UNIVERSIDADE ESTADUAL DE CAMPINAS

FACULDADE DE ENGENHARIA ELÉTRICA
E DE COMPUTAÇÃO

Implementação de Canais ODP sobre Plataforma CORBA

Rodrigo Chavez Monteiro do Prado
Orientador: **Prof. Dr. Eleri Cardozo**

Dissertação de Mestrado

Campinas - SP Brasil
1997

UNIVERSIDADE ESTADUAL DE CAMPINAS

FACULDADE DE ENGENHARIA ELÉTRICA
E DE COMPUTAÇÃO

Implementação de Canais ODP sobre Plataforma CORBA

Rodrigo Chavez Monteiro do Prado
Orientador: **Prof. Dr. Eleri Cardozo**

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

Campinas - SP Brasil
1997

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

C398i

Chavez Monteiro do Prado, Rodrigo
Implementação de canais ODP sobre plataforma CORBA. / Rodrigo Chavez Monteiro do Prado.-- Campinas, SP: [s.n.], 1997.

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

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

À Karina
pela cumplicidade.
Ao meu pai César,
à minha mãe Tânia,
à minha irmã Fátima e
ao meu irmão Marcos
pela presença
apesar da distância.

"O preço da sabedoria de *Odin* - São numerosos os poemas que perpetuam os conselhos que *Odin* dava aos homens e as regras de conduta que ele lhes ensinava. Contudo, a sua ciência não era infusa: teria sido obtida por um preço dolorosíssimo. Diz uma lenda escandinava que o seu grande mestre e conselheiro foi *Mimir* (aquele que pensa), gênio ou demônio do rio da inteligência e da sabedoria, cuja nascente ficava junto de uma das raízes do freixo de *Yggdrasil*.

Quando *Odin* se dirigiu a *Mimir*, para beber do Poço da Sabedoria, foi-lhe revelado que teria que pagar, por isso, um preço terrível: o seu olho direito. Ficou muito perturbado e sentiu quase o desejo de regressar a *Asgard*, desistindo de alcançar o saber que tanto ambicionava. Mas, ao voltar-se, olhou para o Sul, na direção de *Muspellsheim* (a terra do fogo), e viu *Surt* com a espada flamejante - uma figura terrível que um dia se juntaria aos Gigantes na guerra contra os deuses; virou-se para o Norte e viu *Niflheim*, o lugar da escuridão e do terror. Bem sabia que o mundo estava entre *Surt*, que o destruiria com o fogo, e *Niflheim*, de onde viria aquele que havia de levar novamente o mundo para a escuridão e o nada. Competia-lhe a ele, o maior dos deuses, conquistar a sabedoria e ajudar a salvar o mundo. E aceitando o sofrimento que teria de suportar, dirigiu-se para o poço. *Mimir*, que tudo conhecia, soube logo quem estava na sua frente, e saudou-o. *Odin* curvou-se diante daquele que era o mais sábio dos seres, e disse-lhe: 'Quero beber da tua água, *Mimir*!' E ouviu esta resposta: 'Terás que pagar um preço. Todos os que, até hoje, aqui vieram, se recusaram a isso. Quererás pagá-lo, tu, o mais poderoso dos deuses?' Corajosamente, *Odin* afirmou: 'Não recearei pagar o preço que tem que ser pago.' *Mimir* pegou no chifre que estava junto da raiz de *Yggdrasil*, encheu-o de água e deu-lho: 'Neste caso, bebe.'

Odin bebeu. Então o futuro tornou-se-lhe claro. Viu todos os desgostos e trabalhos que sobreviriam aos deuses e aos homens. Mas viu igualmente o motivo por que tinham que acontecer, e também como podiam ser suportados, de maneira que os deuses e os homens, mostrando-se nobres nos dias difíceis, deixassem no mundo uma força que, um dia, muito mais tarde, acabaria por destruir o mal, que causava terror, amargura e desespero. Depois de ter bebido, *Odin* arrancou o seu olho direito; sentiu uma dor terrível, mas não soltou um gemido. Curvou a cabeça e tapou a cara com um manto. *Mimir* pegou no olho e deixou-o afundar-se no Poço da Sabedoria. O olho brilhou através da água, como um sinal, para que todos os que fossem àquele lugar ficassem conhecendo o preço que o maior dos deuses pagara pela sua sabedoria"[1].

---

[1] M. Lamas, Mitologia Geral - O Mundo dos Deuses e dos Heróis, Volume IV, Editorial Estampa, Portugal, 1973

Agradecimentos

Pela orientação e dedicação, ao prof. Eleri.

Pelas idéias, explicações, observações, dicas, incentivo e amizade a todos os amigos.

Pelos desenhos de microfone e alto-falante ao Luís Fernando Faina.

Pelo apoio financeiro, à Capes.

Por isso e muito mais, sou-lhes profundamente grato.

Sumário

Lista de figuras

Lista de tabelas

Lista de abreviações

CC	Channel Controller
CORBA	Common Object Request Broker Architecture
IDL	Interface Definition Language
ISO	International Organization for Standadization
ITU-T	International Telecommunication Union
ODP	Open Distributed Processing
OMA	Object Management Architecture
OMG	Object Management Group
OMT	Object Modeling Technique
ORB	Object Request Boker
QoS	Quality of Service
RFC	Request for Comments
RM-ODP	Reference Model of Open Distributed Processing
RPC	Remote Procedure Call
RSVP	Resource Reservation Protocol
RTP	Real Time Protocol
S	Stub
SnkB	SinkBinder
SnkF	SnkChannelFactory
SnkP	SinkProtocol
SrcB	SourceBinder
SrcF	SrcChannelFactory
SrcP	SourceProtocol
TCP	Transport Control Protocol
TINA	Telecommunication Information Networking Architecture
UDP	User Datagram Protocol

Resumo

Este trabalho objetiva a implementação de canais ODP (*Open Distributed Processing*). O desenvolvimento do trabalho se inicia com o estudo da especificação ODP, padronizada pela ISO (*International Organization for Standardization*), em resposta ao aumento da demanda por sistemas distribuídos que se tornaram possíveis com os avanços tecnológicos recentes. Com base neste estudo, é proposta uma implementação de canal para transmissão de fluxos contínuos de dados, por exemplo, áudio e vídeo. A implementação utiliza como infra-estrutura de distribuição de objetos uma plataforma comercial CORBA (*Common Object Request Broker Architecture*). Enquanto o controle do canal é realizado por intermédio da plataforma, os dados são transmitidos por um sistema de transporte que utiliza TCP ou UDP. Uma aplicação de áudio-conferência é construída para avaliar a implementação. Algumas medidas de monitoramento de qualidade de serviço do canal são apresentadas. Este trabalho contribui na identificação dos requisitos necessários ao desenvolvimento de aplicações distribuídas e no desenvolvimento de uma infra-estrutura de suporte à implementação de aplicações multimídia distribuídas.

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

Abstract

This work aims at the implementation of a component of the ODP (Open Distributed Processing) reference model: the channel. The work begins with a study of the ODP standards, a ISO (International Organization for Standadization) international effort in response to the rapid growth of the distributed processing due to recent advances in computing and network technologies. After that, the work proposes an implementation of ODP channels for supporting the transport of continous media, such as audio and video over the network. The implementation employs a commercial CORBA (Common Object Request Broker Architecture) plataform. The control of the channel is processed though the CORBA plataform while the media flows though a transport infrastructure based on TCP or UDP. An audio-conference application was developed in order to validate the implementation. Finally, this work contributes to the development of distributed applications as well as to the design of infrastructure for supporting distributed multimedia applications.

Keywords: Distributed processing. Objetct-oriented programming. Computer networks.

Capítulo 1

Introdução

O avanço tecnológico de processadores e de redes locais, a partir da década de 1980, permitiu a conexão de um grande número de unidades de processamento, por intermédio de redes de alta velocidade, formando um único sistema computacional. Este tipo de sistema é denominado sistema distribuído.

Os sistemas distribuídos apresentam algumas vantagens sobre os tradicionais sistemas centralizados. Entre elas pode-se destacar: os microprocessadores apresentam uma relação custo/rendimento mais alta que os processadores de grande porte; um sistema distribuído pode ter um poder computacional total mais alto do que os centralizados; algumas aplicações são inerentemente distribuídas; sistema como um todo pode continuar a funcionar se uma máquina falhar; e o sistema distribuído permite um crescimento incremental.

O rápido crescimento dos sistemas distribuídos induziu a criação de uma especificação por parte de organizações internacionais de padronização. O RM-ODP (*Reference Model of Open Distributed Processing*) é o resultado do trabalho da ISO[2] (*International Organization for Standardization*) em associação com a ITU-T[3] (*International Telecommunications Union*). O objetivo da padronização RM-ODP é definir uma arquitetura com a qual possa ser integrado o suporte à distribuição, interconexão e portabilidade.

---

[2] http://www.iso.ch/
[3] http://www.itu.ch/

Mais recentemente foi criado por algumas empresas de informática o grupo OMG[4] (*Object Management Group*). Este consórcio internacional tem o objetivo de criar especificações para sistemas distribuídos orientados a objetos visando reusabilidade, portabilidade e interoperabilidade. O trabalho do OMG vêm tendo grande aceitação no mercado e o número de implementação das especificações e de membros do consórcio é crescente.

Em 1992, fabricantes, operadoras e centros de pesquisa ligados ao setor de telecomunicações criaram, nos moldes do OMG, o consórcio TINA-C[5] (*Telecommunication Information Networking Architecture - Consortium*), destinado a propor uma arquitetura aberta para nortear o desenvolvimento de sistemas de software para o setor de telecomunicação. A arquitetura TINA adota o RM-ODP como modelo de referência para o processamento distribuído.

1.1. Motivação

As motivações para este trabalho são:

- necessidade de um estudo mais aprofundado da especificação ODP dada a sua grande importância para sistemas distribuídos com o objetivo de traduzir a especificação em arquitetura de implementação, avaliar o modelo face à aplicações complexas e adequar o modelo às aplicações multimídia distribuídas;
- adoção do RM-ODP como modelo de referência para a arquitetura TINA-C [TINA95];
- criação de um componente básico (canal ODP) para a construção de aplicações multimídia distribuídas;
- implementação dos serviços de gerenciamento da especificação ODP já realizada no âmbito da Unicamp [Araújo96];
- disponibilidade de uma infra-estrutura com suporte ao desenvolvimento de aplicações distribuídas orientadas a objetos.

[4] http://www.omg.org/
[5] http://www.tinac.com/

1.2. Objetivos

Os objetivos definidos para este trabalho são:

- estudo da especificação ODP, principalmente seus aspectos de comunicação, identificando os requisitos para o desenvolvimento de aplicações distribuídas;
- proposição de uma arquitetura de implementação para os serviços de comunicação ODP;
- implementação de canais ODP utilizando como infra-estrutura a plataforma CORBA;
- desenvolvimento de uma aplicação multimídia distribuída para avaliação da implementação.

1.3. Apresentação

Este trabalho está estruturado em seis capítulos. O primeiro capítulo é esta introdução. O Capítulo 2 aborda os conceitos relacionados às tecnologias de objetos distribuídos ODP e CORBA que direcionam o desenvolvimento deste trabalho.

O Capítulo 3 apresenta a arquitetura proposta para o canal ODP a partir dos conceitos do capítulo anterior. O Capítulo 4 detalha as considerações, restrições e decisões tomadas no decorrer da implementação do canal ODP. O Capítulo 5 descreve um exemplo de aplicação construído para avaliar a implementação do canal.

O Capítulo 6 finaliza este trabalho fazendo as considerações finais e sugerindo alguns trabalhos futuros.

Capítulo 2

Tecnologias de Objetos Distribuídos

Este capítulo apresenta dois modelos de referência para sistemas distribuídos: ODP e CORBA. O modelo ODP é mais genérico constituindo quase um meta-modelo: pode-se refiná-lo ou especializá-lo para se adequar às necessidades particulares do domínio de aplicação gerando modelos ou padrões específicos. CORBA é uma solução tecnológica mais viável em função dos objetivos práticos do OMG, constituído principalmente por empresas.

2.1. ODP

2.1.1. Objetivos

O objetivo da padronização ODP é o desenvolvimento de uma especificação que permita a realização da distribuição de serviços de processamento de informação em ambiente de recursos tecnológicos heterogêneos e em múltiplos domínios organizacionais. Isto compreende a provisão de componentes de infra-estrutura e funções que acomodem as dificuldades inerentes ao projeto e programação de sistemas distribuídos.

A importância dos sistemas distribuídos se deve ao aumento da necessidade da interconexão de sistemas de processamento de informação. E esta necessidade surge com as tendên-

cias organizacionais como *downsizing*, que demandam muita troca de informação, dentro de uma mesma organização ou entre organizações diferentes. Estes desenvolvimentos são possíveis graças ao avanço tecnológico, dando crescente importância às redes de serviços de informação e estações de trabalho pessoais. Estes fatores permitem a construção de aplicações distribuídas rodando em um grande conjunto de sistemas interconectados.

Sistemas distribuídos apresentam algumas características inerentes:

- Componentes remotos: os componentes de um sistema distribuído podem estar espalhados fisicamente e as interações podem ser locais ou remotas;
- Concorrência: qualquer componente pode executar em paralelo com os outros componentes;
- Ausência de estado global: o estado global de um sistema distribuído não pode ser determinado precisamente;
- Falhas parciais: qualquer componente do sistema pode falhar independentemente dos outros componentes;
- Assincronismo: atividades de comunicação e processamento não são dirigidas por um relógio global;
- Heterogeneidade: não há garantia de que os componentes usem a mesma tecnologia e o conjunto de tecnologias certamente mudará ao longo do tempo. A heterogeneidade aparece em muitos lugares: *hardware*, sistema operacional, rede e protocolos de comunicação, linguagens de programação, aplicações, etc.;
- Autonomia: a autoridade de gerência e controle pode estar fragmentada entre várias entidades. O grau de autonomia especifica a extensão do controle dos recursos de processamento e dispositivos associados pelas entidades autônomas;
- Evolução: durante sua vida útil um sistema distribuído pode ser modificado por progressos técnicos que permitam um melhor rendimento, por decisões estratégicas ou por novos tipos de aplicações;
- Mobilidade: as fontes de informação, os nós de processamento e os usuários podem ser móveis.

Para acomodar estas características os sistemas devem ser construídos com as seguintes propriedades:

- Abertura: propriedade que permite portabilidade e interconexão;
- Integração: propriedade de incorporação de vários recursos e sistemas em um todo sem um trabalhoso desenvolvimento prévio;
- Flexibilidade: propriedade de suportar a evolução do sistema;
- Modularidade: propriedade estrutural definindo que as partes do sistema são autônomas mas interrelacionadas;
- Federação: propriedade que permite combinar sistemas de diferentes domínios técnicos ou administrativos para obter um único objetivo;
- Gerenciabilidade: propriedade de monitoramento, controle e gerenciamento dos recursos do sistema para suportar uma política de configuração e qualidade de serviço;
- Provisão de qualidade de serviço: propriedade de obediência a uma série de requisitos de qualidade no comportamento do sistema;
- Segurança: propriedade de assegurar que as facilidades e dados do sistema estejam protegidos contra acesso não autorizado;
- Transparência: propriedade de mascarar às aplicações os detalhes e diferenças nos mecanismos utilizados para superar os problemas relacionados à distribuição.

A especificação ODP reconhece que não é possível uma única infra-estrutura comum dar suporte a todas as propriedades que um sistema distribuído requer. Ela reconhece, também, a necessidade de se poder selecionar componentes para fornecer uma infra-estrutura de acordo com os requisitos de um tipo particular de aplicação.

2.1.2. Modelo de Referência de Processamento Distribuído Aberto

O Modelo de Referência de Processamento Distribuído Aberto (RM-ODP) consiste de quatro documentos:

- Visão Geral [ODP95a]: contém a motivação do ODP dando o escopo, justificativa e explicações dos conceitos chaves e as linhas gerais da arquitetura ODP. Explica ainda

como o Modelo de Referência deve ser interpretado e aplicado pelos seus usuários. Esta parte não é normativa.

- Fundamentos [ODP95b]: contém a definição de conceitos e da estrutura analítica para uma descrição normalizada de sistemas de processamento distribuído genérico. Este é somente um nível de detalhe para dar suporte ao documento Arquitetura e para estabelecer os requisitos para novas técnicas de especificação. Esta parte é normativa.
- Arquitetura [ODP95c]: contém a especificação das características requeridas que qualificam um sistema distribuído como aberto. Estas são restrições do padrão ODP. Esta parte é normativa.
- Semântica Arquitetônica [ODP95d]: contém a formalização dos conceitos de modelamento ODP definidos no documento Fundamentos. A formalização é alcançada pela interpretação de cada conceito em termos de diferentes técnicas de descrição formal. Esta parte é normativa.

2.1.3. Pontos de vista

A especificação completa de um sistema distribuído não trivial envolve uma quantidade muito grande de informação. Para facilitar esta especificação a maioria das metodologias de desenvolvimento de projetos estabelece uma série de modelos interrelacionados, cada um tentando capturar uma faceta do problema.

Em ODP esta separação de contexto é estabelecida pela identificação de cinco pontos de vista:

- Ponto de vista empresarial: refere-se à proposta, ao escopo e à política para o sistema;
- Ponto de vista de informação: enfoca a semântica da informação e o processamento da informação;
- Ponto de vista computacional: descreve o sistema como um conjunto de objetos que se relacionam através de interfaces, possibilitando a distribuição deste sistema;
- Ponto de vista de engenharia: define o mecanismo e funções requeridas para permitir as interações distribuídas entre os objetos do sistema;

- Ponto de vista tecnológico: trata da escolha da tecnologia para a construção do sistema.

