Uma proposta para modelagem de Funções de Gerenciamento Para Processamento Distribuído Aberto

Cláudio Martins Garcia

Uma proposta para modelagem de Funções de Gerenciamento Para Processamento Distribuído Aberto

Este exemplar corresponde à redação final da dissertação devidamente corrigida e defendida pelo Sr. Cláudio Martins Garcia e aprovada pela Comissão Julgadora.

Campinas, 21 de dezembro de 1994.

Edmundo R. M. Madeira

Prof. Dr. Edmundo Roberto Mauro Madeira
Orientador

Dissertação apresentada ao Instituto de Matemática, Estatística e Ciência da Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

Uma proposta para modelagem de Funções de Gerenciamento Para Processamento Distribuído Aberto [1]

Cláudio Martins Garcia[2]

Departamento de Ciência da Computação
IMECC – UNICAMP

Banca Examinadora:

- Edmundo Roberto Mauro Madeira (Orientador)
- Waldomiro Pelágio Diniz de Carvalho Loyolla [4]
- Célio Cardoso Guimarães [3]
- Maria Beatriz Felgar de Toledo (Suplente) [5]

---

[1] Dissertação apresentada ao Instituto de Matemática, Estatística e Ciência da Computação da UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

[2] O autor é Bacharel em Ciências da Computação pela Universidade Federal de Goiás.

[3] Professor do Departamento de Ciência da Computação - IMECC - UNICAMP.

[4] Professor do Departmento de Engenharia Elétrica da Faculdade de engenharia e tecnologia - Unesp - Bauru.

[5] Professora do Departamento de Ciência da Computação - IMECC - UNICAMP.

À Paula pelo tempo passado distante.

Agradecimentos

Ao CNPq e à FAPESP pelo apoio financeiro recebido.

Ao Departamento de Matemática da Universidade Federal de Goiás pela autorização de uso de seus equipamentos.

Ao professor Edmundo, meu orientador, pela orientação e atenção recebidas.

Aos funcionários do DCC e do Centro de Computação da Unicamp, sem os quais a realização desse trabalho seria mais difícil.

Aos Colegas e amigos que com sua convivência tornaram o tempo em que realizei esse mestrado um constante aprender. Seja no dia a dia de estudo, seja em um gesto para compreender que mesmo diante das maiores dificuldades existe um limite ao qual não devemos ultrapassá-lo para continuarmos humanos. Afinal o importante é sermos seres humanos, um mestrado é apenas um detalhe que só vale a pena se nos ajudar a nos conhecer melhor.

A liberdade de pensar e de se expressar enfim.

" Daddy's flown across the ocean
Leaving just a memory
Snapshot in the family album
Daddy what else did you leave for me?
Daddy, what'd'ja leave behind for me?!?
All in all it was just a brick in the wall.
All in all it was all just bricks in the wall."

" We don't need no education
We dont need no thought control
No dark sarcasm in the classroom
Teachers leave them kids alone
Hey! Teachers! Leave them kids alone!
All in all it's just another brick in the wall.
All in all you're just another brick in the wall".

" I don't need no arms around me
And I dont need no drugs to calm me.
I have seen the writing on the wall.
Don't think I need anything at all.
No! Don't think I'll need anything at all.
All in all it was all just bricks in the wall.
All in all you were all just bricks in the wall."

Roger Waters, Pink Floyd,
"Another Brick in the wall "

Resumo

Este trabalho cobre um estudo do modelo de referência para processamento distribuído aberto da ISO/ITU-T (*RM-ODP Reference Model for Open Distributed Processing*) com ênfase nas funções de gerenciamento descritas no modelo. Sendo que é apresentado um estudo de tais funções e um refinamento das mesmas, e como forma de validação das idéias a implementação de um protótipo do modelo. O Capítulo 1 do texto apresenta uma descrição geral do que é ODP e o RM-ODP com seus principais conceitos e ainda alguns conceitos relativos a plataformas existentes que tentam prover processamento distribuído.

No capítulo 2 é realizado um estudo detalhado do chamado *modelo de engenharia* do RM-ODP, descrevendo seus objetos componentes, e também as funções propostas no RM-ODP, realizando um estudo da interação entre as mesmas. O enfoque maior será dado às funções de gerenciamento que consistem no ponto central desse trabalho. Tais funções serão apresentadas em um nível de detalhamento maior no capítulo 3.

O quarto capítulo apresenta um conjunto de estruturas de dados e conceitos para modelagem das funções de gerenciamento. Os conceitos aqui levantados são resultado da implementação de um protótipo do modelo com ênfase nas funções de gerenciamento.

No último capítulo são levantadas as dificuldades na realização do trabalho, bem como as principais contribuições que acreditamos ter apresentado, ainda aqui são apresentadas as limitações existentes no trabalho e possíveis extensões e trabalhos futuros.

Abstract

This work performs a study of the ISO/ITU-T Open Distributed Processing Reference Model, with emphasis on the management functions of the model. A study of the management functions is presented, with a refinement of the functions, and to validate the main ideas a prototype implementation of the model was developed. The chapter 1 of the text presents a general description about the ODP and RM-ODP with their main concepts.

In chapter 2 is showed a study in details of the engineering model of the RM-ODP and it is described their component objects and the RM-ODP functions, besides a study of the interaction of the functions. The principal focus is the management functions that are the central point of this work, such functions are presented in chapter 3 with more details.

The fourth chapter presents a set of data structures and concepts to the modeling of the management functions, this chapter is result of the implementation of a prototype with emphasis to the management functions.

In the last chapter the main difficults in the elaboration of this work, and the main contribuitions of this work, are presented. The existent limitation and possibles extensions of the work are also presented, such as suggestions to the future works.

Conteúdo

Lista de Tabelas

Lista de Figuras

Siglas e Abreviaturas

API - Application Programming Interface

ANSA - Advanced Network Systems Architecture

BEO - Basic Engineering Object

CLM - Cluster Manager

CPM - Capsule Manager

DCE - Distributed Computing Environment

ISO - International Standardization Organization

IR - Interface Reference

ITU-T - International Telecommunication Union

ODP - Open Distributed Processing

OM - Object Management

OMG - Object Management Group

ORB - Object Request Broker

OSE - Open Service Environment

OSI - Open Systems Interconnection

RM-ODP - Reference Model for ODP

Capítulo 1

Introdução

Neste capítulo realizamos a apresentação de conceitos gerais, necessários para os capítulos seguintes, bem como conceitos históricos relacionados a processamento distribuído e em especial a processamento distribuído aberto (ODP).

Começamos com um histórico e uma discussão de ambientes abertos e sistemas distribuídos para a seguir apresentar o modelo proposto pela ISO/ITU-T (*International Standardization Organization/ International Telecommunication Union*) para ODP (RM-ODP) de uma forma geral, já que alguns detalhes específicos serão estudados no capítulo 2.

No final do capítulo realizamos uma discussão da plataforma Multiware em desenvolvimento na qual esse trabalho pretende dar contribuições em uma parte da referida plataforma.

1.1 Conceitos Básicos

A disponibilidade de computadores pessoais, estações de trabalho, redes locais e redes metropolitanas de alto desempenho modificaram drasticamente a maneira pela qual a informação é processada. O normal não é mais se ter sistemas dedicados e mono-usuários. Hoje, os usuários esperam ser capazes de comunicar-se além dos limites de seus computadores de mesa para um vasto nível de características, funcionalidades e desempenhos. Interoperabilidade é a nova regra [46].

Com vista a resolver o problema de interoperabilidade em redes a ISO (*International Standardization Organization*) propôs o modelo OSI (*Open Systems Interconnection*) que visa justamente prover uma estrutura para tal interação. Resolvido o problema de interconectividade, com a expansão e evolução dos sistemas computacionais apresenta-se uma nova questão . Tendo-se em vista a ne-

cessidade de realizarmos processamento distribuído nesse ambiente heterogêneo (aberto) de redes, computadores e sistemas operacionais, fica a questão de que se seria possível tal fato e, se sim, como deveríamos proceder.

O modelo de referência OSI da ISO provê uma estrutura para a interoperabilidade em redes [40]. Para permitir interoperabilidade , os usuários necessitam de um ambiente no qual todos os sistemas e seus recursos estejam altamente disponíveis. Redes, que fornecem comunicação, são somente uma parte da solução.

Com o avanço do uso de sistemas de computação distribuídos que hoje estão cada vez mais complexos e tem seu porte cada vez maior deixando o ambiente restrito de um edifício, em uma rede local, em que a maioria dos equipamentos e do *software* são do mesmo tipo ou ao menos compatíveis, temos hoje um contexto em que sobressaem-se sistemas que fazem uso de redes metropolitanas em que convivem com todo tipo de *hardware* e *software*. Não existe uma metodologia de projeto comum a esses sistemas de computação, nem uma arquitetura que permita interoperabilidade entre os mesmos.

A construção de sistemas distribuídos e interconectados em um ambiente multi-vendedor, heterogêneo, depende da criação de padrões adequados para os comportamentos requeridos dos componentes que constituem o sistema [34].

O rápido crescimento de processamento distribuído tem mostrado a necessidade de uma estrutura coordenada para a padronização de processamento distribuído aberto (ODP - *Open Distributed Processing*). A ISO/ITU-T [1, 2, 3, 4] apresenta um modelo de referência para ODP que fornece tal estrutura. O mesmo cria uma arquitetura na qual o suporte a distribuição, interoperabilidade e portabilidade podem ser integrados [2].

Na elaboração do modelo de referência para ODP surge a definição de um conjunto de funções necessárias para suportar processamento distribuído aberto, entre as quais as *funções de gerenciamento* que compõem o assunto principal deste texto.

1.1.1 Sistemas distribuídos abertos

Ambientes abertos de serviços surgem como uma resposta a uma série de fatores e necessidades entre os quais destacam-se [64]:

- *Desenvolvimento da tecnologia de comunicação.* O avanço tecnológico na área de comunicações consiste na base para um ambiente aberto de serviços (OSE - *Open Service Environment*), fornecendo condições para troca de informação digitalizada de forma eficiente, bem como a interconexão de sub-redes diferentes. Com o desenvolvimento da fibra ótica foi possível

uma maior velocidade de transmissão, reduzindo a taxa de erros, maior confiabilidade, levando a um maior *throughput.*

- *Avanço das necessidades dos usuários.* Caminham rumo a um sistema interconectado, em que, por exemplo, exista uma integração das ilhas de automação já instaladas, a integração de diferentes departamentos com estruturas hierárquica e computacional diferentes, bem como suporte a tecnologias emergentes como multimídia, e *groupware.* Essas necessidades levarão certamente à imposição de um trabalho cooperativo em sistemas distribuídos.

- *Progresso da Padronização e projetos relacionados.* Como reflexo das necessidades acima mencionadas surge a tentativa de padronização por parte de várias entidades como ISO/ITU-T(ODP) [1] e ECMA [1] (ODP-SE) [2] [19], bem como de diversos projetos de pesquisa que nos levam à necessidade de padronização, como em DCE [3] (OSF) [4][46], ANSA [5] (APM) [6] [8], BERCIM [64, 63] e ORB [7] (OMG) [8] [42].

Um sistema aberto pode ser visto como um sistema capaz de interagir com outros sistemas no ambiente. Temos como principais características de um ambiente aberto o seguinte [64]:

- *Ilimitado*, ou seja, público e livre de restrições, sejam elas do ponto de vista geográfico, organizacional ou técnico.

- *De livre acesso para todos*, ou seja, o ambiente pode admitir quaisquer tipos de usuários, componentes ou aplicações.

- *Heterogêneo*, no qual cada organização ou pessoa pode decidir sobre sua existência, projeto, implementação, uso e gerenciamento de seus componentes.

- *Autônomo*, em que cada entidade está submetida às características de seu ambiente local, tendo liberdade para determinar suas propriedades, seu comportamento, e a evolução de seus componentes.

---

[1] European Computer Manafacturers Association
[2] Open Distributed Processing - Service Environment
[3] Distributed Computing Environment
[4] Open Software Foundation
[5] Advanced Network Systems Architecture
[6] Architecture Projects Management
[7] Object Request Broker
[8] Object Management Group

- *Descentralizado*, em que propriedades e comportamentos dos componentes são gerenciados a nível local, sem qualquer influência ou controle externo.

Em um ambiente aberto não existe como observar o comportamento de todo o sistema, nenhum usuário ou administrador é capaz de realizar isto, assim como não é capaz de influenciar o sistema como um todo. Cada observador tem apenas uma visão parcial do sistema, válida sobre um local e um determinado instante. Logo, cada informação ou ação relacionada a um ambiente aberto é válida somente para um certo ponto de vista do observador ou ação, com área de influência limitada, e um período de tempo dedicado.

Quando consideramos um ambiente aberto em relação a sua integridade e privacidade devemos levar em conta as adoções de medidas de restrição de acesso de forma tal que o ambiente seja mantido livre do acesso indevido de entidades não autorizadas, operações locais devem manter-se sem distúrbios. Considera-se ainda o fato de que informações sobre os componentes e o ambiente serem incertas e dúbias, podendo serem ainda atrasadas e incompletas.

Podemos notar diferenças primordiais entre sistemas tradicionais (distribuídos e *stand-alone*) e sistemas ODP que são [1, 65]:

- *Controle.* Controle central em sistemas tradicionais, autonomia em sistemas ODP.
- *Nomes.* Nomes globais em um espaço único de nomes em sistemas tradicionais, federação de nomes em sistemas ODP.
- *Memória compartilhada.* Memória global compartilhada em sistemas tradicionais, em oposição a estados locais encapsulados em sistemas ODP.
- *Consistência.* Consistência global em sistemas tradicionais, consistência fraca em sistemas ODP.
- *Execução.* Seqüencial em sistemas tradicionais, concorrente em sistemas ODP.
- *Tolerância a falhas.* Sistemas tradicionais são vuneráveis a falhas, sistemas ODP são tolerantes a falhas.
- *Localidade da Interação.* De certa forma é local em sistemas tradicionais, remota em ODP.
- *Localização.* Fixa em sistemas tradicionais, permite migração em sistemas ODP.

- *Configuração.* De certa forma é fixa em sistemas tradicionais, configuração incremental em sistemas ODP.

- *Homogenidade.* Sistemas tradicionais são homogêneos, sistemas ODP são heterogêneos.

Em sistemas distribuídos tradicionais temos sistemas fechados, separados do ambiente externo como uma conseqüência da limitação da tecnologia, facilidades de comunicação, decisões de projeto e medidas de segurança. Os mesmos são caracterizados por apresentarem um número limitado de componentes, grupos de usuários fechados, onde todos os componentes pertencem a uma única organização que controla o projeto, instalação, operação e gerenciamento do sistema. A distribuição é um resultado de decisões de projeto dedicadas, aplicadas para permitir o desempenho necessário ou tolerância a falhas, ou para o uso compartilhado de componentes caros. Componentes estão fortemente integrados e acoplados por conceitos comuns, mecanismos e aplicações. A independência e autonomia local dos componentes é limitada. Tais sistemas são caracterizados como *sistemas fortemente acoplados*.

Em sistemas autônomos cooperativos, aplicações desenvolvem-se em um ambiente de serviços abertos. Temos mais grupos lógicos do que estruturas físicas e federações para propósito específico. O sistema apresenta-se "virtualmente distribuído", não necessariamente limitado à estrutura física da rede. Os critérios de organização podem ser a estrutura organizacional (direitos do dono, regulamentação de acesso), condições de operação (carga, taxa de erro), critério de otimização (balanceamento de carga, custo da otimização). A distribuição é uma conseqüência das condições de ambiente em vez de decisão de projeto. Como usuários e componentes estão distribuídos a aplicação também está. Cooperação está limitada em tempo, espaço, e aspectos funcionais. Os componentes permanecem sob controle local e as características locais são preservadas. Tais sistemas são caracterizados como *federações fracamente acopladas*. O ambiente de serviços aberto requer novos princípios de projeto para sistemas distribuídos.

1.2 O Modelo de Referência para ODP da ISO (RM-ODP)

O escopo do padrão ODP pode ser resumido por fornecer uma estrutura para construção de sistemas baseados em computadores distribuídos que [27] :

- Têm muitos vendedores.

- São implementados usando tecnologias heterogêneas.
- Habilita aplicações a interagirem e trocarem dados com outras.
- Reduz custos operacionais e de desenvolvimento.

Para desenvolver padrões de serviços através de múltiplos sistemas e seus componentes, a ISO e ITU-T estão desenvolvendo um esforço conjunto de padronização conhecido como ODP. A regra inicial de ODP é um modelo de referência que integra uma longa faixa de padrões futuros ODP para sistemas distribuídos e mantém a consistência entre esses sistemas, considerando a heterogenidade de hardware, sistemas operacionais, redes, linguagens de programação, bancos de dados e autoridades de gerenciamento. Esses mecanismos irão endereçar questões fundamentais como transparência de acesso, localização, migração, concorrência, falha e replicação.

O modelo ODP proposto deve suportar mecanismos para comunicação e coordenação da informação distribuída relevante à empresa. Os padrões suportando ODP podem ser divididos em categorias que são:

- Um *modelo de referência ODP* básico (RM-ODP) que define conceitos e identifica funções comuns.
- *Modelos de referência específicos* cobrindo tipos individuais de empresas (que usam funções e conceitos comuns definidos no RM-ODP) e acrescentando detalhes conceituais e funções específicas.
- Padrões para a realização de funções comuns (Ex.: Trader [7]).
- Padrões para a realização de funções específicas necessárias para aplicações particulares (Exemplo: Interface para conexão de chamadas telefônicas).

A construção de sistemas distribuídos em um ambiente multi-vendedor depende da criação de padrões aceitáveis para o comportamento requerido dos componentes que constituem o sistema.

O modelo de referência básico para ODP está baseado em conceitos e no uso de técnicas de descrição formal para especificar a arquitetura. São apresentados os seguintes modelos:

- *Modelo Descritivo [2].* Contém a definição dos conceitos além da estrutura analítica e da notação para descrição normalizada de sistemas de processamento distribuído.

- *Modelo Prescritivo [3].* Contém a especificação das características necessárias que qualificam processamento distribuído como aberto. As restrições, as quais o padrão ODP deve ajustar-se, serão expressas em termos do modelo para cada um dos *cinco pontos de vista* (ver abaixo) e funções genéricas definidas no mesmo.

- *Semântica da arquitetura [4].* Relaciona os conceitos do modelo descritivo às técnicas de descrição formal tal que haja uma interpretação uniforme das especificações produzidas quando faz-se uso de diversas técnicas de descrição formal (*TDF*).

1.2.1 Pontos de Vista

Para lidar com a complexidade de um sistema ODP a ISO propõe um modelo em que o sistema é considerado de diversos pontos de vista, cada qual enfocando uma abstração diferente do sistema de interesse. Cinco pontos de vista são identificados, resguardando a especificidade de sistemas ODP particulares [65, 2, 34, 25, 17, 50]:

- *Ponto de vista da Empresa.* Descreve como e quando o sistema de informação é colocado no interior da empresa, e os objetivos que este deve satisfazer. Não está restrito a uma empresa como um todo, ele pode ver partes de uma empresa bem como de várias empresas. O mesmo captura as necessidades comerciais que justificam e governam o projeto do sistema. É expresso pelos papéis dos usuários e pelas políticas de gerenciamento/negócios, pelo sistema ODP e pelo ambiente. Inclui conceitos como interações humanas, segurança e políticas de gerenciamento.

- *Ponto de vista da informação.* Gerentes de informação e engenheiros de informação vêem o sistema deste ponto de vista. Sendo que aqui partes automatizadas não são diferenciadas das partes realizadas manualmente. Aqui estruturas de informação e fluxos são modelados bem como as regras e restrições que governam a manipulação da informação.

- *Ponto de vista Computacional.* Ponto de vista de projetistas de aplicação. Funções de processamento e tipos de dados tornam-se visíveis aqui. Necessidades de transparência de distribuição são identificadas aqui porque a aplicação é estruturada sem levar em conta o sistema computacional e redes disponíveis.

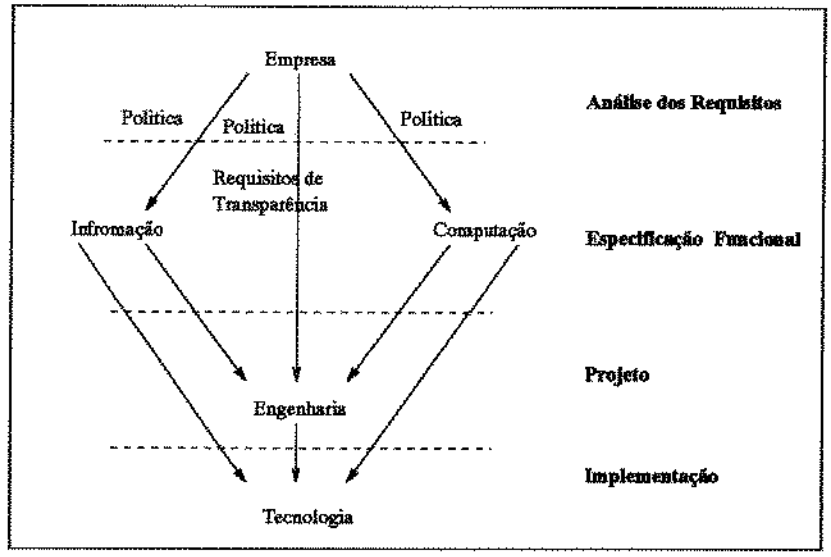

Figura 1.1: **Pontos de Vistas do Modelo de Referência**

- *Ponto de Vista de Engenharia.* Especialistas em sistemas operacionais e comunicação vêem o sistema desse ponto de vista. As especificações do ponto de vista de engenharia estão relacionados com mecanismos de transparência, processos, memória e redes de comunicação que tornam capaz a distribuição de programas e dados. Necessidades de qualidade de serviços e transparências visíveis do ponto de vista computacional são usados para selecionar a partir dos mecanismos disponíveis os necessários a permitir processamento distribuído.

- *Ponto de vista Tecnológico.* Os responsáveis pela configuração, instalação e manutenção de *hardware* e *software* vêem o sistema desse ponto de vista. Os artefatos [9] técnicos dos quais um sistema ODP é constituído são visíveis nesse nível, estes incluem sistema operacional local, dispositivos de I/O, armazenamento, portas de acesso a comunicação.

Um exemplo de como o modelo de pontos de vista pode ser usado durante o desenvolvimento do sistema é dado na figura 1.1 [64].

[9] Um objeto que está envolvido com o uso de recursos, mas incapaz de inicializar ações com respeito a esses recursos

1.2.2 O modelo descritivo

O modelo descritivo utiliza um enfoque de orientação a objetos [2, 15, 20, 61]. Sistemas são modelados em termos de suas interações com o conjunto de objetos componentes e identificadores de interface. Para ajudar a extração de funções comuns, um sistema de tipos de objetos e interfaces é definido. *Templates* de objetos são introduzidos para descrever as restrições sobre as quais objetos são instanciados e os objetos resultantes são organizados em famílias de classes relacionadas por subclasses a superclasses. Este enfoque geral é utilizado repetitivamente nos vários pontos de vista descritos anteriormente, mas os tipos de objetos identificados variam de um componente concreto do subsistema de comunicação para entidades abstratas em um modelo de informação ou empresa.

Os conceitos arquitetônicos necessários para a descrição do sistema ODP são construídos a partir dos diversos conceitos básicos de orientação a objetos. O conjunto de conceitos definidos neste estágio são determinados pelas necessidades da parte prescritiva do modelo [3]. O desenvolvimento do padrão de uma maneira consistente requer estreita colaboração entre os autores do modelo descritivo e prescritivo [34].

Os conceitos arquitetônicos incluem especificação de tipos de estruturação, como estrutura lógica em grupos coordenados e domínios, identificando as obrigações (usando o conceito de *contrato*) e organização temporal via compartilhamento de contexto e ligações [10] (*liaisons*). Conceitos são introduzidos para caracterizar o comportamento do sistema, introduz-se idéias tais como *thread* [11] e *atividade* [12] para suportar a descrição de concorrência e transação. A estrutura é então estendida para expressar causualidade e descrever estruturas computacionais comuns como o relacionamento cliente/servidor.

Conceitos específicos são também desenvolvidos para lidar com aspectos particulares do problema de distribuição, o qual recorre a diferentes pontos de vista.

Uma série de termos e a interação entre os mesmos é definida no modelo, entre os quais os seguintes [3]:

- *Objeto*. Um tipo abstrato de dados que executa funções e oferece serviços, possuindo as seguintes propriedades [64]:

  * Cada objeto encapsula sua informação e interage com outros objetos de uma coleção.

[10] O relacionamento entre um conjunto de objetos o qual resulta da realização de algum comportamento estabelecido.

[11] Uma cadeia de ações, em que pelo menos um objeto participa de todas as ações na cadeia.

[12] Um grafo acíclico direcionado de ações, onde a ocorrência de cada ação no grafo é possível pela ocorrência de todas as ações imediatamente precedentes.

* Acesso externo a informação contida no objeto só é possível via *interações.*

* As interações possíveis de um objeto descrevem seu *comportamento*, sendo este dividido em *interfaces.*

* Interações ocorrem nos *pontos de interação* (operações).

* O comportamento de um objeto é definido por um *template.*

* Um objeto pode ser explicitamente identificado e endereçado.

- *Ações e interações.* Ações são realizadas por objetos, exemplos são a invocação de operações e a criação de objetos. Uma ação é qualquer coisa que aconteça ao objeto. Ações podem ser internas ao objeto ou podem ser observáveis por outros objetos. Quando as mesmas ocorrem entre objetos sincronizados são referenciadas como *interações.*

- *Comportamentos.* Descreve quais ações podem ocorrer. Objetos são caracterizados pelo seu comportamento e pelos seus estados, ambos sendo conceitos duais. O estado de um objeto é definido pelo seu comportamento futuro, e o comportamento futuro descreve o estado corrente.

- *Interfaces.* Uma interface é a projeção de comportamento de um objeto em termos de um conjunto de ações observáveis. Uma *assinatura* é um conjunto de ações observáveis em uma interface. Uma interface deve ser parte de um objeto, sendo que os objetos podem ter múltiplas interfaces. Dois tipos de interfaces estão distinguidas: interface de serviços/funcionais e interfaces de gerenciamento. Objetos (e não interfaces) são o bloco primário de construção.

- *Templates.* Templates são usados para descrever objetos que foram projetados para realizar o mesmo serviço. Um template descreve as características comuns de objetos e abstrai suas diferenças. Por um processo de *instanciação* um objeto é produzido a partir de um template.

- *Tipos e Classes.* Um tipo é um predicado que implicitamente classifica objetos em conjuntos conhecidos como *classes.* Uma *Classe* é um conjunto de objetos satisfazendo um tipo.

Uma série de conceitos são introduzidos mais formalmente, como grupos, domínios e configuração de objetos. Sistemas ODP satisfazem diversas propriedades dentre as quais:

- Transparência. Consiste em ocultar do usuário o comportamento potencial de partes do sistema.
- *Obrigações.* Termo genérico para propriedades relacionadas à cooperação entre objetos. Algumas formas de obrigação são *contratos* e conjuntos de requisitos de serviços.

O comportamento será caracterizado em relação à sua localização (temporal e espacial), ao contrato estabelecido, à causualidade e à confiabilidade. Uma interação pode ser estabelecida entre duas ou mais interfaces através do processo de *binding*. *Trading* é um tipo especial de interação em que a informação sobre potenciais novos contratos é trocada com uma terceira parte (objeto). *Trading* envolve atividades de *exportação* e de *importação*.

1.2.3 O modelo Prescritivo

O modelo prescritivo é organizado usando os cinco *pontos de vista* discutidos anteriormente. O modelo de referência define um conjunto de cinco linguagens, cada uma correspondendo a um ponto de vista. Cada linguagem é definida para ser usada na especificação de sistemas ODP daquele ponto de vista, sendo estas: Linguagem da empresa, linguagem de informação, linguagem de computação , linguagem de engenharia e linguagem de tecnologia. Como conseqüência, as linguagens de pontos de vista são usadas para [2]:

- Definir os conceitos e regras para a especificação do sistema ODP daquele ponto de vista.
- Descrever a organização funcional e as relações com outras funções ODP.

Linguagem de empresa

Uma especificação da empresa é usada para definir os objetivos de um sistema ODP em termo dos papéis desempenhados, das atividades empreendidas e das declarações de políticas sobre o sistema ODP.

Linguagem de informação

Uma especificação de informação é usada para definir a semântica de informação de um sistema ODP, i.e., o sentido que um ser humano atribui aos dados armazenados ou trocados entre componentes do sistema , e as necessidades de processamento de informação em um sistema ODP. Ambos são definidos em termos de uma configuração de objetos de informação e da atividade daqueles objetos .

Linguagem computacional

Uma especificação computacional é usada para descrever a decomposição funcional de um sistema ODP, em termos de transparência de distribuição, como:

- Uma configuração de objetos computacionais.
- As atividades que ocorrem nestes objetos.
- As interações que ocorrem entre estes objetos.

Na linguagem computacional todos os objetos computacionais são identificados e, além disso, a única maneira de se ter acesso a um objeto é através de suas interfaces.

Linguagem de engenharia

A linguagem de engenharia compreende conceitos, regras e estruturas para a especificação de um sistema ODP do ponto de vista de engenharia, definindo a infraestrutura necessária para suportar a distribuição funcional do sistema.

Uma especificação de engenharia é usada para descrever a organização de uma infraestrutura abstrata que habilite a execução das funções de um sistema ODP; identificar as abstrações necessárias para o gerenciamento físico da distribuição, comunicação, processamento e armazenamento; identificar e definir os papéis dos diferentes objetos que suportam as funções ODP; e identificar pontos de referência entre os diferentes objetos.

Uma especificação de engenharia é expressa em termos de :

- Uma configuração de objetos de engenharia.
- A atividade que ocorre nestes objetos.
- As interações que ocorrem entre estes objetos.

O modelo de engenharia será descrito em maiores detalhes no capítulo 2.

Linguagem tecnológica

Uma especificação tecnológica expressa a maneira em que as especificações para um sistema ODP são implementadas; identifica as especificações relevantes para a tecnologia na construção de sistemas ODP; provê uma taxonomia para tais especificações; e expressa o conteúdo das declarações necessárias para a implementação da informação necessária para suportar testes.

