Universidade Estadual de Campinas

Faculdade de Engenharia Elétrica e de Computação

Projeto e Implementação de um Ambiente para Processamento Distribuído Baseado em TINA

Alexandre de Souza Pinto
Orientador: **Prof. Dr. Eleri Cardozo**

Dissertação de Mestrado

Comissão Julgadora:
Eleri Cardozo - FEEC/UNICAMP - Presidente
Edmundo Roberto Mauro Madeira - IC/UNICAMP
Alice Maria B. H. Tokarnia - FEEC/UNICAMP
José Raimundo de Oliveira - FEEC/UNICAMP - Suplente

Campinas, SP - Brasil

Fevereiro de 1999

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

P658p

Pinto, Alexandre de Souza
Projeto e implementação de um ambiente para processamento distribuído baseado em TINA. / Alexandre de Souza Pinto.--Campinas, SP: [s.n.], 1999.

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

1. Processamento eletrônico de dados – Processamento distribuído. 2. Sistemas multimídia. 3. Sistemas de telecomunicação. I. Cardozo, Eleri. II. Universidade Estadual de Campinas. Faculdade de Engenharia Elétrica e de Computação. III. Título.

Resumo

Pinto, A. S., Projeto e Implementação de um Ambiente para Processamento Distribuído Baseado em TINA. Campinas: DCA/FEEC/UNICAMP, 1999. (Dissertação de Mestrado)

A arquitetura TINA (*Telecommunications Information Networking Architecture*) provê conceitos, modelos e mecanismos para o projeto, implementação e gerência de serviços de telecomunicações. Este trabalho trata da construção de um componente central da arquitetura TINA, o DPE (*Distributed Processing Environment*).

O DPE-TINA é uma infra-estrutura de suporte à distribuição dos componentes das aplicações de telecomunicações que segue a arquitetura TINA. Para sua implementação, utilizou-se a plataforma CORBA (*Common Object Request Broker Architecture*) e um banco de dados orientado a objetos. O DPE desenvolvido oferece duas funções básicas às aplicações aderentes à arquitetura TINA: serviço de ciclo de vida e de *stream*.

O serviço de ciclo de vida permite a distribuição e a gerência de objetos em um sistema heterogêneo, enquanto o serviço de *stream* proporciona suporte à comunicação via fluxos de mídia contínua (áudio e vídeo).

Nesta dissertação, é apresentada, também, uma aplicação de teste que faz uso das funções da infra-estrutura implementada.

Palavras-chave: TINA, DPE, CORBA, Sistemas de Computação Móvel, Sistemas Multimídia

Abstract

Pinto, A. S., Design and Implementation of a TINA-based Distributed Processing Environment. Campinas: DCA/FEEC/UNICAMP, 1999. (Masther's Thesis)

TINA (Telecommunications Information Networking Architecture) provides concepts, models and mechanisms for the design, implementation and management of telecommunication services. This work describes the development of a fundamental component of TINA Architecture, the DPE (Distributed Processing Environment).

TINA DPE is an infrastructure necessary to distribute the telecommunication applications components. A CORBA (Common Object Request Broker Architecture) platform and an object-oriented database are employed to build this infrastructure. The implemented DPE offers two basic facilities to the TINA applications: life-cycle and stream facilities.

Life-cycle facilities allow distributed objects to be deployed and managed in a heterogeneous system, while stream facilities allow application objects to exchange continuous media flows (audio and video).

Finally, a prototype application, which makes use of the implemented DPE facilities, is presented.

Keywords: TINA, DPE, CORBA, Mobile Computing Systems, Multimedia Systems

Para Raul e Nilsa

Agradecimentos

A Deus.

À minha família, especialmente aos meus pais e irmãos. Vocês representam um exemplo de amor, dedicação e dignidade a ser seguido.

À Corina, por todo carinho, compreensão e incentivo proporcionados nos momentos mais difíceis.

Ao professor Eleri, pela paciência e firme orientação durante a execução do trabalho.

Ao Faina, pelo valioso auxílio fornecido para a confecção das figuras presentes neste documento.

Ao Affonso, André, Eduardo e Rodrigo, pela produtiva e, acima de tudo, alegre convivência.

À agência CAPES, pelo suporte financeiro.

A todos, ofereço minha profunda gratidão.

"Aprendemos no céu o estilo da disposição, e também o das palavras. As estrelas são muito distintas e muito claras. Assim há de ser o estilo (...); muito distinto e muito claro. E nem por isso temais que pareça o estilo baixo; as estrelas são muito distintas e muito claras, e altíssimas. O estilo pode ser muito claro e muito alto; tão claro que o entendam os que não sabem e tão alto que tenham muito o que entender os que sabem."

Padre Antônio Vieira
(1608-1697)

Sumário

Lista de Figuras

Abreviaturas

AAL5: *ATM Adaption Layer* 5

ACE: *Application Construction Environment*

ACID: *Atomicity, Consistency, Isolation, Durability*

ACTS: *Advanced Communications Technologies and Services*

ANSA: *Advanced Networked Systems Architecture*

API: *Application Programming Interface*

ATM: *Asynchronous Transfer Mode*

B-ISDN: *Broadband Integrated Service Digital Network*

CAD: *Computer-aided Design*

CDR: *Common Data Representation*

CORBA: *Common Object Request Broker Architecture*

DAVIC: *Digital-Audio Visual Council*

DII: *Dynamic Invocation Interface*

DOLMEN: *Service Machine Development for an Open Long-term Mobile and Fixed Network Environment*

DPE: *Distributed Processing Environment*

DSI: *Dynamic Skeleton Interface*

eCO: *engineering Computational Object*

GDMO/GRM: *Guidelines for the Definition of Managed Objects/General Relationship Model*

GIOP: *General Inter-ORB Protocol*

IDL: *Interface Definition Language*

IIOP: *Internet Inter-ORB Protocol*

IN: *Intelligent Network*

INA: *Information Networking Architecture*

IP: *Internet Protocol*

ITU: *International Telecommunications Union*

JMF: *Java Media Framework*

kTN: *kernel Transport Network*

Mbone: *Multicast Backbone*

N-ISDN: *Narrowband Integrated Service Digital Network*

ODBMS: *Object-oriented Database Management System*

ODL: *Object Definition Language*

ODP: *Open Distributed Processing*

OMA: *Object Management Architecture*

OMG: *Object Management Group*

OMT: *Object Modeling Technique*

ORB: *Object Request Broker*

OSI: *Open Systems Interconnection*

POA: *Portable Object Adapter*

PSTN: *Public Switched Telephone Network*

QoS: *Quality of Service*

RM-ODP: *Reference Model for Open Distributed Processing*

ROSA: *RACE Open Service Architecture*

RTP: *Real Time Protocol*

SERENITE: *Services du Reseau Numerique Intelligent des Telecommunications*

SFP: *Simple Flow Protocol*

SQL: *Structured Query Language*

TCP: *Transmission Control Protocol*

TINA: *Telecommunications Information Networking Architecture*

TINA-C: *Telecommunications Information Networking Architecture Consortium*

TMN: *Telecommunications Management Network*

UDP: *User Datagram Protocol*

UML: *Unified Modeling Language*

VAT: *Visual Audio Tool*

VITAL: *Validation of Integrated Telecommunication Architectures for the Long term*

Capítulo 1

Introdução

Nas duas últimas décadas, o mercado de telecomunicações tornou-se um dos mais cobiçados negócios do planeta devido ao imenso aumento da demanda por este tipo de serviço. Para muitos, este crescimento deve ainda continuar por um bom período, já que o fenômeno da globalização, cada vez mais presente na sociedade, acaba instituindo necessidades por novos serviços. Atualmente, por exemplo, grandes usuários corporativos buscam a interligação de suas unidades de negócio por meio de serviços integrados de telecomunicações oferecidos por empresas de atuação global [1].

Além disso, usuários residenciais exigem cada vez mais serviços multimídia interativos, tais como vídeo sob demanda e compra a distância [2]. Lucky [3] afirma que os atuais e futuros serviços de comunicações realizarão sonhos antigos da humanidade: "estar em algum lugar onde não estamos" (teleconferência, telepresença); "estar em algum instante de tempo diferente do atual" (secretária eletrônica, *pager*); "ser alguém diferente de nós" (realidade virtual). Em última análise, ainda segundo Lucky, os novos serviços de telecomunicações possuem grande potencial para melhorar a vida em sociedade.

Espera-se, portanto, que a demanda reprimida por novos serviços de telecomunicações e a progressiva convergência entre os sistemas de telecomunicações e de computação [4, 5] proporcionem um horizonte de oportunidades quase ilimitado para operadoras, provedores de serviços e fabricantes que atuam no mercado.

Entretanto, este mercado tão promissor pode ser vítima do seu próprio sucesso. Muitas oportunidades estão se tornando, na verdade, ameaças. Freqüentemente, a indústria de telecomunicações não consegue oferecer serviços que atendam as necessidades cada vez maiores dos usuários. Isto ocorre devido ao fato da infra-estrutura atual de telecomunicações ter sido projetada para transmitir dados e voz e não para proporcionar serviços mais sofisticados. Assim, a introdução de novos serviços e funções em grande escala converteu-se em um processo complexo, caro e lento. Além de não conseguir acompanhar o ritmo imposto pelas exigências do consumidor, esta situação acaba desencorajando novos empreendedores, já que o custo de entrada no mercado também é cada vez maior.

Para enfrentar estes desafios, é necessária uma nova arquitetura para sistemas de telecomunicações que aproveite os últimos avanços da tecnologia de computação, funcione com os sistemas existentes e utilize padrões abertos.

Com este cenário em vista, em 1992, cerca de 40 organizações de vários países, incluindo ope-

radoras de telecomunicações, fabricantes de equipamentos de telecomunicações e de computação, formaram o consórcio TINA-C (*Telecommunications Information Networking Architecture Consortium*).

1.1 Consórcio TINA

O TINA-C (*Telecommunications Information Networking Architecture Consortium*)[1] [6] é um consórcio internacional composto de grandes companhias de telecomunicações e de computação. O objetivo do consórcio é definir e validar uma arquitetura aberta, denominada arquitetura TINA, adequada para o oferecimento dos futuros serviços de telecomunicações.

Existem dois padrões estabelecidos, amplamente aceitos pela indústria, cujos propósitos são similares aos do consórcio TINA: Rede Inteligente (*Intelligent Network*, IN) [7] e Rede de Gerência de Telecomunicações (*Telecommunications Management Network*, TMN) [8].

Rede Inteligente é uma plataforma utilizada para o oferecimento de serviços avançados de telefonia sobre as atuais redes de faixa estreita, como PSTN (*Public Switched Telephone Network*) e N-ISDN (*Narrowband Integrated Service Digital Network*). Os serviços de uma Rede Inteligente possuem características centralizadas e orientadas a funções. A Rede de Gerência de Telecomunicações é uma rede de dados conceitualmente separada da rede de transporte e que administra seus serviços e recursos.

As especificações IN e TMN podem ser consideradas complementares e proporcionam a infra-estrutura para os atuais serviços de telecomunicações [9]. No entanto, o aumento da demanda por serviços mais sofisticados e o surgimento das redes de faixa larga (*Broadband Integrated Service Digital Network*, B-ISDN) revelaram a necessidade do aperfeiçoamento e evolução destas plataformas.

Portanto, a intenção do consórcio TINA é aproveitar os recentes avanços ocorridos na área de computação distribuída e de metodologias de análise e projeto de *software* orientadas a objetos, a fim de desenvolver uma infra-estrutura mais adequada ao oferecimento de serviços multimídia sobre redes de faixa larga. Desta forma, espera-se proporcionar mais rapidamente (e de maneira mais econômica) serviços flexíveis e que satisfaçam às necessidades do consumidor. Caso alcance êxito, a arquitetura TINA deve, a longo prazo, substituir gradualmente os atuais ambientes IN/TMN [10].

1.1.1 Estrutura

Em janeiro de 1993, o consórcio TINA constituiu um grupo de pesquisadores denominado *Core Team*. Esta equipe era formada por cerca de 40 engenheiros, com dedicação exclusiva, pertencentes a diversos membros do consórcio. Estabeleceu-se como sede do *Core Team* a cidade de Red Bank em New Jersey, EUA[2].

A tarefa do *Core Team* era desenvolver todas as especificações relacionadas com a arquitetura TINA no período de cinco anos. Ou seja, ao final de 1997, deveria ser concluído o conjunto de padrões que define a arquitetura TINA.

Entretanto, a fim de refinar e completar o desenvolvimento da arquitetura, o TINA-C decidiu estender este prazo por mais três anos. Houve também uma mudança na estrutura do consórcio: o

[1] http://www.tinac.com

[2] O *Core Team* utilizava instalações oferecidas pela Bellcore.

Core Team foi dissolvido e a tarefa de desenvolvimento, refinamento e validação das especificações foi atribuída a grupos de trabalho.

Os grupos de trabalho são criados de acordo com o interesse demonstrado pelos membros do consórcio em determinados assuntos. Atualmente, existem cinco grupos de trabalho:

- **Arquitetura DPE**: responsável pela nova especificação do DPE (*Distributed Processing Environment*), que provê suporte aos serviços de telecomunicações;
- **Mobilidade de Próxima Geração**: responsável pela finalização do trabalho sobre mobilidade na arquitetura TINA;
- **Gerência de Serviços**: responsável pelo refinamento da especificação de gerência de serviços;
- **Pontos de Referência da Arquitetura de Serviços**: responsável pela especificação dos pontos de referência relativos aos serviços de telecomunicações;
- **Estrutura de Endereçamento e de Nomes**: responsável pelo desenvolvimento do padrão sobre a estrutura de endereçamento e de nomes.

Existem, ainda, grupos de interesse especial que avaliam novos rumos para o trabalho do consórcio. Ao contrário dos grupos de trabalho, não é necessário ser membro do TINA-C para participar dos grupos de interesse especial.

Na fase atual do consórcio, o trabalho concentra-se naquelas áreas de negócio onde a utilização da arquitetura TINA possui maior chance de gerar produtos a curto prazo. As seguintes aplicações foram consideradas como fortes candidatas a adotarem a arquitetura TINA: gerência de redes e serviços heterogêneos, mobilidade de próxima geração e Internet/intranet de alta qualidade (serviços multimídia).

Este "novo" consórcio TINA pretende racionalizar e simplificar a arquitetura para suportar os domínios de negócio citados no parágrafo anterior. Com o desenvolvimento de produtos TINA, espera-se uma aceleração no processo de especificação, pois existirá uma maior colaboração dos implementadores da arquitetura. Desta forma, acredita-se que a adoção da arquitetura pela indústria, baseada em casos reais de negócio, ocorra antes do fim previsto para esta fase (dezembro de 2000).

1.1.2 Projetos de Validação

Os projetos de validação, ou projetos auxiliares, são esforços realizados por membros do consórcio cujo objetivo é demonstrar que a arquitetura TINA é factível. Além disto, estes projetos desempenham um papel fundamental no processo de validação e revisão das especificações TINA.

Entre os principais projetos de validação, podem ser citados:

- **ReTINA** [11]: projeto iniciado em 1995 dentro do programa europeu ACTS (*Advanced Communications Technologies and Services*). O objetivo do projeto é desenvolver e validar especificações de um ambiente para processamento distribuído adequado para o suporte a serviços multimídia interativos;

- **VITAL**[3] [12, 13]: projeto que também faz parte do programa ACTS. Seu objetivo principal é desenvolver, demonstrar e validar um conjunto de componentes baseados em TINA que possa ser utilizado na implementação de serviços multimídia móveis.

- **DOLMEN**[4] [14]: outro projeto patrocinado pelo ACTS, o DOLMEN tem como meta desenvolver e validar uma extensão da arquitetura TINA que proporcione suporte à mobilidade de usuários e de terminais.

Existem, ainda, demonstrações de protótipos realizadas por membros do consórcio. Estes testes de campo, denominados TINA *Trials* [15, 16], também servem para validar e aperfeiçoar os padrões TINA.

Outros projetos auxiliares estão relacionados com a construção de ferramentas de especificação e desenvolvimento de aplicações TINA, como o ambiente ACE (*Application Construction Environment*) [17, 18].

1.1.3 Colaboração com Outras Organizações

Para construir sua arquitetura, o consórcio TINA baseou-se em padrões relevantes de computação distribuída e de telecomunicações. As principais especificações consideradas foram:

- Modelo de Referência para Processamento Distribuído Aberto (*Reference Model for Open Distributed Processing*, RM-ODP) da ITU (*International Telecommunications Union*);

- Arquitetura de Gerência de Objetos (*Object Management Architecture*, OMA) do OMG (*Object Management Group*);

- Rede de Gerência de Telecomunicações (*Telecommunications Management Network*, TMN) da ITU.

Vários outros trabalhos também influenciaram a arquitetura TINA, entre eles: ANSA (*Advanced Networked Systems Architecture*) [19], IN (*Intelligent Network*) [7], INA (*Information Networking Architecture*) [20], ROSA (*RACE Open Service Architecture*) [21] e SERENITE (*Services du Reseau Numerique Intelligent des Telecommunications*) [22].

Além de utilizar padrões abertos já estabelecidos, o consórcio TINA procura colaborar com os principais organismos de padronização, a fim de garantir a coerência entre a arquitetura TINA e outras especificações. O TINA-C possui cooperação mais ativa com o OMG, ITU, ATM (*Asynchronous Transfer Mode*) Forum e DAVIC (*Digital-Audio Visual Council*).

[3] *Validation of Integrated Telecommunication Architectures for the Long term*
[4] *Service Machine Development for an Open Long-term Mobile and Fixed Network Environment*

1.2 Contribuição

Neste trabalho, aborda-se o projeto e a implementação de um componente fundamental da arquitetura TINA, o Ambiente para Processamento Distribuído (*Distributed Processing Environment*, DPE).

A função básica do DPE é permitir que os componentes dos serviços de telecomunicações (aplicações TINA) sejam distribuídos de maneira transparente sobre um ambiente de *hardware/software* heterogêneo.

Além de aderir à especificação TINA, o DPE implementado baseia-se em padrões CORBA (*Common Object Request Broker Architecture*) e RM-ODP (*Reference Model for Open Distributed Processing*) e tem como destaque o suporte à migração de objetos e à comunicação via fluxos multimídia.

1.3 Conteúdo

Este documento está organizado da seguinte forma. O capítulo 2 apresenta uma visão geral da arquitetura TINA, incluindo suas principais subdivisões. O capítulo 3 descreve o ambiente DPE e mostra os requisitos que devem ser seguidos por uma implementação deste componente da arquitetura TINA. No capítulo 4, são discutidos aspectos de projeto e implementação de uma plataforma DPE, utilizando uma infra-estrutura baseada em CORBA e em um banco de dados orientado a objetos. O capítulo 5 trata de alguns detalhes relacionados com a implementação realizada e descreve uma aplicação de teste que faz uso das funções do DPE. Finalmente, o capítulo 6 apresenta as conclusões do trabalho e suas possíveis extensões.

O apêndice A contém as definições de todas as interfaces desenvolvidas neste trabalho. O apêndice B lista as interfaces da especificação de controle e gerência de *streams* de áudio e vídeo proposta pelo consórcio OMG.

Capítulo 2

Visão Geral da Arquitetura TINA

TINA [23, 24] é uma arquitetura de *software* para sistemas de telecomunicações. Um sistema de telecomunicações consiste de serviços[1] e de redes de transporte[2] oferecidos a diferentes participantes (clientes, usuários, provedores de serviço e operadoras de rede) e que pode ser dividido em vários domínios administrativos (figura 2.1). Sistemas TINA devem ser capazes de operar, nos níveis de serviço ou de rede, com outros sistemas (não-TINA).

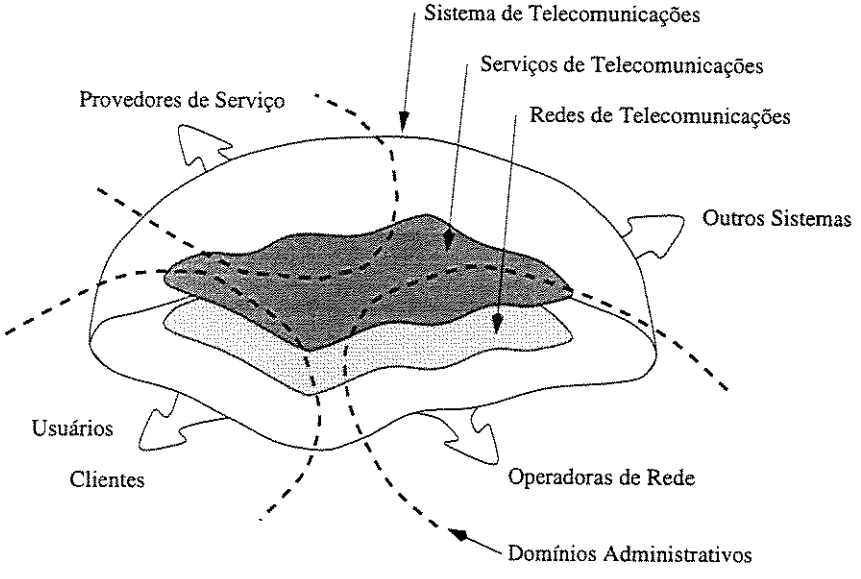

Figura 2.1: Sistema de telecomunicações [23]

Incorporando princípios de orientação a objetos e de computação distribuída, a arquitetura TINA pretende melhorar a interoperabilidade e o reaproveitamento de *software* e de especificações e permitir uma distribuição flexível dos componentes de serviço pela rede.

[1]Serviços de telecomunicações incluem: serviços baseados em voz, serviços multimídia interativos, serviços de informação e serviços de gerência.

[2]Uma rede de transporte é composta de equipamentos de transmissão e de comutação.

2.1 Características Básicas

A estrutura básica de um ambiente TINA é apresentada na figura 2.2. A camada superior é composta de objetos, distribuídos por vários elementos da rede, que proporcionam as funções dos serviços de telecomunicações. O ambiente para processamento distribuído (*Distributed Processing Environment*, DPE) oferece o suporte às aplicações TINA. As funções básicas do DPE são proporcionar uma visão independente de tecnologia das camadas inferiores (potencialmente heterogêneas), esconder das aplicações a natureza distribuída do sistema e fornecer suporte à localização e interação remota dos objetos de aplicação.

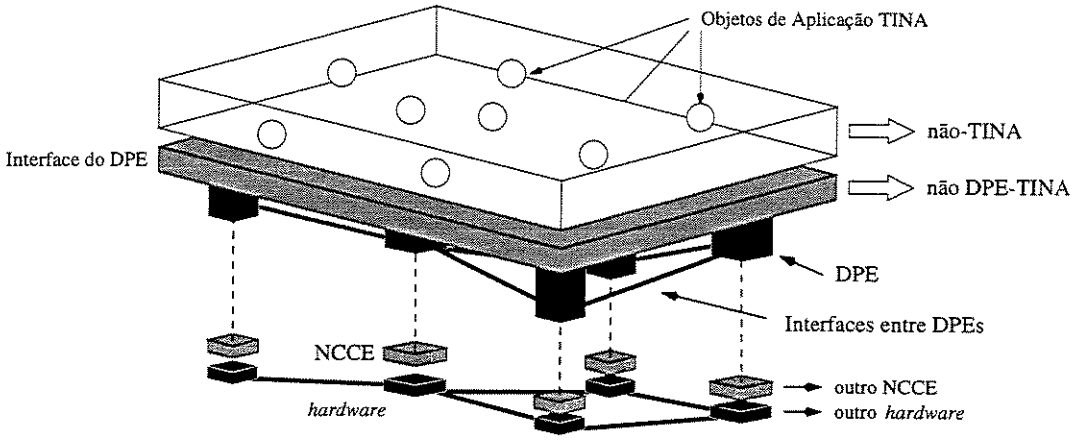

Figura 2.2: Ambiente TINA [23]

Abaixo dos serviços de telecomunicações e do DPE existem mais duas camadas. A camada mais baixa engloba os recursos de *hardware* (processadores, memória e dispositivos de comunicação). Acima desta, existe uma camada que contém o sistema operacional, protocolos de comunicação e outros programas de suporte a um sistema computacional (compiladores, editores). Tal camada é denominada ambiente de comunicação e computação nativa (*Native Computing and Communications Environment*, NCCE). O projeto e implementação dos NCCEs e dos recursos de *hardware* está fora do escopo da arquitetura TINA.

Em geral, as camadas NCCE e de *hardware* não são homogêneas e podem existir diferentes implementações do DPE. A diversidade nas camadas inferiores pode ser devida a questões técnicas ou de mercado. Entretanto, todos os DPEs existentes devem apresentar à camada de aplicação as mesmas funções (interface do DPE). Além disto, os DPEs necessitam ter uma interface padrão para comunicação inter-DPE, a fim de que diferentes implementações trabalhem em conjunto, dando a impressão de uma infra-estrutura homogênea. Finalmente, um sistema TINA pode intercomunicar-se, em qualquer nível, com sistemas não-TINA.

2.2 Decomposição da Arquitetura

A arquitetura TINA aborda uma grande variedade de aspectos e oferece um conjunto amplo de conceitos e princípios. A fim de tratar esta complexidade, a arquitetura é dividida em partições de escopos bem definidos (figura 2.3):

- **Arquitetura de Computação**: fornece um conjunto de conceitos e princípios para o projeto e implementação de *software* e também para o projeto e implementação do ambiente de suporte às aplicações distribuídas;
- **Arquitetura de Serviço**: provê conceitos e princípios para o oferecimento, acesso e utilização de serviços de telecomunicações;
- **Arquitetura de Rede**: define um modelo para os recursos da rede de transporte em diferentes níveis de detalhe;
- **Arquitetura de Gerência**: fornece especificações genéricas para o projeto e implementação de sistemas de *software* utilizados para administração de serviços, de outros *softwares* e de recursos da rede de transporte.

