UNIVERSIDADE ESTADUAL DE CAMPINAS

Faculdade de Engenharia Elétrica

Departamento de Engenharia de Computação e Automação Industrial

PLATAFORMA MULTIWARE: INTERFACE DE PROGRAMAÇÃO

por: Marcio Maezi

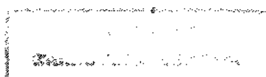
orientador: Eleri Cardozo

Este exemplar corresponde à redação final da tese defendida por MARCIO MAEZI e aprovada pela Comissão Julgadora em 30/08/1995.

Eleri Cardozo
Orientador

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.

Agosto/1995



UNIDADE BC
N.º CHAMADA:
T/UNICAMP
M268p
V.
TOMBO 25815
PROC. 433/95
C [ ] D [X]
PREÇO R$11,00
DATA 05/10/95
N.º CPD

CM-00077404-7

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

M268p	Maezi, Marcio Plataforma multiware: interface de programação. / Marcio Maezi.--Campinas, SP: [s.n.], 1995. Orientador: Eleri Cardozo. Dissertação (mestrado) - Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica. 1. Interfaces (computador). 2. Redes de Computação. 3. Programação orientada a objetos. 4. Computadores-Programação. 5. Processamento eletrônico de dados-Processamento distribuidos. I. Cardozo, Eleri. II. Universidade Estadual de Campinas. Faculdade de Engenharia Elétrica. III. Título.

Dissertação aprovada como requisito parcial para obtenção do título de Mestre do **Curso de Pós Graduação em Engenharia Elétrica da Universidade Estadual de** Campinas, pela Comissão formada pelos professores:

Orientador: Prof. Dr. Eleri Cardozo (DCA/FEE).

Prof. Dr.Edmundo R. M. Madeira (DCC/IMECC).

Prof. Dr. Ivan L. M. Ricarte (DCA/FEE).

Prof. Dr. Maurício F. Magalhães (DCA/FEE) (suplente)

Campinas, 30 de agosto de 1995.

Para os meus pais, Takachi e Maria (in memorian) e a minha irmã Solange.

AGRADECIMENTOS

Aos meus pais, Takachi e Maria que sempre me ajudaram, tornando possível este trabalho.

Aos meus amigos de Rio Claro, da famosa república da 8, MicroChip, Sagui, Tomate, Baratão, Shao, Sapão, Marcelo, Jair e Jesus, dentre outros, pelo companheirismo e pela amizade.

Ao Prof Dr. Eleri Cardozo pela orientação ao longo deste trabalho.

Aos meus amigos de Campinas, Tetsu, Neil, Gonzaga, Douglas, Cláudio e Marcone, pelos momentos de alegria e descontração.

Aos amigos da Casa M - 7 que tanto me aturaram.

Aos meus amigos, Affonso, Ivan e Maurício pelas discussões intermináveis.

A todos os colegas do LCA que de alguma forma me ajudaram na conclusão deste trabalho.

A CAPES pelo apoio financeiro.

Ouço, Esqueço.
Vejo, Recordo.
Faço, Compreendo.

Confúcio

CONTEÚDO

RESUMO x

ABSTRACT xi

CAPÍTULO 1 INTRODUÇÃO 1

1.1 Motivação 1

1.2 Objetivos 2

1.3 Plataforma Multiware 2
1.3.1 Camada Middleware 4
1.3.2 Componentes da Camada Middleware 5

1.4 Apresentação 5

CAPÍTULO 2 PROCESSAMENTO DISTRIBUÍDO ABERTO 7

2.1 Introdução 7

2.2 Modelo de Referência ODP 7
2.2.1 Abordagem de Modelagem e Conceitos Gerais 8
2.2.1.1 Conceitos de Modelagem Básica 8
2.2.1.2 Conceitos de Especificação 9
2.2.1.3 Conceitos Arquiteturais 11
2.2.2 Pontos de Vista 14
2.2.2.1 Ponto de Vista Empresarial 15
2.2.2.2 Ponto de Vista de Informação 16
2.2.2.3 Ponto de Vista Computacional 16
2.2.2.4 Ponto de Vista de Engenharia 17
2.2.2.5 Pontos de Vista Tecnológico 17

2.3 Common Object Request Broker Architecture (Corba) 18
2.3.1 Núcleo ORB 19
2.3.2 Skeleton 19
2.3.3 Interface ORB 20
2.3.4 Implementação de Objetos 20
2.3.5 Linguagem de Definição de Interface (Interface Definition Interface - IDL) 20
2.3.5.1 Interface de Invocação Estática. 21
2.3.5.2 Interface de Invocação Dinâmica 21
2.3.6 Repositório de Interface 21
2.3.7 Adaptador de objeto 21

2.4 System Object Model (SOM) 22
2.4.1 SOM Distribuído 22
2.4.2 Framework Repositório de Interface 22
2.4.3 Framework Persistência 23
2.4.4 Framework Replicação 23
2.4.5 Framework Emissor 23
2.4.6 Adaptador de Objeto SOM (SOMOA) 24
2.4.7 Linguagem de Definição de Interface do SOM (SOM IDL) 24
2.4.8 Mecanismos de Resolução de Métodos 24

2.5 Conclusão 25

CAPÍTULO 3 SERVIÇOS DE OBJETOS SEGUNDO O PADRÃO RM-ODP 26

3.1 Introdução 26

3. 2 Ponto de Vista Computacional 26
3.2.1 Template de Objeto Computacional 26
3.2.1.1 Template de Objeto Básico Computacional 26
3.2.1.2 Template de Objeto Binding 27
3.2.2 Template de Interface Computacional 27
3.2.2.1 Template de Interface Sinal 27
3.2.2.2 Template de Interface Operacional 27
3.2.2.3 Template de Interface Stream 28
3.2.2.4 Template de Sinal 28
3.2.3 Contrato 28
3.2.3.1 Restrições de Qualidade de Serviço 28
3.2.3.2 Restrições de Gerenciamento e Uso 29
3.2.4 Comportamento 29
3.2.5 Tipo 29

3.3 Ponto de Vista de Engenharia 29
3.3.1 Associação 29
3.3.2 Cluster 29
3.3.3 Cápsula 30
3.3.4 Nodo 31
3.3.5 Canal 32
3.3.5.1 Stub 33
3.3.5.2 Binder 34
3.3.5.3 Protocolo 34
3.3.5.4 Interceptador 35

3.5 Conclusão 35

CAPÍTULO 4 PROPOSTA DE UMA INTERFACE DE PROGRAMAÇÃO PARA OS SERVIÇOS DE OBJETOS ODP 36

4.1 Introdução 36

4.2 Object Modeling Technique (OMT) 36
4.2.1 Modelo de Objeto 36
4.2.2 Objetos 37

4.2.3 Classes 37
4.2.4 Diagramas de Objetos 37
4.2.5 Atributos 37
4.2.6 Operações e Métodos 38
4.2.7 Links e Associações 38
4.2.8 Qualificador 39
4.2.9 Multiplicidade 39
4.2.10 Agregação 40
4.2.11 Generalização 40

4.3 Interface de Programação de Aplicação **41**

4.4 Conclusão **45**

CAPÍTULO 5 IMPLEMENTAÇÃO **46**

5.1 Introdução **46**

5.2 External Data Representation **46**

5.3 Armazenamento de Templates **46**
5.3.1 Descrição da Estrutura de Armazenamento dos Templates 47

5.4 Comportamento (BEHAVIOUR) **47**

5.5 Métodos Bind e Accept **48**

5.6 Métodos Queue e Pop **48**

5.7 Métodos Submit, Deliver e Unbind **49**

5.8 Métodos readAttribute e modifyAttribute **50**

5.9 Métodos addInterface e deleteInterface **50**

5.10 Métodos makeObject, makeCluster e makeCapsule **50**

5.11 Métodos getInterfaces, getObjects, getClusters e getCapsules **51**

5.12 Métodos checkpoint<X>, recover<X>, delete<X> e deactivate<X> **51**

5.13 Conclusão **52**

CAPÍTULO 6 EXEMPLO DE UTILIZAÇÃO **53**

6.1 Introdução **53**

6.2 Criação de um Cluster **53**

6.3 Desativação de um Cluster **54**

6.4 Criação de um Canal **55**

6.5 Envio de dados pelo Canal Sinal 58

6.6 Conclusão 59

CAPÍTULO 7 CONCLUSÕES E TRABALHOS FUTUROS 60

7.1 Trabalhos Futuros 61

BIBLIOGRAFIA 62

ANEXOS

ANEXO A DESCRIÇÃO DOS MÉTODOS DA API EM C++ 64

ANEXO B ESTRUTURA DE ARMAZENAGEM DOS TEMPLATES 71

ANEXO C PARÂMETROS ENVIADOS ENTRE A API E A BASE DE OBJETOS 75

LISTA DE FIGURAS

Figura 1.1 Estruturação de uma Plataforma Multimídia ... 3
Figura 2.1 Composição de Objetos ... 9
Figura 2.2 Decomposição de Objeto ... 10
Figura 2.3 Instanciação de um template para produzir objetos ... 10
Figura 2.4 Binding Explícito ... 12
Figura 2.5 Objeto Cliente A no Processo de Binding ... 12
Figura 2.6 Instanciação de um Objeto Servidor B ... 13
Figura 2.7 Criação de uma Interface x do Tipo T1 Associada ao Objeto Servidor B ... 13
Figura 2.8 O Objeto Servidor B anuncia a Existência de uma Interface do Tipo T1 ... 13
Figura 2.9 Criação de uma Interface y do Tipo T2 para o Objeto Cliente A ... 13
Figura 2.10 O Objeto A decide comunicar-se com uma Inteface do Tipo T ... 14
Figura 2.11 O Objeto A encontra uma Interface x do Tipo T ... 14
Figura 2.12 Binding entre as Interfaces y e x ... 14
Figura 2.13 Pontos de Vista do RM-ODP ... 15
Figura 2.14 Componentes e Interfaces CORBA ... 18
Figura 2.15 Interface CORBAa e Repositórios de Implementação ... 20
Figura 3.1 Exemplo de Estrutura Suportando o Objeto Básico de Engenharia ... 30
Figura 3.2 Exemplo de Estrutura de Cápsula ... 31
Figura 3.3 Exemplo de Estrura de Nodo ... 32
Figura 3.4 Exemplo de um Canal Cliente/Servidor ... 33
Figura 3.5 Exemplo de um Canal Cliente/Servidor com Interceptador ... 34
Figura 4.1 Representação Gráfica de Classe e Objetos ... 37
Figura 4.2 Representação Gráfica de Classe com Atributos ... 38
Figura 4.3 Representação Gráfica de Classe, Atributo e Operação ... 38
Figura 4.4 Representação Gráfica de Qualificador ... 39
Figura 4.5 Representação Gráfica de Multiplicidade ... 40
Figura 4.6 Representação Gráfica de Agregação ... 40
Figura 4.7 Representação Gráfica de Generalização ... 41
Figura 4.8 Representação Gráfica da API ... 42
Figura 5.1 Uso dos Métodos Queue e Pop ... 49
Figura 5.2 Uso dos Métodos Submit e Deliver ... 49

Resumo

O rápido crescimento do processamento distribuído tem levado a necessidade de se estabelecer padrões para o processamento distribuído aberto (ODP: Open Distributed Processing). O modelo de referência ODP propõe uma arquitetura baseada em objetos que dá suporte a distribuição, interconexão e portabilidade.

A plataforma *multiware* é uma arquitetura para o suporte ao processamento distribuído aberto e presta-se a identificar os vários componentes e serviços de uma plataforma de suporte ao processamento distribuído aberto. A proposta deste trabalho é prover um dos serviços da plataforma multiware: a Interface de Programação (API: Application Programming Interface).

A API tem por objetivo oferecer mecanismos de acesso para o processamento distribuído aberto, provendo serviços com as mesmas funcionalidades descritas pelo modelo de referência ODP. A principal contribuição deste trabalho foi traduzir os elementos do modelo ODP para uma estrutura computacional de acesso à estes elementos, enfocando-se nos elementos de gerenciamento e comunicação, sem dúvida os mais importantes do modelo ODP.

Abstract

The rapid growth of distributed processing has led to a need for the standardization of Open Distributed Processing (ODP). The ODP reference model proposes an architecture based on objects that supports distribution, interworking and portability.

The *multiware* plataform is an architecture for open distributed processing and gives a highlight of the many components and services that a plataform compliant to ODP must provide. The proposal of this work is to provide one of multiware plataform components: Application Programming Interface (API).

The API has as a goal the offering of mechanisms for open distributed processing, providing services with the same funcionalities as described by the reference model ODP. The main contribution of this work was to translate the elements of ODP model into a computational structure that gives access to those elements, focusing on the management and communication elements, which are the most important of the ODP model.

ABREVIAÇÕES

API - Application Programming Interface;

ATM - Assynchronous Transfer Mode;

CASE - Computer Aided Software Engineering;

CORBA - Common Object Request Broker Architecture;

CSCW - Computer Supported Cooperative Work;

DAI - Distributed Artificial Intelligence;

DSOM - Distributed System Object Model;

IBM - International Bussiness Machine;

IDL - Interface Definition Language;

ISO - International Standard Organization;

ITU - International Telecomunication Union;

LAN - Local Area Network;

LCA - Laboratório de Engenharia de Computação e Automação Industrial;

ODP - Open Distributed Processing;

OMG - Object Management Group;

OMT - Object Modeling Technique;

ORB - Object Request Broker;

QoS - Quality of Service;

RPC - Remote Procedure Call;

RM-ODP - Reference Model for Open Distributed Processing;

SOM - System Object Model;

UCP - Unidade Central de Processamento;

XDR - eXternal Data Representation.

CAPÍTULO 1

INTRODUÇÃO

1.1 Motivação

Até recentemente não havia necessidade e nem tecnologia para a construção de sistemas distribuídos. A partir da década de 60, os computadores começaram a se tornar comuns. Os computadores eram muito caros e era importante que o limitado poder de processamento não fosse desperdiçado. Sistemas de processamento em *batch* maximizavam a execução de tarefas, mas a um custo de uma tediosa preparação e a um rendimento ainda baixo. Com o passar do tempo, as taxas de custo e desempenho para sistemas de computação foram se tornando mais favoráveis. Sistemas de *timesharing* começaram a se tornar mais comuns. A partir daí, aplicações mais complexas passaram a ser construídas, por exemplo, na utilização de grandes *mainframes* para processamento simultâneo de transações de banco de dados [Beretes93].

Estes complexos sistemas, muitas vezes necessitavam de integração para execução de certas tarefas. Entretanto, esta integração era tarefa difícil, devido ao fato de tais sistemas não terem sido projetados para interoperarem. Mesmo a transferência *off-line* de dados com fita magnética era difícil, pois cada fabricante provia um formato diferente de arquivo para seus produtos. Ao mesmo tempo, uma invenção revolucionária, o microprocessador, começava a prover vastas melhorias na taxa de custo/desempenho para computação. Porém, os avanços no *software* não acompanharam na mesma proporção a evolução do *hardware*. A construção de complexas aplicações de *software* ainda apresenta as mesmas dificuldades encontradas no passado. Devido a estas dificuldades, a tecnologia de desenvolvimento de *software* tende a limitar o surgimento de novos *hardwares*, pois não existirá sistemas que possam tirar proveito destes *hardwares*.

Com o surgimento das redes locais (LANS), tornou-se possível que dezenas (centenas) de máquinas pudessem se conectar e trocar informações a uma velocidade de milhões de bits/seg. Sistemas distribuídos podem ser caracterizados por possuirem vários processadores que se comunicam através de uma rede [Tanenbaum92].

Sistemas distribuídos utilizam *softwares* complexos que devem ser projetados cuidadosamente. O desenvolvimento de *softwares* para sistemas distribuídos envolvem vários tópicos, tais como transparência, flexibilidade, confiabilidade e desempenho.

A transparência visa esconder dos usuários os aspectos referentes a distribuição. O conceito de transparência pode ser aplicado em vários aspectos de um sistema distribuído. É importante ressaltar que um sistema distribuído seja flexível, facilitando a evolução futura do

sistema. Além disso, o sistema deve ser confiável, sendo tolerante a falhas e possuindo bom desempenho de processamento.

1.2 Objetivos

Devido ao rápido desenvolvimento e crescimento de sistemas distribuídos, houve a necessidade de se padronizar o processamento distribuído aberto. O modelo de referência para o processamento distribuído aberto provê distribuição, interconexão e portabilidade.

O objetivo deste trabalho é fornecer facilidades ao usuário para o desenvolvimento de sistemas distribuídos, através de uma interface de programação.

A interface de programação tem como objetivo oferecer mecanismos para o processamento distribuído aberto, provendo serviços com as mesmas funcionalidades descritas pelo modelo de referência ODP (Open Distributed Processing) [RM-ODPa,b,c,d].

A plataforma *multiware* é uma arquitetura para o suporte ao processamento distribuído aberto. Esta arquitetura presta-se a identificar os vários componentes e serviços de uma plataforma de suporte ao processamento distribuído aberto. A plataforma *multiware* possui entre os seus serviços a interface de programação [Fapesp95].

1.3 Plataforma Multiware

A plataforma *multiware* deve executar sobre múltiplas plataformas de hardware (RISC, CISC, etc.), sobre diversos sistemas operacionais (UNIX, OS/2, etc.) e sobre diversas infraestruturas de comunicação (FDDI, ATM, ETHERNET, etc). Em termos funcionais, a plataforma *multiware* é dividida em quatro camadas (figura 1.1) [Fapesp95, Lento95]:

- Camada de processamento local: é constituída pelo processador com o seu software básico como o sistema operacional (podendo ser baseado em *microkernel*), protocolos de comunicação, recursos de apresentação, etc. Esta camada não provê nenhuma facilidade de processamento distribuído.

- Camada *middleware*: 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. Esta camada deve implementar suas funcionalidades segundo o paradigma de orientação a objetos.

- Camada *groupware*: provê as funcionalidades requeridas por diferentes classes de aplicações, como o 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: gerenciamento do diálogo, protocolos de interação, manipulação de documentos multimídia, dentre outros.

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

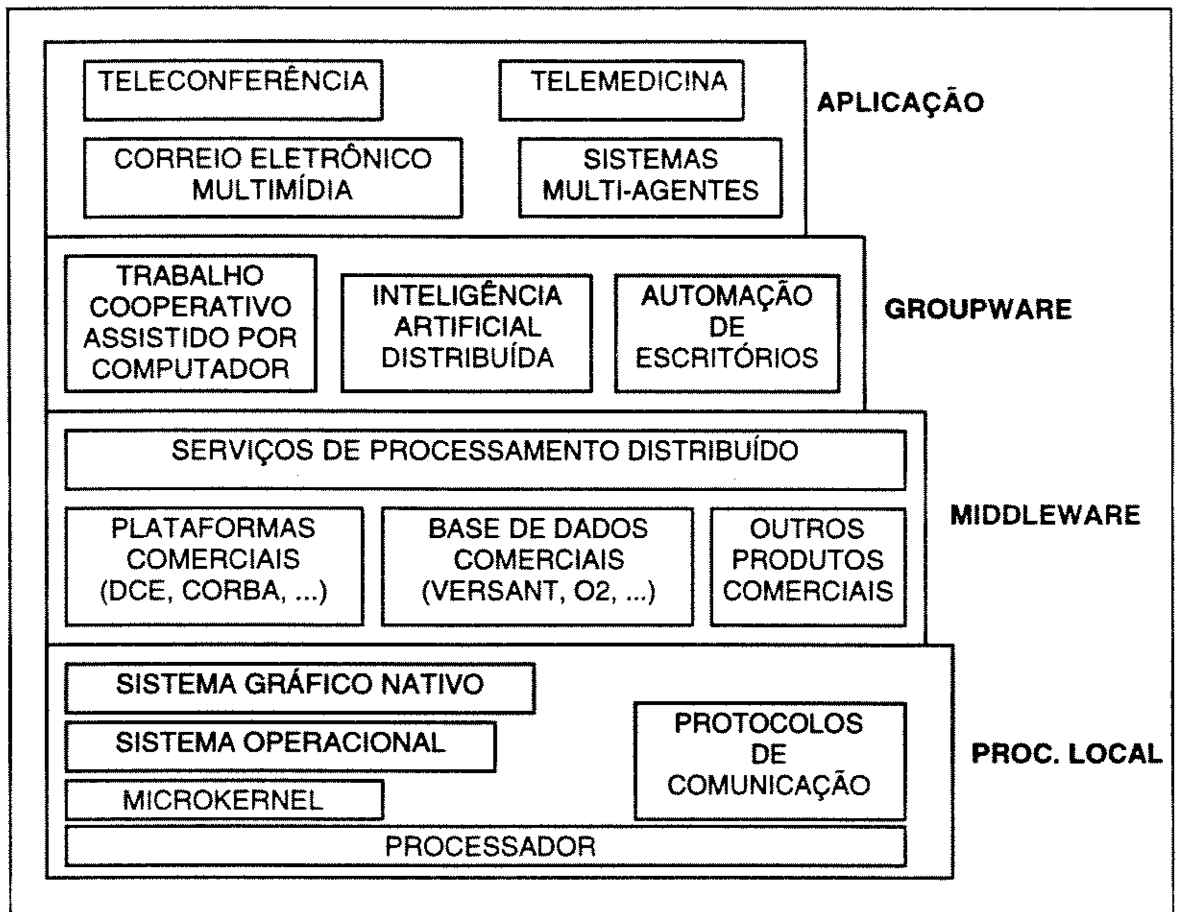


Figura 1.1 Estruturação de uma Plataforma Multiware

Plataformas não se constituem de produtos comerciais, mas da combinação de vários produtos e implementações que necessitam interoperar. A interoperabilidade só é atingida com o amadurecimento de padrões. Grande parte destes produtos e padrões (sistemas operacionais, redes de computadores, etc.) não foram concebidos para manipular fluxos contínuos de informação como áudio e vídeo. Como exemplo de tais produtos temos os sistemas operacionais UNIX, OS/2, e Windows, e as pilhas de protocolos de rede TCP/IP. Novos produtos estão surgindo ou passando por adaptações para se adequarem aos requisitos impostos pelas aplicações multimídia. É o caso dos protocolos de rede ST II e RTP em desenvolvimento no escôpo da Internet e o X-media, uma extensão do X Window para a manipulação de vídeo sobre este sistema gráfico.