A mesma esta relacionada com a configuração, instalação e manutenção do hardware e software para suportar os sistemas ODP (Ex.: Sistema operacional local, dispositivos de I/O, portas de acesso a comunicação, etc.).

1.3 O modelo de Computação

O modelo computacional apresenta o sistema ODP como um conjunto de interações de objetos provendo funções de aplicações específicas suportadas pela infraestrutura do modelo de engenharia ODP (capítulo 2). Sendo que os conceitos principais do modelo que apresentaremos serão o de *binding* e *trading*.

1.3.1 Trading

A função de *trading* é provida por um objeto *trader*, sendo que a mesma encontra-se em estado avançado de padronização. O trader consiste em um negociador de serviços que prove um conjunto de operações para registrar serviços oferecidos por outros objetos (função de *import*) e para oferecer esses serviços a outros objetos quando solicitado (função de *export*), entre outros.

Uma discussão sobre traders pode ser encontrada em [7, 12] e sobre federação de traders em [33].

1.3.2 Binding

O processo de estabelecimento de interação entre objetos no modelo de computação é conhecido como *binding*, sendo que o mesmo pode ser de dois tipos possíveis: *Binding implícito* em que o mesmo é feito automaticamente quando uma dada interface é usada pela primeira vez, e o *binding explícito* que é especificado explicitamente via funções de *binding* [1].

No *binding implícito* um objeto obtém um identificador de um outro objeto de alguma maneira (o identificador pode ter sido obtido de um trader ou pode ter sido recebido como resultado de alguma outra interação). Em algum momento não especificado entre o recebimento do identificador da interface e a primeira invocação de uma operação da interface, um processo de *binding* pode ocorrer. O exato instante em que ocorrerá depende de considerações do modelo de engenharia e não do modelo computacional. Quaisquer erros em um processo de *binding* são retornados como falhas na infraestrutura ODP em responder a primeira operação. A duração do *binding* não é especificada na linguagem computacional, sendo que o *binding* pode ser quebrado e refeito com base em considerações de gerenciamento de recursos.

O processo de *binding explícito* é realizado como uma ação por um objeto computacional. A ação *bind* cria um objeto que encapsula os mecanismos de *binding*, permitindo o *binding* ser manipulado. Esta ação toma como argumentos o tipo do *binding* solicitado e um conjunto inicial de identificadores de interfaces a ligar. Se bem sucedida a ação retorna um identificador para uma interface que controla o *binding* que foi criado. A 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*. Se a ação de *binding* falha informações sobre o erros devem ser retornadas como resultado.

Veremos agora alguns aspectos do estabelecimento de um *binding* entre dois objetos, sendo o exemplo contido em [1]. Espera-se no final da interação o estabelecimento de um *binding* entre dois objetos, no caso I e E. Consideremos o cenário passo a passo (o processo de criação pode ser visto na figura 1.2):

- 1) Em um estado inicial temos a existência de um objeto I que irá desempenhar o papel de importador (cliente).
- 2) No passo seguinte é realizada a instanciação de um objeto E que atuará como exportador no exemplo, sendo que o objeto E poderia já estar criado.
- 3) Cria-se então a interface X do tipo $T1$ associada ao objeto E. A interface X consiste num comportamento que pode ser observado de E.
- 4) O objeto E comunica a existência da interface X (por exemplo, pela exportação para um trader da referência de interface da interface *"X"* do tipo referenciado *"T1"*).
- 5) É criada uma interface $Y{:}T2$, sendo que o papel associado a interface Y é o papel de cliente. Dependendo da definição do objeto I a criação da interface Y pode ter sido realizada quando o objeto foi instanciado.
- 6) O objeto I decide interagir com alguma interface do tipo T.
- 7) O objeto I descobre (por exemplo, através de uma operação de *import* do trader) que *"X"* é uma referência de interface do tipo T (visto que $T1$ pode ser subtipo de $T2$).
- 8) Os objetos I e E iniciam então um período de interação. Como exemplo, I pode estar fazendo uma invocação e E estar preparando uma terminação. Ambos os objetos, I e E, possuem o identificador do *binding*, que em um caso simples de operação cliente-servidor pode ser a referência de interface do servidor.

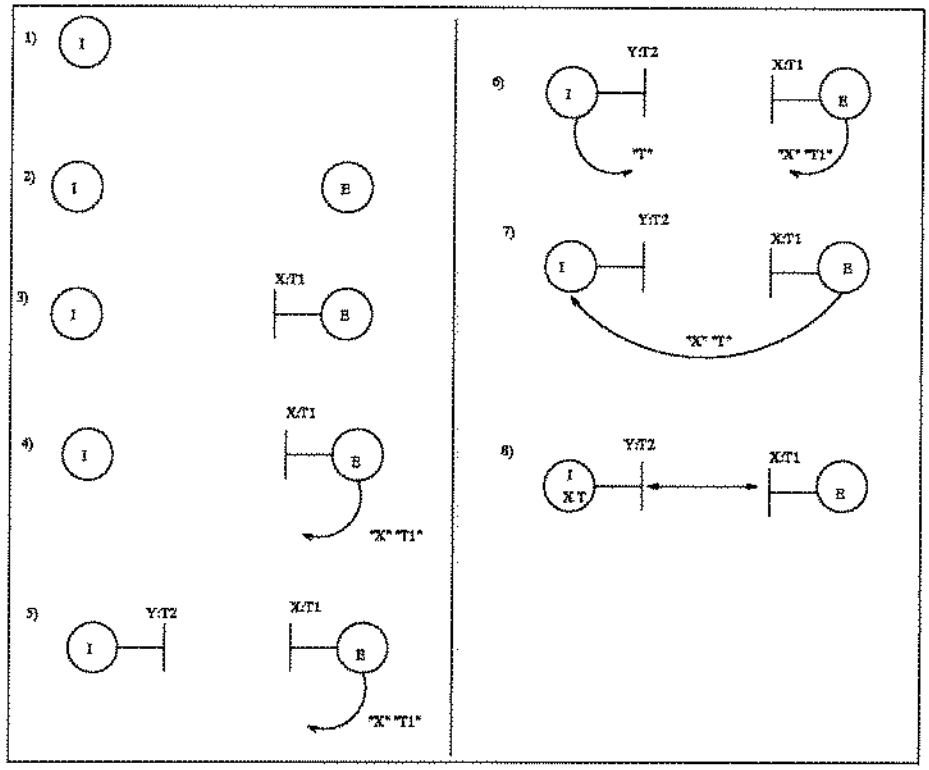

Figura 1.2: **Exemplo de Binding entre dois objetos**

Detalhes específicos do processo de *binding* serão abordados na seção 2.7.1 no capítulo 2, bem como no capítulo 3. Aqui apenas apresentamos a idéia inicial do processo de *binding*.

1.4 Plataforma Multiware

A plataforma multiware consiste em uma plataforma que irá prover uma estrutura de hardware e software para comunicação física para permitir processamento distribuído. Sendo que a mesma irá incluir muitas funções e idéias já conhecidas do RM-ODP. A plataforma á composta de três níveis: *Software/Hardware básico*, *Middleware* e *Groupware*. Descreveremos os mesmos a seguir [35, 38].

- O software/hardware básico é composto por um sistema operacional local e por protocolos de comunicação. Este nível não provê suporte a sistemas distribuídos.

- O nível middleware é responsável por prover facilidades de processamento distribuído ao nível groupware e às aplicações. Este nível é composto de dois subníveis: Subnível de processamento multimídia (que permite a troca de informações multimídia com uma qualidade de serviço específica) e o subnível ODP.

- O nível de groupware provê as funcionalidades demandadas por diferentes classes de aplicação, como CSCW [13] [16] e inteligência artificial distribuída (DAI), entre outros. Serviços típicos suportados por este nível são gerenciamento de diálogo, protocolos de interação e manipulação de documentos multimídia.

veremos a seguir uma descrição do nível middleware da plataforma *Multiware*.

1.4.1 Nível Middleware

O nível *Middleware* na plataforma *Multiware* é responsável pelo fornecimento de um ambiente de processamento distribuído aberto às aplicações, independente do tipo de software/hardware básicos existentes. O modelo para implementação desse nível proposto pode ser visto na figura 1.3 , sendo que possui duas subcamadas, a subcamada inferior que é composta por plataformas comerciais já desenvolvidas (ANSA, DCE e CORBA) e a subcamada superior que oferece suporte aos serviços ODP definidos no modelo de referência RM-ODP.

Neste modelo a implementação da subcamada superior , *subcamada ODP*, agrega abertura as plataformas, providenciando transparência de plataforma independente da orientação (processo ou objeto) e dos serviços disponíveis. Esta subcamada é por sua vez composta por três subcamadas *Gerenciamento ODP*, *Transparência e Segurança*, e *Suporte a ODP*. A primeira oferece os serviços básicos de gerenciamento permitindo a utilização de objetos de engenharia (objetos básicos de engenharia, clusters e cápsulas). Este gerenciamento é auxiliado por funções de repositório de acordo com a especificação ODP.

A segunda oferece transparências e funções de segurança da especificação ODP.

A última (*Suporte a ODP*) pode ser vista na figura 1.4 sendo que seus blocos componentes oferecem serviços da especificação ODP às aplicações. A mesma esta dividida nos seguintes blocos funcionais:

- **Suporte a aplicações.** Provê as funcionalidades básicas de uma plataforma de serviços abertos, tais como a definição e instanciação dos objetos

[13] Computer Support Cooperative Work

(processos) que compõem as aplicações e como estes objetos estão estruturados em subsistemas, a definição dos requisitos desejados de transparência e segurança, e a implantação das comunicações necessárias (*bindings*) entre estes objetos, entre outras.

- **Trading.** Oferece a negociação de serviços entre servidores (exportadores) e clientes (importadores). Exemplos de serviços: exportar um serviço, procurar por um tipo de serviço, selecionar o melhor serviço segundo critérios (estáticos - por exemplo, certas qualidades de serviço e custos- e dinâmicos - por exemplo, tamanho da fila de espera) que são passados em parâmetros apropriados, entre outros.

- **Suporte a grupos.** Provê suporte à cooperação entre membros de um mesmo grupo, como por exemplo, a transmissão de uma invocação de um cliente para membros servidores apropriados do grupo, e a garantia do envio de invocações para membros do grupo numa determinada ordem.

- **Suporte a Transações.** Garante a uma invocação de operação transacional ter as propriedades ACID requeridas.

- **Suporte a Multimídia.** Permite o envio e a recepção de informação multimídia e de tempo real com qualidades de serviço desejáveis.

- **Suporte a OODBMS.** Permite o acesso a bases de dados orientadas a objetos para armazenar principalmente informações multimídia.

Apresentaremos nas duas subseções seguintes uma visão superficial do ORB (*Object Request Broker*) e dos serviços fornecidos pelo sistema operacional SunOS. O primeiro por ser adotado inicialmente no projeto da camada middleware da plataforma *Multiware*, e o segundo por ter sido adotando em nossa implementação (discutida no capítulo 4).

1.4.2 ORB

A proposta central da OMG (*Object Management Group*) é a de criar um padrão que realize interoperabilidade entre aplicações desenvolvidas independentemente em redes heterogêneas de computadores. Com esse objetivo o OMG adota especificações de interfaces e protocolos que definem uma arquitetura de gerenciamento de objetos OMA (*Object Management Architecture*) para suportar aplicações interoperáveis baseada em objetos distribuídos interoperáveis. As especificações estão baseadas em tecnologias existentes. Como parte principal da arquitetura

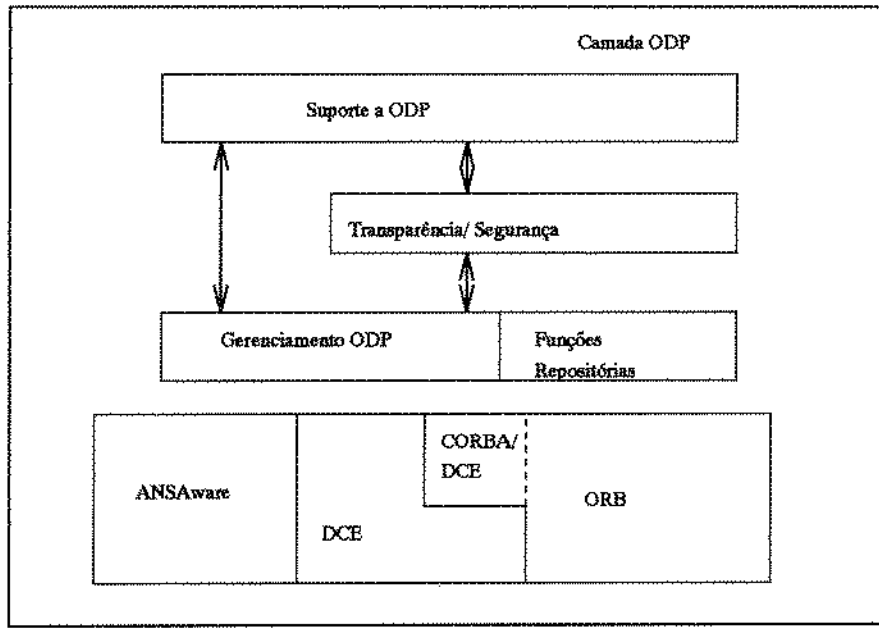

Figura 1.3: **Estruturação do nível Middleware**

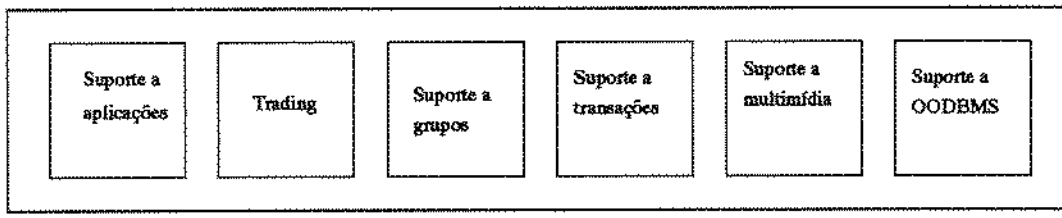

Figura 1.4: **Estruturação da camada de suporte a ODP**

OMA temos o ORB que dá suporte a interação entre objetos, sendo que uma descrição detalhada do mesmo pode ser encontrada em [42].

Como definido pela OMG, o ORB fornece um mecanismo para que objetos realizem requisições e recebam respostas transparentemente. O ORB provê interoperabilidade entre aplicações de diferentes máquinas em ambientes distribuídos heterogêneos e permite a interconexão de múltiplos sistemas de objetos.

Para realizar uma requisição um cliente pode fazer uso da interface de invocação dinâmica (DII - *Dynamic Invocation Interface*) ou um *stub* IDL (*Interface Definition Language*), no caso de DII a mesma interface pode ser usada independente da interface do objeto, enquanto que com o uso de stubs a especificação do *stub* depende da interface do objeto destino. O cliente pode ainda interagir diretamente com o ORB para algumas funções.

Stubs fornecem um mecanismo para interação entre interfaces descritas em tempo de compilação, provendo uma linguagem para tais descrições , IDL (*Interface Description Language*), sendo portanto estáticas as interações, enquanto DII fornece um mecanismo para interação dinâmica entre objetos.

O uso de ORB como plataforma primária no desenvolvimento da *Multiware* deve-se a ser uma plataforma simples, em que existem conceitos básicos que permitem o fornecimento de transparência de distribuição, e interação entre objetos em ambientes heterogêneos. Além do que, pela facilidade de se adicionar objetos, facilita a elaboração e implementação das funcionalidades ODP, além de que com a elaboração de *object services* [43, 44] que estão sendo incorporados a CORBA [42] facilita o desenvolvimento e implementação do modelo RM-ODP.

O ORB pode substituir o mecanismo tradicional de RPC [14], com a vantagem de fornecer transparência de distribuição. O mesmo, a exemplo de RPC, fornece uma linguagem para descrição de interface, IDL, que é orientada a objetos, enquanto XDR, fornecida por RPC, não é orientada a objetos.

1.4.3 Serviços oferecidos pelo SunOS

O sistema operacional SunOS é derivado do sistema 4.2BSD Unix, estendendo o mesmo com os serviços de rede providos no SunOS. Os serviços básicos de rede presentes no SunOS foram inicialmente criados para o 4.2BSD Unix. Aqui também aparece a idéia de transferir certas tarefas do núcleo do sistema operacional (*kernel*) para *daemons* especializados na execução de certas tarefas. Basicamente, os serviços de rede oferecidos pelo SunOS utilizam uma arquitetura baseada em *daemons* servidores.

O mecanismo de comunicação utilizado pelos serviços de rede do SunOS se baseia em um esquema de RPC's, bem definidos pela própria Sun, e cujo protocolo

se tornou um padrão *de facto* dentro da Internet. Este protocolo utiliza a forma de representação canônica XDR que também também é especificada pela Sun [56].

Os *daemons*, que funcionam em toda máquina que implementa um determinado serviço, funcionam em cima destes protocolos RPC/XDR e assim qualquer aplicação que siga tais protocolos pode ser considerada uma aplicação de rede.

Como principais serviços providos pelo SunOS temos os seguintes [56, 36]:

- **Remote Procedure Call (RPC).** A chamada remota de procedimento é uma biblioteca de procedimentos que permite um processo cliente solicitar a um processo servidor a execução de um procedimento. A execução se processa como se a chamada tivesse ocorrido dentro do próprio espaço de endereçamento do processo cliente, mesmo quando os dois processos estão em máquinas diferentes.

- **Network File System (NFS).** Consiste em um serviço, independente de sistema operacional, que permite aos usuários compartilharem arquivos dentro de uma rede, de forma transparente. O serviço oferece ainda mecanismos que permitem autenticação baseada em DES (*Data Encryption Standard*).

- **Portmapper.** Consiste em um serviço fundamental para o funcionamento dos serviços baseados em RPC. Funciona como um cartório de registros que mantém um mapeamento entre portas (canais lógicos de comunicação) e serviços em uma dada máquina. O serviço oferece uma interface para permitir que um cliente procure a porta associada a um dado serviço oferecido por um servidor suportado pelo *portmapper*.

Uma descrição das características de rede do SunOS pode ser encontrada em [56].

Capítulo 2

O modelo de engenharia

Nesse capítulo realizaremos o detalhamento do modelo de engenharia do RM-ODP aprofundando no que consiste o mesmo e quais são seus objetos componentes. É apresentado ainda uma discussão do modelo de engenharia como uma API (*Application Programming Interface*) para sistemas operacionais distribuídos, e uma análise de seus objetos componentes em termos de conceitos de sistemas operacionais. Finalmente apresenta-se as funções descritas no RM-ODP como necessárias para suportar ODP no RM-ODP, e uma discussão do que acreditamos ser um interelacionamento entre as mesmas de forma a servir de base aos capítulos 3 e 4.

Nossas contribuições aqui podem ser notadas na forma de estruturar o texto e na análise do modelo sob a ótica de uma API uniforme para sistemas operacionais distribuídos durante o capítulo e no estudo do interelacionamento entre as funções do RM-ODP.

2.1 Introdução

O modelo de engenharia ODP pode ser visto como uma máquina abstrata que é colocada sobre os diversos ambientes existentes onde o sistema ODP irá "conviver" provendo uma arquitetura comum.

Como tarefa principal da infraestrutura ODP temos a de prover transparência de localização de tal forma que um programador de aplicação não tenha necessidade de conhecer a natureza da distribuição do sistema básico. Outra tarefa consiste em mapear objetos de aplicação em recursos do sistema operacional básico como processos, threads, espaço de endereçamento e canais de comunicação [39].

A infraestrutura ODP pode ser vista como uma camada que possuindo objetos

computacionais,(como vistos na figura 2.1) mapeia os mesmos enquadrando-os em um modelo executável, ou em um nível mais restrito, enquadra-os sobre o(s) sistema(s) operacional(ais) disponível(is), como pode ser visto na figura 2.2.

No que se relaciona ao modelo de engenharia temos que o mesmo provê uma estruturação de tal infraestrutura (a qual pode ser vista na figura 2.3) fornecendo um mecanismo para prover o mapeamento entre os níveis abstrato (modelo computacional) e real (modelo de engenharia), formando assim uma "camada de mapeamento" entre tais níveis. Descreveremos agora de forma suscinta o comportamento de tal modelo (figura 2.3), visto que uma análise mais detalhada de cada objeto será o tema do restante do capítulo.

Podemos notar pela figura 2.3 uma estruturação por níveis no que se relaciona a organização dos objetos do modelo e uma organização de interação entre os mesmos. A estruturação dá-se em níveis de **cápsulas**, **clusters** e **objetos básicos de engenharia** agregados em cada **nó**. Uma **cápsula** consiste como que em uma máquina virtual que provê um conjunto de recursos e espaços de endereçamento disjuntos das demais cápsulas no nó, no seu interior. No interior da cápsula podemos notar objetos **clusters** que possuem um conjunto de **objetos básicos de engenharia** cada um. Os **objetos básicos de engenharia** formam a unidade básica do modelo, sendo que os **clusters** são agrupamentos de objetos básicos de engenharia para o propósito de *checkpoint* [1], *desativação* [2], *migração* [3], *replicação*, *reativação* [4] e *recuperação* [5].

Um cluster está contido totalmente em uma cápsula sendo que é gerenciado por um objeto gerente de cluster associado ao mesmo (CLM), cápsulas da mesma forma estão contidas totalmente no espaço de um nó sendo que cada cápsula é gerenciada por um objeto gerente de cápsula (CPM) que determina a realização de suas tarefas.

O nó consiste no "domínio de recursos" sendo que o mesmo é gerenciado por um objeto **núcleo** que tem como funções coordenar as funções de processamento, armazenamento e comunicação a serem usadas pelos outros objetos de engenharia do nó a que pertencem.

A nível de comunicação entre objetos no modelo a mesma dá-se da seguinte forma. A ligação dos gerentes de cápsula e de cluster (CPM e CLM) ao núcleo dá-se através de *interfaces de controle*. Os objetos básicos de engenharia no

[1] Um objeto template derivado do estado e estrutura de um objeto que está em estado consistente em relação a alguma invariante.

[2] *Checkpoint* de um *cluster*, seguido da deleção do cluster.

[3] Mover o cluster para outra cápsula.

[4] *Cloning* de um cluster depois da desativação.

[5] *Cloning* de um *cluster* depois da falha ou deleção do cluster.

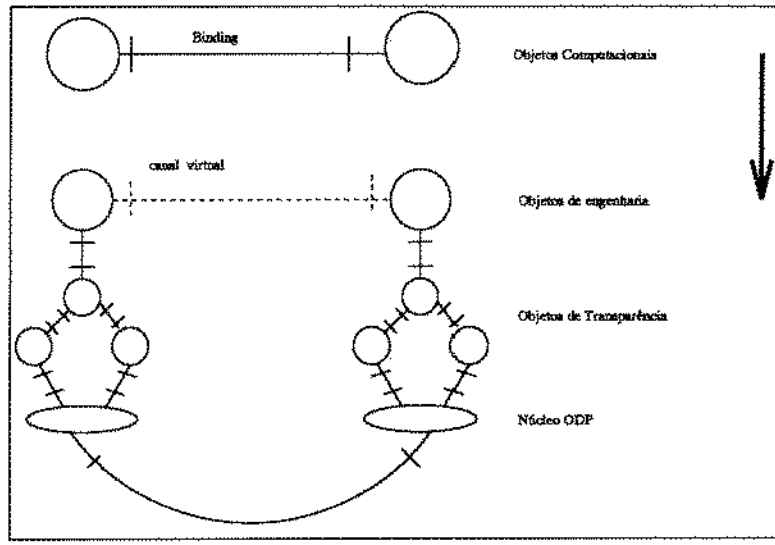

Figura 2.1: **Infraestrutura básica**

interior de um mesmo cluster usam de interfaces, usando de **identificadores de interface de engenharia** [6], para se comunicarem entre si, e quando em clusters diferentes, sejam na mesma cápsula ou não, fazem uso de **canais** para se comunicarem entre si. Tais canais são referenciados via uma **referência de interface de engenharia** [7], sendo que quando se agrupam um conjunto de nós sobre o controle de um gerente de referências de interface de engenharia temos um domínio de gerenciamento de interfaces de engenharia.

No que se relaciona aos objetos do modelo tentaremos apresentar a descrição de cada objeto, suas funcionalidades e o enquadramento de cada um sobre uma série de características que enumeramos:

- **Espaço de endereçamento:** Refere-se ao espaço de endereçamento provido e se o mesmo é comum entre objetos (compartilhado) ou individual (privado).

- **Comunicação:** Identifica a forma pela qual dá-se a comunicação entre diferentes objetos do mesmo tipo, ou seja quais mecanismos utilizam para comunicar-se entre si e que conseqüência acarretará no desempenho.

[6] Um identificador, em um contexto de nomes de uma cápsula, para uma interface de um objeto de engenharia, usada para o propósito de interação.

[7] Um identificador, no contexto de um domínio de gerenciamento de referências de interface de engenharia, para uma interface de um objeto de engenharia que está disponível para *binding*.

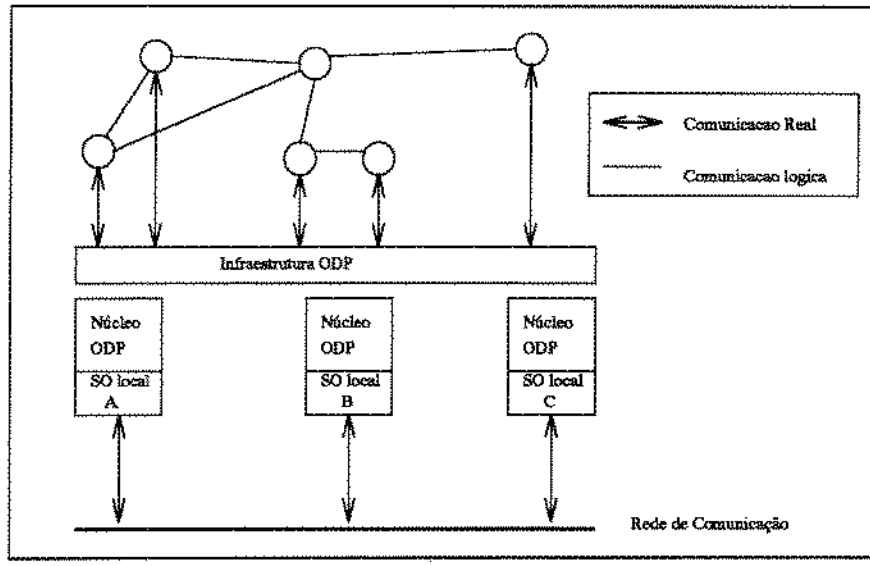

Figura 2.2: **Infraestrutura básica**

- **Administrador:** Identifica que objeto é responsável pela administração do objeto ao longo de seu tempo de vida.
- **Configuração:** Refere-se a se novos objetos de um determinado tipo (Objetos básicos de engenharia, cluster , ...) podem ser incorporados dinamicamente no grupo (incremental) ou se não podem (estática).
- **Homogeneidade:** Refere-se a que em um conjunto (grupo) de objetos todos sejam do mesmo tipo (homogêneo), ou permite-se objetos de tipos diferentes no conjunto (heterogêneo).
- **Criador:** Refere-se a qual objeto é responsável pela instanciação ou criação do objeto.
- **Identificador:** Refere-se ao escopo de validade dos identificadores de objetos.

Um resumo para os objetos de tais características pode ser visto nas tabelas 2.1 e 2.2.

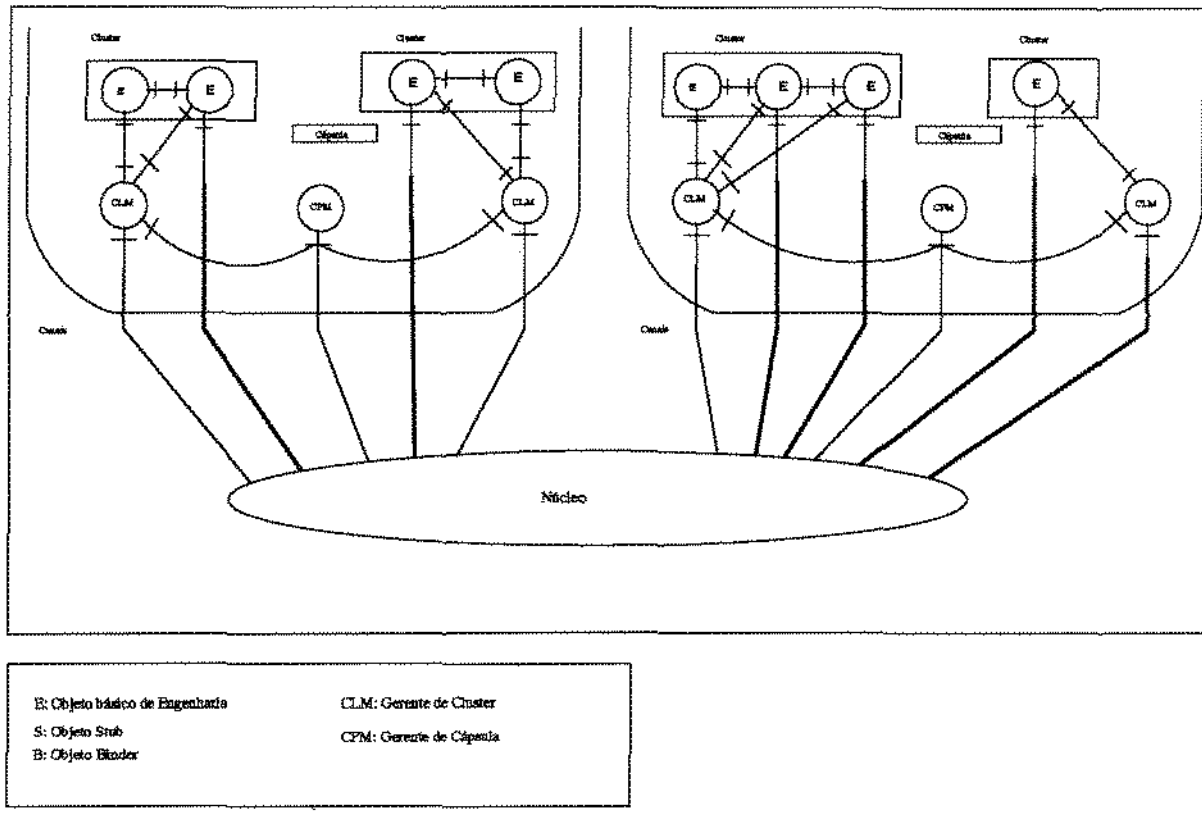

Figura 2.3: **modelo de Engenharia do RM-ODP**

Característica de objetos			
Objeto	Espaço de Endereçamento	Comunicação	Identificador
Thread	Compartilhado	Memória compartilhada, IPC	BEO
BEO	Individual	IPC	Cápsula
Cluster	Individual	Canal	Cápsula
Cápsula	Individual	Canal	Domínio de Ref. de interface
Nó	Individual	Canal e Interceptador	Domínio de comunicação

Tabela 2.1: **Características dos Objetos do RM-ODP**

Característica de objetos				
Objeto	Criador	Administrador	Configuração	Homogeneidade
Thread	Núcleo	Núcleo	incremental	Homogêneo
BEO	CPM	CLM e Núcleo	incremental	Heterogêneos
Cluster	CPM	CLM	incremental	Heterogêneos
Cápsula	Núcleo	CPM	incremental	Heterogêneos
Nó	Externo	Núcleo	estática	Heterogêneos

Tabela 2.2: **Características dos objetos do RM-ODP (Continuação)**

2.2 Descrição de threads

Threads fornecem um modelo que permite que aplicações explorem seu paralelismo.