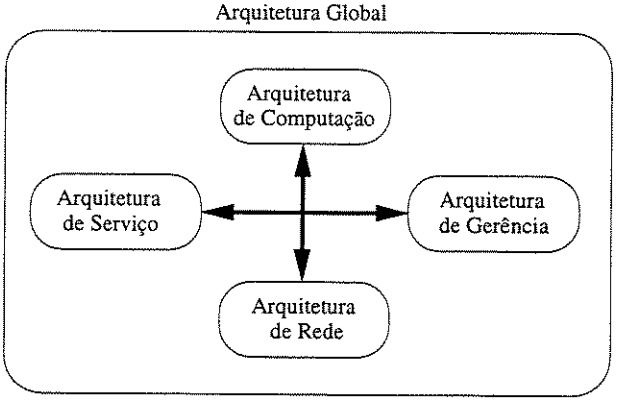

Figura 2.3: Divisão da arquitetura TINA

As subarquiteturas são complementares e, portanto, possuem grande interdependência (figura 2.3). Por exemplo, aplicações de serviço devem ser projetadas e implementadas de acordo com os princípios e conceitos da arquitetura de computação. Os conceitos da arquitetura de gerência são especializados por cada uma das demais arquiteturas. Aplicações de gerência oferecidas como serviços aos usuários devem seguir os conceitos e princípios da arquitetura de serviço. A arquitetura de serviço depende das abstrações oferecidas pela arquitetura de rede para estabelecer, modificar e encerrar conexões na rede de transporte.

Existe também uma **Arquitetura Global**, que define conceitos e princípios mais genéricos a fim de garantir a coerência na especificação, projeto e implementação de qualquer sistema de *software* aderente à arquitetura TINA.

Nas próximas seções, serão descritas, de maneira geral, cada uma das quatro principais divisões da arquitetura TINA.

2.3 Arquitetura de Computação

A arquitetura de computação estabelece conceitos que devem ser utilizados para especificação de *software* em sistemas TINA. Define-se, também, um ambiente para processamento distribuído que proporciona suporte para os serviços de telecomunicações.

Os conceitos da arquitetura de computação baseiam-se no modelo de referência ODP (*Open Distributed Processing*) [25, 26]. Tal como no padrão ODP, a arquitetura TINA determina cinco pontos de vista que devem ser considerados durante a especificação de um sistema de telecomunicações. São eles: empreendimento, informação, computacional, engenharia e tecnologia.

Na arquitetura, são determinados conceitos de modelagem para os quatro primeiros pontos de vista[3]. Estes modelos serão apresentados nas seções a seguir.

2.3.1 Modelo de Empreendimento

O modelo de empreendimento [27] concentra-se no propósito, escopo e políticas do sistema de telecomunicações. Neste ponto de vista, o sistema é descrito a partir da perspectiva das organizações e pessoas que utilizarão suas funções. Os conceitos do modelo incluem um conjunto de diferentes papéis que podem ser desempenhados pelos participantes de um ambiente TINA (usuário, cliente, provedor de serviço, operadora de rede, entre outros).

2.3.2 Modelo de Informação

No modelo de informação [28], aborda-se a semântica dos dados e as atividades de processamento da informação dentro do sistema. Uma especificação de informação concentra-se na definição das entidades de informação e do seu relacionamento mútuo. O modelo define uma notação para descrição dos elementos de informação denominada *quasi* GDMO/GRM (*Guidelines for the Definition of Managed Objects/General Relationship Model*). Além disto, adota-se a técnica OMT (*Object Modeling Technique*) [29] para a representação gráfica do modelo de informação.

2.3.3 Modelo Computacional

O modelo computacional [30, 31] descreve as aplicações de *software* em termos de unidades de programação e encapsulação denominadas objetos computacionais. Esses objetos interagem entre si para proporcionar as funções da aplicação.

Objetos computacionais comunicam-se exclusivamente por meio do envio e do recebimento de informação através de suas interfaces. As interfaces oferecidas pelos objetos são classificadas em dois tipos: interface operacional e interface *stream* (figura 2.4). A interação que ocorre na interface operacional é estruturada em termos de invocações de uma ou mais operações e possíveis respostas a estas invocações. Em uma interface *stream*, a interação ocorre via fluxos de mídia contínuos e

[3] A definição de noções do ponto de vista de tecnologia, que trata da escolha das ferramentas a serem utilizadas na construção do sistema, está fora do escopo da arquitetura TINA.

unidirecionais (áudio e vídeo, por exemplo). Um objeto pode oferecer várias interfaces do mesmo tipo ou de tipos distintos.

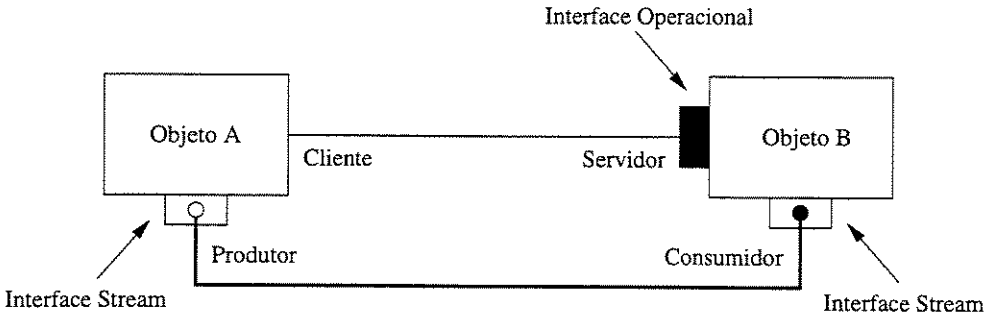

Figura 2.4: Tipos de interface

Além do conceito de objeto, o modelo computacional introduz a noção de grupo de objetos cujo propósito é proporcionar um maior nível de modularidade na definição dos componentes de aplicação. Um grupo de objetos (figura 2.5) é uma coleção de objetos computacionais que funciona como uma entidade única para fins de instalação, uso e manutenção. Em geral, a estrutura e os componentes internos do grupo não são visíveis por objetos clientes. Apenas interfaces definidas como externas, denominadas contratos, são acessíveis por objetos de fora do grupo. Cada grupo de objetos deve possuir um objeto responsável por sua administração, o gerente de grupo.

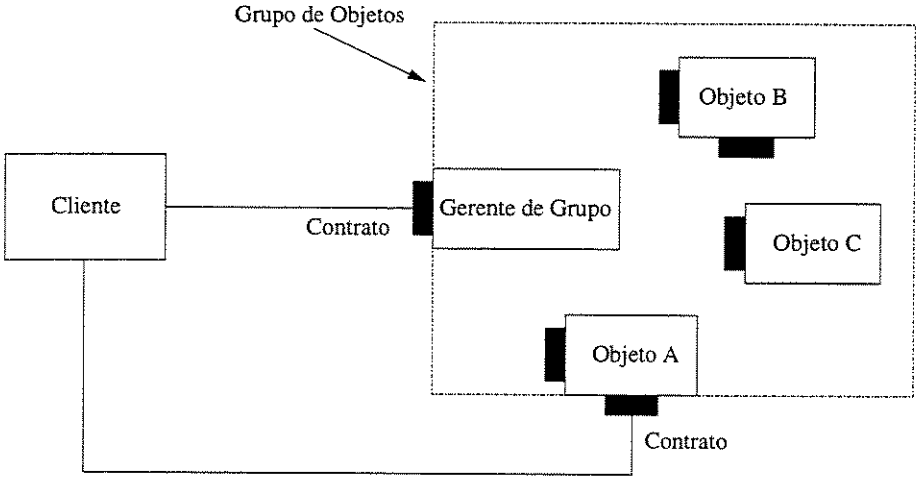

Figura 2.5: Grupo de objetos

O modelo computacional também descreve um conjunto de funções genéricas relacionadas com a gerência do ciclo de vida dos objetos de aplicação[4]. As principais funções são:

- **Criação e remoção de objetos e interfaces:** a criação de um objeto é realizada por meio de uma instanciação de um *template* de objeto. O instanciador do objeto pode ser um construtor de uma linguagem de programação ou um objeto fábrica. Quando um objeto é removido, todas as suas interfaces também são eliminadas. Interfaces podem ser dinamicamente adicionadas e removidas de um objeto.

- **Ativação e desativação de interfaces e objetos:** quando uma interface está desativada, nenhuma interação é possível através desta interface. A desativação de um objeto implica na desativação de todas as suas interfaces. Analogamente, o processo de ativação de um objeto ativa todas as suas interfaces.

- **Migração:** os objetos podem ser movidos de localização durante o seu ciclo de vida.

A notação proposta para a especificação computacional é chamada de ODL (*Object Definition Language*) [32]. A linguagem ODL permite a declaração de interfaces do tipo operacional e *stream* e a definição de atributos não funcionais como qualidade de serviço (tempo de resposta e variação máxima do atraso, por exemplo) e *trading* (proprietário da interface e custo da operação, por exemplo). Além disto, é possível especificar grupos de objetos e seus contratos associados.

2.3.4 Modelo de Engenharia

Para tratar de aspectos relacionados com a distribuição dos componentes das aplicações TINA, utiliza-se o modelo de engenharia [33]. Neste ponto de vista, os objetos definidos na especificação computacional são implementados como componentes de engenharia e espalhados pelos nós da rede de telecomunicações.

O modelo de engenharia também define uma infra-estrutura, denominada DPE (*Distributed Processing Environment*), responsável pelo suporte à execução e interação dos objetos de aplicação.

O DPE (figura 2.6) consiste de núcleos (*kernels*) implementados sobre diversos ambientes de computação (NCCEs). Porém, do ponto de vista da aplicação, o DPE é considerado como uma infra-estrutura homogênea, que esconde a complexidade e a heterogeneidade dos recursos de computação e de rede. A rede de transporte do núcleo (*kernel Transport Network*, kTN) representa uma rede lógica, independente de tecnologia, que suporta a comunicação entre os núcleos do DPE. Além das funções básicas do núcleo, outros componentes, chamados de serviços DPE, oferecem operações genéricas (segurança, transação, entre outras) às aplicações de telecomunicações.

No capítulo 3, serão discutidos, com maior detalhe, os conceitos do modelo de engenharia e os requisitos de uma plataforma DPE.

[4]Grupos de objetos possuem operações de ciclo de vida similares.

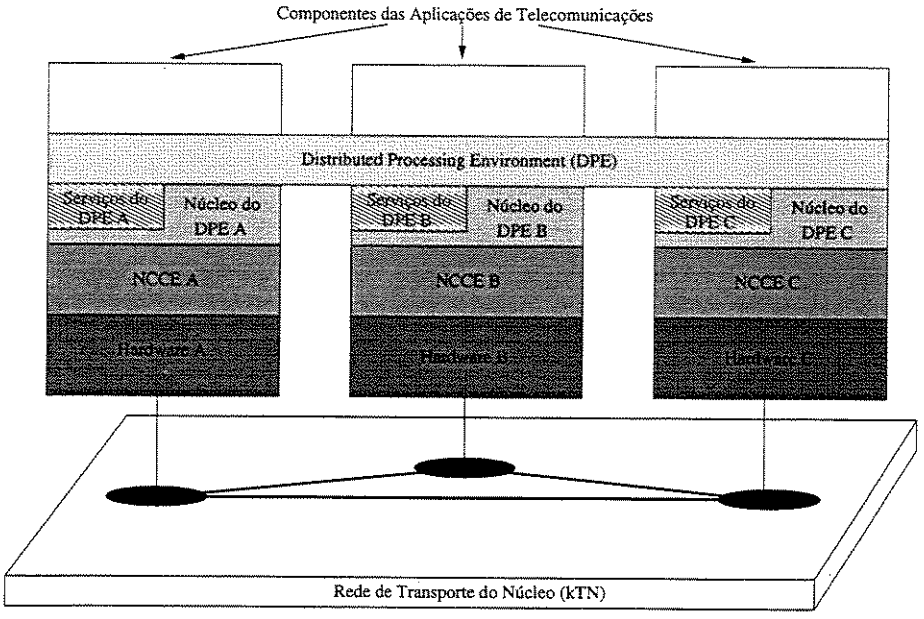

Figura 2.6: Modelo do ambiente para processamento distribuído

2.4 Arquitetura de Serviço

A arquitetura de serviço [34] consiste num conjunto de conceitos e princípios fundamentais para a construção, implantação e utilização de serviços de telecomunicações aderentes à arquitetura TINA. Além disto, a arquitetura define um grupo de componentes reaproveitáveis a ser utilizado na implementação dos serviços.

2.4.1 Paradigma Usuário/Provedor

A arquitetura de serviço faz uso do modelo universal usuário/provedor. Este paradigma serve para definir qualquer relacionamento onde uma entidade (usuário) utiliza serviços oferecidos por outra (provedor).

2.4.2 Separação entre Acesso e Uso

Uma noção importante definida na arquitetura é a separação entre acesso e uso (figura 2.7).

O acesso engloba as atividades realizadas durante o diálogo usuário/provedor e que são necessárias para iniciar a utilização de um serviço. Identificação dos domínios, troca de informação entre usuário/provedor e descoberta dos serviços disponíveis são alguns exemplos de interações de acesso.

O conceito de uso está relacionado com tarefas que governam o comportamento de um serviço e de seus fluxos de dados. Funções de controle de serviço incluem, por exemplo, alteração das características ou do conteúdo de um serviço e troca de informações de gerência. Atividades de controle

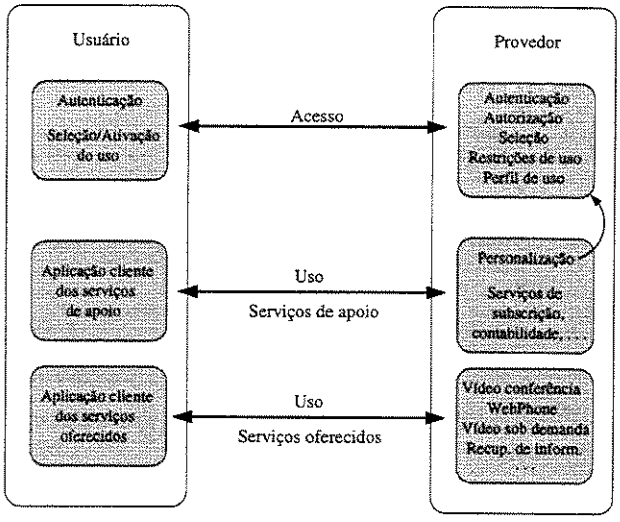

Figura 2.7: Separação entre acesso e uso

de fluxos de dados envolvem o estabelecimento e manutenção das conexões entre componentes do serviço e a modificação das condições da qualidade da comunicação (*Quality of Service*, QoS).

2.4.3 Conceito de Sessão

Como a arquitetura TINA tem como objetivo suportar serviços de telecomunicações mais sofisticados, o consórcio estabelece a noção de "sessão" que substitui a idéia tradicional de "chamada"(*call*). Uma sessão é uma "relação temporária entre um grupo de recursos que são designados a efetuar, de forma conjunta, uma tarefa por um período de tempo" [34]. Ou seja, o conceito de sessão proporciona uma forma de agrupar atividades específicas de um serviço durante um determinado intervalo de tempo. O propósito da idéia de sessão é separar diferentes aspectos dos serviços e promover a distribuição de suas funções.

A arquitetura de serviço define três tipos de sessão: sessão de acesso, sessão de serviço e sessão de comunicação (figura 2.8).

A sessão de acesso representa o ponto de contato inicial entre o usuário e o conjunto de serviços disponíveis. Na sessão de acesso, o usuário é capaz de requisitar ou associar-se, de maneira segura e personalizada, a serviços de telecomunicações. Uma característica importante deste conceito é que a sessão permite acesso ubíquo aos serviços (mobilidade do usuário).

A sessão de serviço proporciona um contexto para a execução, controle e administração dos serviços. Existem duas divisões da sessão de serviço: sessão de serviço do provedor e sessão de serviço do usuário. A visão global do serviço (lista de participantes, relacionamento entre os membros da sessão de serviço, entre outras informações) é representada pela sessão de serviço do provedor. Já a sessão de serviço do usuário corresponde à visão local de cada participante do serviço.

A sessão de comunicação oferece uma abstração dos recursos de rede necessários para estabelecer as conexões entre os usuários. Assim, este tipo de sessão proporciona funções de comunicação, independentes da tecnologia de rede, para as demais sessões. A sessão de comunicação utiliza

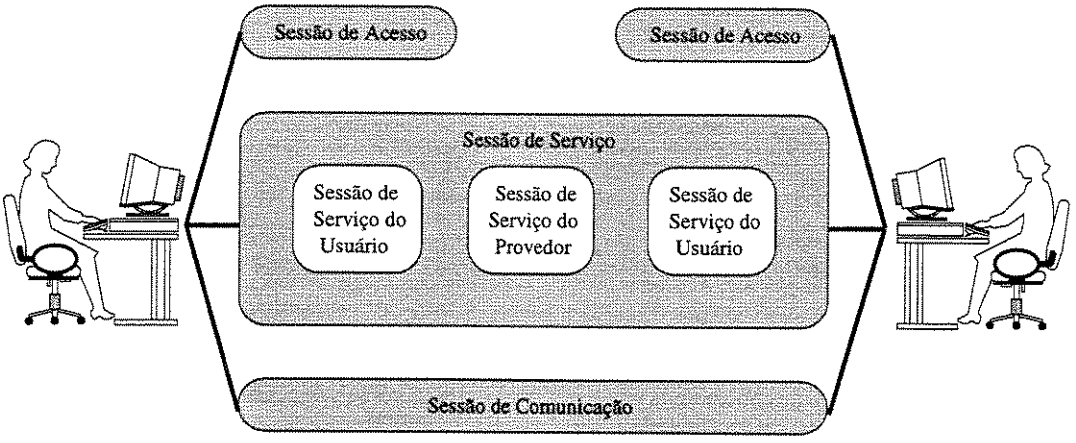

Figura 2.8: Tipos de sessão

conceitos definidos nas arquiteturas de rede e de gerência.

2.4.4 Componentes de Serviço

Seguindo a arquitetura computacional, os serviços TINA são descritos em termos de componentes (objetos) que interagem via interfaces. De acordo com os tipos de sessão apresentados na seção anterior, a arquitetura de serviço introduz um conjunto de componentes genéricos utilizados na construção de serviços de telecomunicações. Ou seja, cada tipo de sessão é implementado por meio de um conjunto específico de objetos.

Ao contrário da sessão de serviço, as sessões de acesso e de comunicação são independentes do tipo de serviço oferecido. Conseqüentemente, os componentes que implementam as funções de acesso e de comunicação são mais gerais, enquanto os objetos responsáveis pela sessão de serviço são, na verdade, bem específicos. De qualquer forma, a arquitetura define um grupo de objetos genéricos para a sessão de serviço.

Este "*kit* de construção de serviço" possui, por exemplo, objetos do tipo: agente do usuário e do provedor (sessão de acesso), gerente da sessão de serviço e fábrica de serviços (sessão de serviço) e gerente da sessão de comunicação (sessão de comunicação)[5]. Estes componentes podem servir para a fabricação de outros objetos por meio da aplicação de princípios de orientação a objetos como especialização e composição.

2.5 Arquitetura de Rede

O propósito da arquitetura de rede [35] é oferecer um conjunto de conceitos que descrevam, de maneira independente da tecnologia, a rede de transporte e fornecer mecanismos para o estabelecimento, modificação e liberação das conexões de rede.

[5]A relação completa dos componentes de serviço e a descrição de suas interações estão na especificação TINA [34].

A arquitetura de rede incorpora princípios das recomendações G.803 [36] e M.3100 [37] do ITU. Os principais conceitos derivados da especificação G.803 estão relacionados com a partição e com a estruturação em camadas da rede de transporte.

O conceito de partição determina que uma rede pode ser decomposta em sub-redes conectadas por enlaces. Esta divisão pode ser aplicada sucessivamente às sub-redes até se atingir a quantidade de detalhe desejada. Geralmente, no nível mais baixo de abstração, uma sub-rede é equivalente a um único elemento de rede (comutador, por exemplo).

A idéia da estruturação é descrever a rede como um grupo de camadas. Cada camada apresenta um conjunto de entradas e saídas (pontos de acesso, *access points*) para fins de intercomunicação e é caracterizada pelo tipo de informação que transporta.

Em geral, as camadas de rede possuem um relacionamento cliente/servidor entre si. Um enlace entre duas sub-redes na camada cliente é suportado por uma conexão fim a fim (*trail*) na camada servidora. A figura 2.9 mostra uma rede com duas camadas, divididas em sub-redes e enlaces, que têm um relacionamento cliente/servidor.

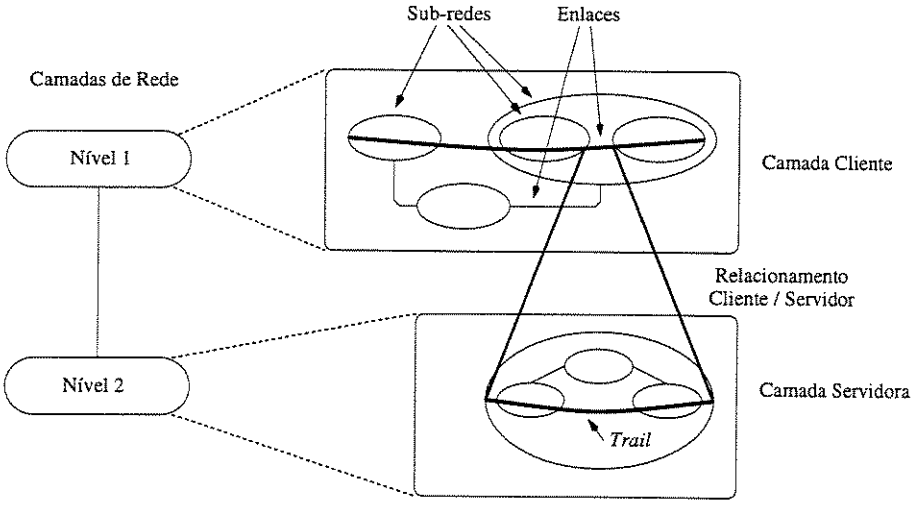

Figura 2.9: Partição e estruturação em camadas de uma rede de transporte

A arquitetura também possui uma especificação relacionada com a modelagem dos recursos de rede (*Network Resource Information Model*, NRIM) [38]. O modelo NRIM descreve como os elementos de rede estão relacionados, conectados e configurados para suportar e manter conexões fim a fim. Esta descrição genérica dos componentes de rede pode ser especializada para cada tipo de tecnologia em particular.

O modelo NRIM fornece uma descrição detalhada da rede de transporte. Entretanto, muitas destas informações não precisam, ou não devem, ser conhecidas pelo usuário da rede[6]. Para proporcionar uma visão da conexidade da rede mais orientada aos serviços, utiliza-se o conceito de grafo de conexão (*connection graph*).

Em um grafo de conexão (figura 2.10), os arcos representam a conexidade entre as portas dos

[6]Exceto no caso de atividades de gerência de rede.

vértices. Existem dois tipos de grafos de conexão. No grafo de conexão lógica, os vértices representam objetos computacionais, as portas indicam interfaces do tipo *stream* e os arcos representam o fluxo de dados. No grafo de conexão física, os vértices representam nós computacionais, as portas descrevem os pontos de acesso da rede (*sockets*, por exemplo) e os arcos indicam as conexões físicas.

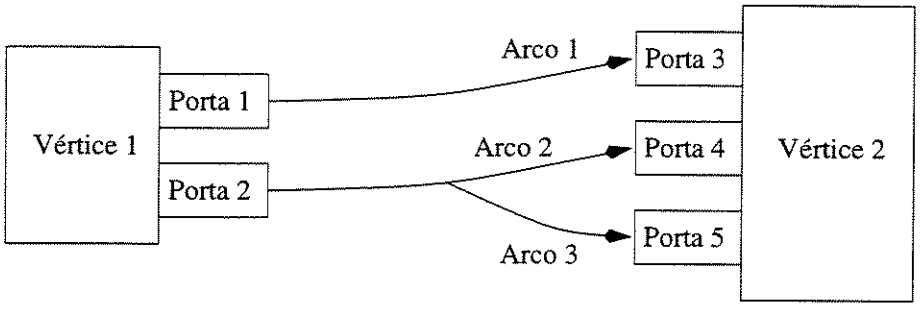

Figura 2.10: Grafo de conexão

Por fim, para tratar do estabelecimento, modificação e liberação de conexões de rede, é definido um conjunto de objetos computacionais chamado de arquitetura de gerência de conexão (figura 2.11).

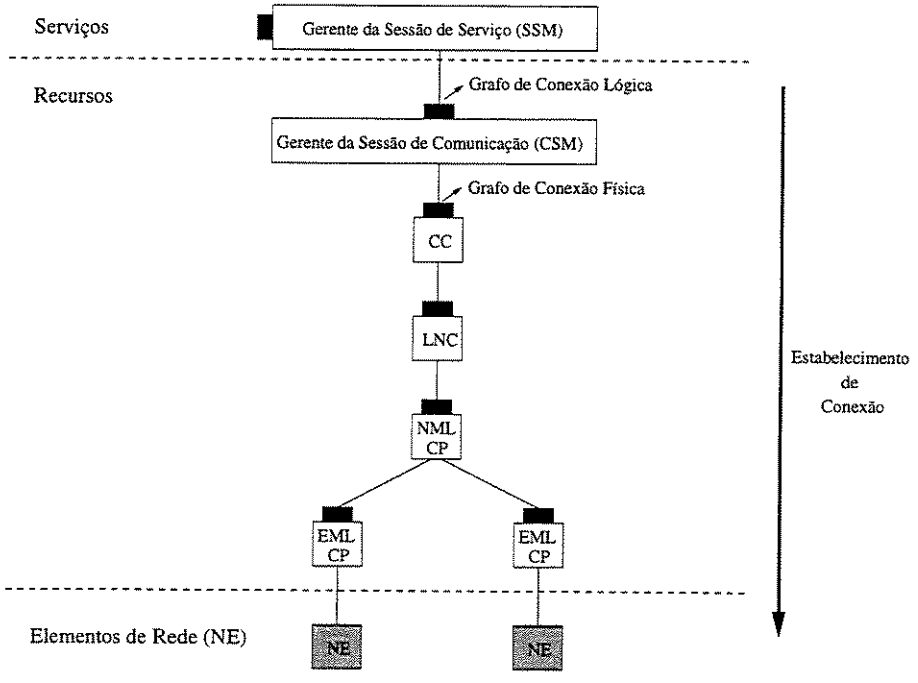

Figura 2.11: Arquitetura de gerência de conexão

Para requisitar uma ligação na rede de transporte, um componente de serviço, o Gerente de Sessão de Serviço (*Service Session Manager*), envia um grafo de conexão lógica a um objeto da

arquitetura de gerência de conexão, o Gerente de Sessão de Comunicação (*Communication Session Manager*, CSM). A função do CSM é converter o grafo de conexão lógica em um grafo de conexão física correspondente. O CSM também é responsável pelo controle (estabelecimento, modificação e liberação) da conexão física entre os elementos da rede. Para realizar esta última tarefa, o CSM utiliza serviços de outros objetos da arquitetura de gerência de conexão (figura 2.11):

- Coordenador de Conexão (*Connection Coordinator*, CC);
- Coordenador da Camada de Rede (*Layer Network Coordinator*, LNC);
- Realizador de Conexão da Camada de Gerência de Rede (*Network Management Layer Connection Performer*, CP);
- Realizador de Conexão da Camada de Gerência de Elemento (*Element Management Layer Connection Performer*, CP).