1.3.1 Camada Middleware

A camada *middleware* deve ser estruturada levando-se em conta a tecnologia de orientação a objetos. Esta restrição visa propiciar às camadas superiores uma interface uniforme para os serviços de processamento distribuído que esta camada deve prover. A camada middleware dá suporte ao processamento distribuído aberto aderente ao modelo ODP, provendo as seguintes funcionalidades:

- Gerenciamento de objetos: para fins de gerenciamento, objetos são agrupados em *clusters* e *clusters* são agrupados em cápsulas. Todos os objetos de uma cápsula devem residir num mesmo nodo.
- Transparência: o modelo prevê várias formas de transparência onde propriedades como localização, acesso, concorrência, migração, replicação e falha, dentre outras, podem ser abstraídas pela aplicação (isto é, estas propriedades devem ficar circunscritas à plataforma).
- Interação entre objetos: o modelo ODP provê que a interação entre objetos (onde um objeto cliente evoca serviços oferecidos por um objeto servidor) se dê através de suas interfaces, se estes objetos estão localizados no mesmo *cluster*, caso contrário, é necessário o estabelecimento de uma conexão denominada canal. O canal pode transportar fluxo contínuo de dados, tendo as suas grandezas monitoradas (taxa, atraso, etc.) permitindo que seu fluxo possa ser sincronizado com o de outro canal, além de prover comunicação multiponto (um-para-muitos).
- Negociação (*trading*): objetos servidores podem exportar suas funcionalidades para uma entidade denominada *trader* [Tschammer93] e objetos clientes podem requerer um determinado serviço (contactando-se o *trader* para a obtenção do identificador do serviço requisitado).
- Segurança: provê serviços de proteção aos objetos.
- Serviços de repositório: provê serviços de armazenamento e persistência aos objetos.

O modelo de referência ODP especifica *quais* funcionalidades devem estar disponíveis às aplicações, mas não explicita *como* estas funcionalidades devam ser implementadas.

Resumidamente, podemos dizer que os requisitos de projeto da camada *middleware* é prover o maior número de funcionalidades ODP possível.

1.3.2 Componentes da Camada Middleware

A camada *middleware* é composta atualmente de quatro componentes e tem por objetivo prover funcionalidades com padrão ODP. Os componentes são listados abaixo:

- serviços de comunicação;
- serviços de núcleo;

- serviços de objetos;
- interface de programação (API: Application Programming Interface).

Serviços de Comunicação
Os serviços de comunicação se constituem da base para o processamento distribuído. Nesta implementação os serviços de comunicação provêem:

- difusão confiável de mensagens (*reliable multicast*);
- comunicação confiável uniponto (*reliable unicast*);
- difusão não confiável com garantia de banda (utilizada na comunicação envolvendo meios contínuos).

Serviços de Núcleo
O núcleo provê os serviços básicos de processamento distribuído. Nesta implementação o núcleo implementa serviços para:

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

Serviços de Objeto
Os serviços de objetos provêem a infraestrutura necessária à existência de objetos. Os objetos possuem propriedades fundamentais tais como instanciação, classificação e encapsulamento, além de prover mecanismos para gerenciamento a nível de cápsula, *cluster* e objeto, além de prover facilidades para o agrupamento de objetos em *clusters* e de *clusters* em cápsulas. Estes serviços estão disponíveis numa base de objetos.
Outra funcionalidade importante da base de objetos é prover a interação entre objetos (onde um canal liga uma interface de um objeto cliente a uma interface de um objeto servidor).

Interface de Programação (API)
A API fornece um acesso uniforme aos serviços da camada *middleware*. A camada *groupware* acessa estes serviços via API. O projeto e implementação da API foi objeto deste trabalho. Nesta implementação, a API consiste de uma hierarquia de classes especificadas segundo a metodologia OMT [Rumbaugh91] (Object Modeling Technique) e disponíveis na linguagem de programação C++.

1.4 Apresentação

Este trabalho está organizado da seguinte forma. O capítulo 2 apresenta uma abordagem do modelo de referência para processamento distribuído aberto, em fase de padronização pela ISO e ITU. Descreve ainda dois padrões para suportar o processamento distribuído aberto,

o CORBA e o SOM. O CORBA é um padrão comercial sendo desenvolvido por um consórcio de empresas e o SOM é fornecido pela IBM. Todos têm um objetivo em comum: facilitar a interoperabilidade de objetos distribuídos. Este objetivo é a interoperabilidade entre sistemas.

O capítulo 3 descreve os serviços de objetos definidos no modelo de referência do processamento distribuído aberto, enfocando os pontos de vista computacional e de engenharia.

O capítulo 4 descreve uma proposta para o desenvolvimento de uma interface para programação para os serviços de objetos ODP. A interface de programação é modelada através da metodologia OMT, utilizando o modelo de objetos.

O capítulo 5 faz uma descrição funcional da interface de programação proposta, abordando os pontos mais relevantes da implementação.

O capítulo 6 traz exemplos de utilização da interface de programação.

O capítulo 7 contêm as conclusões e trabalhos futuros.

CAPÍTULO 2

PROCESSAMENTO DISTRIBUÍDO ABERTO
(Open Distributed Processing - ODP)

2.1 Introdução

O processamento distribuído aberto relaciona o desenvolvimento, uso e gerenciamento de aplicações distribuídas através de sistemas computacionais, estendendo e incorporando trabalhos executados independentemente nos campos das comunicações e da computação.

O objetivo principal da padronização do processamento distribuído aberto é permitir que os procedimentos de utilização de um conjunto de recursos (*hardware e software*) heterogêneos e distribuídos, que possibilitam o acesso e o processamento da informação, possam ser realizados de uma forma consistente, confiável e flexível.

O item 2.2 descreve o modelo de referência para processamento distribuído aberto. Os itens 2.3 e 2.4 descrevem sistemas orientados a objetos, como o Commom Object Request Broker Architecture - CORBA, desenvolvido por um consórcio de empresas, e o System Object Model - SOM, fornecido pela IBM. CORBA e SOM são resultados de esforços conduzidos por empresas e consórcios de empresas na linha de processamento distribuído aberto e são soluções mais simplificadas que o Reference Model - Open Distributed Processing (RM-ODP), podendo ser utilizadas para suporte a determinadas funcionalidades do ODP (e.g. serviços de núcleo). O item 2.5 traz uma breve conclusão deste capítulo.

2.2 Modelo de Referência ODP

O Modelo de Referência Básica do Processamento Distribuído Aberto (Reference Model - Open Distributed Processing - RM-ODP) é um esforço conjunto das organizações de padronização internacional ISO (ISO/IEC 10746) e ITU-T (ITU-T Recommendations X.901 a X.904) em desenvolver um *framework* para a padronização do processamento distribuído aberto [RM-ODPa,b,c,d]. O modelo de referência descreve uma arquitetura que suporta distribuição, *interworking*, interoperabilidade e portabilidade. O *framework* RM-ODP define os conceitos de ODP usando cinco pontos de vista (abstrações), chamadas de empresarial, informação, computacional, engenharia e tecnológico.

O RM-ODP possui metas a serem alcançadas. As principais metas pretendidas são:

- portabilidade de aplicações através de plataformas heterogêneas;

- *interworking* entre sistemas ODP, isto é, a troca de informação entre sistemas ODP e o uso conveniente das funcionalidades em todo o sistema distribuído;

- transparência de distribuição, isto é, esconder do programador e do usuário as consequências da distribuição.

O modelo de referência provê uma visão ampla do sistema e organiza as peças de um sistema ODP dentro de um conjunto coerente e não tem por objetivo padronizar os componentes do sistema, nem influenciar desnecessariamente a escolha da tecnologia a ser utilizada.

O RM-ODP deve ser adequado para descrever os mais variados sistemas distribuídos. Portanto o RM-ODP deve ser abstrato, mas não vago. O RM-ODP deve cuidadosamente descrever seus componentes sem prescrever uma implementação.

O RM-ODP propõe organizar serviços dentro de sistemas autônomos a fim de facilitar o *interworking* de componentes de software distribuído. Como ferramentas, o RM-ODP oferece uma abordagem de modelagem geral e um conjunto de conceitos gerais, descrevendo uma metodologia para dividir a especificação de um sistema em cinco pontos de vista, a fim de simplificar a tarefa de especificação.

2.2.1 Abordagem de Modelagem e Conceitos Gerais

O RM-ODP define uma abordagem de modelagem baseada em objetos e envolve três conjuntos de conceitos. O primeiro conjunto, chamado de conceitos de modelagem básica, introduz o termo objeto e outros termos. O segundo conjunto, chamado de conceitos de especificação, focalizam sobre os requerimentos das linguagens de especificação. O terceiro conjunto, chamado de conceitos arquiteturais, consiste de conceitos de estruturação que surgiram devido aos diferentes enfoques em distribuição e sistemas distribuídos.

2.2.1.1 Conceitos de Modelagem Básica

Objetos

Todo sistema ODP deve ser baseado em objetos. Objetos são entidades contendo informação e oferecendo serviços. Um sistema é composto de uma interação de objetos e um objeto é único através de sua identidade, sua encapsulação e abstração, e seu comportamento.

Encapsulação e Abstração

Encapsulação é uma propriedade no qual a informação contida por um objeto é somente acessada através de interações realizadas pelas interfaces que compõem o objeto. Abstrações implicam que os detalhes internos de um objeto sejam escondidos de outros objetos.

Comportamento e Estado

Estado e comportamento são conceitos interrelacionados. O comportamento de um objeto é definido como o conjunto de todas as ações potenciais que o objeto possa realizar. O estado caracteriza a situação de um objeto em um dado instante. O comportamento de um objeto descreve as mudanças de estado de um objeto [Kilov93].

Interface

A única forma de se acessar um objeto é através de suas interfaces. A interface pode ser vista como parte do comportamento de um objeto. Cada interface compreende um conjunto de interações que podem invocar, ou serem invocadas por outros objetos. Um objeto ODP pode possuir muitas interfaces. Identificadores de interface são utilizados para identificar e distinguir as interfaces existentes [Rudkin92].

2.2.1.2 Conceitos de Especificação

Composição e Decomposição

No processo de desenvolvimento de uma aplicação distribuída, a composição e decomposição de objetos podem ser necessárias.

A composição de objetos é uma forma de descrever as relações hierárquicas entre o objeto composto e seus componentes. É um conceito de modelagem que permite um subsistema ser tratado como um simples objeto de mais alto nível.

A figura 2.1 representa a composição de dois objetos *A* e *B*. Na composição dos objetos *A* e *B*, o comportamento de *A* deve interagir apropriadamente com *B*. Com a composição dos objetos *A* e *B*, as interações entre *A* e *B* passam a ser escondidas, tornando-se ações internas de um objeto composto.

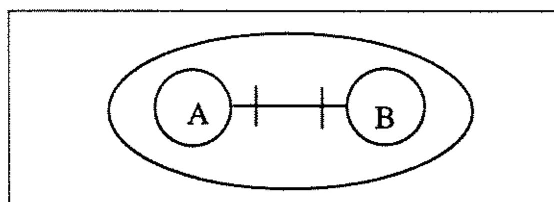


Figura 2.1 Composição de Objetos

A decomposição é relacionada com o conceito de sucessivos refinamentos, permitindo que complexas aplicações distribuídas possam ser decompostas em objetos mais simples. A figura 2.2 representa a decomposição de um objeto.

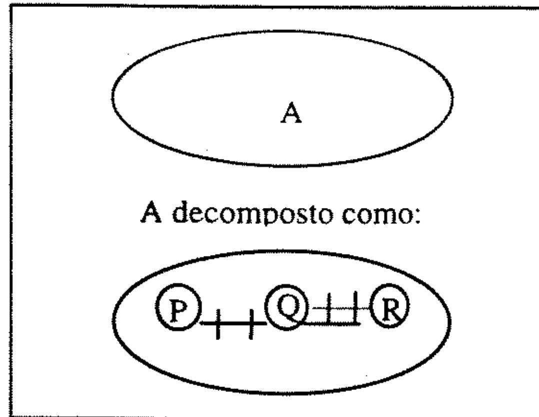


Figura 2.2 Decomposição de Objeto

Templates

Templates são usados para descrever objetos. Objetos com o mesmo comportamento podem ser descritos pelo mesmo *template*. Um *template* descreve suas características comuns e abstrai as suas diferenças (por exemplo, descrevendo os parâmetros de estados e operações) [Rudkin93].

O processo de instanciação é utilizado para se produzir novos objetos a partir de um template. (Veja figura 2.3).

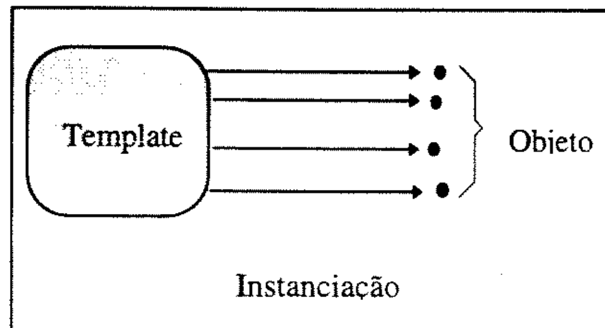


Figura 2.3 Instanciação de umTemplate para produzir Objetos.

Papéis (Roles)

Em uma interação, objetos assumem papéis. Cada objeto pode possuir vários papéis, podendo os papéis serem iguais ou diferentes. O papel revela o motivo pelo qual o objeto participa da interação. Típicos papéis que objetos podem assumir são: cliente, servidor, produtor e consumidor de informação.

Tipos e Classes

Um tipo é um predicado, sendo uma propriedade de uma coleção de objetos. Um objeto satisfaz um tipo, ou é de um tipo, se o predicado for verdadeiro para o objeto. Objetos não precisam ser todos similares para satisfazer o mesmo tipo, somente necessitam possuir as propriedades prescritas pelo tipo.

Tipos implicitamente classificam objetos dentro de conjuntos conhecidos como classes. Uma classe é uma coleção de objetos associados a um tipo.

2.2.1.3 Conceitos Arquiteturais

Nome

Nome é um aspecto importante na construção de sistemas distribuídos. Nome é usado para distinguir e acessar componentes de um sistema distribuído. Os princípios de heterogeneidade, autonomia e federação demandam nomes dependentes de contexto. O RM-ODP não especifica o *framework* de nomes para o ODP, sendo sujeitos a uma padronização em separado.

Grupos

O conceito de grupo é utilizado em sistemas distribuídos de várias maneiras. O conceito de grupo adotado pelo RM-ODP é uma definição genérica e permite a especificação de diferentes tipos de grupos. O grupo é descrito como um conjunto de objetos mantidos juntos por motivos estruturais ou devido ao fato dos objetos possuírem comportamentos com características comuns.

Contrato

Contrato é um conceito geral para caracterizar e regulamentar a cooperação entre objetos. Um contrato é um acordo que restringe a cooperação entre um conjunto de objetos. Incorpora as idéias de obrigação e expectativa associados com a cooperação de objetos. Um contrato pode especificar as tarefas dos objetos, que são os comportamentos e obrigações esperadas na cooperação, os aspectos de qualidade de serviço e o tipo de comportamento que invalida o contrato. O contrato é uma especificação dinâmica da configuração dos objetos. É dinâmico porque, embora o potencial de cooperação entre os objetos seja constante, as regras providas pelo contrato restringem o potencial de cooperação de certos objetos.

Liasion e Binding

Liasion e *binding* são ambos conceitos contratuais no sentido que eles envolvem contrato entre os objetos participantes. *Binding* é a fase preliminar do estabelecimento do contrato, permitindo a cooperação de objetos. *Liasion* é a relação que caracteriza a cooperação de

objetos sobre os aupícios de um contrato estabelecido. O *binding* produz um *liasion*. O *binding* pode ser implícito ou explícito.

Binding Explícito

O *binding* explícito é executado como uma ação. A ação de ligação cria um objeto que encapsula os mecanismos de *binding*, permitindo o *binding* ser manipulado. Esta ação toma como argumentos o tipo de binding requerido e o conjunto inicial de identificadores de interface. Se a ação obtiver sucesso, a ação retorna um identificador de interface para uma interface que controla o *binding* criado. Esta interface contém pelo menos uma operação, *unbind*, que termina o *binding*. A interface de controle do binding pode conter outras operações, dependendo do tipo do *binding* (figura 2.4).

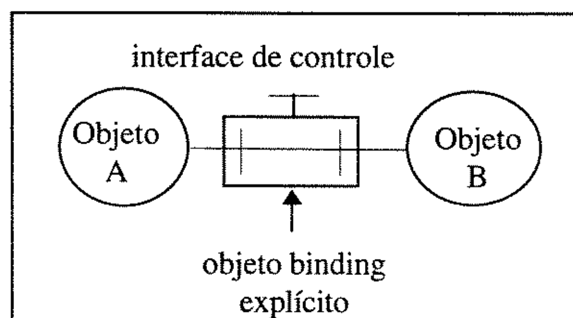


Figura 2.4 Binding Explícito

Binding Implícito

Um objeto obtém um identificador de interface de outro objeto por algum meio. O identificador de interface pode ser obtido de um *trader* ou pode ser recebido como resultado de alguma interação.

O processo de *binding* implícito e a interação entre interfaces é discutido através de uma série de diagramas. Estes diagramas seguem abaixo:

1. Existência de um objeto *A* (cliente) no estado inicial.

Figura 2.5 Objeto Cliente A no Processo de Binding.

2. Em um dado instante, é instanciado um objeto B (servidor).

Figura 2.6 Instanciação de um Objeto Servidor B.

3. A criação de uma interface *x* do tipo *T1* (descrito como *x:T1*) sendo associada ao objeto servidor B. O símbolo '⊥' partindo de *B* representa uma interface *x*. A flecha direcionada para *B* indica que a interface tem sido associada ao servidor. Esta interface poderia ter sido criada no instante em que o objeto fora instanciado.

Figura 2.7 Criação de uma Interface x do Tipo T1 Associada ao Objeto Servidor B.

4. O objeto servidor *B* anuncia a existência da interface *x* do tipo *T1*.

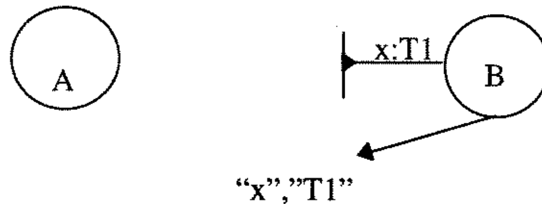


Figura 2.8 O Objeto Servidor B anuncia a Existência de uma Interface do Tipo T1.

5. Criação de uma interface *y:T2* por parte do objeto *A*. Esta interface é associada ao objeto cliente *A*. A interface *y:T2* poderia ter sido criada no processo de instanciação do objeto *A*.

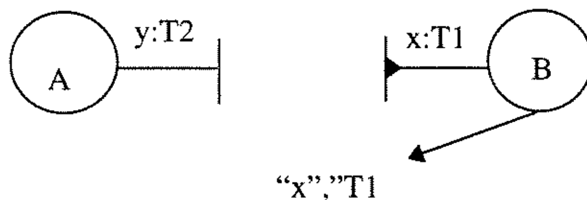


Figura 2.9 Criação de uma Interface y do Tipo T2 para o Objeto Cliente A.

6. O objeto cliente *A* decide comunicar-se com alguma interface do tipo *T*.

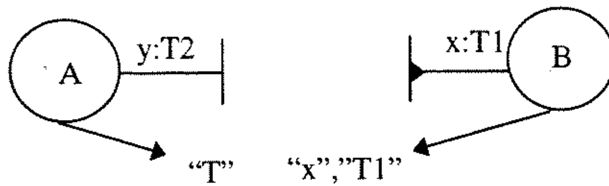


Figura 2.10 O Objeto A decide comunicar-se com uma Inteface do Tipo T.

7. O objeto *A* encontra uma interface "x" do tipo *T* (Neste caso, o tipo *T1* é subtipo de *T*).

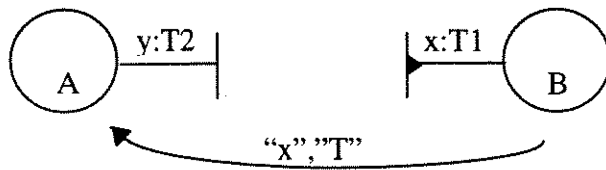


Figura 2.11 O Objeto A encontra uma Interface x do Tipo T.

8. Os objetos *A* e *B* estão interagindo através de suas interfaces, ocorrendo o *binding* entre os objetos.

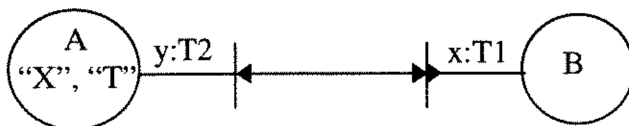


Figura 2.12 Binding entre as Interfaces y e x.

2.2.2 Pontos de Vista

A especificação completa de um sistema distribuído não trivial acarreta na aquisição de grande quantidade de informação. Um sistema distribuído é um sistema complexo. O RM-ODP lida com a complexidade do sistema distribuído através do processo de abstração. O RM-ODP identifica cinco diferentes abstrações (chamadas de pontos de vista) que permitem uma compreensão de um sistema onde cada um enfatiza um interesse particular e ignora outras características que são temporariamente irrelevantes ao contexto. Um ponto de vista é simplesmente um mecanismo de abstração para construção de modelos que descrevem o sistema. O RM-ODP prevê cinco pontos vista, onde cada ponto de vista é uma visão parcial da especificação de sistema completo:

- ponto de vista empresarial (propósito, escopo e políticas);

- ponto de vista de informação (semânticas de informação e processamento de informação);
- ponto de vista computacional (decomposição funcional);
- ponto de vista de engenharia (infraestrutura requerida para suportar distribuição); e
- ponto de vista tecnológico (escolhas de tecnologia para implementação).