Como característica básica temos que threads em um objeto básico de engenharia terão espaço de endereçamento compartilhado, usando mecanismos de memória compartilhada e mensagens para comunicação. Um conjunto de threads está agrupado em um objeto básico de engenharia, sendo os mesmos administrados pelo núcleo. Tendo ainda configuração incremental em um objeto básico de engenharia, visto que podem ser criados novos threads em um objeto básico de engenharia dinamicamente pelo núcleo. Threads em um dado objeto básico de engenharia são do mesmo tipo, ou seja homegêneos, e seus identificadores são locais ao objeto básico de engenharia que os contém.

2.3 Descrição do Objeto Básico de engenharia

Um objeto básico de engenharia consiste em um objeto de engenharia que necessita do suporte da infraestrutura de distribuição [3]. Sendo que consiste na representação de um objeto computacional incluindo seu *contrato de ambiente.*

Em um *cluster* um objeto básico de engenharia está sempre ligado ao núcleo (em uma interface provendo a função de gerenciamento de nó) e ao gerente do *cluster* a que pertence. As outras interfaces de um objeto básico de engenharia encontram-se ou ligadas a outro objeto básico de engenharia no mesmo *cluster,* ou ligadas através de canais a outros objetos de engenharia em outros clusters. Podemos ver tal estrutura de ligações na figura 2.4 (os componentes do canal serão descritos na seção 2.7.1).

Um objeto básico de engenharia pode ser visto como sendo a unidade de

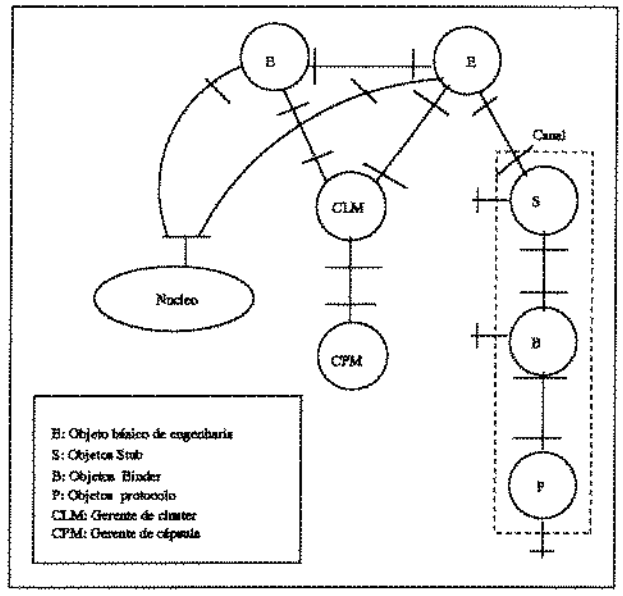

Figura 2.4: **Objeto Básico de Engenharia**

execução do modelo, sendo que o mesmo pode conter vários threads simultaneamente, ou seja, ser multithread.

Em relação às características básicas de um objeto básico de engenharia temos que objetos básicos de engenharia distintos possuem espaços de endereçamento distintos. Sendo que são criados pelo gerente de cápsula no processo de instanciação do *cluster*, ficando a administração a cargo do gerente de cluster e do núcleo. Identificadores de objetos básicos de engenharia são válidos apenas na cápsula que os contém, e a comunicação entre objetos básicos de engenharia no mesmo cluster dá-se via mecanismo de IPC (*inter-process communication*) dispensando o uso de canais. Os mesmos se agrupam em clusters, sendo que a configuração de objetos básicos de engenharia em um cluster pode ser incremental, devido a um objeto básico de engenharia poder criar outro no mesmo cluster em que está. Os objetos básicos de engenharia em clusters não precisam necessariamente ser do mesmo tipo.

2.4 Descrição de um Cluster

Um *cluster* consiste de uma configuração de objetos básicos de engenharia formando uma unidade para o propósito de desativação, *checkpoint*, reativação,

recuperação e migração [3].

Um *cluster* contém um conjunto de objetos básicos de engenharia associados com um gerente de cluster. Sendo que cada membro do cluster pode ter uma interface suportando a função de gerenciamento de objeto. Cada interface de gerenciamento de objeto está ligada ao gerente de cluster, sendo este ligado a um gerente de cápsula. Um cluster sempre está contido em uma única cápsula.

A instanciação de um template de cluster (incluindo *cloning* em casos especiais) é realizada por um gerente de cápsula, sendo que um template de cluster especifica a configuração de objetos no cluster e qualquer atividade necessária para instanciá-los e estabelecer *bindings*.

Se o template é um *checkpoint* de cluster, a instanciação (isto é , *cloning*) habilita o novo cluster a atuar como um substituto para o cluster original do qual o template de cluster foi derivado. Nesse caso o processo de *cloning* inclui o restabelecimento de qualquer *binding* que existia para o cluster original.

A instanciação de um *cluster* cria um *cluster* ativo. A desativação de um *cluster* ativo destrói o *cluster* ativo (deleção de objeto) e cria um *cluster* desativado (criação de objeto) representando o mesmo estado. Reativar um *cluster* desativado cria um *cluster* ativo representando o mesmo estado.

Um *cluster* pode ser destruído quando o mesmo é incapaz de atividade futura (o *cluster* não tem qualquer atividade e não tem interfaces capazes de inicializar atividades futuras).

Um cluster é responsável pela sua própria segurança, mas pode contudo ser assistido pelas funções de segurança em tal tarefa. Sendo ainda responsável pelo gerenciamento de referências de interface de engenharia para as interfaces de objetos no cluster, mas podendo ser auxiliado pela função de gerenciamento de referências de interface (seção 2.7.2).

Um cluster pode ser visto como um conjunto de processos com alguma afinidade que serve como unidade de migração, desativação, ativação e replicação. Conforme o RM-ODP [3] um cluster poderia ser , por exemplo, um segmento de memória virtual contendo objetos.

Em relação às características de um cluster temos que o mesmo é criado (instanciado) por um gerente de cápsula, sendo administrado por um gerente de cluster que é "criado" junto com o cluster.

Em um cluster o espaço de endereçamento é distinto do espaço de endereçamento dos demais clusters, para se permitir migração, ativação e desativação. Identificadores de clusters são válidos no contexto de uma cápsula, sendo que a comunicação entre objetos básicos de engenharia pertencentes a diferentes clusters dá-se via canal. Clusters agrupam-se em uma cápsula, sendo que a configuração é incremental visto que novos clusters podem ser instanciados ou removidos da

cápsula. Podendo existir ainda clusters de diferentes tipos em uma cápsula.

2.5 Descrição de uma Cápsula

Uma cápsula consiste em uma configuração de objetos formando uma unidade com o propósito de encapsulamento de processamento e armazenamento [3].

Uma cápsula consiste de :

- Um conjunto de um ou mais *clusters*.
- Gerentes de *clusters*, um para cada *cluster* na cápsula.
- Objetos do canal (*stub*, *binder* e protocolo - seção 2.7.1) para cada canal ligado a uma interface de um objeto básico de engenharia em qualquer dos clusters.
- Um gerente de cápsula onde cada um dos gerentes de cluster desta cápsula estão ligados .

Um gerente de cápsula tem uma interface que provê a função de gerenciamento de cápsula. Uma cápsula sempre está contida em um único nó.

Todos os objetos em uma cápsula estão ligados a uma mesma interface de gerenciamento de nó. Sendo que objetos em outras cápsulas estão ligados a diferentes interfaces de gerenciamento de nó.

Em uma cápsula, o gerente de cápsula está ligado a cada interface de gerenciamento de *cluster* de cada gerente de *cluster*.

A instanciação de uma cápsula é realizada pelo núcleo usando um *template* de cápsula que especifica a configuração inicial de objetos na cápsula, incluindo clusters, gerentes de clusters, canais (stubs, binders e protocolos) e gerente de cápsula.

Uma cápsula é um contexto de nomes para identificadores de interface de engenharia. O modelo de referência não requer que os identificadores sejam válidos em um escopo maior. Sendo que para propósito de *binding* usam-se referências de interface de engenharia para comunicação de conhecimento de interfaces de objetos de engenharia entre cápsulas.

É importante destacar que o modelo de referência não restringe a maneira pela qual gerentes de cápsula e de cluster interagem entre si e com o núcleo, de forma a permitir implementações diferentes do mesmo.

Uma cápsula pode ser vista como uma máquina virtual ou um espaço de endereçamento e de recursos, como o conceito de *actor* do sistema operacional

Chorus [51, 10, 26] encapsulando os recursos oferecidos pelo nó e distribuindo os mesmos.

Cápsulas são criadas (instanciadas) pelo núcleo, sendo que podem ser terminadas ou pelo gerente de cápsula (CPM), ou pelo núcleo quando este detecta "inatividade" na cápsula e chama a função de deleção do CPM para tal fim. Uma cápsula é administrada pelo gerente de cápsula. No RM-ODP a cápsula consiste na unidade de alocação de recursos e encapsulação e isso inclui memória, logo cápsulas distintas terão espaços de endereçamento distintos. Cápsulas se agrupam em nós, sendo que a comunicação entre as mesmas dá-se via canal (com o uso de referências de interface de engenharia). Sendo que podem ser criadas cápsulas dinamicamente em um nó, tendo assim configuração incremental, podendo ainda termos cápsulas de diferentes tipos (heterogenidade).

2.6 Descrição do nó

O nó consiste de um núcleo e de um conjunto de cápsulas. Todos os objetos em um nó compartilham funções de processamento, armazenamento e comunicação comuns. Sendo que um nó é membro de um ou mais domínios de gerenciamento de referências de interface. A estrutura de um nó pode ser vista na figura 2.3.

O objeto núcleo provê um conjunto de interfaces de recursos do nó, uma para cada cápsula no nó.

O procedimento de instanciação de um nó está fora da estrutura ODP. Devendo, contudo, resultar em:

- Introdução de um objeto núcleo do nó e funções associadas de processamento, armazenamento e comunicação.
- Introdução de qualquer função de *trading* necessária para o processo de instanciação.
- Criação de qualquer canal necessário como parte da configuração inicial do nó (Exemplo: Para suportar objetos como um relocador - seção 2.7.1).

O conjunto de objetos protocolos introduzidos durante a instanciação do nó determina o conjunto de domínios de comunicação a que o nó pertence.

A interface de recursos do núcleo consiste de:

- Operações de escalonamento para uso de threads.
- Operações de controle de comunicação para controlar canais.

- Operações para realizar chamadas remotas via canais.
- Operações de *binding* para estabelecer caminhos de comunicação para se criar canais.
- Operações de acesso e uso de timers para se gerenciar temporizações.
- Operações de uso e acesso a segurança de chave para prover segurança.

Um nó pode ser visto como sendo uma estação de trabalho em que residem clusters em cápsulas, tal como o conceito de *site* do sistema operacional Chorus [51], sendo que tal estação pode ser multiprocessada, cabendo ao núcleo mascarar e tratar tal fato fazendo bom uso dos recursos oferecidos.

O processo de criação de um nó, como já foi dito, é externo ao padrão sendo que o mesmo é "introduzido". Um nó é administrado pelo núcleo provendo a função de gerenciamento de nó, sendo que cada nó possui espaço de endereçamento distinto. Identificadores de nós são válidos em um domínio de comunicação. A comunicação entre nós é realizada via canais, e se for ultrapassado um domínio de comunicação faz-se uso ainda de interceptadores no canal. Nós agrupam-se em domínios de comunicação, sendo que a configuração de um nó é estática uma vez que o mesmo é introduzido e não se altera. Nós em um domínio de comunicação podem ser de tipos diferentes, visto que estamos em um ambiente aberto.

2.7 Mecanismos de comunicação

Descreveremos agora os mecanismos utilizados pelos objetos do modelo para comunicar-se entre os mesmos, no caso interfaces, referências de interface de engenharia e canais, descreveremos cada um explicitando seu uso em cada caso.

Objetos no interior de um cluster comunicam-se via suas interfaces, contudo quando ultrapassado esse contexto necessita-se do uso de canais com vista a se ter um identificador que univocamente identifique um objeto em um contexto de nomes (um domínio de referências de interfaces de engenharia) utiliza-se para tal fim uma referência de interface de engenharia. Apresentaremos agora tais conceitos a princípio o de canal, seguido do conceito de referência de interface de engenharia, explicitando seus componentes.

2.7.1 Descrição do Canal

Um canal consiste em uma configuração de objetos *stub*, *binder*, protocolo e interceptador interconectando um conjunto de objetos de engenharia, através do

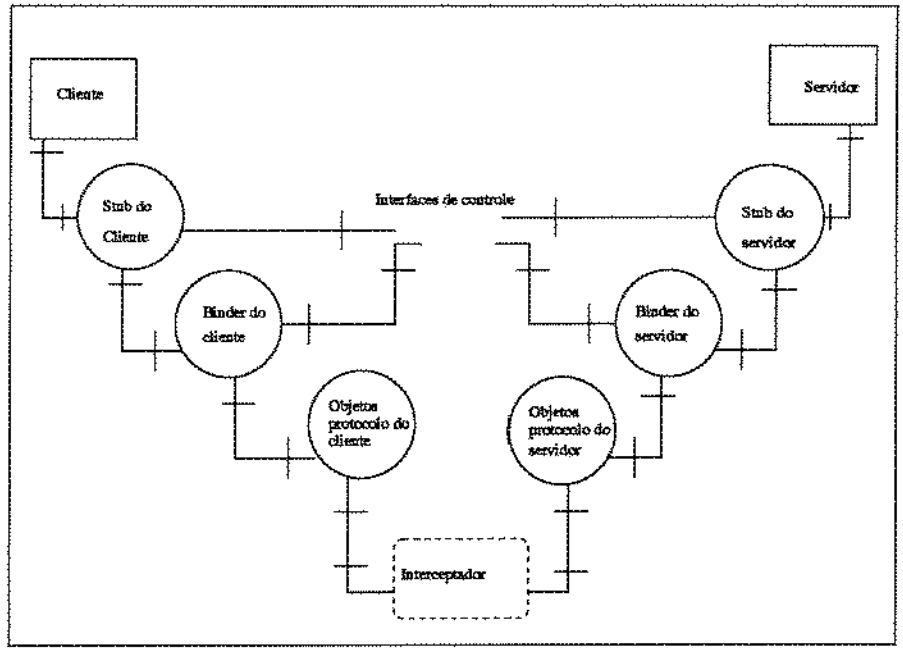

Figura 2.5: **Canal Simplificado**

qual interações podem ocorrer. A configuração consiste em um grafo acíclico em que cada caminho no grafo é formado por uma sequência de objetos stub-binder-protocolo-protocolo-binder-stub ou com um objeto interceptador entre os objetos protocolo. A estrutura pode ser vista na figura 2.5 que mostra um canal simples entre dois objetos e na figura 2.6 que mostra um canal com estrutura multiponto. Sendo que objetos stub, binder, protocolo e interceptadores em um canal podem ter ligações com outros objetos fora do canal , por exemplo provendo funções de coordenação, como relocação (via relocador - figura 2.7).

Um canal suporta interação transparente de distribuição entre objetos de engenharia, incluindo :

- Execução de operações entre um objeto cliente e um objeto servidor.
- Um conjunto de objetos estar ligado a outro conjunto de objetos, permitindo comunicação multiponto.
- Interações *streams* envolvendo múltiplos objetos produtores e objetos consumidores.

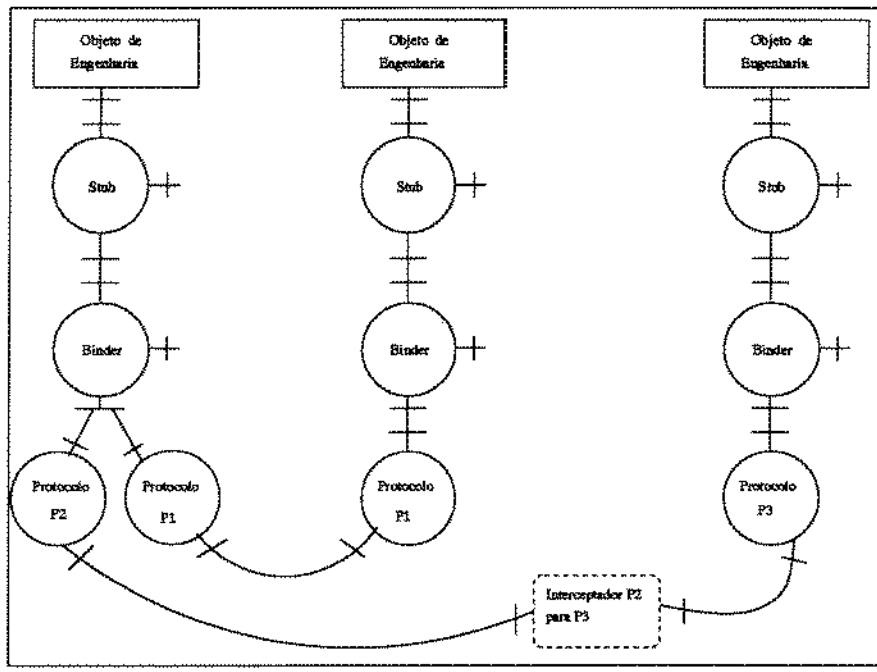

Figura 2.6: **Canal Múltiplo entre Objetos**

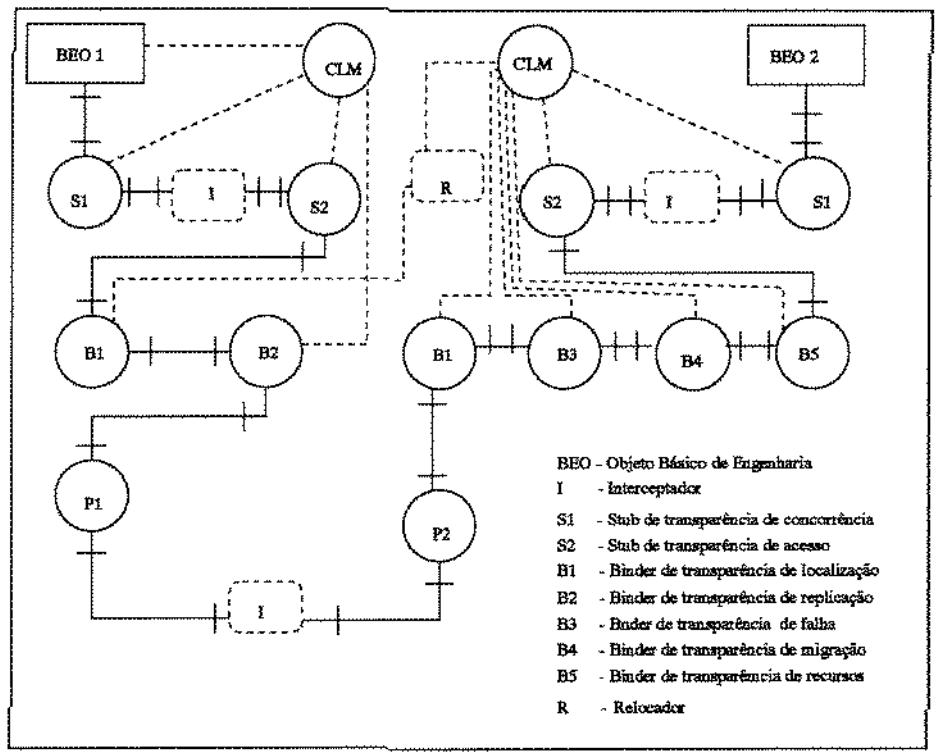

Figura 2.7: **Canal suportando diversas Transparências**

Descreveremos agora os diversos objetos que compõem a estrutura de um canal. O processo de estabelecimento do canal em si será descrito posteriormente quando tratarmos da função de gerenciamento de nó (seção 3.2.3).

Objeto protocolo

Um objeto que comunica-se com outros objetos protocolos para permitir interação entre objetos de engenharia em outros nós.

Um protocolo é identificado em uma localização no tempo e por sua localização no espaço, contudo objetos protocolos diferentes podem ocupar a mesma localização no espaço em tempos diferentes (por exemplo devido a mudança no endereço de rede). Um objeto protocolo possui uma interface para a interação com o binder e uma interface de comunicação. Quando objetos protocolo no canal são de diferentes tipos eles requerem um interceptador para prover conversões entre os protocolos.

Objeto Interceptador

Um objeto que realiza checagem para oferecer ou monitorar políticas em interações permitidas entre objetos em domínios diferentes, e realiza transformações para mascarar as diferenças na interpretação de dados por objetos em domínios diferentes. O mesmo permite transparência de federação.

Objeto Stub

Um objeto com funções de conversão (isto é, *marshalling* e *unmarshalling* de dados para suportar interação transparente de acesso entre objetos de engenharia), contribuíndo assim para permitir transparência de distribuição pela adaptação da informação trocada.

Objetos de engenharia quando em interação têm uma interface conectada a um stub, sendo que os mesmos provêem a conversão dos dados que transitam pela interação. Objetos stub podem ainda aplicar controle e gerar registros [8], bem como interagir com objetos fora do canal se necessário. Como interfaces um objeto stub possui uma interface de apresentação para ser usada pelo objeto de engenharia, uma interface de controle para gerenciamento de qualidade de serviço e uma interface para interação com o binder.

Quando stubs interconectados usam diferentes sintaxes de transferência eles devem ter um interceptador no caminho entre eles que possa transformar os dados

[8] por exemplo para segurança ou contabilidade

de uma sintaxe para outra (veja figura 2.7).

Objeto Binder

Os binders em um canal gerenciam a integração fim-a-fim e a qualidade de serviço do canal. Quando necessário provêem transparência de relocação pela monitoração de falhas de comunicação e reparam os *bindings* "desfeitos". Binders em um canal podem interagir com objetos fora do canal para obter a localização de dados (por exemplo com um relocador).

Como interfaces um binder possui uma interface de interação com um stub, uma interface de interação com um objeto protocolo, e uma interface de controle. Sendo que a interface de controle permite a modificação da configuração do canal e a destruição do mesmo.

Podemos ter em um canal diversos tipos de binders dependendo da seletividade de transparência esperada (como visto na figura 2.7).

Relocador.

Não consiste em um objeto natural do canal, mas como agregado provê algumas operações (notadamente funções de coordenação) para:

- validar uma referência de interface de engenharia.
- Registrar as maneiras para reativar o *cluster* contendo uma interface, quando este *cluster* esteja desativado.
- Registrar uma modificação na localização da interface. Se uma requisição é feita para validar uma referência de interface de engenharia para qual há uma modificação de localização conhecida, uma referência revisada é retornada. Se uma requisição é feita para validar uma referência de interface de engenharia para uma interface que está em um *cluster* desativado, e o relocador tem uma maneira de reativar o *cluster*, o *cluster* será reativado e uma referência de interface de engenharia revisada será retornada, indicando a nova localização da interface. Se uma requisição é feita para validar uma referência de interface não conhecida do relocador um erro é indicado.

2.7.2 Referências de Interface de Engenharia

Uma *referência de interface de engenharia* provê a informação necessária para permitir , no tempo e no espaço, o *binding* de uma interface de engenharia com

outra. A localização de uma interface é especificada por objetos referência de interface de engenharia.

Referências de interface de engenharia consistem em identificadores não ambíguos em um **domínio de gerenciamento de referências de interface** (um conjunto de nós) sendo gerenciados pela função de gerenciamento de referências de interface de engenharia.

Uma **localização de engenharia** é definida em termos da localização de interfaces de comunicação e de *templates* de *binding* para os objetos *stub* e *binder* ligados a objetos protocolo suportando as interfaces de comunicação.

Um *cluster* que é desativado e reativado (possivelmente em cápsulas diferentes) tem a propriedade de que as interfaces de seus objetos básicos de engenharia estão associadas a diferentes localizações de engenharia em momentos distintos. Em *clusters* estacionários, as interfaces estão associadas com as mesmas localizações durante o tempo de vida do *cluster*.

Uma **referência de interface de engenharia** contém dados para estabelecimento de *binding* para a interface de engenharia, como:

- Um *template* de canal apropriado, descrevendo os interceptadores, objetos protocolo, binders e stubs que podem ser selecionados quando configurando um canal para suportar *binding*.

- A localização no tempo e no espaço das interfaces de comunicação as quais o processo de *binding* pode ser iniciado.

sendo que se o núcleo suportar interfaces de engenharia fornecendo diferentes protocolos (processos *bindings* e sintaxe de transferência) a interface de engenharia irá indicar as combinações válidas que podem ser selecionadas em algum *binding* particular, sendo que diferentes *bindings* podem fazer diferentes seleções.

A maneira pela qual as interfaces de comunicação associadas com as interfaces referenciadas são determinadas depende das transparências selecionadas para a interface.

Os binders em um canal detectam quando alguma relocação tenha invalidado o canal. Quando necessário, binders provêem transparência de relocação pelo uso dos dados nas referências de interface para restabelecer o canal. Os dados contidos em uma interface de engenharia são suficientes para assegurar que nenhuma das ações de relocação possam invalidar o mapeamento correto entre referências de interface de engenharia para referências de comunicação.

Um interceptador que está entre domínios de referências de interface de engenharia mantém um mapeamento entre referências de interface de engenharia entre estes domínios.

O identificador de engenharia de interface em uma referência de interface é checado pelos *binders* para assegurar que interações ocorram somente em interfaces de comunicação que suportam as interfaces referenciadas. Este processo de validação pode iniciar reativação, recuperação, migração ou replicação de *clusters* contendo a interface referenciada. O processo de validação detecta erros em configuração de canais, incluindo erros relacionados a falhas em gerenciamento da rede.

Uma **referência de interface de engenharia** consiste em uma configuração de:

- Uma *descrição da interface de engenharia* para a interface sendo referenciada.
- Uma *descrição da interface de engenharia* para um conjunto de *relocadores*.
- Uma relação ordenada do conjunto de relocadores.

Uma **descrição de interface de engenharia** consiste em uma configuração de:

- Um *"nonce"* para checagem fim-a-fim da consistência de um *binding*.
- Informação sobre ser membro de um *grupo de réplica* [9].
- Um identificador não ambíguo para a interface (*Identificador de interface de engenharia*, válido na cápsula).
- Uma *descrição da localização* para cada réplica da interface.

A **descrição da localização** é uma configuração de descrições, uma para cada protocolo para o qual um *binding* pode ser estabelecido, cada descrição sendo uma configuração de :

- Um identificador para o tipo de protocolo.
- Informação de endereçamento para o objeto protocolo.
- Uma descrição da capacidade de *qualidade de serviço* do objeto protocolo.
- Os nomes das operações acessíveis via o objeto protocolo.

[9] Uma referência de interface que não pertence a grupo de réplica é tratada como pertencendo a um grupo de réplica de tamanho um.

O núcleo é responsável pelo mapeamento de referências de interface de engenharia para stubs e interfaces de comunicação. Binders em cooperação com a função de relocação atualizam o mapeamento quando objetos são relocados.

Relocadores podem ser especificados para uma única interface ou comum a diversas interfaces. Objetos de engenharia podem ser relocados como resultado de reativação e desativação, *checkpoint*, recuperação e migração.

Os objetos referência de interface provêem operações para ler e trocar seu conteúdo.

Uma referência de interface de engenharia possui um contexto em que é válido, sendo que se uma referência de interface de engenharia é transferida para fora deste contexto ela deve ser modificada por um interceptador. Se a transferência é realizada pela troca de meio, existe a necessidade de se definir um procedimento para cruzar contextos.

2.8 O modelo de engenharia como uma API para Sistemas Operacionais Distribuídos

Tentaremos agora dar uma visão do modelo de engenharia como sendo uma API que se pode colocar sobre sistemas operacionais ou sistemas operacionais distribuídos diferentes para dar uma visão uniforme às aplicações dos mesmos. Vale contudo lembrar que a função do modelo de engenharia não consiste unicamente em prover essa API, sendo a mesma apenas um produto da estruturação gerada pelo modelo de engenharia.

Em um modelo ideal para computação distribuída teríamos um sistema operacional distribuído que exportaria a mesma API para as aplicações em todas as máquinas (nós) (figura 2.8). Nesse caso não precisaríamos do modelo de engenharia ODP, mas o que acorre na realidade não é isso. Na maioria das vêzes não temos um sistema operacional distribuído, e quando temos os mesmos são incompatíveis entre sí. Surge então a necessidade de prover tal interface uniforme, o modelo de engenharia do RM-ODP provê justamente essa máquina abstrata. Tendo-se um sistema operacional distribuído ele irá comportar-se como uma camada acima do SOD [58, 60, 31] (sistema operacional distribuído) provendo uma interface uniforme a todos (figura 2.9).

