do sinal-1 (int);
  - valor do parâmetro-n do sinal-1 (char, int, float ou double);
  - :
  - :
  - :
  - tipo do parâmetro-n do sinal-m (int);
  - valor do parâmetro-n do sinal-m (char, int, float ou double).

Parâmetro de resposta:

- indicação de SUCESSO ou FRACASSO (int).

A.5.9 Receber um dado por um Canal Stream (Deliver)

Parâmetros de entrada:

- identificação do pedido (int);
- nome do host para envio de resposta (char *);
- porta de comunicação para envio de resposta (int);
- identificador de um Objeto Accept-Bind (unsigned long).

Parâmetro de resposta:

- formatação dos dados recebidos:
  - nome do sinal (char *);
  - número de sinais (int);
  - numero de parâmetros do sinal (int);
  - tipo do parâmetro-1 do sinal-1 (int);
  - valor do parâmetro-1 do sinal-1 (char, int, float ou double);
  - tipo do parâmetro-2 do sinal-1 (int);
  - valor do parâmetro-2 do sinal-1 (char, int, float ou double);
  - :
  - :
  - tipo do parâmetro-n do sinal-1 (int);
  - valor do parâmetro-n do sinal-1 (char, int, float ou double);
  - :
  - :
  - :
  - tipo do parâmetro-n do sinal-m (int);
  - valor do parâmetro-n do sinal-m (char, int, float ou double).

Abstract

The fast growth of distributed processing has brought the necessity of standardizing the Open Distributed Processing (ODP). The reference Model of ODP proposes an architecture that supports distribution, internetworking and portability.

In the ODP model the basic processing unit is the object. An object is composed of atributes (state) and interfaces (methods). The objects can be grouped within a cluster and the clusters can be grouped within a capsule. The grouping of objects (clusters and capsules) makes the object management easier. Objects can interact between them using rules as defined in the ODP model.

The multiware plataform is a model that supports ODP. This model is useful to identify the several components and services of a ODP plataform.

The proposal of this work is to provide one of the multiware plataform services, namely the run-time support for the objects: the creation of objects (with theirs atributes and interfaces), clusters and capsule as well as the communication mechanism between objects.

Bibliografia

[1] Anido, Ricardo O. e Mendonça, Nabor C. *Using Extended Hierarchical Quorum Consensus to Control Replicated Data: From Traditional Voting to Logical Structures* Relatório Técnico DCC - IMECC - UNICAMP

[2] Bach, M. J. *The Design of the UNIX Operating System* Englewwod Cliffs, NJ: Prentice Hall, 1987

[3] Berets, James C.; Cherniack Natasha e Sands, M. Richard *Introduction to Cronus* BBN Systems and Technologies, 1993

[4] Booch, G. *Object Oriented Design with Appications* Benjamin Cummings,1991

[5] Bowen, Dyfed *Open distributed processing* Computer Networks and ISDN Systems 23 (1991) 195-201 North-Holland

[6] Cardozo, Eleri *Núcleo da Plataforma ODP* Relatório Interno DCA-FEE-UNICAMP, jan/95

[7] Cardozo, Eleri *Relatório referente a Plataforma Multiware* Projeto Temático FAPESP 92/3507-0 Abr/94

[8] Cardozo, Eleri *Relatório referente a Plataforma Multiware* Projeto Temático FAPESP 92/3507-0 Abr/95

[9] Campbell, Andrew; Coulson, Geoff; García, Francisco; Hutchison, David e Leopond, Helmut *Integrated Quality od Service for Multimedia Communications* IEEE INFOCOM 93, San Francisco, March 1993

[10] Chin, Roger S. e Chanson, Samuel T. *Distributed Object-Based Programming Systems* ACM Computing Surveys, Vol. 23, No.1, Março 1991

[11] Cockshot, W. P.; Atkinson, M. O. e Chisholm, K. J. *Persistent Object Management System* Software-Practice and Experience, Vol. 14, 49-71 (1984)

[12] Davies, N.; Blair, G. S.; Friday, A.; Cross, A. D. e Raven, P. F. *Mobile Open Systems Technologies for the Utilities Industries* IEE Colloquium on CSCW Issues for Mobile and Remote Workers, London, U.K., 16th March 1993.