Para representar um sistema ODP, de um determinado ponto de vista, é necessário definir um conjunto estruturado de conceitos em termos de como esta representação pode ser expressa. Este conjunto de conceitos fornece uma linguagem para escrever especificações de sistema daquele ponto de vista. O RM-ODP define uma linguagem para cada ponto de vista contendo conceitos e regras relevantes aos contextos.

Cada ponto de vista é uma abstração que leva à especificação de todo o sistema. Eles devem ser considerados como projeções sobre certos conjuntos de interesse e não como camadas ou métodos de projeto. A arquitetura é expressa em termos do conjunto completo de pontos de vista relacionados, o RM-ODP não estabelece como esta especificação completa deve ser construída.

A Figura 2.1 mostra como os pontos de vista estão relacionados às fases de desenvolvimento de um sistema ODP.

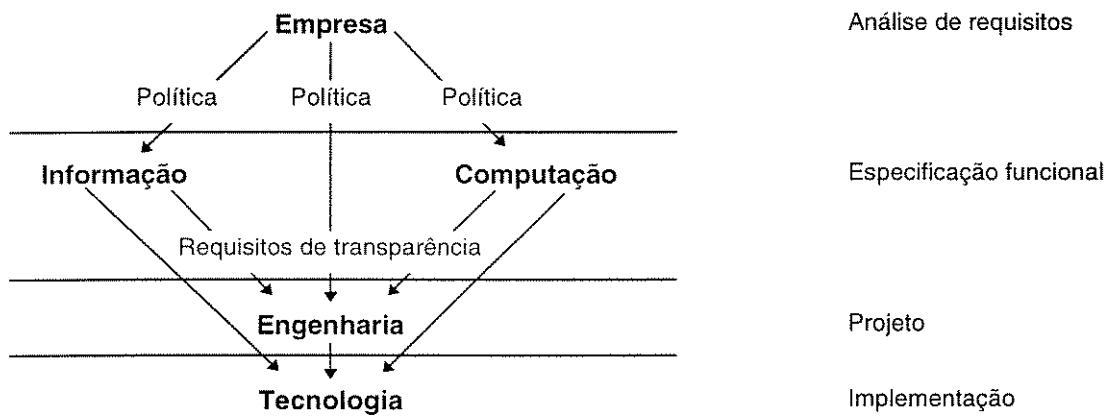

Figura 2.1: Os pontos de vista no ciclo de vida de um sistema ODP.

2.1.4. Transparências

Transparências são utilizadas para esconder do usuário aspectos do sistema distribuído que surgem com a distribuição. O RM-ODP define para um sistema ODP um conjunto de transparências de distribuição que a infra-estrutura deve dar suporte. Estas transparências são:

- Transparência de acesso: mascara as diferenças na representação de dados e o mecanismo de invocação para habilitar a interação entre objetos.
- Transparência de falha: mascara a falha de um objeto e a possível recuperação de outros objetos (ou dele mesmo), permitindo a tolerância a falhas.
- Transparência de localização: mascara o uso de informação sobre a localização espacial de um objeto.
- Transparência de migração: mascara a capacidade do sistema de mudar a localização de um objeto. É freqüentemente usada para obter balanceamento de carga e reduzir latência.
- Transparência de relocação: mascara a relocação de uma interface das outras interfaces ligadas a esta.
- Transparência de replicação: mascara o uso de um grupo de objetos compatíveis para dar suporte a uma interface. É freqüentemente usada para garantir desempenho e disponibilidade.
- Transparência de persistência: mascara a um objeto a desativação e reativação de outros objetos (ou dele mesmo). Desativação e reativação são freqüentemente usadas para manter a persistência de um objeto quando um sistema não consegue provê-la com funções de processamento, armazenamento e comunicação continuamente.
- Transparência de transação: mascara a coordenação de atividades por meio da configuração de objetos para obter consistência.

2.1.5. A Arquitetura ODP

O documento Arquitetura do RM-ODP estabelece um modelo prescritivo que deve ser adotado por um sistema para ele seja caracterizado como um sistema ODP. São utilizados conceitos e terminologias definidos no documento Fundamentos a fim expressar as restrições necessárias para tornar possível a distribuição de forma aberta.

2.1.5.1. Linguagem de empresa

A linguagem de empresa introduz conceitos básicos, necessários para representar um sistema ODP no contexto da empresa na qual ele vai operar. A intenção de uma especificação de empresa é expressar os objetivos e restrições do sistema de interesse. Para isto o sistema é modelado por um ou mais objetos dentro de uma comunidade de objetos, a qual representa a empresa, e, também, pelas regras que relacionam os objetos. Estas regras descrevem, por exemplo, os usuários, proprietários e fornecedores da informação processada pelo sistema. Criando um ponto de vista separado contendo esta informação desassociam-se os objetivos do sistema da maneira como ele vai ser construído.

2.1.5.2. Linguagem de informação

Um sistema ODP pode ser representado pelos objetos de informação e as suas relações. Objetos de informação são abstrações de entidades que existem no mundo real, no sistema ODP ou em outros pontos de vista. A linguagem de informação contém conceitos que permitem a especificação do significado da informação manipulada e armazenada em um sistema ODP, independentemente da maneira como as funções de processamento serão distribuídas.

2.1.5.3. Linguagem de computação

O ponto de vista de computação está relacionado diretamente à distribuição e para isto decompõe o sistema em objetos que realizam funções individuais e interagem através de interfaces. A especificação computacional provê, então, a base às decisões de como será a distribuição das tarefas pois as interfaces podem estar localizadas independentemente, assumindo que mecanismos de comunicação podem ser definidos para dar suporte ao comportamento das interfaces.

O coração da linguagem computacional é o modelo de objetos, que define as interfaces que um objeto pode ter, a maneira de se ligar às interfaces e as formas de interação entre elas. A linguagem define ainda as ações que um objeto pode realizar de tal maneira que novos objetos e interfaces possam ser criados e possam ser estabelecidas ligações entre elas.

A linguagem de computação possibilita a especificação de restrições à distribuição de uma aplicação sem especificar o grau real da distribuição. Isto assegura que as aplicações não estabeleçam nenhuma suposição que afete a distribuição dos seus componentes. Por conse-

qüência, a configuração e o grau de distribuição do hardware, sobre o qual as aplicações estão rodando, pode ser facilmente alterado, sujeito às restrições de ambiente, sem maior impacto sobre os softwares aplicativos.

As interações entre objetos computacionais ocorrem em uma de suas interfaces computacionais e podem ser de três formas: sinais, operações ou fluxos.

Os sinais são o mais baixo nível de descrição de interações entre objetos. Um sinal é uma ação atômica resultante de comunicação unidirecional entre dois objetos. As operações e os fluxos podem ser definidos em termos de sinais.

As operações refletem o paradigma cliente/servidor. Existem dois tipos de operações: indagação em que o servidor retorna um sinal de resposta ao sinal de requisição do cliente e anúncio em que não há resposta à requisição.

Os fluxos são abstrações de seqüências contínuas de dados (sinais) transmitidas entre interfaces. Eles podem ser utilizados, por exemplo, para modelar fluxo de informação de áudio ou vídeo em uma aplicação multimídia. Um fluxo é caracterizado por seu nome e tipo, que especifica a natureza e formato dos dados trocados. A semântica exata do fluxo não é definida no modelo computacional.

2.1.5.4. Linguagem de engenharia

A linguagem de engenharia está focada na maneira como as interações entre objetos são realizadas e os recursos necessários. O ponto de vista de computação está preocupado com quando e porque os objetos interagem, enquanto o ponto de vista de engenharia se preocupa com como eles interagem. Nesta linguagem o maior interesse é o suporte às interações individuais entre objetos computacionais. Aqui, os objetos computacionais são vistos como objetos básicos de engenharia e as ligações entre eles como canais.

Um objeto básico de engenharia é um objeto que requer o suporte de uma infra-estrutura distribuída. A linguagem de engenharia lida com objetos básicos de engenharia e os vários outros objetos que dão suporte a eles. Além disso, relaciona os objetos aos recursos disponíveis do sistema pela identificação de uma série de agrupamentos aninhados conforme ilustrado na Figura 2.2.

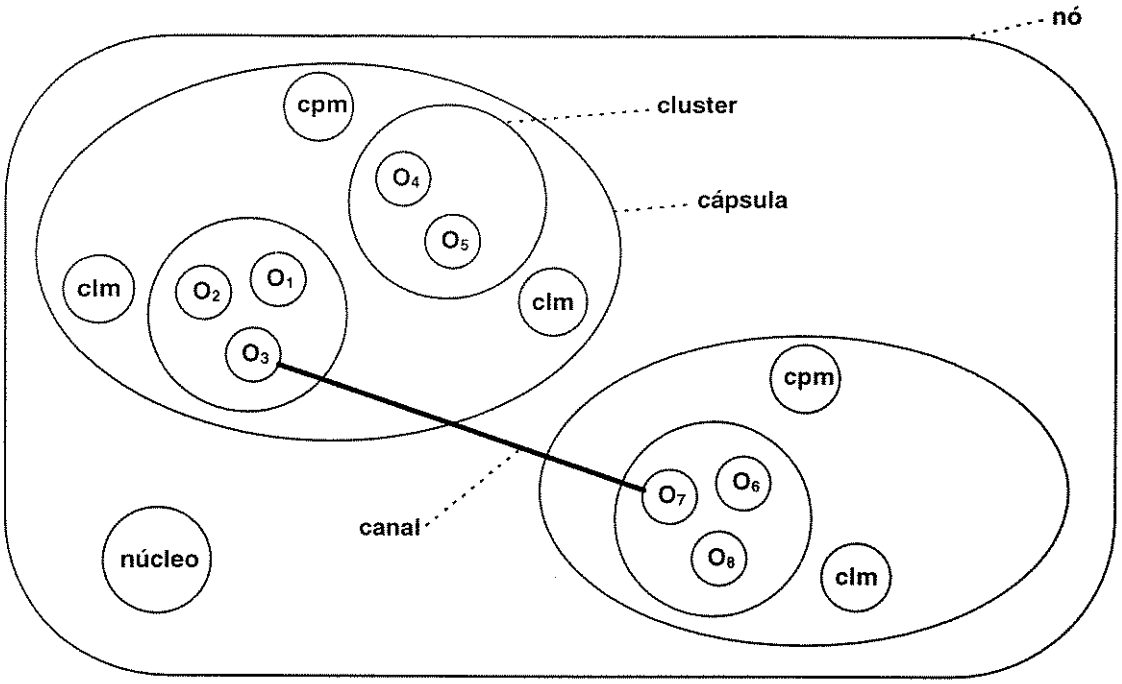

Figura 2.2: Nó, cápsulas e clusters.

No nível mais externo, os objetos são agrupados em uma unidade com o propósito de localização no espaço e que incorpora um conjunto de funções de processamento, armazenamento e comunicação (o nó). Um nó pode ser qualquer visão de recursos fortemente integrada, de tal maneira que o projetista do sistema possa considerá-la um todo. Um computador e o software que ele suporta (sistema operacional e aplicações) é um exemplo de nó.

O nó está sob controle do núcleo que é o objeto de engenharia responsável pela iniciação e criação de grupos de objetos, disponibilização das facilidades de comunicação e fornecimento dos serviços básicos. Um exemplo de núcleo seria o sistema operacional.

Dentro de um nó pode haver um número de cápsulas. A cápsula é uma configuração de objetos de engenharia formando uma unidade com o propósito de encapsulamento de processamento e armazenamento. Um processo do sistema operacional pode ser considerado um exemplo de cápsula. Associado a cada cápsula existe o gerente de cápsula (cpm) cuja função é a gerência dos objetos de engenharia que a compõem.

O menor agrupamento de objetos é o cluster cuja finalidade é a redução do custo de sua manipulação. A linguagem de engenharia introduz os conceitos de *checkpoint* e de clonagem.

Checkpoint é uma especificação das características derivadas do estado e da estrutura de um objeto que pode ser utilizada para instanciar outro objeto de engenharia, consistente com o estado do objeto original. Clonagem corresponde à instanciação de um cluster a partir do *checkpoint*. Os objetos em um cluster podem sofrer *checkpointing* (criação de um *checkpoint*), desativação (*checkpointing* seguido da eliminação do cluster), recuperação (clonagem do cluster após falha ou eliminação), reativação (clonagem do cluster após sua desativação) e migração (mudança do cluster para uma cápsula diferente). A atividade de gerência de objetos do cluster é de responsabilidade do objeto de engenharia denominado gerente de cluster (clm).

Quando objetos de diferentes clusters interagem existe a necessidade de um mecanismo de suporte à interação. Mesmo que os objetos estejam no mesmo nó ou na mesma cápsula este mecanismo é necessário em virtude da possibilidade de que um dos objetos termine, falhe ou migre. O conjunto de mecanismos necessários para este fim constitui o canal que é composto de uma série de objetos de engenharia: *stubs*, *binders*, *protocols* e *interceptors*, como pode ser visto na Figura 2.3.

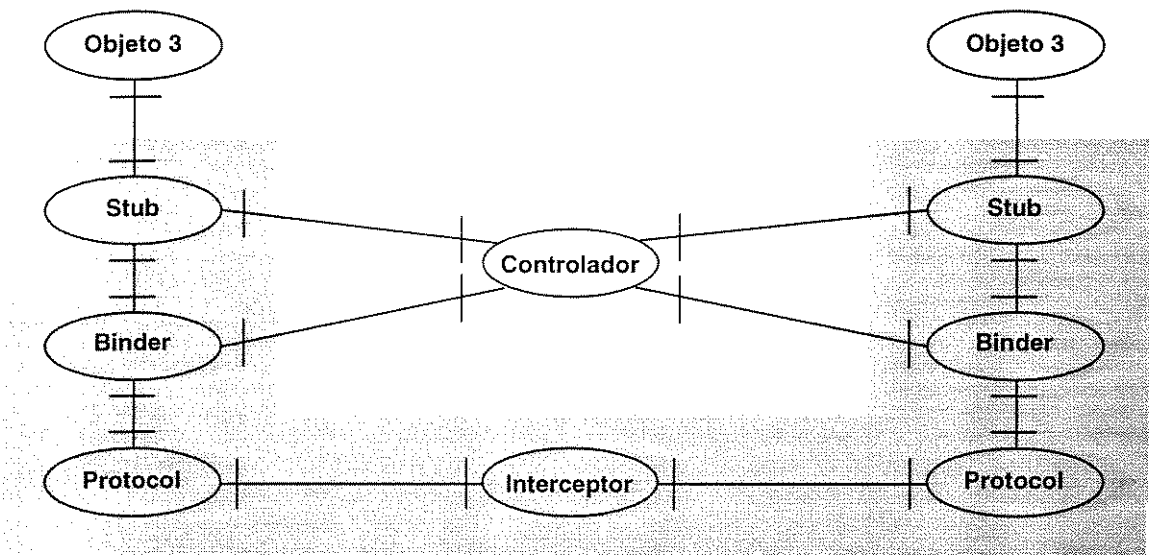

Figura 2.3: Um exemplo de canal entre dois objetos.

O *stub* interage diretamente com o objeto básico de engenharia e interpreta as interações transportadas pelo canal e executam qualquer transformação ou monitoramento necessários

baseados nesta interpretação. Portanto, o *stub* realiza funções como conversão, codificação e decodificação de dados, serialização de parâmetros e registro de informação sobre as interações. Para isto, o *stub* tem acesso à informação sobre o tipo de interação (tipo da interface), distinguindo-o do *binders* e *protocol* que transmitem a informação sem se preocuparem com a sua estrutura interna. O *stub* pode aplicar controle sobre recursos ou manter relatório. Ele pode, ainda, interagir com objetos de engenharia fora do canal.

A principal função do *binder* é a manutenção da integridade fim-a-fim do canal. Para isto ele mantém informações sobre o canal e a qualidade de serviço requisitada, monitora a migração ou falha de objetos e recupera conexões quebradas. O *binder* estabelece a ligação quando o canal é criado e sua interface de controle permite alterar a configuração do canal e destruir todo ou parte do canal. Ele pode, também, interagir com objetos de engenharia externos ao canal para realizar suas funções se for necessário.

O objeto *protocol* se comunica com outro objeto *protocol* do mesmo canal para obter a interação entre objetos de engenharia. Ele detém informação sobre o protocolo que está sendo usado. Se for necessário também pode interagir com outros objetos de engenharia fora do canal. Quando em um mesmo canal existirem *protocols* de tipos diferentes é necessária a presença de *interceptors*.

O *interceptor* é um objeto de engenharia que está presente na fronteira de dois domínios. Ele faz verificações para reforçar ou monitorar políticas e transformações para mascarar diferenças de dados entre domínios diferentes. Um exemplo de *interceptor* é um *gateway*.

O comportamento de um canal, com respeito à configuração e qualidade de serviço, é controlado via interfaces de controle de *stub*, *binder*, *protocol* e *interceptor* sendo que estas interfaces são opcionais. Pode existir um objeto controlador de canal apresentando uma interface de controle que abstrai as interfaces de controle de *stub*, *binder*, *protocol* e *interceptor*.

Para estabelecer um canal entre objetos de engenharia é usada a referência de interface que identifica e descreve uma interface de engenharia com detalhes suficientes para permitir a ligação a ela. Esta referência pode ser obtida através de uma interação entre objetos de engenharia permitindo ao objeto que a recebeu iniciar o canal sem qualquer outra informação.