O núcleo ODP irá mascarar as diferenças de recursos entre os diversos sistemas operacionais. Se, por exemplo, um sistema operacional não oferecer threads o núcleo ODP irá emular tais objetos e fornecê-los ao restante do modelo, caso contrário, apenas proverá uma interface uniforme para o modelo de recursos oferecido pelo sistema operacional. Em termos de implementação podemos considerar

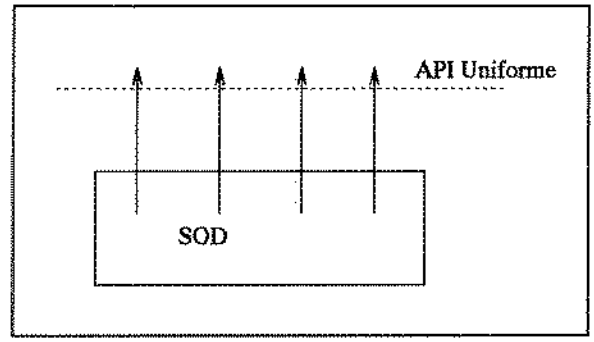

Figura 2.8: **Sistema Operacional Distribuído como API**

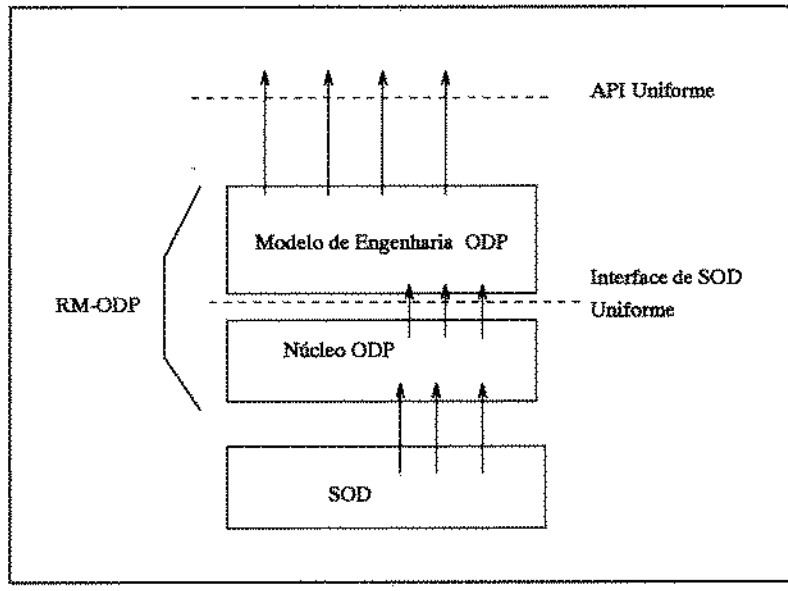

Figura 2.9: **Modelo de engenharia com API sobre um SOD**

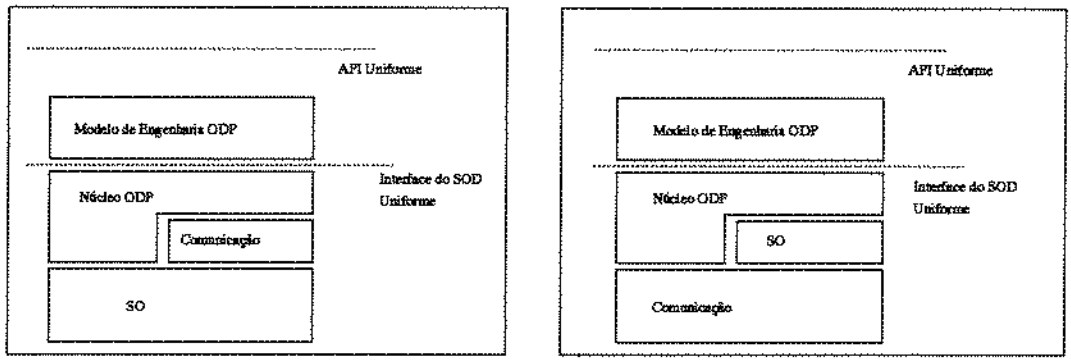

Figura 2.10: **Sistema Operacional e Camada de Comunicação**

o núcleo ODP como uma parte do modelo dependente de implementação e o restante como independente.

Na falta de um sistema operacional distribuído temos que prover mais uma camada: a comunicação que algumas vêzes se confunde com o núcleo . Podemos ver pela figura 2.10 dois enfoques para isso. O primeiro é considerar o serviço de comunicação acima do sistema operacional e o segundo abaixo, as vantagens de cada uma não serão discutidas aqui, contudo a tendência é ter o nível de comunicação abaixo do sistema operacional.

Uma outra alternativa que surge consiste em usar uma camada middleware, tal como ANSA [8, 62, 9],DCE [24, 46, 47, 48, 49, 13, 18] ou ORB [42, 28, 29, 30, 45] para prover uma série de serviços extras ao sistema operacional (figura 2.11). Neste caso, o desenvolvimento do núcleo é facilitado, contudo vale uma discussão maior se as middlewares são o enfoque ideal para base de implementação do modelo, sendo que uma discussão do fato pode ser encontrada em [54] e [66].

2.9 Mapeamento do ponto de vista computacional no ponto de vista de engenharia

Do ponto de vista computacional o sistema ODP aparece como um vasto ambiente de programação capaz de construir e executar aplicações distribuídas. Isto é equivalente à especificação de uma máquina abstrata. O modelo de engenharia provê um ambiente de execução independente de máquina para a execução de aplicações. O modelo de computação esconde dos programadores de aplicação detalhes da realização da máquina abstrata básica que o suporta [23].

O refinamento de templates computacionais em templates de engenharia cor-

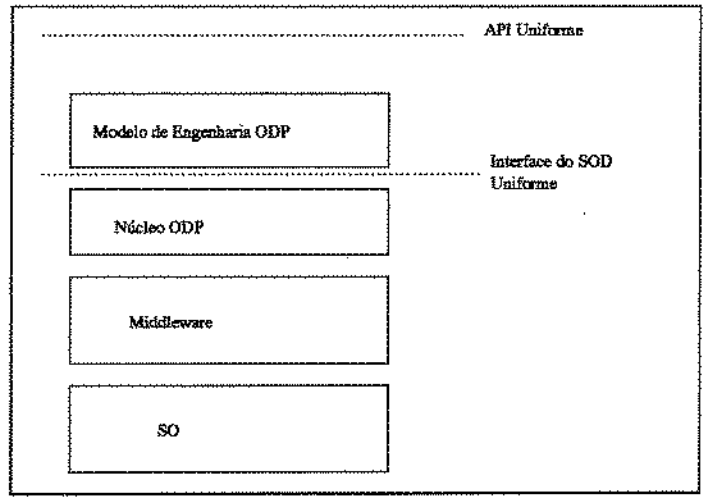

Figura 2.11: **Uso de middleware para implementação do modelo de engenharia**

responde a noção de *compilação* de programas para produzir código objeto.

O refinamento de templates de engenharia em templates de clusters corresponde a noção de *linkar* módulos para formar uma imagem executável do programa.

As seguintes correspondências foram identificadas entre objetos computacionais e objetos básicos de engenharia [1, 3]:

- Objetos básicos de engenharia correspondentes a diferentes objetos básicos computacionais podem ser membros de um mesmo cluster.
- Cada objeto *binding* computacional corresponde a um canal de engenharia.
- Cada interface de controle de um objeto *binding* computacional corresponde a uma interface de controle ou de um *stub* ou de um *binder*.
- Cada identificador de interface computacional corresponde a um ou mais identificadores de interface de engenharia.
- Cada identificador de interface computacional corresponde a um ou mais referências de interface de engenharia.
- Limites de localização nos contratos do ambiente computacional correspondem a nós no modelo de engenharia.

2.10 Interelacionamento de Funções ODP no modelo RM-ODP ISO

Como resultado da análise do sistema ODP a partir dos diversos pontos de vista do modelo surgem funções básicas relacionadas ao mesmo. Tais funções definem um conjunto de objetos que exercem papéis básicos no modelo, e são agrupadas como descrito abaixo.

- **Funções de Gerenciamento .** Funções responsáveis por prover e gerenciar o modelo de engenharia. Sendo que fornecem o gerenciamento do nó realizando a criação de cápsulas, controle de timers, criação de canais e gerenciamento de threads (*gerente de nó*); fornecimento de operações básicas de *checkpoint* e remoção sobre objetos básicos de engenharia (*gerenciamento de objeto*), operações sobre cluster (*gerenciamento de cluster*) , operações sobre cápsulas (*gerenciamento de cápsula*) e operações sobre referências de interface de engenharia (*gerenciamento de referência de interface de engenharia*).

- **Funções de Coordenação.** Têm como função coordenar atividades desempenhadas pela infraestrutura (camada) ODP. Envolve: coordenação de *checkpoint* de clusters (*Função de checkpoint* , que faz uso do gerente de cluster), recuperação de clusters falhos (*função de recuperação*, que faz uso da função de gerenciamento de cápsula), desativação e reativação de clusters (respectivamente *função de desativação*, que faz uso da função de gerenciamento de cluster, e *função de ativação* que faz uso da função de gerenciamento de cápsula), geração de históricos de eventos (*Função de notificação de evento*, que faz uso da função de gerente de nó), geração de *binding* entre grupos de objetos (função de grupo), migrar um cluster de uma cápsula para outra (*função de migração*, que faz uso das funções de gerencia de cluster e de cápsula), replicação de objetos (*função de replicação*, sendo que é usada pela função de migração) e coordenação e controle de transações (*função de transação* e *função de transação ACID*).

- **Funções de repositório.** Estão envolvidas na tarefa de garantir armazenamento e disponibilidade de dados e informações, de forma persistente e "imediata" (trader). Estas tarefas envolvem armazenar dados no sistema ODP (realizado pela *função de armazenamento*) gerenciar o interelacionamento entre objetos e interfaces, contido em um repositório (*função de repositório de relacionamento*), gerenciar um repositório de localização de

interfaces, incluindo a localização de funções de gerenciamento para clusters suportando essas interfaces (*função de relocação*), gerenciar um repositório de especificações de tipos e relacionamentos de tipos (*função de repositório de tipos*), e a *função de trading* com funções para importar e exportar serviços de objetos, sendo realizada pelo trader.

- **Funções de segurança.** Tais funções irão garantir o provimento de funções de segurança para validação, autorização e autenticação de acesso a objetos, envolvendo a prevenção de interações não autorizadas com um objeto (*função de controle de acesso*), prover monitoração e coleta de informação sobre ações relacionadas à segurança e análise dessa informação (*função de auditoria de segurança*), assegurar a autenticação de um objeto (*função de autenticação*), garantir que não haja criação, alteração e remoção não autorizada de dados (*função de integridade*), evitar a descoberta não autorizada de informação (*função de confidencialidade*), evitar a recusa por um objeto envolvido em uma interação de ter participação em toda ou parte da interação (*função de não-repudiação*), prover facilidades para geração e manutenção de chaves criptográficas (*função de gerenciamento de chave*).

- **Transparências.** Transparências ODP não são funções, mas sim comportamentos que espera-se ter ao se fazer uso da infraestrutura ODP, garantindo transparência de distribuição. Pode ser subdividida, de forma a prover transparência seletiva, em:

  Transparência de acesso. Que mascara diferenças entre representações de dados e mecanismos de interação, sendo fornecida pelos stubs.

  Transparência de falhas. Mascara para um objeto o fornecimento de tolerância a falhas.

  Transparência de localização. Mascara a localização de interfaces no espaço.

  Transparência de migração. Mascara para um objeto a habilidade da função de migração de modificar a localização de um objeto.

  Transparência de persistência. Mascara para um objeto o uso das funções de desativação e reativação.

  Transparência de relocação. Mascara para um objeto a possibilidade de mudança de localização do mesmo.

  Transparência de grupo. Mascara o uso de comportamento de grupo para suportar uma interface.

Transparência de transação. Mascara a coordenação de atividades em uma configuração de objetos para permitir consistência.

Transparência de recursos. Mascara a alocação e desalocação de recursos para componentes e fornece compartilhamento de recursos.

Transparência de replicação. Mascara a existência de mais de um objeto suportando a interface.

Transparência de federação. Mascara o cruzamento de domínios tecnológicos e/ou administrativos.

Podemos ver o interelacionamento dessas funções nas figuras 2.12, 2.13 e 2.14.

Na figura 2.14 podemos notar que as funções de coordenação e gerenciamento têm uma parte em comum (parte tracejada) o que é jusitificado pela discussão logo abaixo (devido a existência de ciclos entre as mesmas), bem como as funções de segurança estarem na vertical por serem necessárias aos vários níveis.

As funções acima são as identificadas na versão corrente do documento de padronização [3], sendo que podem não ser definitivas. Detectam-se alguns ciclos nas figuras os quais discutiremos agora.

- Ciclo C1 - A-B-C-D .
- Ciclo C2 - E-C-D.
- Ciclo C3 - H-I.
- Ciclo C4 - F-G .

Os ciclos podem ser vistos notadamente entre as funções de gerenciamento e as funções de coordenação. Tais ciclos caracterizam uma recursão, que deve ser tratada com algum ponto de parada para a mesma. Os ciclos C1 e C2 devem-se ao fato da função de *checkpoint* de cluster usar W e Z e da função de desativação de cluster usar W-C2 e migração. Se não provermos migração eliminamos os ciclos C1 e C2. C4 deve-se a que todo acesso que é parametrizado por uma referência de interface ao cluster deve fazer uso da função de gerenciamento de referência de interface e a mesma deve fazer uso do gerente de cluster para localizar as interfaces. C3 deve-se a notificação de eventos usar o núcleo e este usar notificação de evento.

Uma discussão das funções de gerenciamento serão o tema do capítulo 3, sendo que as demais funções do RM-ODP encontram-se descritas em [3].

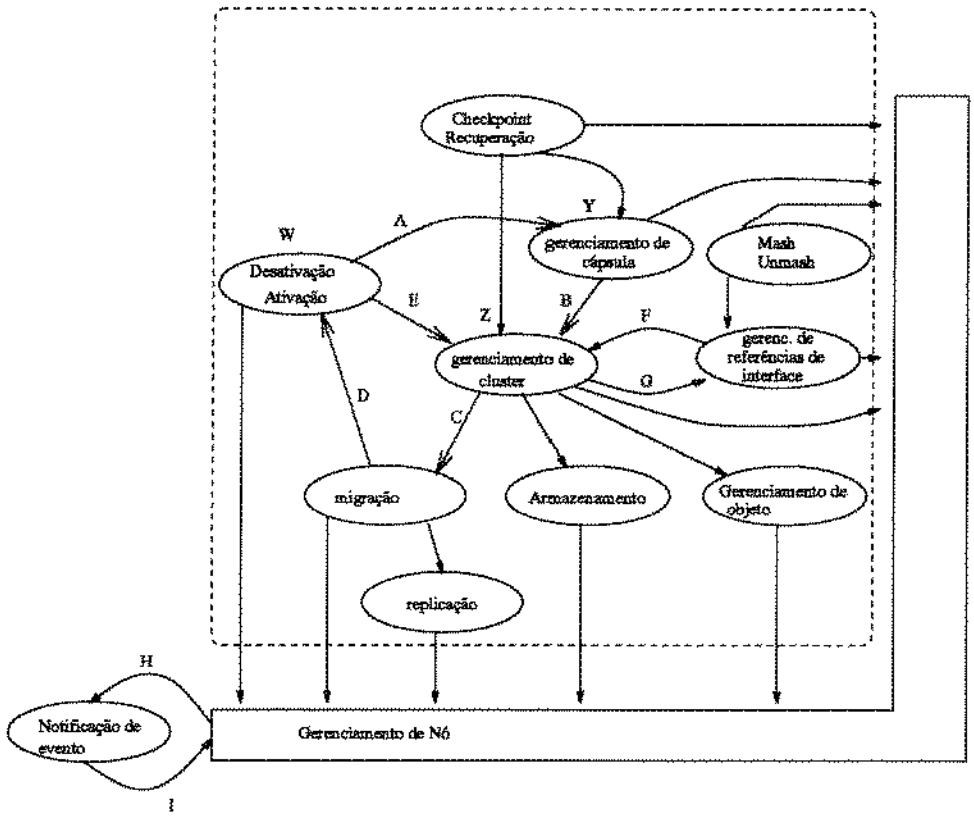

Figura 2.12: **Interelacionamento entre funções**

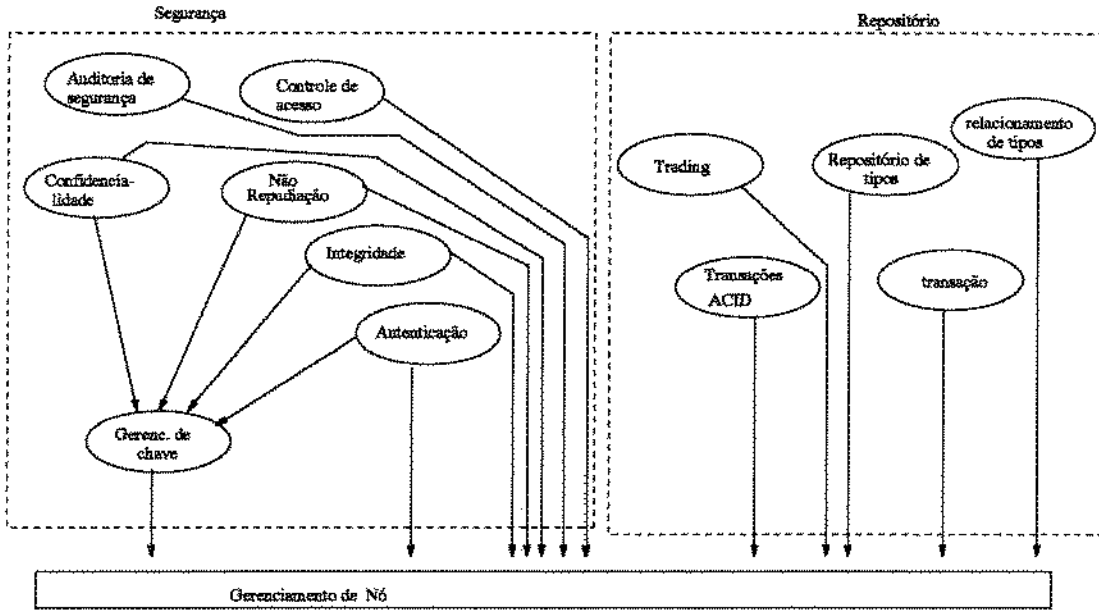

Figura 2.13: **Interelacionamento entre funções - Continuação**

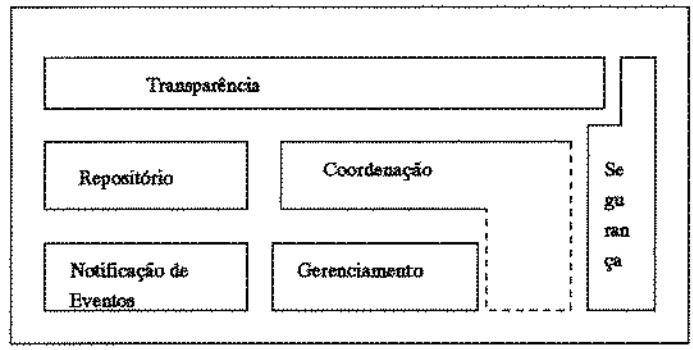

Figura 2.14: **Nível de camadas do RM-ODP**

Capítulo 3

Funções de Gerenciamento

3.1 Introdução

Nesse capítulo realizamos um estudo geral detalhando cada função de gerenciamento do modelo proposto, explicitando para cada uma suas subfunções e estruturas internas.

Nossa contribuição pode ser aqui notada na explosão das funções definindo e detalhando cada módulo das mesmas. Um estudo das estruturas de dados internas das funções e de algoritmos para as mesmas será dado em maior ênfase no capítulo 4.

As funções de gerenciamento tem por objetivo prover o gerenciamento e a estruturação da infraestrutura ODP discutida em seus objetos básicos (capítulo 2) e garantir sua operacionalidade no decorrer do tempo de uso do sistema, suportando aplicações distribuídas. É importante notar que o gerenciamento ao qual nos referenciamos consiste no gerenciamento de serviços para aplicações distribuídas (como armazenamento distribuído, processamento distribuído e comunicação em grupo) e não no gerenciamento OSI [21, 52] (*Open Systems Interconnection*).

Na atual versão dos documentos de padronização de processamento distribuído aberto [1, 2, 3] temos um conjunto de funções de gerenciamento definidas, as quais discutiremos no decorrer do capítulo. As funções são :

- **Função de Gerenciamento de Nó.** Fornecida pelo núcleo. Trata do gerenciamento de threads, acesso a relógios e gerenciamento de timers, criação de canais, localização de interfaces e instanciação de templates de cápsula.

- **Gerenciamento de objeto.** Realiza o gerenciamento de objetos básicos

de engenharia, fazendo *checkpoint* e destruição de objetos básicos de engenharia.

- **Gerenciamento de Cluster.** Que realiza desativação, *checkpoint*, migração e destruição de cluster.
- **Gerenciamento de Cápsula.** Realiza criação, reativação e recuperação de clusters e *checkpoint*, desativação e destruição da cápsulas.
- **Gerenciamento de referência de interface de engenharia .** Trata do gerenciamento de referência de interface de engenharia.

A seguir serão apresentadas as funções de gerenciamento, com o detalhamento de suas operações, apresentando ainda alguns parâmetros principais para as mesmas, uma discussão maior, a nível de implementação, será o tema do capítulo seguinte.

3.2 Função de gerenciamento de Nó

A função de gerenciamento de nó pode ser vista como uma junção de subfunções (como visto na figura 3.1), dentre elas da antiga função *Factory* [5] que cuida da instanciação de cápsulas, da função de gerenciamento de referências de interface (seção 3.6), gerenciamento de timers (seção 3.2.2), gerenciamento de threads (seção 3.2.1) e gerenciamento de canais (seção 3.2.3). Sendo que a mesma é responsável pelo controle de funções de processamento, armazenamento e comunicação no interior de um nó.

3.2.1 Gerenciamento de threads

A interface de recursos do núcleo provê um conjunto de operações para criar e gerenciar threads em um objeto básico de engenharia em um nó.

As operações de threads são providas via uma interface do núcleo, sendo fornecidas na verdade pelo gerenciador de threads do sistema operacional local (como por exemplo o pacote multithreads da Sun Microsystems [57]). Cabe ao gerenciamento de threads do modelo apenas especificar a forma pela qual será fornecida tal interface, ou seja, como serão invocadas as funções de threads do sistema operacional.

Contudo, se o ambiente local não fornecer threads, o núcleo deverá prover os mesmos via emulação, dando uma visão transparente a seus usuários. Como operações sobre threads temos as que discutiremos agora.

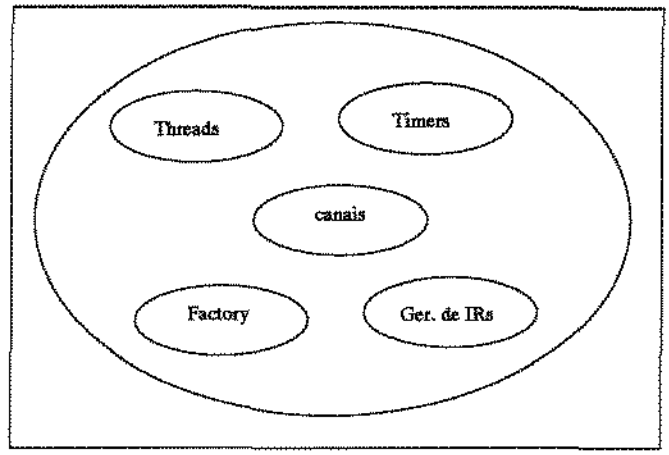

Figura 3.1: **Estrutura do núcleo**

Criar (Spawn) um thread

Operação que realiza a criação de um novo thread com código independente do thread pai, permitindo assim que o mesmo termine separadamente, possuindo a semântica:

```
Result Thread_Spawn ( );
```

onde *result* contém os possíveis resultados e erros.

Fork (Ramificar) um thread

Tal operação cria um thread executando o mesmo código do thread pai, sendo que ambos devem terminar juntos. Apresenta uma semântica semelhante à chamada *Fork* do Unix BSD [55], sendo que a semântica dessa operação será:

```
Result Thread_Fork();
```

onde os possíveis erros serão retornados em *result*, bem como a identificação do novo thread.

Join (Juntar) um thread

Tal operação trabalha de forma similar à chamada ao sistema *wait* do sistema operacional Unix [11], suspendendo o thread que a requisitou até que um thread específico termine. Sendo a semântica da mesma a abaixo:

```
Void  Thread_Join(Signal_Id);
```

onde *signal_Id* especifica por qual sinal o thread que foi suspenso deve esperar.

Delay (atrasar) um thread

Tal operação irá retardar a chamada de um thread por um período de tempo especificado. Tendo a seguinte semântica:

```
Result Thread_Delay ( Time_Interval );
```

em que *Time_Interval* consiste no intervalo de tempo especificado para o retardo do thread e *result* os possíveis resultados.

Sincronizar threads

Essa operação permite a sincronização entre threads para a realização de uma tarefa específica, sendo a semântica a seguinte:

```
Result Thread_Sincronizate ( List_Threads );
```

em que *List_Threads* consiste na lista de threads a serem sincronizados e *result* nos possíveis erros.

3.2.2 Acesso a relógios e gerenciamento de timers

A função de gerenciamento de timers permitirá o gerenciamento de temporizações e o acesso a relógios.

3.2.3 Criação de canais

A função de criação de canais do núcleo irá permitir a criação de canais entre objetos de engenharia , para tanto proverá operações para :

- Criar um canal, tendo-se um template de canal como parâmetro.

  ```
  Result Channel_Create  ( Template  );
  ```

  Onde *Template* descreve o canal a ser criado.

- Estabelecer um *binding* entre um objeto de engenharia em uma cápsula e um conjunto de objetos de engenharia , identificados por referências de interface de engenharia.

```
Result Bind_Objects ( IR_EO1 , IR_EO2,  ..., IR_EOn);
```

sendo que aqui *IR_EO1* consiste na referência de interface do objeto origem do canal e *IR_EO2, ...IR_EOn*, nas referências de interface dos objetos de engenharia da outra extremidade do canal. Podemos ver tal estrutura de maneira similar à da figura 2.6, ou tendo-se apenas um objeto com destino algo como a figura 2.5. *Result* consiste no identificador do canal ligado.

- Produzir informação sobre o tipo de canal que está suportando o *binding* e a localização no tempo e espaço das interfaces de comunicação através das quais modificações no *binding* podem ser realizadas.

```
Result Get_Channel_Info ( Canal);
```

Sendo que nas descrições acima *Result* também consiste nos possíveis erros das funções.

Vejamos agora como se daria o processo de criação de um canal. A estrutura básica encontra-se descrita no modelo de referência [3]. O contexto consiste em estabelecer um canal entre um objeto E1 e E2. Em primeiro lugar um objeto C1, que controlará o *binding* do lado de E1, inicializa a configuração do canal interagindo com o núcleo, sendo que essa interação é parametrizada pelo tipo do canal, um papel e uma interface de um objeto (E1 no caso) para ser ligado.

- O núcleo liga uma cadeia de objetos, que compreende um stub S1, um binder B1 e um objeto protocolo a serem ligados à interface de E1.
- Os tipos dos stubs, binders e protocolos que são selecionados são determinados pelos parâmetros tipo do canal e papel.
- Os resultados da interação são uma referência de interface IR para comunicação com outros objetos e ligações de interfaces de controle dos binders B1 e stub S1 ao objeto C1.

A seguir a referência de interface é comunicada a outro objeto de engenharia C2, que controlará o outro lado do canal, possivelmente em outro cluster, em outra cápsula, em outro nó. Sendo que o objeto C2 interage com o seu núcleo para ligar o canal, a interação é parametrizada por um tipo de canal, um papel, e uma interface de um objeto E2 para o qual o *binding* é requerido. O núcleo determinará o tipo de canal e localização no tempo e no espaço dos objetos protocolo para os outros participantes do canal da referência de interface de engenharia IR.

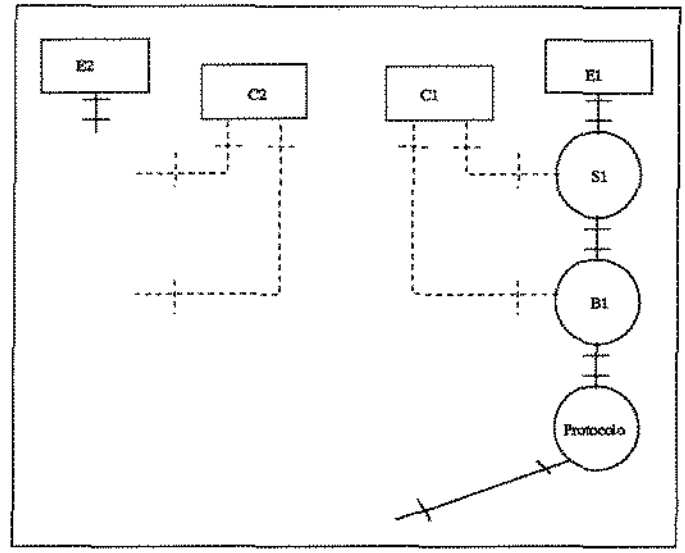

Figura 3.2: **Criação de Canal** - *Parte 1*

- O núcleo conectará uma cadeia de objetos, compreendendo um stub S2, um binder B2, um objeto protocolo e se necessário um interceptador ao objeto E2.
- Os tipos dos stubs, binders, protocolos e interceptador que são selecionados são determinados pelos parâmetros de papel do canal, um supertipo comum de tipo de canal, e os tipos de canais dos outros participantes.
- O binder B2 interage com os binders no canal para habilitar comunicação através do canal, incluindo fornecimento de outros objetos protocolo e interceptador.
- O núcleo liga o objeto C2 as interfaces de controle do binder B2 e stub S2, e a interface do objeto E2 a interface de apresentação do stub S2.

Tal estrutura pode ser vista nas figuras 3.2, 3.3 e 3.4. Sendo que no processo de estabelecimento de um canal multiponto os últimos passos serão repetidos diversas vezes, uma para cada objeto básico de engenharia.