A figura 2.13 mostra como os pontos de vista do RM-ODP podem ser relacionadas com o processo de engenharia de software [Raymond93].

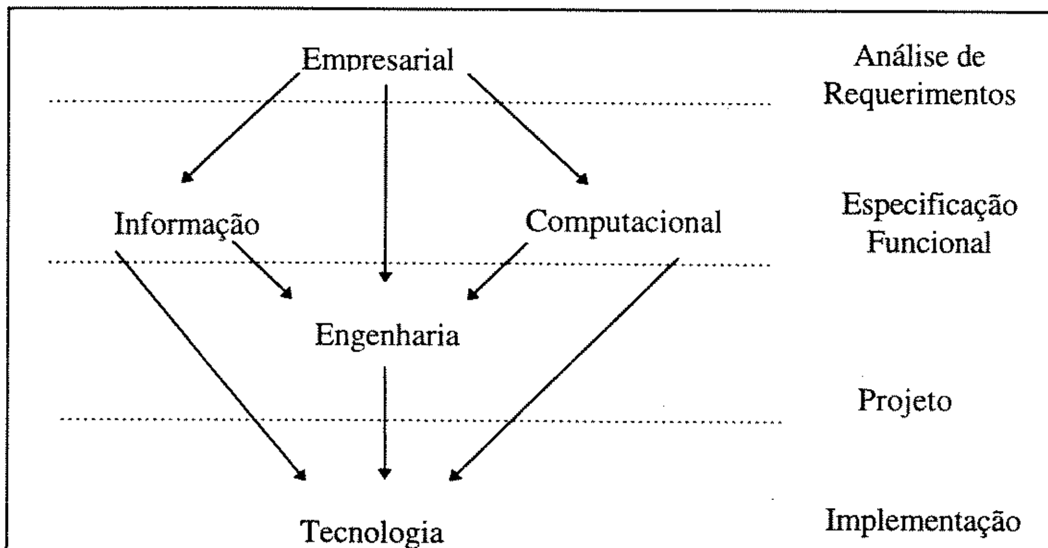


Figura 2.13 Pontos de Vista do RM-ODP.

2.2.2.1 Ponto de Vista Empresarial

O ponto de vista empresarial é usado em requerimentos e estruturas organizacionais. No ponto de vista empresarial, políticas organizacionais e sociais podem ser definidas em termos de:

- agentes - objetos ativos que iniciam ações executáveis (e.g. gerente de banco, cliente);
- artefatos - objetos passivos que não iniciam ações (e.g. conta corrente);
- comunidades - agrupamentos de agentes e artefatos (e.g. banco consiste de gerente, clientes e conta corrente);
- tarefas de agentes, artefatos e comunidades, são expressos em termos das seguintes políticas:
    - permissão - o que pode ser feito (e.g. retirar dinheiro);
    - proibição - o que não deve ser feito (e.g. retirar mais que 500 reais por dia);
    - obrigações - o que deve ser feito (e.g. indicar ao cliente sobre aplicações melhores).

2.2.2.2 Ponto de Vista de Informação

O ponto de vista de informação utiliza esquemas, relações e regras de integridade para descrever as informações requeridas por uma aplicação ODP.

Um esquema descreve o estado e a estrutura de um objeto, podendo ser dividido em três tipos de esquemas:

- esquema estático: captura o estado e a estrutura de um objeto de informação. O esquema estático é utilizado para especificar o estado inicial de um objeto (e.g. a quantidade de dinheiro que um cliente retira de um caixa eletrônico é zerado a meia noite);
- esquema invariante: restringe o estado e a estrutura de um objeto em todo o período (e.g. a quantidade de dinheiro retirado em um caixa eletrônico deve ser inferior ao limite de retirada permitido por dia).;
- esquema dinâmico: especifica as possíveis mudanças de estado e estrutura de um objeto, sendo restrito aos esquemas invariantes (e.g. a retirada de X reais de um caixa eletrônico decrementa o saldo bancário em X reais e não deve superar o limite de retirada permitido por dia).

A relação descreve as associações entre objetos (e.g. um cliente possui conta).

Uma regra de integridade é uma asserção sobre um esquema (e.g todo cliente deve possuir pelo menos uma conta bancária e toda conta bancária deve possuir pelo pelo um cliente).

A especificação de informação de uma aplicação ODP pode ser expressa através de vários métodos, tais como modelos de entidade-relacionamento e esquemas conceituais.

2.2.2.3 Ponto de Vista Computacional

O ponto de vista computacional é utilizado na especificação das funcionalidades da aplicação ODP. O sistema é descrito como um conjunto de objetos que interagem, sem se preocupar com os detalhes desta interação. Este ponto de vista especifica os componentes lógicos e individuais que são os recursos e *sinks* de informação. Os elementos deste modelo são escolhidos para abrigar os requerimentos referentes aos sistemas distribuídos. O ponto de vista computacional é baseado em objetos, onde:

- objetos encapsulam dados e processamento (comportamento);
- objetos oferecem interfaces para interação com outros objetos;
- objetos podem oferecer múltiplas interfaces.

A especificação computacional define os objetos dentro de um sistema ODP, as atividades dentro destes objetos, e as interações que ocorrem entre os objetos.

Transparência de Distribuição

O RM-ODP define um conjunto de transparências de distribuição. Estas transparências de distribuição tem a função de ocultar os detalhes técnicos de distribuição de programas de aplicação.

As transparências de distribuição definidas pelo RM-ODP são:

- transparência de acesso: esconde os diferentes mecanismos de acesso utilizados pelos recursos locais e remotos;
- transpârencia de localização: esconde a topologia do sistema;
- transparência de replicação: esconde os efeitos de múltiplas cópias de serviço e de informação;
- transparência de falha: esconde as falhas do sistema e as recupera se possível;
- transparência de persistência: esconde do objeto a possibilidade de ser desativado e reativado durante o seu tempo de vida para economizar recursos de processamento, armazenamento e comunicação;
- transparência de relocação: esconde do objeto o fato das interfaces a ele associadas terem sido mudadas;
- transparência de transação: mascara a coordenação de atividades entre uma configuração de objetos, para manter a consistência;
- transparência de migração: esconde a heterogeneidade dos componentes do sistema para permitir a migração das funções do sistema e aplicações entre componentes.

O RM-ODP define um conjunto transparências e permite que novas transparências de distribuição possam ser criadas e adicionadas ao grupo de transparências existentes no RM-ODP.

2.2.2.4 Ponto de Vista de Engenharia

O ponto de vista de engenharia é usado para descrever o projeto nos aspectos de distribuição orientada a objetos de um sistema ODP, definindo um modelo para a infraestrutura de sistemas distribuídos. O ponto de vista de engenharia não está interessado na semântica da aplicação ODP, exceto na determinação de seus requerimentos para distribuição e transparência de distribuição. Maiores detalhes serão vistos no próximo capítulo.

2.2.2.5 Pontos de Vista Tecnológico

Uma especificação tecnológica descreve a implementação de um sistema ODP em termos da configuração de objetos, representando os componentes de hardware e software da

implementação, sendo restrito pelo custo e disponibilidade de hardware e/ou produtos de software que satisfazem esta especificação.

2.3 Common Object Request Broker Architecture (CORBA)

O Object Management Group - OMG é um consórcio de empresas dedicados a produzir especificações de ambientes orientados a objetos.

O consórcio tem várias centenas de membros. A especificação CORBA é uma tentativa inicial da OMG de satisfazer suas metas.

O CORBA especifica uma arquitetura *bare-bone* para gerenciamento de objetos distribuídos [Nicol93]. Ela inclui cinco interfaces dependentes do Object Request Broker (ORB). A figura 2.14 mostra estas interfaces, adicionadas a oito componentes maiores de arquitetura. Os componentes são: cliente, implementação de objeto, invocação dinâmica, *stubs* IDL, interface ORB, *skeleton* IDL e Adaptador de objeto [Mello94].

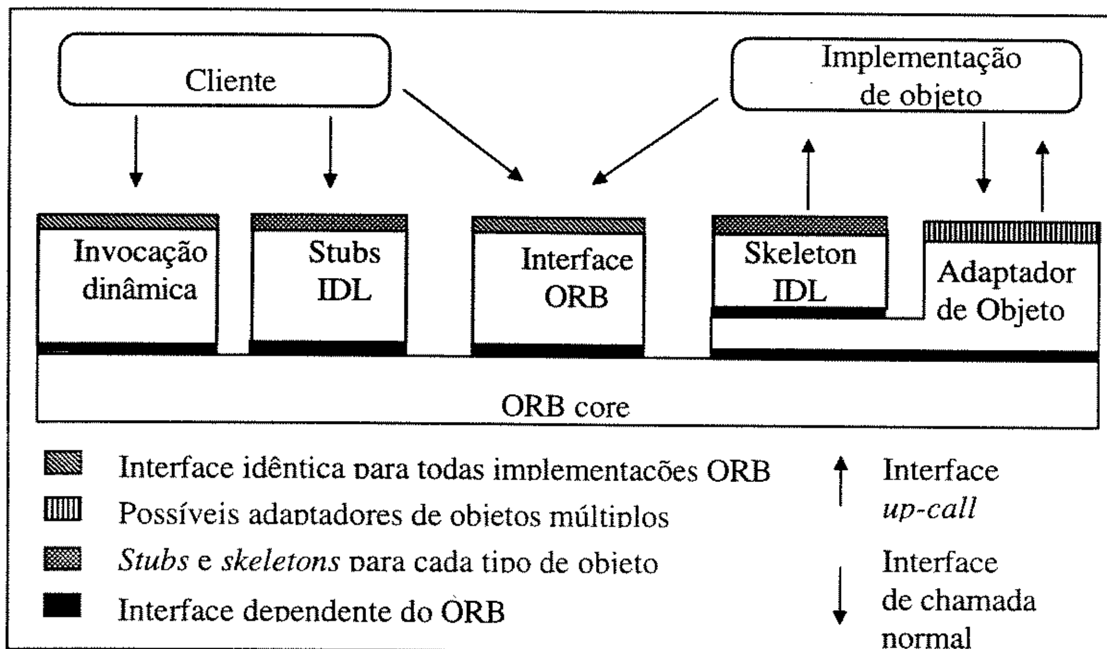


Figura 2.14 Componentes e Interfaces CORBA

Um cliente é uma entidade que espera invocar uma operação sobre um objeto destino via ORB. A implementação do objeto compreende o código e os dados que realizam o comportamento do objeto destino. O ORB localiza uma implementação de objeto para uma requisição, assegura que a implementação do objeto está pronta para receber a requisição e transmite a requisição de dados e resultados entre o cliente e o objeto destino.

Abaixo serão descritos os elementos que compõem o *framework* CORBA.

- Modelo de Objeto: O modelo de objeto CORBA provê facilidades para a definição dos objetos. O grupo de gerenciamento de objeto mantém um modelo simples para facilitar o mapeamento entre as variedades de modelos de objetos específicos que suportarão as implementações concretas. O modelo tem várias características chaves:
  - Objetos são identificáveis, onde entidades encapsuladas provêem um ou mais serviços. Objetos são criados e destruídos como resultado de execução de requisições de objetos. Um cliente identifica um objeto via uma referência de objeto, que sempre denota univocamente o mesmo objeto.
  - Requisições são mecanismos através do qual clientes requisitam outros serviços de objetos. Várias peças de informação são associadas a cada requisição. Estas podem ser uma operação, um objeto destino, zero ou mais parâmetros e um contexto de requisição normal. O cliente usa o contexto de requisição para enviar informação adicional a um objeto destino O objeto destino retorna uma exceção caso uma condição anormal ocorra durante o processamento da requisição.
  - Tipos classificam os objetos de acordo com as características de compartilhamento. Desenvolvedores usam tipos em assinaturas de operação para restringir parâmetros incluídos em requisições.
  - Interfaces descrevem as operações que o cliente pode requisitar em relação ao objeto.
  - Operações são entidades identificadas por nomes denotando serviços que podem ser requisitados. Toda operação possui uma assinatura que especifica os argumentos requeridos para invocação da operação, os resultados obtidos através da invocação da operação e retornados ao cliente e as exceções que devem ser retornadas caso aconteça um processamento anormal da operação.

2.3.1 Núcleo ORB (Object Request Broker)

O núcleo ORB negocia requisitos entre clientes e implementações de objetos. A negociação envolve a localização de objetos (destino), distribuição de mensagens e métodos de ligação. Desde que o núcleo ORB deva suportar uma variedade de mecanismos de objetos, o CORBA é estruturado de forma que os componentes acima do núcleo ORB forneçam interfaces que mascarem as diferenças entre os mecanismos encontrados nos diferentes núcleos ORB.

2.3.2 Skeleton

Skeleton é um componente ORB para especificação da interface do objeto o qual auxilia um adaptador de objeto passar pedidos para os métodos particulares do servidor.

2.3.3 Interface ORB

A interface ORB é uma interface que age diretamente sobre o ORB o qual é a mesma para todos os ORBs e não depende da interface de objeto ou adaptador de objeto.

2.3.4 Implementação de Objetos

Uma implementação de objeto compreende os dados e códigos requeridos para implementar um objeto com certo comportamento. Uma implementação de objeto geralmente define o código (método) de implementação de cada operação, devendo ainda incluir códigos para ativação e desativação de objetos. As implementações de objetos acessam serviços CORBA via uma interface chamada de "adaptador de objeto".

2.3.5 Linguagem de Definição de Interface (Interface Definition language- IDL)

A linguagem de definição de interface CORBA descreve as operações e os atributos de uma interface de objeto. IDL faz o possível para traduzir a funcionalidade oferecida pelos recursos tais como dispositivos de rede, base de dados, e outras aplicações.

Deste modo, IDL permite que uma implementação de objeto informe aos potencias clientes sobre as operações que oferece e como estas devem ser invocadas.

IDL provê clientes com um conjunto uniforme de semânticas para todas as interfaces clientes no CORBA.

Um cliente pode invocar operações via interface estática ou dinâmica. No primeiro caso, especificações IDL são compiladas gerando *stubs* do cliente e *skeletons* do servidor. No último caso, interfaces são armazenadas em um repositório de interfaces acessível através da interface de invocação dinâmica (figura 2.15).

IDL é independente da linguagem de programação através do uso de convenientes mapeamentos.

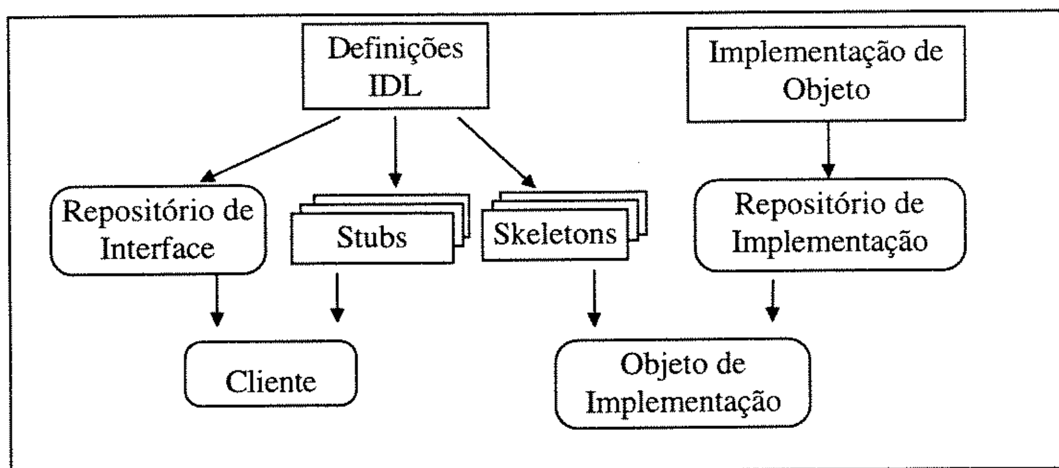


Figura 2.15 Interface Corba e Repositórios de Implementação

2.3.5.1 Interface de Invocação Estática.

As rotinas *stub* compiladas das especificações de interface IDL suportam interfaces de invocação estática do CORBA. Um programa cliente utiliza um conjunto de rotinas *stub*, cada um dos quais corresponde a uma operação particular de um objeto. Para se invocar a operação de um objeto, o cliente chama a rotina *stub* correspondente ao objeto destino.

2.3.5.2 Interface de Invocação Dinâmica

A interface de invocação dinâmica é um mecanismo para especificação de requisitos em tempo de execução. A interface de invocação dinâmica é necessária quando não se pode conhecer o tipo de interface em tempo de compilação. A interface de invocação dinâmica permite construção de invocações de objetos onde o cliente pode especificar o objeto a ser invocado, a operação a ser executada, e ou seu conjunto de parâmetros, através de uma chamada ou sequência delas [Mendes94].

2.3.6 Repositório de Interface

O repositório de interface suporta a interface de invocação dinâmica. O repositório de interface armazena objetos representando informações ID e contém informações que permitem o ORB localizar e ativar objetos. Para tanto a invocação estática utiliza a informação armazenada no repositório de interface para facilitar a verificação de tipo de objetos em tempo de execução. O repositório de interface pode ser usado para armazenar informações adicionais associadas às interfaces, tais como informações de *debugging* e bibliotecas de *stubs* e *skeletons*.

2.3.7 Adaptador de Objeto

Adaptadores de objetos CORBA servem a um duplo propósito. Primeiro, eles provêem a interface principal através do qual as implementações invocam os serviços do núcleo ORB. Segundo, eles incrementam o adaptador básico do CORBA pela adição de suporte a modelos mais ricos de objetos. O adaptador de objeto provê suporte a vários modelos de objetos de diferentes granularidades bem como a forma como estes objetos são ativados.

O CORBA define o adaptador de objeto básico para ser largamente disponível oe suportado por uma ampla variedade de implementações de objetos. Este adaptador de objetos provê interfaces para as seguintes funções:

- ativação e desativação de implementações de objetos.
- generalização e interpretação de referências de objetos;
- autenticação de requisitos de clientes;
- registro de implementações de objetos;

2.4 System Object Model (SOM)

SOM é um conjunto de bibliotecas, utilitários e convenções utilizados para a criação de bibliotecas de classes que podem ser utilizadas por programas de aplicação escritas em várias linguagens de programação orientada a objetos, tais como C++ e Smalltalk, ou por linguagens procedurais tradicionais, tais como C e Cobol [IBM93].

SOM é uma nova tecnologia de programação orientada a objetos para construção, armazenamento, e manipulação de bibliotecas de classes. SOM suporta todos os conceitos e mecanismos normalmente associados aos sistemas orientados a objetos, incluindo herança, encapsulação e polimorfismo. Em adição o SOM possui um número de mecanismos de objetos avançados, incluindo suporte para metaclasses, três tipos de mecanismos de resolução de métodos e criação de classes dinâmicas.

SOM possui um conjunto de *frameworks*. Os *frameworks* são inter relacionamentos de objetos SOM, que são projetados para resolver problemas particulares. Estes *frameworks* são: SOM distribuído, o *framework* de repositório de interfaces, *framework* de persistência, *framework* de replicação e *framework* emissor.

A meta em comum entre o SOM e o CORBA é facilitar a interoperabilidade de objetos independentes de sua localização, a linguagem de programação em que são implementados, ou o sistema operacional ou arquitetura de *hardware* sobre o qual executam.

2.4.1 SOM Distribuído

SOM distribuído (ou DSOM) permite que programas de aplicação acessem objetos SOM em qualquer espaço de endereçamento. Ou seja, programas de aplicação podem acessar outros objetos em outros processos, mesmo sobre máquinas diferentes. O DSOM provê esta transparência de acesso de objetos remotos através de seu *Object Request Broker* (ORB). A localização e a implementação de objetos são escondidos do cliente, e os clientes acessam o objeto como se fossem locais.

2.4.2 Framework Repositório de Interface

O repositório de interface é uma base de dados, opcionalmente criada e gerenciada pelo compilador SOM, que contém todas as informações contidas na descrição IDL de uma classe de objetos. Definições de interface de classe ou assinaturas são registrados no repositório de interface. O DSOM faz uso extensivo da informação armazenada no repositório de interface. Antes que um objeto possa ser acessado remotamente pelo DSOM, é necessário registrar a interface de classe e o nome do repositório de interface. O *framework* repositório de interface consiste de onze classes definidas no padrão CORBA para acesso ao repositório de interface. O framework repositório de interface provê acesso

em tempo de execução para todas as informações contidas na descrição IDL de uma classe de objetos.

2.4.3 Framework Persistência

O *framework* persistência é uma coleção de classes SOM que provêem métodos para salvar objetos e restaurá-los mais tarde. Isto significa que o estado de um objeto pode ser preservado independete da terminação do processo no qual foi criado. Esta facilidade é usada por exemplo na construção de base de dados orientados a objetos e *spreadsheets* .

O *framework* persistência inclui as seguintes características:

- objetos podem ser armazenados em grupos;
- objetos podem ser armazenados em formatos padrões ou em formatos especiais;
- objetos de arbitrárias complexidades podem ser armazenados.

2.4.4 Framework Replicação

O *framework* replicação é uma coleção de classes SOM que permite que uma réplica de um objeto exista em múltiplos espaços de endereçamento. O *framework* replicação trata de bloquear, sincronizar e atualizar as réplicas de um objeto, garantindo ainda a consistência múltipla entre as réplicas. O *framework* replicação possui as seguintes características:

- possui bons tempos de resposta para escritores e leitores;
- tolerância a falhas contra nodos falhos e perdas de mensagens;
- regras simples de codificação para construção de objetos replicados;
- degradação suave sobre redes de longa distância;
- mínimo *overhead* quando a replicação não é utilizada.

2.4.5 Framework Emissor

O *framework* emissor é uma coleção de classes SOM que permitem aos programadores minimizar a operação do compilador SOM. O compilador SOM processa arquivos IDL que definem interfaces de classes. O *framework* emissor simplifica o processo de criação de um emissor, gerando *bindings* específicos de um arquivo IDL.