O estabelecimento do canal está a cargo do núcleo que é o responsável pelas funções de gerência e de comunicação de um nó. Para realizar sua tarefa o núcleo apresenta interface com funções para: disponibilizar uma interface de engenharia para ligação a objetos em outras cáp-

sulas, estabelecer a ligação entre um objeto de engenharia de uma cápsula e um conjunto de outros objetos de engenharia e determinar o tipo de canal e a interface de comunicação para uma interface de engenharia específica.

A transparência de distribuição na interação entre objetos de engenharia que o canal deve suportar inclui a interação entre um objeto cliente e um servidor, um grupo de objetos interagindo com outro grupo e fluxo de dados envolvendo múltiplos produtores e múltiplos consumidores.

2.1.5.5. Linguagem de tecnologia

A linguagem de tecnologia descreve a implementação do sistema ODP em termos da configuração de objetos representando o hardware e os componentes de software. Existe, neste caso, restrições de custo e tecnologia disponível. Esta linguagem estabelece uma ligação entre o conjunto de especificações dos ponto de vista e a implementação real.

2.2. CORBA

O consórcio OMG introduziu o modelo de referência OMA (*Object Management Architecture*) como uma resposta ao objetivo de produzir especificações para sistemas distribuídos orientados a objetos visando reusabilidade, portabilidade e interoperabilidade. Como mostra a Figura 2.4 o modelo compreende cinco componentes:

Os Serviços de Objetos (*Objects Services*) definem, por exemplo, a gerência do ciclo de vida de objetos. Neste serviço são fornecidas interfaces para criar objetos, controlar acesso a objetos, manter o mapeamento de objetos relocados e controlar a relação entre tipos de objetos (gerência de classe). Exemplos de outros Serviços de Objetos são: serviço de nome de objetos, notificação de eventos, persistência de objetos e transação. Estes serviços fornecem consistência a aplicação e auxiliam o aumento da produtividade do programador.

As Facilidades Comuns (*Common Facilities*) correspondem a um conjunto de aplicações genéricas que podem ser configuradas de acordo com os requisitos específicos de uma determinada aplicação. Estas facilidades de situam na nível de usuário, como por exemplo, facilidades de impressão, correio eletrônico ou gerência de documentos.

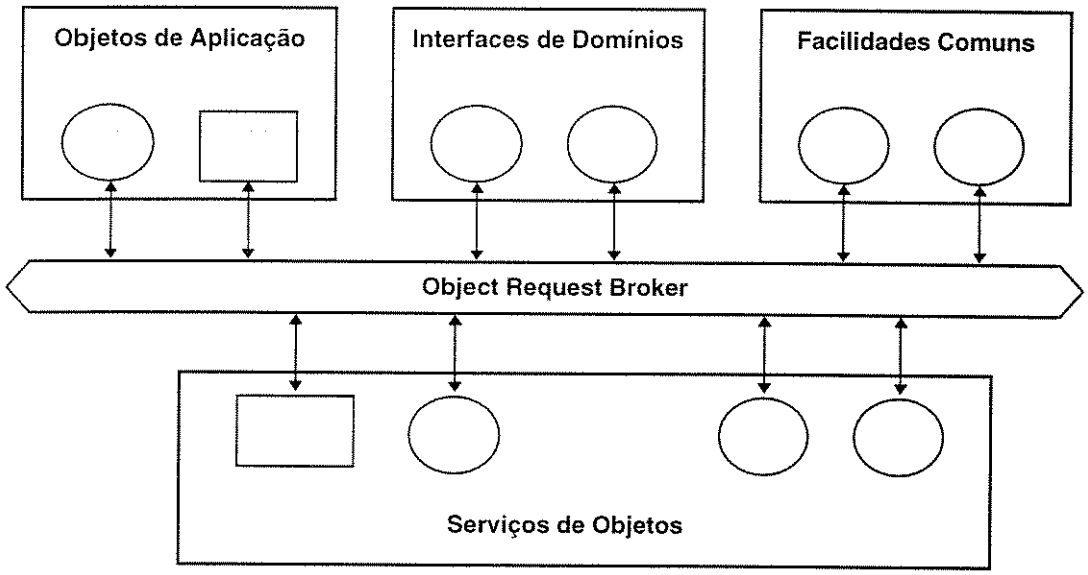

Figura 2.4: O Modelo de Referência OMA

As Interfaces de Domínio (*Domain Interfaces*) representam áreas verticais que fornecem funcionalidades de interesse do usuário final em domínios de aplicações particulares. As interfaces podem combinar algumas Facilidades Comuns e Serviços de Objetos mas são projetadas para realizar tarefas particulares dentro de mercados ou indústrias específicos.

Objetos de Aplicação (*Application Objects*) representam componentes (objetos) que realizam tarefas particulares ao usuário final. Uma aplicação é tipicamente construída a partir de um grande número de objetos básicos - alguns específicos à aplicação, outros ao domínio, alguns construídos a partir de Serviços de Objetos e outros de um conjunto de Facilidades Comuns.

Object Request Broker (ORB) é o mecanismo de comunicação da arquitetura. O ORB fornece uma infra-estrutura que permite a interação entre objetos independente da plataforma e técnicas utilizadas na implementação destes objetos. O ORB é o principal componente da arquitetura.

Para que diferentes implementações ORB possam fornecer serviços e interfaces comuns foi definido pelo grupo OMG a especificação CORBA [CORBA95]. Esta especificação utiliza o paradigma cliente-servidor. O cliente é a entidade que solicita uma operação em um objeto e a implementação do objeto corresponde ao código e dados que realmente executam o serviço

desejado. O ORB é responsável por todos os mecanismos necessários para localizar a implementação do objeto, para preparar a implementação para receber a requisição e para transmitir os dados da requisição. A interface que o cliente vê é completamente independente de onde o objeto servidor está localizado, qual a linguagem de programação usada na sua implementação e qualquer outro aspecto que não esteja refletido na sua interface.

2.2.1. Estrutura de um ORB

A Figura 2.5 mostra a estrutura de um ORB na arquitetura CORBA. As setas indicam o sentido da requisição.

Para fazer uma requisição o cliente pode usar a interface de invocação dinâmica ou o *stub* IDL. O cliente pode ainda interagir diretamente com o ORB para algumas funções. A implementação do objeto recebe uma requisição através do *skeleton* IDL ou do *skeleton* dinâmico. A implementação do objeto pode chamar o adaptador de objeto e o ORB enquanto estiver processando uma requisição ou a qualquer momento.

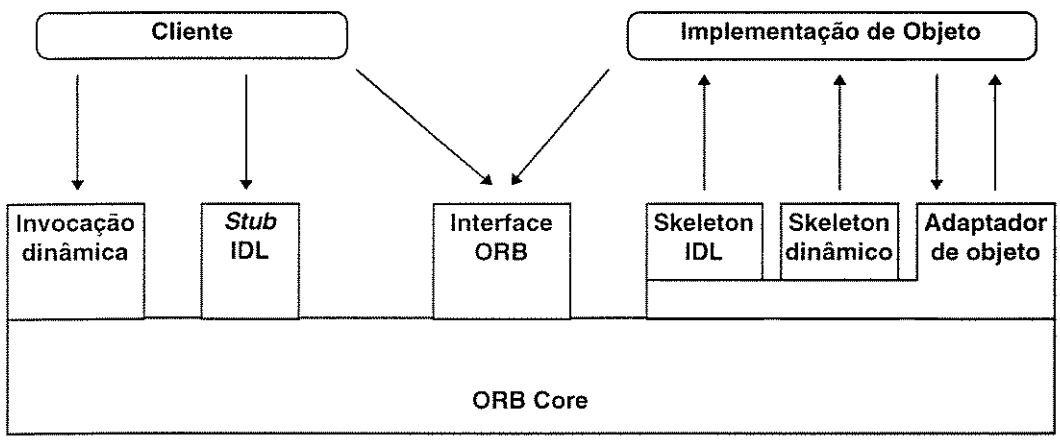

Figura 2.5: A estrutura de um ORB

As definições de interfaces de objetos podem ser feitas de duas maneiras. As interfaces podem ser definidas estaticamente através da linguagem IDL (*Interface Definition Language*). Esta linguagem define os tipos de objetos de acordo com as operações que eles podem executar e os parâmetros destas operações. As interfaces podem também ser adicionadas a um ser-

viço de Repositório de Interfaces. Este serviço representa os componentes da interface como objetos permitindo acesso a estes componentes em tempo de execução.

O cliente executa uma requisição tendo acesso a uma referência de objeto, sabendo o tipo do objeto e a operação que deseja executar. O cliente inicia a requisição pela chamada da rotina *stub* que é específica ao objeto ou pela construção de uma requisição dinamicamente.

O ORB localiza o código de implementação apropriado, transmite os parâmetros e transfere o controle à implementação de objeto através de um *skeleton* IDL ou *skeleton* dinâmico. Ao executar uma operação a implementação de objeto pode obter alguns serviços do ORB através do adaptador de objeto. Quando a requisição está completa o controle e os valores de saída são retornados ao cliente.

2.2.1.1. Object Request Broker

Na arquitetura CORBA não é exigido que o ORB seja implementado como um componente único mas que seja definido por suas interfaces. Qualquer implementação ORB que forneça a interface apropriada é aceitável. A interface é organizada em três categorias: operações idênticas para todas as implementações ORB, operações específicas a um tipo particular de objeto e operações específicas a um estilo particular de implementação de objeto.

Diferentes ORBs podem fazer diferentes escolhas de implementação e juntos com compiladores IDL, repositórios e vários adaptadores de objeto proverem um conjunto de serviços aos clientes e implementações de objetos com diferentes propriedades e qualidades. Podem existir múltiplas implementações ORB com diferentes representações para a referência de objetos e diferentes meios de executar invocações. Pode ser possível ao cliente ter acesso a duas referências de objetos gerenciadas por implementações ORB distintas simultaneamente. Quando dois ORBs pretendem trabalhar juntos devem conseguir distinguir suas referências de objetos, pois isto não é responsabilidade do cliente.

O ORB core é a parte do ORB que provê a representação básica de objetos e a comunicação das requisições. A arquitetura CORBA é projetada para dar suporte a diferentes mecanismos de objetos. Isto é realizado estruturando-se componentes sobre o ORB core o qual provê interfaces que mascaram as diferenças entre os ORB cores.

2.2.1.2. Clientes

Um cliente de um objeto tem acesso à sua referência e invoca operações sobre esta. Um cliente só tem conhecimento sobre a estrutura lógica do objeto de acordo com sua interface e sabe o seu comportamento através das invocações. Embora geralmente considere-se o cliente como sendo um programa ou processo iniciando pedidos em um objeto, é importante reconhecer que qualquer entidade pode ser um cliente de um objeto particular. Por exemplo, a implementação de um objeto pode ser cliente de outros objetos.

Os clientes são portáveis e devem poder trabalhar sem modificação de código em qualquer ORB que dê suporte ao mapeamento da linguagem desejada com qualquer instância do objeto que implementa a interface desejada. Os clientes não têm conhecimento da implementação do objeto, qual adaptador de objetos é usado ou qual o ORB é usado para acessá-la.

2.2.1.3. Stubs IDL

Para o mapeamento a uma linguagem não orientada a objetos deve existir uma interface de programação de *stubs* para cada tipo de interface. Os *stubs* fazem chamadas ao resto do ORB utilizando interfaces que são privadas a, e presumivelmente otimizadas para, um ORB core específico. Se mais de um ORB estão disponíveis devem existir diferentes *stubs* correspondendo aos diferentes ORBs. Neste caso, é necessária a cooperação entre o ORB e o mapeamento da linguagem para associar os *stubs* corretos com as referências de objetos. As linguagens de programação orientadas a objeto, como C++ e Smalltalk, não necessitam de interfaces *stub*.

2.2.1.4. Invocação Dinâmica

É uma interface que permite a construção dinâmica de invocações a objetos, ou seja, ao invés de chamar uma rotina *stub* que é específica a uma operação particular de um objeto particular, um cliente pode especificar o objeto a ser invocado, a operação a ser executada e o conjunto de parâmetros da operação através de uma chamada ou uma seqüência de chamadas. O código do cliente deve fornecer informação sobre a operação a ser executada e os tipos dos parâmetros que estão sendo passados (provavelmente obtidos do Repositório de Interfaces ou de outra fonte). A natureza da interface de invocação dinâmica pode variar substancialmente de uma linguagem de programação para outra.

2.2.1.5. Implementação de objetos

Uma implementação de objeto fornece a semântica do objeto geralmente definindo dados para a instância do objeto e código para os métodos do objeto. Freqüentemente a implementação usará outros objetos ou software adicional para implementar o comportamento do objeto. Em alguns casos, a função primordial do objeto é provocar efeitos em entidades que não são objetos.

Uma variedade de implementações de objeto pode ser suportada, incluindo servidores separados, bibliotecas, programa por método, aplicação encapsulada, banco de dados orientado a objeto, etc. Através do uso de adaptadores de objeto adicionais é possível suportar virtualmente qualquer estilo de implementação de objeto.

Geralmente, implementações de objeto não dependem do ORB ou como o cliente faz a invocação. As implementações podem selecionar interfaces para serviços dependentes do ORB pela escolha do adaptador de objeto.

2.2.1.6. Skeleton IDL estático

Para uma linguagem de mapeamento particular, e possivelmente dependendo do adaptador de objeto, existirá uma interface aos métodos que implementam cada tipo de objeto. Nesta interface a implementação do objeto escreve a rotina que está em conformidade com a interface IDL e o ORB as chama através do skeleton. A existência de skeleton não implica na existência do correspondente *stub* do cliente, os clientes podem fazer requisições via interface de invocação dinâmica.

2.2.1.7. Skeleton dinâmico

É uma interface disponível que permite tratamento dinâmico de invocações de objetos. Ao invés de ser acessado via *skeleton*, que é específico a uma operação particular, uma implementação de objeto é alcançada através de uma interface que dá acesso ao nome e parâmetros da operação de maneira análoga à interface de invocação dinâmica do lado cliente. Podem ser usados conhecimentos puramente estáticos ou dinâmicos (possivelmente determinados através do Repositório de Interfaces) para determinar os parâmetros.

O código da implementação deve prover descrição para todos os parâmetros ao ORB e este por sua vez fornece os valores de qualquer entrada de parâmetros para o uso na operação. Depois de executada a operação o *skeleton* dinâmico fornece os parâmetros de saída ou uma exceção ao ORB. Os *skeletons* dinâmicos podem ser invocados através de *stubs* do cliente ou da interface de invocação dinâmica, sendo o resultado da requisição é idêntico.

2.2.1.8. Adaptador de objeto

O adaptador de objeto é a maneira primária pela qual uma implementação de objeto acessa os serviços fornecidos pelo ORB. Os serviços oferecidos pelo ORB ao adaptador de objeto incluem: geração e interpretação de referências de objetos, invocação de método, segurança de interações, ativação e desativação de objetos e implementações, mapeamento de referências de objetos para implementações e registro de implementações.

O amplo espectro de granularidade, tempo de vida, política, estilo de implementação e outras propriedades de objetos torna difícil ao ORB core prover uma única interface que seja conveniente e eficiente a todos os objetos. Portanto, através dos adaptadores de objeto é possível ao ORB core atingir grupos particulares de implementação de objetos com características similares.

2.2.1.9. Interface do ORB

A interface ORB é idêntica para todos ORBs. Como a maioria das funcionalidades do ORB estão nos adaptadores de objeto, *stubs*, *skeletons* ou interfaces de invocação dinâmica, existem poucas operações que são comuns a todos os objetos. Estas operações são úteis aos clientes e às implementações de objeto.

2.2.1.10. Repositório de interface

O repositório de interface é um serviço que provê objetos persistentes que representam as informações IDL disponibilizadas em tempo de execução. Estas informações podem ser utilizadas pelo ORB para executar requisições. Além disso, com a informação do repositório um programa pode encontrar um objeto cuja interface não era conhecida quando o programa foi compilado, e ainda, ser capaz de determinar quais operações são válidas no objeto e fazer uma invocação sobre ele.

Além dessa funcionalidade, o repositório de interfaces é um lugar comum para armazenar informações adicionais associadas à interface, por exemplo, informação para depurar, bibliotecas de *stubs* e *skeletons*, rotinas que podem localizar determinados tipos de objetos, etc.

2.2.1.11. Repositório de implementação

O repositório de implementação contém informação que permite ao ORB localizar e ativar implementações de objetos. Embora a maior parte da informação seja específica ao ORB ou ambiente de operação, este é o local mais adequado para o seu armazenamento. Normalmente instalação de implementações e controle de políticas de ativação e execução de implementações de objetos são realizadas através de operações no repositório de implementação.

Além dessa funcionalidade, o repositório de implementação é um lugar comum para armazenar informações adicionais associadas a implementações, por exemplo, informação para depurar, controle administrativo, alocação de recursos, segurança, etc.

2.2.2. Referências de objetos

Uma referência de objeto é a informação necessária para especificar um objeto em um ORB. As implementações de ORBs podem diferir na escolha da representação de referências de objetos. A referência de objeto manipulada por um cliente só é válida durante o ciclo de vida deste cliente.

Todos ORBs devem prover o mesmo mapeamento, em uma linguagem de programação particular, para uma referência de objeto, permitindo ao programa acessar referências de objeto independente do ORB. O mapeamento da linguagem pode prover também maneiras adicionais para acessar referências de objeto de uma maneira conveniente.