3.2.4 Instanciação de template de cápsula

Em versões anteriores do modelo de referência ([5]) temos menções ao objeto *factory* que era responsável pela instanciação de cápsula, mas na versão corrente

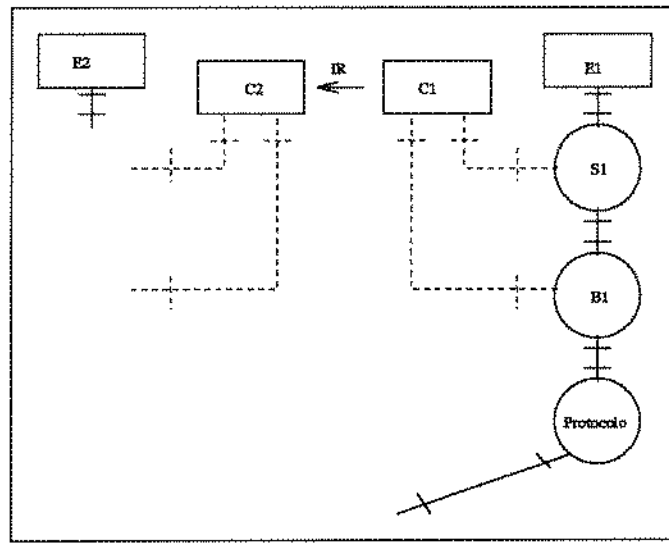

Figura 3.3: **Criação de Canal** - *Parte 2*

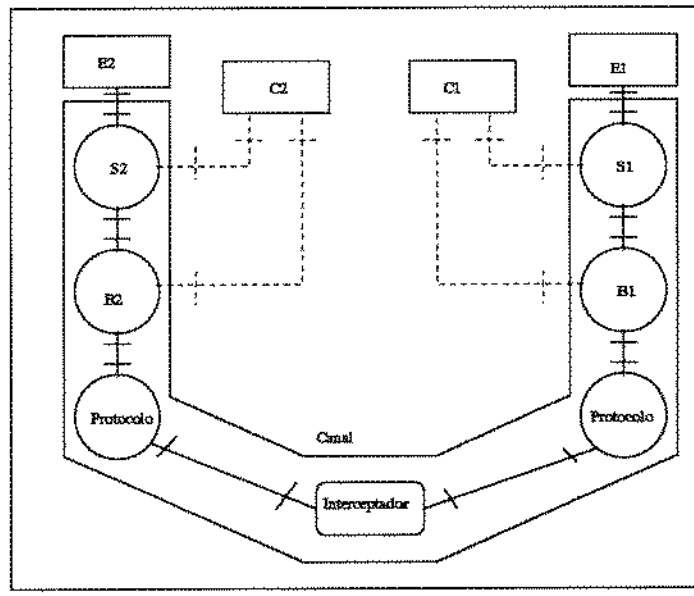

Figura 3.4: **Criação de Canal** - *Parte 3*

temos a localização de tal componente como parte do núcleo, sendo uma função do gerenciamento de nó.

A operação de instanciação de cápsula irá realizar a criação de uma nova cápsula no nó e de toda a estrutura necessária à mesma. Podemos notar que ao se requisitar a instanciação de uma cápsula a mesma recebe como parâmetros um template de cápsula que descreve a atividade inicial a ocorrer na cápsula, bem como, uma referência de interface para um *objeto de monitoração* contendo operações de notificação, que serão chamadas se a cápsula terminar (por chamada explícita ao gerente de cápsula ou por falha). Podemos notar a seguinte semântica para a operação:

```
Result Instantiate_Capsule ( Template,  Ir_Monit );
```

em que *Template* consiste no template de cápsula, que descreve a atividade inicial a ocorrer na cápsula e *Ir_Monit* consiste na referência de interface para um objeto de monitoração.

A função de criação de cápsula irá atuar da seguinte forma:

- Alocar via função *Get_Int_resources* uma interface de gerenciamento de nó, contendo funções de processamento, armazenamento e comunicação. Tal interface proverá os recursos do núcleo para a cápsula, tendo a semântica descrita abaixo, onde *Interface* consiste na interface de recursos retornada.

  ```
  Result Get_Int_Resources ( Interface ) ;
  ```

- Criar um gerente de cápsula para a nova cápsula, via requisição ao núcleo pela função *Create_CPM*, que apresenta a seguinte semântica:

  ```
  Result Create_CPM ( Interface_CPM) ;
  ```

  onde *Interface_CPM* consiste na interface do gerente de cápsula retornado.

- Criar uma interface de controle para a nova cápsula no núcleo.

  ```
  Result Create_Control_Int (Interface_CPM, Int_control);
  ```

  que recebe a interface da cápsula *Interface_CPM* como parâmetro e retorna possíveis erros, como falta de recursos, em *result*, bem como a interface de controle do núcleo em *Int_Control*.

Como resultado da instanciação da nova cápsula será retornada a interface de gerenciamento de cápsula, bem como a interface de controle criada, provendo operações para consulta do estado da cápsula (se existe atividade ou não na mesma) e para forçar sua terminação.

Templates de cápsula são parametrizados por uma função de notificação de evento para relatar eventos significativos na nova cápsula.

3.3 Função de Gerenciamento de Objeto

A função de gerenciamento de objeto gerencia objetos básicos de engenharia, habilitando que um objeto seja *checkpointed* ou destruído. A mesma é fornecida pelos objetos básicos de engenharia através de uma interface, que provê gerenciamento do objeto engajado nas atividades de *checkpoint* e destruição, sendo contudo que tais operações são necessárias principalmente quando o cluster a que pertencem pode ser desativado, *checkpointed* ou migrado.

Diferentes interfaces de gerenciamento de objetos podem ter diferentes tipos de interfaces: as operações suportadas por um objeto básico de engenharia dependem de suas necessidades de gerenciamento . Um objeto pode, por exemplo, não prover a operação de *checkpoint*.

3.3.1 Capturar o estado de um Objeto em um checkpoint

A operação para capturar o estado de um objeto em um *checkpoint* irá receber a identificação do objeto, *Obj_Id* (um identificador local ao cluster), com a seguinte semântica:

```
Result Obj_checkpoint  ( Obj_Id );
```

sendo que realizará o *checkpoint* do objeto pelo uso da função de armazenamento na interface de armazenamento especificada para tal fim na interface de gerenciamento do cluster (seção 3.4.1) a que pertence o objeto, sendo que para obter o *checkpoint* é feita uma chamada à interface do objeto básico de engenharia para *checkpoint*. *Result* conterá o possíveis erros, tais como tentar-se um *checkpoint* de um objeto que não suporte tal operação.

3.3.2 Destruir um objeto

A função de deleção de objeto promove a destruição de um objeto, tendo a seguinte semântica:

```
Result Obj_Delete( Obj_Id);
```

onde *Obj_Id* consiste na identificação do objeto.

A mesma faz uso das funções de threads do núcleo (seção 3.2.1) para terminar a atividade (threads) no objeto e depois é chamado a interface do objeto para *delete*, e finalmente removendo o mesmo da lista de objetos do cluster.

3.4 Função de Gerenciamento de Cluster

A função de gerenciamento de cluster irá prover o gerenciamento dos clusters em uma cápsula. Ela permite que clusters sejam *checkpointed*, recuperados, migrados, desativados e destruídos, sendo que a mesma é fornecida por um objeto gerente de cluster, que está engajado nas atividades de desativação de cluster, *checkpoint* de cluster, migração de cluster, replicação de cluster e destruição de cluster.

A realização de *Checkpoint* e desativação de cluster somente estão disponíveis se todos os objetos no cluster tem uma interface de gerenciamento de objeto suportando as funções de *checkpoint* de objeto (seção 3.3.1). Desativação e destruição de cluster requerem que todos os objetos básicos de engenharia no cluster tenham uma interface de gerenciamento de objeto incluindo a função de deleção de objeto (seção 3.3.2).

Um gerente de cluster tem uma interface contendo operações que habilitam: modificar a política do cluster (exemplos: identificar a interface de armazenamento na qual *checkpoints* e a versão desativada do cluster serão colocados, e identificar a interface do relocador para as interfaces do cluster), criar uma cópia desativada do cluster (seção 3.4.2) , criar um *checkpoint* para o cluster (seção 3.4.3), trocar o cluster com uma nova instância de cluster de um *checkpoint* de cluster, migrar o cluster para outra cápsula (seção 3.4.4), e deletar o cluster (seção 3.4.5). Sendo que diferentes interfaces de gerenciamento de cluster podem ter diferentes tipos de interfaces. As operações suportadas por um cluster específico dependem de suas necessidades de gerenciamento .

Descreveremos agora as operações presentes na interface de gerenciamento do gerente de cluster.

3.4.1 Especificar a interface de armazenamento para checkpoints do cluster

A operação para especificação da interface de armazenamento para *checkpoints* dos clusters recebe a interface do cluster e associa à mesma uma interface de

armazenamento, tendo a seguinte semântica:

```
result Define_Checkpoint_IR ( Cl_Id, Interface);
```

que na verdade realiza uma busca na tabela de interfaces de armazenamento associada à interface de recursos da cápsula a que o cluster pertence à procura de uma interface de armazenamento livre que possa ser associada ao cluster para *checkpoint*. Deve-se considerar os casos de erros, por exemplo, quando a interface solicitada não existir (o que é retornado em *result*).

3.4.2 Criar uma cópia desativada do cluster

Irá produzir um *checkpoint* do cluster, que pode ser reinstalado usando a função de reativação de cluster do gerente de cápsula (ver seção 3.5.1). A função recebe a interface de engenharia do cluster, tendo a seguinte semântica:

```
Result Desative_Cluster ( Cluster_ID );
```

tal função fará chamadas sucessivas à função de *checkpoint* de objeto e de remoção de objeto do gerenciamento de objeto , sendo que a tarefa de armazenar a cópia desativada será realizada pela função de armazenamento usando a interface de armazenamento associada ao cluster, conforme a seção 3.4.1.

Para o sucesso de tal operação é necessário que todos os objetos básicos de engenharia no cluster forneçam funções de *checkpoint* e terminação em suas interfaces de gerenciamento, caso contrário, ocasionará um erro, havendo ainda a necessidade de se ter associada ao cluster uma interface de armazenamento.

3.4.3 Criação de um checkpoint para o cluster

A operação de criação de um *checkpoint* de um cluster recebe como parâmetro a interface de engenharia do cluster, tendo a seguinte semântica:

```
Result Checkpoint_cluster (Cluster_ID);
```

A função atuará de forma análoga a de desativação (seção 3.4.2), fazendo uso da função de *checkpoint* do gerenciamento de objetos, *Obj_Checkpoint*, cabendo à função de armazenamento a tarefa de armazenar o *checkpoint* com base na informação fornecida da interface para o armazenamento descrita na seção 3.4.1.

Realizar o *checkpoint* de um cluster pode requerer realizar o *checkpoint* de outros clusters para obter uma cópia consistente de todas as atividades presente nestes clusters, como regras básicas temos:

- O estado do cluster inicial deve estar em estado consistente antes de ser gerado o *checkpoint*.
- deve haver consistência entre todos os *checkpoints* de vários clusters inicialmente *checkpointed* em conjunto (Exemplo: Todos os *checkpoints* refletem o mesmo conjunto de interações que ocorreram entre os clusters).
- Que, tendo decidido gerar o *checkpoint* de um cluster, o processo é recursivamente aplicado para determinar o conjunto completo de clusters a serem *checkpointed*.

Se uma cápsula falha então todos os seus clusters que necessitavam de transparência de falha devem ser reativados usando o *checkpoint* mais recente para aquele cluster, via função específica do gerente de cápsula.

Quando gerando o *checkpoint*, o estado do cluster é armazenado pelo gerente de cluster usando a função de armazenamento. Sendo que as interfaces dos relocadores do cluster são informadas da política de recuperação para o *checkpoint* do cluster.

A política de recuperação especifica como escolher uma cápsula na qual será feita a recuperação do *checkpoint*. Recuperação é feita ou por requisição explícita para restaurar o estado no instante do último *checkpoint* (uma operação do gerente de cluster), ou automaticamente seguindo o processo de relocação depois da falha. Sendo que a recuperação é permitida pela reativação do *checkpoint*.

Como na desativação, o sucesso dessa operação depende de que todos os objetos básicos de engenharia no cluster forneçam em suas interfaces de gerenciamento a função de *checkpoint* de objeto, *Obj_Checkpoint* (seção 3.3.1).

3.4.4 Migração do cluster para outra cápsula

A migração de um cluster é coordenada pela função de migração. A operação de migração de cluster apresenta a seguinte semântica:

```
Result Migrate_Cluster ( Cluster_id , Cp_Ir_Dest );
```

onde *Cluster_ID* consiste na interface de engenharia do cluster que desejamos migrar e *Cp_Ir_Dest* a referência de interface de engenharia da cápsula destino para onde deseja-se migrar o cluster.

Podemos ver a implementação dessa função como uma chamada à função de desativação de cluster para o cluster *cluster_id* seguida da chamada da função de reativação do gerente de cápsula *Cp_ir_Dest* para reativar o *checkpoint* na cápsula destino.

3.4.5 Destruir o cluster e liberar seus recursos

A função de destruição de cluster irá tomar a interface de engenharia do cluster como parâmetro, tendo a seguinte semântica:

```
Result Delete_Cluster( Cluster_ID );
```

que realizará sucessivas chamadas à função de destruição de objeto do gerenciamento de objeto, *Obj_Delete*, realizando depois a remoção do cluster da cápsula via função do gerente de cápsula.

Um fato que deve ser considerado é o caso em que o cluster que desejamos destruir esteja desativado, nesse caso é realizada a reativação do mesmo seguida de sua eliminação.

3.5 Função de Gerenciamento de Cápsula

A função de gerenciamento de cápsula gerencia cápsulas, sendo fornecida pelos objetos gerentes de cápsula. Um gerente de cápsula está engajado em atividades de instanciação de clusters (instanciação, reativação e restauração), desativação de cápsula, *checkpointing* de cápsula e destruição de cápsula.

Cada gerente de cápsula tem uma interface provendo operações que habilitam: instanciar um template de cluster na cápsula, reativar um cluster desativado na cápsula (seção 3.5.1), desativar (seção 3.5.2), *checkpoint* (seção 3.5.3) e destruir (seção 3.5.4) uma cápsula. Sendo que diferentes gerentes de cápsula podem ter diferentes tipos de interface operacionais do gerente de cápsula. As operações suportadas por uma cápsula dependem de suas necessidades de gerenciamento.

Discutiremos a seguir as operações da interface de gerenciamento do gerente de cápsula.

3.5.1 Instanciação de cluster na cápsula

A função de instanciação de template de cluster atuará nas atividades de instanciação de um novo cluster, reativação de um cluster a partir de um *checkpoint* e na restauração de clusters, a partir de *checkpoints*, depois de uma falha. Apresentaremos a mesma como três subfunções começando pela criação de um novo cluster.

Instanciação de um novo cluster

Para realizar a instanciação do template de cluster na cápsula essa função recebe como parâmetro um template de cluster, tendo a seguinte semântica:

```
result Instantiate_Cluster ( Cluster_Template );
```

em que *result* consiste no resultado da função, discutido a seguir. Sendo que a função atuará da seguinte forma:

- Realiza a cópia do template de cluster na cápsula que dar-se-á pelo uso da operação de cópia de template na cápsula, da seguinte forma:

  ```
  result Copy_Template ( Cluster_Template , CP_Id );
  ```

  que copia o template *Cluster_Template* na cápsula *CP_Id* retornando o resultado da operação em *result* (se tudo bem ou se aconteceu algum erro). Tal operação irá acrescentar uma entrada na tabela de templates de clusters da cápsula.

- Realiza a criação de um novo gerente de cluster para o cluster a ser criado, via função *Create_CLM*, que retorna a interface do novo CLM, *Interf_CLM*, como visto abaixo.

  ```
  Interf_CLM  Create_CLM();
  ```

- Realiza a conversão do template de cluster à forma ativa, associando o mesmo ao gerente de cluster alocado no passo anterior. Tal ação é realizada pela função abaixo que recebe como parâmetro a interface do gerente de cluster e o template de cluster.

  ```
  result Ative_Template ( Cluster_Template , Interf_CLM );
  ```

- Como último passo na operação de instanciação de clusters devemos retornar os identificadores de interfaces resultantes da ativação do template, dentre eles a interface de gerenciamento de cluster, que estarão presentes em *result*.

Reativação de um cluster desativado na cápsula

A reativação de cluster desativado na cápsula consiste em uma tarefa que, em parte, se assemelha com o processo de instanciação de template de cluster, sendo que a operação recebe no entanto um *checkpoint* de cluster como parâmetro no lugar do template de cluster. A função apresenta a semântica abaixo:

```
result Reativate_Cluster ( Checkpoint_cl);
```

Sendo que a mesma atua da seguinte forma:

- Realiza a cópia do *checkpoint* na cápsula, o que pode ser feito pela função *Copy_Checkpoint* que tem como parâmetro a identificação do *checkpoint* de cluster, e a identificação da cápsula:

```
Result Copy_Checkpoint ( Checkpoint_cl, CP_Id);
```

- Realiza a criação de um novo gerente de cluster para o cluster a ser criado, via função *Create_CLM*, que retorna a interface do novo CLM, *Interf_CLM*, como visto abaixo.

```
Interf_CLM  Create_CLM();
```

- Sendo que a seguir o *checkpoint* é convertido à forma ativa pela função *Ativate_checkpoint*, que recebe as interfaces do CLM alocado e a identificação do *checkpoint* a ativar, conforme a semântica abaixo:

```
Result Ativate_Checkpoint (Checkpoint_cl, Interf_CLM);
```

  Uma das tarefas de *Ativate_checkpoint* consiste em realizar as transformações nas referências, como descrito logo abaixo.

- Como passo final são retornadas as novas interfaces e o cluster reassume sua atividade de onde havia parado quando foi desativado.

A reativação tem três transformações para cuidar, entre elas:

- Referências no interior do cluster desativado.
- Referências externas ao cluster desativado.
- Referências sobre o cluster desativado.

Clusters desativados algumas vêzes têm que ser transformados quando o cluster é ativado em uma cápsula com uma tecnologia diferente daquela em que o cluster foi criado.

Clusters que necessitam de controle de acesso devem somente ser migrados para cápsulas ou funções de armazenamento as quais estão sujeitas a alguma política de segurança.

Referências no interior de um cluster devem ser transformadas quando o cluster é ativado em uma localização diferente da que as referências foram criadas (ou por último reparadas).

Referências fora do cluster devem ser transformadas quando o cluster é ativado em uma localização diferente da que as referências foram criadas (ou por último reparadas).

Referências sobre o cluster devem ser bloqueadas enquanto o cluster está desativado e transformadas quando o mesmo é reativado com ajuda da função de relocação.

Recuperar o estado do cluster a partir do checkpoint

A operação de restauração do estado de um cluster a partir de um *checkpoint* irá tomar um *checkpoint* de cluster e reativá-lo na cápsula, desempenhando uma tarefa semelhante à reativação de um cluster que foi descrito na função anterior de reativação, sendo contudo que aqui necessitamos refazer todo o contexto do cluster, visto que a recuperação deve-se a uma queda brusca no sistema ODP e devemos portanto garantir a integridade do sistema ao restaurar o cluster, garantindo que por exemplo não haja uma segunda cópia do cluster ativa no sistema. A mesma apresenta a semântica seguinte:

```
Result Restaure_Cluster ( Checkpoint_cl);
```

em que *Checkpoint_cl* consiste no identificador do *checkpoint* de cluster e *Result* nos possíveis resultados, incluindo a interface do gerente de cluster a ser alocado e erros.

3.5.2 Desativação de uma cápsula

A desativação de uma cápsula é realizada via a desativação de todos os clusters ativos na mesma. Podemos ver essa função como uma chamada sucessiva a função de desativação de cluster de cada gerente de cluster na cápsula. Algo da seguinte forma :

```
Result Desative_capsule( Capsule_ID)
{
 Para cada Cluster na Capsula
   Desative_Cluster ( Cluster_Id)
 Elimine a capsula
}
```

que percorre toda a tabela de clusters, chamando para cada cluster a função de desativação do gerente de cluster com a interface de engenharia do cluster, *Cluster_ID*, como parâmetro. Se houver algum erro (por exemplo se algum cluster não suportar desativação) este deve ser tratado.

Uma discussão que nos cabe levantar é sobre a manutenção da cápsula, ou seja, se desativando todos os clusters na cápsula devemos mante-la sem atividade ou devemos destruí-la. Acreditamos na última hipótese visto que se não o fizermos o próprio núcleo fará tal tarefa como descrito no modelo prescritivo [3]. Logo como ação complementar podemos ver a chamada a função de destruição de cápsula do gerente de cápsula.

3.5.3 Realizar um checkpointing da cápsula

A realização de um *checkpoint* de todos os clusters da cápsula pode ser vista de maneira similar à da operação de desativação de cápsula (seção 3.5.2) cómo uma chamada recursiva a uma função do gerente de cluster, nesse caso a função de *checkpoint* de cluster, possuindo a seguinte semântica:

```
Result  Checkpoint_Capsule ( Capsule_ID);
```

Onde *Capsule_ID* consiste no identificador de engenharia da cápsula.

3.5.4 Destruição da cápsula

A função de destruição de cápsula irá realizar a eliminação da cápsula, pela eliminação de todos os clusters no interior da mesma, e após isso, com a eliminação da infraestrutura da cápsula, o que deve ser comunicado ao núcleo para reaproveitamento de recursos. Tal função dar-se-á pela chamada recursiva a todos os clusters na cápsula da função de remoção de cluster de cada gerente de cluster. Podemos ver tal operação com a seguinte semântica:

```
Result Delete_Capsule ( Capsule_ID);
```

que recebe o identificador de engenharia da cápsula a destruir como parâmetro.

3.6 Gerenciamento de Referência de interface de engenharia

A interface da função de gerenciamento de referência de interface mantém, no escopo da mesma :

- Informação sobre a posse de uma dada categoria de referência de interfaces, detectando quando nenhuma cópia de uma referência de interface está mantida fora do cluster que suporta a referência de interface. Um gerente de cluster é notificado se todas as interfaces pertencentes a uma dada categoria neste cluster não são referenciadas, de forma a detectar inatividade.

- Informação sobre a existência de interfaces da categoria dada, notificando a cápsula que são proprietárias de referências de interfaces de uma interface que está falha ou parada.

A notificação fornecida por esta função permite que recursos que não são requeridos há muito possam ser liberados.

Um **Domínio de gerenciamento de referência de interface** consiste em um conjunto de nós sob o controle de uma única função de gerenciamento de referência de interface.

Uma **Política de gerenciamento de referência de interface** consiste em um conjunto de permissões e proibições que governam a política de gerenciamento de domínios de referência de interface.

A função de gerenciamento de referência de interface é chamada pelos gerentes de cluster:

- Quando uma interface é criada.

- Quando uma referência de interface é passada entre clusters.

- Quando uma interface falha ou para.

A função de gerenciamento de referência de interface chama clusters quando todas as suas interfaces estão não referenciadas.

A função de gerenciamento de referência de interface chama cápsulas quando qualquer interface para qual a proprietária da referência de interface está falha ou parada.

A função de gerenciamento de referências de interfaces pode ser vista como uma função que fornece um servidor de nomes distribuído em um sistema computacional [22], se analisarmos referências de interface de engenharia sob a ótica de serem identificadores únicos em um contexto de nomes consistentes.

Se assim vista a função de gerenciamento de referências de interfaces irá prover um conjunto de funções/operações a serem realizadas sobre referências de interface, as quais apresentaremos agora.

- Provê uma operação que dado um identificador de engenharia de uma interface realize a criação de uma referência de interface para o mesmo, sendo que tal função é mostrada a seguir:

```
IR Registre_IR ( Ident_eng);
```

onde *IR* consiste na referência de interface de engenharia gerada e *ident_eng* no identificador de engenharia da interface.

- Uma operação que dada uma referência de interface *IR_I* realize a validação da mesma, e se esta foi modificada retorne a nova referência de interface *IR_N*, ou se a mesma não existe mais é retornado um erro.

```
IR_N Validate_IR ( IR_I ) ;
```

- Eliminar uma dada referência de interface *IR* de um domínio de referências de interface.

```
Result Delete_IR ( IR );
```

em que *Result* contém os possíveis erros, por exemplo se a referência de interface *IR* não existir.

- Registrar um nó *No_ID* em um domínio de referência de interface *IR_D*.

```
Result Registre_no_IRD ( IR_D , No_ID );
```

- Checar a consistência e existência de um nó *No_ID* de um domínio de referências de interface *IR_D*.

```
Result Validate_no_IRD(IR_D, No_ID);
```

- Eliminar um nó *No_ID* de um domínio de referências de interface *IR_D*.

```
Result Delete_no_IRD ( IR_D, No_ID);
```

Tais funções, bem como a estrutura de referências de interface, serão discutidos em maiores detalhes no capítulo 4.

Capítulo 4

Modelagem para implementação

4.1 Introdução

Descrevemos nesse capítulo a implementação de um protótipo que provê as funções básicas de gerenciamento, descritas no capítulo 3, desenvolvido sobre o ambiente Unix SunOS (algumas das características do SunOS para nossa utilização específica foram descritas no capítulo 1). Contudo a migração do protótipo desenvolvido para outra plataforma Unix não seria uma tarefa complexa, tendo em vista que buscou-se fazer uso, na medida do possível, de características padrões do sistema Unix evitando características específicas de implementação.

Vale ressaltar que o protótipo desenvolvido apresenta uma série de limitações se comparado ao modelo proposto em [3], mas cumpre seu objetivo, que consiste em validar algumas das idéias sobre gerenciamento que foram levantadas durante o decorrer deste texto. O mesmo foi implementado inicialmente como um simulador do modelo, em que testou-se as idéias sobre estruturação das funções e estruturas de dados, sendo que posteriormente foi expandido de forma a prover uma camada sobre o sistema operacional Unix provendo a estruturação do modelo de engenharia em forma de nó, cápsulas, clusters e objetos básicos de engenharia. Um fato que convém lembrar consiste em que o modelo de referência ODP, como o nome diz, consiste em um modelo de referência e não em um modelo para implementação o que dificulta a implementação de alguns conceitos abstratos levantados.

Discutiremos inicialmente as soluções adotadas para mapear as diversas abstrações descritas no modelo de referência que foram discutidas no capítulo 2 e a

forma como foi estruturado o protótipo. Posteriormente discutiremos para cada função de gerenciamento, que se considerou no protótipo, a forma pela qual a mesma foi modelada. Como observação final vale lembrar que as descrições existentes nesse capítulo estão apresentadas na linguagem de programação C [32], assumindo então o conhecimento da mesma.

Um fato ainda não mencionado foi sobre a escolha da plataforma utilizada. O sistema Unix foi escolhido por ser a única plataforma disponível no momento da realização do protótipo, sendo que um protótipo que trabalhasse em ambiente heterogêneos teria que ser implementado sem as restrições que realizamos. A plataforma *Multiware* descrita no capítulo 1 seria a escolha natural, contudo a mesma encontra-se em fase de término de especificação e início de implementação.

4.2 Estruturação do Protótipo

Uma das principais decisões tomadas na elaboração do protótipo consistiu em como implementar o mesmo, ou seja, como modelar os conceitos abstratos do RM-ODP sobre a plataforma Unix adotada, e como dividir o protótipo em módulos componentes. Veremos agora as soluções imaginadas, que consistem, na primeira em termos todo o modelo de engenharia implementado como um único módulo que realizaria todas as tarefas, e o segundo enfoque em que divide-se o modelo de engenharia em um conjunto de módulos que provêem as diversas funções de gerenciamento interagindo entre si, sendo alocados sob demanda. Essa segunda solução foi adotada na implementação, sendo que a primeira alternativa foi utilizada no simulador.

No primeiro enfoque teríamos um único módulo em cada máquina na rede que cuidaria das tarefas de gerenciamento do modelo, como pode ser visto na figura 4.1. Sendo que ao inicializarmos cada máquina tal módulo seria carregado e ficaria associado a uma porta específica funcionado como um *daemon* Unix. A comunicação com o servidor dar-se-ia via mecanismo de sockets ou RPC, sendo que a interação entre os módulos do modelo dar-se-ia pelo uso de estruturas de dados comuns entre os mesmos. Tal solução foge um pouco do modelo de referência ODP (RM-ODP) visto que existe a figura de apenas um gerente de cápsula e um gerente de cluster por nó, estando implementado internamente nos mesmos a gerencia dos diversos clusters/cápsulas na forma de listas geridas internamente pelos módulos. O ganho que aqui pode ser sentido é a integração dos diversos módulos podendo os mesmos compartilhar informação de uma forma mais rápida do que se forem implementados separadamente. Contudo perde-se em modularidade, ou seja, o modelo fica muito fechado dificultando a troca de

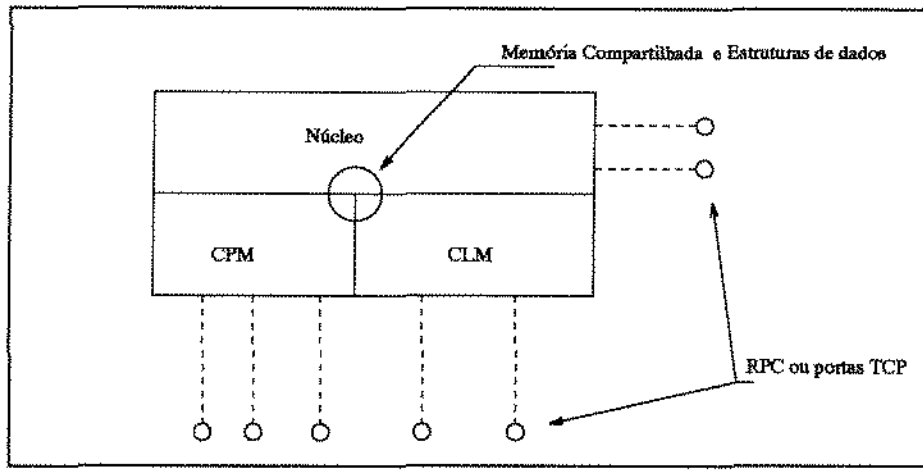

Figura 4.1: **Modelo centralizado do protótipo**

um módulo e a alteração de mecanismos como comunicação entre objetos e afins.