2.4.6 Adaptador de Objeto SOM (SOMOA)

O adaptador de objeto (SOMOA) é uma interface entre a aplicação servidora e o desenvolvimento em run-time do DSOM. Quando os clientes enviam requisitos para um servidor, os requisitos são recebidos e processados pelo SOMOA [IBM94].

O SOMOA trabalha juntamente com o objeto servidor para criar e resolver referências DSOM para objetos locais, e mecanismos de resolução sobre estes objetos.

O objeto SOMOA é uma implementação do Adaptador de Objeto definido pelo CORBA.

2.4.7 Linguagem de Definição de Interface do SOM (SOM IDL)

O SOM possui uma linguagem de definição de interface conhecida como SOM IDL (SOM IDL: SOM's Interface Definition Language). O SOM IDL é compatível com o padrão IDL definido pelo CORBA. O SOM IDL é utilizado na definição das interfaces de um objeto.

2.4.8 Mecanismos de Resolução de Métodos

O SOM suporta três tipos de mecanismos de resolução de métodos: *offset*, *name-lookup* e *dispatch*. Estes mecanismos são distintos pela quantidade de informação requerida sobre o objeto, pelo método e a assinatura do método conhecido no momento em que a aplicação do cliente é compilada.

- O mecanismo *offset* é padrão para invocação de um método sobre o objeto SOM. Este mecanismo é utilizado quando são conhecidos as seguintes informações em tempo de compilação: nome do método, assinatura do método e a classe que introduziu o método;
- O mecanismo *name-lookup* é apropriado quando a assinatura do método é conhecida em tempo de compilação, mas o nome da classe ou nome do método não é conhecido. Este mecanismo é menos eficiente que a resolução *offset*, mas é mais flexível, pois o método particular e o objeto sobre o qual aquele método é invocado pode ser determinado em tempo de execução;
- O mecanismo *dispatch* é o menos eficiente, mas o mais flexível entre os mecanismos de resolução. Usando-se a resolução *dispatch*, o programador pode dinamicamente construir a chamada do método de um objeto. O mecanismo *dispatch* pode ser descrito como uma interface de invocação dinâmica, sendo que uma requisição de um objeto pode ser construída e invocada em tempo real.

2.5 Conclusão

O RM-ODP provê um *framework* completo para o desenvolvimento de sistemas distribuídos abertos. O objetivo do RM-ODP é padronizar o processamento distribuído aberto, provendo uma visão ampla do sistema ao qual possa organizar as peças de um sistema ODP de uma forma coerente. Sua função não é padronizar os componentes do sistema e nem influenciar desnecessariamente a escolha da tecnologia. O CORBA e o SOM são padrões desenvolvidos por empresas privadas e têm a finalidade de prover processamento distribuído. O RM-ODP, CORBA e SOM compartilham a meta de facilitar a interoperabilidade em sistemas distribuídos.

Apesar da promessa do esforço inicial do grupo de gerenciamento de objeto em produzir um padrão industrial para sistemas de gerenciamento de objetos distribuídos, a especificação CORBA é limitada em um número de áreas:

- Interoperabilidade: o sucesso da abordagem ORB para sistemas de gerenciamento de objeto distribuído dependerá em muito da facilidade com que interações entre objetos atravessam diferentes implementações ORB [Raymond93];
- Portabilidade: uma das metas atuais dos padrões CORBA é permitir a construção de aplicações distribuídas através da síntese de objetos existentes;
- Invocações Dinâmicas: embora a interface de invocação dinâmica ofereça flexibilidade para descobrir interfaces e operações em tempo de execução, interfaces adicionais não podem ser incluídas no repositório em run-time. Sem esta capacidade, a vantagem da interface de invocação dinâmica é significativamente reduzida.

O SOM possui *frameworks* que auxiliam no desenvolvimento de sistemas distribuídos. O framework de distribuição do SOM, o DSOM obedece a arquitetura CORBA. O SOM define uma linguagem de definição que é compatível com a IDL do CORBA e possui algumas características que a diferenciam do CORBA. O repositório de interfaces segue a especificação CORBA. O adaptador de objeto do SOM é uma implementação do adaptador de objeto definido pelo CORBA.

O SOM suporta todos os conceitos e mecanismos normalmente associados com sistemas orientados a objetos, como herança, encapsulação e polimorfismo, mas inclui adicionamente suporte a metaclasses, três tipos de mecanismos de resolução de métodos e criação de classes dinâmicas, ao contrário do CORBA.

CAPÍTULO 3

SERVIÇOS DE OBJETOS SEGUNDO O PADRÃO ODP

3.1 Introdução

No modelo ODP, a unidade básica para o processamento é o objeto. Um objeto é composto de atributos e interfaces. O RM-ODP propõe uma estrutura para a criação de objetos: o *template*. Objetos são criados através de instanciações feitas ao *template*. Os pontos de vista computacional e de engenharia [RM-ODPc] propõem vários tipos de objetos e *templates*. Através dos *templates*, objetos específicos podem ser criados para prover funcionalidades específicas para o sistema. Este capítulo descreve os principais *templates* definidos pelo RM-ODP. Este conjunto de *templates* gera os objetos necessários ao desenvolvimento de sistemas distribuídos abertos.

O item 3.2 descreve os objetos segundo o ponto de vista computacional e o item 3.3 descreve os objetos segundo o ponto de vista de engenharia [Garcia94].

3. 2 Ponto de Vista Computacional

3.2.1 Template de Objeto Computacional

Um objeto computacional pode ser tanto um objeto básico computacional quanto um objeto *binding*.

3.2.1.1 Template de Objeto Básico Computacional

O objeto básico computacional é constituído de um conjunto de *templates* descrevendo interfaces, uma especificação de comportamento e um contrato de desenvolvimento. O objeto básico computacional é definido como sendo do tipo *cliente-servidor* ou *produtor-consumidor.*

O RM-ODP descreve funcionalidades para o objeto básico computacional tais como *checkpoint* de objeto (armazena as informações do objeto), remoção de objetos, remoção de interfaces, adição de interfaces, leitura e modificação dos atributos de objeto, obtenção das interfaces associadas ao objeto, desativação do objeto (*checkpoint* seguido de remoção), recuperação do objeto (utilizando as informações armazenadas pelo *checkpoint*), dentre outras.

3.2.1.2 Template de Objeto Binding

O objeto *binding* é um objeto computacional que suporta *binding* entre um conjunto de objetos computacionais. O objeto *binding* é constituído de parâmetros formais (indicam por exemplo, se foram criados a partir do cliente ou do servidor), assinaturas de interface (número, nome e tipo dos parâmetros), conjunto de *templates* descrevendo interfaces, uma especificação de comportamento e um contrato de desenvolvimento

O objeto *binding* possui métodos para encerrar um *binding* entre objetos, realizar o monitoramento do *binding* (e.g. verificando a qualidade de serviço) e métodos para estabelecimento de comunicação entre objetos.

3.2.2 Template de Interface Computacional

A interface computacional é definida como sendo um *template* de interface sinal, ou um *template* de interface operacional, ou um *template* de interface *stream*.

3.2.2.1 Template de Interface Sinal

Sinal é uma ação atômica compartilhada resultando em uma comunição unilateral entre o objeto que inicia e o objeto que responde. Sinal é definido como uma interação.

A interface sinal é uma interface no qual todas as interações são sinais. A assinatura de interface descreve as características de cada sinal.

A interface sinal é formada por uma especificação de comportamento, uma especificação de contrato e um conjunto de *templates* descrevendo sinais.

3.2.2.2 Template de Interface Operacional

A interface operacional é uma interface no qual todas as interações são operações.

Uma operação corresponde a uma interação entre o objeto cliente e o objeto servidor no qual a operação pode ser definida como sendo uma *interrogação* ou um *comunicado*. Um *comunicado* consiste de:

- uma invocação (sinal), que iniciada por um objeto cliente resulta no transporte de informação de um objeto cliente para um objeto servidor, requisitando a execução de uma função pelo objeto servidor.

Uma interrogação consiste de:

- uma invocação (sinal);

- uma terminação (sinal), que iniciada pelo objeto servidor, resulta no transporte de informação de um objeto servidor para o objeto cliente, em resposta a invocação.

No caso de interrogações, as invocações e terminações são sempre pareadas, ou seja, não existe a possibilidade de uma operação consistir de uma invocação seguida de uma seqüência de terminações.

O *template* de interface operacional possui como argumentos uma especificação de comportamento, um contrato de desenvolvimento e *templates* descrevendo sinais (invocação e terminação).

3.2.2.3 Template de Interface Stream

Uma interface *stream* é uma interface onde todas as interações são fluxos. Um fluxo é uma abstração de uma seqüência de interações, resultando no transporte de informação entre um objeto produtor e um objeto consumidor. O fluxo pode ser visto como um conjunto de sinais.

O *template* de interface *stream* é composto de uma indicação de comportamento, um contrato de desenvolvimento, a direção do fluxo e um *template* descrevendo o sinal.

3.2.2.4 Template de Sinal

O *template* de sinal possui como argumentos o nome do sinal, assinaturas de interfaces (número, nome e tipo de parâmetros) e uma indicação de causalidade (inicializando, respondendo).

3.2.3 Contrato

O contrato tem a finalidade de regulamentar a cooperação entre objetos. O contrato é formado de restrições de qualidade de serviço, gerenciamento e uso.

3.2.3.1 Restrições de Qualidade de Serviço

As restrições de qualidade de serviço (QoS[1]) em um contrato de desenvolvimento têm a finalidade de restringir e regulamentar certos parâmetros, garantindo a qualidade das informações estabelecidas por um contrato de desenvolvimento. As restrições mais comuns são:

- restrições temporais (e.g. deadlines)
- restrições de volume (e.g. vazão, banda);

[1] QoS é abreviação para Quality of Service

- restrições de dispositivos (e.g. vídeo na máquina X, câmera na máquina Y).

3.2.3.2 Restrições de Gerenciamento e Uso

As restrições de gerenciamento e uso em um contrato de desenvolvimento podem incluir restrições de localização (e.g., nodos selecionados para execução) e restrições de transparência de distribuição (e.g. selecionadas transparências de distribuição).

3.2.4 Comportamento

Comportamento é descrito por um conjunto de ações em que o objeto possa tomar parte. A seqüência de ações pode incluir ações internas. O comportamento é restrito ao desenvolvimento do sistema no qual o objeto está localizado.

3.2.5 Tipo

Tipo descreve um conjunto de regras de tipagem. Estas regras avaliam se um determinado objeto é subtipo de outro objeto. O modelo prescritivo do RM-ODP provê várias regras de tipagem [RM-ODPc].

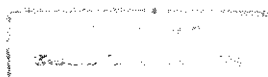
3.3 Ponto de Vista de Engenharia

3.3.1 Associação

Associação são rotas (lógicas) por onde fluem ações de gerenciamento ou comunicação. Via de regra uma associação liga uma interface de um objeto gerenciador a uma interface de um objeto gerenciado

3.3.2 Cluster

Cluster contêm um conjunto de objetos básicos de engenharia (representam objetos computacionais), associados a um gerenciador de *cluster* (veja figura 3.1). Cada membro (objeto) de um *cluster* pode ter uma interface suportando a função de gerenciamento de objeto. A interface de gerenciamento de objeto é associada a um gerenciador de *cluster*. Um *cluster* está sempre contido em uma cápsula. O objeto básico de engenharia (pertencente ao *cluster*) é associado ao núcleo, através da interface que provê a função de gerenciamento de nodo e ao gerenciador de *cluster* . O objeto básico de engenharia de um *cluster* pode ser associado a outros objetos básicos de engenharia, localizados no mesmo *cluster* ou em *clusters* diferentes. Cada gerenciador de *cluster* de uma cápsula é associado ao gerenciador de cápsula. O gerenciador de *cluster* provê uma função de



gerenciamento de *cluster* que trata de políticas de gerenciamento para os objetos pertinentes ao *cluster*.

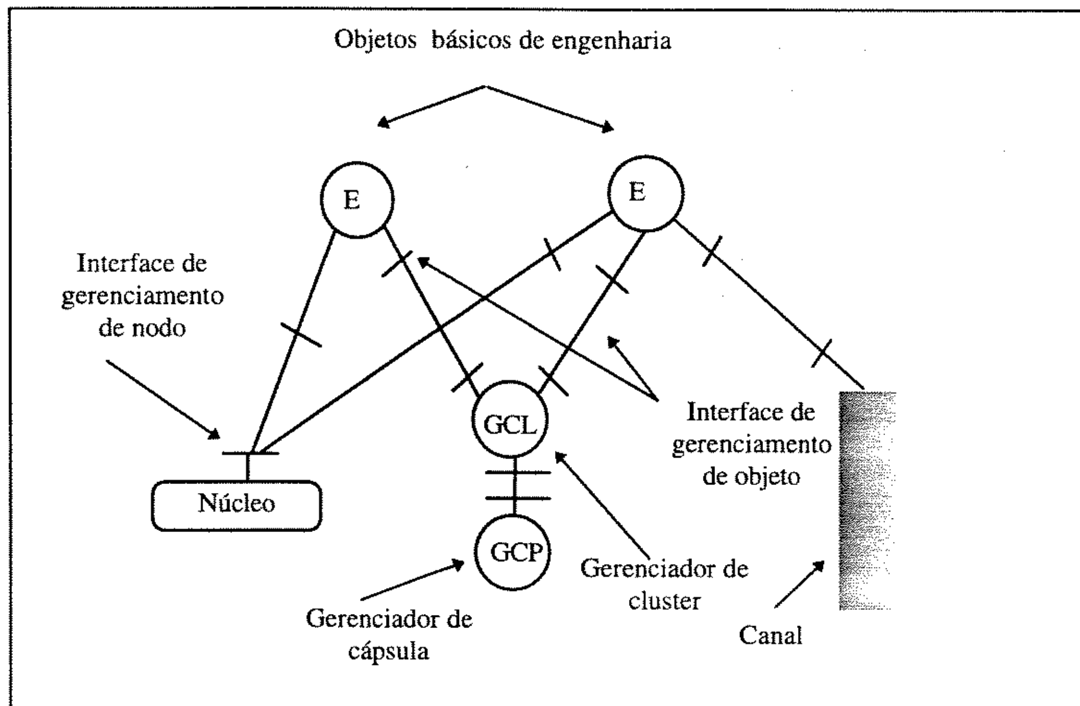


Figura 3.1 Exemplo de Estrutura Suportando o Objeto Básico de Engenharia

A função de gerenciamento de *cluster* provê funções para o *checkpoint*, recuperação, migração, desativação ou remoção de *clusters*, sendo que a função de gerenciamento de *cluster* pode suportar parte destas funções.

3.3.3 Cápsula

Cápsula é uma configuração de objetos que formam um conjunto distinto para os propósitos de encapsulação do processamento (figura 3.2). A cápsula é constituída de *clusters*, gerenciadores de *cluster* (um para cada *cluster*), objetos *stub*, *binder* e protocolo de cada canal associado a uma interface de um objeto básico de engenharia e um gerenciador de cápsula no qual os gerenciadores de *cluster* na cápsula são associados.

O gerenciador de cápsula trata de políticas de gerenciamento para os *clusters* da cápsula. A cápsula é contida em um nodo. O gerenciador de cápsula possui uma interface que provê a função de gerenciamento de cápsula. Dentro de uma cápsula, o gerenciador de cápsula é associado a cada gerenciador de *cluster*, através da interface de gerenciamento de *cluster*.

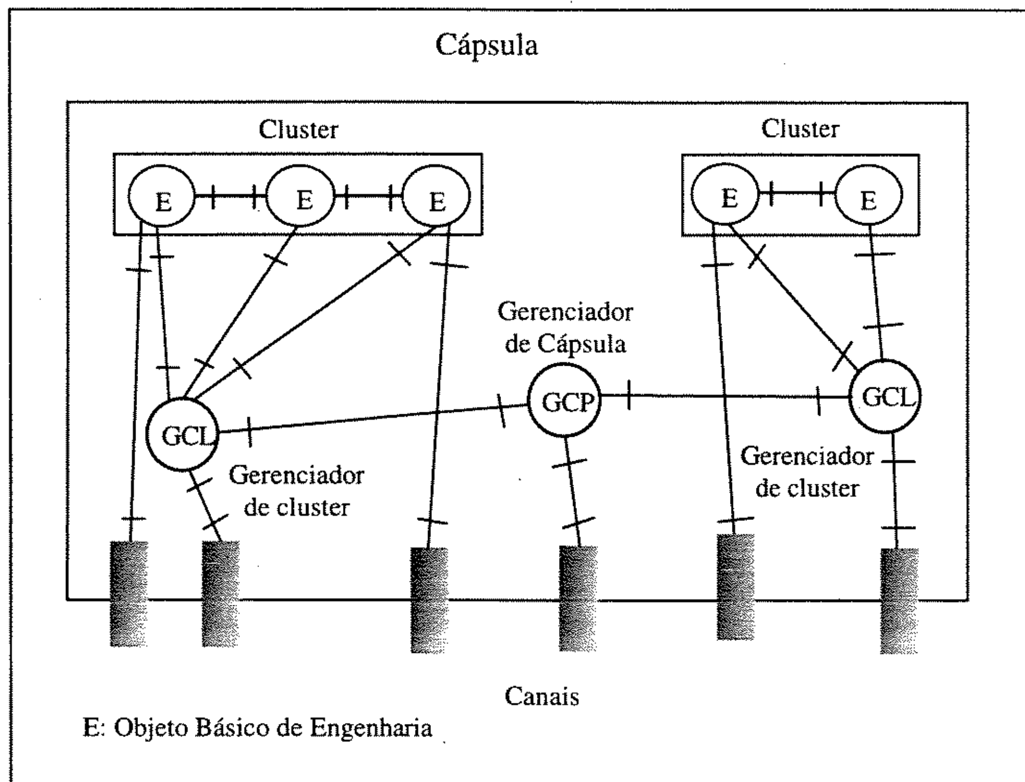


Figura 3.2 Exemplo de Estrutura de Cápsula

As funções de gerenciamento de cápsula são: instanciação de *clusters* (recuperação e reativação), *checkpoint* de todos os *clusters* de uma cápsula, desativação de todos os *clusters* da cápsula e remoção de cápsulas.

3.3.4 Nodo

Nodo consiste de um núcleo e um conjunto de cápsulas. Todos os objetos em um nodo compartilham funções de processamento , armazenagem e comunicação (figura 3.3).

O núcleo é associado a uma interface de gerenciamento de nodo, uma para cada cápsula.

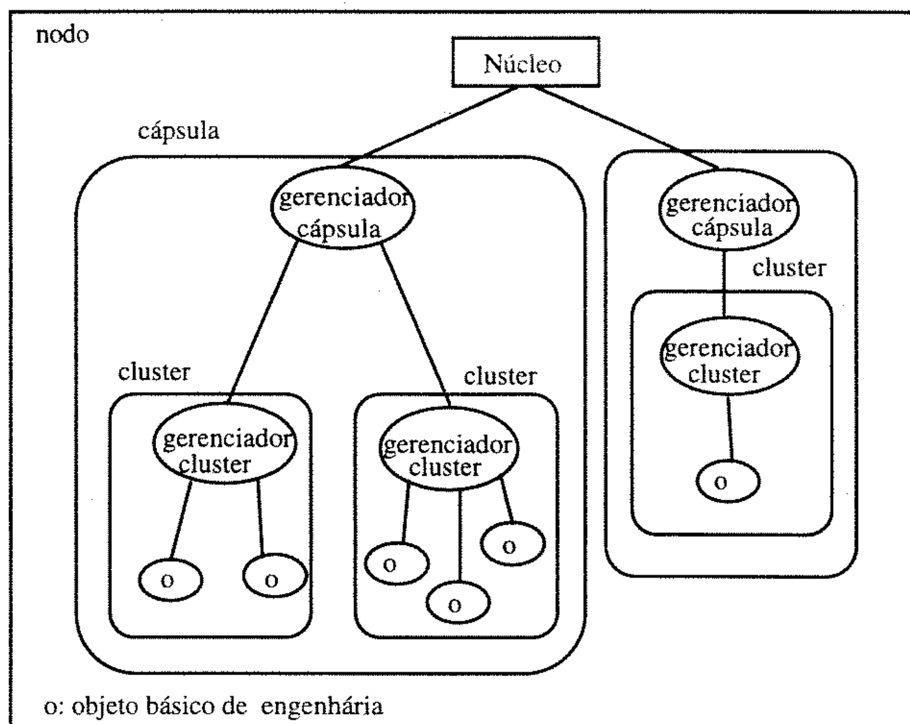


Figura 3.3 Exemplo de Estrura de Nodo

A função de gerenciamento de nodo controla o processamento, armazenagem e comunicação dentro de um nodo. A função de gerenciamento de nodo é provida pelas interfaces de gerenciamento de nodo de seu respectivo núcleo.
A função de gerenciamento de nodo trata de gerenciar *threads*, acessos a relógios e gerenciadores de tempo, criação de canais, localização de interfaces, além de criação e remoção de cápsulas.

3.3.5 Canal

Canal é uma configuração de objetos *stub*, *binder*, protocolo e interceptador interconectando um conjunto de objetos básicos de engenharia. O canal pode ser visto como um grafo acíclico, no qual os objetos *stub* são os vértices mais extremos do canal. O canal pode ser visto como uma sequência dos seguintes objetos:

- *stub*, *binder*, protocolo, protocolo, *binder* e *stub*; ou
- *stub*, *binder*, protocolo, interceptador, protocolo, *binder* e *stub*.

Esta seqüência representa uma canal uniponto (figura 3.4). Um canal multiponto (figura 3.5) pode ser visto como uma seqüência de objetos *stub, binder* e o conjunto de objetos:

- protocolo1, protocolo1, *binder e stub*;
- protocolo2, interceptador, protocolo3, *binder e stub*.