Cada um destes objetos tem como função estabelecer conexões fim a fim nas sub-redes das diversas camadas que compõem a rede de transporte.

2.6 Arquitetura de Gerência

A arquitetura de gerência [39] provê um conjunto de princípios e conceitos genéricos de administração para as demais subdivisões da arquitetura TINA (serviço, rede e computação). Ou seja, todas as demais subarquiteturas TINA sofrem grande influência destes princípios de gerência.

A definição da arquitetura baseia-se principalmente nos padrões de gerência TMN [8] e OSI (*Open Systems Interconnection*).

As camadas TMN (negócio, serviço, rede e elemento de rede) são adotadas pela arquitetura TINA. A arquitetura de serviço concentra-se na camada de gerência de serviço, enquanto a arquitetura de rede trata das camadas de administração de rede e de elemento de rede. Na arquitetura TINA, ainda não foram especificados conceitos relacionados com a camada de gerência de negócio.

Além disto, a arquitetura de gerência segue as áreas funcionais definidas no padrão OSI: administração de configuração, de falha, de desempenho, de contabilidade e de segurança.

Assim, na arquitetura TINA, tem-se a definição de componentes (objetos) de administração para as diversas áreas funcionais (padrão OSI) dentro de cada camada de gerência (padrão TMN).

Na arquitetura de serviço, a ênfase reside na área de gerência de configuração e na administração dos vários tipos de sessão (acesso, serviço e comunicação). A gerência de sessão reúne funções necessárias para ativar, modificar, suspender, continuar e encerrar uma sessão. Estas tarefas são realizadas por objetos computacionais chamados de "gerentes de sessão". São definidos, ainda, objetos computacionais genéricos cuja responsabilidade é administrar a subscrição e contabilidade dos serviços.

A arquitetura de rede concentra-se na administração dos recursos da rede de transporte. A especificação incorpora as cinco áreas funcionais do padrão OSI e faz um importante aperfeiçoamento. A área de administração de configuração é especializada em duas subáreas: gerência de recurso e gerência de conexão. A gerência de recurso (configuração) engloba o suporte à instalação, monitoração e controle dos elementos de rede. A gerência de conexão inclui funções relacionadas com o estabelecimento, controle e liberação de conexões de rede.

A gerência de conexão representa uma nova abordagem para o controle de conexões. Tradicionalmente, a administração das ligações de rede é vista como responsabilidade dos equipamentos de comutação. Na arquitetura TINA, o controle e estabelecimento de conexões é considerado como um serviço de gerência de rede e ocupa o mesmo nível das demais funções de administração. Desta forma, os objetos computacionais da arquitetura de gerência de conexão (figura 2.11, seção 2.5) oferecem funções de conexidade, padronizadas, para os componentes localizados na camada de gerência de serviço (gerente de sessão de serviço, SSM).

A administração na arquitetura de computação é dividida em duas partes: gerência genérica de *software* e gerência de infra-estrutura. A gerência genérica de *software* trata da instalação, configuração e remoção de um sistema de *software*. Já a gerência de infra-estrutura é responsável pela administração dos nós computacionais (NCCEs)[7], do ambiente de suporte (DPE) e da rede de transporte do núcleo (kTN).

[7]Na verdade, a gerência direta dos NCCEs está fora do escopo da arquitetura TINA, porém o DPE pode oferecer funções úteis para sua administração indireta.

Capítulo 3

Arquitetura do DPE

A especificação computacional de uma aplicação TINA consiste de objetos que interagem entre si via interfaces (seção 2.3.3). Sob esta perspectiva, o projetista não se preocupa em como realizar a distribuição dos objetos pelo sistema.

Já o modelo de engenharia [33] trata de aspectos relacionados com a especificação de aplicações de *software* distribuídas. Os conceitos de engenharia determinam como estruturar e dispersar os componentes de serviço TINA pelos diversos elementos da rede de telecomunicações.

Para suportar a execução dos objetos de engenharia, é definida uma infra-estrutura (*Distributed Processing Environment*, DPE) cujo objetivo é esconder os mecanismos necessários para realizar a interação entre os componentes de aplicação que estão dispersos em um sistema distribuído (potencialmente) heterogêneo.

A arquitetura do DPE contém requisitos na forma de conceitos, mecanismos e interfaces exigidos para a construção de um ambiente para processamento distribuído aderente aos padrões TINA.

3.1 Conceitos de Distribuição (*Deployment Concepts*)

No modelo de engenharia, um objeto computacional é representado como um objeto computacional de engenharia (*engineering Computational Object*, eCO - figura 3.1).

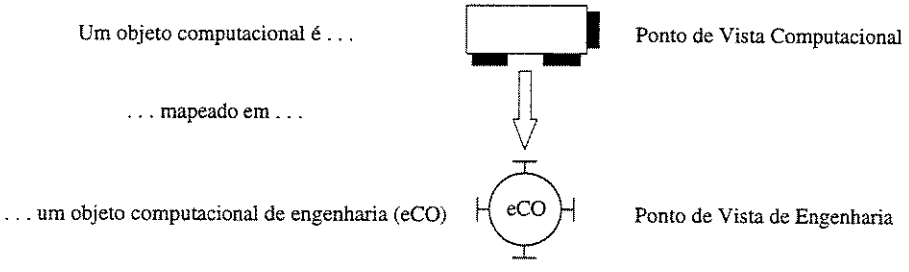

Figura 3.1: Objeto computacional e seu eCO correspondente

A diferença entre um objeto computacional e seu eCO correspondente está no nível de abstração. Um eCO pode interagir com objetos de engenharia que não têm representação correspondente no

modelo computacional. Por exemplo, um eCO pode invocar objetos de engenharia cuja função exclusiva é estabelecer canais de comunicação com outros eCOs.

As interfaces do eCO representam as mesmas interfaces do objeto computacional correspondente. Estas interfaces podem ser ligeiramente modificadas por meio de conversões de tipos e/ou adições de operações necessárias para garantir a interação com objetos de engenharia. Além disto, interfaces de gerência podem ser acrescentadas ao eCO. O comportamento definido para o objeto computacional não é alterado durante este processo de transformação.

Na visão de engenharia, uma especificação computacional pode gerar várias configurações de distribuição (figura 3.2). A estrutura de engenharia a ser escolhida vai depender de fatores como qualidade de serviço, balanceamento de carga, entre outros.

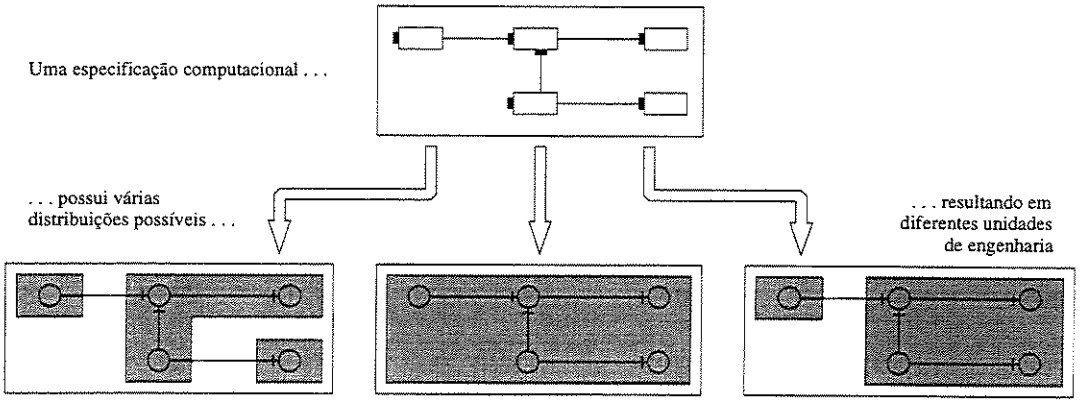

Figura 3.2: Distribuição de objetos computacionais em unidades de engenharia

Os objetos computacionais de engenharia são agrupados em unidades de recurso e de distribuição (figura 3.3).

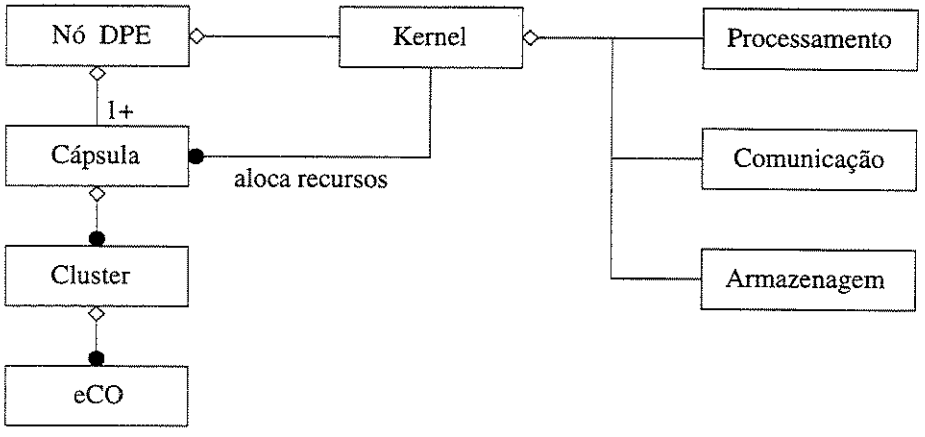

Figura 3.3: Relacionamento entre os conceitos de distribuição na notação OMT

As unidades de recurso representam uma coleção de recursos computacionais e de políticas de reserva destes recursos. No modelo de engenharia, duas unidades de recurso são definidas: nó e cápsula.

Existem várias classificações para o conceito de nó. Um nó de rede é uma unidade que representa qualquer terminal da rede (telefone, computador e roteador, por exemplo). Um nó de rede que possui capacidade de processamento é chamado de nó computacional. Se um nó computacional possui uma plataforma DPE, é também classificado como um nó DPE. Um nó DPE, a partir daqui denominado simplesmente nó, consiste de recursos de processamento (processos, *threads*, escalonador), de armazenagem (RAM, discos) e de comunicação (pilhas de protocolo, mecanismos de comunicação inter-processos). O nó administra estes recursos de forma autônoma. Exemplos de nó: computadores com um ou mais processadores, redes locais com um sistema operacional distribuído.

A cápsula é uma unidade para fins de alocação dos recursos computacionais de um nó. Os objetos localizados em uma cápsula compartilham as mesmas políticas de reserva de recursos realizadas pelo núcleo (*kernel*) do nó. Exemplo de cápsula: processo UNIX.

O modelo de engenharia define apenas uma unidade de distribuição: o *cluster*. O *cluster* é um conjunto de objetos que possui as seguintes propriedades:

- Unidade de localização: um conjunto de eCOs forma uma unidade de localização se todos os objetos estão sempre localizados em uma mesma cápsula;
- Unidade de ativação: um conjunto de eCOs forma uma unidade de ativação se as operações de ativação/desativação sempre são aplicadas sobre todos os objetos da unidade;
- Unidade de migração: um conjunto de eCOs forma uma unidade de migração se todos os objetos sempre são movidos de um local para o outro de forma conjunta.

Como o *cluster* não possui propriedades de uma unidade de instanciação[1], é possível adicionar e remover eCOs durante o seu ciclo de vida.

Os conceitos de nó, cápsula, *cluster* e eCO[2] do modelo de engenharia do consórcio TINA são similares aos do modelo de referência ODP.

3.2 Gerência do Ciclo de Vida dos Objetos

Uma plataforma DPE deve oferecer suporte a operações básicas de gerência do ciclo de vida para os componentes das aplicações TINA (seção 2.3.3). A gerência do ciclo de vida envolve a definição de interfaces de administração de eCOs, *clusters*, cápsulas e nós.

3.2.1 Gerência de eCOs

A criação de instâncias de objetos TINA, normalmente, é responsabilidade das fábricas de objeto (*object factories*). A função básica de uma fábrica é a seguinte: ao receber uma requisição para a

[1] Uma unidade de instanciação requer que todos os objetos da unidade sejam instanciados no mesmo momento.
[2] No modelo ODP, esta unidade é chamada de Objeto Básico de Engenharia (OBE).

criação de um objeto, a fábrica instancia um novo eCO, insere o objeto em um *cluster* determinado e retorna sua referência. Em geral, o próprio eCO oferece a operação de autodestruição.

A desativação de um objeto muda seu estado de ativo para um outro inativo (recuperável). O processo inverso é chamado de reativação. O estado do objeto reativado é idêntico ao do momento imediatamente anterior à sua desativação.

3.2.2 Gerência de *Clusters*

A operação de criação de um *cluster* necessita da referência da cápsula onde ele deve ser instanciado. Como no caso dos eCOs, esta tarefa normalmente é responsabilidade de uma fábrica. A remoção de um *cluster* implica no desaparecimento de todos os eCOs presentes no *cluster*.

Como o *cluster* não é um grupo estático (não é uma unidade de instanciação), é possível adicionar (remover) eCOs dinamicamente ao *cluster*. As operações de adição/remoção de objetos invocam as funções construtoras/destrutoras correspondentes no nível de gerência de eCOs.

A ativação de um *cluster* coloca todos os eCOs que o compõem em um estado ativo, ou seja, prontos para atender requisições de operações por meio de suas interfaces.

A desativação envolve a preservação do estado de cada eCO pertencente ao *cluster*. Esta informação é armazenada de tal forma que possa ser recuperada posteriormente. Tal operação é chamada de *checkpoint*. Além do *checkpoint* do *cluster*, a desativação coloca todos os objetos em um estado inativo. O processo de reativação é inverso e recupera o estado do *cluster* armazenado durante o *checkpoint*.

O *checkpoint* pode ser invocado diretamente sobre o *cluster*. Ou seja, após a operação de *checkpoint*, o *cluster* ainda permanece em um estado ativo. Durante o ciclo de vida do *cluster*, é possível executar diversas vezes tal operação, a fim de armazenar vários estados do *cluster* ao longo do tempo. Assim, esta função pode ser útil no auxílio à recuperação dos objetos em caso de falha na cápsula.

O modelo não determina como deve ser implementada a movimentação dos objetos de engenharia. Uma forma de realizar a migração de um *cluster* é a seguinte: desativar o *cluster* (removendo-o da cápsula antiga) e reativá-lo em uma nova cápsula.

3.2.3 Gerência de Nós e Cápsulas

As operações de gerência de nós e de cápsulas ainda não foram definidas pelo consórcio TINA.

3.3 Transparências de Distribuição

Transparências de distribuição escondem certos aspectos relacionados com a interação entre objetos distribuídos em um sistema heterogêneo. Estas abstrações são um conceito importante utilizado na especificação computacional, onde o projetista não deve se preocupar com os mecanismos necessários para realizar a comunicação entre os objetos. A especificação TINA descreve as seguintes transparências de distribuição [33]:

- **Acesso**: oculta diferenças relacionadas com a representação dos dados e com os mecanismos de invocação existentes em sistemas de computação heterogêneos;

- **Localização**: esconde a localização do objeto de outros componentes da aplicação que interagem com ele;
- **Migração**: permite que o componente que está migrando preserve seu estado durante a mudança de local sem prejudicar a comunicação com outros objetos;
- **Federação**: oculta diferenças existentes entre domínios administrativos distintos;
- **Falha**: esconde dos demais objetos as falhas e ações para recuperar um objeto defeituoso;
- **Recurso**: oculta de um componente da aplicação os processos de ativação e desativação de outros objetos;
- **Concorrência**: possibilita o acesso concorrente de um objeto por vários outros componentes;
- **Replicação**: mascara a replicação de um componente dos demais objetos da aplicação;
- **Transação**: esconde os mecanismos que garantem as propriedades ACID[3] das operações que ocorrem no sistema.

Diferentes aplicações possuem requisitos de transparência distintos. Ou seja, na especificação computacional, o projetista pode selecionar aquelas transparências que mais lhe convém. O DPE é responsável pelo oferecimento de mecanismos que garantam estas transparências. Segundo o padrão, o suporte às transparências de acesso e localização deve existir em toda plataforma DPE aderente à arquitetura TINA, enquanto o suporte às demais transparências é considerado opcional.

3.4 Comunicação entre Objetos

Uma interação envolvendo múltiplos objetos é chamada de *binding*. Os *bindings* são classificados em dois tipos: implícitos e explícitos.

O *binding* implícito é aquele estabelecido pela infra-estrutura de distribuição (DPE) sem a participação direta da aplicação. Tipicamente, o *binding* implícito ocorre quando um objeto invoca funções de uma interface operacional.

O *binding* explícito é estabelecido por meio de uma requisição expressa feita por algum objeto. Tal *binding* é modelado como um objeto computacional comum (*binding object*) que encapsula os mecanismos de *binding* e oferece operações de controle sobre a interação (adição e remoção de participantes, modificação da qualidade de serviço). Interfaces *stream* são conectadas via *binding* explícito.

A comunicação entre objetos que estão no mesmo *cluster* está fora do escopo do modelo de engenharia. Este tipo de interação pode ser realizado via memória compartilhada, chamadas locais de procedimento ou comunicação inter-processos, por exemplo.

Para a comunicação entre objetos pertencentes a *clusters* distintos, mecanismos devem proporcionar suporte às transparências de distribuição. Estes mecanismos são oferecidos como parte de um canal (figura 3.4), como na especificação RM-ODP. O conceito de canal vale tanto para *bindings*

[3] *Atomicity, Consistency, Isolation, Durability*

explícitos quanto implícitos. No caso de *bindings* explícitos, o canal oferece uma interface de controle sobre o *binding*. Ou seja, na visão de engenharia, a interface do *binding object* é representada pela interface de controle do canal.

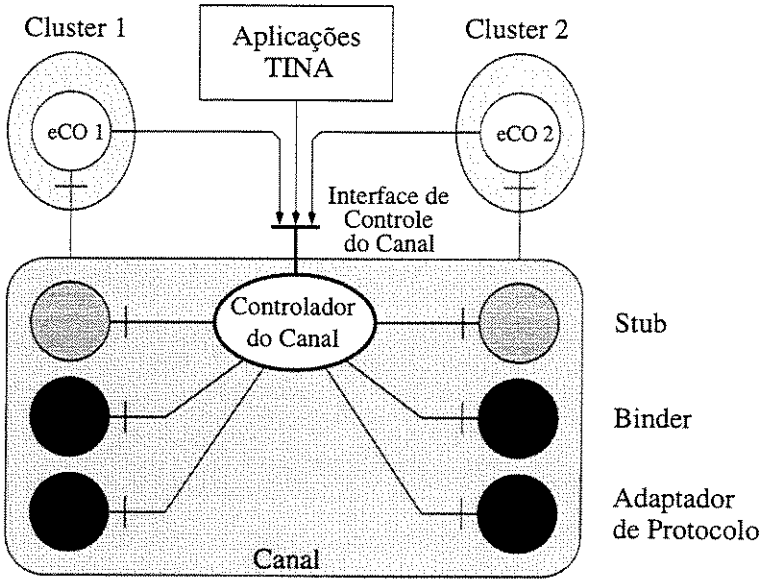

Figura 3.4: Canal ligando eCOs pertencentes a *clusters* distintos

O canal possui três componentes fundamentais: *stub*, *binder* e adaptador de protocolo (figura 3.4).

A função do *stub* é executada antes e depois da comunicação propriamente dita. Sua responsabilidade é realizar as conversões necessárias sobre os dados (*marshalling/unmarshalling*) e adicionar outras informações que proporcionem transparência de acesso. O código das funções de *stub* pode ser gerado automaticamente por meio de ferramentas de compilação a partir da descrição das interfaces dos objetos.

O *binder* tem a responsabilidade, juntamente com os respectivos eCOs, de manter a integridade da ligação. O *binder* armazena informações sobre o canal e interage com outras funções da infraestrutura para mantê-las consistentes em caso de erro do *binding* (devido a falha, migração ou desativação dos objetos, por exemplo).

O adaptador de protocolo oferece mecanismos para que objetos localizados em nós diferentes se comuniquem. O adaptador suporta um protocolo que garante a semântica da interação durante a comunicação inter-nó. Este protocolo pode ser construído acima de um protocolo de transporte orientado a conexão (TCP/IP, por exemplo) ou não (UDP, por exemplo).

Onde os componentes do canal são colocados, é um aspecto de implementação do DPE e, portanto, não especificado no modelo de engenharia.

3.5 Serviços DPE

As funções básicas do DPE (comunicação, gerência do ciclo de vida dos objetos e suporte às transparências de distribuição) fazem parte do núcleo (*kernel*) da plataforma.

Além das funções de núcleo, a infra-estrutura pode proporcionar uma série de serviços genéricos e bastante úteis para uma ampla variedade de aplicações. Estes serviços "comuns", chamados de serviços DPE, são oferecidos por meio de interfaces computacionais disponíveis aos objetos de aplicação. Os seguintes serviços foram classificados pelo consórcio TINA como serviços DPE[4]:

- **Serviço de *trading***: o cliente (importador) deste serviço invoca um *trader* para encontrar dinamicamente ofertas de serviços exportadas por objetos servidores.
- **Serviço de transação**: garante as propriedades ACID para operações oferecidas por um objeto.
- **Serviço de repositório**: oferece armazenagem persistente para *templates* de interfaces, de objetos e de grupos de objetos (repositório de especificação). Também mantém informações sobre as implementações (código executável) dos objetos (repositório de implementação).
- **Serviço de notificação**: permite que um objeto envie (receba) notificações para (de) um conjunto de objetos sem ter que interagir explicitamente com cada um dos componentes da aplicação.
- **Serviço de segurança**: proporciona funções de suporte aos requisitos de segurança das aplicações TINA (autenticação, autorização, criptografia).
- **Serviço monitor de desempenho**: verifica o desempenho das atividades realizadas pelos elementos da rede. Exemplos de parâmetros que podem ser pesquisados: carga, índice de disponibilidade, taxa de erros, tempo de resposta, *throughput*.

3.6 Propriedades e Qualidade de Serviço do DPE

Além dos requisitos funcionais da plataforma DPE, a arquitetura possui um conjunto de exigências não funcionais expressas por meio de propriedades e atributos de qualidade de serviço (*Quality of Service*, QoS).

As propriedades são requisitos intrínsecos do sistema e, portanto, não negociáveis. Escalabilidade é um exemplo de propriedade. Para garantir o suporte às propriedades do DPE, é necessária a colaboração da infra-estrutura e das próprias aplicações.

Outra categoria de requisitos não funcionais é chamada de qualidade de serviço. A qualidade de serviço envolve requisitos negociáveis (via *trader*, por exemplo) entre um cliente e um provedor de serviços. Disponibilidade, desempenho e tolerância a falhas podem ser classificados nesta categoria.

[4] A lista não é exaustiva. No futuro, outras funções genéricas podem ser classificadas como serviços DPE.

3.7 Perfil do DPE

Nem todos os serviços TINA necessitam do mesmo tipo de suporte oferecido pela infra-estrutura. Algumas funções do DPE podem ou não ser utilizadas por uma aplicação específica. Ou seja, cada aplicação possui um conjunto de requisitos a ser atendido pela plataforma.

A noção de perfil do DPE corresponde a um conjunto de conceitos, serviços, propriedades e qualidades de serviço que uma implementação do DPE é capaz de prover. Estes requisitos, identificados no modelo de engenharia, são classificados como fundamentais ou opcionais. Os requisitos fundamentais devem estar presentes em qualquer implementação de uma infra-estrutura DPE-TINA.

O suporte à comunicação entre componentes de aplicação é um requisito fundamental: assume-se que qualquer ambiente para processamento distribuído aderente à arquitetura TINA proporcione mecanismos que garantam a interação entre objetos. Assim, o suporte a canais, *stubs*, *binders* e adaptadores de protocolo é obrigatório em qualquer plataforma DPE.

Entre as transparências de distribuição, duas são consideradas fundamentais: a transparência de acesso, para garantir a comunicação entre ambientes heterogêneos e a transparência de localização, a fim de permitir a cooperação entre os objetos, independente da sua posição no sistema.

Os conceitos de distribuição (nó, cápsula, *cluster* e eCO) também devem ser incorporados por qualquer implementação do DPE.

À exceção das transparências de acesso e localização, o suporte a todas as demais transparências de distribuição é considerado um requisito opcional do DPE.

Operações de gerência do ciclo de vida das unidades de engenharia são especificadas como opcionais, visto que nem todas as aplicações TINA precisam destas funções.

O suporte aos serviços DPE é considerado opcional.

Requisitos relacionados com as propriedades e com a qualidade de serviço do DPE também são classificados como opcionais.

3.8 Aspectos de Interoperabilidade

Dentro de um sistema de telecomunicações, diferentes nós computacionais podem possuir implementações distintas da plataforma DPE (seção 2.3.4). No entanto, estas implementações devem oferecer às aplicações TINA uma aparência homogênea.

Várias características da arquitetura do DPE ajudam a alcançar o propósito da intercomunicação de plataformas distintas:

- Suporte obrigatório às transparências de localização e de acesso, que são essenciais para ocultar a natureza distribuída e heterogênea do sistema de telecomunicações.
- Modelo de objetos comum (unidades de engenharia) a todas as implementações.
- Operações de gerência do ciclo de vida com semântica padronizada. Desta forma, é possível administrar remotamente (a partir de outro DPE) os componentes da aplicação.
- Definição dos serviços DPE como aplicações TINA comuns. Assim, qualquer objeto pode invocar, local ou remotamente, estes serviços via interfaces padronizadas.

Entretanto, outros aspectos também estão relacionados com a questão de interoperabilidade. Estes fatores envolvem escolhas tecnológicas como a definição do protocolo de comunicação inter-DPE e do formato canônico para descrição das referências de objetos TINA.