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

[14] Drummond, Rogério, Júnior, Celso G. *Objetos Distribuídos em Cm* XII Simpósio Brasileiro de Redes de Computadores UFPR, Curitiba-PR, 1994

[15] Drummond, Rogério; Júnior, Celso G. e Teles, Alexandre P. *Aspectos da Implementação de Objetos Distribuídos* XI Simpósio Brasileiro de Redes de Computadores, UNICAMP, Campinas-SP, 10-13 de maio de 1993

[16] Eager, Derek L.; Lazowska, Edward D. e Zahorjan, John *Adaptive Load Sharing in Homogeneous Distributed Systems* IEEE Transactions on Software Engineering, Vol. SE-12, No. 5, May 1986

[17] Free Software Foundation, Inc. *Manual GNU dbm* Edition 1.4.1 for gdbm Version 1.7.3

[18] Griethuysen, J. J. van *Enterprise modelling, a necessary basis for modern information systems* Elsevier Science Publishers B.V. (North-Holland) 1992 IFIP

[19] Gunji, Tetsu *Gerenciamento de Objetos - Relatório 4* Relatório Interno DCA-FEE-UNICAMP, Jun/94

[20] Herbert, Andrew *The Challenge of ODP* Elsevier Science Publishers B.V. (North-Holland) 1992 IFIP

[21] Iona Technologies Ltd. *ORBIX - distributed object technology* Release 1.3.1 February 1995

[22] ISO/IEC JTC1/SC21/N7074 *Working Document on Topic 9.1 : ODP Trader* June 1992

[23] ITU-T X.901 — ISO/IEC JTC1/SC21/WG7/N885 10746-1 *Basic Reference Model ODP - Part 1 : Overview and Guide to Use* Meeting of Torino (Italy), 3-11 November 1993

[24] ITU-T X.902 — ISO/IEC DIS 10746-2 *Basic Reference Model ODP - Part 2 : Descriptive Model*

[25] ITU-T X.903 — ISO/IEC DIS(E) 10746 *Basic Reference Model ODP - Part 2 : Prescriptive Model* 1994 Genebra Suiça

[26] ITU-T X.904 — ISO/IEC SC21/N7056 10746-4 *Basic Reference Model ODP - Part 4 : Architectural Semantics*

[27] Jajodia, Sushil e Mutchler, David *Dynamic Voting Algorithms for Maintaining the Consistency of a Replicated Database* ACM Transactions on Database Systems, Vol. 15, No. 2, June 1990, Pages 230-280

[28] Júnior, Alencar de Melo *Uma Estratégia de Escalonamento de Processos Periódicos e Esporádicos em Sistemas de Tempo Real Crítico Monoprocessados* Tese de Mestrado - Faculdade de Engenharia Elétrica - Jan/93

[29] Kilov, Haim *Precise specification of behaviour in object-oriented standardization activities* Computer Standards & Interfaces 15 (1993) 275-285 North-Holland

[30] Kim, Chonggun e Kameda, Hisao *An Algorithm for Optimal Static Load Balancing in Distributed Computer Systems* IEEE Transactions on Computers, Vol. 41, No. 3, March 1992

[31] Kock, Frank *A first approach for an ODP - Description Language from the Computational Point of View (ODP-DLcomp)* Relatório Técnico GMD Fokus Berlin, Alemanha.

[32] Kong, Qinzheng e Berry, Andrew *A General Resource Discovery System for Open Distributed Processing* Relatório Técnico do Projeto DSTC, Universidade de Queensland, Austrália, 4072

[33] Kurose, Jim *Open Issues and Challenges in Providing Quality of Services Guarantees in High-Speeed Networks* 3rd International Workshop on Network and Operating System Support for Digital Audio and Video, Nov. 12-13, 1992, San Diego, CA.

[34] Kumar, Akhil *Hierarchical Quorum Consensus: A New Algorithm for Managing Replicated Data* IEEE Transactions on Computer, Vol. 40, No. 9, September 1991

[35] Ladislau, Conceição *Plataforma Multiware: Protocolos de Comunicação* Tese de Mestrado - Faculdade de Engenharia Elétrica - Término previsto para Set/95

[36] Lamport, Leslie *A Simple Approach to Specifying Concurrent Systems* Communication of the ACM / January 1989 Volume 32 Number 1

[37] Lento, Luiz Otávio Botelho; Madeira, Edmundo Roberto Mauro *Um Esquema para Acessar Objetos em Ambientes Distribuídos* XIII Simpósio Brasileiro de Redes de Computadores, Abril de 1995

[38] Leopold, Helmut; Coulson, Geoff; Ansah, Kwaku Frimpong; Hutchison, David e Singer, Nikolaus *The evolving relationship between OSI and ODP in the new communications environment* Relatório Técnico, ED.02-V1 / 3BY 00212 4001 UPZZA Departamento de Computação - Universidade de Lancaster May 7, 1993