A segunda solução apresentada, e adotada no protótipo, tenta prover algo mais próximo do modelo de engenharia do RM-ODP no sentido de termos a estruturação do mesmo como um conjunto de servidores, tal proposta pode ser vista na figura 4.2. Veremos agora a descrição da mesma.

Temos aqui um núcleo por máquina como um servidor que irá prover as funções de gerenciamento de nó, e associados ao núcleo um conjunto de servidores CPM (gerentes de cápsulas) e a cada um destes um conjunto de servidores CLM (gerentes de clusters). Inicialmente temos apenas o núcleo no nó sendo que quando da instanciação de uma cápsula é alocado dinamicamente um novo servidor CPM e associado ao núcleo, da mesma forma quando da criação de um cluster cria-se um novo servidor CLM dinamicamente e associa-se o mesmo a um gerente de cápsula, mantendo assim uma hierarquia em que cada núcleo conhece seus CPMs e cada CPM conhece seus CLMs, sendo responsável pela gerencia e interação com os mesmos. Temos aqui que cada CPM é filho (no sentido do Unix) do Núcleo e cada CLM é filho de um CPM. Objetos básicos de engenharia são criados pelos gerentes de cluster sendo filhos dos mesmos. A forma como se dá a comunicação entre os diversos servidores e destes com os objetos básicos de engenharia foi imaginada inicialmente como sendo via sockets ou sinais Unix sendo que no final foi adotado um mecanismo baseado em RPC entre os servidores e baseado em RPC e sinal entre os objetos básicos de engenharia e os servidores. Tal solução justifica-se pela simplicidade do modelo de RPC e a facilidade que pode trazer no desenvolvimento do protótipo proposto, sendo contudo que outra

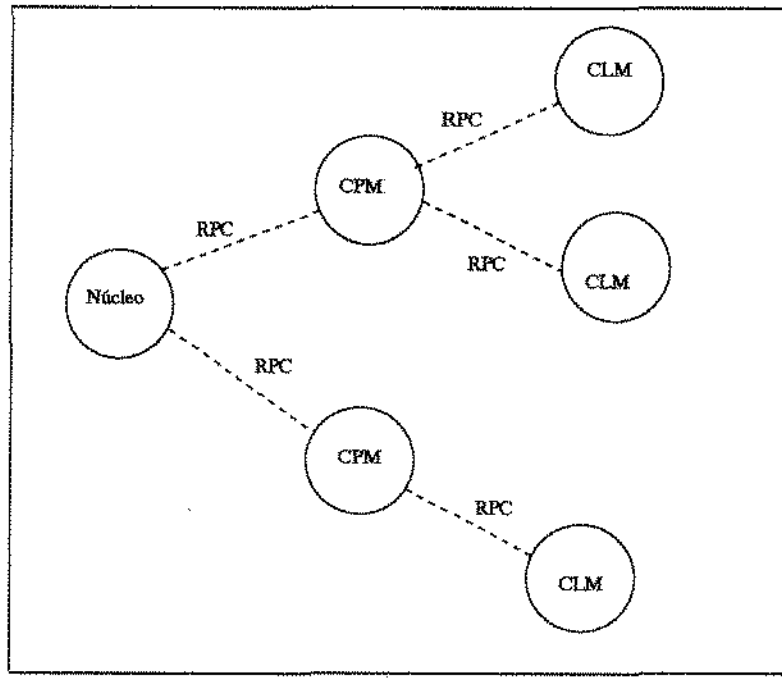

Figura 4.2: **Modelo Descentralizado do protótipo**

forma de comunicação poderia ser adotada, visto que RPC algumas vêzes, como na interação local ao nó, não é a forma de comunicação mais eficiente.

Como mapeamento do modelo de engenharia para o protótipo adotou-se o seguinte: uma cápsula consiste em uma estrutura de gerenciamento formada por um gerente de cápsula associada a um núcleo que tem associada a si um conjunto de clusters (na forma de gerentes de clusters). Um cluster consiste em uma estrutura de gerenciamento formada por um gerente de cluster, associado a um gerente de cápsula, que tem associado a si um conjunto de objetos básicos de engenharia. Um objeto básico de engenharia consiste em um processo Unix que está associado para fins de gerenciamento e modelagem a um gerente de cluster. O conceito de threads não foi considerado na implementação.

Como interface com o programador é oferecida uma biblioteca com um conjunto de chamadas para acesso às funções dos servidores e de inicialização do modelo que foram implementadas.

Um fato que surge é como modelar conceitos do mundo geral em objetos do modelo de engenharia, ou seja como estabelecer a estrutura de cápsulas, clusters, objetos básicos de engenharia no nó, ou em outras palavras, como dizer que os objetos básicos de engenharia B1,B2,B3 pertencem ao Cluster CL1, se objetos básicos de engenharia forem mapeados em processos Unix. Tal função cabe ao módulo de gerenciamento de aplicação imaginado na plataforma *Multiware* em desenvolvimento (capítulo 1, seção 1.4), sendo que a mesma não foi realizada, o que foi feito, como experimento, foi a descrição do mapeamento computacional no modelo de engenharia manualmente na forma da descrição dos templates de cápsulas, clusters e de objetos básicos de engenharia.

4.3 Estruturas de dados do Protótipo

As estruturas do modelo de engenharia foram mapeadas basicamente em um conjunto de listas ligadas, distribuídas pelos diversos gerentes (Núcleo, gerentes de cápsulas, gerentes de clusters). Sendo que tais listas provêem as informações necessárias para gerenciar o modelo. Apresentaremos a princípio como ficaram o mapeamento dos objetos cápsulas, clusters, objetos básicos de engenharia, núcleo e os mecanismos de comunicação de canais e referências de interface.

4.3.1 Referência de Interface de engenharia

O modelo de referência define que referências de interface são identificadores únicos para objetos em um dado domínio (um domínio de gerenciamento de referências de interface, ou seja um conjunto de nós).

Como solução para uma referência de interface adotou-se uma estrutura formada pelo nome do nó a que o objeto pertence, mais um número sequencial local ao nó que identifica o objeto no nó, de forma que tenhamos uma tabela que é indexada por esse número sequencial (devido a dinamismo a tabela está implementada como uma lista). Como cada item da lista temos a descrição do objeto, consistindo do identificador da cápsula a que pertence, e de uma identificação local à cápsula (que forma o identificador de engenharia do objeto).

Em termos de estruturação quando necessita-se estabelecer um *binding* faz-se a criação de uma referência de interface para o objeto criar o canal.

Para representar uma referência de interface será utilizada a estrutura *Name_IR* descrita a seguir, em linguagem C:

```
struct Name_IR{
              char      IP_no[SZ_HOST_NAME];
              long      nome;
              int       status;
              };
```

em que *IP_no* consiste na identificação do nó em que encontra-se a interface, pode-se para tanto fazer uso do número IP da máquina ou um identificador gerado pelo gerente de referência de interface e associada ao nó no momento de sua criação. Optamos pelo nome da máquina. *nome* é o nome local de referência de interface gerada pelo núcleo. *Status* consiste no estado da referência de interface (discutido posteriormente).

Uma forma de visualizar a descrição de cada objeto contido na lista de referências de interface no núcleo pode ser vista na declaração abaixo, bem como pela figura 4.3.

```
struct ID_ENG {
              int cap_name;
              int local_int;
              };

struct Item_IR{
              struct    Name_IR          ir;
              struct    ID_ENG           int_obj;
              struct    Item_IR           * next_ir;
              struct    Item_chan         * list_channel_cliente;
              struct    Item_Chan_Serv    * list_channel_servidor;
                        int               socket;
```

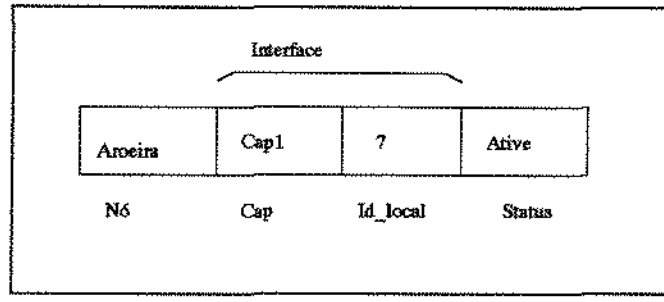

Figura 4.3: **Ítem da lista de referências de interface**

```
};
```

onde *ir* consiste no nome da referência de interface; *int_obj* na identificação de engenharia do objeto referenciado (formada pelo nome da cápsula *cap_nome* e pelo nome do objeto *local_int*); os demais campos da estrutura se referem a dados sobre canais, o quais discutiremos posteriormente. Um exemplo contendo alguns dos ítens da estrutura *Item_IR* pode ser visto na figura 4.3, que apresenta a descrição de um item da lista de referências de interfaces para um objeto ativo no nó.

No processo de busca de uma referência de interface a mesma contém o identificador do nó em que está, se o objeto não estiver no nó (devido à migração) estará assinalado na entrada correspondente no nó da referência de interface para onde migrou o mesmo (via tabela do relocador) ou se foi desativado, caso contrário o objeto terminou. Um exemplo é o da figura 4.4, que mostra um conjunto de ítens da lista de referências de interface, no caso duas interfaces de objetos ativos no nó, e uma interface de um objeto que sofreu migração para outro nó (como visto a seguir).

O campo de estado *status* de *Name_IR* pode ter um dos seguintes valores:

- *Ative* - Indica um objeto ativo.
- *Desative* - Denota um objeto desativado.
- *Migrate* - Objeto que foi migrado. Deve-se procurar na tabela de relocação do relocador local para onde o objeto migrou.

A tabela do relocador possui uma estrutura semelhante à da figura 4.5, descrita abaixo em linguagem C, onde no exemplo podemos encontrar a entrada correspondente a da referência de interface da figura 4.4 que encontra-se com o estado assinalado como *migrate*.

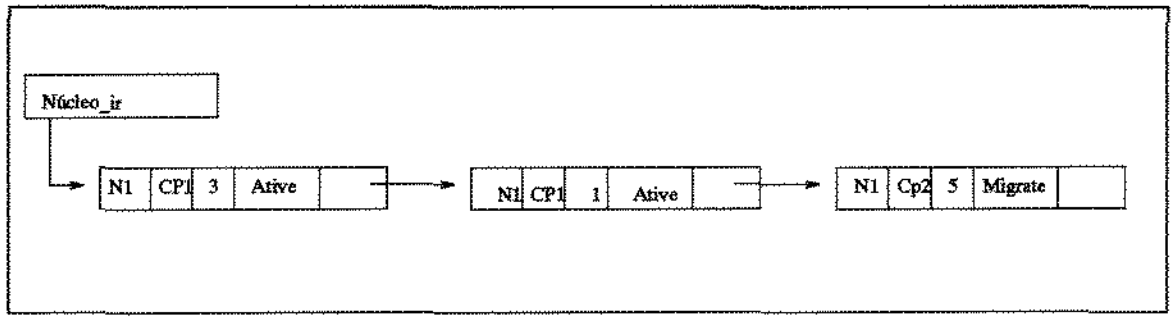

Figura 4.4: **Exemplo de lista de referências de interface**

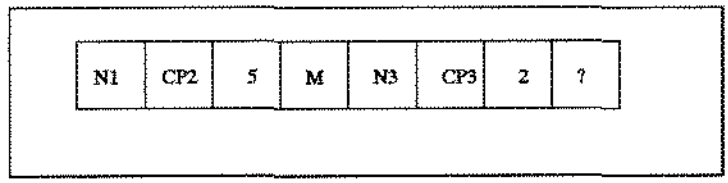

Figura 4.5: **Exemplo de ítem da tabela do relocador**

```
struct Reloc_table{
                    struct Item_IR    Local;
                    struct Item_IR    Destino;
                    };
```

em que *Local* consiste na referência de interface local antes da migração e *Destino* na nova referência de interface. Se for realizada uma consulta é retornada a nova referência de interface. Contudo, podemos ter como *Destino* apenas o nome do nó destino provendo assim uma estrutura mais simples ao relocador.

Uma referência de interface contém ainda os dados tradicionais, como dados sobre grupos de réplica.

4.3.2 Cápsula

Uma cápsula no mapeamento é representada como um servidor *CPM* e internamente, neste servidor é representada como uma lista que contém informação sobre os clusters presentes na cápsula e dos recursos que possui.

```
struct CAPSULE  {
                  struct Descriptor_Capsule    descrip;
                  struct CLUSTER               * next_cluster;
```

```
        struct CAPSULE                  * next_capsule;
    };
```

onde *descrip* consiste em um descritor da cápsula com suas características, e *Next_Cluster* em um ponteiro para o próximo cluster na lista. A não existência de um próximo cluster é especificada por *Next_Cluster* igual a *NULL*. *Next_capsule* consiste em um ponteiro para a próxima cápsula na lista de cápsula do núcleo, sendo que esse campo é usado quando manipulado pela núcleo para manter a lista de cápsulas existentes no mesmo.

O descritor de cápsula pode ser visto abaixo contendo a identificação da cápsula no nó (*id*); um identificador para a próxima interface a ser criada na cápsula (*next_id_interf*) (como, por exemplo, para o próximo gerente de cluster criado na cápsula); o nome do template que foi utilizado para instanciar a cápsula (*template_pai*); o descritor do servidor *CPM* e núcleo associados à cápsula (*serv* e *nucleo_pai* respectivamente).

```
struct Descriptor_Capsule{
        struct   INTERFACE    id;
                 int          pid;
        struct   Servidor     nucleo_pai;
        struct   Servidor     serv;
                 long         next_id_interf;
                 char         template_pai[SZ_FL_NAME];
    };
```

A estrutura de *INTERFACE* pode ser vista abaixo, consistindo de *nome* o nome do objeto, *status* o estado do objeto e *tipo* o tipo do objeto (que pode ser cápsula, cluster, objeto básico de engenharia).

```
struct INTERFACE{
        int    nome;
        int    status;
        int    tipo;
        };
```

A estrutura de *Servidor*, que consiste no descritor de um servidor RPC, pode ser vista abaixo:

```
struct Servidor{
       char            protocolo[SZ_PROTOCOL_NAME];
       unsigned  long  versao;
```

```
unsigned  long  prog_number;
unsigned  short portnum;
unsigned  int   numero_processo;
char            swap_dir[SZ_FL_NAME];
};
```

em que *prog_number* consiste no número do servidor RPC associado pelo *portmapper* ao servidor, *versao* na versão do servidor, *portnum* no número da porta *TCP* associada ao servidor, *protocolo* o nome do protocolo associado ao servidor, *numero_processo* no número do *process id* do processo servidor, e *swap_dir* no nome do diretório a ser usado como interface de armazenamento (conforme discutiremos na seção 4.4) pelo servidor.

A de se notar que não foi tratado aqui o problema de espaço de endereçamento compartilhado e encapsulado para os objetos na cápsula. Visto que nossa implementação de cápsula no final conterá um conjunto de processos representando os objetos básicos de engenharia dos clusters, funcionando como uma unidade de gerenciamento, não permitindo desta forma o compartilhamento de espaço de endereçamento entre seus objetos internos. Uma forma de solucionar isso seria a implementação de um módulo gerenciador de memória no espaço de endereçamento da cápsula pelo uso das funções de memória compartilhada do Unix, contudo tal solução não seria a ideal visto que tenta duplicar um módulo do sistema operacional colocando-o no espaço do usuário necessitando assim que os gerentes de memória das diversas cápsulas colaborassem estritamente para manter a ordem.

4.3.3 Cluster

Um cluster consistirá em um gerente de cluster implementado como um servidor baseado em RPC que gerenciará o cluster mantendo as informações sobre seus objetos básicos de engenharia em uma lista formada por estruturas *Cluster* como a descrita a seguir.

```
struct CLUSTER  {
          struct Descriptor_Cluster   descrip;
          struct CLUSTER              * next_cluster;
          struct BEO                  * next_beo;
          };
```

onde *Next_Beo* consiste em um ponteiro para o primeiro objeto básico de engenharia do cluster. Os objetos básicos de engenharia encontram-se organizados em forma de uma lista. *Next_cluster* é um ponteiro para o próximo cluster na lista

de clusters da cápsula (usado pelo gerente de cápsula), se for o último membro da lista esse campo terá valor igual a *NULL*. *Descrip* consiste no descritor do gerente de cluster sendo assim formado:

```
struct Descriptor_Cluster{
        struct   INTERFACE id;
                 int       pid;
        struct   INTERFACE d_cpm;
        struct   Servidor  nucleo_pai;
        struct   Servidor  cpm_pai;
        struct   Servidor  serv;
                 char      template_pai[SZ_FL_NAME];
                 int       checkpoint;
                 int       delete;
        };
```

onde *id*, *pid*, *template_pai* e *nucleo_pai* têm seu correspondente na estrutura *Descriptor_Capsule*. *cpm_pai* consiste no servidor CPM ao qual o cluster está associado, *serv* nos dados sobre o servidor CLM do cluster. *d_cpm* na identificação de engenharia do CPM pai. *checkpoint* assinala se o cluster suporta *checkpoint* ou não, da mesma forma *delete* em relação à terminação.

4.3.4 Objeto Básico de engenharia

Um objeto básico de engenharia será representado como uma estrutura contendo sua descrição, *descrip*, e um ponteiro para o próximo objeto básico de engenharia do cluster *next_beo*. Podemos ver a estrutura de um objeto como definido em *BEO*.

```
struct BEO {
          struct  Descriptor_BEO  descrip;
          struct  BEO             * next_beo;
                    } ;
```

A descrição de um objeto básico de engenharia será formada, a exemplo do cluster, pela identificação do objeto no cluster (*id*), pela identificação do template que o originou (*template_pai*) e *pid* que consiste no *process id* do processo.

```
struct Descriptor_BEO{
       struct INTERFACE  id;
              int        pid;
```

```
    char        template_pai[SZ_FL_NAME];
};
```

Em caso de execução representamos a unidade executável como um arquivo executável associado à descrição de um objeto básico de engenharia (em seu template), e quando realizada a instanciação usamos de uma chamada ao sistema *exec* (no caso do Unix) para executar o arquivo. Surge nesse contexto de um processo por objeto básico de engenharia o problema de memória compartilhada, já que o espaço de endereçamento, pelo conceito ODP, em cápsulas distintas serão distintos, e objetos em um cluster compartilham memória. Para simular tal fato podemos usar de um artifício de memória compartilhada do sistema Unix, como mencionamos na seção 4.3.2.

4.3.5 Threads

Não implementamos threads em nosso protótipo, contudo os mesmos podem ser simulados via processos (um thread igual a um processo) com o uso de memória compartilhada entre os mesmos, deixando assim nesse caso do objeto básico de engenharia de ser a unidade executável, e passando a ser os threads.

4.3.6 Núcleo

Consiste em um processo servidor criado na inicialização do nó que é responsável pelo gerenciamento de recursos providos pelo nó. Sendo responsável pela criação de canais, cápsulas e pelo gerenciamento de threads (em um nível superior - o gerenciamento básico é feito pelo sistema operacional se o sistema operacional prover threads, caso contrário é responsável por prover tal característica).

O núcleo é representado como um servidor que tem associado a si um conjunto de cápsulas e referências de interfaces que são gerenciadas como listas. Sendo que tal estrutura de listas é agrupada na estrutura *NUCLEO* vista a seguir.

```
struct NUCLEO{
          char      node_name[SZ_FL_NAME];
    struct Servidor serv;
    struct CAPSULE  *next_capsule;
           long     next_id_capsule;
    struct Iref     ir;
    struct Canal    canal;
           };
```

onde *node_name* é o nome do nó, criado no momento da introdução do mesmo; *Next_capsule* consiste no início da lista de cápsula no nó, sendo que quando criamos uma nova cápsula o que fazemos é acrescentar a descrição da mesma nessa lista; *Next_id_capsule* que é o identificador da próxima cápsula a ser instanciada no nó; *ir* consiste em dados sobre referências de interface, *Canal* contém os dados sobre canais no nó. *serv* consiste nos dados do servidor RPC que representa o núcleo.

A estrutura de *Iref* pode ser vista abaixo sendo composta por *next_id_ir* que consiste na próxima identificação de referência de interface que for gerada localmente; *l_ir_domain* na informação sobre o domínio de gerenciamento de referências de interface a qual o nó pertence, e por *int_ref_local* na lista de referências de interfaces locais ao nó.

A estrutura *IR_DOMAIN* consiste nos dados dos nós pertencentes ao domínio de referência de interface, no caso o nome .

```
struct Iref{
        struct      IR_DOMAIN  l_ir_domain[N_MAX_IRD];
        struct      Item_IR    * int_ref_local;
                    long       next_id_ir;
          };
```

A estrutura de canais *Canal* é apresentada agora:

```
struct Canais_info{
       struct Descriptor_Channel * next_channel;
       long                        next_channel_id;
       };

struct Canal{
       struct Canais_info      cliente;
       struct Canais_info      servidor;
       struct PROTOCOL         protocols[N_MAX_PROTOCOL];
};
```

em que temos as descrições dos canais do qual o objeto presente no nó participante do canal atua como cliente (*cliente*), ou como servidor (*servidor*), além dos dados sobre os protocolos (*protocols*), no caso o nome. Na estrutura *Canais_info* temos os campos *next_channel_id* que é nome do próximo canal a ser criado, e *next_channel* que consiste na lista de canais no núcleo. A estrutura *Descriptor_Channel*, o descritor de canal, será apresentada na seção 4.3.7.

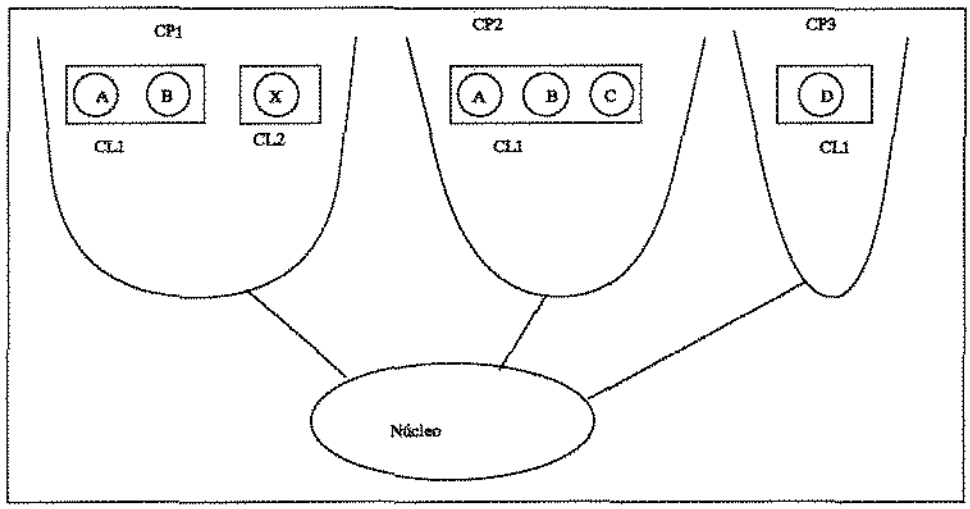

Figura 4.6: **Exemplo de aplicação no modelo de engenharia**

Um exemplo do uso de tais estruturas discutidas pode ser visto nas figuras 4.6 e 4.7. A figura 4.6 consiste no modelo de engenharia e a figura 4.7 na nossa representação da mesma através dos servidores.

4.3.7 Canal

Canais foram implementados como uma estrutura consistindo de dois sockets associados. Considerando o uso de sockets no processo de criação de um canal teremos um módulo responsável pela criação e configuração de canais no núcleo do nó , sendo que objetos só tem conhecimento da identificação do canal e das operações que podem ser realizadas sobre o mesmo.

Como identificador do canal podemos retornar no caso de sockets o próprio número de socket ou um número que corresponde ao deslocamento em uma tabela com os dados dos sockets, ficando assim menos sucetível a mudanças nos canais.

Descreveremos agora as estruturas necessárias para suportar canais em nossa proposta, primeiro as necessárias do lado cliente do canal e posteriormente as necessárias do lado servidor.

A estrutura *Descriptor_Channel* que descreve o lado cliente de um canal pode ser vista abaixo, sendo que na mesma *nome* consiste no nome do canal, *status* no estado do mesmo, *nome_destino* o identificador do canal no outro nó, *origem* e *destino* consistem nas referências de interfaces associadas a origem (*origem*) do canal e ao destino (*destino*), *socket_destino* no número do socket associado ao

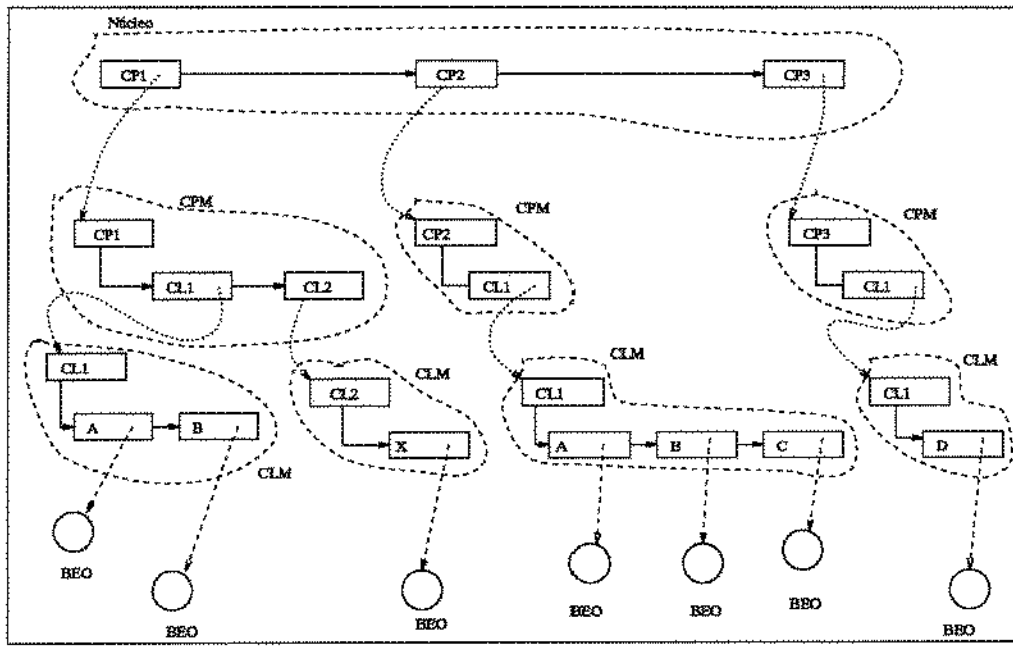

Figura 4.7: **Emprego das estruturas de dados**

canal do lado do servidor e *next_channel* no ponteiro para o próximo canal na lista de canais do núcleo.

```
struct Descriptor_Channel{
            int                 nome;
            int                 status;
            int                 nome_destino;
            int                 socket_destino;
      struct Name_IR            destino;
      struct Name_IR            origem;
      struct Canal_Config       cfig;
      struct Descriptor_Channel   *next_channel;
      };
```

A estrutura de *cfig* que descreve a configuração do canal é apresentada agora.

```
struct Canal_Config{
      char     protocolo[SZ_PROTOCOL_NAME];
      int      connection;
      int      tamanho;
      };
```

em que *protocolo* o protocolo usado, *connection* se o mesmo é um socket do tipo stream ou datagrama, e *tamanho* o tamanho máximo do pacote de dados enviado.

Ao se criar um canal, após receber os dados do lado servidor, anexamos à referência de interface do lado cliente um ponteiro para o descritor de canais, isso é feito pela estrutura *Item_Chan*, que faz parte de *Item_IR*, apresentada agora:

```
struct Item_chan{
          struct Descriptor_Channel  *dcanal;
          struct Item_chan         * next_channel;
              };
```

no caso formada pelo ponteiro para o descritor de canal *dcanal* e por um ponteiro para o próximo canal da lista associada à referência de interface *next_channel*.

Do lado do servidor do canal, quando estabelecido o mesmo, é associado a referência de interface do servidor à descrição do lado cliente do canal na forma de uma lista associada à *Item_IR* formada pela estrutura *Item_Chan_Serv*, encontrada abaixo.

```
struct Item_Chan_Serv{
```

```
            int                 nome;
            int                 nome_cli;
struct      Name_IR             ir_cli;
struct      Canal_Config        cfig;
struct      Item_Chan_Serv      *next_channel;
      };
```

temos aqui o nome local do canal *nome*, *nome_cli* que consiste no nome do lado do cliente do canal, *ir_cli* na referência de interface do cliente, *next_channel* em um ponteiro para o próximo canal na lista e *cfig* na descrição da configuração do canal.

A implementação de canais com o uso de sockets consiste em uma simplificação dos canais propostos no RM-ODP, sendo que o papel de stubs não existe na implementação, e binders podem ser vistos, com grande simplificação, como presente nas funções para a conexão entre pares de sockets (como exemplo, *Binding* do Unix), sendo ainda desempenhado parcialmente pela função *bind_objects_1* (seção 4.5), que cria parte da estrutura do binding relacionadas às referências de interface. Contudo uma implementação futura de canais com uso de outra técnica, provavelmente DII do ORB [42], deve ser considerada em uma extensão.

4.4 Estrutura de diretórios e interface de armazenamento

Como forma de permitir que diretórios contendo informações do modelo de engenharia proposto, como por exemplo o *checkpoint* de objetos, sejam visíveis a diversos nós temos necessidade que a árvore de diretórios do modelo proposto esteja montada em todos os nós de um dado domínio. No caso do sistema operacional Unix em uso, SunOS, faz-se uso do NFS (Network File System , visto parcialmente no capítulo 1). A de se notar que o NFS consiste em um conceito bem estável no mundo Unix sendo que existem implementações do mesmo para a maioria dos sistemas do mercado não nos levando a grande problemas ao utilizarmos o mesmo em nossa implementação ao atendermos um ambiente diferente do SunOS, em extensões futuras.

Em relação à estrutura de diretório do modelo de engenharia proposto definimos a estrutura da figura 4.8 a qual detalharemos a seguir a função de cada diretório componente.