Capítulo 4

Projeto e Implementação de um DPE-TINA

A principal contribuição deste trabalho refere-se ao desenvolvimento de um ambiente para processamento distribuído (DPE) aderente à arquitetura TINA. Tal infra-estrutura foi construída sobre a plataforma CORBA (*Common Object Request Broker Architecture*), especificada pelo consórcio OMG (*Object Management Group*). Utilizou-se, também, um banco de dados orientado a objetos para realizar a armazenagem persistente dos objetos que compõem o sistema.

Neste capítulo, aborda-se os apectos de projeto e implementação das soluções encontradas para atender os requisitos de um DPE-TINA (capítulo 3). Além disto, são descritos alguns outros sistemas existentes, aderentes ou não ao padrão TINA, que podem oferecer suporte, nos aspectos de mobilidade e de comunicação via fluxos multimídia, a aplicações de telecomunicações.

4.1 CORBA (*Common Object Request Broker Architecture*)

A plataforma CORBA [40, 41] do consórcio OMG permite a interação entre objetos distribuídos em um sistema heterogêneo. Clientes podem executar operações de servidores sem se preocupar com detalhes como: localização do objeto invocado, linguagem de programação utilizada na implementação do servidor e diferenças existentes entre os ambientes dos objetos (*hardware*, sistemas operacionais, protocolos de comunicação). Os principais componentes da arquitetura CORBA são (figura 4.1):

- ***Servant***: responsável pela execução das operações especificadas na interface do objeto CORBA. A interface é definida por meio da linguagem declarativa IDL (*Interface Definition Language*) [40] e os *servants* podem ser implementados em várias linguagens de programação (C, C++, Java, Cobol, por exemplo).

- **Cliente**: para executar operações dos *servants*, o cliente deve obter uma referência de objeto que identifica univocamente o servidor. A forma como as operações dos *servants* são invocadas não depende da localização (local ou remota) do servidor em relação ao cliente.

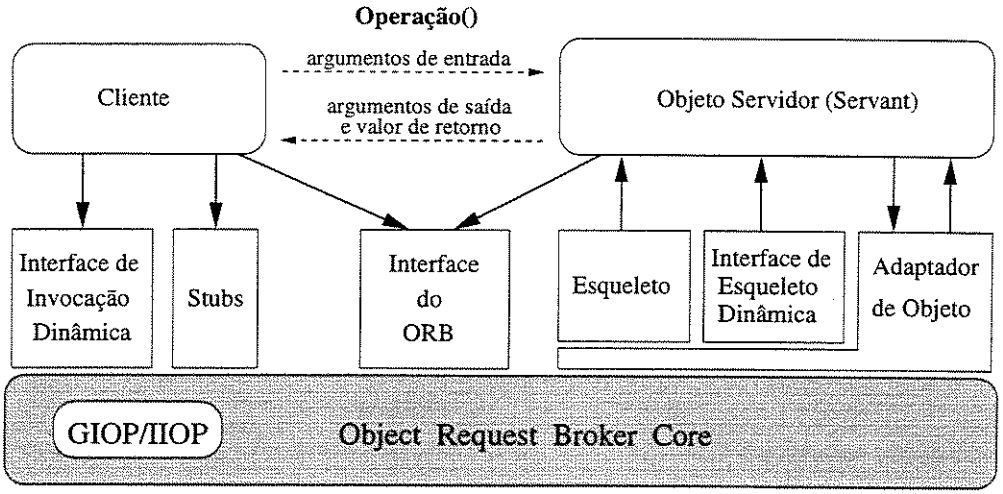

Figura 4.1: Componentes da arquitetura CORBA

- **Núcleo do ORB (ORB *Core*)**: responsável pela transmissão da requisição de operação para o servidor e do retorno, se for o caso, da resposta ao cliente. O núcleo do ORB pode ser implementado de várias maneiras (processo servidor, recurso do sistema operacional, bibliotecas de funções). Para realizar a comunicação entre diferentes núcleos de ORB, utiliza-se o protocolo padrão IIOP (*Internet Inter-ORB Protocol*) [40], uma versão do GIOP (*General Inter-ORB Protocol*) que executa sobre o protocolo TCP.

- **Interface do ORB**: oferece algumas funções que podem ser diretamente utilizadas pelas aplicações (clientes ou servidores). Exemplos: conversões de referências de objetos para *strings* e vice-versa, criação de requisições dinâmicas.

- ***Stubs* e Esqueletos**: a função destes componentes é realizar conversões (*marshalling* e *unmarshalling*) dos dados enviados ou recebidos para uma representação canônica chamada de CDR (*Common Data Representation*). Normalmente, são gerados de forma automática a partir da definição da interface do objeto CORBA.

- **Interface de Invocação Dinâmica (*Dynamic Invocation Interface*, DII)**: permite que um cliente invoque uma operação sem ter o conhecimento, em tempo de compilação, da interface (*stub*) do objeto servidor.

- **Interface de Esqueleto Dinâmica (*Dynamic Skeleton Interface*, DSI)**: função, análoga à DII, utilizada por servidores. Ou seja, um *servant* é capaz de atender requisições de clientes sem conhecer, em tempo de compilação, a interface (esqueleto) em questão.

- **Adaptador de Objeto**: responsável pela associação entre o *servant* e o ORB. O adaptador é quem identifica para qual objeto deve ser enviada a requisição do cliente e qual operação a ser executada. Outras funções do adaptador estão relacionadas com o registro e com a ativação dos *servants*.

4.2 Banco de Dados Orientado a Objetos

A partir da década de 1970, o modelo relacional passou a ser a tecnologia dominante em sistemas de gerência de banco de dados. Entretanto, aplicações mais recentes (CAD[1], banco de dados multimídia, automação de escritório, por exemplo) instituíram novos requisitos que não são atendidos, de maneira adequada, por estes bancos de dados tradicionais.

Os sistemas de gerência de banco de dados orientados a objetos (*Object-oriented Database Management System*, ODBMS) [42] procuram atender as novas exigências das aplicações, utilizando o paradigma de objetos. A principal vantagem de um banco de dados orientado a objetos é que as entidades da aplicação são modeladas e manipuladas de forma mais natural. Assim, reduz-se a "lacuna semântica" entre o domínio da aplicação e sua representação em uma armazenagem persistente.

Outra vantagem é a diminuição da "impedância" entre a linguagem de programação e o sistema de gerência de banco de dados. Em bancos de dados relacionais, geralmente as informações são recuperadas via uma linguagem de *query* específica (SQL[2], por exemplo) e manipuladas por uma linguagem procedural (C++, por exemplo). Em sistemas ODBMS, é possível recuperar dados persistentes, via navegação pelos objetos, utilizando a própria linguagem de programação[3].

Como a arquitetura TINA, incluindo o próprio DPE, é baseada em conceitos de orientação a objeto, a utilização de um ODBMS possibilita a implementação de uma plataforma mais consistente e elegante.

4.3 Arquitetura da Implementação

A arquitetura do DPE implementado é apresentada na figura 4.2. O núcleo do DPE é composto de uma plataforma CORBA e dos serviços de ciclo de vida e de *stream*[4]. Estes dois serviços propostos são uma extensão das funções básicas da infra-estrutura CORBA necessária para o suporte às aplicações TINA.

O serviço de ciclo de vida permite a distribuição e a gerência de objetos em um sistema heterogêneo, enquanto o serviço de *stream* proporciona suporte à comunicação via fluxos de mídia contínua (áudio e vídeo). Estas funções estão disponíveis às aplicações via interface do DPE. A interface do DPE representa os mecanismos de invocação padronizados (*stub* ou DII) da plataforma CORBA. Ou seja, os serviços de ciclo de vida e de *stream* correspondem a um conjunto de objetos CORBA que oferecem seus métodos por meio de interfaces definidas em IDL.

Além dos recursos básicos do NCCE (sistema operacional), os serviços do núcleo utilizam diretamente funções de protocolos de comunicação (serviço de *stream*) e de um banco de dados orientado a objetos (serviço de ciclo de vida).

Instâncias do DPE presentes em cada nó da rede são conectadas via protocolo IIOP (*Internet Inter-ORB Protocol*) da plataforma CORBA.

---

[1] *Computer-aided Design*

[2] *Structured Query Language*

[3] No caso de buscas mais complexas, o ODBMS oferece mecanismos alternativos de *query* similares ao SQL.

[4] Estas funções não devem ser confundidas com os serviços DPE descritos na seção 3.5.

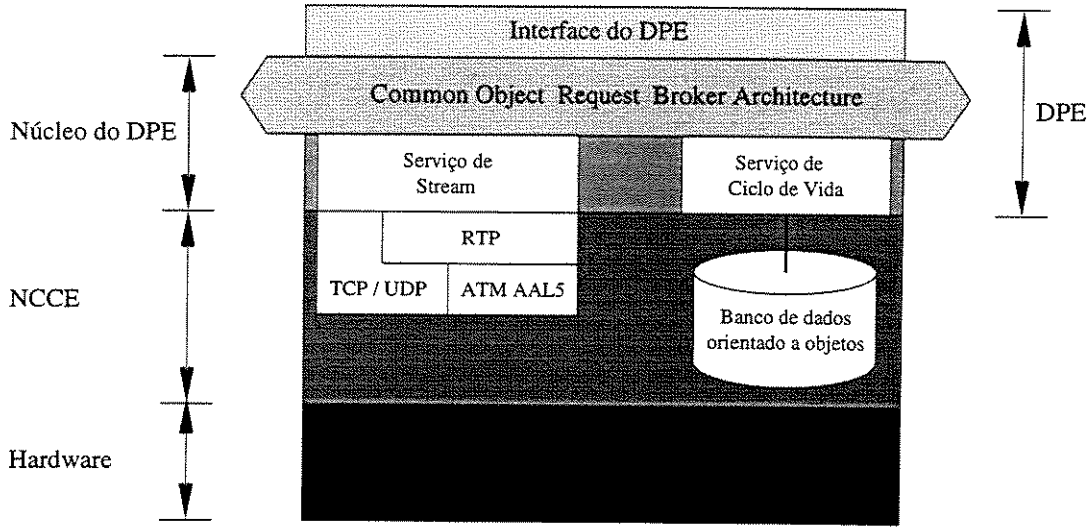

Figura 4.2: Arquitetura do DPE implementado

4.4 Serviço de Ciclo de Vida

O serviço de ciclo de vida fornece suporte aos conceitos do modelo computacional e às unidades de engenharia da arquitetura TINA. Para implementar este serviço, é necessário tratar as diferenças entre os modelos de objetos de uma especificação derivada do RM-ODP (TINA) e do consórcio OMG.

De acordo com o modelo computacional do padrão TINA (seção 2.3.3), objetos podem ter múltiplas interfaces. Tais interfaces (operacionais e *stream*) são especificadas em ODL (*Object Definition Language*). Em contrapartida, objetos CORBA oferecem apenas uma única interface (operacional) descrita em IDL (*Interface Definition Language*)[5].

Assim, para manter a compatibilidade com a especificação TINA, um objeto computacional de engenharia (eCO) com múltiplas interfaces é representado por vários objetos CORBA, onde cada objeto oferece os serviços de uma das interfaces do eCO. Em outras palavras, um objeto TINA é especificado por meio de várias interfaces IDL. Em geral, cada interface IDL é implementada por um objeto distinto. Tal abordagem segue a solução proposta por Araújo [44].

Além das interfaces definidas pelo projetista da aplicação, cada eCO possui uma interface adicional para fins de gerência (interface `eCOManager`). Os objetos que implementam a interface `eCOManager` são chamados de gerentes de eCO. A figura 4.3 mostra como um objeto TINA com três interfaces de aplicação é implementado em CORBA. Os quatro objetos CORBA são administrados como uma entidade única.

Interfaces de gerência também são definidas para *clusters*, cápsulas e nós. Objetos que oferecem estas interfaces são chamados de gerentes de *cluster*, gerentes de cápsula e gerentes de nó, respectivamente. Tais interfaces são descritas a seguir.

[5] O consórcio OMG está considerando a introdução do conceito de múltiplas interfaces em seu modelo de objetos [43].

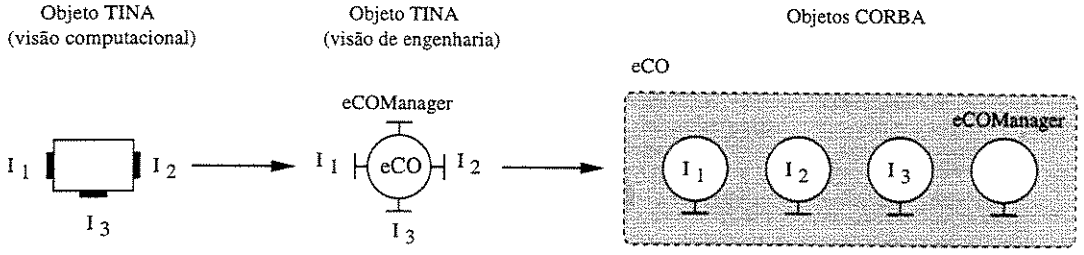

Figura 4.3: Objeto TINA e sua implementação CORBA correspondente

4.4.1 Gerência de eCOs

Como discutido anteriormente, objetos computacionais de engenharia são representados por meio de um ou mais objetos CORBA, um deles atuando como gerente e os demais proporcionando funções específicas da aplicação (figura 4.3). Além disto, todos os objetos de aplicação devem implementar algumas funções de gerência. Tais operações são invocadas pelo gerente de eCO e possuem a seguinte descrição IDL:

- `long checkpoint(in string template)`: armazena o estado do objeto CORBA em um *template*;
- `long recover(in string template)`: recupera o estado do objeto CORBA a partir de um *template*;
- `long Delete()`: destrutor do objeto CORBA, libera todos os recursos reservados ao objeto.

Normalmente, os *templates* correspondem a entradas em um banco de dados que guarda o estado dos objetos. No ambiente implementado, essa armazenagem é feita por um sistema de banco de dados orientado a objetos (figura 4.2).

A interface `eCOManager` descreve as funções de administração que podem ser realizadas em um eCO[6]:

```
typedef sequence<Object> interface_seq;
interface eCOManager {
  long addInterface(in Object interface_ref);
  long removeInterface(in Object interface_ref);
  interface_seq getInterfaces();
  long checkpoint(in string template);
  long recover(in string template);
  long Delete();
  long deactivate(in string template);
};
```

[6]A interface `eCOManager` possui outros métodos, descritos na seção 4.7, relacionados à migração de fluxos multimídia.

Interfaces de aplicação são adicionadas e removidas de um eCO via métodos `addInterface()` e `removeInterface()`, respectivamente. Estas operações têm como parâmetro de entrada a referência do objeto CORBA que oferece a interface de aplicação. O gerente mantém uma lista com as referências dos objetos adicionados ao eCO[7]. A função `getInterfaces()` retorna esta lista de componentes do eCO.

As operações de armazenagem e recuperação do estado de um eCO são realizadas pelos métodos `checkpoint()` e `recover()`, respectivamente. O parâmetro de entrada destes métodos é um *template* (*string*). O *template* não possui nenhum formato especial definido pela implementação DPE, a única exigência feita é que o mesmo *template* não pode ser utilizado em duas ou mais operações de *checkpoint* sobre um mesmo objeto.

A execução do método `checkpoint()` (`recover()`) do gerente implica na armazenagem (recuperação) do estado do gerente e na invocação do método `checkpoint()` (`recover()`), com o mesmo *template*, em cada objeto de aplicação.

Resumindo, os métodos `checkpoint()`, `recover()` e `Delete()` do gerente invocam operações correspondentes nos objetos administrados (figura 4.4)[8].

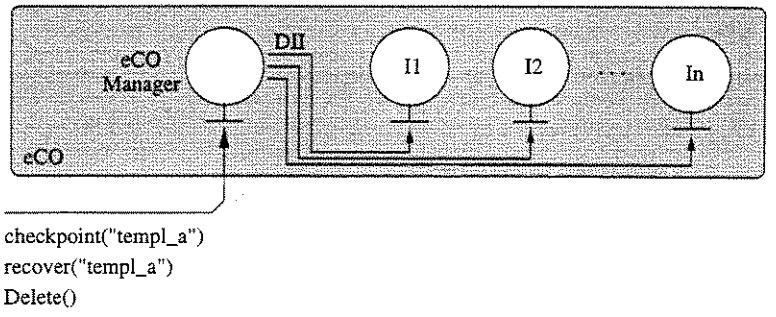

Figura 4.4: Gerência de eCOs

O gerente utiliza o mecanismo de invocação dinâmica (DII) para interagir com os demais componentes do eCO. Desta forma, o código do gerente é totalmente independente dos *stubs* dos objetos de aplicação e, portanto, não necessita ser modificado ou recompilado para realizar a comunicação com os elementos gerenciados.

No DPE implementado, a armazenagem persistente do estado dos gerentes (`eCOManager` e `ClusterManager`) é realizada por meio de um banco de dados orientado a objetos. Já para os componentes de aplicação, o projetista tem a responsabilidade de implementar o seu próprio esquema de persistência.

Desta forma, permite-se que os serviços TINA utilizem alternativas diferentes, com variados graus de complexidade, de acordo com suas necessidades (figura 4.5). Em contrapartida, o projetista acaba sendo obrigado a preocupar-se com aspectos não diretamente relacionados com sua

[7] A função `removeInterface()`, ao eliminar a referência do objeto, executa o método `Delete()` do componente removido do eCO.

[8] Como as operações de gerência não são executadas, no mesmo instante de tempo, em todos os objetos de aplicação, podem surgir inconsistências durante a armazenagem do estado das interfaces de um eCO. Para evitar tal problema, seria necessária a implementação de um protocolo de sincronização (*two-phase commit* [45], por exemplo) por parte da plataforma e dos próprios objetos de aplicação.

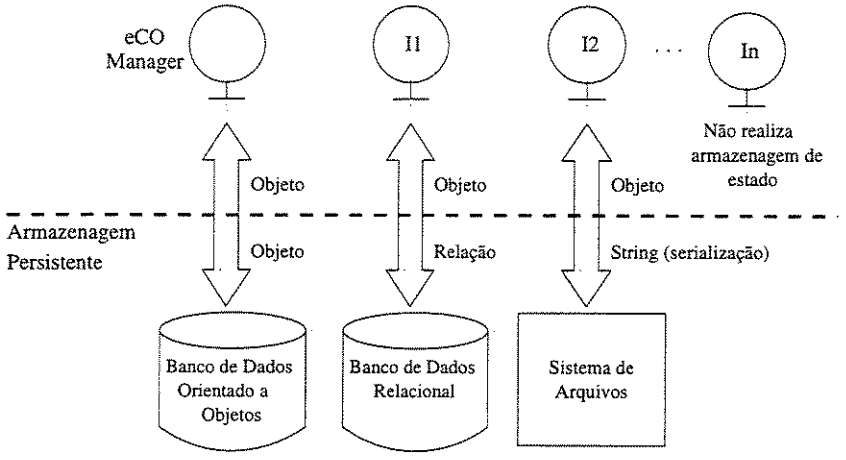

Figura 4.5: Armazenagem persistente dos componentes do eCO

aplicação. Como opção, adaptadores de objetos CORBA que suportam persistência [46] e linguagens de programação com serialização nativa (Java [47], por exemplo) poderiam possibilitar que esta tarefa fosse realizada exclusivamente pelo DPE.

O método `deactivate()` executa as operações de `checkpoint()` e `Delete()` em todos os objetos administrados pelo gerente de eCO.

4.4.2 Gerência de *Clusters*

A administração de *clusters* é proporcionada por gerentes que implementam a seguinte interface IDL:

```
typedef sequence<eCOManager> eco_seq;
interface ClusterManager {
  eCOManager makeECO(in string eco_name);
  eco_seq getECOs();
  long checkpoint(in string template);
  long recover(in string template);
  long Delete();
  long deactivate(in string template);
};
```

A operação `makeECO()` cria um novo objeto de engenharia dentro de um *cluster* e retorna a referência de seu gerente. O parâmetro de entrada é o nome pelo qual o eCO é identificado no *cluster*. O novo objeto é criado sem nenhuma interface de aplicação. Uma fábrica (interface `ManagerFactory`) assume o trabalho de instanciação dos gerentes (figura 4.6).

A interface da fábrica de gerentes é a seguinte:

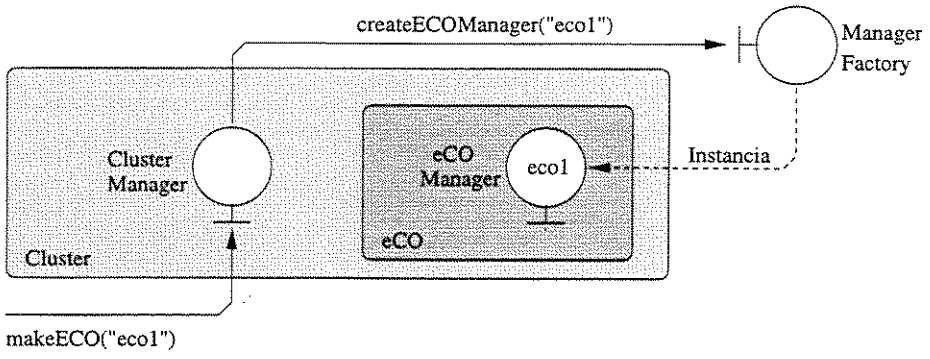

Figura 4.6: Criação de um eCO dentro de um *cluster*

```
interface ManagerFactory {
  eCOManager createECOManager(in string name);
  ClusterManager createClusterManager(in string name);
  CapsuleManager createCapsuleManager(in string name);
  NodeManager createNodeManager(in string name);
};
```

A função `getECOs()` devolve a lista com as referências dos gerentes de eCO que compõem o *cluster*.

Os métodos restantes (`checkpoint()`, `recover()`, `Delete()` e `deactivate()`) executam as funções de administração correspondentes em cada gerente de eCO pertecente ao *cluster* (figura 4.7)[9].

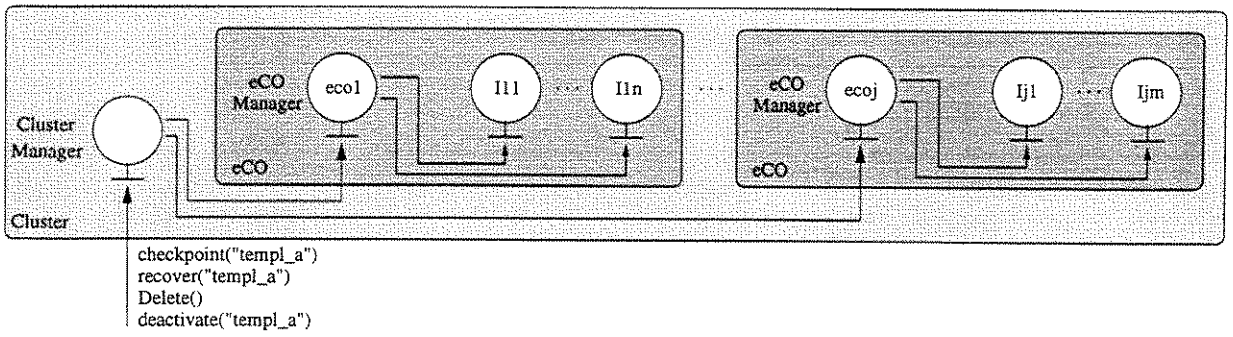

Figura 4.7: Gerência de *clusters*

A realização de migração de *clusters* é responsabilidade do gerente de cápsula (interface `CapsuleManager`).

[9]Neste caso, a invocação dos métodos dos gerentes de eCO pode ser feita via *stub*, pois o gerente de *cluster* conhece, em tempo de compilação, a interface `eCOManager`.

4.4.3 Gerência de Cápsulas

Funções de gerência de cápsulas ainda não foram especificadas pelo consórcio TINA. Assim, propôs-se um conjunto mínimo de operações considerado importante para atender os requisitos da arquitetura do DPE:

```
typedef sequence<ClusterManager> cluster_seq;
interface CapsuleManager {
  ClusterManager makeCluster(in string cluster_name);
  long reactivate(in string cluster_name, in string template);
  long migrate(in string cluster_name, in string capsule_name,
               in string node_name);
  cluster_seq getClusters();
  long Delete();
};
```

A operação `makeCluster()` cria um *cluster* vazio dentro da cápsula e retorna a referência de seu gerente (figura 4.8). O parâmetro de entrada é o nome pelo qual o *cluster* é identificado dentro da cápsula. Tal função utiliza os serviços da fábrica de gerentes (interface `ManagerFactory`).

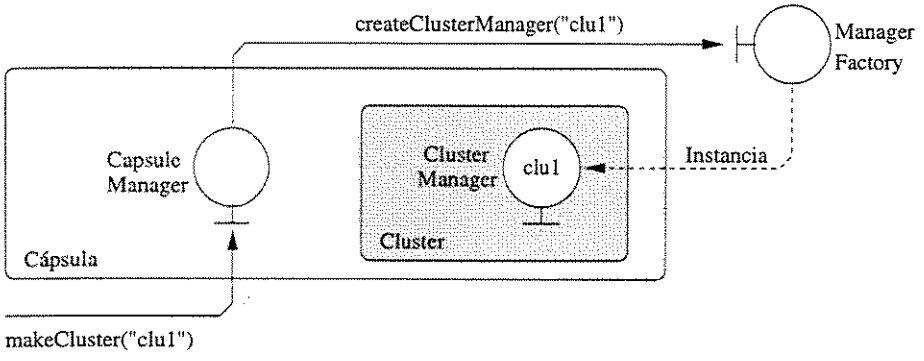

Figura 4.8: Criação de um *cluster* dentro de uma cápsula

O método `reactivate()` instancia um novo *cluster* a partir de um *template* que foi gerado durante uma operação de *checkpoint*. A figura 4.9 mostra a recuperação de um *cluster* a partir de um *template*.

Primeiramente, o gerente de cápsula invoca o método `createClusterManager()` da fábrica de gerentes e logo depois executa a operação `recover()` do objeto criado. O gerente de *cluster*, ao receber a invocação de `recover()`, cria todos os gerentes de eCOs por ele administrados (de novo, via fábrica). Por sua vez, cada gerente de eCO invoca a fábrica de objetos de aplicação (interface `AppFactory`) para criar cada uma das interfaces que compõem o eCO. Como resultado final, todos os objetos pertencentes ao *cluster* são reativados e ficam com o estado correspondente ao *template* fornecido.