2.2.3. IDL - Interface Definition Language

A linguagem IDL define os tipos de objetos pela especificação de suas interfaces. Uma interface consiste de um conjunto de operações e seus parâmetros. Embora IDL forneça um *framework* conceitual para descrição de objetos manipulados pelo ORB não é necessário haver código fonte IDL disponível para o ORB operar. Tão logo a informação equivalente esteja

disponível, na forma de rotinas *stub* ou repositório de interface de tempo de execução, um ORB particular pode ser capaz de funcionar corretamente.

IDL é o meio pelo qual uma implementação de objeto particular diz aos seus potenciais clientes quais operações estão disponíveis e como elas devem ser invocadas. Da definição IDL é possível mapear objetos CORBA em objetos de uma linguagem de programação específica ou sistema de objetos.

2.2.3.1. Mapeamento de IDL para linguagens de programação

Diferentes linguagens de programação podem preferir acessar objetos CORBA de diferentes maneiras. Para linguagens orientadas a objeto é desejável ver objetos CORBA como objetos da linguagem. Mesmo para linguagens não orientadas a objeto é uma boa idéia esconder a exata representação ORB das referências de objetos, nomes de métodos, etc. Um mapeamento específico de IDL para uma linguagem de programação deve ser o mesmo para todas as implementações ORB. Isto inclui a definição dos tipos de dados da linguagem, interfaces para acessar objetos através do ORB, estrutura da interface *stub* do cliente (não necessária para linguagens orientadas a objeto), interface de invocação dinâmica, skeletons da implementação, adaptadores de objeto e interface do ORB. O mapeamento da linguagem também define as interações entre invocações de objetos e as *threads* que controlam o cliente ou a implementação.

2.2.4. Desenvolvimento em CORBA

A Figura 2.6 mostra o processo de desenvolvimento de aplicações em ambiente CORBA. Inicialmente são definidas as interfaces em IDL. Esta definição é utilizada pelo compilador para gerar os *stubs* do cliente e os *skeletons* do servidor. Além disso, as definições são armazenadas no repositório de interfaces.

Depois de geradas as implementações de cliente e servidor, seus códigos são ligados respectivamente aos *stubs* e *skeletons*. O último passo é registrar o servidor no repositório de implementação.

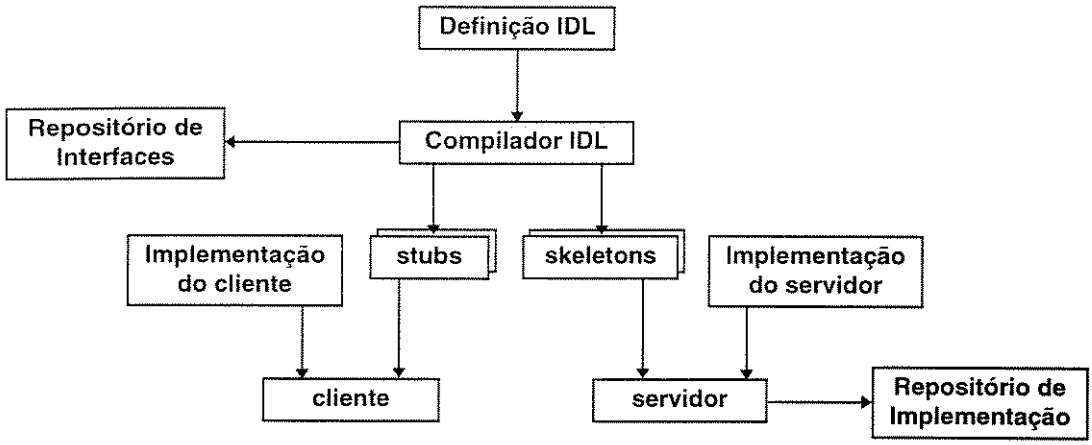

Figura 2.6: Desenvolvimento em CORBA

2.3. Relação entre ODP e CORBA

Tanto a ISO quanto a OMG têm por objetivo o desenvolvimento de um conjunto de especificações para o suporte a sistemas de objetos distribuídos. Por este motivo está sendo feita uma tentativa de unir esforços através da fusão das arquiteturas ODP e OMA em uma arquitetura unificada que servirá de base para os trabalhos futuros de cada organização. Desde 1994 representantes da ISO participam oficialmente em certos comitês da OMG [Siegel94]. Ainda está em discussão como cada uma das partes efetivará este trabalho conjunto [ISO97].

Esta ligação é de interesse mútuo devido às características particulares das organizações. A ISO é a organização de especificação internacional reconhecida e sua sanção tem enorme impacto em certos mercados. Além disso, a arquitetura ODP está sendo desenvolvida em colaboração com a ITU-T o que garante o atendimento aos requisitos da indústria de telecomunicações. Por outro lado, a OMG tem um vasto conjunto de membros de grande influência e sua especificação forma a base para um corpo grande e crescente de implementações.

Capítulo 3

Arquitetura

A seguir é apresentada uma proposta de implementação para os serviços de comunicação definidos na arquitetura ODP. Esta proposta se desenvolve a partir do estudo da especificação com o objetivo de levantar os componentes, suas respectivas funcionalidades e os relacionamentos relevantes aos serviços em questão. Neste capítulo são apresentados os resultados deste estudo e as decisões de projeto tomadas de acordo com interpretações da especificação e motivadas pela adequação aos propósitos do trabalho. Os detalhes da implementação são discutidas no Capítulo 4.

Por serviços de comunicação entende-se a disponibilização de meios pelos quais pode ser estabelecida e controlada a transmissão de informação entre objetos de acordo com os preceitos da especificação ODP. A arquitetura completa ODP compreende ainda alguns outros serviços. Uma proposta para implementação de serviços de gerenciamento de nó, cápsula, *cluster* e objeto foi desenvolvida no âmbito da Unicamp por [Araújo96].

Para implementação dos serviços de comunicação foi utilizada uma plataforma comercial baseada na arquitetura CORBA. O trabalho foi coordenado no sentido de possibilitar sua integração aos serviços de gerenciamento já desenvolvidos.

3.1. Serviço de comunicação

A linguagem de engenharia ODP, conforme discutido na seção 2.1.5.4, estabelece que a comunicação entre objetos básicos de engenharia de diferentes *clusters* é realizada pelo canal. O canal é uma composição de objetos *stub*, *binder*, *protocol* e *interceptor*. Cada um destes objetos possui uma funcionalidade específica apresentada a seguir.

Os *stubs* interagem diretamente com os objetos que estão se comunicando e atuam sobre as informações transmitidas pelo canal. Eles são responsáveis por:

- transformação (conversão, codificação, decodificação, etc.) de dados;
- registro de informações sobre as interações;
- controle de recursos locais;
- notificação de eventos.

A principal função dos *binders* é a gerência da integridade fim-a-fim do canal. Para realizar esta tarefa eles são responsáveis por:

- manutenção de informações sobre o canal e sobre a qualidade de serviço (QoS) requisitada;
- monitoramento de migração e falhas de objetos;
- recuperação de conexões quebradas;
- alteração da configuração do canal;
- destruição do canal.

Os *protocols* de um canal estão ligados entre si permitindo a interação remota entre objetos de diferentes nós.

O *interceptor* é utilizado para verificar e transformar as interações quando os objetos que estão se comunicando através do canal pertencerem a domínios diferentes.

O controlador de canal apresenta uma interface de controle que abstrai as interfaces de controle de *stubs*, *binders*, *protocols* e *interceptors* permitindo a gerência do canal através de uma única interface.

O núcleo é responsável pelo estabelecimento do canal. Sua interface de controle deve apresentar funções que permitam:

- disponibilizar uma interface para ligação a outros objetos;
- estabelecer a ligação entre um objeto e um conjunto de outros objetos;
- determinar o tipo de canal e a interface de comunicação para uma interface específica.

Considera-se neste trabalho que os objetos que utilizarão o canal pertencem ao mesmo domínio tornando-se, portanto, desnecessária a adoção de políticas ou a transformação de dados. Como a atribuição dos *interceptors* deixa de existir eles não estão presentes nesta proposta de implementação de canal ODP. A Tabela 2.1 apresenta os objetos relacionados ao canal com sua respectiva funcionalidade.

Tabela 3.1: Os objetos relacionados ao canal.

Objeto	Função
stub	• transformação de dados • registro de informações sobre as interações • controle de recursos locais • notificação de eventos
binder	• manutenção de informações sobre o canal e QoS requisitado • monitoramento de migração e falhas de objetos • recuperação de conexões quebradas • alteração da configuração do canal • destruição do canal
protocol	• interação remota entre objetos de diferentes nós
núcleo	• disponibilização de interface para ligação a outros objetos • estabelecimento de ligação entre objetos • determinação do tipo de canal e da interface de comunicação
controlador de canal	• disponibilização de uma única interface para gerência do canal

As interações entre objetos ODP podem ser de dois tipos: interações de operação ou de fluxo. Uma operação é uma interação realizada entre um cliente que faz uma requisição e um servidor que executa a requisição e retorna os resultados se necessário. A interação do tipo

fluxo corresponde à transmissão contínua de dados entre um objeto produtor que gera os dados e um objeto consumidor que os consome. O canal ODP deve ser utilizado nos dois casos para objetos de diferentes *clusters*.

A arquitetura CORBA, conforme apresentado na seção 2.2, adota o paradigma cliente-servidor, portanto a utilização de uma implementação CORBA fornece as interações de operação entre seus objetos.

O uso de CORBA não é adequado para transmissão de fluxo por dois motivos. Primeiro, a linguagem de definição de interfaces IDL ainda não prevê a definição de interfaces de fluxo, embora o OMG reconheça a necessidade da adoção de uma especificação neste sentido e já tenha iniciado o processo [OMG96]. Segundo, as implementações CORBA são desenvolvidas para atender o requisito de confiabilidade, ou seja, é necessário garantir que tanto a invocação da operação pelo cliente em um servidor quanto os valores de retorno serão entregues, mesmo que tenha que ser feita a retransmissão de dados. No caso do fluxo, geralmente, a prioridade se inverte, os requisitos de temporais como atraso e variação do atraso são mais importantes do que a confiabilidade. Por exemplo, para fluxos de áudio e vídeo em teleconferência opta-se por perder parte dos dados a atrasar o fluxo.

3.2. Descrição da implementação

Dentro deste contexto é proposto uma canal no qual o controle é feito por intermédio de uma implementação CORBA e para a transmissão de fluxo é utilizado um sistema de transporte alternativo. A escolha do sistema de transporte é livre e decida em função da qualidade de serviço requisitada e da disponibilidade de infra-estrutura.

A Figura 3.1 apresenta a estrutura da proposta de implementação de canal ODP. Alguns objetos possuem interface IDL e neste caso é representada graficamente por um T invertido. As setas indicam a transmissão de fluxo de dados e as áreas sombreadas a parte do canal no lado produtor ou lado consumidor.

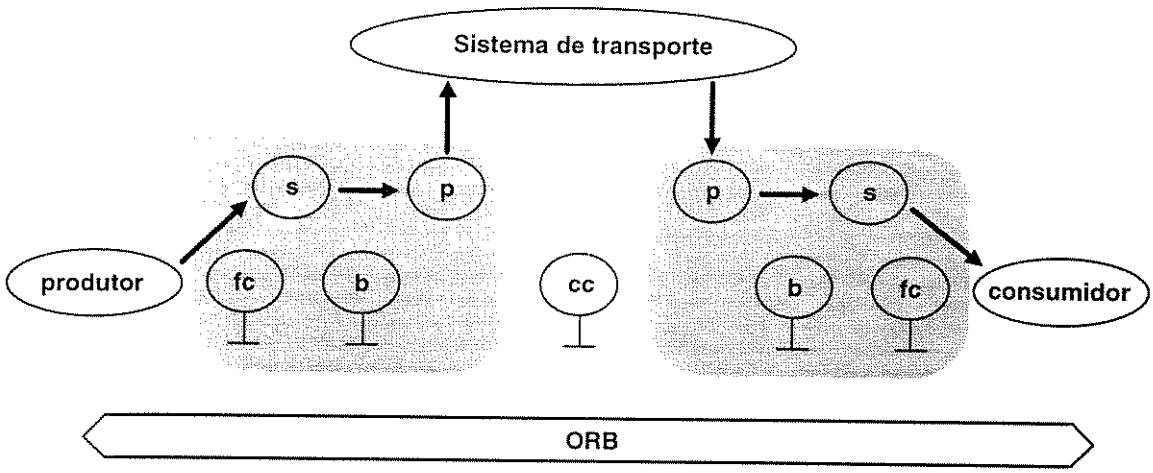

Figura 3.1: Arquitetura proposta para implementação de canal ODP.

As atribuições do núcleo, criação dos objetos do canal e estabelecimento de ligação entre eles, estão centralizadas no objeto **fábrica de canal** (fc). A interface deste objeto apresenta os seguintes métodos:

- initChannel: cria os objetos do canal no lado produtor de fluxo;
- bindChannel: cria os objetos do canal no lado consumidor de fluxo e estabelece a ligação ao lado produtor.

O ***binder*** (b) para manutenção da integridade fim-a-fim do canal controla o *stub* e o protocol associado a ele. O fluxo não é transmitido através do *binder* pois para exercer sua funcionalidade basta que tenha acesso à interface do *stub* e *protocol*. Consegue-se assim aumentar o desempenho da implementação através da diminuição do número de cópias de dados. A interface de controle do *binder* possui os seguintes métodos:

- Start: dá início à transmissão;
- Pause: suspende a transmissão;
- Continue: continua a transmissão;
- modifyChannel: modifica a configuração do canal de acordo com novos requisitos de QoS;

- getChannel: retorna os parâmetros utilizados na criação do canal;
- getStatus: retorna medidas de monitoramento do canal.
- Delete: elimina parte do canal (*stub* + *binder* + *protocol*)

O **controlador de canal** (cc) apresenta uma interface através da qual se faz o controle de todo o canal. As operações invocadas nesta interface são propagadas aos objetos *binder* do canal. Os métodos do controlador são:

- startChannel: dá início à transmissão em todo o canal;
- pauseChannel: suspende a transmissão em todo o canal;
- continueChannel: continua a transmissão em todo o canal;
- modifyChannel: modifica a configuração de todo o canal de acordo com novos requisitos de QoS;
- getBinders: retorna uma lista de todos os objetos binder que estão participando do canal;
- getChannel: retorna os parâmetros utilizados na criação do canal para um *binder* específico;
- getStatus: retorna medidas de monitoramento do canal para um *binder* específico;
- getReport: retorna uma lista contendo todos os *binders* do canal e suas respectivas medidas de monitoramento;
- deleteChannel: elimina todo o canal.

Os métodos dos objetos apresentados acima são disponibilizados em interfaces IDL. A ligação a estas interface se dá por intermédio de um ORB. As interfaces de ***stub*** (s) e ***protocol*** (p) não são apresentadas pois as funções desenvolvidas para estes objetos são inacessíveis aos usuários do canal. Estas funções são discutidas no capítulo 4.

A implementação permite a criação de canais ponto-multiponto. A conexão e desconexão de um consumidor a um canal em funcionamento se dá sem interferência nos demais consumidores. Os canais fluem em um único sentido. A conjugação de dois ou mais canais permite a obtenção de fluxos em todos os sentidos desejados.

O usuário do canal deve construir sua aplicação e utilizar as bibliotecas da implementação do canal para criar os servidores CORBA. Uma descrição mais detalhada do uso da implementação é feita no capítulo 5.

3.3. Modelo de objetos

A Figura 3.2 apresenta o modelo de objetos da proposta de implementação segundo a notação OMT (*Object Modeling Technique*) [Rumbaugh91].

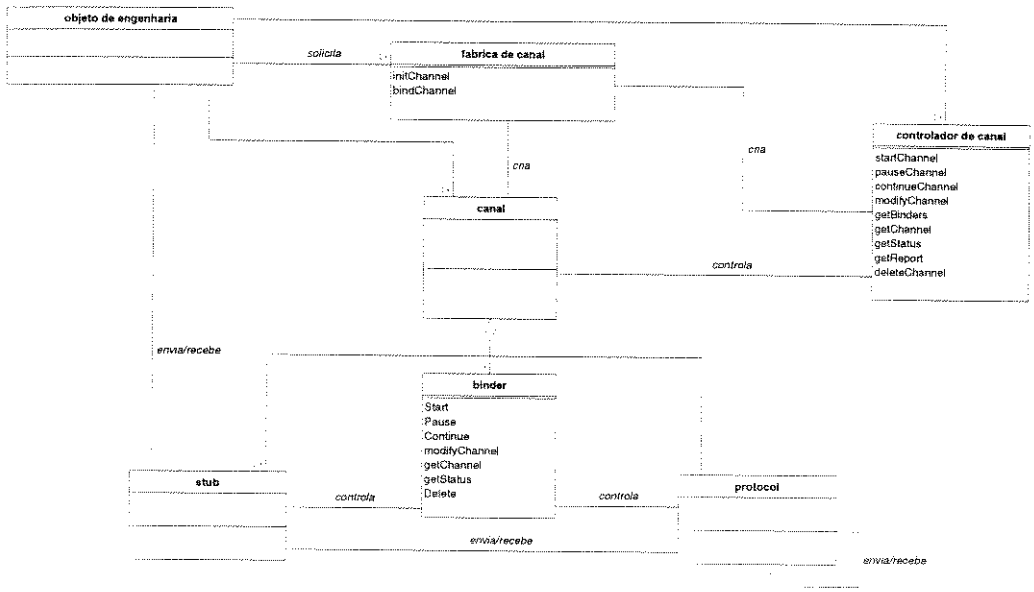

Figura 3.2: Modelo de objetos da proposta de implementação de canal ODP.