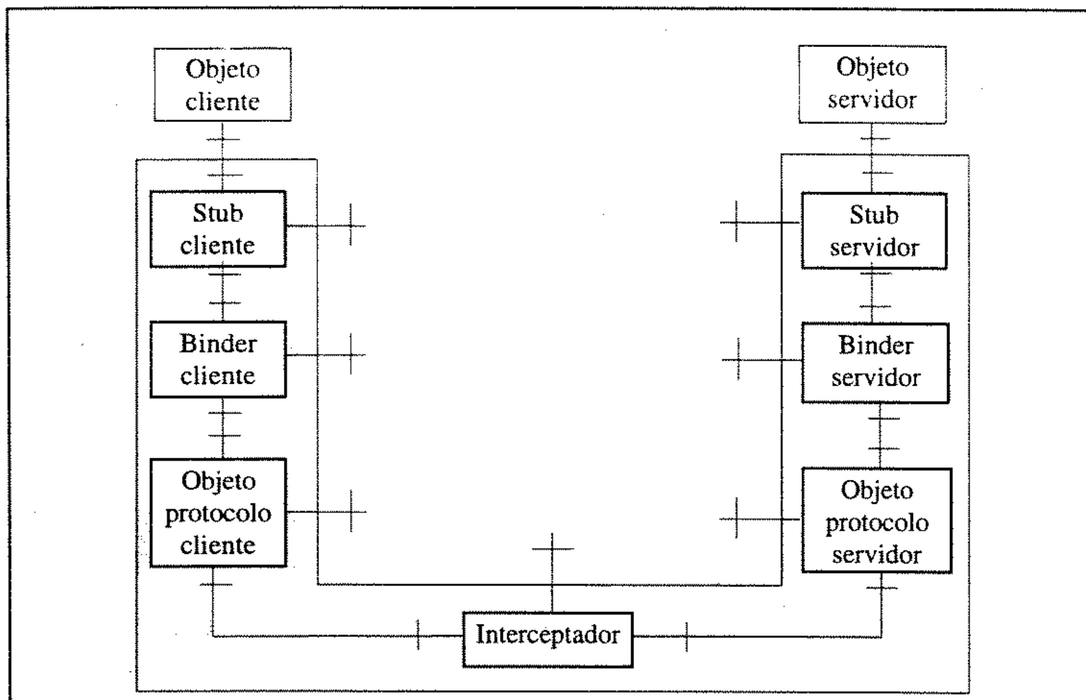


Figura 3.4 Exemplo de um Canal Cliente/Servidor

3.3.5.1 Stub

Stub é um objeto de engenharia, que interpreta as interações transmitidas pelo canal, executando transformações ou efetuando o monitoramento. Objetos básicos de engenharia que interagem através de canais são associados aos *stubs*. Em um canal, *stubs* provêem a conversão de dados. O *stub* de um canal é associado a uma interface de um objeto básico de engenharia e a uma interface do objeto *binder*. O *stub* pode ter uma interface de controle (i.e. para gerenciamento de qualidade de serviço) e pode assumir uma das seguintes formas:

- específico para a instância de interface do objeto básico de engenharia ao qual é associado;
- específico para o tipo da interface do objeto básico de engenharia ao qual é associado (entretanto o *stub* pode ser compartilhado entre um número de canais do mesmo tipo).
- genérico (i.e. não é específico a nenhum tipo de interface), tais *stubs* podem ser compartilhados entre um número de canais de diferentes tipos.

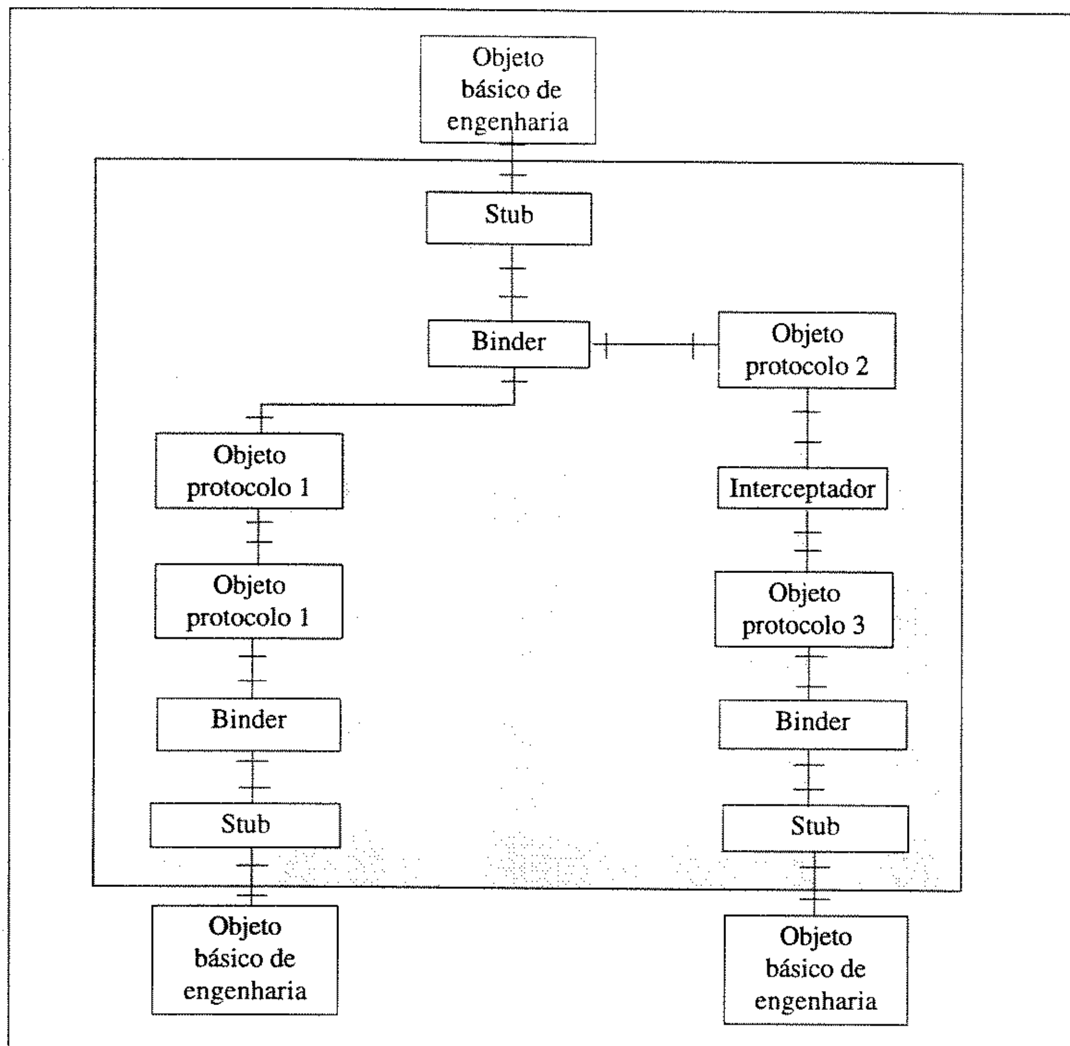


Figura 3.5 Exemplo de um Canal Cliente/Servidor com Interceptador

3.3.5.2 Binder

Binder é um objeto de engenharia que gerencia a integridade fim a fim do canal. Quando requerido, o *binder* provê transparência de relocação para monitorar as falhas de comunicação e reparar as conexões interrompidas. O *binder* pode interagir com objetos de engenharia localizados fora do canal para obtenção de dados adicionais necessários a execução de suas funções (e.g. um relocador para obtenção da localização do dado). O *binder* é associado ao *stub* e ao objeto protocolo e pode possuir uma interface de controle. A interface de controle permite mudanças na configuração do canal bem como a destruição do canal.

3.3.5.3 Protocolo

Protocolo é um objeto de engenharia pertencente a um canal, que comunica com outros objetos protocolo do mesmo canal para fins de transporte de informação.

O objeto protocolo provê funções de comunicação. Objetos protocolo podem interagir com objetos de engenharia localizados fora do canal para obtenção das informações necessárias à comunicação. Um objeto protocolo tem uma interface para interação com um *binder* e pelo menos uma interface de comunicação para interação com outros objetos protocolo (via interceptadores se requerido). Um objeto protocolo pode ter uma interface de controle. Quando objetos protocolo em um canal são de tipos diferentes, eles requerem um interceptador no qual provê a conversão de protocolo.

3.3.5.4 Interceptador

Interceptador é um objeto de engenharia associado a um canal que realiza o monitoramento ou a execução de políticas de interação entre domínios, executando transformações para mascarar as diferenças entre domínios (e.g. formatação dos dados, protocolos de comunicação, etc). O interceptador possui pelo menos duas interfaces de comunicação associadas aos objetos protocolo e pode possuir ainda uma interface de controle.

3.5 Conclusão

Os pontos de vista computacional e de engenharia do RM-ODP fornecem os elementos básicos para o desenvolvimento de sistemas distribuídos abertos. Os elementos descritos nestes pontos de vista são definidos como objetos, que são instâncias de *templates* e que formam um conjunto básico para o desenvolvimento de sistemas distribuídos abertos.

CAPÍTULO 4

PROPOSTA DE UMA INTERFACE DE PROGRAMAÇÃO PARA OS SERVIÇOS DE OBJETOS ODP

4.1 Introdução

Este capítulo descreve uma proposta de interface de programação de aplicação (Application Programming Interface - API) para uma plataforma ODP.

A interface de programação de aplicação é baseada nos objetos dos pontos de vista computacional e de engenharia. Adotou-se a metodologia OMT desenvolvida por [Rumbaugh91] para a especificação dos objetos que compõem a API.

O item 4.2 descreve a metodologia desenvolvida por Rumbaugh. O item 4.3 descreve a interface de programação de aplicação e o item 4.4 traz uma breve conclusão deste capítulo.

4.2 Object Modeling Technique (OMT)

OMT [Rumbaugh91] é uma metodologia de desenvolvimento de software baseada em modelagem de objetos do mundo real, independentemente da linguagem de programação. OMT define um conjunto de conceitos e uma notação gráfica independente da linguagem.

A metodologia OMT utiliza três tipos de modelos para descrição do sistema:

- *modelo de objeto:* descreve a estrutura dos objetos do sistema e suas relações;
- *modelo dinâmico:* descreve as interações entre os objetos do sistema;
- *modelo funcional:* descreve as transformações de dados do sistema.

A interface de programação de aplicação utiliza o modelo de objeto para especificação do sistema. O modelo de objeto é descrito abaixo.

4.2.1 Modelo de Objeto

O modelo de objeto descreve a estrutura estática dos objetos de um sistema, as relações entre os objetos, os atributos e as operações que caracterizam cada classe de objetos, sendo o modelo mais importante dentre os três modelos. O modelo de objeto contém diagramas de objetos. Um diagrama de objetos é um grafo cujos vértices são classes de objetos e cujos arcos são relações entre as classes.

4.2.2 Objetos

Objetos são definidos como conceitos, abstrações, ou entes materiais. Objetos servem a dois propósitos: promover o entendimento do mundo real e prover uma base prática para a implementação computacional. A decomposição de um problema dentro de objetos depende do julgamento e da natureza do problema. Não existe uma representação correta para isto. Todos objetos têm identidade e são distinguíveis.

4.2.3 Classes

Uma classe de objetos descreve um grupo de objetos com propriedades similares (atributos), comportamento comum (operações), relações e semânticas comuns. Objetos de uma mesma classe possuem os mesmos atributos e padrões de comportamento.

4.2.4 Diagramas de Objetos

Diagramas de objetos provêem uma notação gráfica formal para modelar objetos, classes e suas relações. Há dois tipos de diagramas de objetos: diagramas de classe e diagramas de instância. Um diagrama de classe é um esquema, modelo, ou *template* para descrição das possíveis instâncias de dados. Um diagrama de classe descreve classes de objetos. Um diagrama de instância descreve instâncias de objetos como um conjunto particular de objetos que se relacionam. Um diagrama de classe corresponde a um conjunto de diagramas de instâncias. A classe é representada através de uma caixa e instâncias de objetos são representadas através de caixas arredondadas (figura 4.1).

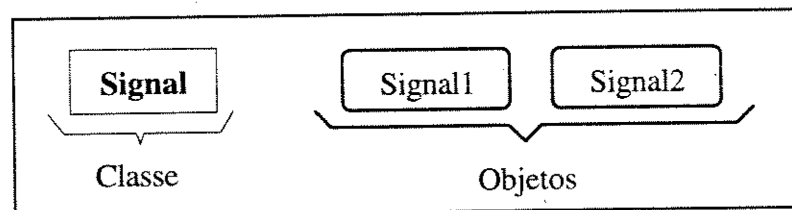


Figura 4.1 Representação Gráfica de Classe e Objetos

4.2.5 Atributos

Atributos identificam dados mantidos pelos objetos de uma classe. Diferentes instâncias de objetos podem ter valores iguais ou diferentes de um dado atributo. Cada nome de atributo é único dentro de uma classe. Atributos são listados na segunda parte da caixa classe. Cada nome de atributo pode ser seguido por informações opcionais, tais como tipo e valor *default* (figura 4.2).

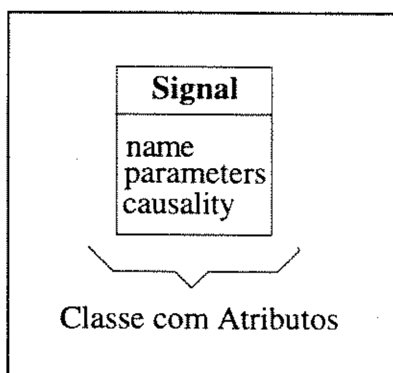


Figura 4.2 Representação Gráfica de Classe com Atributos

4.2.6 Operações e Métodos

Uma operação é uma função ou transformação que pode ser aplicada aos objetos de uma classe. Todos os objetos de uma classe compartilham as mesmas operações. A mesma operação pode ser aplicada a diferentes classes. Tal operação é dita polimórfica, isto é, a mesma operação toma diferentes formas em diferentes classes. Um método é a implementação de uma operação. Operações são listados na terceira parte da caixa (figura 4.3).

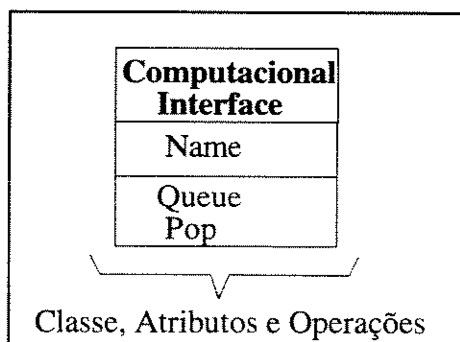


Figura 4.3 Representação Gráfica de Classe, Atributo e Operação

4.2.7 Links e Associações

Links e associações são utilizados para estabelecer relações entre as classes. Um *link* é uma conexão física ou conceitual entre instâncias de objetos. Um *link* é uma instância de uma associação. A associação descreve um grupo de *links* com estrutura e semânticas comuns. Todos os *links* de uma associação conectam objetos da mesma classe.

Associações descrevem um conjunto potencial de *links*, da mesma forma que uma classe descreve um conjunto de potenciais objetos.

Associações são freqüentemente implementadas em linguagens de programação como ponteiros entre objetos. Um ponteiro é um atributo em um objeto que contém uma referência explícita a outro objeto.

Um *link* mostra a relação entre dois ou mais objetos. Modelando-se um *link* como um ponteiro, pode-se mascarar o fato do *link* não ser parte de um objeto, mas dependente de ambos. A notação OMT para associação é representada como uma linha entre as classes.

4.2.8 Qualificador

Uma associação qualificada relaciona duas classes de objetos e um qualificador. O qualificador é um atributo especial que reduz a multiplicidade efetiva de uma associação. Uma associação qualificada pode ser considerada como uma forma de associação ternária.

O qualificador provê uma melhor semântica e melhora a visibilidade de navegação numa hierarquia de classes. Um qualificador freqüentemente é desenhado como uma pequena caixa sobre o final da linha de associação, juntamente com a classe que a qualifica. A figura 4.4 mostra um exemplo de qualificador.

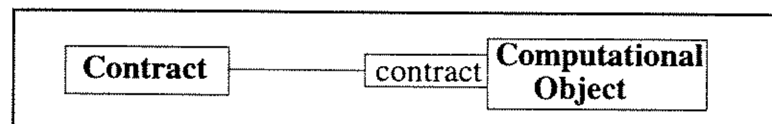


Figura 4.4 Representação Gráfica de Qualificador

4.2.9 Multiplicidade

Multiplicidade especifica como as instâncias de uma classe podem relacionar-se com as instâncias de uma classe associada. A multiplicidade restringe o número de objetos relacionados. Existem terminadores de linha especiais para indicar certos valores de multiplicidade. Um bola sólida é o símbolo OMT que representa zero ou mais. Uma bola oca indica opcional, ou seja, significa zero ou um. Uma linha sem multiplicidade indica uma associação um a um. Pode-se ainda restringir a multiplicidade utilizando-se numerações do tipo 1+ (um ou mais) no final da linha. A figura 4.5 mostra alguns exemplos de multiplicidade.

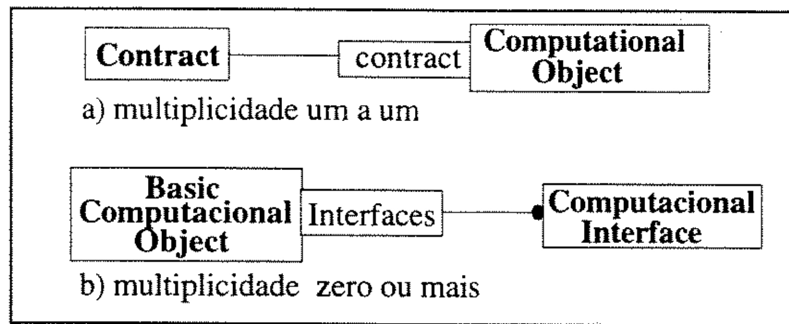


Figura 4.5 Representação Gráfica de Multiplicidade

4.2.10 Agregação

Agregação é uma relação do tipo "parte-todo" ou "parte de". Pode-se definir dois tipos de objetos: objeto componente e objeto de montagem. A propriedade mais significativa de agregação é a transitividade (i.e. A é parte de B e B é parte de C, então A é parte de C), sendo também antisimétrica (i.e. se A é parte de B, então B não é parte de A). Agregação é uma relação entre a classe de montagem e a classe componente. Uma montagem com alguns tipos de componentes correspondem a muitas relações de agregação. Define-se cada par individual como uma agregação, para que se possa especificar a multiplicidade de cada componente dentro da montagem. Esta definição enfatiza o fato de uma agregação ser uma forma especial de associação. A agregação é desenhada como uma associação, exceto pelo fato da existência de um pequeno diamante indicando a montagem final da relação. A figura 4.6 representa o modelo de agregação.

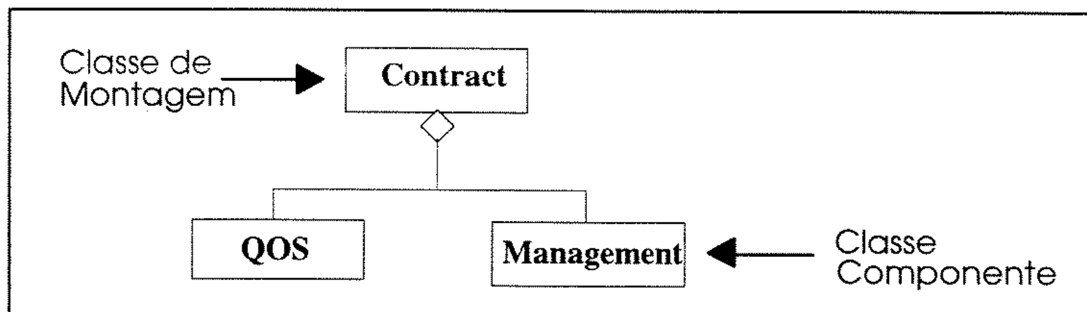


Figura 4.6 Representação Gráfica de Agregação

4.2.11 Generalização

Generalização é a relação entre uma classe e suas versões refinadas de classe. A classe sendo refinada é chamada de superclasse e a versão refinada é chamada de subclasse.

Atributos e operações comuns a um grupo de subclasses são unidos à superclasse e compartilhadas por cada subclasse. Cada subclasse herda as características de sua superclasse. A generalização é algumas vezes chamada de relação do tipo "é um", pois

cada instância de uma subclasse é uma instância da superclasse. A notação de generalização é representada através de um triângulo conectando-se a superclasse com suas subclasses. A superclasse é conectada ao ápice do triângulo através de uma linha e as subclasses são conectadas a base do triângulo através de linhas. A figura 4.7 representa um modelo de generalização.

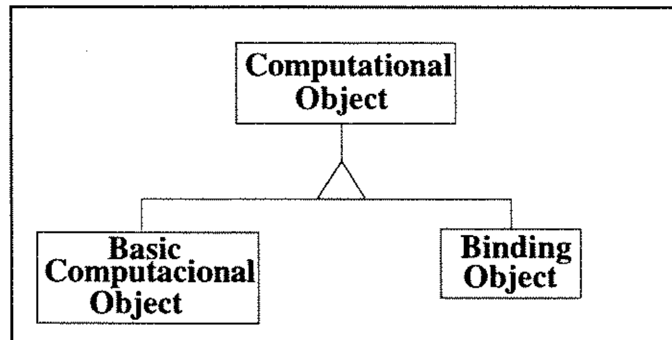


Figura 4.7 Representação Gráfica de Generalização

4.3 Interface de Programação de Aplicação

A interface de programação de aplicação (API: Application Programming Interface) é constituída de um conjunto de classes de objetos denominados *templates*. Os *templates* utilizados pela API baseiam-se nos *templates* descritos nos pontos de vista computacional e de engenharia do RM-ODP [RM-ODPc]. A API é representada através do modelo de objetos descrito na notação OMT e disponível na linguagem de programação C++ (figura 4.8[1]).

A API tem por objetivo oferecer mecanismos para o processamento distribuído aberto, provendo serviços aos usuários com as mesmas funcionalidades descritas pelo modelo de referência ODP. As funcionalidades providas pela API formam um subconjunto significativo daquelas encontradas no RM-ODP.