A fábrica de objetos de aplicação é um componente que implementa a interface `AppFactory` (padrão *Abstract Factory* [48]):

A transparência de localização é obtida por meio de mecanismos como os serviços de localização da arquitetura CORBA. O DPE desenvolvido utiliza um *locator* proprietário [49, 50], embora um serviço padronizado (serviço de nomes [51], por exemplo) fosse o recomendado.

O suporte à transparência de recursos é garantido, pois o estado de um *cluster* sempre é armazenado antes de sua desativação. Desta forma, é possível ativar e desativar um *cluster* sem que os clientes percebam qualquer efeito deste processo.

Apesar de proporcionar funções para a movimentação dos objetos, o DPE implementado não suporta transparência de migração. Isto ocorre, pois as referências dos objetos CORBA tornam-se inválidas após a mudança de localização do *cluster*. Novos ambientes de suporte a agentes móveis [52], baseados em Java, são capazes de garantir a transparência de migração, preservando a integridade da referência do objeto CORBA. Tais facilidades poderiam ser incluídas em um DPE baseado nesta linguagem. Outra solução seria utilizar o novo adaptador de objetos da plataforma CORBA: *Portable Object Adapter* (POA)[13] [53]. O POA suporta um modo de ativação chamado de sob demanda (*on demand*), onde é possível redirecionar, transparentemente, requisições para os objetos que migraram de cápsula.

4.6 Serviço de *Stream*

O suporte ao *binding* implícito entre objetos é proporcionado diretamente pela plataforma CORBA utilizada na implementação do DPE. A função de conversão dos dados para um formato canônico é realizada pelo código dos *stubs* e esqueletos. Tais componentes são gerados automaticamente a partir da definição das interfaces IDL. O núcleo do ORB é responsável pela manutenção da integridade da conexão. Já o protocolo IIOP garante a semântica das interações durante a comunicação entre os objetos.

No DPE implementado, o *binding* explícito é oferecido pelo serviço de *stream*. Este serviço é uma extensão de uma especificação (*Control and Management of Audio/Video Streams*) do consórcio OMG [54].

4.6.1 Especificação de Controle e Gerência de A/V *Streams*

Para atender principalmente os requisitos das aplicações de telecomunicações, o consórcio OMG definiu um padrão para uma infra-estrutura distribuída de suporte a *streams* [54].

Streams representam um conjunto de fluxos contínuos e unidirecionais de mídia estruturada (áudio e vídeo, por exemplo). A figura 4.11 mostra a comunicação entre dois objetos por meio de um *stream* composto de dois fluxos de dados.

Os principais componentes da infra-estrutura proposta pelo consórcio OMG, cujas interfaces estão definidas no apêndice B, são (figura 4.12):

- **Dispositivo Multimídia** (interface `MMDevice`): representa dispositivos físicos (microfone, alto-falante, por exemplo) ou lógicos (programa que armazena o fluxo de áudio em um arquivo, por exemplo) que geram ou consomem fluxos de mídia. Dispositivos multimídia são conectados por meio de *streams*, estabelecendo um fluxo entre um produtor e um ou mais consumidores de mídia.

[13] No momento, ainda não existem implementações consolidadas deste adaptador.

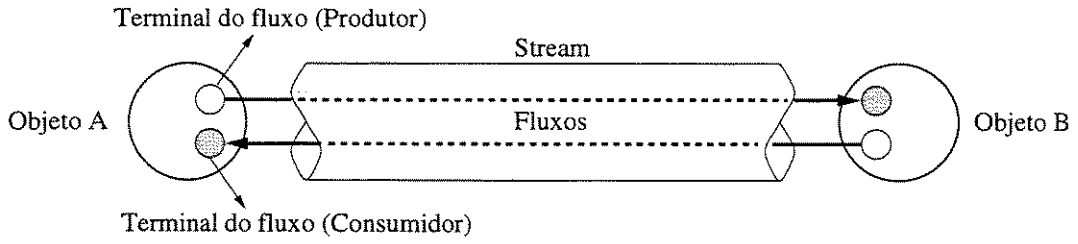

Figura 4.11: Comunicação entre objetos via *stream*

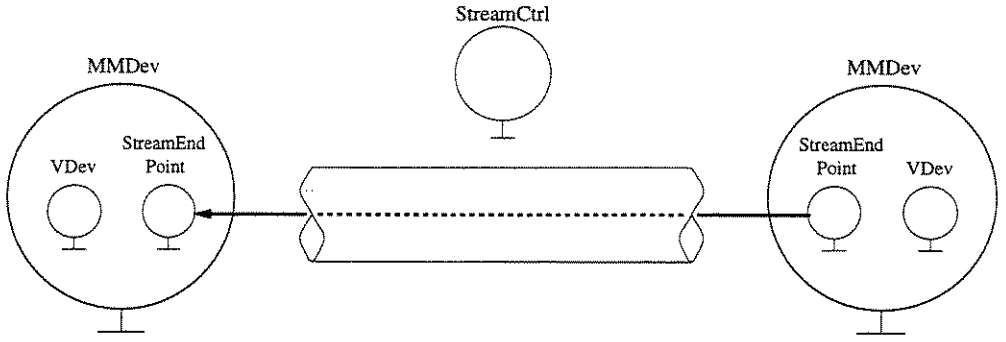

Figura 4.12: Componentes da infra-estrutura de controle e gerência de *streams*

- **Dispositivo Virtual** (interface `VDev`): é criado por um dispositivo multimídia em resposta a uma requisição para criação de uma nova conexão. Responsável pela negociação de parâmetros relacionados à configuração dos dispositivos multimídia (formato da codificação do fluxo de mídia, taxa de transferência, por exemplo).

- **Terminal de *Stream*** (interface `StreamEndPoint`): também criado pelo dispositivo multimídia durante o estabelecimento de uma nova conexão. Este componente encapsula parâmetros de transporte do fluxo (tipo do protocolo utilizado na transmissão, endereço dos nós comunicantes, qualidade de serviço, por exemplo).

- **Controlador de *Stream*** (interface `StreamCtrl`): objeto responsável pelo estabelecimento/encerramento de um *stream* entre dispositivos multimídia e pelo controle (iniciar, parar) dos fluxos presentes.

Todas estas interfaces podem ser especializadas para cada tipo de fluxo a ser transmitido (áudio, vídeo, áudio/vídeo, por exemplo).

O padrão do consórcio OMG define um protocolo específico para o transporte dos fluxos denominado SFP (*Simple Flow Protocol*). A finalidade deste protocolo é proporcionar a interoperabilidade dos terminais de *stream*[14]. Entretanto, sua implementação não é obrigatória. Assim, os terminais de *stream* podem utilizar qualquer protocolo para o transporte dos dados multimídia.

[14]Tal como o protocolo GIOP, que garante a interoperabilidade de ORBs.

Para a construção do serviço de *stream* da plataforma DPE, foi utilizada uma implementação disponível desta especificação de controle de fluxos multimídia. Tal implementação permite a transmissão dos fluxos (áudio ou vídeo) por meio dos protocolos UDP, TCP ou RTP/UDP.

4.6.2 *Binding* Explícito

Para proporcionar o suporte ao *binding* explícito na plataforma DPE, o serviço de *stream* estende, via herança, as interfaces dos dispositivos multimídia (`MMDevice`) e do controlador de *stream* (`StreamCtrl`). A figura 4.13 mostra o diagrama de classes, em UML, das extensões realizadas.

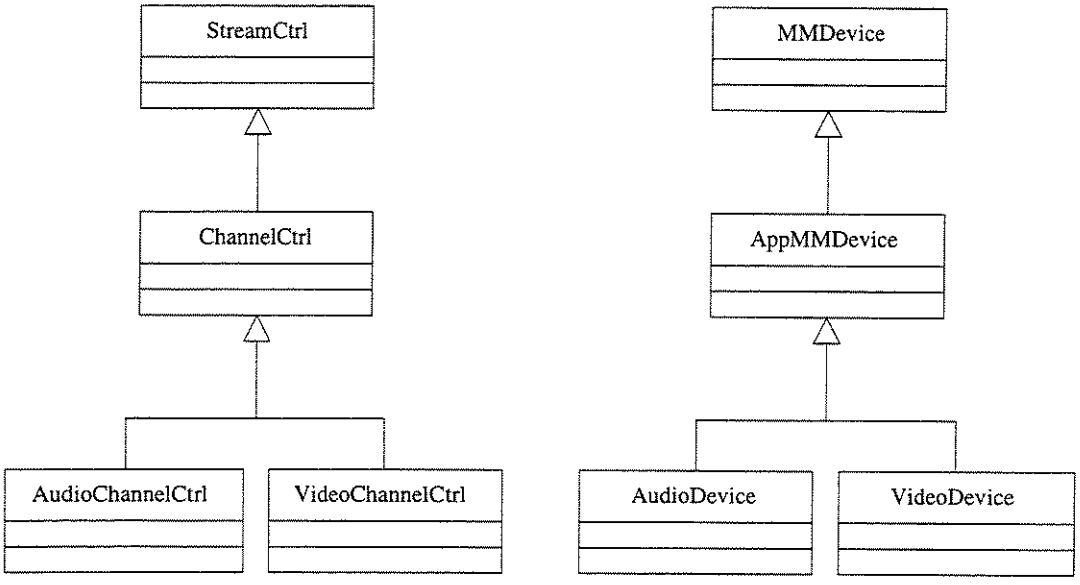

Figura 4.13: Diagrama de classes do serviço de *stream*

O controlador do canal (`ChannelCtrl`) herda todo o comportamento do objeto que implementa a interface `StreamCtrl` e adiciona algumas operações[15]:

```
interface ChannelCtrl : AVStreams::StreamCtrl {
  long configureChannel(in AVStreams::streamQos the_qos,
                        in AVStreams::flowSpec the_spec);
  long configureEndPoints(in eCOManager a_eco, in AppMMDevice a_party,
                          in eCOManager b_eco, in AppMMDevice b_party);
};
```

O método `configureChannel()` configura o canal com as informações referentes à qualidade de serviço e à especificação dos fluxos[16]. O método `configureEndPoint()` indica quais objetos (e

[15] A interface `ChannelCtrl` possui outros métodos, descritos na seção 4.7, relacionados com a migração dos fluxos.
[16] O formato destes parâmetros é definido pela especificação do consórcio OMG [54].

respectivos gerentes de eCO) participam da interação. A interface do controlador de canal pode ser especializada para cada tipo de mídia (`AudioChannelCtrl`, `VideoChannelCtrl`).

Os dispositivos multimídia também são estendidos para que possam ser administrados como uma interface de aplicação de um eCO[17]. A interface `AppMMDevice` proporciona métodos de armazenagem/recuperação do estado e de autodestruição do objeto:

```
interface AppMMDevice : AVStreams::MMDevice {
  long checkpoint(in string template);
  long recover(in string template);
  long Delete();
};
```

Os dispositivos de áudio e vídeo (`AudioDevice` e `VideoDevice`, respectivamente) são especializados a partir da interface `AppMMDevice`.

O gerente de nó é responsável pela criação e destruição do canal. Para isto, foram definidos dois métodos adicionais para a interface `NodeManager`:

```
interface NodeManager {
  long createChannel(in eCOManager a_eco, in AppMMDevice a_party,
                     in eCOManager b_eco, in AppMMDevice b_party,
                     in AVStreams::streamQos the_qos,
                     in AVStreams::flowSpec the_spec,
                     in ChannelCtrl the_ctrl);
  long destroyChannel(in ChannelCtrl the_ctrl);
};
```

Para criar um canal entre dois objetos `AppMMDevice` localizados em *clusters* distintos, o cliente deve obter as referências[18] dos dispositivos multimídia, dos gerentes de eCO correspondentes, do controlador de canal, definir a qualidade de serviço (`the_qos`) e a especificação dos fluxos (`the_spec`) e invocar `createChannel()`. O gerente de nó se encarrega de executar as funções de configuração do canal (`configureChannel()` e `configureEndPoints()`). Após criado o *binding*, o cliente controla os fluxos (iniciar, parar) via `ChannelCtrl`.

O método `destroyChannel()` destrói o canal controlado pelo objeto definido como parâmetro de entrada, liberando todos os recursos reservados aos fluxos daquela conexão.

A figura 4.14 mostra um canal com um fluxo de áudio conectando dois dispositivos (microfone e alto-falante) que estão em *cluster* distintos.

[17] Os objetos `VDev` e `StreamEndPoint` são gerenciados (criação/destruição) diretamente pelo dispositivo multimídia. Assim, achou-se desnecessário especializar tais interfaces para fins de administração do gerente de eCO.

[18] Via serviço de diretório, por exemplo.

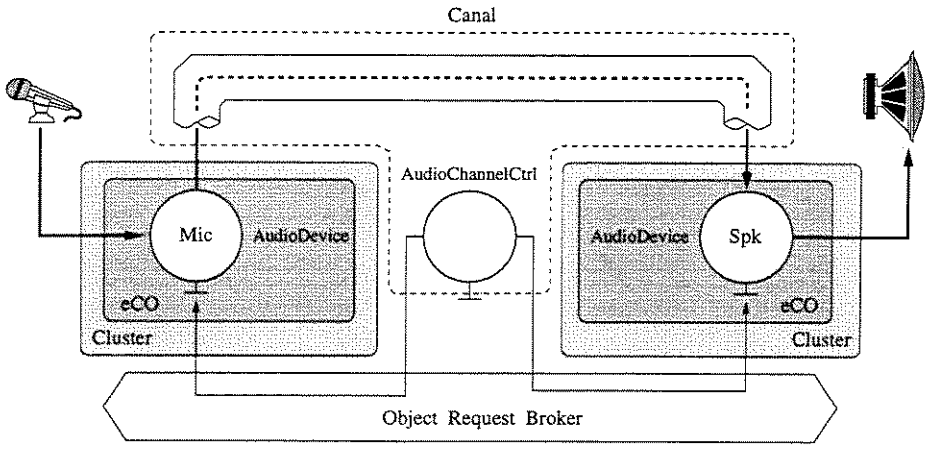

Figura 4.14: Canal de áudio oferecido pelo serviço de *stream*

4.7 Mobilidade de *Streams* de Áudio e Vídeo

A implementação do DPE permite não somente a movimentação de *clusters* como também a migração automática dos canais associados aos objetos. Para isto, é necessária a integração do serviço de *stream* com o serviço de ciclo de vida.

Para proporcionar esta facilidade, o serviço de ciclo de vida oferece funções extras. Cada gerente de eCO guarda uma lista com as referências de todos os controladores de canal que controlam os fluxos transmitidos/enviados pelo eCO. Para isto, ao criar o canal, o gerente de nó executa um método adicional do gerente de eCO, `addChannel()`, informando a existência de um novo canal. Uma função análoga, `deleteChannel()`, é invocada durante a destruição da conexão.

```
interface eCOManager {
  long addChannel(in ChannelCtrl the_ctrl);
  long deleteChannel(in ChannelCtrl the_ctrl);
};
```

O serviço de *stream* também possui outras atribuições. O controlador de canal armazena todas as informações necessárias para o restabelecimento da conexão (especificação dos fluxos, parâmetros de qualidade de serviço e referências dos objetos comunicantes) e oferece quatro operações adicionais:

```
interface ChannelCtrl : AVStreams::StreamCtrl {
  long deactivateChannel();
  long reactivateChannel();
  long changeEndPoint(in eCOManager eco, AppMMDevice party);
  long changeNode(in string node_name);
};
```

O método `deactivateChannel()` destrói todos os fluxos pertencentes ao canal mas mantém as informações, nos gerentes de eCO e controladores de canal, necessárias para uma possível reativação do *binding*.

A reativação do canal é responsabilidade da função `reactivateChannel()`. A conexão é restabelecida a partir das informações presentes no controlador. Entretanto, dois parâmetros alteram-se no processo de migração: as referências dos objetos CORBA (gerente de eCO e dispositivo multimídia) que mudaram de local. Para atualizar estas informações, durante a reativação do eCO, o gerente executa o método `changeEndPoint()` do controlador, comunicando as novas referências dos objetos em questão. Outra modificação pode acontecer durante a migração, caso o *cluster* mude de nó. Neste caso, o endereço de rede do terminal de fluxo altera-se[19]. Assim, durante o processo de reativação, o gerente de eCO também sempre invoca o método `changeNode()`, indicando sua (nova) localização na rede.

Resumindo, o processo de migração do *binding* explícito é mostrado no diagrama de seqüência a seguir (figura 4.15)[20]. Durante o processo de desativação do eCO, o gerente invoca o método `deactivateChannel()` dos controladores, a fim de liberar os recursos de comunicação alocados para cada fluxo. Na cápsula destino, no momento da reativação do eCO, o gerente informa sua nova localização, via métodos `changeEndPoint()` e `changeNode()`, e reativa o canal (`reactivateChannel()`).

Assim, a tarefa de migração dos canais é de inteira responsabilidade do DPE. Ou seja, o projetista não precisa se preocupar em guardar informações (qualidade de serviço e especificação dos fluxos) sobre os canais existentes e nem executar as funções de destruição (`destroyChannel()`) ou de estabelecimento de conexão (`createChannel()`) durante os processos de desativação/reativação dos objetos de aplicação.

Finalmente, é importante observar que o controlador de canal não migra e sua localização não interfere na migração do *cluster* (figura 4.16). O controlador pode ou não estar em uma das cápsulas que contém os objetos comunicantes.

No projeto do DPE, considerou-se desnecessário o suporte automático à migração do controlador de canal. Caso alguma aplicação precise deste recurso, o projetista deve estender a interface `ChannelCtrl` para proporcionar métodos de armazenagem/recuperação do estado do controlador e de autodestruição. Desta forma, o controlador torna-se um objeto de aplicação comum, gerenciável, e pode migrar normalmente como os demais componentes do *cluster*.

[19]Na verdade, isto não ocorre caso o terminal de fluxo esteja utilizando um endereço *multicast* ou exista suporte à mobilidade no nível de camada de rede (IP móvel [55, 56], por exemplo).

[20]Para simplificar o diagrama, não são representadas as fábricas de gerente e de aplicação.

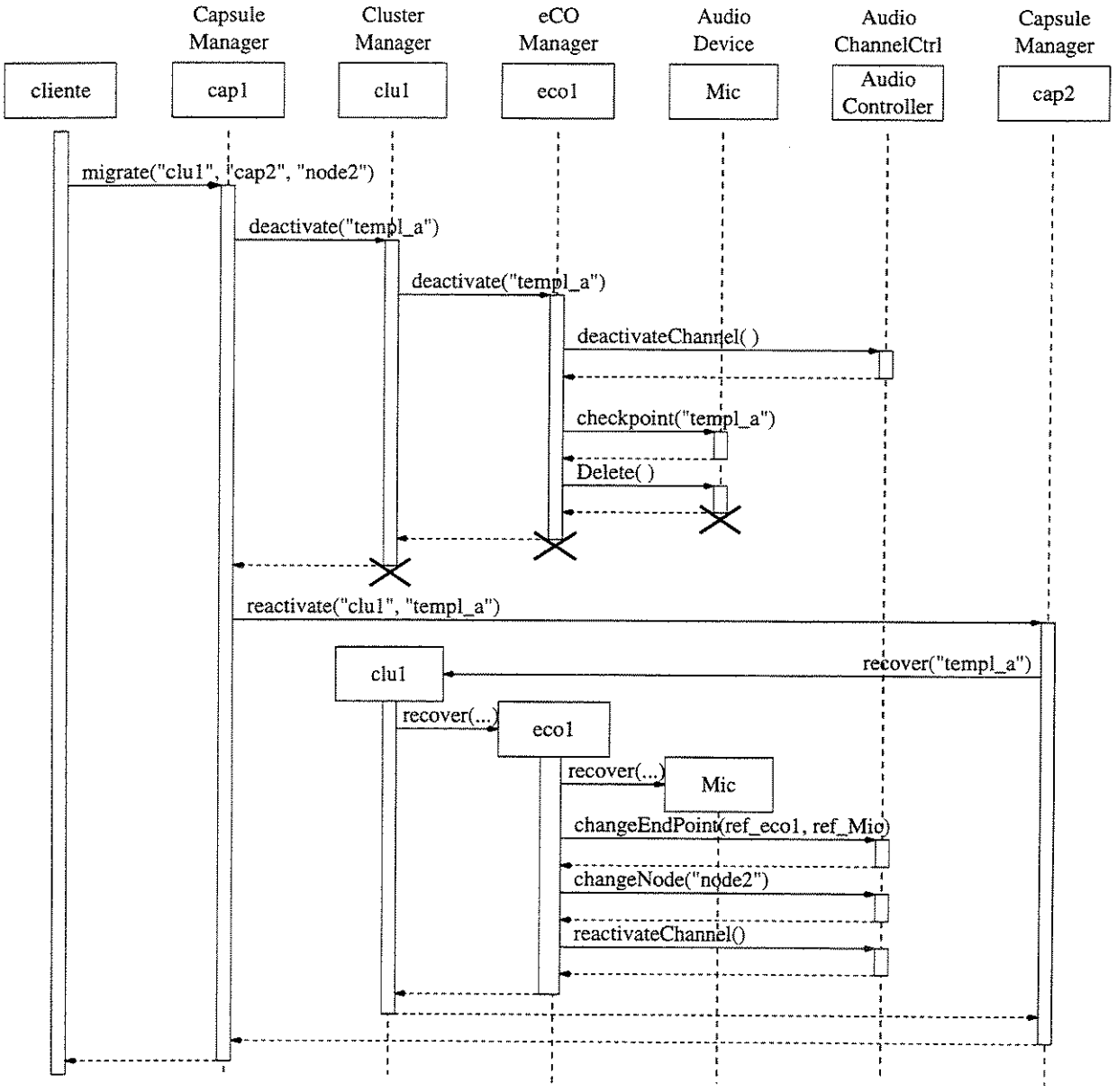

Figura 4.15: Migração de um canal

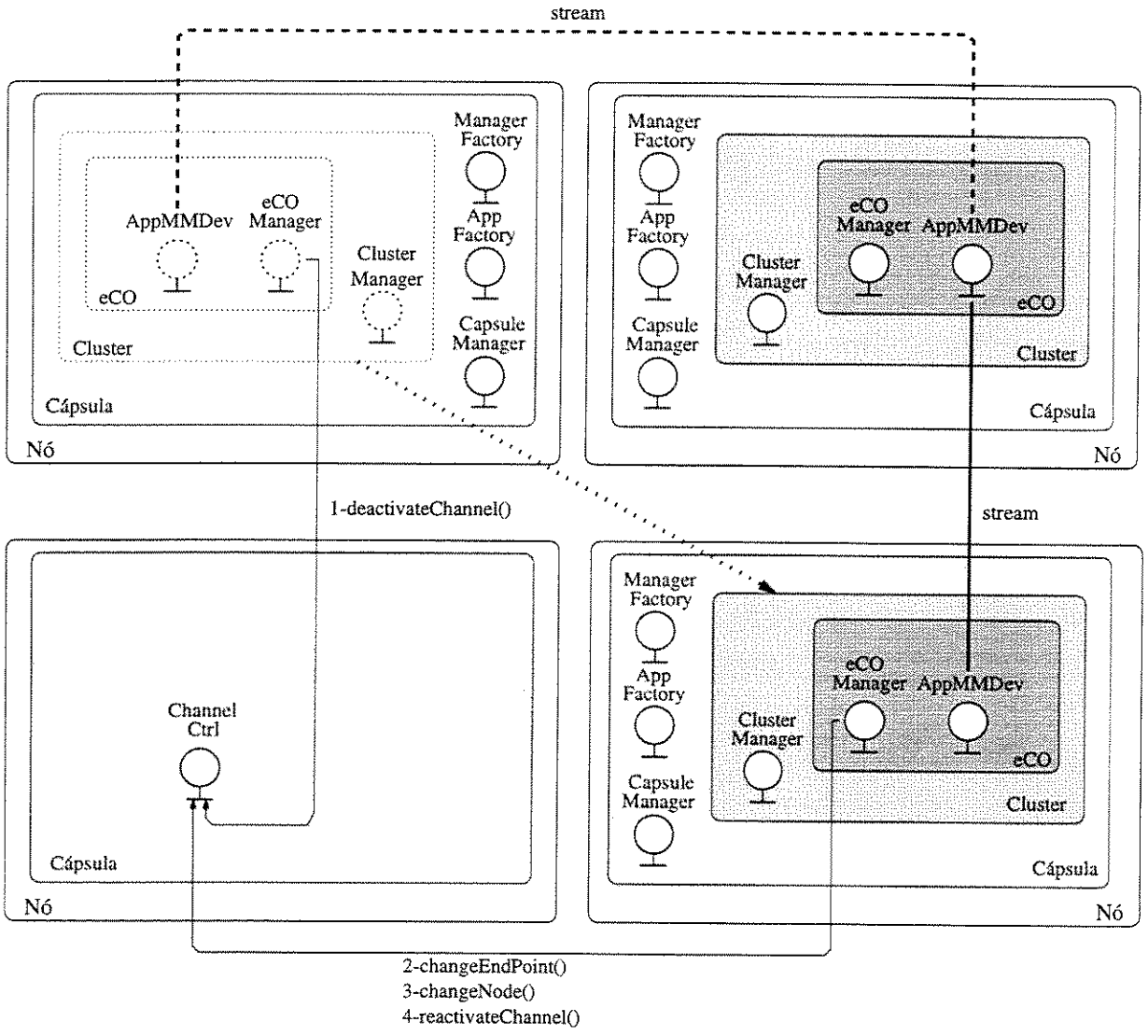

Figura 4.16: Migração de um *cluster* com terminação de canal

4.8 Serviços DPE

O consórcio TINA ainda não definiu os requisitos de cada um dos serviços DPE. Enquanto isto não é feito, a solução mais natural é utilizar implementações disponíveis dos serviços comuns (eventos, *trading*, nomes, segurança, entre outros) da arquitetura do consórcio OMG [51].

A implementação atual do DPE não oferece nenhum serviço comum às aplicações TINA.

4.9 Aspectos Não-Funcionais

A especificação que descreve as propriedades e qualidades de serviço de uma plataforma DPE também é bastante geral. Assim, não é possível fazer uma análise mais rigorosa das características não-funcionais de uma implementação do DPE.

De qualquer maneira, um aspecto interessante a ser citado é a característica *multithreaded* da plataforma implementada. Como o padrão CORBA não prescreve um modelo de concorrência[21], o DPE utiliza uma solução proprietária [49, 50].