- **/odp** . Consiste no ponto de montagem primário para NFS delimitando que o seu conteúdo será visível em um conjunto de nós. Sendo que internamente

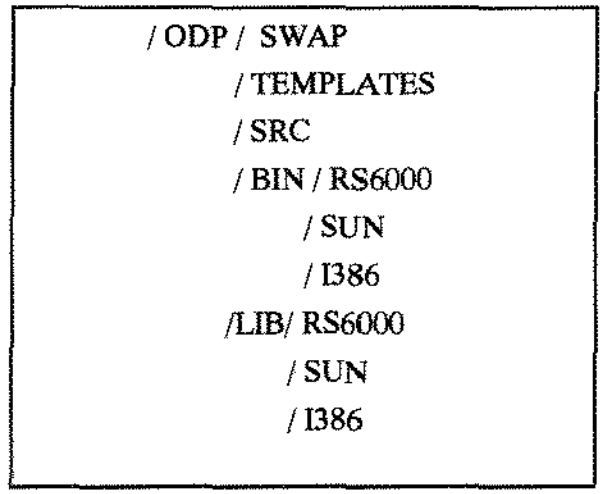

Figura 4.8: **Estrutura de diretórios do modelo**

o mesmo é estruturado de forma a considerar a heterogenidade dos sistemas em uso.

- **/odp/templates** . Contém os arquivos de templates de cápsulas, clusters e de objetos básicos de engenharia, a descrição do formato de cada arquivo de template será dada posteriormente.

- **/odp/src** . Contém o código fonte dos programas para a compilação e geração de objetos básicos de engenharia, levando em conta as regras descritas nos templates em **/odp/templates**. A existência de fontes aqui é considerada visto que possa em algum momento não existir um executável de um objeto básico de engenharia para uma arquitetura específica e seja necessário a geração dinâmica do mesmo.

- **/odp/bin** . Contém os executáveis dos objetos básicos de engenharia organizados em diretórios pelas arquiteturas suportadas , no exemplo diretórios **SUN** (para sparc sun) , **RS6000** (para rs6000 IBM) e **I386** (para 386 Intel). Sendo que aqui também estão contidos os executáveis dos gerentes que compõem o modelo para cada arquitetura (Ex.: núcleo).

- **/odp/swap** . Contém os arquivos com *checkpoint* de cápsula, clusters e objetos básicos de engenharia como será descrito a seguir.

- **/odp/lib** . Contém, a exemplo de **/odp/bin**, bibliotecas dependentes da

arquitetura utilizada.

Descrevermos agora nossa solução para as interfaces de armazenamento para *checkpoint* de cápsulas, clusters e objetos básicos de engenharia.

O processo de criação de interfaces de armazenamento é distribuído na verdade pelos diversos gerentes (núcleo, gerente de cápsula e gerente de cluster) sendo que funciona da seguinte forma.

No processo de introdução de um nó é verificado se existe um diretório com nome */swap/node_name* , sendo que se o mesmo existir é questionado se o diretório deve ser removido (ou se existir *checkpoints* no interior do mesmo e a existência deste se deverem a uma falha no nó, deve-se fazer a restauração dos *checkpoints*), caso contrário é criado um diretório com tal nome.

Em relação às interfaces para a cápsula a função *instantiate_capsule* cria um diretório com nome */swap/node_name/cpm_id* quando é realizada a criação do gerente de cápsula. Da mesma forma quando é criado um gerente de cluster pela função *instantiate_cluster* é criado o diretório */swap/node_name/cpm_id/clm_id* sendo o mesmo associado ao novo cluster.

A função *checkpoint_cluster* cria um arquivo de template de desativação do cluster para o cluster e um arquivo para cada objeto básico de engenharia contendo o seu *checkpoint* (se os mesmos suportarem *checkpoint*) , sendo que tais arquivos estarão no diretório correspondente do gerente de cluster.

Da mesma forma a função *checkpoint_capsule* realizará a criação de um arquivo de template de desativação para a cápsula que conterá as informações necessárias para reativar a cápsula, e além disso a estrutura já citada quando da criação do *checkpoint* de cluster, para cada cluster na cápsula.

Um exemplo pode ser visto na figura 4.9 que contém uma estrutura de um nó e na figura 4.10 que mostra a estrutura de diretórios e arquivos gerados no caso de *checkpoint*. Podemos notar na estrutura de árvore da figura 4.10 a existência, além das estrutura mencionada, de arquivos de templates de reativação tanto para as cápsulas, como para os clusters. Tais arquivos descrevem como restabelecer um determinado objeto desativado/*checkpointed*, levando em conta fatos como a arquitetura suportada, visto que a reativação só é permitida em uma mesma arquitetura.

4.4.1 Arquivos de templates

Os arquivos de templates contêm descrições que permitem a partir de um arquivo fonte gerar o executável do mesmo, bem como informações sobre a estruturação da aplicação dividindo a mesma em clusters, cápsulas e objetos básicos de engenharia.

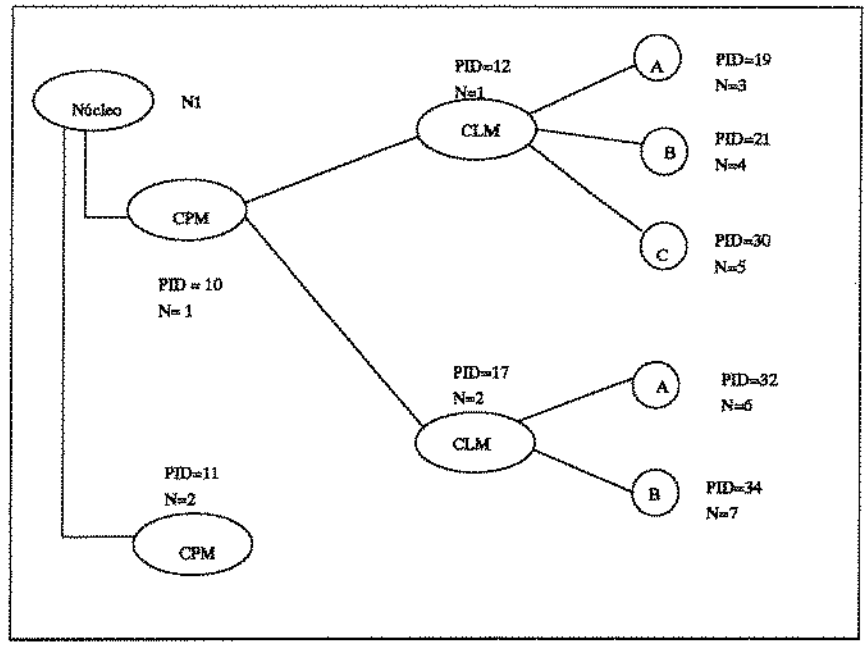

Figura 4.9: **Exemplo de Estrutura para Swap**

Um arquivo de template de cápsula é formado pelo nome da cápsula e pelos nomes dos arquivos contendo a descrição de cada cluster na cápsula.

Um arquivo de template de cluster é formado pelo nome do cluster e pelos nomes dos arquivos contendo a descrição de cada objeto básico de engenharia componente do cluster.

Um arquivo de template de objeto básico de engenharia é formado pelo nome do objeto básico de engenharia e pelo nome de um arquivo executável que consistirá no processo a ser executado no momento da criação do objeto básico de engenharia.

4.5 Descrição das operações de gerenciamento

Apresentaremos agora uma descrição das funções de gerenciamento apresentadas no Capítulo 3, detalhando cada função de gerenciamento levando em conta as estruturas detalhadas anteriormente e o modelo considerado para o protótipo.

Na medida do possível tentaremos apresentar descrições algoritmicas para as funções.

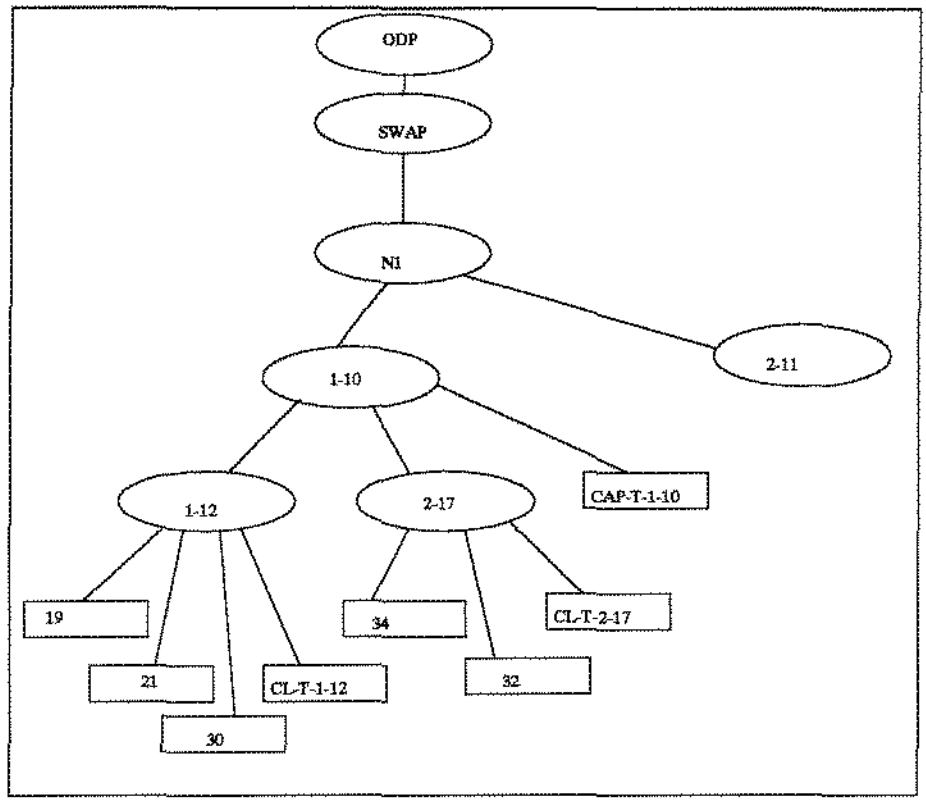

Figura 4.10: **Arquivos/diretórios de swap gerados no caso da figura anterior**

Os servidores serão a princípio apresentados pelas suas descrições em XDR, sendo que as mesmas são colocadas abaixo.

- **Servidor Núcleo.** Realiza as funções do núcleo e de gerenciamento de referências de interface, bem como trata dos canais.

```
program RMODPPROG{
version RMODPVERS{
     int                NODE_INIT(void)=1;
     Descriptor_Capsule INSTANTIATE_CAPSULE ( string ) = 2;
     Name_IR            REGISTRE_IR ( ID_ENG)=3;
     Name_IR            VALIDATE_IR ( Name_IR)= 4;
     int                REMOVE_IR ( Name_IR ) = 5;
     int                TERMINATE_CPM( Descriptor_Capsule) = 6;
     int                BIND_OBJECTS (Bind_Par) = 7;
     Descriptor_Channel REMOTE_BIND (Descriptor_Channel) = 8;
     int                REMOVE_BIND ( int ) = 9;
     Descriptor_Channel GET_CHANNEL_INFO(int)=10;
     int                REMOVE_REMOTE_BIND ( Remove_Par  ) = 11;
  }=1; }=0x20000080;
```

- **Servidor CPM.** Realiza a função de gerenciamento de cápsula.

```
program CPM_PROG{
version CPM_VERS{
    int                INIT_CPM (Descriptor_Capsule )=1;
    Descriptor_Cluster INSTANTIATE_CLUSTER ( string )=2;
    Descriptor_Cluster REATIVE_CLUSTER ( string )=3;
    Str_Status         CHECKPOINT_CAPSULE ( void )=4;
    Str_Status         DESATIVE_CAPSULE ( void ) =5;
    int                DELETE_CAPSULE (void ) =6;
    int                TERMINATE_CLM ( Descriptor_Cluster) = 7;
  }=1; }=0x20000081;
```

- **Servidor CLM.** Realiza a função de gerenciamento de cluster.

```
program CLM_PROG{
version CLM_VERS{
    int        INIT_CLM (Descriptor_Cluster)=1;
    int        CREATE_BEOS ( TEMPLATE_CLUSTER)=2;
```

```
    int         RESTORE_BEOS ( TEMPLATE_CLUSTER_REAT)=3;
    Str_Status CHECKPOINT_CLUSTER ( void ) =4;
    Str_Status DESATIVE_CLUSTER ( void)= 5;
    int         DELETE_CLUSTER (void) = 6;
}=1; }=0x20000082;
```

A seguir será realizada a descrição das funções dos gerentes do modelo proposto, bem como de funções auxiliares e das funções que compõem a biblioteca para o programador. As funções serão apresentadas com suas interfaces em linguagem C, considerando o processo de conversão da linguagem XDR para linguagem C, descrito em [56].

A de se observar que foi implementado apenas um subconjunto das funções descritas no capítulo 3 no protótipo, devido à natureza do mesmo, sendo que algumas vêzes a implementação não foi realizada devido à dificuldade inerente (como no caso de migração de cluster), e algumas vêzes por se fazer uso de características da plataforma em uso, Unix (como no caso de funções para gerenciamento de referências de interface em que se fez uso de características do Unix para tal fim). Em especial podemos citar como não implementadas as funções para: reativação de cápsula, gerenciamento de threads e gerência de relógios, as duas últimas do núcleo.

4.5.1 Interface de Gerenciamento de nó

O gerenciamento do nó fornecido pelo servidor núcleo, descrito na seção anterior em XDR, provê as funções necessárias para lidar com a inicialização do nó, criação de cápsulas, e gerenciamento de referências de interface e canais. Veremos a descrição agora de cada função.

int * node_init_1()

Essa função realiza a inicialização das listas do núcleo. A mesma deve ser chamada pelo usuário que carregou o núcleo explicitamente, sendo que realiza o seguinte:

- Obtém o nome da máquina em que o núcleo está, e guarda em *nucleo.node_name*.
- Inicializa o contador de cápsula *nucleo.next_id_capsule* para 1, sendo que o mesmo será incrementado sempre quando da criação de uma nova cápsula.
- Inicializa o contador de referências de interface *nucleo.ir.int_ref_local*, e contador de canais clientes, e servidor para 1.

- Inicializa os ponteiros das listas do núcleo para NULL.
- Cria o diretório */odp/swap/nucleo.node_name* .
- termina.

um fato que vale ser discutido aqui consiste na opção por identificadores de objetos no modelo. Para o núcleo, CPM e CLM poderíamos usar simplesmente o número do servidor RPC correspondente ao mesmo ou o número do processo, e para objetos básicos de engenharia o número do processo (o que foi adotado). Contudo, decidimos manter um esquema de identificadores independente do portmapper (no caso de RPC) e do Unix (no caso de número de processo) de forma a termos um modelo mais facilmente migrável para plataformas heterôgenas.

Descriptor_Capsule * instantiate_capsule_1 (char * temp_name[])

Essa função realiza a criação de uma cápsula em um nó com a geração do gerente de cápsula correspondente. A mesma recebe o nome de um arquivo contendo o template de cápsula e retorna o descritor do gerente de cápsula associado à nova cápsula, ou um erro, funcionando da seguinte forma:

- Lê o template de cápsula para a memória, se o mesmo não existir é retornado um erro.
- Gera um número para ser associado ao novo servidor CPM a ser criado, visto que por termos mais de um gerente de cápsula no mesmo nó não podemos fazer uso do número gerado pelo rpcgen (discutido em [56]).
- Executa o arquivo que contém o código objeto do CPM, passando como parâmetro o número gerado no ítem anterior para o mesmo se registrar junto ao portmapper.
- Aloca um descritor de cápsula dinamicamente.
- Gera um diretório de swap para a nova cápsula (conforme o que foi descrito na seção 4.4).
- Preenche os dados do Descritor de cápsula com o identificador do processo (*process id*) do CPM, nome do diretório de swap, número e versão do servidor CPM.
- Chama a rotina *init_cpm_1* do novo gerente de cápsula (que realiza a inicialização do CPM) com o descritor de cápsula como parâmetro.

- Anexa o descritor de cápsula à lista de descritores do núcleo.
- Termina retornado o descritor de cápsula como resultado.

se ocorrer erros o processo de criação da cápsula é terminado, e um erro é retornado.

int * terminate_cpm_1(Descriptor_Capsule * dcap)

Função que realiza a terminação de um gerente de cápsula.

- Localiza o descritor da cápsula.
- Termina o processo CPM.
- Elimina o descritor da lista de cápsula.

Se ocorre algum erro o processo é terminado e o erro é retornado. Se existir algum objeto no interior da cápsula o mesmo deve ser terminado antes, ou via *delete_cluster_1* ou *delete_capsule_1*.

Name_IR * registre_ir_1 (ID_ENG identificador)

Função que recebendo o identificador de engenharia de um objeto e o número de um socket (que pode ser nulo) gera e retorna uma referência de interface para o objeto. Atua como descrito abaixo.

- Verifica se já existe uma referência de interface associada ao identificador, se existir retorna um erro.
- Aloca um descritor (*Aux*) para a lista de referências de interface.
- Gera um número local para a referência de interface.
- Copia no descritor *Aux* os dados recebidos e o identificador gerado.
- Inicializa os ponteiros do descritor alocado para NULL.
- retorna a referência de interface formada pelo nome do nó e pelo identificador local.

Um fato que deve ser notado é que se estivermos registrando um servidor teremos um socket associado ao mesmo para a criação de canais, sendo contudo o mesmo um parâmetro opcional.

int * remove_ir_1 (Name_IR dir)

Essa função realiza a remoção de uma referência de interface associada a um objeto.

- Localiza o descritor associado a referência de interface.
- Se a referência de interface for local realiza a remoção da mesma da lista e retorna.
- Se for de outro nó ou se migrou (status = MIGRATE) solicita ao núcleo do nó destino a eliminação da mesma, sendo que retorna possíveis erros.
- Remove os canais associados .

Name_IR * validate_ir_1 (Name_IR * dir)

Quando um processo deseja realizar a validação de se um determinado objeto continua válido no nó, se terminou ou foi migrado para outro nó faz uso dessa função que atua dá seguinte forma.

- Localiza o descritor da referência de interface.
- Se a referência de interface é local e continua válida (status = ATIVE) retorna.
- Se a referência de interface é remota ou se migrou então solicita ao núcleo do nó destino a validação da referência de interface.
- Se a referência de interface migrou e é válida no nó destino então muda o estado da mesma de MIGRATE para REMOTE.
- Se a referência de interface é remota e não consta na lista local então inclui a mesma na lista local como REMOTE.
- Se a referência de interface não existe no nó destino então remove a mesma da lista local. Remove os canais associados.
- Se é valida retorna então a nova referência de interface (se MIGRATE) para o chamador.

O que tentamos aqui é manter além da consistência uma cache local de referências de interface.

int * bind_objects_1(Canal_Info * cinfo)

Essa função realiza a criação de um canal sendo chamada pelo cliente do canal e estabelecendo o *binding* entre o cliente e o servidor. Retorna o identificador do canal que é único no nó.

Aqui existiam duas possibilidades em relação ao uso do canal: o cliente pode fazer uso do mesmo, ou quem tiver posse do identificador do canal pode fazer uso do mesmo. Foi adotado a segunda semântica, sendo contudo que a alteração da semântica dependendo da situação pode ser feita via parâmetro especificando que o canal é privado (PRIVATE).

A função funciona da seguinte forma:

- Verifica se existe a referência de interface do objeto cliente via *validate_ir_1*.
- Verifica se existe a referência de interface do objeto servidor via *validate_ir_1*.
- Aloca uma estrutura para o descritor de canal
- Cria um identificador de canal para o canal.
- Solicita via *remote_bind_1* a criação do lado do servidor do canal.
- Guarda os valores retornados por *remote_bind_1* (e a configuração para o canal recebida, *cinfo*) no descritor.
- Associa o descritor à lista de canais clientes da referência de interface.
- Retorna o identificador gerado para o canal.

Erros que porventura venham a ocorrer na execução da função são retornados pela mesma encerrando sua execução.

Descriptor_Channel * remote_bind_1 (Descriptor_Channel * dca)

Essa função realiza a associação dos dados do canal à referência de interface do objeto servidor. Funcionando da seguinte forma:

- Aloca uma estrutura para o descritor do canal do servidor.
- Gera um identificador para o lado servidor do canal.
- Salva dados recebidos do cliente no descritor.
- Insere o descritor na lista de canais servidores da referência de interface.
- retorna o identificador do canal e o socket do lado do servidor.

Descriptor_Channel * get_channel_info_1 (int * canal)

Função que dado o identificador de um canal retorna a descrição sobre o mesmo.

- Localiza o descritor do canal na lista do núcleo.
- Retorna o descritor do canal.

int * remove_bind_1 (int * canal)

Função que realiza a remoção de um canal criado por *bind_objects_1*.

- Localiza o descritor do canal .
- Elimina o lado servidor do canal via *remove_remote_bind_1*.
- Remove a associação com a referência de interface.
- Remove o descritor do canal da lista de canais.

int * remove_remote_bind_1(int * canal);

Essa função remove a associação existente entre o canal e à referência de interface do servidor. Para tanto remove o descritor da lista de canais servidores associada a referência de interface.

4.5.2 Interface de gerenciamento para Cápsula

O gerenciamento de cápsula é realizado pelo gerente de cápsula, que teve sua interface apresentada em XDR, na seção 4.5, sendo que agora discutiremos cada uma das funções do mesmo.

int init_cpm_1 (Descriptor_Capsule * dcap)

Essa função, a exemplo de *node_init_1*, realiza a inicialização das estruturas do gerente de cápsula. Realiza a cópia do descritor de cápsula que recebeu do núcleo no CPM, inicializando ainda o ponteiro para a lista de cluster para NULL e o contador de identificadores para 1.

Descriptor_Cluster * instantiate_cluster_1 (char * name_temp[])

A função *instantiate_cluster_1* recebe o nome de um arquivo contendo um template de cluster e realiza a criação de um novo cluster com seus objetos básicos de engenharia e seu gerente de cluster. A mesma é descrita a seguir.

- Realiza a leitura do template de cápsula para a memória, se o mesmo não existir é retornado um erro.
- Gera um número para ser associado ao novo servidor CLM a ser criado.
- Executa o arquivo que contém o código objeto do CLM, passando como parâmetro o número gerado no ítem anterior para o mesmo se registrar junto ao portmapper.
- Aloca um descritor de cluster dinamicamente para o novo cluster.
- Gera um diretório de swap para o novo cluster (conforme o descrito na seção 4.4).
- Preenche os dados do descritor de cluster com o identificador de processo (*process id*) do CLM , nome do diretório de swap, número e versão do servidor CLM.
- Chama a rotina *init_clm_1* do novo gerente de cluster (que realiza a inicialização do CLM), tendo como parâmetro o descritor do cluster.
- Chama a rotina *create_beos_1*, do novo gerente de cluster, que realiza a criação dos objetos básicos de engenharia do cluster, com o nome do template de cluster como parâmetro.
- Anexa o descritor de cluster à lista de descritores na cápsula.
- Termina retornado o descritor de cluster como resultado.

A tarefa de criar os objetos básicos de engenharia fica a cargo do novo gerente de cluster, sendo que a mesma é realizada na função *create_beos_1*, como já dissemos.

Descriptor_Cluster * reative_cluster_1 (char * name_temp[])

Essa função cuida da reativação de um cluster que foi desativado ou pela função de desativação do CPM ou do CLM, ou pela função de *checkpoint* do CPM ou CLM. A mesma recebe como parâmetro o nome de um arquivo contendo um template de reativação de cluster.

A mesma deve, com base no template, reativar os *checkpoints* e realizar as alterações em seus identificadores, atuando da seguinte forma.

- Lê o template de reativação, se o mesmo não existir retorna um erro.
- Aloca um descritor de cluster para o novo cluster.
- Cria um novo gerente de Cluster, conforme descrito na seção anterior.
- Realiza para cada *checkpoint* de objeto descrito no template a sua reativação (via *restore_beos_1*).
- Anexa o Descritor do cluster à lista de clusters da cápsula.
- Retorna o descritor do novo cluster.

A tarefa de realizar a alteração dos identificadores no objeto restaurado fica a cargo da função *restore_beos_1* (do CLM).

char * checkpoint_capsule_1 ( )

Essa função realiza o *checkpoint* de uma cápsula gerando um arquivo de template de *checkpoint* e a estrutura de *checkpoint* para cada cluster na mesma conforme descrito na seção 4.5.3.

- 1. Para cada cluster na cápsula faz o seguinte:

  1.1 Obtém o descritor do cluster da lista do CPM.

  1.2 Chama a função *checkpoint_cluster_1* do CLM associado ao cluster para Gerar o *checkpoint* do cluster, que retorna o nome do arquivo de *checkpoint* de cluster.

  1.3 Guarda no arquivo de *checkpoint* de cápsula o nome de arquivo criado em 1.2.

  1.4 Obtém o próximo descritor.

- 2. Marca o tipo do arquivo de *checkpoint* como CHECK_CAPSULE, indicando que se trata de um arquivo de *checkpoint* de cápsula.
- 3. Retorna o nome do arquivo de *checkpoint* de cápsula.

char * desative_capsule_1 ( )

Essa função realiza a desativação de uma cápsula, sendo que a mesma na verdade faz chamada a função de *checkpoint* de cápsula para gerar um *checkpoint* da cápsula e depois chama a função de terminação de cápsula para terminar a mesma. Retorna o nome do template de *checkpoint* gerado com o tipo setado para DESATIVE.

int * delete_capsule_1( )

A função de deleção de cápsula realiza a terminação de uma cápsula, atuando da seguinte forma.

- 1. Para cada cluster na cápsula faça o seguinte:

  1.1 Pegue o descritor do cluster da lista do CPM.

  1.2 Chame a função *delete_cluster_1* do CLM associado ao cluster para realizar a terminação do cluster.

  1.3 Chame a função do portmapper para realiza a remoção registro do CLM no mesmo.

  1.4 Termine o CLM associado ao cluster e o remova da lista da cápsula.

  1.5 Pegue o próximo descritor de cluster.

- 2. Remove o gerente de cápsula.

Como descrito no capítulo 3 realizamos a eliminação do gerente de cápsula; isso é feito pela rotina de biblioteca que chama a função *terminate_cpm_1* do núcleo para realizar a remoção do CPM.

int * terminate_clm_1(Descriptor_Cluster * dcl)

Função que realiza a terminação de um gerente de cluster.

- Localiza o descritor da cluster.
- Termina o processo CLM.
- Elimina o descritor da lista de clusters na cápsula.

Se ocorre algum erro o processo é terminado e o erro é retornado. Se existir algum objeto no interior do cluster o mesmo deve ser terminado antes, via *delete_cluster_1*.

4.5.3 Interface de gerenciamento para Cluster

Um fato que deve ser lembrado aqui é que as operações de *checkpoint*, restauração e delete atuarão sobre um cluster como unidade e não apenas sobre um subconjunto de objetos básicos de engenharia do mesmo.

int * init_clm_1 (Descriptor_Cluster * dt)

A função *init_clm_1* realiza a inicialização do gerente de cluster. Atuará da seguinte forma.

- Inicializa o descritor do CLM com o descritor recebido, os ponteiro locais para NULL e os contadores para 1.

int * create_beos_1 (char * template)

Essa função realiza a criação dos objetos básicos de engenharia no cluster, da seguinte forma:

- 1. Para cada nome de template de beo contido no template de cluster recebido faça.

  1.1 Leia o template de objeto para a memória.

  1.2 Aloque um descritor para o processo a ser criado.

  1.3 Execute o processo associado ao objeto básico de engenharia descrito no template de objeto com parâmetros: Número, versão e protocolo dos gerentes associados ao processo (Núcleo, CPM e CLM), nome do diretório de swap do cluster e identificador da cápsula em que o objeto está.

  1.4 Guarde os dados do objeto no descritor.

  1.5 Anexe à lista de objetos (processos) no cluster.

  1.6 Pegue o próximo nome de template.

É importante notar que a primeira tarefa a ser executada em um processo é a rotina *main*, ligada (*linkada*) ao processo, que guarda os parâmetros recebidos e faz a setagem do tratador da rotina de *checkpoint* do processo. *main* será descrita na seção 4.6.1.

Aqui abrimos mão da portabilidade pela simplicidade adotando como identificador do objeto gerado o número do processo Unix, mesmo sendo o ideal o uso de um identificador interno à cápsula (como descrito nos capítulos 2 e 3) o que tornaria o modelo de maior portabilidade.

int * restore_beos_1(char * template)

Essa rotina realiza a restauração de um *checkpoint* de um cluster, sendo que recebe o nome do arquivo contendo o template de *checkpoint* de cluster como parâmetro, realizando a reativação de cada objeto básico de engenharia contido no cluster. Faz ainda a rearrumação dos identificadores do objeto, tais como identificação dos gerentes núcleo, CPM e CLM, e referências de interface do processo, atuando da seguinte forma.

- 1. Para cada nome de template de objeto básico de engenharia contido no template de cluster recebido faça.

  1.1 Aloque um descritor para o processo a ser criado.

  1.2 Execute o processo associado ao objeto básico de engenharia descrito no template de objeto com parâmetros: Número, versão e protocolo dos gerentes associados ao processo (Núcleo, CPM e CLM), nome do diretório de swap do cluster, identificador da cápsula em que o objeto está e nome do arquivo que originou o objeto.

  1.3 Guarde o descritor.

  1.4 Anexe à lista de objetos (processos) no cluster.

  1.5 Pegue o próximo nome de template.

char * checkpoint_cluster_1( )

A função *checkpoint_clsuter_1* realiza a criação do *checkpoint* do cluster. Atuando da seguinte forma.

- 1. Para cada objeto no cluster realiza o seguinte:

  1.1 Pegue o descritor do processo na lista do CLM .

  1.2 Chame a função *obj_checkpoint* com o identificador do processo.

  1.3 guarde no arquivo de *checkpoint* de cluster o nome de arquivo gerado.

  1.4 Pegue o próximo descritor.

- 2. Marque o tipo do arquivo de *checkpoint* como CHECK_CLUSTER, indicando que se trata de um arquivo de *checkpoint* de cluster.

- 3. Retorne o nome do arquivo de *checkpoint* de cluster.

char * desative_cluster_1 ( )

A função de desativação de cluster consiste em uma chamada, para cada objeto do cluster, da rotina de *checkpoint* via chamada a *checkpoint_cluster_1* e depois da rotina de terminação de cluster *cluster_delete_1*, retornando o template de *checkpoint* com o tipo DESATIVE setado.

int * delete_cluster_1 ( )

Essa função realiza a terminação de um cluster.