O primeiro conjunto de classes, *Node, Capsule, Cluster e Basic Computacional Object* são responsáveis pela estruturação da aplicação como prescrita pelo padrão ODP.

O segundo conjunto de classes, *Basic Computacional Object* e *Computational Binding Object* definem objetos, sendo especializações do *Computacional Object.*

O terceiro conjunto de classes implementa interfaces. As classes *Signal Interface, Stream Interface e Operational Interface* são especializações da classe *Computational Interface.*

A classe *Behaviour* é responsável pelo comportamento dos objetos e a classe *Contract* é responsável pela restrições de qualidade de serviço e gerenciamento (*QoS* e *Management).*

[1] A API, através da representação gráfica do modelo de objetos, é definida utilizando-se os termos equivalentes na língua inglesa.

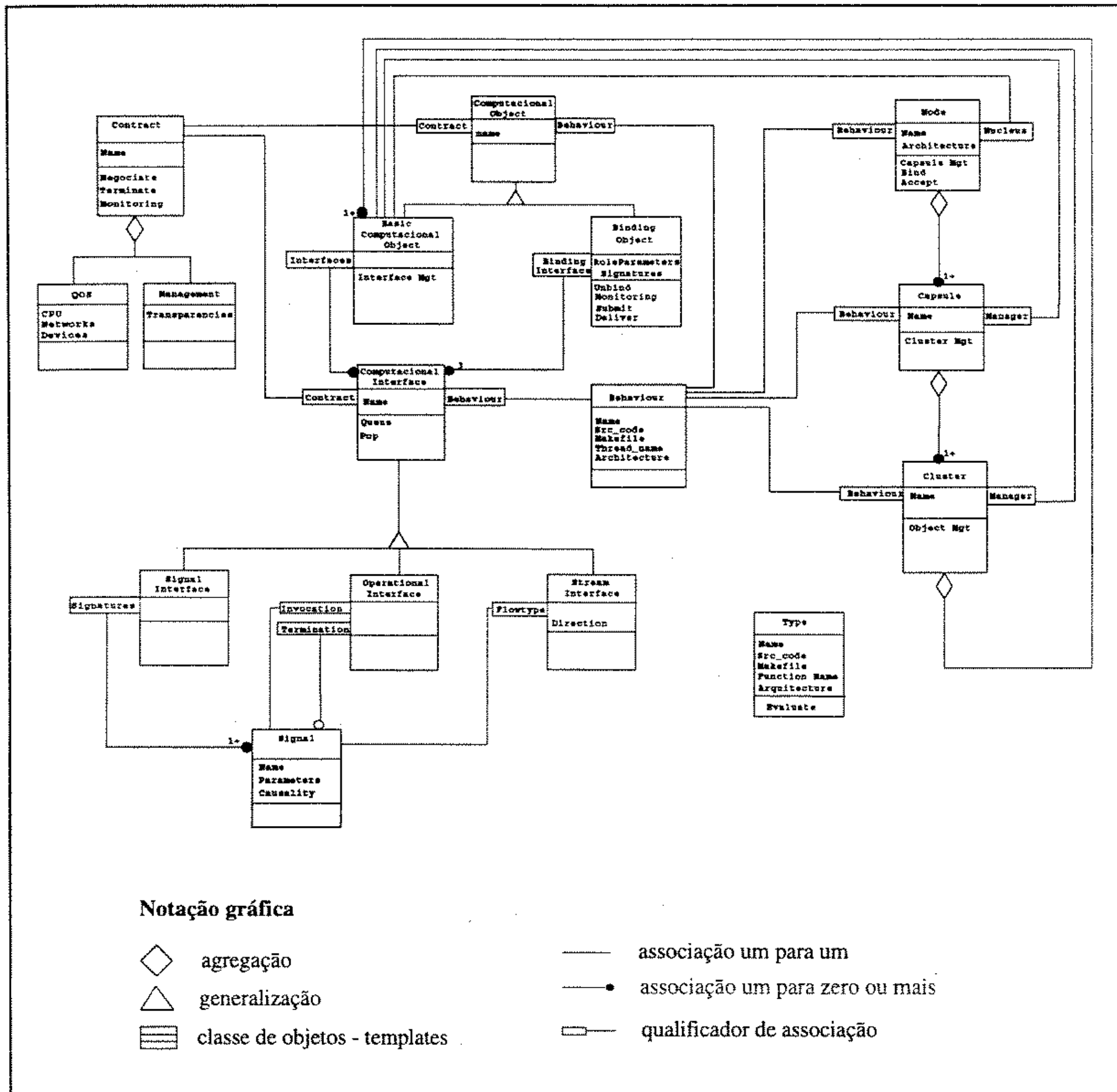


Figura 4.8 Representação Gráfica da API

O RM-ODP define o nodo como sendo composto por cápsulas, onde cada cápsula é composta de *clusters* e cada *cluster* é composta de objetos. Utilizando-se o modelo de objetos do OMT, o nodo (*Node*) é formado por uma agregação de um ou mais cápsulas (*Capsule*), onde cada cápsula é formada por uma agregação de uma ou mais *clusters* (Cluster). Cada *cluster* é formado por uma agregação de um ou mais objetos básicos de engenharia. O objeto básico de engenharia representa objetos básicos computacionais (*Basic Computacional Object*).

O nodo possui como atributos, o nome (*name*), a arquitetura em que se encontra o nodo (*architecture*), e dois qualificadores. Estes qualificadores distiguem dentre outros objetos,

o comportamento (*behaviour*) e o objeto básico computacional que implementa as funções de núcleo (nucleus - núcleo do nodo) desejados. O nodo possui como métodos:

- *makeCapsule* é responsável pela criação das cápsulas;
- *getCapsules* é responsável pela obtenção da cápsulas pertencentes ao nodo.
- *Bind* é responsável pela criação do objeto *binding* do lado cliente.
- *Accept* é responsável pela criação do objeto *binding* do lado servidor.

A cápsula possui como atributos, o nome (*name*) e dois qualificadores. Estes qualificadores são o comportamento (*behaviour*) e uma referência ao objeto básico computacional (*manager* - gerenciador de cápsula) que provê as funções de gerenciamento de cápsula. A cápsula possui como métodos:

- *makeCluster* é responsável pela criação de *clusters* em uma cápsula;
- *getClusters* é responsável pela obtenção dos *clusters* pertencentes à cápsula.

O cluster possui como atributos, o nome (*name*) e dois qualificadores: um qualificador para comportamento (*behaviour*) e outro apontando para o objeto básico computacional que provê as funções de gerenciamento de *cluster* (*manager* - gerenciador de cluster). A classe *cluster* possui como métodos:

- *makeObject* é responsável pela criação de objetos nos *clusters*;
- *getObjects* é responsável pela obtenção dos objetos pertencentes ao *cluster*;
- *checkpointCluster* é responsável por armazenar as informações pertinentes ao *cluster*;
- *recoverCluster* é responsável pela recuperação de um determinado *cluster* utilizando as informações salvas através do *checkpoint*;
- *deleteCluster* é responsável por remover o *cluster*; e
- *deactivateCluster* é responsável por realizar um *checkpoint* de *cluster*, antes de removê-lo.

O RM-ODP define o objeto computacional (*Computacional Object*) como sendo, o objeto básico computacional (*Basic Computacional Object*), e o objeto *binding* (*Binding Object*). O objeto computacional consiste de uma generalização dos objetos básico computacional e *binding*. Estes herdam as características do objeto computacional o qual contém os atributos pertinentes a ambos objetos: o nome (*name*) e o contrato (*contract*).

O objeto básico computacional possui como atributos, o nome (*name*), uma especificação de comportamento (*behaviour*), um contrato de desenvolvimento (*contract*), e *templates* descrevendo interfaces (interfaces). O *behaviour*, *interfaces* e *contract* são associações qualificadas. O objeto computacional básico possui como métodos:

- *getInterfaces* é responsável pela obtenção das interfaces pertencentes ao objeto;
- *AddInterface* é responsável por adicionar uma interface ao objeto;

- *deleteInterface* é responsável por remover uma interface;
- *readAttribute* é responsável por ler um atributo de objeto, retornando o valor do atributo;
- *modifyAttribute* é responsável por modificar o valor de atributo do objeto;
- *checkpointObject* é responsável por salvar as informações do objeto;
- *recoverObject* é responsável pela recuperação do objeto através das informações salvas pelo *checkpoint*;
- *deactivateObject* é responsável pela realização do *checkpoint* do objeto, antes da remoção do objeto;
- *deleteObject* é responsável por remover o objeto, sem realizar o *checkpoint* de objeto.

O objeto *binding* possui como atributos, o nome (*name*), uma especificação de comportamento (*behaviour*), um contrato de desenvolvimento (*contract*), parâmetros formais (*role parameters*), assinaturas de interface (*signatures*: número, nome e tipo de parâmetros) e interfaces (binding interface). Esta interface é divida em três elementos: *stub, binder* e protocolo. O objeto *binding* possui como métodos:

- *unbind* é responsável por terminar um *binding* entre objetos;
- *submit* é responsável por enviar os dados pelo canal.;
- *deliver* é responsável por receber os dados enviados através do método *submit*.

O RM-ODP define a interface computacional (*Computacional Interface*) como sendo ora a interface sinal (*Signal Interface*), ora a interface *stream* (*Stream Interface*) e ora a interface operacional (*Operational Interface*). Através do modelo de objetos, a interface computacional consiste de uma generalização das interfaces sinal, operacional e *stream*. Estas interfaces herdam as características da interface computacional a qual contém os atributos nome (*name*), contrato (*contract*) e comportamento (*behaviour*) e os seguintes métodos:

- *queue* é responsável pela comunicação entre objetos localizados no mesmo *cluster* (enviando dados entre objetos);
- *pop* é responsável em receber os dados enviados através do método *queue*.

As interfaces sinal, operacional e *stream* possuem como atributos, nome (*name*) da interface, o contrato de desenvolvimento, o comportamento e os *templates* representando sinais. O objeto sinal possui como atributos, a assinatura de interface (número, nome e tipo de parâmetros), uma indicação causalidade (inicializando, respondendo) e o nome do sinal.

A interface sinal é composta de vários sinais. No OMT, a relação entre o sinal e a interface sinal é representada através de uma associação qualificada. Esta associação qualificada possui uma indicação de multiplicidade (i.e. número "+1" sobre uma bola maciça) indicando que a interface sinal é composta de vários sinais.

A interface operacional possui dois qualificadores (invocação, terminação). O RM-ODP descreve a interface operacional como sendo uma interrogação ou um comunicado. A interrogação consiste de dois sinais (invocação e terminação) e o comunicado consiste de apenas um sinal (invocação). A multiplicidade da terminação é representada através de uma bola oca, onde a multiplicidade é opcional (0 ou 1) e a invocação possui uma multiplicidade unitária.

O contrato (*Contract*) é composto de restrições de qualidade de serviço (QoS) e gerenciamento (*Management*). Na visão do modelo de objetos, contrato é definido como uma agregação de qualidade de serviço e gerenciamento. Os métodos apresentados pela API em relação a classe contrato não foram implementados O contrato associado a qualidade de serviço e gerenciamento se encontram em fase de estudos [Guedes95]. Para fins de comunicação qualidade de serviço possui como atributos, o tamanho da mensagem, o retardo máximo da mensagem e o período de rede [Conceição95].

O comportamento é definido como um *template*. Este tem como atributos, o nome do comportamento (*behaviour*), o nome do arquivo fonte (*src_code*), o nome do arquivo *makefile* (*makefile*), o nome da função *thread* (thread_*name*) a ser executado pelo comportamento e a arquitetura (*architecture*) em que o sistema executará. Maiores detalhes serão vistos no próximo capítulo.

O tipo é definido na API como sendo um *template*. O tipo tem como atributos, o nome do tipo (*name*), o nome do arquivo fonte (*src_code*), o nome do arquivo *makefile* (*makefile*), nome da função (*function_name*) que executará a regra de tipagem e a arquitetura em que o sistema executará (*arquitecture*). O tipo não foi implementado nesta versão.

4.4 Conclusão

Através dos estudos realizados sobre o RM-ODP (Reference Model - Open Distributed Processing) [RM-ODPc] pode-se determinar os principais objetos necessários para o desenvolvimento de sistemas distribuídos abertos.

A interface de programação da aplicação baseia-se nos *templates* descritos na visão computacional e de engenharia do RM-ODP. A API é modelada através da metodologia OMT desenvolvida por Rumbaugh, utilizando-se o modelo de objetos. A notação OMT provê uma representação gráfica mais intuitiva do sistema, facilitando a comunicação com o usuário e servindo como uma documentação da estrutura do sistema.

O próximo capítulo descreverá os detalhes funcionais da API.

CAPÍTULO 5

IMPLEMENTAÇÃO

5.1 Introdução

Neste capítulo são abordados os aspectos mais relevantes a respeito da implementação dos serviços oferecidos pela interface de programação.

5.2 External Data Representation

O protocolo eXternal Data Representation (XDR) [Sun90] é utilizado na transmissão de dados entre diferentes tipos de máquina. XDR é utilizado para representar os dados em uma forma canônica. Um programa executando em uma determinada máquina utiliza o XDR para converter os dados da representação local para a representação canônica. Estes dados são transmitidos a uma outra máquina, eventualmente diferente da primeira, onde são recebidos na representação canônica e convertidos da representação canônica para a representação local. O tempo gasto na conversão ou desconversão dos dados para a representação canônica é insignificante, especialmente em aplicações de rede, onde o tempo gasto na transferência de dados é muito maior. A API utiliza a notação XDR na transmissão de requisições entre a API e a base de objetos [Gunji95]. Os tipos de dados enviados pela API, para a base de dados, são apresentados no anexo C.

5.3 Armazenamento de Templates

A API utiliza o gerenciador de banco de dados GNU gdbm [Gaumond94] para o armazenamento das informações relevantes a cada *template*. A utilização de *hashing* pelo GNU gdbm torna a busca de informações mais eficiente.
O GNU gdbm armazena pares de chave/dados em um determinado arquivo. A chave determina o *template* armazenado. Cada chave deve ser única e deve ser agrupada a um item de dados. A unidade básica de dados no gdbm é dada pela seguinte estrutura:

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

Esta estrutura permite que as chaves e itens de dados possam ter tamanhos arbitrários. O único inconveniente é que as informações devem ser serializadas antes do armazenamento e desserializadas após a recuperação da informação. O gdbm permite que uma aplicação tenha múltiplas bases de dados abertas ao mesmo tempo. Quando uma aplicação abre uma base de dados gdbm, o acesso ao dados é designada a ser de leitura ou de escrita. Muitos leitores podem abrir a base de dados simultâneamente. Contudo, somente um único escritor poderá abrir a base de dados. Leitores e escritores não podem abrir a base de dados gdbm ao mesmo tempo. O GNU gdbm não suporta o processamento de transações. O núcleo do sistema foi encarregado em fornecer esta facilidade [Fapesp95], tornando-se responsável em intermediar as chamadas de funções do gerenciador da base de dados gdbm.

Criou-se uma classe chamada *template* para o armazenamento destas informações.

5.3.1 Descrição da Estrutura de Armazenamento dos Templates

O *template* possui um conjunto de informações necessárias a criação dos respectivos objetos. As informações dos *templates* são armazenados através do gerenciador de banco de dados: o GNU gdbm. A identificação dos *templates* no GNU gdbm se dá através da chave de identificação. A base de objetos utiliza as informações dos respectivos *templates* para execução de serviços. Os parâmetros de cada *template* que são armazenados no banco de dados GNU gdbm, encontram-se no anexo B. Um exemplo dos dados a serem serializados para armazenagem no arquivo gdbm é apresentado abaixo.

O *template* de objeto básico computacional possui um conjunto de informações que são necessárias na criação do objeto. Estas informações devem ser serializadas em uma ordem pré-estabelecida, antes de serem armazenadas. As informações armazenadas são utilizadas pela base de objetos e são descritas abaixo:

- nome (char*)
- chave do contrato (contract) (char*)
- chave do comportamento (behaviour) (char*)
- número de interfaces (int)
- chave da interface 1 (char*)
- :
- chave da interface N (char*)
- número de atributos (int)
- nome do atributo 1 (char*)
- tipo do atributo 1 (int)
- valor do atributo 1 (char, int, double ou float)
- :
- nome do atributo N (char*)
- tipo do atributo N (int)
- valor do atributo N (char, int, double ou float)

5.4 Comportamento (BEHAVIOUR)

O comportamento define ações que são executadas por alguma abstração do ODP (nodo, cápsula, *cluster*, objeto e interface) e é codificado pelo usuário da aplicação. O comportamento é uma função que é executada por uma *thread*. A limitação da execução da função como uma *thread* é o fato da função só poder aceitar um único parâmetro inteiro (para a SUN).

O comportamento foi definido no capítulo 4 como sendo composto de vários atributos. O atributo *makefile* indica o arquivo *make* a ser compilado com a aplicação, possuindo comandos para compilação de um determinado código fonte. Este código fonte possui uma função que ao ser compilado, será executado na forma de uma *thread*. Portanto, o atributo *thread* corresponde ao nome da função definida pelo código *src_code*. O atributo *architecture* será empregado na execução do comando *make*, especificando a arquitetura no qual o sistema executará. Para isto, o código fonte deverá ser adequado para execução na determinada arquitetura.

O comportamento é utilizado pela base de objetos durante o processo de instanciação de objetos.

5.5 Métodos Bind e Accept

Os métodos *Bind* e *Accept* são responsáveis pela criação de objetos de *binding* do lado cliente e do lado servidor. Estes objetos são utilizados na comunicação entre objetos localizados em diferentes *clusters*. Após a criação destes objetos, ocorre a interação entre eles (criação do canal de comunicação).

O método *Bind* envia um pedido de criação de canal para a base de objetos (cliente), passando o nome do serviço remoto e os nomes dos *templates* de *stream* e *Accept* dentre outros. Enquanto isto, pelo lado servidor, o método *Accept* é executado. O método *Accept* envia o nome do serviço, dentre outros parâmetros, para a base de objetos (servidor).

Com o envio do método *Accept*, a interação entre os objetos cliente e servidor é concluída.

A base de objetos do lado cliente retorna para a API, o identificador do objeto que compõe o canal neste lado. De forma análoga, a base de objetos do lado servidor retorna para API, o identificador do objeto que compõe o canal neste lado.

O retorno dos identificadores de objeto indicam a criação do canal e a interação entre os objetos pode-se iniciar.

5.6 Métodos Queue e Pop

Os métodos *Queue* e *Pop* estão definidos na interface sinal, operacional e *stream*. Estes métodos são utilizados na comunicação entre objetos localizados no mesmo *cluster*. O

objeto cliente requisita (Queue) à base de objetos o envio de informações para um objeto servidor. Caso o envio de informações se concretize, a base de objetos retornará para a API, uma indicação de sucesso. Caso contrário, a base de objetos retornará uma indicação de fracasso.

O objeto servidor, através do método *Pop,* requisita para a base de objetos as informações enviadas pelo cliente. A base de objetos retornará para a API, o identificador de interface pelo qual os dados foram enviados e as informações enviadas pelo cliente.

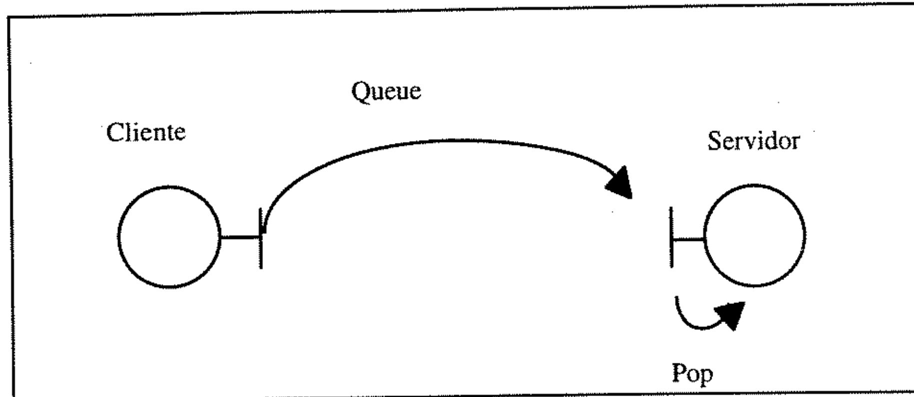


Figura 5.1 Uso dos Métodos Queue e Pop

5.7 Métodos Submit, Deliver e Unbind

Os métodos *Submit* e *Deliver* fazem parte do objeto de *binding*. Estes métodos são utilizados na comunicação entre objetos localizados em diferentes *clusters*.

Os métodos *Submit* e *Deliver* só podem ser utilizados no caso de existirem os objetos de *binding* (*Bind e Accept*) e funcionam da mesma forma que os métodos *Queue e Pop.*

O método *Unbind* é responsável em terminar um *binding* entre objetos. O método *Unbind* requisita a base de objetos o encerramento do *binding*. Entre os parâmetros enviados pela API, constam o identificador do objeto de *binding* (*Bind*). Através deste objeto, a base de objetos encerra o *binding*.