Esta abordagem, conhecida como filtro (padrão *Chain of Responsability* [48]), permite a definição de vários modelos de concorrência (*pool* de *threads*, *thread* por objeto, *thread* por operação).

Na implementação realizada, o modelo de concorrência escolhido para os objetos gerentes (eCO, *cluster* e cápsula) é o de *thread* por objeto, similar ao descrito por Schmidt [58][22]. Neste caso, cada *servant* possui uma *thread* exclusiva de execução, mas continua a atender apenas uma requisição por vez.

A solução via filtros proporciona flexibilidade ao projeto dos serviços TINA. Objetos de aplicação podem utilizar o mesmo modelo de concorrência dos gerentes ou optar por outras abordagens mais convenientes (figura 4.17).

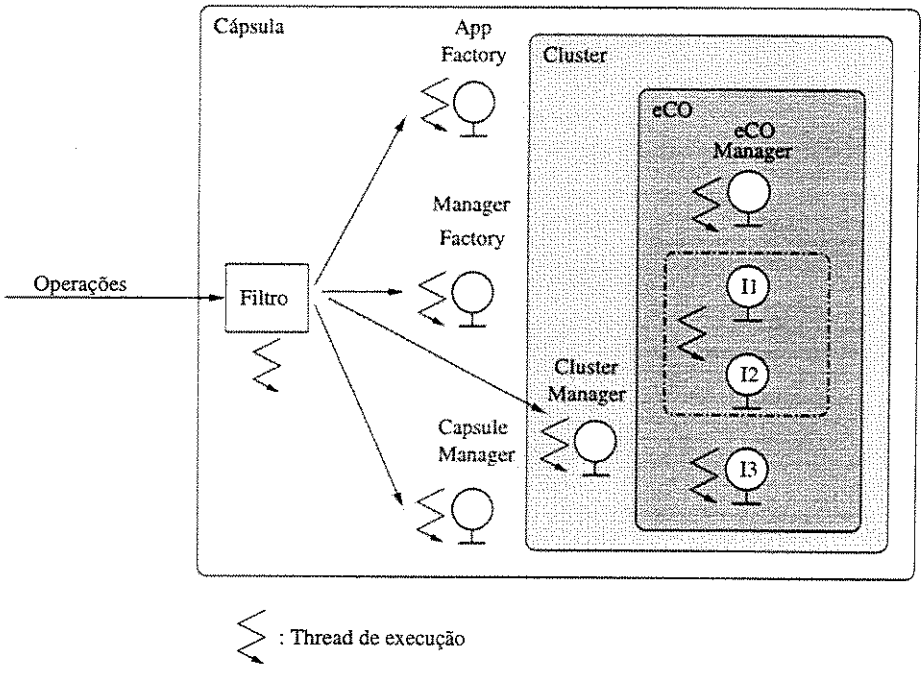

Figura 4.17: Característica *multithreaded* do DPE implementado

[21] Na verdade, a especificação mais recente (versão 2.2) aborda este assunto: o novo tipo de adaptador de objetos (*Portable Object Adapter*, POA) oferece mecanismos de concorrência [57].

[22] Schmidt denominou este modelo de *thread* por sessão.

4.10 Trabalhos Correlatos

Esta seção descreve brevemente alguns outros sistemas de *software* que podem ser úteis para o desenvolvimento de aplicações de telecomunicações. Nesta discussão, procurou-se enfatizar as funções de comunicação e de mobilidade oferecidas pelas plataformas.

Dos trabalhos apresentados, apenas o primeiro projeto, o ReTINA, segue as especificações TINA. Os demais sistemas utilizam outros padrões abertos ou soluções proprietárias. Tais iniciativas foram divididas em dois grupos: ambientes de suporte à mobilidade de aplicações e plataformas para comunicação multimídia.

4.10.1 ReTINA

O projeto ReTINA [11], conduzido sob os auspícios do programa europeu ACTS, tem como objetivo principal o desenvolvimento de um ambiente para processamento distribuído (DPE) aderente à arquitetura TINA. A meta é prover uma infra-estrutura de qualidade industrial, isto é, que possua as características necessárias, em termos de desempenho e garantias de qualidade de serviço, para o oferecimento, em larga escala, de serviços multimídia interativos. Entre os participantes do projeto, encontram-se: Alcatel, France Telecom, Siemens, HP, Chorus Systems, GMD Fokus e Universidade de Lancaster.

O DPE-ReTINA também foi projetado para atender as especificações do consórcio OMG, ou seja, é CORBA-*compliant*. Porém, tal iniciativa é mais ambiciosa que o DPE proposto nesta dissertação de mestrado, pois pretende estender os padrões do OMG para incluir o suporte a requisitos de aplicações TINA[23]. Esta extensão aborda aspectos como:

- suporte aos conceitos de modularidade do modelo computacional (grupos de objetos, contratos);
- ajuste fino sobre as políticas de alocação e escalonamento de recursos, de forma a garantir requisitos de qualidade de serviço (ORB de tempo-real);
- comunicação via fluxos multimídia (suporte a *bindings* explícitos, inclusão do tipo *stream* à linguagem IDL, entre outras extensões);
- encapsulação de protocolos alternativos (de sinalização, de gerência e de tempo-real, por exemplo) dentro do ORB.

Assim, existe uma colaboração por parte dos participantes do ReTINA no processo de elaboração de padrões do OMG, especialmente na especificação de um ORB que atenda as necessidades de aplicações de tempo-real [59].

Com relação aos serviços DPE, a estratégia do ReTINA é especializar os atuais serviços comuns do OMG. Os serviços de notificação, *trading*, transação, persistência e de *query* estão sendo refinados e integrados ao DPE-ReTINA.

Para realizar o suporte à comunicação via fluxos multimídia, existem estudos sobre a conveniência de se utilizar o serviço de A/V *Streams* do OMG como uma solução alternativa. Esta abordagem evitaria a necessidade de se efetuar algumas modificações sobre padrões CORBA consolidados. O serviço de *stream* (seção 4.6) é uma demonstração de como tal serviço do OMG pode ser adaptado ao DPE.

[23] Além de influenciar as próprias especificações do DPE.

O ReTINA também utiliza um ODBMS para prover persistência aos objetos. Aparentemente, o projeto não trata de aspectos relacionados com o suporte do DPE à migração dos objetos de aplicação.

Além de projetar e implementar um ambiente DPE, o ReTINA deve também apresentar um conjunto de ferramentas de auxílio ao desenvolvimento de serviços TINA e uma série de aplicações experimentais construídas sobre o DPE.

4.10.2 Ambientes de Suporte à Mobilidade de Aplicações

Vários sistemas de *software* se propõem a oferecer funções de mobilidade às aplicações. Esta seção trata de uma classe específica de tais plataformas, os sistemas de suporte a agentes móveis (SBAM) [60].

O termo agente é utilizado por diferentes áreas, como inteligência artificial e engenharia de *software*, e, assim, possui significados distintos. Em sistemas distribuídos, o conceito de agentes refere-se a programas autônomos que executam tarefas em benefício de seus proprietários. Este paradigma de desenvolvimento enfatiza as características de cooperação, inteligência e mobilidade dos componentes de um sistema distribuído e é uma alternativa ao tradicional modelo cliente/servidor.

Os sistemas de suporte a agentes móveis têm como objetivo permitir a migração de agentes por uma rede potencialmente heterogênea. O surgimento da linguagem Java [47] causou um grande impacto sobre a evolução destas plataformas. Aspectos como suporte nativo a *multithreading* e serialização de objetos, além da portabilidade de código, tornam a tecnologia Java bastante adequada para a construção de SBAMs. Assim, a maioria dos atuais projetos de suporte a agentes móveis empregam Java como linguagem de implementação.

Além de basearem-se em Java, as plataformas apresentadas a seguir possuem outra característica importante: os agentes podem ser implementados como objetos CORBA. Desta forma, tal como no DPE-TINA construído, alia-se a capacidade de movimentação dos objetos com as vantagens da infra-estrutura proposta pelo consórcio OMG.

Três sistemas são descritos nesta seção. O primeiro é uma iniciativa acadêmica e os demais são produtos disponíveis comercialmente.

No Instituto de Computação da Unicamp, Vasconcellos [61] desenvolveu um ambiente de suporte a agentes móveis sobre uma plataforma CORBA. Neste sistema, agentes são implementados como uma especialização de um classe básica (`BaseAgent`) que possui uma interface IDL (`Agent`). Esta classe possui métodos como `run()` (código a ser executado logo após a migração do objeto) e `set_itinerary()` (determina o itinerário a ser percorrido pelo agente). Objetos representando agências (interface `MAF1`) são responsáveis pela criação dos agentes.

O processo de migração dos objetos ocorre da seguinte maneira. Primeiramente, os agentes armazenam todo o seu estado em um vetor de *bytes*, via serialização (método `save()` da classe `BaseAgent`). Esta seqüência de octetos é enviada à agência receptora (interface `MAF2`), associada ao nó destino, como argumento de um método (`receive_agent()`). A agência `MAF2` encarrega-se de realizar a instanciação e recuperação do estado dos agentes no novo local. Assim, o transporte dos objetos ocorre por meio do próprio protocolo IIOP.

Ao chegarem em um outro nó, os agentes sempre informam à sua agência "natal" a sua nova referência CORBA. Desta forma, aplicações clientes podem obter, via agência `MAF1`, referências atuais (e válidas) dos agentes.

A tecnologia Voyager [62], desenvolvida pela ObjectSpace, é uma infra-estrutura destinada à construção de aplicações distribuídas e que possui suporte à migração de objetos.

Voyager contém um ORB proprietário que proporciona transparências de acesso e localização a objetos implementados em Java. Apesar deste *broker* não atender a especificação CORBA, a plataforma possui características semelhantes a aquelas definidas pelo consórcio OMG, como mecanismos de invocação estática e dinâmica. Voyager também inclui alguns serviços básicos (persistência, eventos, diretório) que podem ser úteis às aplicações.

Objetos Voyager são localizados por meio de um identificador global único de 16 *bytes*. Existe um serviço de diretório, que permite a definição de um *alias* para o objeto. Este nome atribuído ao objeto Voyager pode ser utilizado, posteriormente, para recuperar sua referência. Clientes executam operações em objetos Voyager, remotos ou não, utilizando esta referência "virtual".

Qualquer objeto Java (serializável) pode ser transferido de uma cápsula para outra por meio da execução do método `moveTo()`, implementado pela infra-estrutura. O parâmetro de entrada desta função corresponde à identificação da cápsula destino[24]. A plataforma se encarrega de garantir que o objeto atenda todas as requisições pendentes antes de movê-lo para a nova localização.

A transparência de migração de objetos Voyager também é garantida. Ao mover um objeto, a infra-estrutura deixa em seu lugar um representante (*forwarder*) responsável pelo redirecionamento das requisições para o objeto alvo. Assim, clientes que possuem referências "antigas" ainda conseguem executar funções do objeto em questão[25].

Voyager proporciona uma integração bidirecional com ORBs aderentes à especificação CORBA. Existem funções de conversão de referências de objetos CORBA (IORs) para referências de objetos Voyager e vice-versa. Além disto, a plataforma transforma automaticamente as referências Voyager, quando elas são enviadas (via parâmetros de métodos, por exemplo) a um *broker* CORBA, e referências IOR que chegam ao ambiente Voyager. Assim, de maneira transparente, um cliente Voyager pode executar funções de um objeto CORBA e aplicações CORBA podem invocar servidores Voyager.

A plataforma Jumping Beans [52], desenvolvida pela Ad Astra, também se propõe a realizar a migração de objetos Java. Entretanto, possui mecanismos mais eficientes, em relação aos dois sistemas apresentados anteriormente, que conferem maior segurança (autenticação via certificados digitais e chaves públicas/privadas) e robustez (funções de *store-and-forward*, entrega garantida e persistência) ao ambiente.

Outra característica interessante do produto da Ad Astra é a sua capacidade de manter a integridade das referências dos objetos CORBA que migram[26]. Ou seja, clientes que possuiam referências "antigas" podem executar transparentemente funções, via os mecanismos padronizados, dos objetos CORBA que migraram.

---

[24]No ambiente Voyager, as cápsulas são representadas pelo nome do nó onde residem e por um número de porta.

[25]Na verdade, os clientes utilizarão o *forwarder* apenas uma vez. Depois da primeira invocação, a infra-estrutura é capaz de atualizar automaticamente a referência virtual que o cliente possui. Assim, requisições subseqüentes já são enviadas diretamente ao objeto em sua nova cápsula.

[26]De acordo com a documentação do produto, apenas interfaces IDL específicas podem receber o suporte à transparência de migração. No futuro, espera-se estender tal facilidade a qualquer objeto CORBA genérico.

4.10.3 Plataformas para Comunicação Multimídia

Atualmente, existem várias infra-estruturas de suporte ao estabelecimento e controle de tráfego de dados multimídia (áudio e vídeo, principalmente). Esta seção descreve sistemas que apresentam algumas das características mais comuns encontradas em tais plataformas.

Prado [63] realizou a implementação de um serviço de comunicação, aderente ao padrão RM-ODP, utilizando a plataforma CORBA. Os conceitos do modelo de referência (*stub*, *binder*, adaptador de protocolo, controlador e fábrica de canal) são representados como objetos CORBA e, assim, oferecem às aplicações clientes suas funções via interfaces IDL. O transporte dos dados multimídia é feito por meio dos protocolos RTP e UDP.

O ambiente Maestro [64], desenvolvido na Universidade de Ciência e Tecnologia de Pohang (Coréia do Sul), proporciona suporte a aplicações multimídia colaborativas, como vídeo-conferência e ensino a distância.

A arquitetura do ambiente Maestro é dividida em camadas. Na camada mais superior, residem os serviços colaborativos, que utilizam funções de uma interface de programação (*Multimedia Collaborative Service API*). Esta API, por sua vez, usa vários outros componentes (serviços de comunicação, gerência de sessão, segurança, diretório e persistência, entre outros) da camada, imediatamente inferior, de serviços de multimídia distribuída. Uma plataforma CORBA é empregada para realizar a implementação das aplicações e dos serviços de multimídia distribuída.

A interface de programação do Maestro define um conjunto de objetos que representam dispositivos de mídia (produtores e consumidores). O estabelecimento e controle dos fluxos é realizado pela camada de serviços de multimídia distribuída. O ambiente proporciona comunicação ponto a ponto e multiponto (um para muitos e muitos para muitos). Entretanto, não existe suporte, por parte da infra-estrutura, à movimentação dos objetos das aplicações colaborativas.

Por fim, outros ambientes foram projetados especificamente para transmitir fluxos multimídia em uma arquitetura Internet, como o RealVideo e o Mbone.

O RealVideo [65], desenvolvido pela RealNetworks, é um dos sistemas de comunicação mais populares e utiliza os protocolos TCP ou UDP para transportar fluxos de áudio e vídeo. Tal infra-estrutura também suporta tráfego multiponto (via IP *multicast*). Entretanto, as funções de estabelecimento e controle dos fluxos e os algoritmos de compressão dos dados de áudio e vídeo desta plataforma são proprietários.

Já o Mbone (*Multicast Backbone*) [4], uma solução aberta desenvolvida e testada na rede de pesquisa DARTnet, é uma rede virtual construída sobre a Internet e que utiliza o protocolo IP *multicast* para a difusão dos dados. Este ambiente é adequado para aplicações que envolvam múltiplos usuários, como ensino a distância. Produtores de dados enviam seus pacotes a um endereço (de grupo) alvo e consumidores expressam seu interesse em receber o tráfego destinado a aquele grupo. Atualmente, existem várias aplicações que fazem uso desta infra-estrutura, como o *Visual Audio Tool* - VAT [66], por exemplo.

Capítulo 5

Aspectos de Implementação Específicos

Este capítulo trata de detalhes relacionados com a implementação de um DPE específico. São apresentadas as duas principais ferramentas de *software* utilizadas para a construção de tal ambiente. Além disto, descreve-se a plataforma de computação e comunicação disponível durante o desenvolvimento e testes do DPE. Finalmente, mostra-se um protótipo de aplicação TINA que faz uso dos serviços de ciclo de vida e de *stream* presentes no DPE.

5.1 Orbix

A ferramenta Orbix [49, 50], da Iona Technologies, é um produto que atende todos os requisitos da especificação CORBA 2.0 e suporta o mapeamento padrão IDL/C++. Este ORB é implementado como um par de bibliotecas, uma deve ser ligada ao código da aplicação que atua como cliente e a outra ao código da aplicação servidora. Desta forma, a invocação de uma operação é transmitida diretamente do processo cliente para o servidor.

Além das duas bibliotecas citadas no parágrafo anterior, Orbix apresenta um utilitário (*daemon*), chamado **orbixd**, responsável pela (re)ativação dos processos servidores. Este *daemon* utiliza uma base de dados simples, denominada repositório de implementação, que contém informações necessárias para iniciar a execução das aplicações servidoras. O repositório de implementação armazena, por exemplo, o nome do código executável associado ao servidor e o modo de ativação da aplicação (compartilhado ou não). Além disto, **orbixd** oferece às aplicações clientes os dados necessários para o estabelecimento de conexões com o processo servidor.

A ferramenta possui, ainda, mecanismos, não previstos no padrão CORBA, que auxiliam o desenvolvimento de aplicações distribuídas. Filtros, *locators*, *loaders* e *smart proxies* fazem parte desta extensão proprietária.

Orbix está disponível em mais de 20 sistemas operacionais, incluindo Unix (Solaris, HP/UX, AIX, SCO, IRIX, entre outros), Windows 95/98, Windows NT, OS/2, VMS, MVS, QNX, VxWorks e Macintosh.

Na implementação do DPE, utilizou-se a versão 2.3 do produto (com suporte a *multithreading*).

5.2 ObjectStore

Para a construção do DPE, a ferramenta ObjectStore [67], produzida pela Object Design, foi escolhida como sistema de gerência de banco de dados. ObjectStore é um banco de dados orientado a objetos (ODBMS), com características multi-cliente/multi-servidor. Tal produto possui interfaces de programação disponíveis nas linguagens C, C++, Smalltalk e Java.

Neste ODBMS, a base de dados é vista como uma extensão da memória do processo. A transferência de dados entre a armazenagem persistente e a memória alocada ao programa é realizada de forma automática e totalmente transparente à aplicação. ObjectStore é capaz de detectar qualquer referência (apontador em C++, por exemplo) a dados armazenados no banco de dados. Neste caso, a página (bloco de memória) que contém o dado em questão é copiada para o *cache* da aplicação. Finalizado o mapeamento destas informações para a memória virtual do processo, o acesso aos dados "persistentes" torna-se tão rápido quanto o acesso aos dados comuns ("transientes").

A ferramenta é composta de dois tipos de processos: **ObjectStore Server** e **Cache Manager**.

ObjectStore Server gerencia o acesso ao banco de dados, incluindo a armazenagem e a recuperação dos dados persistentes. Além disto, este processo é responsável pela detecção de *deadlocks* e pelo registro de *log* e *backup* do banco de dados. Existe um **ObjectStore Server** para cada sistema de arquivos que compõe o banco de dados.

Cache Manager é ativado automaticamente sempre que uma aplicação de acesso ao banco de dados é iniciada. Este processo executa no mesmo nó onde reside a aplicação. Sua função é administrar o *cache* da aplicação, que é uma área de memória local reservada para os dados mapeados ou à espera de mapeamento. Se outras aplicações forem executadas no mesmo nó, o mesmo **Cache Manager** gerencia os *caches* destas aplicações.

A figura 5.1 ilustra a localização destes processos em um banco de dados distribuído.

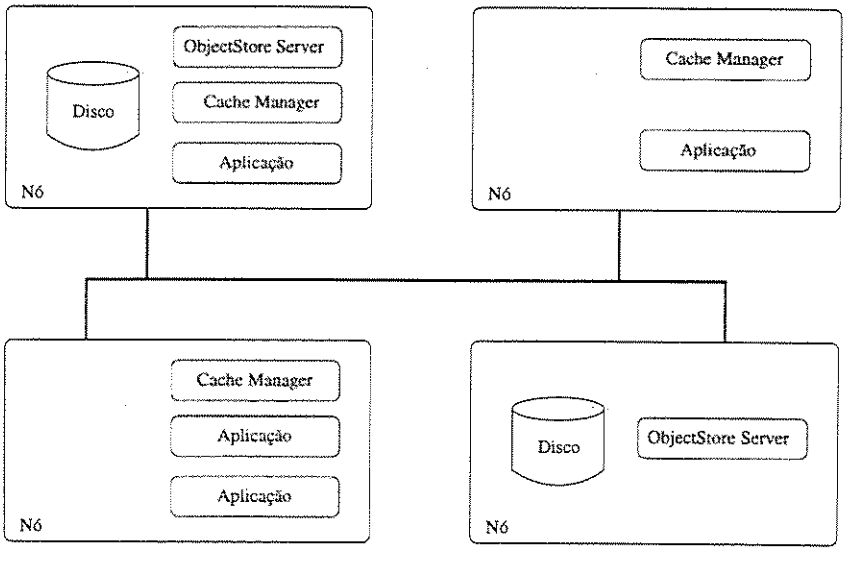

Figura 5.1: Localização dos processos do ObjectStore

A interface de programação do ODBMS inclui funções de gerência do banco de dados (criação, abertura, fechamento, destruição), de configuração de parâmetros do sistema (tamanho do *cache* da aplicação, por exemplo) e de demarcação de transações. Além disto, ObjectStore oferece uma biblioteca de classes (*templates*, em C++) que podem auxiliar o desenvolvimento das aplicações. Dicionário, conjunto, lista e vetor são exemplos de objetos disponíveis para o programador.

A versão 5.0 da ferramenta, com interface de programação em C++, foi empregada para realizar a construção do DPE.

5.3 Infra-estrutura de Computação e Comunicação

O DPE foi implementado e testado sobre a seguinte infra-estrutura de computação e comunicação. Os nós computacionais consistem de estações de trabalho SPARC 5 e Ultra 1, da Sun, com sistema operacional Solaris (versão 2.6) e compilador C++ SPARCCompiler (versão 4.1). Cada nó possui uma interface de rede ATM (OC3, 155Mbits/s) e é conectado à rede de transporte por meio de um comutador da Xylan. A figura 5.2 mostra este ambiente de desenvolvimento.

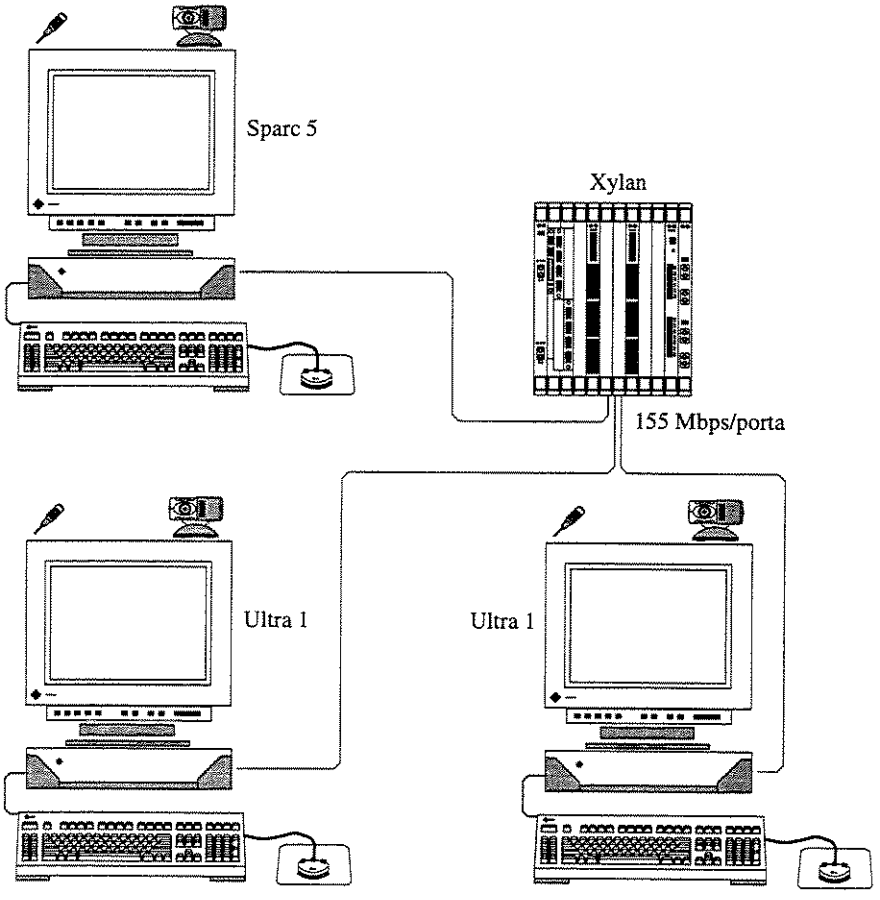

Figura 5.2: Plataforma de computação e comunicação

5.4 Cenário de Utilização

Os serviços do DPE (ciclo de vida e *stream*) foram implementados na forma de bibliotecas que devem ser ligadas ao código dos objetos de aplicação. O serviço de ciclo de vida oferece duas bibliotecas, uma deve ser ligada ao código que contém os gerentes das unidades de engenharia e a outra às aplicações clientes que executam operações dos gerentes. Também foram desenvolvidas duas bibliotecas para o serviço de *stream*, uma deve ser ligada ao código que possui objetos multimídia (áudio ou vídeo) ou controladores de canal, a outra é destinada às aplicações clientes que utilizam funções do controlador de canal.

Outra biblioteca implementada trata dos aspectos de *multithreading*. Sua função é encapsular a maioria dos mecanismos relacionados com a concorrência, como filtros e semáforos (para sincronização), a fim de facilitar o desenvolvimento de aplicações *multithreaded*. Tal biblioteca deve ser ligada ao código das aplicações servidoras.

Além das bibliotecas descritas nos dois parágrafos anteriores, o código do cliente e do servidor deve ser ligado aos *stubs* e esqueletos, associados aos objetos de aplicação desenvolvidos, e às bibliotecas oferecidas pelas ferramentas Orbix e ObjectStore.

Um cenário típico de utilização da infra-estrutura desenvolvida é mostrado na figura 5.3. Os serviços TINA são compostos de processos servidores que possuem fábricas, objetos de gerência e de aplicação. Os objetos gerentes e suas fábricas possuem *threads* de execução exclusiva. Os demais objetos podem ou não utilizar o mesmo modelo de concorrência. Em cada nó deve existir, além do núcleo (sistema operacional), um processo servidor que possua um gerente de nó.