[39] Linington, P. F. *Open Distributed Processing* Elsevier Science Publishers B.V. (North-Holland) 1992 IFIP

[40] Maezi, Márcio *Plataforma Multiware: Interface de Programação* Tese de Mestrado - Faculdade de Engenharia Elétrica - Término previsto para Ago/95

[41] Mohan, C. e Lindsay, B. *Efficient Commit Protocols for the Tree of Processes Model of Distributed Transactions* Proceedings 2nd ACM SIGACT/SIGOPS Sysmposium on Principles of Distributed Computing Montreal, Canada, 17-19 August 1983

[42] Nicol, John R.; Wilkes, C. Thomas e Manola, Frank A. *Object Orientation in Heterogeneous Distributed Computing Systems* Computer June 1993

[43] Ramarao, K. V. S. *Design of Transaction Commitment Protocols* Information Sciences 55, 129-149 (1991)

[44] Raymond, K. A. *Reference Model of Open Distributed Processing: a Tutorial* Proceedings of the IFIP TC6/WG6.1 International Conference on Open Distributed Processing Berlim - Alemanha

[45] Robinson, David *Remote Procedure Call: a stepping stone towards ODP* Computer Networks and ISDN Systems 23 (1991) 191-194 North-Holland

[46] Rothermel, Kurt e Pappe, Stefan *Open Commit Protocols for the Tree of Processes Model* Proceedings The 10th Internation Conference on Distributed Computing Systems Paris, France 1990

[47] Rudkin, S. *Modelling object interfaces for open distributed processing* BT Technol Journal Vol. 10 No. 2 April 1992

[48] Rudkin, S. *Templates, types and classes in open distributed processing* BT Technol Journal Vol. 11 No. 3 July 1993

[49] Rumbaugh, James; Blaha, Michael; Premerlani, William; Eddy, Frederick e Lorensen, Willian *Modelagem e Projetos Baseados em Objetos* Editora Campus, 1994

[50] Rumbaugh, James; Blaha, Michael; Premerlani, William; Eddy, Frederick e Lorensen, Willian *Object Oriented Modelling and Design* Prentice Hall, 1991

[51] Satyanarayanan, M. *Integrating Security in a Large Distributed System* Relatório Técnico Computer Science Department, Carnegie Mellon, November 2, 1987

[52] Sloman, Morris and Kramer, Jeff *Distributed Systems and Computer Networks* Prentice Hall Internacional Series in Computer Science, 1987

[53] Smith, Reid G. *The Contract Net Protocol: High-Level Communication and Control in a Distributed Problem Solver* IEEE Transactions on Computer, Vol. C-29, No. 12, December 1980

[54] Stevens, Richard W. *UNIX Network Programming* Prentice Hall Software Series, 1990

[55] Stocks, Phil; Raymond, Kerry; Carrington, David e Lister, Andrew *Modelling open distributed systems in Z* Computer communications, Vol. 15, No. 2, March 1992

[56] Stroustrup, Bjarne *The C++ Programming Language - Second Edition* Addison Wesley, 1994

[57] Sun Microsystems *Manual SUN - Network Programming* Sun Microsystems Inc, 1990

[58] Sun Microsystems *Manual SUN - System Services Overview* Sun Microsystems Inc, 1988

[59] Takahashi, Tadao; Liesenberg, Hans K. E. e Xavier, Daniel T. *Programação Orientada a Objetos* VII Escola de Computação São Paulo - SP 1990

[60] Tanenbaum, Andrew S. *Modern Operating Systems* Prentice Hall International Edition, 1992

[61] Taylor, Calvin J. *Object-oriented concepts for distributed systems* Computer Standards & Interfaces 15 (1993) 167-170 North-Holland

[62] Tizzo, Neil Paiva *Trabalho Sobre Sincronização de Fluxos Multimídia* Tese de Mestrado - Faculdade de Engenharia Elétrica - Término previsto para Dez/95

[63] Tschammer, Volker; Mendes, Manuel J.; Souza, Wanderley L.; Madeira, Edmundo R. M. e Loyolla, Waldomiro P. *Processamento Distribuído Aberto e o Modelo RM-ODP / ISO* XI Simpósio Brasileiro de Redes de Computadores, UNICAMP, Campinas-SP, 10-13 de maio de 1993

[64] Vários autores e artigos *Concurrent Object-Oriented Programming* Communication of the ACM / September 1993 Volume 36 Number 9