- 1. Para cada objeto (processo) no cluster faça:
  - 1.1 Pegue o descritor do objeto na lista do CLM.
  - 1.2 Elimine as IRs e canais associadas.
  - 1.3 Pegue o próximo descritor de objeto.

A terminação do CLM é feita via chamada ao CPM da função *terminate_clm_1* do CPM que termina um CLM sem atividade.

4.5.4 Operações sobre objetos básicos de engenharia

Como operações sobre objetos básicos de engenharia (processos) temos operações para terminá-los (*obj_delete*) e gerar *checkpoint* dos mesmos (*obj_checkpoint*), sendo que as mesmas são rotinas do CLM .

char * obj_checkpoint (int idbeo)

Tal função irá realizar uma chamada à função de *checkpoint* do processo, sendo isso realizado via um sinal Unix da seguinte forma.

- Envia um sinal para o identificador do BEO, *idbeo*, para o tratador de sinais de *checkpoint* (SIGUSR1).
- Verifica se o arquivo de *checkpoint* foi gerado.
- Retorna o nome do arquivo de *checkpoint* (igual o nome do diretório de swap do cluster, concatenado com *idbeo*).

int obj_delete (int idbeo)

A função de deleção de um objeto básico de engenharia realiza a terminação do processo associado.

- Envia sinal de KILL para o processo.
- Retorna.

4.6 Biblioteca para o programador

Como forma de acessar as funções de gerenciamento de uma forma mais simples, evitando do usuário preocupar-se com a semântica de RPC, elaboramos um conjunto de rotinas que funcionam como um *front-end* para as funções de gerenciamento. A tabela 4.1 contém um relacionamento das funções da biblioteca e as funções dos servidores. Descrevermos a seguir algumas características das funções, em especial de *main*, e funções sobre canais (*open_channel*, *write_channel*, *read_channel* e *close_channel*) visto que tais funções realizam um pouco mais do que um *front-end* para acesso aos servidores.

As funções, com raras exceções, funcionam como mero front-end para o servidor recebendo os dados passados como parâmetros, realizando uma conexão RPC com o servidor correspondente que provê a função e chamando a respectiva função. Retorna a seguir o resultado da chamada, escondendo assim que a implementação, que poderia ter sido realizada usando uma plataforma como ORB [42] ou mesmo outro mecanismo de IPC do Unix, foi realizada usando RPC. É realizado ainda uma filtragem dos resultados retornados pela função do servidor retornando apenas o que for de interesse do cliente. Vamos então às exceções.

A função *int_node* pode receber o nome de um arquivo contendo um template de cápsula como parâmetro e nesse caso chamar *node_init_1* para inicializar o nó e depois chamar *instantiate_capsule_1* no núcleo para realizar a criação da cápsula.

As funções *desative_capsule* e *delete_capsule* após chamarem as funções correspondentes do CPM fazem uma chamada à função *terminate_cpm_1* do núcleo para fazer a eliminação do servidor CPM que ficou inativo.

As funções *desative_cluster* e *delete_cluster*, da mesma forma, após chamarem as funções no CLM realizam uma chamada á função *terminate_clm_1* do CPM a que pertencem. A função *main* e as de canais merecem um tratamento especial sendo descritas nas subseções seguintes.

Relacionamento de Funções	
Função Biblioteca	Chama
Geral	
main	
Núcleo	
init_node	node_init_1 , [instantiate_capsule_1]
instantiate_capsule	instantiate_capsule_1
reative_capsule	reative_capsule_1
registre_ir	registre_ir_1
validate_ir	validate_ir_1
remove_ir	remove_ir_1
open_channel	bind_objects_1, [get_channel_info_1]
write_channel	[get_channel_info_1]
read_channel	[get_channel_info_1]
close_channel	remove_bind_objects
CPM	
instantiate_cluster	instantiate_cluster_1
reative_cluster	reative_cluster_1
checkpoint_capsule	capsule_checkpoint_1
desative_capsule	capsule_desative_1, terminate_cpm_1
delete_capsule	capsule_delete_1, terminate_cpm_1
CLM	
checkpoint_cluster	cluster_checkpoint_1
desative_capsule	cluster_desative_1, terminate_clm_1
delete_capsule	clsuter_delete_1, terminate_clm_1

Tabela 4.1: **Relacionamento entre Biblioteca ODP e Servidores**

4.6.1 main

A função *main* irá substituir a função main do programa escrito em linguagem C pelo programador sendo que o mesmo, no lugar da função main, deve escrever uma função *_main* que será chamada por *main*. A função *main* recebe os dados dos servidores do processo e armazena os mesmo para uso futuro pela funções de biblioteca, além do que prepara a rotina tratadora de sinais para *checkpoint* da seguinte forma.

- 1. Salva valores recebidos como parâmetros em _param .
- 2. Seta a rotina tratadora de sinais de SIGUSR1 para rotina de *checkpoint*.
- 3. Se a chamada for para reativação:
- 3.1 Chama a rotina de reativação.
- 4. Chama a função _main da aplicação.
- 5. Termina.

A rotina tratadora de sinais ao receber um sinal gera o nome do arquivo para *checkpoint* e chama a rotina que realiza o *checkpoint* propriamente dito *save_world*. Esta consiste em uma rotina da biblioteca *save_world* desenvolvida por *Bennet S. Yee* [1], sendo que a mesma está operacional apenas em ambiente Unix SunOS, necessitando-se desenvolver rotinas de *checkpoint* e restauração para outras plataformas. Além do que o serviço provido pelo pacote em uso é bem elementar. Fazemos uso do mesmo apenas para teste.

4.6.2 Canais

As funções de acesso a canais merecem uma atenção especial visto que mantêm no seu corpo parte do gerenciamento dos canais. Quando da criação de um canal podemos especificar duas semânticas: que o canal seja público (PUBLIC), ou seja quem estiver de posse do identificador do canal pode fazer uso do mesmo, ou de uso privado (PRIVATE) em que apenas o criador (e filhos) podem fazer uso do mesmo. A semântica influenciará no funcionamento das rotinas open_channel, write_channel, read_channel e close_channel. Vejamos cada caso.

- *Open_Channel.* Adotando a semântica de termos o canal público essa rotina apenas chamará *bind_objects_1* para criar a estrutura do *binding* entre cliente

[1] Carnegie Mellon University

e servidor. Caso seja privado a mesma ainda criará um socket local e realizará o *binding* entre o socket origem e destino.

- *Write_channel e Read_channel.* Caso o canal seja público a rotina irá criar um socket local, realizar o *binding* do mesmo com o do servidor e depois de fazer o *write/read* no socket irá fechar o mesmo, desfazendo o *binding* entre os sockets. Caso seja Privado apenas faz o *write/read* no canal, visto que o *binding* entre os sockets já foi todo feito na rotina Open_Channel.

- *Close_channel.* Se o canal for público irá chamar *remove_bind_1* para desfazer o canal. Se for privado também chamará *close* para fechar o socket antes de chamar *remove_bind_1*.

Na implementação realizada foi adotada a semântica de ter-se um canal público.

4.7 Interelacionamento de funções da implementação

Na tabela 4.1 descrevemos o interelacionamento das funções da biblioteca para o usuário e as dos servidores, resta-nos descrever a interelação entre as funções dos diversos servidores, a qual acreditamos já estar clara, mas que pode ser vista na figura 4.11, que mostra, a exemplo da figura 2.12 do capítulo 2, tal relacionamento.

Na figura 4.11 podemos notar uma estruturação formada pelos diversos servidores com as funções dos mesmo no seu interior, bem como a representação das funções da biblioteca na forma de um retângulo. A interação é representada por setas sólidas, quando entre servidores, e por setas pontilhadas, quando entre uma função da biblioteca e um servidor, em ambos os casos representado uma interação via RPC.

Como exemplo, podemos citar a função de instanciação de cluster que será realizada através da função *instantiate_cluster* que irá chamar a função *instantiate_cluster_1* do gerente de cápsula, que por sua vez criará um novo gerente de cluster e solicitará ao mesmo, através da função *init_clm_1*, sua inicialização e posteriormente, através da função *create_beos_1* do mesmo, a criação de seus objetos básicos de engenharia. O exemplo acima é válido para as demais funções da figura com exceção do relacionamento entre gerente de cluster e objeto básico de engenharia que se dá através de sinais.

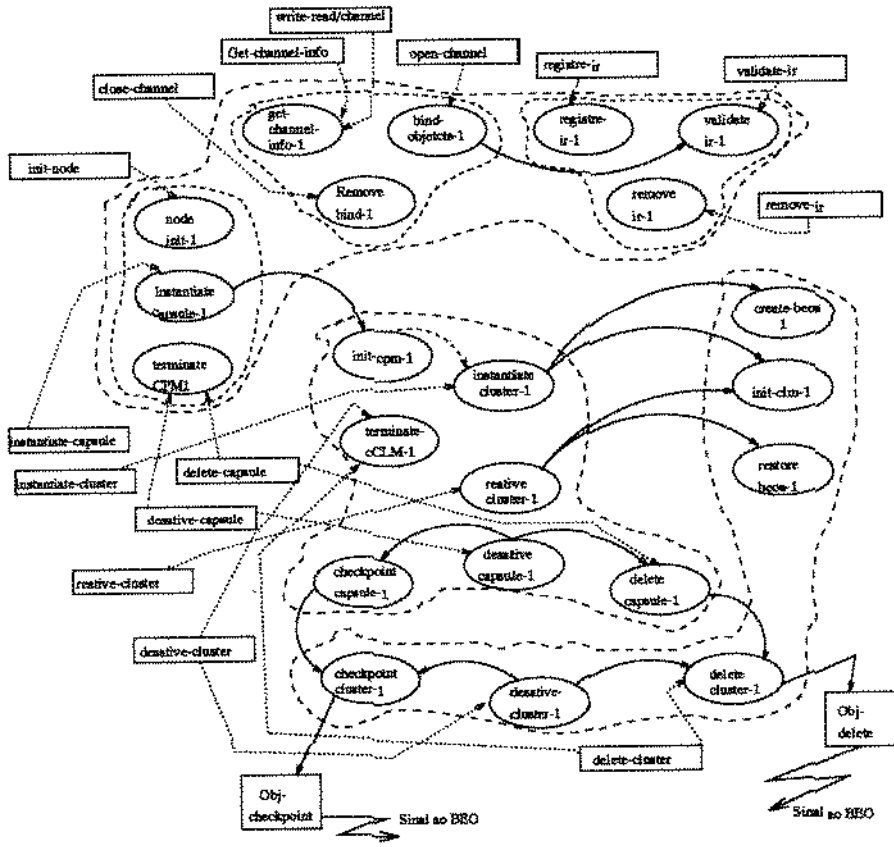

Figura 4.11: **Interelação das funções dos servidores**

4.8 Implementação com uso da plataforma Multiware

Nossa implementação, como já citamos anteriormente, foi realizada fazendo uso do SunOS devido a plataforma *Multiware* não estar ainda disponível para uso. Tentaremos agora tecer alguns comentários gerais do possível uso na plataforma *Multiware*, e conseqüentemente do ORB para a implementação.

No processo de elaboração e implementação do protótipo foi sentido diversas vêzes a ausência das demais funções definidas no RM-ODP que deveriam estar disponíveis na *Multiware*, sobretudo funções de repositório e de segurança, bem como de funções de trading e do módulo de suporte a ODP.

As funções de repositório foram necessárias para fornecer armazenamento persistente às estruturas dos gerentes (cluster, cápsula, núcleo), bem como prover armazenamento para cópias desativadas e *checkpoints* de clusters/cápsulas. Como está implementado não existe qualquer mecanismo de restrição de acesso aos servidor, sendo que para tanto (prover controle de acesso, entre outros) seria necessário o uso das funções de segurança. Funções de trading foram necessárias para prover um repositório de serviços exportados. O módulo de suporte a aplicações é necessário para prover o mapeamento e estruturação sobre o modelo de engenharia de aplicações.

O uso do ORB, como base da plataforma *Multiware*, seria interessante no sentido de prover um mecanismo para identificação única de objetos, o que facilitaria em muito o desenvolvimento e funcionalidade do módulo que trata referências de interfaces. O uso de RPC como mecanismo de interação entre os servidores, como hoje implementado, pode ser substituído por *requests* do ORB, bem como pelo uso de DII para interação dinâmica. Contudo, de maior utilidade ainda seria o uso de tais mecanismos de chamadas e de DII para realizar a implementação de canais ODP, substituindo a implementação baseada em sockets hora vigente.

As funcionalidades de gerenciamento ODP podem ser inseridas na plataforma *Multiware* como um conjunto de objetos provendo funções de estruturação de aplicações ODP, sobre o ORB. Tendo em vista a estrutura do ORB uma implementação do modelo de gerenciamento pode dar-se através de três objetos de gerenciamento (um gerente de cluster, um gerente de cápsula e um núcleo) por máquina, sendo que cada nova cápsula criada tem alocada para a mesma uma interface de gerenciamento do objeto gerente de cápsula, da mesma forma a criação de um cluster ocasionaria a alocação de uma interface do objeto gerente de cluster.

Capítulo 5

Conclusão

Procuraremos aqui discutir algumas conclusões relacionadas à realização desse trabalho, bem como dificuldades encontradas e sugestões para trabalhos futuros.

Nosso trabalho consistiu em um estudo e refinamento das funções de gerenciamento do modelo de referência RM-ODP, e em parte realizamos uma análise do mesmo. As partes 2 e 3 do RM-ODP encontram-se em estado de DIS (*Draft International Standard*), ou seja a um passo de se tornarem um padrão. A nosso ver parece um modelo estável que tende a ser aceito como padrão, visto que consegue cobrir o desenvolvimento de sistemas distribuídos em ambiente aberto de uma maneira ampla, desde a fase de especificação até a de operação, via o modelo de pontos de vista.

Em relação aos diversos níveis de estruturação propostos no modelo RM-ODP, acreditamos que objetos básicos de engenharia como unidades básicas do modelo, e o núcleo consistem em alternativas claramente viáveis, bem como o cluster como unidade para *checkpoint*, reativação, desativação. migração. Contudo a existência dos níveis de cápsula e cluster merece ser considerada, visto que as funcionalidades existentes na cápsula podem ser agregadas em parte ao núcleo e em parte ao cluster. Tomando a cápsula como unidade de recursos podemos transferir a funcionalidade de gerir os recursos ao núcleo, e considerar o cluster como a unidade de recursos. Considerando a cápsula nas suas demais funções, a criação e reativação de clusters podem ser deixadas a cargo sob responsabilidade núcleo. As funções de *checkpoint* de cápsula, desativação e terminação de cápsula podem também ser colocadas a cargo do núcleo, consistindo em um conjunto de chamadas aos gerentes de clusters. Além do que a implementação do conceito abstrato de uma cápsula, como uma máquina virtual, é difícil em muitos sistemas operacionais.

Com o uso da infraestrutura fornecida pelo RM-ODP, sobretudo pelo modelo de engenharia, ganhamos em facilidade de termos aplicações que podem ser, de forma rápida, transportadas de uma plataforma heterogênea para outra, ganhando ainda métodos para se realizar *checkpoint*, desativação, reativação e migração entre máquinas (nós). O que em uma última instância resultará no uso das transparência do modelo RM-ODP com as vantagens fornecidas pelas mesmas. O ganho dado pela estrutura formada por clusters, cápsulas, nós e objetos básicos de engenharia, consiste em fornecer uma plataforma (ou melhor uma API) sobre a qual o modelo computacional do RM-ODP pode ser fornecido em um mundo real, levando ao fornecimento, com as demais funções do modelo de engenharia, das transparências previstas no RM-ODP. Fornecendo uma forma para se estruturar aplicações distribuídas em um ambiente heterogêneo.

O Modelo de engenharia do RM-ODP, em especial, encontra-se em evolução ainda, sendo que as funções de gerenciamento por serem a base do modelo acabam então em evolução constante. Foram dificuldades na realização do trabalho, primeiro a ausência de literatura sobre gerenciamento no RM-ODP no início do trabalho, e a constante evolução dos documentos de padronização, levando o texto a ser re-escrito diversas vêzes a fim de acomodá-lo às novas versões dos documentos de padronização visto que algumas vêzes as mudanças no mesmo não foram sutis.

Podemos citar ainda como fator determinante na demora na conclusão do trabalho a plataforma *Multiware* estar em desenvolvimento, terminando a fase de especificação e começando as implementações, o que nos levou em uma determinada instância a fazer uso do sistema operacional Unix como plataforma de desenvolvimento; levando a adotar uma série de restrições em relação à implementação.

Uma dificuldade que ainda pode ser levantada consiste em que não pudemos fazer uso de quaisquer funções propostas no modelo RM-ODP, tais como funções de segurança ou de repositório, visto que as mesmas ainda estão em fase de especificação na plataforma *Multiware*, o que nos levou a impor mais restrições ao protótipo. Optamos por impor limitações a fazer uso de características específicas do sistema Unix a fim de ter um protótipo facilmente migrável para outra plataforma quando estiver disponível.

Um fato que pode ser destacado consiste, levando em conta a implementação realizada, em que algumas vêzes existe a necessidade de interação maior com o sistema operacional em uso. Um exemplo que pode ser citado são as rotinas para dar suporte a realização de *checkpoint* e reativação de objetos (no caso da implementação processos) que necessitam ter acesso a um conjunto de informações disponível apenas ao nível de núcleo de alguns sistemas operacionais. Da mesma

forma a implementação da cápsula como uma máquina virtual necessita ter acesso ao núcleo do sistema operacional em uso. Uma forma de minimizar tais problemas, e ao mesmo tempo otimizar o desempenho, seria fazer uso de um sistema operacional baseado em microkernel, como *chorus* [51], na implementação.

Como contribuições esperamos ter apresentados o modelo de engenharia do RM-ODP de forma mais clara. Discutindo o modelo de engenharia e apresentando o mesmo sob a ótica de sistemas operacionais. Em relação as funções de gerenciamento em si, apresentamos um estudo detalhado das funções, considerando a estruturação das mesmas no que se relaciona a seus componentes, bem como estruturas de dados para as mesma. Realizamos ainda a implementação das funções de gerenciamento na forma de um protótipo. O mesmo destina-se a validar algumas das idéias levantadas, algumas das vezes com restrições, e a servir de base para uma futura implementação do módulo de gerenciamento ODP previsto na plataforma *Multiware*.

A estrutura do RM-ODP apresenta-se como válida para o desenvolvimento de um ambiente de suporte a ODP, valendo ressaltar a necessidade de um estudo da real necessidade dos dois níveis de estruturação, em cápsulas e clusters, o que em uma implementação pode levar a uma sobrecarga que poderia ser minimizada.

Uma consideração que vale levantar aqui consiste em que a estruturação proposta na figura 4.1 não consiste em má idéia visto que a mesma apresentou bons resultados enquanto no simulador e seria, a princípio, a escolha para uma implementação do modelo de gerenciamento tomando como base para plataforma de suporte o ORB [42].

Como extensões possíveis ao trabalho imaginamos basicamente a integração com a plataforma *Multiware*, levando algumas vêzes à necessidade de alterações na implementação realizada. Podemos citar como extensões:

- A necessidade da implementação do gerente de aplicação (plataforma *Multiware*) a fim de realizar um estudo maior do mapeamento do modelo computacional para o modelo de engenharia.
- Realizar a migração do protótipo para outra plataforma, como ORB.
- Realizar um estudo maior para permitir a migração entre plataformas heterogêneas, via mecanismos de *checkpoint* e reativação.
- Realizar a integração do modelo proposto com o restante da especificação da plataforma *Multiware*.
- Realizar um estudo sobre a necessidade de termos o nível de cápsula na implementação, ou apenas como um nível conceitual.

Bibliografia

[1] ISO/IEC JTC1 SC21/WG7 N 885, ITU-T X.901 — ISO/IEC 10746-1 *ODP Reference Model part1 : Overview and Guide to use*, November 1993.

[2] ISO 10746-2 DIS 10746-2, ITU-T Draft X.902. *Basic Reference Model of ODP - Part 2: Descritive Model*, February 1994.

[3] ISO/IEC JTC1 SC21 - ISO/IEC DIS 10746-3-1, X.903. *Basic Reference Model of ODP - Part 3: Prescriptive Model*, February 1994.

[4] ISO/IEC SC21 N7056, X.904. *Basic Reference Model of ODP - Part 4: Architectural Semantics*, August 1994.

[5] ISO/IEC JTC1 SC21 - ISO/IEC CD 10746-3 N7525, X.903. *Basic Reference Model of ODP - Part 3: Prescriptive Model*, April 1993.

[6] ISO/IEC/JTC1/SC21/WG7 N671, *Summary of Australian Concerns about part3* . Australian National Body, September 1992.

[7] ISO/IEC JTC 1/SC21/N7047, *Working Document on topic 9.1 - Trader ODP*, June 1993.

[8] Architecture Projects Management Limited,*ANSA: An Engineer's introduction to the Architecture*, release TR-03-02, November 1989.

[9] Architecture Projects Management Limited, *ANSA the Architecture - Phase Three*, Issue 1, june 1991.

[10] Armand, Francois, at all, *Revolution 89 or "Distributed Unix brings it back to its original virtues"*, CS-TR-89-36, chorus systems, august 1989.

[11] Bach, Maurice, J. , *The Design of the Unix Operating System*, Prentice-Hall, 1986.

[12] Bearman, Mirion J., *ODP Trader*, Proceedings of the ICODP - September 1993, pages 19-23.

[13] Beitz, A.D.; King, P.W., and Raymond, K.A., *Is DCE a support Environment for ODP ?*, Proceedings of the ICODP - September 1993, pages 188-202.

[14] Birrell, Andrew; Nelson, Bruce Jay, *Implementing Remote Procedure Calls"*, ACM TOCS, Vol2, N1, February 1984, pages 39-59.

[15] Blair, Gordon S.; Lea, Rodger, *The Impact of Distribution on the object oriented approach to software development*, Chorus Systems, CS-TR-92-40, 2 may 1992.

[16] Blair, Gordon S.; Rodden, Tom, *The Impact of CSCW on Open Distributed Processing*, IFIP 1992, pages 143-153.

[17] Blattler, D.; Hogrefe, D., *The 5 ODP viewpoints on the schindler group*, IFIP 1992, pages 357-364.

[18] Bosco, Pier Giorgio; Martini, Giovanni, and Moiso, Corrado, *A distributed Object oriented platform based on DCE and C++*, Proceedings of the ICODP - September 1993, pages 176-187.

[19] Chabernaud, Christian and Vilain, Bernard, *Telecommunication Services and Distributed Applications*, IEEE Network Computer, November 1990, pages 10-13.

[20] Chin, Rodger, and Chanson, Samuel T., *Distributed Object-based programming systems*, ACM Computing Surveys, Vol 23, No 1, March 1991, Pages 91-124.

[21] Dreo, G.; Heverhagen, U.; Leischmer, M., *Using the OSI management information model for ODP*, IFIP 1992, pages 203-214.

[22] Drummond, Rogério and Cianni, Cassius Di, *OMNI - Sistema de suporte a aplicações distribuídas*, Relatório Interno, Projeto A_HAND, DCC-UNICAMP, 1992.

[23] Faroqui, Kazi and Logrippo, Luigi,*Viewpoint Transformation*, Proceedings of the ICODP - September 1993, pages 352-362.

[24] Fauth, Dr. Dietmar; Gossels, Jonathan; at all,*Distributed Computing Environment Evaluation Team*, Open Software Foundation , May 1990.

[25] Fedaoui, Linda; Twbi, Wassim, and Horlait, Eric, *Distributed Multimedia Systems Quality of service in ODP Framework of abstraction ? A first study*, Proceedings of the ICODP - September 1993, pages 235-244.

[26] Gien, Michel, *From Operating Systems to Cooperative Operating Environments*, Chorus Systems, CS-TR-92-82, October 1992.

[27] Hebert, Andrew, *The Challenge of ODP*, IFIP 1992, pages 15-27.

[28] HP Company, Sun Microsystems, inc : *The Object Management Group, Object Request Broker*, RFP joint response, OMG Document Number 91.1.4.8, 1991.

[29] HP Company, Sun Microsystems, inc : *The Object Management Group, Class Definition Language Specification*, RFP joint response, OMG Document Number 91.1.4.9, 1991.

[30] HP company, Sun Microsystems, inc : *Distributed Object Management Facility Core especification*, OMG Document Number 91.1.4.10,1991.

[31] Kato, Kazuhiko; Inohara, Shigekazu; Narita, Atsunobu; Chira, Shigeru and Masuda, Takashi, *Design of the XERO open distributed operating systems*, Technical report 92-1, Departament of information science, faculty of science, University of Tokio, Japan, February 1992, pages 1-29.

[32] Kerninghan, B.W. and Ritchie, D.M., *The C Language*, Prentice-Hall, 1988.

[33] Lima Jr., Luiz Augusto de Paula, *Um modelo para a implementação de federação de traders*, Dissertação de mestrado apresentada no DCC-Unicamp, 1994.

[34] Linington, P. F., *Introduction to the Open Distributed Processing Basic Reference Model*, IFIP 1992, pages 3-13.

[35] Loyolla, W.P.D.C; Madeira, E.R.M; Mendes, M.J.; Cardozo, E. and Magalhães, M.F., *Multiware Plataform: an Open Distributed Environment for Multimedia Cooperative Applications*, COMPSAC'94 - IEEE Computer Software & Application Conference, Taipei, taiwan, November 1994.

[36] Mello, Alexandre Magalhães Vaz de; Lento, Luiz Otávio Botelho and Euzébio, Marcos José Cândido, *Suporte a Aplicações Distribuídas : Descrição de Algumas Plataformas*, Relatório Interno, grupo de Redes-ODP DCC/DCA - Unicamp, November 1993.

[37] Mendes, M.J.; Loyolla, W. and Madeira, E. R. M. , *Demos: A distributed Decision making open support system*, 4 th Workshop on future trends of distributed computing systems, IEEE Computer Society, Lisbon September 1993, pages 208-214.

[38] Mendes, M.J, Madeira, E.R.M, *Plataforma Multiware: Projeto e desenvolvimento da Camada Middleware*, Anais do XII SBRC, May 1994, Pages 93-109.

[39] Meyer, B. and Popien, C., *Object configuration by ODP traders*, Proceedings of the ICODP - September 1993, pages 425-430.

[40] Nance, Barry, *Interoperability today*, Byte, November 1991, pages 187-196.

[41] Nicol, R. John; Wilkes, C. Thomas, and Manola, Frank A., *Object Orientation in Heterogeneous Distributed Computing Systems*, IEEE Computer, June 1993, Pages 57-67.

[42] Object Management Group, *The Commom Object Request Broker: Architecture and Specification*, OMG Document Number 91.12.1, Revision 1.1, 10 December 1991.

[43] Object Management Group, *OMG Object Services request for proposal 2*, OMG TC Document 93-6-1, June 93, pages 1-70.

[44] Object Management Group,*Object Services Architecture, Revision 6.0*, OMG Document 92.8.4, August 1992, pages 1-75.

[45] Object Management Group, *Object Request Broker Architecture*, OMG TC Document 93.7.2, July 1993, Pages 1-38.

[46] Open Software Foundation, *Distributed Computing Environment.*

[47] Open Software Foundation, *Directory Services for a Distributed Computing Environment*, September 1990.

[48] Open Software Foundation, *File Systems in a Distributed Computing Environment*, September 1990.

[49] Open Software Foundation, *Remote Procedure Call in a Distributed Computing Environment*, october 1990

[50] Raymond, K. , *Reference Model of Open Distributed Processing a Tutorial*, Proceedings of the ICODP - September 1993, pages 3-14.

[51] Rozier, M. at all, *Overview of CHORUS distributed operating systems*, CS-TR-90-25, chorus systems, april 1990.

[52] Schurmann, Gerd and Holzmann-Kaiser, Uwe, *Distributed Multimedia Information Handling and Processing*, IEEE Network Computer, November 1990, pages 23-31.

[53] Singhal, Mukesk and Casavant, Thomas L., *Distributed Computing Systems*, Computer, Vol 24, No 8, August 1991, pages 12-15.

[54] Slonim, Jacob, at all, *Does Midware provide an adequate distributed application environment ?*, Proceedings of the ICODP - September 1993, pages 34-46.

[55] Stevens, W. Richard, *Unix Network Programming*, Prentice-Hall, 1990.

[56] Sun microsystems, *Network Programming Guide*, Part Number 800-3850-10, Revision A of 27 March, 1990.

[57] Sun microsystems, *SunOS Reference Manual - Volume 2*, Part Number 800-3827-10, Revision A of 27 March, 1990.

[58] Tanembaum, Andrew S., and Renesse, Robbert Van, *Distributed Operating Systems*, ACM Computing Surveys, Vol 17, No 4, December 1985, pages 419-470.

[59] Tanenbaum, Andrew S., *Computer Networks*, 2nd , Prentice-hall, 1989.

[60] Tanembaum, Andrew S., *Modern Operating Systems*, Prentice-hall, 1992.

[61] Telecom Canada, *Comparison of the OMG and ISO/CCITT object Models* - The report of The joint NM Forum / OMG Taskforce on Object modelling, April 1993, pages 1-24.

[62] The Open Systems Newsletter, *ANSA: Object Pioneers*, technology Approisals ltd, Vol 7 Issue 2, February 1993, pages 1-10.

[63] Tschammer, V.; Eckert, K. P.; Hall,J.; Schurman, G. and Strick , *OAI - Concepts for Open Systems Cooperation*, LNCS 433, pages 174-191, 1989.

[64] Tschammer, V., *"CODE, Cooperating Open Systems, Open Distributed Processing, Distributed Computing Environment, Experiments and Projects"*, Curso intensivo da UNICAMP-FEE-DCA, Novembro de 1992.

[65] Tschammer, V.; Mendes, Manuel de Jesus; Souza, Wanderley Lopes de; Madeira , Edmundo R. Mauro; e Loyolla, Waldomiro, *Processamento Distribuído Aberto e o Modelo RM-ODP/ISO*, Anais do XI SBRC, May 1993.

[66] Vogt, F. and Andrae, C., *Middleware for Distributed Applications Support : ODP and/or CORBA*, Proceedings of the ICODP - September 1993, pages 405-410.