Aplicações clientes executam operações nos servidores (gerentes e objetos de aplicação) por meio dos mecanismos previstos na especificação CORBA, ou seja, *stub* ou interface de invocação dinâmica (DII). A ferramenta Orbix encarrega-se de transmitir as invocações de operações, e possíveis respostas, pelo sistema[1].

[1]Como Orbix é implementado na forma de bibliotecas, na verdade, cada processo da aplicação (cliente ou servidor) realiza funções do ORB. Porém, conceitualmente, a plataforma comporta-se como uma infra-estrutura única e homogênea sob a perspectiva dos objetos do sistema.

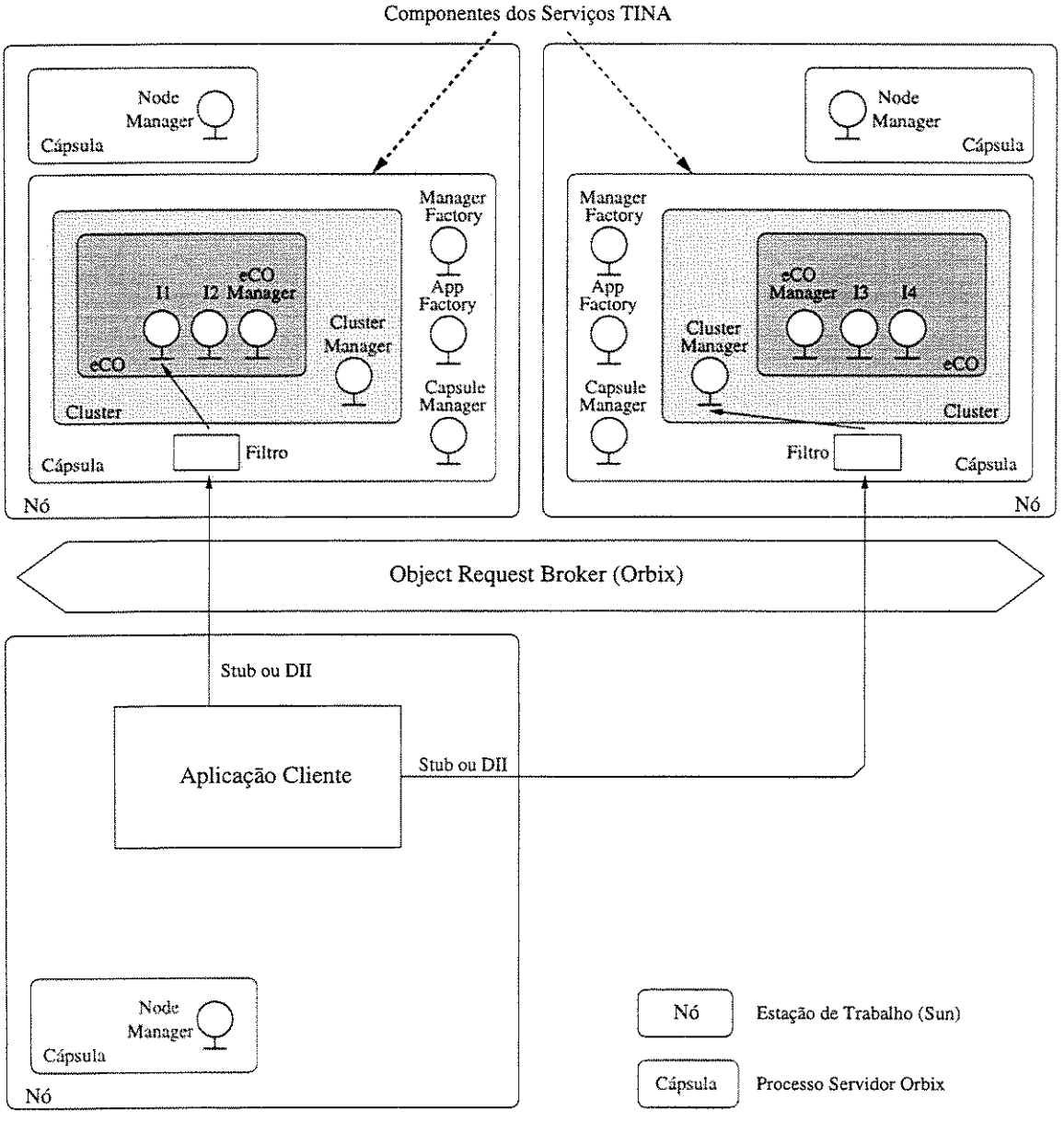

Figura 5.3: Cenário de utilização das funções do DPE

5.5 Aplicação Multimídia Móvel

A fim de testar os serviços proporcionados pelo DPE implementado, projetou-se um protótipo de aplicação TINA, que faz uso de boa parte das funções oferecidas pela infra-estrutura. Tal aplicação consiste de um serviço de comunicação, ponto a ponto, de áudio. Além de realizar conexões de áudio, a aplicação é capaz de mover-se, durante a interação dos usuários do serviço, pelo nós da rede.

O serviço desenvolvido, denominado audiofone, funciona da seguinte forma (diagrama de *Use Case*, figura 5.4). Primeiramente, todos os usuários da aplicação devem se registrar, associando-se a um terminal, a fim de iniciar ou receber ligações. Desta forma, o serviço é capaz de encontrar os nós correspondentes a cada participante. Usuários podem estabelecer uma comunicação entre si caso os terminais em questão não estejam ocupados (isto é, não participam de nenhuma outra conexão de áudio). Além disto, qualquer usuário é capaz de, a qualquer instante, transferir uma ligação para outro nó da rede, desde que o terminal destino não esteja ocupado. Com exceção da breve interrupção do serviço durante o período necessário para realizar a migração da aplicação, toda a mudança é transparente para o usuário que permanece "fixo".

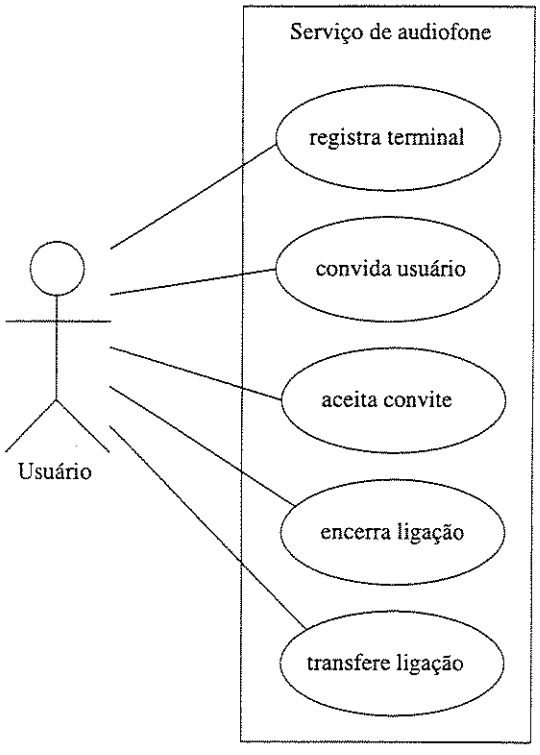

Figura 5.4: Diagrama de *Use Case* do serviço de audiofone

5.5.1 Gerência do Serviço

A gerência da aplicação de audiofone é efetuada de maneira bastante simples. Os componentes que realizam tal tarefa não atendem os requisitos da arquitetura TINA [34]. A implementação dos objetos descritos na arquitetura de serviço está fora do escopo deste trabalho.

Assim, os objetos desenvolvidos não tratam de aspectos importantes, como segurança (autenticação e autorização), contabilidade e tolerância a falhas (armazenagem persistente do estado do serviço). Seu papel é simplesmente oferecer as funções mínimas necessárias para que o serviço de audiofone possa ser utilizado.

Dois objetos CORBA foram definidos para administrar o serviço de audiofone: o agente de terminal e o localizador de usuário.

O agente de terminal (interface `TerminalAgent`) representa cada terminal (nó) que oferece o serviço de audiofone. Sua interface IDL é descrita a seguir.

```
interface TerminalAgent {
  exception UserNotAtTerminal {};
  exception CallRefused {};
  exception TerminalBusy { string CurrentUser; };

  readonly attribute string nodename;

  long associate(in string user);
  long dissociate(in string user);
  long local_call(in string user) raises (UserNotAtTerminal, TerminalBusy);
  long remote_call(in string caller, in string callee)
                  raises (UserNotAtTerminal, TerminalBusy, CallRefused);
  long disconnect();
};
```

Além do `nodename`, que guarda o nome do nó, o agente de terminal possui mais três atributos, que não são acessíveis via interface IDL. Um atributo indica se o terminal está ocupado no momento, outro armazena a lista com os nomes dos usuários associados ao terminal (pode haver mais de um usuário associado ao mesmo nó em um dado instante) e o último contém o nome do usuário que está utilizando o serviço de audiofone (caso o terminal esteja ocupado).

Os métodos `associate()` e `dissociate()` associam e dissociam usuários e terminais, respectivamente.

A função `local_call()` serve para indicar que um determinado usuário deseja utilizar o terminal local para realizar uma chamada para outro nó. O usuário só conseguirá ocupá-lo se estiver associado ao nó local e este terminal não estiver sendo utilizado por outra ligação.

Para chamar um usuário a participar do serviço, executa-se a função `remote_call()` do agente do terminal remoto. Tal método avisa o usuário convidado, via sinal sonoro, da existência da chamada. Se o terminal remoto estiver ocupado ou o usuário não estiver associado a aquele nó, a chamada é cancelada automaticamente. O usuário também pode rejeitar o convite.

O método `disconnect()` serve para indicar o encerramento da ligação atual, desocupando o terminal.

Cada nó que oferece o serviço de audiofone deve possuir um processo servidor (*daemon*) com um agente de terminal.

O localizador de usuário (interface `UserLocator`) é responsável pelo cadastro e busca dos usuários da aplicação de audiofone e possui a seguinte interface IDL:

```
interface UserLocator {
  exception UserNotFound {};

  long Register(in string user, in TerminalAgent terminal);
  long Unregister(in string user);
  TerminalAgent findUser(in string user) raises (UserNotFound);
};
```

Uma tabela representando o mapeamento entre o nome do usuário e a referência de seu terminal associado é mantida pelo localizador de terminal.

O método `Register()` cria (ou atualiza) a entrada da tabela do localizador, correspondente ao usuário em questão. Tal função executa o método `associate()` do agente do terminal especificado, indicando a nova associação[2]. Analogamente, a função `Unregister()` remove o usuário da lista de participantes do serviço e o dissocia do nó atual (método `dissociate()` do agente de terminal).

O método `findUser()` retorna a referência do terminal associado ao usuário em questão.

O localizador de usuários é um componente centralizado e, portanto, existe um único processo servidor responsável pelo atendimento das requisições das aplicações de audiofone.

A figura 5.5 ilustra o processo de registro dos usuários e ativação do serviço de audiofone. Inicialmente, ambos os participantes se registram (`Register()`) no localizador, informando o terminal ao qual estão associados. Então, um dos usuários toma a iniciativa de fazer um convite, executando as funções de busca (`findUser()`), de chamada local (`local_call()`) e remota (`remote_call()`). Se nenhum dos nós estiver ocupado e o usuário convidado aceitar a chamada, a aplicação de audiofone estará pronta para estabelecer a conexão de áudio entre os dois terminais.

5.5.2 Componentes do Serviço

Outros três elementos implementam o serviço de audiofone propriamente dito (figura 5.6):

- **Cápsula de áudio**: corresponde a um processo (*daemon*) que contém objetos relacionados com a aplicação de áudio (gerentes, fábricas, dispositivos de áudio).
- **Controle de conexão**: processo (*daemon*) que possui um controlador de canal de áudio (`AudioChannelCtrl`). Responsável pela gerência dos fluxos de áudio da ligação que foi iniciada pelo terminal local.
- **Cliente**: representa a interface do usuário com o serviço. Por meio do cliente, o usuário pode registrar-se junto ao serviço, convidar outro participante, transferir a ligação e encerrar a conexão.

[2]Se houve uma atualização de terminal, o método também executa a função `dissociate()` do "antigo" terminal associado ao usuário.

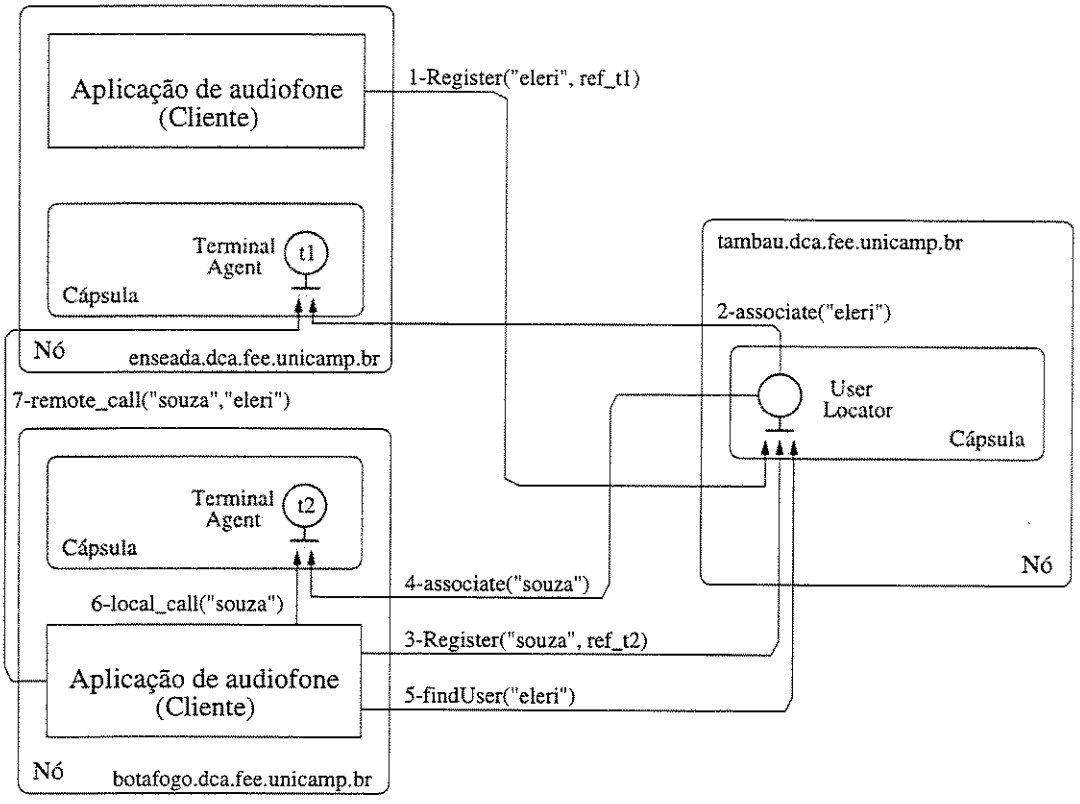

Figura 5.5: Registro dos usuários e ativação do serviço de audiofone

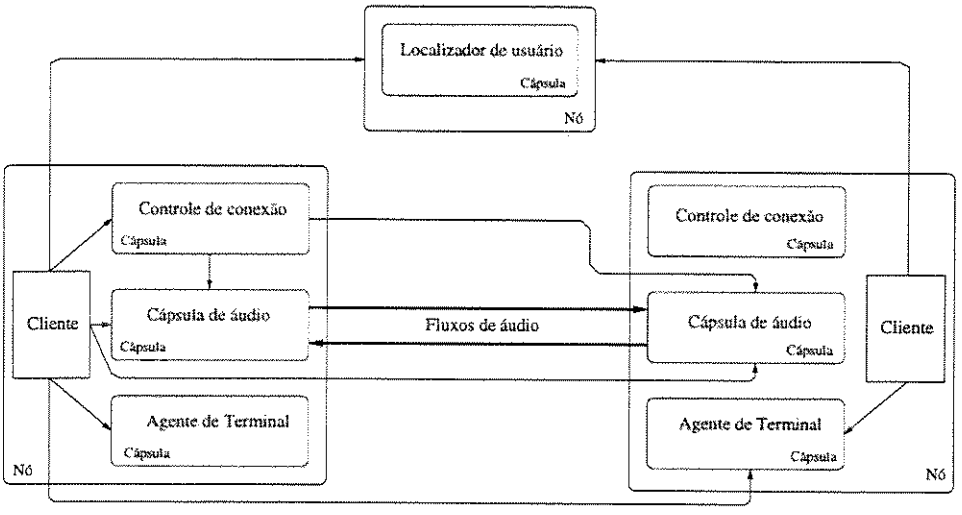

Figura 5.6: Arquitetura do serviço de audiofone

A cápsula de áudio, o controle de conexão e o cliente estão presentes em todos os nós que oferecem o serviço de audiofone.

Além de possuir as fábricas e os gerentes das unidades de engenharia, a cápsula de áudio possui um objeto específico, denominado audiofone (interface `AudioPhone`):

```
interface AudioPhone {
  long createAudioCluster(in string cluster_name, in string audiodevice_name);
};
```

O audiofone funciona como uma fábrica, instanciando *clusters* compostos de um eCO que possui uma interface do tipo `AudioDevice`.

Inicialmente, a cápsula de áudio é iniciada sem nenhum *cluster*. A função `createAudioCluster()` executa o método `makeCluster()` do gerente de cápsula (seção 4.4.3). Logo depois, invoca `makeECO()` (seção 4.4.2) do gerente criado, instanciando um novo eCO. Então, um dispositivo de áudio (`AudioDevice`, seção 4.6.2) é criado e adicionado como uma interface do eCO (método `addInterface()`, seção 4.4.1). A figura 5.7 mostra a configuração da cápsula de áudio antes e depois de uma execução do método `createAudioCluster()`.

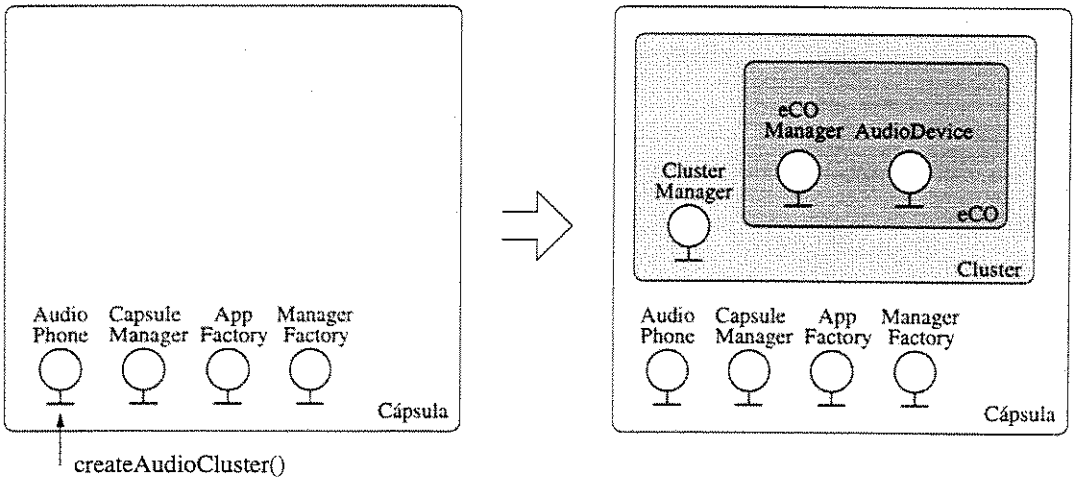

Figura 5.7: Efeito da execução de `createAudioCluster()` sobre a cápsula de áudio

O controle de conexão (figura 5.8) contém uma instância do controlador de canal de áudio (`AudioChannelCtrl`). Convencionou-se que a gerência dos fluxos de áudio do serviço deve ser realizada pelo componente localizado no terminal que iniciou a chamada.

A interface entre o usuário e o serviço é proporcionada pelo cliente da aplicação. Tal componente oferece, via parâmetros de linha de comando, todas as funções do audiofone (figura 5.4).

Para realizar o cadastro dos participantes do serviço, o cliente utiliza as funções do localizador de usuário, descritas na seção anterior.

O estabelecimento de uma ligação é realizado em três etapas. Em primeiro lugar, o cliente realiza o procedimento de convite ao usuário remoto (seção 5.5.1). Uma vez que a chamada foi

Figura 5.8: Controle de conexão

aceita, o cliente cria nas cápsulas de áudio (local e remota), via método `createAudioCluster()`, os objetos necessários para iniciar a conexão de áudio.

Finalmente, após a instanciação dos objetos nas cápsulas de áudio, o cliente é capaz de estabelecer uma comunicação entre os dois terminais. Para isto, segue o procedimento normal de criação de canais (seção 4.6.2): obtém as referências dos dispositivos multimídia, dos gerentes de eCO correspondentes e do controlador de canal (presente no processo de controle de conexão); define a qualidade de serviço a ser oferecida aos dois fluxos de áudio (um para cada sentido da comunicação); e executa a função `createChannel()` do gerente do nó local.

O encerramento da ligação envolve a destruição do *cluster*, via função `Delete()` do gerente, existente na cápsula de áudio local e a execução do método `disconnect()` do agente do terminal local. Assim, cada cliente é responsável pelo término (isto é, destruição do *cluster* de áudio e desconexão do terminal) da sua parte da ligação[3].

A transferência de uma ligação ocorre da seguinte maneira. Se o terminal para onde se pretende transferir a ligação estiver desocupado, o cliente executa a função `disconnect()` do agente local, associa o usuário ao novo nó (`Register()`) e assume o controle do terminal destino (`local_call()`). Depois, simplesmente invoca o método `migrate()` do gerente da cápsula de áudio local, indicando a cápsula e o nó destinos. A partir de então, o processo de migração é realizado automaticamente (figuras 4.15 e 4.16) e a conexão de áudio é restabelecida tão logo os objetos terminem de se mover.

5.5.3 Avaliação do Desenvolvimento do Serviço

Além de testar as funções do DPE, o desenvolvimento do serviço de audiofone revelou algumas vantagens relacionadas com a utilização desta infra-estrutura para a implementação de serviços de telecomunicações.

O DPE encapsula grande parte dos mecanismos necessários para o suporte à comunicação dos componentes das aplicações. Mecanismos da plataforma CORBA (*stubs*, por exemplo) possibilitam interações dos objetos, via interfaces operacionais, em um sistema heterogêneo. O serviço de *stream* proporciona, de maneira simples e padronizada, o estabelecimento e controle de tráfego multimídia.

O serviço de ciclo de vida simplifica a tarefa de gerência, em vários níveis de granulosidade, dos componentes do serviço. Por fim, o suporte automático à migração dos fluxos multimídia também pode ser muito útil a uma grande variedade de aplicações.

Usando o DPE, o projetista da aplicação pode-se concentrar naqueles aspectos diretamente relacionados com a lógica do serviço. No caso do audiofone, por exemplo, o maior esforço durante o desenvolvimento esteve relacionado com a implementação da gerência do serviço (agente de terminal

[3]Tal comportamento segue o modelo do sistema telefônico tradicional.

e localizador de usuário) e da aplicação cliente. Os demais componentes (controle de conexão e cápsula de áudio) foram facilmente implementados com o auxílio das funções do DPE.

Capítulo 6

Conclusão

O desenvolvimento de um ambiente para processamento distribuído, aderente a padrões abertos, constitui a principal contribuição deste trabalho de mestrado. Neste capítulo, é feita uma avaliação final do esforço realizado e são apresentadas algumas propostas de futuros tópicos de pesquisa.

6.1 Avaliação

O ambiente para processamento distribuído (DPE) implementado segue especificações do consórcio TINA, atendendo todos os requisitos definidos como fundamentais (seção 3.7). Entre as principais características da plataforma, destacam-se:

- suporte à distribuição e gerência das unidades de engenharia TINA (eCO, *cluster*, cápsula e nó);
- mecanismos de *binding* explícito (canais) que permitem a comunicação por meio de fluxos de mídia contínua (áudio e vídeo);
- funções que realizam migração de *clusters*, incluindo o restabelecimento automático da comunicação via canais multimídia;
- implementação *multithreaded*, permitindo a escolha de diferentes modelos de concorrência para os objetos de aplicação.

A maioria das atuais plataformas para processamento distribuído não possui suporte nativo às unidades de engenharia TINA, derivadas do modelo ODP. As infra-estruturas orientadas a objeto, em geral, proporcionam funções de gerência apenas para o nível de objetos. O DPE-TINA estende a administração dos objetos para diversos níveis de granulosidade (eCO, *cluster*, cápsula e nó), facilitando o desenvolvimento de aplicações distribuídas.

A existência, no ambiente implementado, de mecanismos de suporte a *bindings* explícitos (canais) e abstrações de dispositivos físicos (microfone e alto-falante, por exemplo) proporciona uma simplificação e padronização aos procedimentos de estabelecimento e controle de tráfego de áudio e vídeo. Tal característica é fundamental em uma plataforma que se propõe a suportar serviços multimídia interativos.

Os novos serviços de telecomunicações contemplarão, cada vez mais, aspectos de mobilidade. Portanto, há a necessidade de que infra-estruturas computacionais permitam, da forma mais transparente possível, a movimentação de componentes de aplicação pela rede. Neste aspecto, o DPE construído possui uma virtude interessante, não encontrada em outros ambientes pesquisados, que é o restabelecimento automático das conexões de áudio/vídeo dos objetos que sofrem migração.

A característica *multithreaded* do projeto desenvolvido, que pode trazer benefícios em termos de desempenho, também é um fator importante visto que a maioria das aplicações de telecomunicações possui requisitos de tempo-real.

A utilização de outros padrões abertos, como a plataforma CORBA e o serviço de A/V *streams* do consórcio OMG, no projeto da arquitetura do DPE proporciona boa flexibilidade à implementação do ambiente. Outros desenvolvedores podem construir, sem maiores dificuldades, DPEs com funções similares, usando produtos diferentes, desde que aderentes aos padrões em questão.

Com relação ao trabalho em si, a pesquisa e o desenvolvimento do DPE provaram ser uma típica tarefa de engenharia. Várias decisões de compromisso foram tomadas, levando em consideração os recursos das ferramentas disponíveis, durante o projeto e a implementação do ambiente. Esta experiência de interpretação de padrões recentes, incluindo as inevitáveis extrapolações realizadas sobre aspectos pouco definidos ou ausentes nas especificações, resultou em uma oportunidade única de aprendizado para o autor desta dissertação de mestrado.