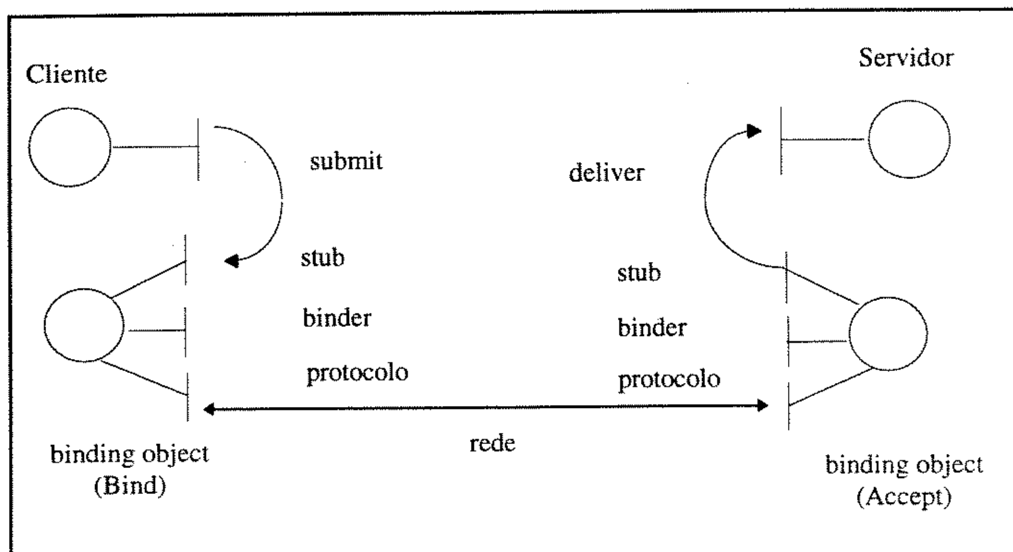


Figura 5.2 Uso dos Métodos Submit e Deliver

5.8 Métodos readAttribute e modifyAttribute

Os métodos *readAttribute* e *modifyAttribute* são utilizados pelo objeto básico computacional. O método *readAttribute* requisita para a base de objetos, o valor de um determinado atributo. Entre os parâmetros enviados pela API para a base de objetos constam o nome do atributo e o tipo de atributo. A base de objetos utiliza as informações enviadas pela API para localização do respectivo atributo. Localizando-se o atributo, a base de objetos retorna para a API, o seu valor.

O método *modifyAttribute* modifica um determinado atributo do objeto. A API requisita a base de objetos a mudança do valor do atributo. Isto é feito através do envio do nome do atributo, tipo de atributo e o novo valor do atributo, dentre outros parâmetros enviados pela API. Através das informações enviadas pela API, a base de objetos localiza o atributo e modifica o seu valor. Se a modificação ocorrer, a base de objetos retornará para a API, uma indicação de sucesso. Caso contrário, retornará uma indicação de fracasso.

5.9 Métodos addInterface e deleteInterface

O métodos *addInterface* e *deleteInterface* são métodos do objeto básico computacional. O método *addInterface* adiciona uma determinada interface a um respectivo objeto. A API requisita à base de objetos a adição de uma determinada interface para um respectivo objeto. Dentre os parâmetros enviados pela API, constam o identificador de objeto (no qual a interface será adicionada) e o nome do *template* de interface (descreve a interface a ser criada e adicionada ao objeto). Caso a adição se concretize a base de objetos é responsável por retornar para a API o identificador da interface adicionada.

O método *deleteInterface* remove uma interface de um determinado objeto. Dentre os parâmetros enviados pela API para a base de objetos, constam o identificador de objeto (no qual se encontra a interface) e o identificador de interface. A base de objetos utiliza as informações enviadas pela API para localização e remoção da interface. Caso a remoção se concretize a base de objetos é responsável por enviar uma indicação de sucesso. Caso contrário, retornará uma indicação de fracasso.

5.10 Métodos makeObject, makeCluster e makeCapsule

O método *makeObject* (*makeCluster ou makeCapsule*) é utilizado na criação de objetos (cluster ou cápsula).

O método *makeObject* requisita à base de objetos a criação de um objeto em um determinado *cluster*. Dentre os parâmetros enviados para a base de objetos, constam o

identificador de *cluster* e o nome do *template* de objeto a ser criado. A base de objetos é responsável por retornar para a API o identificador do objeto criado.

O método *makeCluster* requisita à base de objetos a criação de um *cluster* em uma determinada cápsula. A API verifica através do *portmapper* [Fapesp95] a localização da cápsula a ser utilizada no processo de criação do *cluster*, requisitando para a base de objetos da respectiva cápsula a criação do *cluster*. A base de objetos é responsável por retornar para a API o identificador do *cluster* criado.

O método *makeCapsule* é responsável em instanciar a base de objetos em um determinado nodo, enviando como parâmetros, o nome do *template* de cápsula e o identificador global da cápsula. Pode-se dizer que a base de objetos é a própria cápsula. O identificador global é fornecido pelo núcleo [Fapesp95].

Após a criação da cápsula, a base de objetos é responsável em incluir ao *portmapper*, o identificador da cápsula criada, bem como o seu *port* e *host* de comunicação.

5.11 Métodos getInterfaces, getObjects, getClusters e getCapsules

Os métodos *getInterfaces*, *getObjects*, *getClusters* e *getCapsules* retornam os identificadores de interface, de objeto, de *cluster* e de cápsula. Estes identificadores são armazenados em listas e podem ser acessadas pelo usuário.

- genList L_CAPSULES;
- gentList L_INTERFACES;
- genList L_CLUSTERS;
- genList L_BASIC_OBJECTS;

O método *getCapsules* retorna os identificadores de cápsula associados ao nodo. Os identificadores de cápsula são obtidos através de consulta realizada ao *portmapper*. O identificadores de cápsula são armazenados na lista L_CAPSULES.

O método *getClusters* retorna os identificadores de *cluster* associado a uma determinada cápsula. A API solicita para base de objetos, os *clusters* associados a cápsula. O identificadores retornados são armazenados na lista L_CLUSTERS. Similarmente, *getObjects* retorna na lista L_BASIC_OBJECTS os objetos pertencentes a um determinado *cluster*.

O método *getInterfaces* envia entre os seus parâmetros, um identificador de objeto. Através deste identificador, a base de objetos localiza o respectivo objeto, retornando para a API, os identificadores de interfaces associados ao objeto. O identicadores de interfaces (objetos) são armazenados na lista L_INTERFACES.

5.12 Métodos checkpoint<X>, recover<X>, delete<X> e deactivate<X>

Os métodos *checkpoint*<X>, *recover*<X>, *delete*<X> e *deactivate*<X> são utilizados pelo *cluster* e pelo objeto básico computacional (<X> representa *Cluster ou Object*, por exemplo *deleteObject*). Ao se realizar um *checkpoint*<X>, a API envia para a base de objetos, entre os seus parâmetros, o nome do *template* (chave de identificação para armazenagem e recuperação das informações relevantes ao *cluster* ou objeto armazenadas GNU gdbm) e o identificador de *cluster* ou objeto que terá as suas informações armazenadas. A base de objetos é responsável por retornar para API uma indicação de sucesso ou de fracasso.

O método *deactivate*<X> envia para a base de objetos, o nome do *template* (i.e. a chave utilizada no armazenamento das informações do *cluster* ou objeto no banco de dados GNU gdbm) dentre outros. Após a armazenagem das informações, o *cluster* ou objeto é removido. A base de objetos é responsável por retornar para a API uma indicação de sucesso ou de fracasso.

O método *recover*<X> envia para a base de objetos, o nome do *template* e o identificador do respectivo *cluster* ou objeto para recuperação, dentre outros parâmetros. A base de objetos utiliza as informações enviadas pelo método *recover*<X> para recuperação do *cluster* ou objeto. A base de objetos é responsável por enviar para a API uma indicação de sucesso ou de fracasso.

O método *delete*<X> remove um determinado *cluster* ou objeto. Dentre os parâmetros enviados para a base de objetos pelo método *delete*<X>, encontra-se o identificador de *cluster* ou objeto. Esta informação é utilizada pela base de objetos para localização e remoção do respectivo *cluster* ou objeto. A base de objetos é responsável por retornar como parâmetro, uma indicação de sucesso ou de fracasso.

5.13 Conclusão

A API provê um conjunto de métodos para criação, agrupamento, gerenciamento e comunicação. Estes métodos são responsáveis pelo desenvolvimento de sistemas distribuídos abertos. A API utiliza o GNU gdbm para o armazenamento dos *templates*. As informações armazenadas no banco de dados GNU gdbm são importantes para a criação dos respectivos objetos. A base de objetos oferece um conjunto de serviços para a API. A interação existente entre a base de objetos e a API é feita através do protocolo de comunicação *reliable unicast* [Conceição95]. Os dados são convertidos para o padrão XDR antes de serem enviados para a base de objetos, tornando-os independentes da arquitetura das máquinas envolvidas na comunicação.

CAPÍTULO 6

EXEMPLO DE UTILIZAÇÃO

6.1 Introdução

Neste capítulo, serão abordados alguns exemplos de utilização da interface de programação de aplicação (API). A API é uma biblioteca de funções implementada em uma linguagem de programação orientada a objetos (C++). Esta é compilada e ligada ao programa principal da aplicação.

6.2 Criação de um Cluster

A criação de um *cluster* se dá através do método *makeCluster*. Este método tem como parâmetros de entrada:

- o identificador global da cápsula (unsigned long);
- o nome do *template* de *cluster* (char *).

O identificador global da cápsula indica em qual cápsula será criado o *cluster*.

A API consulta o *portmapper* para localização da cápsula no qual o *cluster* será criado.

Exemplo:

```
# include "api.hpp"

CAPSULE cp;
unsigned long clusterid;
unsigned long capsuleid;
char * cluster_template;

clusterid=cp.makeCluster(capsuleid,cluster_template);
```

Recebida esta requisição, a API envia para a base de objetos um conjunto de parâmetros, em uma ordem pré estabelecida. Estes dados são convertidos para o formato XDR antes de serem enviados para a base de objetos.

Os dados enviados para a base de objetos possuem as seguintes informações:

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

Após o envio do pedido de criação de *cluster*, a API fica aguardando a criação do objeto pela base de objetos. Com a criação do *cluster*, a base de objetos retorna para a API, o identificador global do *cluster* criado. Este identificador é enviado pela base de objetos no formato XDR. A API então converte o identificador global para o formato local. O identificador global do *cluster* é retornado ao usuário através do parâmetro de retorno.

6.3 Desativação de um Cluster

A desativação de um *cluster* se dá através do método *deactivateCluster*. Este método tem como parâmetros de entrada:

- o identificador global do *cluster* a ser desativado (unsigned long);
- o nome do *template* de *cluster* a ser utilizado para armazenamento das informações do respectivo *cluster* (char *).

Exemplo:

```
# include "api.hpp"

CAPSULE cp;
unsigned long clusterid;
unsigned long capsuleid;
char * cluster_template;.

clusterid=cp.makeCluster(capsuleid,cluster_template);

int resp;
char * deactivate_cluster_template;
```

```
CLUSTER cl;

resp= cl.deactivateCluster(clusterid,deactivate_cluster_template);
if (resp==0)
cout<<" Desativação do Cluster Completado";
else cout<<" Nao foi Concluído o Desativação do Cluster";
```

A API verifica em qual cápsula o *cluster* está associado. Após esta verificação, a API consulta o *portmapper* à procura da respectiva cápsula. Encontrada a cápsula, a API envia os dados para aquele *host* e *port*, invocando os serviços da base de objetos.

A API envia para a base de objetos um conjunto de parâmetros, em uma ordem pré estabelecida. Estes dados são convertidos para o formato XDR antes de serem enviados para a base de objetos.

Os dados enviados para a base de objetos possuem as seguintes informações:

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

Após o envio do pedido de desativação de *cluster*, a API fica aguardando o parâmetro de retorno. A base de objetos retorna para a API, uma indicação de sucesso ou de fracasso, enviado no formato XDR. A API então converte o parâmetro de resposta do formato XDR para o formato local, retornando o parâmetro em seguida para o usuário.

6.4 Criação de um Canal

A criação de um canal se dá através da instanciação do método *Bind* e *Accept*. O método *Bind* tem como parâmetros de entrada:

- identificador de interface (unsigned long);
- nome do serviço remoto (char *);
- *timeout* para criação do canal (int);
- nome do *template* do lado cliente (char *);
- nome do *template* do lado servidor (char *);

Exemplo:

```
# include "api.hpp"

NODE no;
unsigned long interfaceid;
char * servico_remoto;
int timeout;
char * bind_template, accept_template;
unsigned long bindid;

bindid=no.Bind(interfaceid, servico_remoto, timeout, bind_template,    accept_template);
```

A API verifica em qual cápsula o objeto de *binding* será associado. A API consulta o *portmapper* para procurar a localização da cápsula. Encontrada a cápsula, a API envia os dados para aquele *host* e *port* definidos no *portmapper*, invocando os serviços da base de objetos.

A API envia para a base de objetos um conjunto de parâmetros, em uma ordem pré estabelecida. Estes dados são convertidos para o formato XDR antes de serem enviados para a base de objetos.

Os dados enviados para a base de objetos possuem as seguintes informações:

- 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 *)

Após o envio dos dados para a base de objetos (para que o canal de comunicação seja criado), a API fica aguardando o parâmetro de retorno. A base de objetos retorna para a API, um identificador global do canal. Este é enviado no formato XDR. A API converte o parâmetro de resposta do formato XDR para o formato local, retornando o parâmetro em seguida para o usuário.

O método *Accept* tem como parâmetros de entrada:

- identificador de interface (unsigned long);
- nome do serviço remoto (char *);
- timeout para criação do canal (int);

Exemplo:

```
# include "api.hpp"

NODE no;.
unsigned long interfaceid;
char * servico;
int timeout;
unsigned long acceptid;

acceptid=no.Accept(interfaceid, servico, timeout);
```

A API verifica em qual cápsula o objeto de *binding* será associado. Após a localização da cápsula, a API envia os dados para aquele *host* e *port*, invocando os serviços da base de objetos.

A API envia para a base de objetos um conjunto de parâmetros, em uma ordem pré estabelecida. Estes dados são convertidos para o formato XDR antes de serem enviados para a base de objetos.

Os dados enviados para a base de objetos possuem as seguintes informações:

- identificação do pedido (int);
- *host* para envio de resposta (char *);
- *port* para envio de resposta (int);
- nome do serviço (char *);
- *timeout* para criação do canal (int);

Após o envio dos dados para a base de objetos, a API fica aguardando o parâmetro de retorno. A base de objetos retorna para a API um identificador global do objeto *binding* (Accept). Este é enviado no formato XDR. Através do envio do identificador do objeto de *binding*, fica-se sabendo que os objetos *binding* do lado cliente e do lado servidor estão em interação (isto é, criou-se um canal de comunicação entre os objetos). A API

converte o parâmetro de resposta do formato XDR para o formato local, retornando o parâmetro em seguida para o usuário.

6.5 Envio de dados pelo Canal Sinal

O envio de dado se dá através do método *Submit*. Este método tem como parâmetros de entrada:

- o identificador global do objeto de ***binding*** (Bind) (unsigned long)
- dados do sinal:
  - ◊ nome do sinal (char *);
  - ◊ número 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).

Exemplo:

```
# include "api.hpp"

NODE no;
unsigned long interfaceid;
char * servico_remoto;
int timeout;
char * bind_template;
char *  accept_template;
unsigned long bindid;

bindid=no.Bind(interfaceid, servico_remoto, timeout, bind_template, accept_template);

BIND_OBJECT  binding;
int resp;
```

```
struct_signal_data dados;

resp= binding.Submit(bindid, dados);
if (resp==0)
cout << "Os dados do tipo sinal foram enviados pelo canal")
else cout << "Os dados nao foram enviados");
```

A API verifica em qual cápsula o objeto de *binding* está associado. Após a verificação da cápsula no qual o objeto está associado, a API consulta o *portmapper* a procura da respectiva cápsula. Encontrada a cápsula, a API envia os dados para aquele *host* e *port*, invocando os serviços da base de objetos.

A API envia para a base de objetos um conjunto de parâmetros, em uma ordem pré estabelecida. Estes dados são convertidos para o formato XDR antes de serem enviados para a base de objetos.

Os dados enviados para a base de objetos é descrita abaixo:

- identificação do pedido (int);
- *host* para envio de resposta (char *);
- *port* para envio de resposta (int);
- identificador de um objeto de *binding* (*Binding*) (unsigned long);
- dados (os mesmos descritos acima);

Após o envio do pedido de *Submit*, a API fica aguardando uma indicativa de sucesso ou de fracasso, enviado no formato XDR. A API então converte o parâmetro de resposta do formato XDR para o formato local, retornando o parâmetro em seguida para o usuário.

6.6 Conclusão

Neste capítulo foram mostrados exemplos de utilização da interface de programação de aplicação. Este capítulo não tem como objetivo demonstrar todos o métodos fornecidos pela interface programação, mas sem exemplificar como alguns destes métodos são utilizados.

CAPÍTULO 7

CONCLUSÕES E TRABALHOS FUTUROS

Devido ao rápido crescimento e desenvolvimento de sistemas distribuídos, houve a necessidade de se padronizar o processamento distribuído aberto. O modelo de referência ODP provê suporte a distribuição, interconexão e portabilidade.

O modelo de referência ODP abrange os aspectos necessários ao desenvolvimento de sistemas distribuídos abertos, especificando quais funcionalidades devem estar disponíveis para a aplicação e não como estas funcionalidades devam ser implementadas.

A complexidade dos sistemas distribuídos é tratada através do processo de abstração do sistema. O modelo de referência ODP utiliza pontos de vista para abstrair um sistema distribuído, abrangendo desde o ponto de vista empresarial, até o ponto de vista tencnológico.

O ODP, por se tratar de uma documentação complexa, exigiu uma atenção especial para que se pudesse entender o modelo.

Os pontos de vista computacional e de engenharia fornecem os elementos básicos para o desenvolvimento de sistemas distribuídos abertos, dando suporte a distribuição, interconexão e portabilidade.

Através dos estudos realizados nos pontos de vista computacional e de engenharia, determinou-se o conjunto básico de elementos para o desenvolvimento de uma interface de programação que visasse a criação de objetos, o gerenciamento de objetos e a comunicação entre objetos.

A metodologia OMT para especificação de sistemas foi de grande valia, ajudando na modelagem do sistema. A grande vantagem deste modelo é o fato de ser orientado a objetos, facilitando a modelagem, além de fornecer um modelo gráfico da especificação do sistema, facilitando o entendimento do sistema.

A principal contribuição deste trabalho foi traduzir os elementos do modelo ODP numa estrutura computacional de acesso a estes elementos. Enfocou-se nos elementos de gerenciamento e comunicação, sem dúvida os mais importantes do modelo ODP.

7.1 Trabalhos Futuros

Atualmente a comunicação entre a API e a base de objetos se faz via protocolo *request/replay*. A API deve conhecer de antemão a localização da base de objetos. Numa próxima fase, uma infraestrutura desenvolvida a partir do Orbix (produto comercial desenvolvido pela IONA Technologies) intermediará esta comunicação.

Devido ao fato da interface de aplicação ser modelada através da metodologia OMT, pode-se sintetizar a API através de ferramentas CASE, como o LOV-OMT (produto comercial desenvolvido pela Verilog Inc.) disponível no LCA.

Regras de tipagem para as assinaturas de interface, funções de negociação e monitoramento de contrato, políticas de segurança, migração e repositório podem ser implementados após um estudo mais aprofundado sobre elas.

BIBLIOGRAFIA

[Berets93] - Berets, J. C. et al., *"Introduction to Cronus"*, BBN Systems and Technologies, report No 6986, version 3.0, jan. 1993.

[Conceição95] - Conceição, L. *"Plataforma Multiware: Serviços de Comunicação"* , tese de mestrado a ser defendido em agosto de 1995, FEE-UNICAMP.

[Fapesp95] - Projeto Temático FAPESP 92/3507-0, *"Plataforma Multiware: Serviços ODP"*, abr. 1995.

[Garcia] - Garcia, C. M., *"Tese de Mestrado - Uma Proposta para Modelagem de Funções de Gerenciamento para o Processamento Distribuído Aberto"*, DCC-IMECC-UNICAMP, dezembro 1994.

[Gaumond94] - Gaumond, P., Nelson, P.A., *"GNU dbm Manual - A Database Manager"*, edition 1.4.1, Free Software Foundantion, 1994.

[Guedes95] - Guedes, L.A., *"Estudos Preliminares sobre Qualidade de Serviço em Sistemas Multimídia"* relatório técnico, DCA-FEE-UNICAMP, julho 1995.

[Gunj95] - Gunji,T., *"Tese de Mestrado - Plataforma Multiware: Serviços de Objetos"*, DCA-FEE-UNICAMP, a ser defendida em agosto de 1995.

[IBM93] - IBM, *"SOMobjects Developer Toolkit: An Overview"*, Version 2.0, Jun. 1993.

[IBM94] - IBM, *"SOMobjects: A Practical Introduction to SOM and DSOM"*, International Technical Support Centers, Jul. 1994.

[Kilov93] - Kilov, H., *"Precise Specification of Behaviour in Object-Oriented Standartization Activities"*, Computer Standards & Interfaces 15 , p. 275-285,1993.

[Lento95] - Lento, L.O.B, Madeira, E. R. M., *"Um Esquema para Acessar Objetos em Ambientes Distribuídos"*, 13º Simpósio Brasileiro de Redes de Computadores,1995.

[Mendes94] - Mendes, M. J. et. al., *"Plataforma Multiware: Projeto e Desevolvimento da Camada Middleware"*, relatório técnico, DCC-IMECC-UNICAMP, 1994.

[Mello95] - Mello, A.M.V, *"Um Protótipo de Negociador de Requisições de Objetos"*, tese de mestrado defendida em março de 1995, FEE-UNICAMP.

[Nicol93] - Nicol, J. R.,Wilkes, T., Manola, F.A., *"Object Orientation in Heterogeneous Distributed Computing Systems"*, IEEE Computer, p. 57-67, Jul. 1993.

[Raymond93] - Raymond, K. A., *"Reference Model of Open Distributed Processing: a Tutorial"*, Proceedings of the IFIP TCG/WG6.1 International Conference on Open Distributed Processing, Berlin, Alemanha, 1993.

[RM-ODPa] - ISO - 10746-1 RM-ODP, *"Part 1: Overview and Guide to Use"*, November 1993.

[RM-ODPb] - ISO - 10746-2 RM-ODP, *"Part 2: Descriptive Model"*, February 1994.

[RM-ODPc] - ISO - 10746-3 RM-ODP, *"Part 3: Prescriptive Model"*, February 1994.

[RM-ODPd] - ISO - 10746-4 RM-ODP, *"Part 4: Architectural Semantics"*, August 1994.