3.4. Trabalhos Correlatos

Nesta seção são apresentados alguns trabalhos importantes na área de sistemas distribuídos. Estes trabalhos estão relacionados ao corrente não por tratarem de um estudo da especificação ODP mas por abordarem de diferentes maneiras a questão da transmissão de fluxo contínuo de dados utilizando a plataforma CORBA.

3.4.1. OMG

O OMG editou em agosto de 1996 uma RFP (*Request for Proposal*), documento solicitando propostas tecnológicas, para obter respostas que permitissem às aplicações no contexto CORBA estabelecer e gerenciar fluxos contínuo de áudio e vídeo entre objetos [OMG96]. O processo de adoção da especificação para fluxos atualmente se encontra no estágio de avaliação da proposta final revisada [OMG97] submetida em conjunto pela Iona, Lucent e Siemens-Nixdorf, a partir de propostas individuais de cada uma delas.

A proposta estabelece um conjunto de interfaces que implementam um *framework* para fluxos de mídia distribuída. Os principais componentes do *framework* são *virtual multimedia devices*, *multimedia device*, *streams*, *stream endpoints*, *flows*, *flow endpoints* e *flow devices*.

Um *stream* representa a transferência de um fluxo contínuo de mídia, usualmente entre dois ou mais *virtual multimedia devices*. Um *stream* pode conter múltiplos *flows*. Cada *flow* carrega dados em uma direção, portanto um *flow endpoint* (ponto final de *flow*) pode ser tanto produtor quanto consumidor de fluxo. Uma operação em um *stream* pode ser aplicada a todos os *flows* do *stream* simultaneamente ou a somente um subconjunto deles.

Um *stream endpoint* (ponto final de *stream*) termina um *stream*. Um *stream endpoint* pode conter múltiplos *flow endpoints*. Ambos *flow endpoint* produtor e consumidor podem estar contidos no mesmo *stream endpoint*. Pode haver um objeto CORBA representando cada *flow endpoint* e *flow*, mas nem todos os sistemas necessitam apresentar interfaces IDL para estes objetos.

Um *multimedia device* abstrai um ou mais itens do hardware multimídia e atua como uma fábrica de *virtual multimedia devices*. Um *multimedia device* pode suportar mais de um *stream* simultaneamente. Para cada *stream* requisitado o *multimedia device* cria um *stream endpoint* e *multimedia device*.

A proposta trata ainda das questões de uso de múltiplos protocolos, qualidade de serviço e sincronização de fluxo.

O próximo passo no processo até adoção da especificação pela OMG é a análise da proposta submetida e votação pelos vários comitês responsáveis.

3.4.2. Universidade de Cambridge e Laboratório de Pesquisa da Olivetti

Neste trabalho [Murphy96], desenvolvido em conjunto pela Universidade de Cambridge e pelo Laboratório de Pesquisa da Olivetti, também em Cambridge, Murphy e Mapp propõem um modelo computacional para integração de fluxos multimídia em sistemas distribuídos. O modelo permite a ligação entre interfaces de fluxo representada por um objeto que encapsula as regras de configuração e os atributos de qualidade de serviço.

No modelo os *devices* são os objetos que tratam com fluxos multimídia podendo ser *sources*, *sinks* ou *modules*. Um *source* é um produtor de mídia e normalmente é uma abstração do dipositivo real de *hardware*, como um microfone ou uma câmera. Um *sink* é um consumidor de mídia e também é uma abstração do dispositivo físico de *hardware*, um alto-falante por exemplo. Um *module* é um processador de mídia que recebe o fluxo contínuo, processa-o de alguma forma e gera o fluxo modificado. Os *devices* interagem via interfaces de comunicação denominadas *ports*.

Os objetos *stream binding* tem a função de estabelecer e controlar um fluxo contínuo que pode ser ponto-ponto ou multiponto. Quando nenhuma manipulação do fluxo é executada entre produtor e consumidor a ligação é classificada como passiva. A ligação ativa compreende duas ou mais ligações passivas separadas por um ou mais *modules*.

Quando existe a necessidade de coordenação de fluxos paralelos, por requisitos sincronização por exemplo, os vários estilos de ligações ativas e composição das ligações são denominadas *smart streams*.

Foi desenvolvido um protótipo de parte do modelo. Contudo, não são apresentados os resultados ou a avaliação do modelo proposto.

3.4.3. Universidade de Lancaster

Coulson e Waddington [Coulson96], membros do Grupo de Pesquisa em Multimídia Distribuída da Universidade de Lancaster, propuseram uma extensão do modelo computacional CORBA pela integração de tipos de dados de mídia contínua como tipos básicos.

Nesta proposta, o modelo computacional CORBA é estendido pela introdução de uma série de conceitos: interface de eventos, eventos, objetos *binding*, especificação de qualidade de serviço e múltiplas interfaces por objeto.

O conceito de interface de evento encapsula interações de eventos. Os eventos são interações unidirecionais que podem transportar dados. Para definição de interfaces de eventos é especificada a linguagem *Event Definition Language* (EDL). A ligação de duas ou mais interfaces de eventos é responsabilidade do objeto *binding* que encapsula os detalhes da infra-estrutura de comunicação. Os objetos *binding* dão suporte a um estilo de ligação na qual a aplicação que inicia o processo não precisa estar participando da ligação.

É definida também uma notação para especificação de qualidade de serviço baseada na lógica de tempo-real QL. Uma expressão QL é constituída por sinais (por exemplo, chegada ou saída de dados em uma interface), marcas de tempo dos sinais e uma relação lógica entre eles. Neste modelo, ao contrário da especificação atual CORBA, os objetos devem suportar múltiplas interfaces.

O ambiente de suporte a objetos deve incluir o Event Object Adaptor (EOL) que é um novo adaptador de objetos específico para as implementações das interfaces EDL e o compilador EDL para mapear EDL em outra linguagem.

São apresentados exemplos de códigos EDL derivados de IDL (*Interface Definition Language*) e um cenário de conexão para transmissão de fluxo de vídeo.

3.4.4. Projeto MAESTRO

O projeto MAESTRO [Yun97] corresponde a um sistema de multimídia distribuída desenvolvido na Universidade de Ciência e Tecnologia de Pohang, Coréia, que utiliza plataforma CORBA para o fornecimento de serviços de multimídia distribuída.

A arquitetura do MAESTRO é dividida em quatro camadas. Na primeira camada, a mais alta, existem várias aplicações multimídia (vídeo conferência, telemedicina, etc.) que utilizam os serviços da segunda camada.

A segunda camada provê uma API de serviços multimídia distribuídos que é usada pelas aplicações. Esta camada consiste de definições de classes que podem ser usadas para transferir de dados ou para abstrair dispositivos.

A terceira camada é formada por objetos CORBA que implementam os serviços necessários no suporte a aplicações multimídia. Estes objetos são: Naming Service Object, Session Service Object, Storage and Retrieval Service Object e Mangement Service Object.

Na quarta camada existem vários sistemas operacionais e tecnologias de rede que conectam fisicamente as partes envolvidas na aplicação.

MAESTRO define os objetos VA_Producer, VA_Consumer, VA_Port e VA_Conector para a transmissão de fluxo de vídeo e áudio em um mesmo domínio ou entre domínios diferentes. Os objetos VA_Producer e VA_Consumer possuem uma *thread* própria de execução e apresentam as operações de iniciar, encerrar, parar e continuar. Eles são utilizados na produção e consumo de dados de vídeo e áudio que fluem através de VA_Port e VA_Conector. Cada objeto VA_Producer ou VA_Consumer está associado a um VA_Port e cada domínio possui um VA_Conector. Existem ainda os objetos CLA_Port e CLA_Conector que se diferenciam de VA_Port e VA_Conector por poderem enviar e receber dados simultaneamente.

3.4.5. Arquitetura de Conexão de Dispositivos Multimídia em Ambientes Distribuídos

O trabalho de [Desideá97] tem como fonte de inspiração o componente MSS (*Multimedia System Services*) integrado pela ISO à padronização PREMO (*Presentation Environment for Multimedia Objects*) [PREMO96]. O objetivo do MSS é prover uma infra-estrutura para construção de plataformas computacionais que suportem aplicações multimídia interativas, envolvendo mídias temporais, sincronizadas em um ambiente distribuído heterogêneo.

A arquitetura MSS estabelece uma série de objetos visíveis aos clientes: fábrica, localizador de fábrica, manipulador de eventos, grupo, dispositivo virtual e conexão virtual. A fábrica é responsável por instanciar os diversos objetos do MSS. O localizador de fábrica permite encontrar a referência de uma fábrica capaz de instanciar objetos cujas propriedades satisfazem uma lista de restrições. O manipulador de eventos trata os eventos recebidos dos outros objetos. O grupo permite a manipulação de múltiplos recursos de uma maneira unificada. O dispositivo virtual abstrai os dispositivos de hardware e de software. A conexão virtual

provê operações para criar uma conexão entre a porta de saída de um dispositivo virtual e a porta de entrada de outro, encapsulando questões de baixo nível a respeito de transporte.

Foi implementado o objeto conexão virtual utilizando a plataforma CORBA como infra-estrutura aos objetos distribuídos. Os dados multimídia, considerados uma seqüência de caracteres e definidos como tipo sequence em IDL, são transmitidos via ORB. Para avaliação da implementação da conexão virtual foram criados dispositivos virtuais simples. Os resultados obtidos indicam que a transmissão do fluxo por fora do ORB poderia trazer ganho de eficiência e um melhor aproveitamento dos recursos oferecidos pelo sistema.

Capítulo 4

Implementação

Este capítulo discute detalhadamente a implementação do canal ODP descrevendo as restrições de projeto, baseadas na especificação e implementação, e justificando as decisões tomadas no decorrer do desenvolvimento, a partir da proposta apresentada anteriormente.

Conforme exposto no Capítulo 3, a implementação do canal utiliza CORBA para o controle do canal e um sistema de transporte alternativo para a transmissão de fluxo. A plataforma aderente a especificação CORBA escolhida foi Orbix 2.1 sobre o sistema operacional SunOS 5.5.1. A escolha se deu em função de sua disponibilidade e da familiaridade com a versão anterior do produto. Todas as interfaces foram implementadas na linguagem C++ [Lippman91].

Inicialmente são apresentados alguns aspectos importantes da plataforma Orbix. Em seguida são discutidas algumas decisões tomadas na implementação. Por fim são apresentadas as interfaces e as respectivas classes C++ dos objetos do canal.

4.1. Orbix

Orbix [Orbix95a] [Orbix95b], um produto da Iona Technologies Inc., é um ambiente para desenvolvimento e integração de sistemas de computação distribuída que implementa e estende a especificação CORBA 2.0.

Um sistema de computação distribuído é definido como um conjunto de componentes de software cooperativos executando em um número de computadores conectados por uma rede. Estes componentes podem ser padronizados e compartilhados por várias aplicações ou específicos às aplicações. O caso mais simples consiste de um processo que interage com alguns componentes compartilhados. Os usuários finais interagem com as aplicações que podem usar qualquer um dos componentes disponíveis na rede, sendo que cada um dos componentes também tem a possibilidade de usar os outros componentes para realização de sua tarefa.

Em Orbix, os componentes de distribuição são os objetos que possuem uma interface bem definida e que podem ser disponibilizados em qualquer nó. Cada objeto está associado a um servidor que gerencia um conjunto de objetos de mesma ou de diferentes interfaces. Cada objeto Orbix tem internamente um identificador único utilizado para localizá-lo no sistema distribuído. Cada servidor pode ter qualquer número de clientes que se comunicam com seus objetos.

Em um programa convencional, não distribuído, o mecanismo usual para conectar diferentes componentes é a chamada de função ou procedimento. Dado que este mecanismo é bem entendido, é razoável esperar que esta facilidade de programação seja usada para conectar componentes em um programa distribuído. Como Orbix é um sistema orientado a objeto, os objetos, ao invés de chamadas isoladas de procedimentos, é que são os elementos da distribuição. Um programa C++ executando sobre Orbix pode invocar funções de um objetos em qualquer nó do sistema. Isto é alcançado com um alto grau de transparência pois o programador pode fazer uma chamada de função C++ normal para invocar um método em um objeto remoto.

A invocação de métodos remotos também pode ser realizada através de RPC (Remote Procedure Call). Os mecanismos de invocação ORB e RPC são bem similares mas com algumas diferenças bem importantes. Através de RPC é chamada uma função específica, todas as funções de mesmo nome são implementadas da mesma maneira. Por outro lado, através do ORB é chamado um método de um objeto específico. Com o polimorfismo classes de objetos distintas podem responder à mesma invocação diferentemente. A chamada atinge um objeto específico que controla seus próprios dados e implementa a função a sua maneira.

4.1.1. Interfaces de objetos

Em um programa distribuído é possível implementar diferentes objetos em diferentes linguagens de programação. Para permitir interações entre estes objetos é comum abstrair as funcionalidades de cada um deles definindo sua interface. A linguagem IDL é uma linguagem padronizada pela OMG para definição de interfaces.

Tendo definido a interface de um objeto em IDL o programador está, em princípio, livre para implementá-lo utilizando qualquer linguagem de programação adequada. Analogamente, o programador que deseja usar o objeto pode empregar qualquer linguagem de programação para fazer requisições ao objeto.

4.1.2. Clientes e servidores

Um sistema de software distribuído consiste de objetos executando em clientes e servidores. Os servidores fornecem objetos para o uso por clientes e outros servidores. Quando um objeto requisita uma operação em outro, o primeiro é denominado cliente e o segundo servidor. O servidor, por sua vez, pode requisitar uma operação e neste caso está agindo como um cliente enquanto durar a operação.

Em Orbix se um servidor estiver desativado[6] ele será ativado quando um de seus objetos for usado. Todos os servidores devem ser registrados no repositório de implementação para este fim.

Um cliente pode conter objetos e qualquer um deles pode ser conhecido por outros clientes e servidores e consequentemente utilizados por eles. Ao contrário dos servidores os clientes não são ativados automaticamente e portanto não precisam ser registrados no Orbix.

4.1.3. Classe IDL C++

Os componentes de um programa Orbix são os objetos. As interfaces de objetos são especificadas usando-se IDL. Uma interface IDL consiste da especificação de operações e atributos.

O compilador IDL, como mostra a Figura 4.1, é usado para produzir a classe C++ correspondente a cada interface IDL, ou seja, o compilador traduz declarações IDL para C++. Esta etapa é necessária antes que a interface possa ser implementada ou usada pelo código C++. Na terminologia Orbix esta classe C++ é denominada classe IDL C++. Uma classe IDL C++ lista as funções que o cliente da interface pode usar.

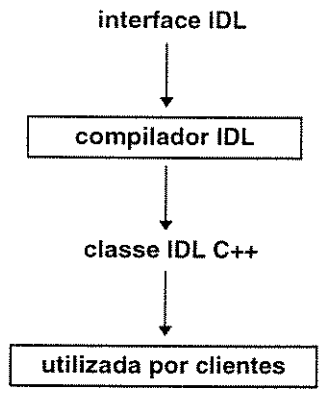

Figura 4.1: O ambiente de programação para clientes.

O implementador deve inserir na classe C++ a definição de cada método, isto é, o código que deve ser executado quando a operação ou atributo correspondente é chamado pelo cliente.

Uma vez que a interface esteja implementada, qualquer número de objetos pode ser criado. Cada objeto corresponde a uma interface IDL e pode ser utilizado a partir de qualquer nó do sistema distribuído.

Cada operação de uma interface IDL é mapeada em um função C++ de mesmo nome na classe IDL C++. Um atributo é mapeado em duas funções C++: uma para ler seu valor e outra para escrevê-lo.

4.1.4. O lado do cliente

Uma referência de objeto em Orbix pode denotar um objeto que é local (no mesmo espaço de endereçamento de memória) ou remoto (em diferente espaço de endereçamento em outro ou no mesmo nó). Para assegurar a integração a C++ as referências de objeto Orbix são

[6] Não executando em memória

ponteiros C++ normais. Quando o objeto é remoto o ponteiro se refere a um objeto *proxy* do objeto remoto. Uma *proxy* é uma representação local do objeto remoto.

Um cliente pode se ligar a um objeto Orbix especificando seu identificador único. A requisição para ligação retorna uma referência de objeto que o cliente pode usar para invocar funções na sintaxe C++. Se o objeto é local ou remoto é transparente ao programador.

Se o objeto é remoto a referência denota a *proxy* que implementa o suporte necessário para enviar a requisição ao objeto. Todas as requisições feitas à *proxy* são direcionadas automaticamente ao objeto remoto, como mostra a Figura 4.2.

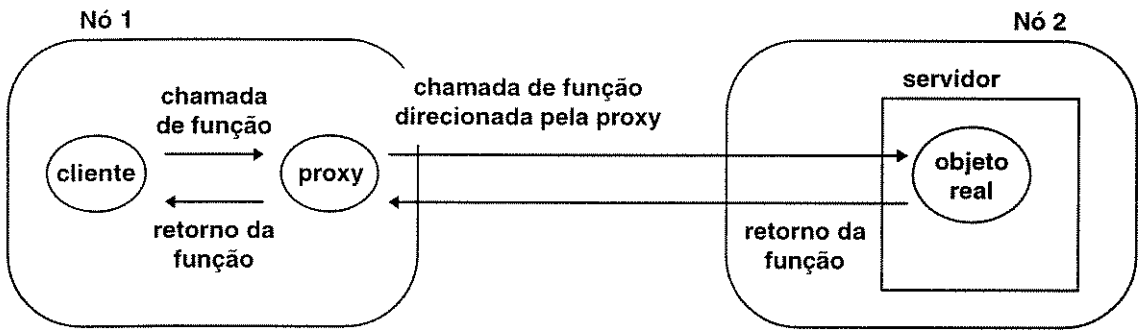

Figura 4.2: Uma chamada de operação em uma *proxy*.