6.2 Trabalhos Futuros

Entre as possíveis melhorias a serem realizadas no ambiente apresentado, podem-se citar os seguintes assuntos relevantes para trabalhos subseqüentes:

- **Canal multiponto**: a atual implementação suporta apenas a criação de canais ponto a ponto. Extensões devem ser efetuadas à interface do controlador de canal a fim de permitir o estabelecimento de conexões *multicast*.

- **Migração**: o mecanismo de migração proposto (desativação/ativação dos objetos) apresenta certos inconvenientes. A necessidade de que a cápsula destino possua acesso aos repositórios onde estão armazenados os estados de todos os objetos que migram pode causar problemas de segurança e de escalabilidade[1]. Outras formas de migração (via serialização, por exemplo) podem ser investigadas. Sugere-se, ainda, o estudo de outras linguagens de programação, mais adequadas para aplicações móveis[2].

- **Serviços DPE**: quando novas especificações definirem mais precisamente os requisitos dos serviços DPE, estudos poderão ser realizados a fim de, por exemplo, adaptar e integrar os serviços comuns do consórcio OMG ao ambiente TINA.

[1] O principal fator que motivou a escolha de tal mecanismo de migração foi tecnológico: a linguagem utilizada para a construção do DPE (C++) é dependente de plataforma, tornando bastante difícil a implementação de movimentação de código em um sistema heterogêneo.

[2] A linguagem Java é uma forte candidata para futuras implementações do DPE. Atualmente, o principal obstáculo que impede a construção de todo o ambiente em Java é a ausência de bibliotecas de suporte à captura e reprodução, em tempo-real, de áudio/vídeo. A tecnologia JMF (*Java Media Framework* [68]), em processo de desenvolvimento pela Sun e IBM, deve eliminar este empecilho.

- **Segurança**: pesquisas sobre este aspecto primordial devem ser conduzidas para oferecer mecanismos de proteção (autenticação, autorização e criptografia) [69] aos serviços de ciclo de vida e de *stream* presentes no DPE.

Finalmente, estão sendo desenvolvidas aplicações de telecomunicações, seguindo a arquitetura de serviço do consórcio TINA, sobre a atual infra-estrutura DPE [70].

Bibliografia

[1] B. Shankar. "Corporate Landscape". *Telecommunications*, Julho 1998.

[2] T. Zahariadis et al. "Interactive Multimedia Services to Residential Users". *IEEE Communications Magazine*, 35(6), Junho 1997.

[3] R. W. Lucky. "New Communications Services - What Does Society Want?". *Proceedings of the IEEE*, 85(10), Outubro 1997.

[4] M. R. Macedonia e D. P Brutzman. "Mbone Provides Audio and Video Over the Internet". *IEEE Computer*, Abril 1994.

[5] D. Minoli e E. Minoli. *"Delivering Voice Over IP Networks"*. John Wiley, 1998.

[6] F. Dupuy e G. Nilsson e Y. Inoue. "The TINA Consortium: Toward Networking Telecommunications Information Services". *IEEE Communications Magazine*, 33(11), Novembro 1995.

[7] ITU-T Recommendation Q.1200 Series. *"Intelligent Networks"*, 1995.

[8] ITU-T Recommendation M.3010. *"Principles for a Telecommunications Management Network"*, 1992.

[9] G. Pavlou e D. Griffin. "Issues in the Integration of IN and TMN". *Lecture Notes in Computer Science*, 998, 1995. Third International Conference on Intelligence in Broadband Services and Networks.

[10] U. Herzog e T. Magedanz. "From IN toward TINA - Potential Migration Steps". *Lecture Notes in Computer Science*, 1238, 1997. Fourth International Conference on Intelligence in Services and Networks.

[11] P. G. Bosco et al. "The ReTINA Project: an Overview". Technical Report RT/TR-96-15.1, Chorus, http://www.chorus.com/Documentation/retina.html, Novembro 1996.

[12] ACTS. "VITAL - Home Page". http://www.mari.co.uk/vital, Accessado em Novembro 1998.

[13] J. Huélamo et al. "A TINA based Prototype for a Multimedia Multiparty Mobility Service". *Lecture Notes in Computer Science*, 1238, 1997. Fourth International Conference on Intelligence in Services and Networks.

[14] ACTS. "Overview Page for the DOLMEN Project". http://www.fub.it/dolmen, Accessado em Novembro 1998.

[15] Deutsche Telecom e France Telecom e Sprint. "Global One Alliance TTT - The TINA Trial". http://www.tinac.com/TTT/GlobalOne/globalone.html, Accessado em Novembro 1998.

[16] FUJITSU e NEC e NTT e IONA et al. "Yet Another TINA Trial". http://www.tinac.com/TTT/YAT/yat.html, Accessado em Novembro 1998.

[17] P. Bosco et al. "Specifying and Developing TINA Applications with ACE". In *TINA'97 Conference*, Santiago, Chile, Novembro 1997.

[18] CSELT. "ACE Home Page". http://www.andromeda.cselt.it/ace/ACE.html, Accessado em Novembro 1998.

[19] ANSA. "The ANSA Computational Model". Technical report, Architecture Projects Management Limited, Agosto 1992.

[20] Bellcore SR-NWT-002282. "INA Cycle 1 Framework Architecture". Technical Report Issue 2, Bellcore, Abril 1993.

[21] RACE Project R.1093. "Foundation of Object Orientation in ROSA, Release Two". Technical Report 93/BTL/DS/A/010/b1, RACE, Maio 1992.

[22] J. B. Stefani e Y. Lepetit. "The SERENITE Long Term IN Architecture". In *TINA'93 - The Fourth Telecommunications Information Networking Architecture Workshop*, L'Aquila, Itália, Setembro 1993.

[23] M. Chapman e S. Montesi. *"Overall Concepts and Principles of TINA"*. http://www.tinac.com, Fevereiro 1995. Versão 1.0.

[24] T. Magedanz. "TINA - Architectural Basis for Future Telecommunications Services". *Computer Communications*, 20(4), Junho 1997.

[25] ISO/IEC 10746-2 / ITU-T Recommendation X.902. *"ODP Reference Model Part 2, Foundations"*, Junho 1995.

[26] ISO/IEC 10746-3 / ITU-T Recommendation X.903. *"ODP Reference Model Part 3, Architecture"*, Junho 1995.

[27] M. Yates et al. *"TINA Business Model and Reference Points"*. http://www.tinac.com, Maio 1997. Versão 4.0.

[28] H. Christensen e E. Colban. *"Information Modelling Concepts"*. http://www.tinac.com, Abril 1995. Versão 2.0.

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

[30] TINA-C Core Team. *"Computational Modelling Concepts"*. http://www.tinac.com, Maio 1996. Versão 3.2.

[31] N. Mercouroff e A. Parhar. "TINA Computational Modelling Concepts and Object Definition Language". *Lecture Notes in Computer Science*, 1238, 1997. Fourth International Conference on Intelligence in Services and Networks.

[32] TINA-C Core Team. *"TINA Object Definition Language Manual"*. http://www.tinac.com, Julho 1996. Versão 2.3.

[33] P. Graubmann et al. *"Engineering Modelling Concepts (DPE Architecture)"*. http://www.tinac.com, Dezembro 1994. Versão 2.0.

[34] C. Abarca et al. *"Service Architecture"*. http://www.tinac.com, Junho 1997. Versão 5.0.

[35] C. Abarca et al. *"Network Resource Architecture"*. http://www.tinac.com, Fevereiro 1997. Versão 3.0.

[36] ITU-T Recommendation G.803. *"Architectures of Transport Networks Based on the Synchronous Digital Hierarchy (SDH)"*, Junho 1992.

[37] ITU-T Recommendation M.3100. *"Generic Network Information Model"*, 1992.

[38] C. Abarca et al. *"Network Resource Information Model (NRIM)"*. http://www.tinac.com, Novembro 1997. Versão 2.2.

[39] L. A. de la Fuente e T. Walles. *"Management Architecture"*. http://www.tinac.com, Dezembro 1994. Versão 2.0.

[40] Object Management Group. *"The Common Object Request Broker: Architecture and Specification"*. http://www.omg.org, Março 1998. Versão 2.2.

[41] S. Vinoski. "CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments". *IEEE Communications Magazine*, 14(2), Fevereiro 1997.

[42] W. Kim, editor. *"Modern Database Systems - The Object Model, Interoperability, and Beyond"*. ACM Press, Addison Wesley, 1995.

[43] Object Management Group. "Multiple Interfaces and Composition Request for Proposals". Technical Report orb/96-01-04, Object Management Group, http://www.omg.org, Agosto 1996.

[44] D. E. Araújo. "Serviços de Gerenciamento ODP Utilizando a Arquitetura CORBA". Master's thesis, Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica e de Computação, 1996.

[45] A. S. Tanenbaum. *"Modern Operating Systems"*. Prentice-Hall International, 1992.

[46] IONA Technologies Ltd. "Orbix + ObjectStore Adapter - White Paper", 1997. http://www.iona.com.

[47] D. Flanagan. *"Java in a Nutshell"*. O'Reilly, second edition, 1997.

[48] E. Gamma et al. *"Design Patterns : Elements of Reusable Object-Oriented Software"*. Addison Wesley, 1995.

[49] IONA Technologies PLC. *"Orbix Programmer's Guide"*, Outubro 1997. Versão 2.3.

[50] IONA Technologies PLC. *"Orbix Programmer's Reference"*, Outubro 1997. Versão 2.3.

[51] Object Management Group. *"CORBAservices: Common Object Services Specification"*. http://www.omg.org, Novembro 1997. formal/98-07-05.

[52] Ad Astra Engineering. "Jumping Beans - White Paper". http://www.jumpingbeans.com, 1998.

[53] D. Schmidt e S. Vinoski. "Object Adapters: Concepts and Terminology (Column 11)". *SIGS C++ Report Magazine*, Outubro 1997.

[54] Object Management Group. *"CORBAtelecoms: Telecommunications Domain Specification"*. http://www.omg.org, Junho 1998. formal/98-07-12.

[55] "IP Mobility Suport". Technical Report RFC 2002, IETF, Outubro 1996.

[56] C. Perkins. "Mobile Networking Through Mobile IP". *IEEE Internet Computing*, Janeiro/Fevereiro 1998.

[57] S. Vinoski. "New Features for CORBA 3.0". *Communications of the ACM*, 41(10), Outubro 1998.

[58] D. Schmidt e S. Vinoski. "Comparing Alternative Programming Techniques for Multi-threaded CORBA Servers (Column 7)". *SIGS C++ Report Magazine*, Julho/Agosto 1996.

[59] Object Management Group. "Realtime CORBA 1.0 Request for Proposal (Abbreviated Version)". Technical Report orbos/97-09-31, Object Management Group, http://www.omg.org, Agosto 1998.

[60] P. C. de Oliveira e E. Cardozo. "Mobile Agent-Based Systems: an Alternative Paradigm for Distributed Systems Development". In *XV Simpósio Brasileiro de Redes de Computadores*, Maio 1997.

[61] F. J. S. Vasconcellos e E. R. M. Madeira. "Projeto e Desenvolvimento de um Suporte a Agentes Móveis baseado em CORBA". In *XVI Simpósio Brasileiro de Redes de Computadores*, Maio 1998.

[62] ObjectSpace. "ObjectSpace Voyager Core Package Technical Overview". http://www.objectspace.com, 1997.

[63] R. C. M. Prado. "Implementação de Canais ODP sobre Plataforma CORBA". Master's thesis, Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica e de Computação, 1997.

[64] J. W. Hong et al. "Design and Implementation of a Distributed Multimedia Collaborative Environment". *Networks Software Tools, and Applications*, Outono 1998.

[65] RealNetworks. "RealNetworks - The Home of Streaming Media". http://www.real.com, 1998.

[66] Lawrence Berkeley Laboratory. "Visual Audio Tool". ftp://ftp.ee.lbl.gov/conferencing/vat, 1998.

[67] Object Design Inc. *"ObjectStore C++ API User Guide"*, Junho 1995. Release 4.0.

[68] Sun Microsystems. *"Java Media Framework Programmer's Guide (v 0.5)"*, Dezembro 1998. http://java.sun.com/products/java-media/jmf.

[69] B. Lampson et al. "Authentication in Distributed Systems: Theory and Practice". *ACM Transactions on Computer Systems*, 10(4), Novembro 1992.

[70] A. S. Pinto et al. "TINA-based Environment for Mobile Multimedia Services". In *TINA'99 Conference*, Havaí, EUA, Abril 1999. Aceito para publicação.

Apêndice A

Interfaces IDL Desenvolvidas

A.1 Fábrica de Gerentes (ManagerFactory)

```
interface ManagerFactory {
  eCOManager createECOManager(in string name);
  ClusterManager createClusterManager(in string name);
  CapsuleManager createCapsuleManager(in string name);
};
```

A.2 Fábrica de Objetos de Aplicação (AppFactory)

```
interface AppFactory {
  Object createApp(in string str_ref, in string template);
};
```

A.3 Gerente de eCO (eCOManager)

```
typedef sequence<Object> interface_seq;
interface eCOManager {
  long addInterface(in Object interface_ref);
  long removeInterface(in Object interface_ref);
  interface_seq getInterfaces();
  long checkpoint(in string template);
  long recover(in string template);
  long Delete();
  long deactivate(in string template);
  long addChannel(in ChannelCtrl the_ctrl);
  long deleteChannel(in ChannelCtrl the_ctrl);
};
```

A.4 Gerente de *Cluster* (ClusterManager)

```
typedef sequence<eCOManager> eco_seq;
interface ClusterManager {
  eCOManager makeECO(in string eco_name);
  eco_seq getECOs();
  long checkpoint(in string template);
  long recover(in string template);
  long Delete();
  long deactivate(in string template);
};
```

A.5 Gerente de Cápsula (CapsuleManager)

```
typedef sequence<ClusterManager> cluster_seq;
interface CapsuleManager {
  ClusterManager makeCluster(in string cluster_name);
  long reactivate(in string cluster_name, in string template);
  long migrate(in string cluster_name, in string capsule_name,
               in string node_name);
  cluster_seq getClusters();
  long Delete();
};
```

A.6 Gerente de Nó (NodeManager)

```
typedef sequence <CapsuleManager> capsule_seq;
interface NodeManager {
  capsule_seq getCapsules();
  long createChannel(in eCOManager a_eco, in AppMMDevice a_party,
                     in eCOManager b_eco, in AppMMDevice b_party,
                     in AVStreams::streamQos the_qos,
                     in AVStreams::flowSpec the_spec,
                     in ChannelCtrl the_ctrl);
  long destroyChannel(in ChannelCtrl the_ctrl);
};
```

A.7 Controlador de Canal (ChannelCtrl)

```
interface ChannelCtrl : AVStreams::StreamCtrl {
  long configureChannel(in AVStreams::streamQos the_qos,
                        in AVStreams::flowSpec the_spec);
  long configureEndPoints(in eCOManager a_eco, in AppMMDevice a_party,
                          in eCOManager b_eco, in AppMMDevice b_party);
  long deactivateChannel();
  long reactivateChannel();
  long changeEndPoint(in eCOManager eco, AppMMDevice party);
  long changeNode(in string node_name);
};
```

A.8 Controlador de Canal de Áudio (AudioChannelCtrl)

```
interface AudioChannelCtrl : ChannelCtrl {};
```

A.9 Controlador de Canal de Vídeo (VideoChannelCtrl)

```
interface VideoChannelCtrl : ChannelCtrl {};
```

A.10 Dispositivo de Aplicação Multimídia (AppMMDevice)

```
interface AppMMDevice : AVStreams::MMDevice {
  long checkpoint(in string template);
  long recover(in string template);
  long Delete();
};
```

A.11 Dispositivo de Áudio (AudioDevice)

```
interface AudioDevice : AppMMDevice {};
```

A.12 Dispositivo de Vídeo (VideoDevice)

```
interface VideoDevice : AppMMDevice {};
```

A.13 Agente de Terminal (TerminalAgent)

```
interface TerminalAgent {
  exception UserNotAtTerminal {};
  exception CallRefused {};
  exception TerminalBusy { string CurrentUser; };

  readonly attribute string nodename;

  long associate(in string user);
  long dissociate(in string user);
  long local_call(in string user) raises (UserNotAtTerminal, TerminalBusy);
  long remote_call(in string caller, in string callee)
                  raises (UserNotAtTerminal, TerminalBusy, CallRefused);
  long disconnect();
};
```

A.14 Localizador de Usuário (UserLocator)

```
interface UserLocator {
  exception UserNotFound {};

  long Register(in string user, in TerminalAgent terminal);
  long Unregister(in string user);
  TerminalAgent findUser(in string user) raises (UserNotFound);
};
```

A.15 Audiofone (AudioPhone)

```
interface AudioPhone {
  long createAudioCluster(in string cluster_name, in string audiodevice_name);
};
```

Apêndice B

Interfaces IDL da Especificação de Controle e Gerência de A/V Streams

```
#include "../PropertyService/PropertyService.idl"

module AVStreams
{
  // ************************************** Data Types
  struct QoS
  {
      string QoSType;
      PropertyService::Properties QoSParams;
  };

  typedef sequence<QoS> streamQoS;
  typedef sequence<string> flowSpec;
  typedef sequence<string> protocolSpec;
  typedef sequence<octet> key;

  // Protocol Names registered by OMG: e.g., TCP, UDP, AAL5, IPX, RTP

  // This structure is defined for SFP1.0
  // Subsequent versions of the protocol may specify new structures
  struct SFPStatus
  {
      boolean isFormatted;
      boolean isSpecialFormat;
      boolean seqNums;
      boolean timestamps;
      boolean sourceIndicators;
  };
```

```
enum flowState {stopped, started, dead};
enum dirType {dir_in, dir_out};

struct flowStatus
{
    string flowName;
    dirType directionality;
    flowState status;
    SFPStatus theFormat;
    QoS theQoS;
};

typedef sequence<flowStatus> seqFlowStatus;
typedef sequence<string> parties;   // A_parties, B_parties
typedef PropertyService::Property streamEvent;

struct resolution
{
  long horz;
  long vert;
};


// ************************************** Exception
exception notSupported {};
exception PropertyException {};
exception FPError{ string flow_name; };  // An flow protocol related error
exception streamOpFailed{ string reason; };
exception streamOpDenied{ string reason; };
exception noSuchFlow{};
exception QoSRequestFailed{ string reason; };


// ************************************** Interface Declaration
interface VDev;
interface MMDevice;
interface StreamEndPoint;
interface StreamEndPoint_A;
interface StreamEndPoint_B;
```

```
// ************************************** Interface Definitions

// ************************************** The Basic_StreamCtrl Interface
interface Basic_StreamCtrl : PropertyService::PropertySet {
  // Empty flowSpec => apply operation to all flows
  void stop(in flowSpec the_spec) raises (noSuchFlow);
  void start(in flowSpec the_spec) raises (noSuchFlow);
  void destroy(in flowSpec the_spec) raises (noSuchFlow);

  boolean modify_QoS(inout streamQoS new_qos, in flowSpec the_spec)
                    raises(noSuchFlow, QoSRequestFailed);

  // Called by StreamEndPoint when something goes wrong with a flow
  oneway void push_event(in streamEvent the_event);

  void set_FPStatus(in flowSpec the_spec,
                    in string fp_name,  // Only SFP1.0 currently specified
                    in any fp_settings)
                   raises(noSuchFlow, FPError);

  Object get_flow_connection(in string flow_name)
                            raises(noSuchFlow, notSupported);

  void set_flow_connection(in string flow_name, in Object flow_connection)
                          raises(noSuchFlow, notSupported);
};


// ************************************** The StreamCtrl Interface
interface StreamCtrl : Basic_StreamCtrl {
  boolean bind_devs(in MMDevice a_party, in MMDevice b_party,
                    inout streamQoS the_qos, in flowSpec the_flows)
                   raises(streamOpFailed, noSuchFlow, QoSRequestFailed);

  boolean bind(in StreamEndPoint_A a_party, in StreamEndPoint_B b_party,
               inout streamQoS the_qos, in flowSpec the_flows)
              raises (streamOpFailed, noSuchFlow, QoSRequestFailed);

  void unbind_party(in StreamEndPoint the_ep, in flowSpec the_spec)
                   raises(streamOpFailed, noSuchFlow);

  void unbind() raises (streamOpFailed);
};
```

```
// ************************************* The Negotiator Interface
interface Negotiator {
  boolean negotiate(in Negotiator remote_negotiator,
                    in streamQoS qos_spec);
};


// ************************************* The MCastConfigIf Interface
interface MCastConfigIf : PropertyService::PropertySet {
  boolean set_peer(in Object peer, inout streamQoS the_qos,
                   in flowSpec the_spec)
                  raises(QoSRequestFailed, streamOpFailed);

  void configure(in PropertyService::Property a_configuration);
  void set_initial_configuration(in PropertyService::Properties initial);

  // Uses <format_name> standardised by OMG and IETF
  void set_format(in string flowName, in string format_name)
                 raises(notSupported);

  // Note, some of these device params are standardised by OMG
  void set_dev_params(in string flowName,
                      in PropertyService::Properties new_params)
                     raises(PropertyException, streamOpFailed);
};


// ************************************* The MMDevice Interface
interface MMDevice : PropertyService::PropertySet {
  StreamEndPoint_A create_A(in StreamCtrl the_requester,
                            out VDev the_vdev, inout streamQoS the_qos,
                            out boolean met_qos, inout string named_vdev,
                            in flowSpec the_spec)
                           raises(streamOpFailed, streamOpDenied,
                                  notSupported, QoSRequestFailed,
                                  noSuchFlow);
  StreamEndPoint_B create_B(in StreamCtrl the_requester,
                            out VDev the_vdev, inout streamQoS the_qos,
                            out boolean met_qos, inout string named_vdev,
                            in flowSpec the_spec)
                           raises(streamOpFailed, streamOpDenied,
                                  notSupported, QoSRequestFailed,
                                  noSuchFlow);
```

```
  StreamCtrl bind(in MMDevice peer_device, inout streamQoS the_qos,
                  out boolean is_met,in flowSpec the_spec)
                 raises(streamOpFailed, noSuchFlow, QoSRequestFailed);

  StreamCtrl bind_mcast(in MMDevice first_peer, inout streamQoS the_qos,
                        out boolean is_met, in flowSpec the_spec)
                       raises(streamOpFailed, noSuchFlow, QoSRequestFailed);

  void destroy(in StreamEndPoint the_ep, in string vdev_name)
              raises(notSupported);

  string add_fdev(in Object the_fdev) raises(notSupported, streamOpFailed);

  Object get_fdev(in string flow_name) raises(notSupported, noSuchFlow);

  void remove_fdev(in string flow_name) raises(notSupported, noSuchFlow);
};


// ************************************** The VDev Interface
interface VDev : PropertyService::PropertySet {
  boolean set_peer(in StreamCtrl the_ctrl, in VDev the_peer_dev,
                   inout streamQoS the_qos, in flowSpec the_spec)
                  raises(noSuchFlow, QoSRequestFailed, streamOpFailed);
  boolean set_Mcast_peer(in StreamCtrl the_ctrl,
                         in MCastConfigIf a_mcastconfigif,
                         inout streamQoS the_qos, in flowSpec the_spec)
                        raises(noSuchFlow, QoSRequestFailed, streamOpFailed);

  void configure(in PropertyService::Property the_config_mesg)
                raises(PropertyException, streamOpFailed);

  // Uses <formatName> standardised by OMG and IETF
  void set_format(in string flowName, in string format_name)
  // Note, some of these device params are standardised by OMG
  void set_dev_params(in string flowName,
                      in PropertyService::Properties new_params)
                     raises(PropertyException, streamOpFailed);

  boolean modify_QoS(inout streamQoS the_qos, in flowSpec the_spec)
                    raises(noSuchFlow, QoSRequestFailed);
};
```

```
// ************************************* The StreamEndPoint Interface
interface StreamEndPoint : PropertyService::PropertySet {
  void stop(in flowSpec the_spec) raises (noSuchFlow);
  void start(in flowSpec the_spec) raises (noSuchFlow);
  void destroy(in flowSpec the_spec) raises (noSuchFlow);

  boolean connect(in StreamEndPoint responder, inout streamQoS qos_spec,
                  in flowSpec the_spec)
                 raises(noSuchFlow, QoSRequestFailed, streamOpFailed);

  boolean request_connection(in StreamEndPoint initiator,
                             in boolean is_mcast, inout streamQoS qos,
                             inout flowSpec the_spec)
                            raises(streamOpDenied, noSuchFlow,
                                   QoSRequestFailed, FPError);

  boolean modify_QoS(inout streamQoS new_qos, in flowSpec the_flows)
                    raises(noSuchFlow, QoSRequestFailed);

  boolean set_protocol_restriction(in protocolSpec the_pspec);

  void disconnect(in flowSpec the_spec) raises(noSuchFlow, streamOpFailed);

  void set_FPStatus(in flowSpec the_spec, in string fp_name,
                    in any fp_settings)
                   raises(noSuchFlow, FPError);

  Object get_fep(in string flow_name) raises(notSupported, noSuchFlow);

  string add_fep(in Object the_fep) raises(notSupported, streamOpFailed);

  void remove_fep(in string fep_name) raises(notSupported, streamOpFailed);

  void set_negotiator(in Negotiator new_negotiator);
  void set_key(in string flow_name, in key the_key);
  void set_source_id(in long source_id);
};
```

```
  // ************************************* The StreamEndPoint_A Interface
  interface StreamEndPoint_A : StreamEndPoint {
    boolean multiconnect(inout streamQoS the_qos, inout flowSpec the_spec)
                        raises(noSuchFlow, QoSRequestFailed, streamOpFailed);

    boolean connect_leaf(in StreamEndPoint_B the_ep, inout streamQoS the_qos,
                         in flowSpec the_flows)
                        raises(streamOpFailed, noSuchFlow, QoSRequestFailed,
                               notSupported);

    void disconnect_leaf(in StreamEndPoint_B the_ep, in flowSpec theSpec)
                        raises(streamOpFailed, noSuchFlow);
  };


  // ************************************* The StreamEndPoint_B Interface
  interface StreamEndPoint_B : StreamEndPoint {

    boolean multiconnect(inout streamQoS the_qos, inout flowSpec the_spec)
                        raises(streamOpFailed, noSuchFlow, QoSRequestFailed,
                               FPError);
  };
};
```