[Rudkin92] - Rudkin, S., *"Modelling Object Interfaces for Open Distributed Processing"*, Bt Technol. J., Vol 10, No 2, April 1992.

[Rudkin93] - Rudkin, S., *"Templates, Types and Classes in Open Distributed Processing"*, bt, Technol. J., Vol 11, No 3, July 1993.

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

[SUN90] - Sun microsystems, *"Network Programming Guide"*, mar. 1990.

[Tanembaum92] - Tanembaum, A. S., *"Modern Operating Systems"*, Prentice Hall International Editions, 1992.

[Tschammer93] - Tschammer, V. et al, *"Processamento Distribuído Aberto e o Modelo RM-ODP/ISO"*, 11º Simpósio Brasileiro de Redes de Computadores, 1993.

ANEXO A

Descrição dos Métodos da API em C++

```
//****************************************
// API.HPP
//****************************************

#ifndef _api_hpp
#define _api_hpp


# include "apilist.hpp"

typedef enum operational{INVOCATION,TERMINATION}OP_TYPE;
typedef enum arq{SUN,IBM}ARQ;
typedef enum causality{INITIATING,RESPONDING}CAUSALITY;
typedef enum direction1{CLIENTE,SERVIDOR}DIRECTION,PARAMETER;
typedef enum interface{SINAL,OPERATIONAL,STREAM}INTERFACE;

typedef struct struct_signal_data{
  char * nome;
  int n_parametro;
  struct parameter{
        TYPE tipo;
        union{
              char char_valor;
              int int_valor;
              float float_valor;
              double double_valor;
              };
        }parametro[30];
 }STRUCT_SIGNAL_DATA;
```

```
typedef struct struct_operational_data{
        char * nome;
        OP_TYPE optipo;
        int n_parametro;
        struct parameter{
         TYPE tipo;
        union{
                char char_valor;
                 int int_valor;
                float float_valor;
                double double_valor;
                };
          }parametro[30];
}STRUCT_OPERATIONAL_DATA;


typedef struct struct_stream_data{
        char * nome;
        int n_sinal;
        int n_parametro;
        struct parameter{
                TYPE tipo;
                union{
                        char char_valor;
                        int int_valor;
                        float float_valor;
                        double double_valor;
                          };
        }parametro[30];
}STRUCT_STREAM_DATA;


//************************************
// template
//************************************
typedef struct parametro{
 char * buffer;
 int tamanho;
}dado;
```

```
typedef class Template{
private:
char * name;
char *buffer;
int tamanho_buffer;

public:
Template(){};
~Template();

dado getBUFF();
void Save();
} TEMPLATE;

//***********************************
// objeto computacional basico
//***********************************
typedef union{
        char char_valor;
        int int_valor;
        float float_valor;
        double double_valor;
        }ATTRIBUTE_RESULT;

typedef class basic_object {
private:

public:
basic_object();
~basic_object();

unsigned long addInterface(unsigned long, char*);
int deleteInterface(unsigned long,unsigned long);

L_COMP_INTERFACE getInterfaces(unsigned long);

ATTRIBUTE_RESULT readAttribute(unsigned long,char*,TYPE);
int modifyAttribute(unsigned long,char *,TYPE,ATTRIBUTE_RESULT);

int checkpointObject(unsigned long,char *);
int deactivateObject(unsigned long, char*);
int recoverObject(unsigned long, char *);
int deleteObject(unsigned long);
```

```
} BASIC_OBJECT;

//***********************************
// objeto computacional de binding
//***********************************
typedef class bind_object {
private:

public:
bind_object();
~bind_object();

int Unbind(unsigned long);

int Submit(unsigned long,STRUCT_STREAM_DATA);
int Submit(unsigned long,STRUCT_OPERATIONAL_DATA);
int Submit(unsigned long,STRUCT_SIGNAL_DATA);

void Deliver(unsigned long,STRUCT_SIGNAL_DATA &);
void Deliver(unsigned long,STRUCT_STREAM_DATA &);
void Deliver(unsigned long,STRUCT_OPERATIONAL_DATA &);
} BIND_OBJECT;



//***********************************
// cluster
//***********************************
typedef class cluster {
private:

public:
cluster();
~cluster();

unsigned  long makeObject(unsigned long,char *);

L_BASIC_OBJECT getObjects(unsigned long);


int checkpointCluster(unsigned long,char *);
int deactivateCluster(unsigned long, char*);
int recoverCluster(unsigned long, char *);
int deleteCluster(unsigned long)
```

```
} CLUSTER;

//************************************
// capsule
//************************************

typedef class capsule {
private:

public:
capsule();
~capsule();
unsigned long makeCluster(char*, char *);
L_CLUSTER  getClusters(unsigned long);
} CAPSULE;



//************************************
// node
//************************************
typedef class node {
private:

public:
node();
~node();

unsigned long makeCapsule(char*);
L_CAPSULE *getCapsules( );
unsigned long Bind(unsigned long, char *,int,char *,char *);
unsigned long Accept(unsigned long, char *,int);
} NODE;
```

```
//***********************************
// Signal Interface
//***********************************
typedef struct signal_receive{
		unsigned long nome_global;
		STRUCT_SIGNAL_DATA dados;
	}SIGNAL_RECEIVE;

typedef class signal_interface {
private:

public:
signal_interface();
~signal_interface();
int queue(unsigned long,unsigned long, STRUCT_SIGNAL_DATA);
SIGNAL_RECEIVE pop(unsigned long);
} SIGNAL_INTERFACE;


//***********************************
// Operational Interface
//***********************************
typedef struct operational_receive{
	unsigned long nome_global;
	STRUCT_OPERATIONAL_DATA dados;
}OPERATIONAL_RECEIVE;

typedef class operational_interface {
private:

public:
operational_interface();
~operational_interface();
int queue(unsigned long,unsigned long, STRUCT_OPERATIONAL_DATA);
OPERATIONAL_RECEIVE pop (unsigned long);
} OPERATIONAL_INTERFACE;
```

```
//************************************
// Stream Interface
//************************************

typedef struct stream_receive{
                unsigned long nome_global;
                STRUCT_STREAM_DATA dados;
        }STREAM_RECEIVE;

typedef class stream_interface {
private:

public:
stream_interface();
~stream_interface();
int queue(unsigned long,unsigned long, STRUCT_STREAM_DATA);
STREAM_RECEIVE pop(unsigned long);
} STREAM_INTERFACE;

#endif
```

ANEXO B

ESTRUTURA DE ARMAZENAGEM DOS TEMPLATES

As informações dos *templates* são armazenados através de um gerenciador de banco de dados: o GNU gdbm. Para armazenar as informações dos respectivos *templates*, as informações são serializadas para armazenagem. A identificação dos *templates* no GNU gdbm se dá através de uma chave de identificação. A base de objetos utiliza as informações dos respectivos *templates* para execução de serviços. Abaixo são descritos as informações que são armazenadas de cada *template*.

Template de Objeto Básico Computacional

- nome (char*)
- chave do contrato (contract) (char*)
- chave do comportamento (behaviour) (char*)
- número de interfaces (int)
- chave da interface 1 (char*)
- chave da interface N (char*)
- número de atributos (int)
- nome do atributo 1 (char*)
- tipo do atributo 1 (int)
- valor do atributo 1 (char, int, double ou float)
- nome do atributo N (char*)
- tipo do atributo N (int)
- valor do atributo N (char, int, double ou float)

Template de Objeto Binding

- nome (char*)
- chave da interface computacional (stub) (char*)
- chave da interface computacional (binder) (char*)
- chave da interface computacional (protocol) (char*)
- chave do comportamento (behaviour) (char*)
- chave do contrato (contract) (char*)
- parâmetros formais (int)

- número de atributos (int)
- nome do atributo 1 (char*)
- tipo do atributo 1 (char*)
- nome do atributo N (char*)
- tipo do atributo N (char*)

Template de Interface Sinal

- nome (char*)
- chave do contrato (contract) (char*)
- chave do comportamento (behaviour) (char*)
- número de sinais (int)
- chave do sinal 1 (char*)
- chave do sinal N (char*)

Template de Interface Operacional

- nome (char*)
- chave do contrato (contract) (char*)
- chave do comportamento (behaviour) (char*)
- chave do sinal (invocation) (char*)
- chave do sinal (termination) (char*)

Template de Interface Stream

- nome (char*)
- chave do contrato (char*)
- chave do comportamento (char*)
- direção do fluxo (int)
- chave do sinal (char*)

Template de Sinal

- nome (char*)
- número de parâmetros (int)
- nome da variável 1 (char*)
- tipo da variavel (int)
- nome da variável N (char*)
- tipo da variável (int)

- causalidade (int)

Template de Contrato

- nome (char*)
- chave da qualidade de serviço (QoS) (char*)

Template de QoS

- tamanho maximo da mensagem (int)
- período de envio entre mensagens (segundos) (double)
- retardo máximo da mensagem (double)

Template de Comportamento

- nome (char*)
- nome do arquivo do código fonte (src_code) (char*)
- nome do arquivo makefile (char*)
- nome da "thread" a ser executada (char*)
- arquitetura a ser utilizada (char*) (int)

Template de Nodo

- nome (char*)
- arquitetura (architecture) (int)
- chave do comportamento (behaviour) (char*)
- chave do objeto computacional básico (nucleus) (char*)
- número de cápsulas associadas (int)
- chave da cápsula 1 (char*)
- chave da cápsula N (char*)

Template de Cápsula

- nome (char*)
- chave do comportamento (behaviour) (char*)
- chave do objeto computacional básico (cpm) (char*)
- número de clusters associados (int)

- chave do cluster 1 (char*)
- chave do cluster N (char*)

Template de Cluster

- nome (char*)
- chave do comportamento (behaviour) (char*)
- chave do objeto computacional básico (clm) (char*)
- número de objetos pertencentes ao cluster (int)
- chave do objeto computacional 1 (char*)
- chave do objeto computacional N (char*)

ANEXO C

PARÂMETROS ENVIADOS ENTRE A API E A BASE DE OBJETOS

A interface de programação de aplicação requisita serviços da base de objetos. A requisição de serviços é feita através do envio de mensagens à base de objetos. O envio de mensagens se dá através do protocolo de comunicação *reliable unicast* [Fapesp95]. Antes dos parâmetros serem enviados, API converte os parâmetros para o formato canônico (utilizando-se o XDR). Ao receber os parâmetros, a base de objetos desconverte os parâmetros recebidos para o formato local e executa os serviços requisitados. Após a execução destes serviços, a base de objetos retorna os resultados para a API. Abaixo são descritos os serviços requisitados à base de objetos, os formatos dos parâmetros que devem ser passados para execução destes serviços e os formatos dos parâmetros de retorno.

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 da resposta (int).

parâmetros de resposta:

- número de *clusters* (int).
- nome dos *clusters* que compõem a cápsula (char*).

Adicionar um cluster a cápsula

parâmetros de entrada:

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

parâmetros de resposta:

- identificador do *cluster* criado (unsigned long).

Obter a lista de objetos que compõem o cluster

parâmetros de entrada:

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

parâmetros de resposta:

- número de objetos (int).
- nome dos objetos que compõem o *cluster* (char*).

Adicionar um objeto ao cluster

parâmetros de entrada:

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

parâmetros de resposta:

- identificador do objeto adicionado (unsigned long).

Retirar um objeto do cluster

parâmetros de entrada:

- identificação do pedido (int).
- nome do *host* para envio de resposta (char *).
- porta de comunicação para envio da resposta (int).
- identificador do *cluster* (unsigned long).
- identificador do objeto a ser retirado (unsigned long).

parâmetros de resposta:

- indicação de sucesso ou fracasso (int).

Fazer checkpoint de cluster

parâmetros de entrada:

- identificação do pedido (int).
- nome do *host* para envio de resposta (char *).
- porta de comunicação para envio da resposta (int).
- identificador do *cluster* (unsigned long).
- nome do *template* de *cluster* a ser utilizado para armazenagem no gdbm (char *).

parâmetros de resposta:

- indicação de sucesso ou fracasso (int).

Desativar o cluster

parâmetros de entrada:

- identificação do pedido (int).
- nome do *host* para envio de resposta (char *).
- porta de comunicação para envio da resposta (int).
- identificador do *cluster* (unsigned long).
- nome do *template* de *cluster* a ser utilizado para armazenagem no gdbm (char *).

parâmetros de resposta:

- indicação de sucesso ou fracasso (int).

Remover o cluster

parâmetros de entrada:

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

parâmetros de resposta:

- indicação de sucesso ou fracasso (int).

Recuperar o cluster

parâmetros de entrada:

- identificação do pedido (int).
- nome do *host* para envio de resposta (char *).
- porta de comunicação para envio da resposta (int).
- identificador do *cluster* (unsigned long).
- nome do *template* de *cluster* a ser utilizado na recuperação do *cluster* (char *).

parâmetros de resposta:

- indicação de sucesso ou fracasso (int).

Obter a lista de interfaces que compõem o objeto

parâmetros de entrada:

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

parâmetros de resposta:

- número de interfaces (int).
- nome das interfaces que compõem o objeto (char *).

Adicionar uma interface ao objeto

parâmetros de entrada:

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

parâmetros de resposta:

- identificador de interface adicionado (unsigned long).

Retirar uma interface do objeto

parâmetros de entrada:

- identificação do pedido (int).
- nome do *host* para envio de resposta (char *).
- porta de comunicação para envio da resposta (int).
- identificador do objeto (unsigned long).
- idenficador de interface a ser retirada (unsigned long).

parâmetros de resposta:

- indicação de sucesso ou fracasso (int).

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 da resposta (int).
- identificador do objeto (unsigned long).
- nome do atributo (char *).
- tipo do atributo (int).

parâmetro de resposta:

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

Modificar um atributo de objeto

parâmetros de entrada:

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

Fazer checkpoint de objeto

parâmetros de entrada:

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

parâmetros de resposta:

- identificador de sucesso ou fracasso (int).

Desativar o objeto

parâmetros de entrada:

- identificação do pedido (int).
- nome do *host* para envio de resposta (char *).
- porta de comunicação para envio da resposta (int).
- identificador do objeto (unsigned long).
- nome do *template* de objeto a ser utilizado para armazenagem no gdbm (char *).

parâmetro de resposta:

- indicação de sucesso ou fracasso (int).

Remover o objeto

parâmetros de entrada:

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

- porta de comunicação para envio da resposta (int).
- identificador do objeto (unsigned long).

parâmetros de resposta:

- indicação de sucesso ou fracasso (int).

Recuperar o objeto

parâmetros de entrada:

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

parâmetro de resposta:

- indicação de sucesso ou fracasso.

Enviar um dado para a interface sinal (queue).

parâmetros de entrada:

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

- ◊ tipo do parâmetro N (char *).
- ◊ valor do parâmetro N (char, int, float ou double).

Parâmetro de resposta:

- indicação de sucesso ou fracasso (int).

Receber um dado de uma interface sinal (pop).

parâmetros de entrada:

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

Parâmetro de resposta:

- identificador da interface que enviou o dado (unsigned long).
- dados a serem recebidos.
    - ◊ nome do sinal (char *).
    - ◊ número de parâmetros do sinal (int).
    - ◊ tipo do parâmetro 1 (char *).
    - ◊ valor do parâmetro 1 (char, int, float ou double).
    - ◊ :
    - ◊ :
    - ◊ tipo do parâmetro N (char *).
    - ◊ valor do parâmetro N (char, int, float ou double).

Enviar um dado para a interface operacional (queue).

parâmetros de entrada:

- identificação do pedido (int).
- nome do *host* para envio de resposta (char *).
- porta de comunicação para envio da resposta (int).
- identificador da interface origem (unsigned long).
- identificador da interface destino (unsigned long).

- dados a serem enviados.
    - ◊ nome do sinal (char *).
    - ◊ tipo da operação (int).
    - ◊ número de parâmetros do sinal (int).
    - ◊ tipo do parâmetro 1 (char *).
    - ◊ valor do parâmetro 1 (char, int, float ou double).
    - ◊ :
    - ◊ :
    - ◊ tipo do parâmetro N (char *).
    - ◊ valor do parâmetro N (char, int, float ou double).

Parâmetro de resposta:

- indicação de sucesso ou fracasso (int).

Receber um dado de uma interface operacional (pop).

parâmetros de entrada:

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

Parâmetro de resposta:

- identificador da interface que enviou o dado (unsigned long).
- dados a serem recebidos.
    - ◊ nome do sinal (char *).
    - ◊ tipo da operação (int).
    - ◊ número de parâmetros do sinal (int).
    - ◊ tipo do parâmetro 1 (char *).
    - ◊ valor do parâmetro 1 (char, int, float ou double).
    - ◊ :
    - ◊ :
    - ◊ tipo do parâmetro N (char *).

◊ valor do parâmetro N (char, int, float ou double).

Enviar um dado para a interface stream (queue).

parâmetros de entrada:

- identificação do pedido (int).
- nome do *host* para envio de resposta (char *).
- porta de comunicação para envio da resposta (int).
- identificador da interface origem (unsigned long).
- identificador da interface destino (unsigned long).
- dados a serem enviados.
    - ◊ nome do sinal (char *).
    - ◊ número de sinais (int).
    - ◊ número de parâmetros do sinal (int).
    - ◊ tipo do parâmetro 1 (char *).
    - ◊ valor do parâmetro 1 (char, int, float ou double).
    - ◊ :
    - ◊ :
    - ◊ tipo do parâmetro N (char *).
    - ◊ valor do parâmetro N (char, int, float ou double).

Parâmetro de resposta:

- indicação de sucesso ou fracasso (int).

Receber um dado de uma interface stream (pop).

parâmetros de entrada:

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

Parâmetro de resposta:

- identificador da interface que enviou o dado (unsigned long).
- dados a serem recebidos.
  - ◊ nome do sinal (char *).
  - ◊ número de sinais (int).
  - ◊ número de parâmetros do sinal (int).
  - ◊ tipo do parâmetro 1 (char *).
  - ◊ valor do parâmetro 1 (char, int, float ou double).
  - ◊ :
  - ◊ :
  - ◊ tipo do parâmetro N (char *).
  - ◊ valor do parâmetro N (char, int, float ou double).

Criar a metade do canal do lado do cliente ou produtor (bind).

parâmetros de entrada:

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

parâmetro de resposta:

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

Criar a metade do canal do lado servidor ou consumidor (accept).

parâmetros de entrada:

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

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

parâmetro de resposta:

- identificador de um objeto Accept (unsigned long).

Destruição do canal pelo cliente (unbind).

parâmetros de entrada:

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

Parâmetro de resposta:

- indicação de sucesso ou fracasso (int).

Enviar um dado para um canal sinal (submit).

parâmetros de entrada:

- identificação do pedido (int).
- nome do *host* para envio de resposta (char *).
- porta de comunicação para envio da resposta (int).
- identificador do objeto Bind (unsigned long).
- dados a serem enviados.
  - ◊ nome do sinal (char *).
  - ◊ número de parâmetros do sinal (int).
  - ◊ tipo do parâmetro 1 (char *).
  - ◊ valor do parâmetro 1 (char, int, float ou double).
  - ◊ :
  - ◊ :
  - ◊ tipo do parâmetro N (char *).
  - ◊ valor do parâmetro N (char, int, float ou double).

Parâmetro de resposta:

- indicação de sucesso ou fracasso (int).

Receber um dado de um canal sinal (deliver).

parâmetros de entrada:

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

Parâmetro de resposta:

- dados a serem enviados.
  - ◊ nome do sinal (char *).
  - ◊ número de parâmetros do sinal (int).
  - ◊ tipo do parâmetro 1 (char *).
  - ◊ valor do parâmetro 1 (char, int, float ou double).
  - ◊ :
  - ◊ :
  - ◊ tipo do parâmetro N (char *).
  - ◊ valor do parâmetro N (char, int, float ou double).

Enviar um dado para um canal operacional (submit).

parâmetros de entrada:

- identificação do pedido (int).
- nome do *host* para envio de resposta (char *).
- porta de comunicação para envio da resposta (int).
- identificador do objeto Bind (unsigned long).
- dados a serem enviados.
  - ◊ nome do sinal (char *).

◊ tipo do sinal (int).

◊ número de parâmetros do sinal (int).

◊ tipo do parâmetro 1 (char *).

◊ valor do parâmetro 1 (char, int, float ou double).

◊ :

◊ :

◊ tipo do parâmetro N (char *).

◊ valor do parâmetro N (char, int, float ou double).

Parâmetro de resposta:

- indicação de sucesso ou fracasso (int).

Receber um dado de um canal operacional (deliver).

parâmetros de entrada:

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

Parâmetro de resposta:

- dados a serem enviados.

  ◊ nome do sinal (char *).

  ◊ tipo da operação (int).

  ◊ número de parâmetros do sinal (int).

  ◊ tipo do parâmetro 1 (char *).

  ◊ valor do parâmetro 1 (char, int, float ou double).

  ◊ :

  ◊ :

  ◊ tipo do parâmetro N (char *).

  ◊ valor do parâmetro N (char, int, float ou double).

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 da resposta (int).
- identificador do objeto Bind (unsigned long).
- dados a serem enviados.
  - ◊ nome do sinal (char *).
  - ◊ número de parâmetros do sinal (int).
  - ◊ tipo do parâmetro 1 (char *).
  - ◊ valor do parâmetro 1 (char, int, float ou double).
  - ◊ :
  - ◊ :
  - ◊ tipo do parâmetro N (char *).
  - ◊ valor do parâmetro N (char, int, float ou double).

Parâmetro de resposta:

- indicação de sucesso ou fracasso (int).

Receber um dado de um canal stream (deliver).

parâmetros de entrada:

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

Parâmetro de resposta:

- dados a serem enviados.
  - ◊ nome do sinal (char *).
  - ◊ número de sinais (int).
  - ◊ número de parâmetros do sinal (int).

◊ tipo do parâmetro 1 (char *).

◊ valor do parâmetro 1 (char, int, float ou double).

◊ :

◊ :

◊ tipo do parâmetro N (char *).

◊ valor do parâmetro N (char, int, float ou double).