4.1.5. Implementação da interface

Cada interface IDL deve ser implementada por uma classe C++. Esta classe deve implementar cada uma das funções que correspondem às operações e atributos IDL. As instâncias desta classe são objetos Orbix.

O compilador IDL pode ser instruído para produzir uma versão preliminar da classe de implementação. O programador adiciona, então, outros membros (construtores, destrutores, variáveis e funções públicas e privadas) e codifica cada uma das funções.

Uma interface IDL pode ter mais de uma classe de implementação, permitindo, por exemplo, a execução em tipos diferentes de máquinas.

Orbix suporta dois mecanismos (BOA e TIE) para relacionar a classe de implementação à sua interface IDL. O programador do cliente não precisa saber qual dos dois mecanismos o programador do servidor decidiu usar.

Na abordagem BOA, Orbix gera uma classe C++ para cada interface IDL, usando o nome da interface e concatenando as letras BOAImpl. Por exemplo, seria gerada a classe testeBOAImpl para a interface teste. Usando esta abordagem, a classe de implementação deve herdar da classe BOAImpl.

Na abordagem TIE, a classe de implementação não herda da classe BOAImpl. Neste caso o programador indica ao Orbix qual classe implementa a interface através de uma macro C++ denominada DEF_TIE. A escolha do mecanismo a ser utilizado quase não interfere no funcionamento do servidor e a diferença entre eles é muito mais uma questão de gosto do programador.

4.1.6. Servidores e repositório de implementação

Os objetos em um sistema distribuído estão contidos em processos servidores. Cada servidor possui um nome único dentro do nó ao qual pertence. O mesmo nome pode ocorrer em nós diferentes. Um servidor pode consistir de um ou mais processos.

O nome de um objeto é composto por: (1) um nome (marker) que assim como o nome da interface é único em um servidor; (2) o nome do servidor; (3) o nó onde está o servidor.

Os objetos gerenciados por um servidor não precisam pertencer à mesma classe de implementação ou a mesma classe IDL C++ portanto o servidor pode suportar diferentes interfaces IDL. Um servidor pode conter objetos C++ que não tenham interface IDL, estes objetos não são invocados diretamente pelo cliente.

Orbix provê um repositório de implementação que mantém o mapeamento dos nomes dos servidores para os nomes dos arquivos executáveis que implementam aquele servidor. O programador deve registrar seu código no repositório de implementação para que ele seja ativado pelo Orbix quando for feita uma invocação a um dos objetos daquele servidor. Servidores diferentes podem usar o mesmo executável.

Orbix apresenta diferentes mecanismos ou modos de ativação de servidores dando o controle ao programador sobre como os servidores são implementados como processos pelo

sistema operacional. No modo compartilhado deve haver pelo menos um processo por servidor. O processo será iniciado pelo Orbix para executar o código do servidor se for feita uma chamada de operação a qualquer um dos objetos do servidor.

No modo não compartilhado existe um processo por objeto ativo, ou seja, cada objeto ativo é executado em seu próprio processo. Embora este modo não seja freqüentemente usado tem algumas vantagens como: não há interferência entre objetos e a invocação de operações em um conjunto de objetos de um servidor pode correr em paralelo.

No terceiro modo, modo por método, Orbix inicia um processo separado para cada chamada de operação. Ao registrar o servidor pode ser especificado um executável para cada operação.

Embora todos os servidores precisem ser registrados no repositório de implementação, os objetos só precisam de registro quando Orbix tiver que ativá-los em processos como no modo não compartilhado.

No modo compartilhado o processo servidor pode ser ativado manualmente antes de qualquer invocação aos seus objetos. As invocações subsequentes são passadas ao processo. Isto é útil por exemplo para depuração de servidores.

4.2. Decisões de implementação

A Figura 3.1 do Capítulo 3 mostra a arquitetura proposta para canal ODP. Nesta arquitetura o canal é composto por uma série de objetos que promovem a comunicação entre um produtor e um ou mais consumidores. A transmissão contínua de dados é feita por meio de um sistema de transporte alternativo enquanto o controle do canal é realizado por intermédio de uma plataforma CORBA. Conforme mencionando anteriormente, a plataforma escolhida foi Orbix 2.1 devido a sua disponibilidade para o projeto e a familiaridade com sua versão anterior.

Como visto na seção anterior, em Orbix todos os objetos estão contidos em processos servidores. A idéia inicial para implementação foi a disposição dos objetos do canal em três servidores. O primeiro conteria o objeto fábrica de canal, o segundo os objetos *stub*, *binder* e *protocol* e um terceiro incluiria o controlador de canal. Neste modelo a comunicação de dados

entre produtor/consumidor e *stub*, por estarem em processos diferentes, se daria por memória compartilhada ou troca de mensagens. Este tipo de comunicação provoca um atraso adicional na transmissão devido ao excesso de cópias de dados. Outro aspecto negativo deste modelo é a impossibilidade de controle de algumas características importantes como, por exemplo, o tamanho da fila de mensagens.

Optou-se, portanto, por um modelo no qual *stub*, *binder*, *protocol* e fábrica de canal estão no mesmo espaço de endereçamento que produtor/consumidor. Para este fim são disponibilizadas bibliotecas contendo os objetos do canal, as quais devem ser ligadas à aplicação desenvolvida pelo programador. Um exemplo de utilização das bibliotecas é descrito no Capítulo 5.

O programador da aplicação que utiliza o canal pode optar por criar servidores diferentes para produção e consumo de fluxo de informação. Por este motivo decidiu-se pela especialização do modelo, criando objetos do canal com funções associadas ao produtor e outros associados ao consumidor. A partir disto, são definidas duas bibliotecas uma para cada categoria de objetos. A construção de um servidor com dupla funcionalidade é possível utilizando-se as duas bibliotecas.

Alguns dispositivos consumidores possuem a restrição de uso por um único processo, isto é, uma vez que um processo adquira o direito de uso do recurso nenhum outro poderá usá-lo até que ele seja liberado. Um exemplo são os dispositivos de áudio. Dentro desta restrição, a implementação permite a criação de *stub* + *binder* + *protocol* dentro de um único processo em número igual ao dos fluxos que se deseja receber.

4.2.1. Definição dos objetos da implementação de canal ODP

A Figura 4.3 apresenta os objetos da implementação do canal ODP. As áreas sombreadas correspondem a processos e as setas indicam o sentido do fluxo de dados.

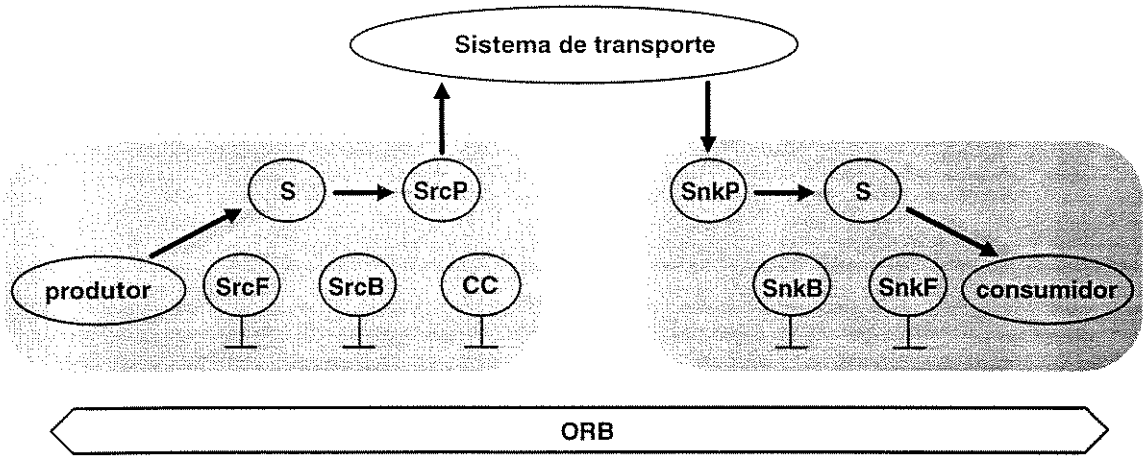

Figura 4.3: Os objetos da implementação de canal ODP.

O canal do lado do produtor de fluxo é composto pelos objetos fábrica de canal SrcChanelFactory (SrcF), *binder* SourceBinder (SrcB), *protocol* SourceProtocol (SrcP), *stub* Stub (S) e o controlador ChannelController (CC).

Para cada fluxo existe um controlador e um produtor. Como estes dois objetos estão fortemente relacionados, optou-se por instanciá-los sempre no mesmo nó. Observe que o controlador faz parte do processo do produtor. Quando é feita uma requisição ao controlador, a posse dos recursos de processamento é passada ao objeto ChannelController sem grande impacto, pois ele pertence ao mesmo processo, ao passo que se ele estivesse em um processo diferente demandaria mais tempo na troca de contexto.

O lado consumidor de fluxo é composto pela fábrica SnkChanelFactory (SnkF), *binder* SinkBinder (SnkB), *protocol* SinkProtocol (SnkP) e *stub* Stub (S). Não existe nenhuma diferença entre os stubs de produtor e de consumidor.

O programador que utiliza o canal deve instanciar um objeto do tipo SrcChanelFactory ou SnkChanelFactory em cada processo servidor, dependendo de sua finalidade. A arquitetura do canal pressupõe que um objeto fábrica está sempre disponível no servidor.

Cada objeto SourceProtocol e SinkProtocol inicia e controla uma *thread* cuja função é fazer continuamente cópias de dados entre Stub e sistema de transporte. A interface de controle dos objetos permite parar e reativar a execução da *thread*, o que significa parar e reativar a transmissão de informação pelo canal.

Um dos objetivos do projeto é permitir que usuário do canal ODP escolha o sistema de transporte adequado aos requisitos de qualidade de serviço e de acordo com a disponibilidade de infra-estrutura. No atual estágio da implementação a escolha pode ser feita entre UDP e TCP [Comer91]. Em ambos os casos utiliza-se RTP [Schulzrinne96] para que obtenção de medidas de monitoramento do canal. Com estas medidas é possível avaliar se os requisitos de qualidade de serviço estão sendo cumpridos e em caso negativo atuar sobre o canal, por exemplo, renegociando estes requisitos.

4.2.2. Modelo de objetos

Na Figura 4.4 é utilizada a notação OMT para representar o modelo de objetos da implementação. Este é um modelo especializado para funcionalidades de produtor e consumidor de fluxo.

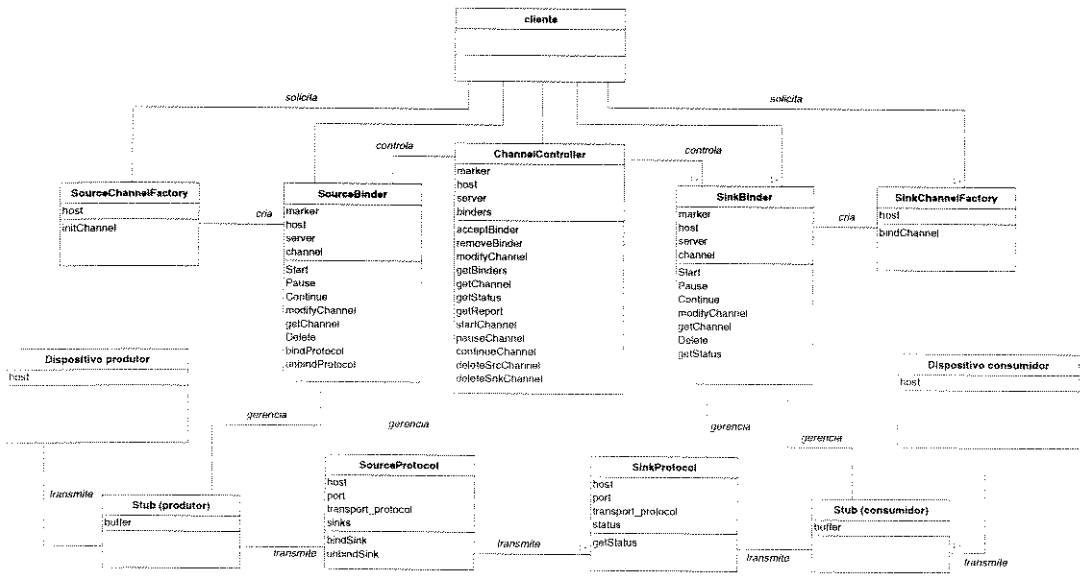

Figura 4.4: Modelo de objetos especializado da implementação de canal ODP.

4.3 Interfaces

Na arquitetura proposta é definida uma estrutura para especificar às fábricas SrcChanelFactory e SnkChanelFactory as características do canal a ser criado. Esta estrutura, denominada **channel_t,** é definida em IDL e contém:

- identificador do canal (string);
- nome (*marker*)[7] do binder associado ao produtor (string);
- nome (*marker*) do binder associado ao consumidor (string);
- tipo de protocolo de transporte a ser usado (string);
- tipo de mídia transportada pelo canal [Reynolds94] (long);
- parâmetros de qualidade de serviço (qos_t descrita a seguir);
- nó onde se localiza o produtor (string);
- nome do servidor CORBA que abriga o produtor (string);
- nó onde se localiza o consumidor (string);
- nome do servidor CORBA que abriga o consumidor (string).

O modelo de qualidade de serviço do canal é baseado no modelo *token bucket* do RSVP [Keshav97]. Os parâmetros de QoS são especificados através da estrutura IDL **qos_t** que compreende:

- tipo de controle de QoS: garantida, carga controlada ou melhor esforço (long);
- atraso máximo em microsegundos (long);
- variação máxima do atraso em microsegundos (long);
- taxa sustentada pela fonte em bytes/s (long);
- tamanho do buffer mantido pelo stub em bytes (long);
- taxa de pico da fonte em bytes/s (long).

---

[7] O *marker* é descrito na Subseção 4.1.6

A estrutura **status_t** define os valores de parâmetros de QoS medidos para o monitoramento do canal. Esta estrutura é composta por:

- banda transmitida pelo canal em bytes/s (unsigned long);
- atraso de comunicação entre objetos protocol em microsegundos (unsigned long);
- variação do atraso em microsegundos (double);
- taxa de perda de pacotes na transmissão em percentual de pacotes perdidos (unsigned long).

Os parâmetros de QoS são medidos no lado consumidor de fluxo. A banda transmitida pelo canal é a relação entre o número de bytes transmitidos no período entre duas medições e o tempo decorrido entre elas. Para o cálculo do atraso mede-se o tempo de trânsito de um pacote entre os nós do produtor e consumidor de fluxo. A variação do atraso e a taxa de perda de pacotes são computados de acordo com a especificação do RTP [Schulzrinne96]. Os valores calculados são apenas estimativas pois não existe um relógio global e o sistema operacional adotado utiliza *timesharing* para gerenciar os processos.

São definidas, ainda, duas listas utilizadas por métodos do controlador do canal. A lista **binders_t** é composta por uma seqüência de estruturas contendo nome do *binder*, nome do servidor CORBA que abriga o *binder* e o nó onde ele está localizado. A lista **report_t** contém estruturas formadas pelo nome do *binder* associado ao consumidor e os parâmetros de QoS (status_t) medidos por este binder. Esta lista corresponde a um relatório dos parâmetros de QoS medidos em todo o canal.

Nas subseções seguintes são apresentadas as interfaces IDL dos objetos fábrica de canal, *binder* e controlador de canal.

4.3.1 SrcChannelFactory

```
interface SrcChannelFactory {

  long initChannel(inout channel_t channel);

};
```

A interface IDL do objeto SrcChannelFactory inicia o canal do lado produtor, isto é, os objetos SourceBinder, SourceProtocol, Stub e ChannelControler, a partir da requisição de

qualquer objeto (**initChannel**). As características do canal solicitado são representadas pela estrutura channel_t. Os campos da estrutura que devem ser fornecidos são: identificador do canal, nome do binder, tipo de protocolo de transporte, tipo de mídia transportada e parâmetros de qualidade de serviço. O controlador de canal já é instanciado com a referência do SourceBinder recém criado.

O valor de retorno indica sucesso ou falha do método. Além disso, a fábrica adiciona à estrutura informação sobre o nó onde se localiza o produtor e o nome do servidor CORBA que abriga o produtor.

4.3.2 SnkChannelFactory

```
interface SnkChannelFactory {
  long bindChannel(inout channel_t channel);
};
```

Através da interface da fábrica SnkChannelFactory o consumidor pode se ligar a um canal já iniciado (**bindChannel**). As informações sobre o canal podem ser obtidas no *binder* do produtor ou no controlador de canal. A estas informações o consumidor acrescenta o nome que deseja para o seu *binder*. A lista de *binders* mantida pelo controlador é atualizada, assim como a lista mantida pelo SourceProtocol na qual estão relacionados todos os SinkProtocol que recebem o fluxo.

O valor de retorno indica o sucesso ou falha do método. Também são adicionados à estrutura channel o nó onde se localiza o consumidor e o nome do servidor CORBA que abriga o consumidor.

A maneira como o consumidor descobre a interface do *binder* do produtor ou do controlador de canal não faz parte dos objetivos deste trabalho. Ela pode ser feita, por exemplo, através de um Trader [Trader95].

4.3.3 SourceBinder

```
interface SourceBinder {
  channel_t getChannel();
```

```
  long Start();
  long Pause();
  long Continue();
  long modifyChannel(inout channel_t channel);
  oneway void Delete();
  long bindProtocol(in string host, in long port, in unsigned long type);
  long unbindProtocol(in string host, in long port);

};
```

O objeto SourceBinder apresenta uma interface que permite obter as características do canal (**getChannel**), iniciar (**Start**), parar (**Pause**) e reiniciar (**Continue**) a transmissão de dados, modificar a configuração do canal (**modifyChannel**) e eliminar o canal (**Delete**). Estas são características comuns aos *binders* do lado produtor e do lado consumidor. Além destas, existem mais duas funções que são utilizadas internamente no canal pelo SinkBinder para ligar os objetos SourceProtocol e SinkProtocol (**bindProtocol**) e para desligá-los (**unbindProtocol**).

Na requisição de ligação entre *protocols* é passado o parâmetro type indica o tipo de dados transmitidos pelo canal, permitindo a verificação da consistência da requisição. Os outros dois parâmetros, também presentes no método que desliga os *protocols*, representam unicamente a interface de SinkProtocol que recebe o fluxo.

A operação **modifyChannel** recebe como parâmetro uma estrutura channel_t. No atual estágio da implementação a mudança de configuração permitida, por limitações de infra-estrutura, é a alteração do tamanho do buffer.

Todas as funções têm valor de retorno que indica sucesso ou falha da operação com exceção do método que elimina o canal que não retorna valor.

4.3.4 SinkBinder

```
interface SinkBinder {

  channel_t getChannel();
  long Start();
  long Pause();
  long Continue();
  long modifyChannel(inout channel_t channel);
  oneway void Delete();
  status_t getStatus();

};
```

A interface de SinkBinder apresenta, além das funções comuns ao SourceBinder, já descritas acima, um método para obtenção das medidas de monitoramento do canal (**getStatus**).

4.3.5 ChannelController

```
interface ChannelController {

  long acceptBinder(in string binder, in string server, in string host);
  long removeBinder(in string binder);
  long modifyChannel(inout channel_t channel);
  binders_t getBinders();
  channel_t getChannel(in string binder_name);
  status_t getStatus(in string binder_name);
  report_t getReport();
  long startChannel();
  long pauseChannel();
  long continueChannel();
  oneway void deleteSnkChannel();
  oneway void deleteSrcChannel();

};
```

O ChannelController mantém uma lista com todos os *binders* que participam do canal, cujo primeiro elemento corresponde ao produtor de fluxo e os outros aos consumidores. *Binders* podem ser adicionados (**acceptBinder**) ou removidos (**removeBinder**) da lista através de métodos da interface do controlador que são usados internamente no canal.

Os outros serviços disponíveis na interface do ChannelController permitem: alterar a configuração do canal (**modifyChannel**), obter a lista dos *binders* (**getBinders**), obter as características do canal (**getChannel**) e as medidas de monitoramento (**getStatus**) de um *binder* específico, obter o relatório de QoS do canal (**getReport**), iniciar (**startChannel**), parar (**pauseChannel**) e reiniciar (**continueChannel**) a transmissão e eliminar o canal (**deleteSnkChannel** e **deleteSrcChannel**).

A eliminação do canal é feita em duas etapas: primeiro elimina-se os consumidores (**deleteSnkChannel**) e depois o produtor e controlador do canal (**deleteSrcChannel**).

Em todos os métodos das interfaces que possuem valor de retorno o sucesso é representado por um inteiro de valor 1 e o fracasso por um inteiro igual a 0.

4.4 Classes

O compilador IDL gera para cada interface uma classe IDL C++. Para cada classe IDL C++ existe uma classe de implementação C++ que implementa os serviços oferecidos pela interface e alguns outros de uso interno no canal.

O compilador IDL mapeia todas as operações das interfaces em funções C++ virtuais e acrescenta um parâmetro extra do tipo **CORBA::Environment** que é usado para transmitir exceções ao cliente. Para tornar a leitura mais agradável, nos códigos abaixo não aparecem a palavra chave **virtual** e o último parâmetro adicionado pelo compilador.

Na desenvolvimento das classes optou-se pela abordagem TIE[8] por considerá-la mais flexível.

As subseções a seguir apresentam as estruturas das classes implementadas para o desenvolvimento do canal ODP.

4.4.1 SrcChannelFactory_i

A interface SrcChannelFactory é mapeada na classe IDL C++ SrcChannelFactory e implementada pela classe C++ SrcChannelFactory_i cuja estrutura é:

```
class SrcChannelFactory_i {
private:
  SourceBinder_ptr  src_binder;
  ChannelController_ptr controller;
public:
  SrcChannelFactory_i();
  CORBA::Long initChannel (channel_t& channel);
  void destroyBinder();
};
```

A função **destroyBinder** é usada pelo SourceBinder para solicitar sua destruição. Este foi o procedimento adotado em função das características do Orbix.

[8] A abordagem TIE é descrita na Subseção 4.1.5

4.4.2 SnkChannelFactory

A interface SnkChannelFactory é implementada por SnkChannelFactory_i. A estrutura desta classe é:

```
class SnkChannelFactory_i {
private:
  SinkBinder_ptr snk_binder[MAXIMUM_NUMBER_BINDERS];
public:
  SnkChannelFactory_i();
  CORBA::Long bindChannel (channel_t& channel);
  void destroyBinder(int id);
};
```

A fábrica do lado consumidor pode instanciar vários *binders* em um mesmo processo, cada um deles participando de um fluxo diferente. O número de *binders* é limitado por **MAXIMUM_NUMBER_BINDERS**.

4.4.3 SourceProtocol_i

```
class SourceProtocol_i {
private:
  thread_t my_thread;
  int flow_port;
  int flow_sock;
public:
  int my_thread_exit;
  Stub_i* my_stub;
  RTP_source* rtp_src;
  sockaddr_in* sinks[MAXIMUM_NUMBER_SINKS];
  unsigned int number_sinks;

  SourceProtocol_i(Stub_i* stub);
  ~SourceProtocol_i();

  int getFlowSock() const;
  int bindSink(const char* host, const long port);
  int unbindSink(const char* host, const long port);
  int Start();
  int Pause() const;
  int Continue() const;
};
```

A classe SourceProtocol_i é quem envia os dados ao sistema de transporte. Para este fim é criada uma *thread*, representada pela variável **my_thread**. A variável **my_thread_exit** permite encerrar a *thread*.

No construtor da classe é passado como parâmetro um ponteiro para Stub_i do qual são lidos os dados. A classe SourceProtocol_i mantém uma lista dos *protocols* dos consumidores. Esta lista é limitada por **MAXIMUM_NUMBER_SINKS**.

A variável **RTP_src** aponta para um objeto da classe RTP_source que é utilizado para montagem dos cabeçalhos RTP.

4.4.4 SourceBinder_i

A classe SourceBinder_i implementa a interface SourceBinder.

```
class SourceBinder_i {
private:
  channel_t_var my_channel;
  SrcChannelFactory_i* my_factory;
  Stub_i* my_stub;
  SourceProtocol_i* my_protocol;
public:
  SourceBinder_i(channel_t& channel, SrcChannelFactory_i* factory);
  ~SourceBinder_i();

  channel_t* getChannel ();
  CORBA::Long Start ();
  CORBA::Long Pause ();
  CORBA::Long Continue ();
  CORBA::Long modifyChannel (channel_t& channel);
  void Delete ();
  CORBA::Long bindProtocol (const char* host, CORBA::Long port,CORBA::ULong type);
  CORBA::Long unbindProtocol (const char* host, CORBA::Long port);
};
```

Esta classe possui variáveis com ponteiros para *stub* e *protocol,* respectivamente, **my_stub** e **my_protocol**. A variável **my_channel** armazena as características do canal e a variável **my_factory** aponta para a fábrica que criou este objeto, estas informações são recebidas como parâmetros do construtor.

4.4.5 SinkProtocol_i

```
class SinkProtocol_i{
private:
  thread_t my_thread;
  int flow_port;
  int flow_sock;
public:
  int my_thread_exit;
```

```
  RTP_sink* rtp_snk;
  Stub_i* my_stub;

  SinkProtocol_i(Stub_i* stub);
  ~SinkProtocol_i();

  int getFlowPort() const;
  int getFlowSock() const;
  void getStatus(status_t& status) const;
  int Start();
  int Pause() const;
  int Continue() const;
};
```

A classe SinkProtocol_i é quem recebe os dados do sistema de transporte. Assim como no caso da classe SourceProtocol_i, é criada uma *thread*, representada por **my_thread**. Para encerrar a *thread* usa-se a variável **my_thread_exit**.

No construtor da classe é passado como parâmetro um ponteiro para Stub_i no qual são escritos os dados recebidos.

A variável **RTP_snk** aponta para um objeto da classe RTP_sink utilizada para o cálculo dos parâmetros de QoS do canal.

4.4.6 SinkBinder_i

A interface SinkBinder é implementada pela classe SinkBinder_i.

```
class SinkBinder_i {
private:
  CORBA::Long  my_id;
  channel_t_var my_channel;
  status_t_var my_status;
  SnkChannelFactory_i* my_factory;
  Stub_i* my_stub;
  SinkProtocol_i* my_protocol;
public:
  SinkBinder_i(channel_t& channel, SnkChannelFactory_i* factory, int id);
  ~SinkBinder_i();

  channel_t* getChannel ();
  CORBA::Long Start ();
  CORBA::Long Pause ();
  CORBA::Long Continue ();
  CORBA::Long modifyChannel (channel_t& channel);
  void Delete ();
  status_t getStatus ();
};
```

No construtor desta classe são passados como parâmetros as características do canal, um ponteiro para a fábrica que a criou e um identificador deste objeto na fábrica. Este identificador é necessário pois a fábrica pode criar vários SinkBinder_i em um mesmo processo. A variável **my_status** mantém os últimos valores dos parâmetros de QoS medidos no canal.

4.4.7 Stub_i

```
class Stub_i {
private:
  sema_t sp1, sp2;
public:
  char* buffer;
  int buffer_size;
  int data_length;

  Stub_i(int size);
  ~Stub_i();

  int lockProducer();
  int unlockProducer();
  int lockConsumer();
  int unlockConsumer();
};
```

A interface da classe Stub_i deve ser conhecida pelo usuário do canal pois o produtor/consumidor deve usar esta classe para escrever/ler dados.

Os dados são escritos ou lidos em **buffer** cujo tamanho é indicado pela variável **buffer_size**. A variável **data_length** corresponde a quantidade de dados que a mensagem contém.

O método **lockProducer** deve ser utilizado antes de se escrever no *buffer* e o método **lockConsumer** antes de se ler. Em ambos os casos depois de completada a operação usa-se o método **unlockProducer** ou **unlockConsumer** equivalente.

A maneira pela qual o produtor ou o consumidor obtém uma referência a um objeto desta classe é explicada no Capítulo 5.

4.4.8 ChannelController_i

A classe ChannelController_i implementa a interface ChannelController e sua estrutura é:

```
class ChannelController_i {
```

```
private:
  getPosition(const char* name);
public:
  binders_t_var binders_list;
  CORBA::ULong  list_size;

  ChannelController_i();
  ~ChannelController_i();

  CORBA::Long acceptBinder(const char* binder, const char* server, const char* host);
  CORBA::Long removeBinder (const char* binder);
  CORBA::Long modifyChannel (channel_t& channel);
  binders_t* getBinders ();
  channel_t* getChannel (const char* binder_name);
  status_t getStatus (const char* binder_name);
  report_t* getReport ();
  CORBA::Long startChannel ();
  CORBA::Long pauseChannel ();
  CORBA::Long continueChannel ();
  void deleteSnkChannel ();
  void deleteSrcChannel ();
};
```

A classe ChannelController_i mantém uma lista dos binder que participam do canal. O método **getPosition** é utilizado internamente na classe para obter a posição de um *binder* específico na lista.

4.5 Diagramas interação entre objetos

Nesta seção são ilustrados os diagramas de eventos de operações do canal. Algumas operações não estão representadas por terem uma dinâmica idêntica aos diagramas aqui presentes ou por estarem contidos neles.

A Figura 4.5 apresenta o diagrama de eventos envolvidos na criação do canal. São utilizadas as operações **initChannel** e **bindChannel**.

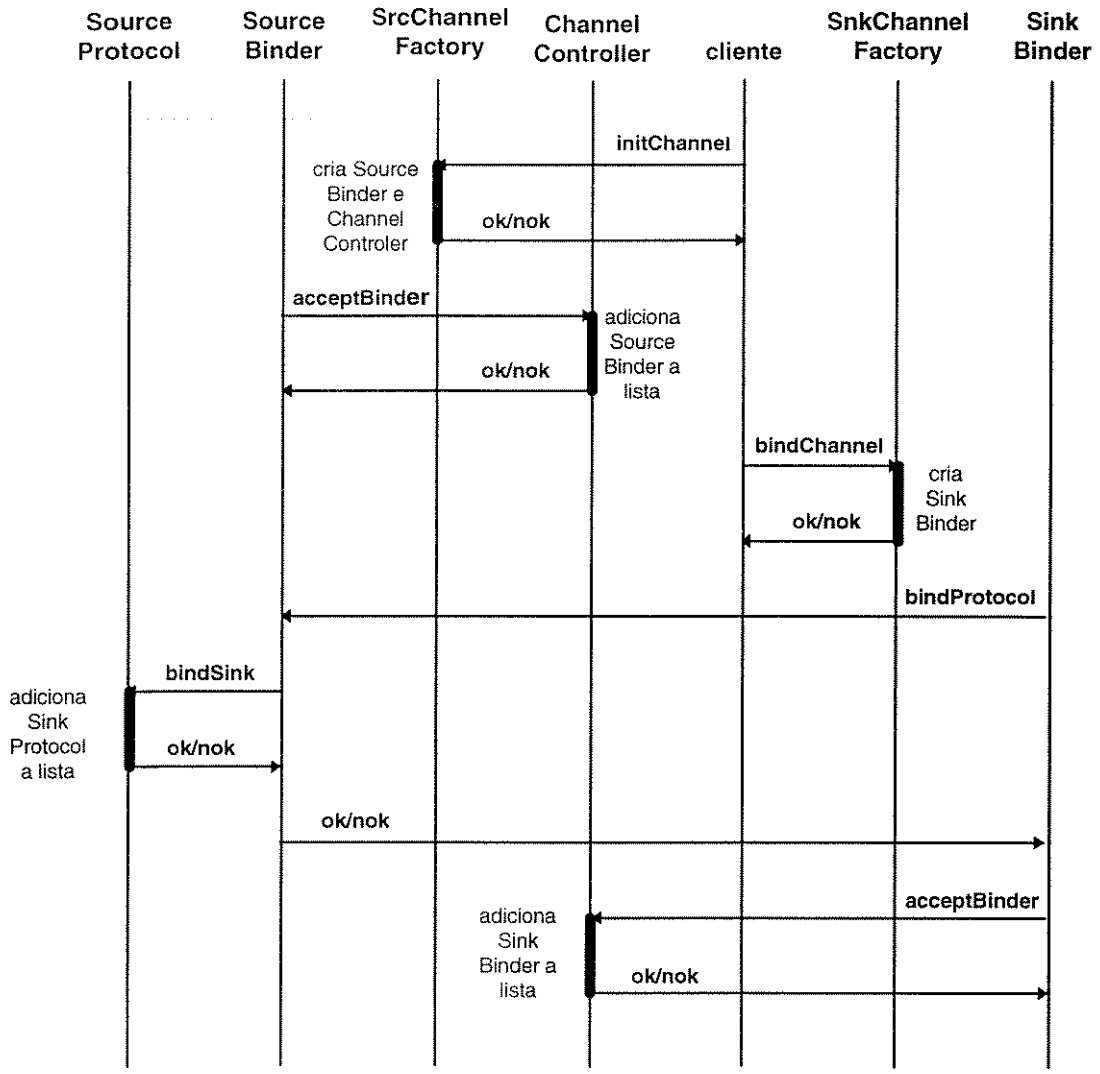

Figura 4.5: Diagrama de eventos para a criação do canal ODP.

Na Figura 4.6 estão representadas as interações entre os objetos para iniciar (**startChannel**) um canal através do controlador. As operações para parar (**pauseChannel**) e reiniciar (**continueChannel**) todo o canal possuem diagramas semelhantes a este.

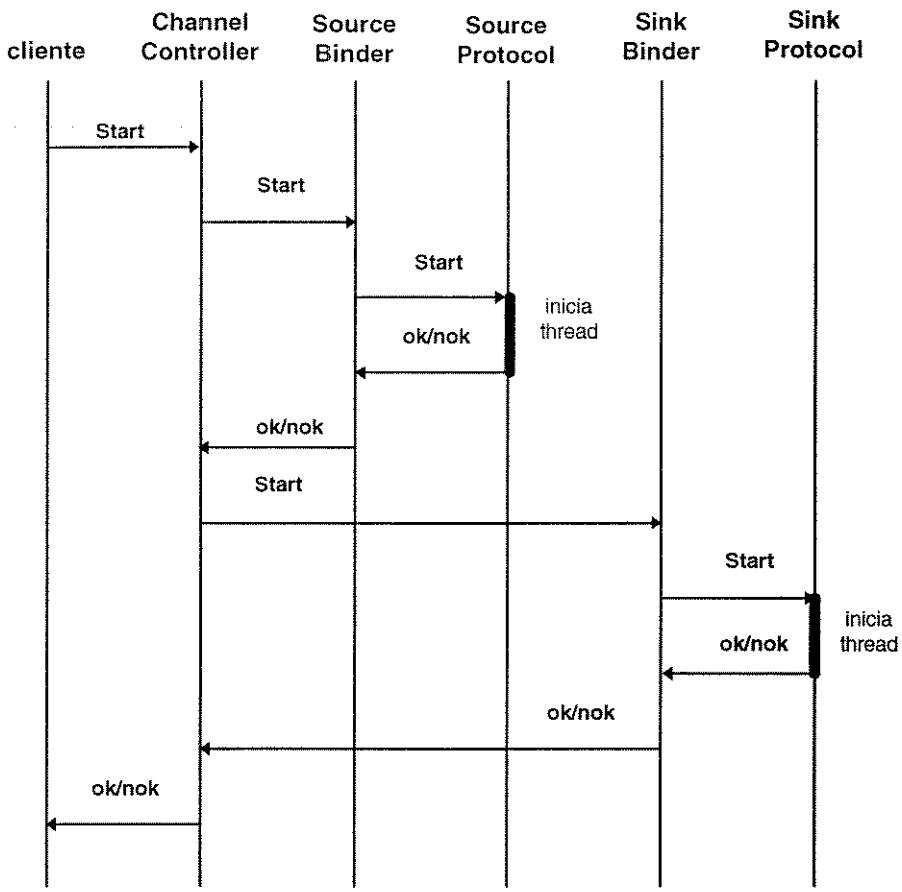

Figura 4.6: Diagrama de eventos para iniciar todo o canal.

Um exemplo das interações que ocorrem na obtenção de relatório de um canal (**getReport**) com dois consumidores é descrito na Figura 4.7. Esta operação é requisitada no controlador que para execução de suas tarefa requisita o método **getStatus** nos *binders* do canal.

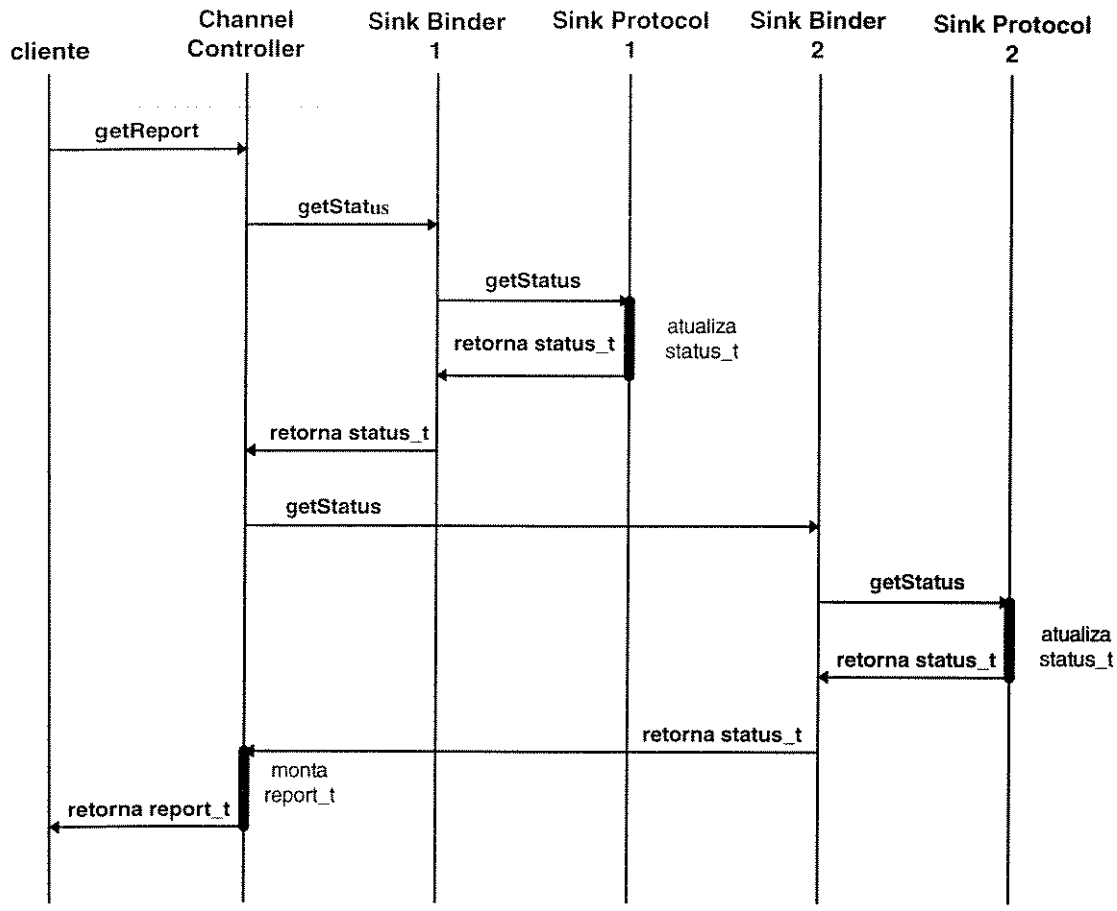

Figura 4.7: Diagrama de eventos para obtenção de relatório do canal.

A Figura 4.8 apresenta os eventos envolvidos na eliminação do canal por intermédio do controlador. A eliminação se dá em duas etapas: primeiro no lado dos consumidores e depois no lado do produtor. O controlador requisita nos *binders* a operação **Delete**.

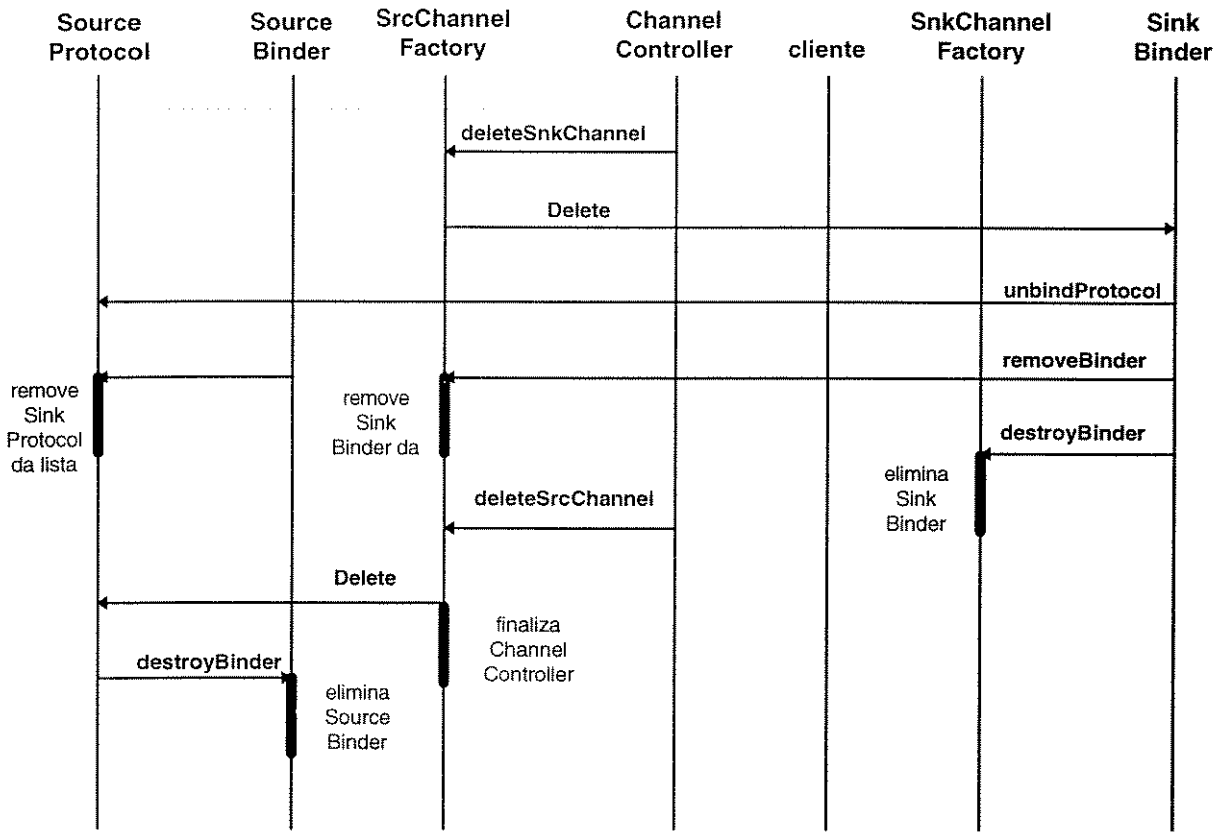

Figura 4.8: Diagrama de eventos da eliminação de todo o canal.

Capítulo 5

Aplicação

Neste capítulo é apresentado um exemplo de aplicação de áudio-conferência que utiliza a implementação de canal ODP para a transmissão de dados. A aplicação é desenvolvida em duas etapas: (1) construção dos servidores que capturam e apresentam áudio e (2) criação do programa cliente que utiliza os servidores para estabelecer e o controlar a áudio-conferência.

A aplicação foi desenvolvida em Orbix 2.1 sobre Solaris 2.5.1.

5.1. Servidores de áudio

No desenvolvimento da aplicação de áudio-conferência decidiu-se pela implementação de dois servidores de áudio: AudioProducer e AudioConsumer, respectivamente, responsáveis pela captura e apresentação das informações de áudio. A Figura 5.1 apresenta a composição dos servidores. O "T invertido" presente em alguns objetos indica a interface IDL.

O servidor AudioProducer contém o objeto Microphone (Mic) que é uma abstração do dispositivo de *hardware* microfone. Ele faz a leitura das informações de áudio do *buffer* associado ao dispositivo e as escreve no Stub. Como o servidor utiliza o canal ODP para a transmissão de dados inclui, ainda, os objetos do lado produtor SrcChanelFactory (SrcF),

SourceBinder (SrcB), SourceProtocol (SrcP), Stub (S) e ChannelController (CC). Na ativação do servidor AudioProducer são instanciados Microphone e SrcChanelFactory.

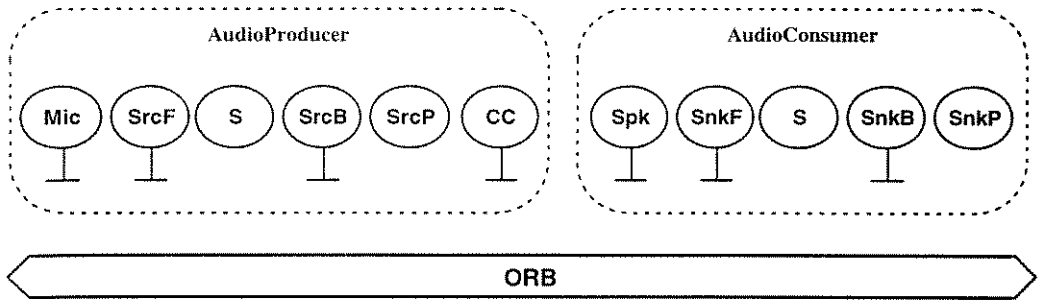

Figura 5.1: Os servidores AudioProducer e AudioConsumer.

O servidor AudioConsumer possui o objeto Speaker (Spk) que abstrai o dispositivo de *hardware* alto-falante. O Speaker é responsável por ler informações de áudio do Stub e escrevê-las no *buffer* de saída associado ao alto-falante. Além deste objeto, o servidor contém: SnkChanelFactory (SnkF), SinkBinder (SnkB), SinkProtocol (SnkP) e Stub (S). Na ativação de AudioConsumer são instanciados Speaker e SnkChanelFactory.

Os objetos Microphone e Speaker possuem interfaces para o controle sobre a captura e apresentação de áudio. As interfaces IDL e as classes C++ que as implementam são apresentadas na próxima subseção. A programação dos dispositivos de áudio é realizada através de uma API não padronizada do sistema operacional.

Para que Microphone e Speaker possam atuar sobre os seus respectivos Stub foi criada a variável global **TheStub** contendo uma referência de objeto Stub. Esta variável deve ser declarada pelo servidor que deseja usar o canal ODP. **TheStub** é atualizada na criação do canal e cada vez que se reinicia o canal. Como o servidor AudioConsumer pode receber fluxo de áudio de mais de uma fonte, a cada reinício é necessária a atualização para garantir que a variável está apontando para o *stub* do canal ativo. Esta situação será explorada na seção 5.3.

Os dois servidores de áudio quando são instanciados ficam ativos indefinidamente. A destruição do canal faz com o que os servidores voltem ao estado anterior à criação do canal.

5.1.1. Interfaces

Para o controle de Microphone e Speaker foi definida a estrutura IDL audioPrinfo_t contendo os parâmetros dos dispositivos de áudio:

- freqüência de amostragem (amostras/s);
- número de bits usados para representar cada amostra (bits/amostra);
- nível de volume (0-255);
- tamanho do *buffer* associado ao dispositivo (bytes);
- método de codificação de dados;
- porta de entrada ou saída selecionada.

Todos os parâmetros da estrutura acima têm o tipo IDL unsigned short.

5.1.1.1. Microphone

```
interface Microphone {

  audioPrinfo_t getAudInfo();
  long setAudInfo( in audioPrinfo_t audioPrinfo );
  long Start();
  long Continue();
  long Pause();

};
```

A interface Microphone apresenta funções para obter (**getAudInfo**) e definir (**setAudInfo**) os valores dos parâmetros de áudio, iniciar (**Start**), parar (**Pause**) e reiniciar (**Continue**) a captura de áudio.

5.1.1.2. Speaker

```
interface Speaker {

  audioPrinfo_t getAudInfo();
  long setAudInfo( in audioPrinfo_t audioPrinfo );
  long Start();
  long Continue();
  long Pause();

};
```

A interface Speaker possui funções idênticas às presentes em Microphone.

5.1.2 Classes IDL C++

A fim de tornar a leitura mais clara, nos códigos abaixo foram omitidos, em cada método C++ mapeado das interfaces, a palavra chave **virtual** e o último parâmetro do tipo **CORBA::Environment**.

5.1.2.1 Microphone_i

A classe Microphone_i implementa a interface Microphone.

```
class Microphone_i {
private:
  audio_info_t audInfo;
  thread_t my_thread;
public:
  int fdAu;
  Microphone_i();
  ~Microphone_i();
  audioPrinfo_t getAudInfo ()
  CORBA::Long setAudInfo (const audioPrinfo_t& audioPrinfo);
  CORBA::Long Start();
  CORBA::Long Continue ();
  CORBA::Long Pause ();
};
```

A variável **fdAu** é o descritor do arquivo associado ao microfone. A variável **audInfo** do tipo audio_info_t é uma estrutura C++ que contém as informações de áudio. A estrutura IDL audioPrinfo_t, usada nas interfaces, é um subconjunto de audio_info_t pois alguns dos parâmetros não são relevantes a esta aplicação. Microphone_i cria uma *thread*, representada por **my_thread**, para transferir dados do *buffer* do dispositivo para o *buffer* do Stub.

5.1.2.1 Speaker_i

A classe Microphone_i implementa a interface Microphone.

```
class Speaker_i {
private:
  audio_info_t audInfo;
  thread_t my_thread;
public:
  int fdAu;
  Speaker_i();
```

```
  ~Speaker_i();
  audioPrinfo_t getAudInfo ();
  CORBA::Long setAudInfo (const audioPrinfo_t& audioPrinfo);
  CORBA::Long Start ();
  CORBA::Long Continue ();
  CORBA::Long Pause ();
};
```

A variável **fdAu** é o descritor do arquivo associado ao alto-falante. A variável **audInfo** corresponde à estrutura das informações de áudio. Speaker_i possui uma *thread*, representada por **my_thread**, que transfere dados do *buffer* do Stub para o *buffer* do dispositivo.

Depois de finalizada a codificação, cada servidor é compilado e ligado à biblioteca do canal correspondente a sua finalidade gerando os programas executáveis. Por fim estes programas são registrados no repositório de implementação do Orbix. O registro dos servidores foi feito no modo *default*, isto é, modo compartilhado[9] por múltiplos clientes.

5.2. Cliente

Para estabelecer e controlar a áudio-conferência foi implementado um programa cliente. O programa requisita os servidores de áudio em um número variável de nós facilitando a configuração de diferentes cenários de utilização.

Inicialmente é descrita uma função Orbix usada na aplicação para obter a referência de objetos. Depois é apresentada a construção do programa cliente e a maneira como são utilizadas as funções das interfaces dos servidores.

5.2.1. A função _bind()

Um programa cliente que deseja invocar as operações de um objeto remoto deve primeiramente obter uma referência do objeto. A função Orbix **_bind()**, definida em cada classe IDL C++, é um dos mecanismos fornecidos para obtenção da referência ao objeto. A operação recebe como parâmetro a informação que identifica um objeto Orbix e retorna uma referência ao objeto, invocações subsequentes serão automaticamente passadas ao objeto.

---

[9] Os modos de ativação são descritos na Subseção 4.1.6

Na chamada do método **_bind()** podem ser fornecido o nome do nó do objeto alvo, seu servidor e seu *maker*[10]. Os clientes podem ainda chamar o **_bind()** usando um subconjunto destas informações. O nome do nó geralmente é omitido permitindo que o código possa ser escrito sem conhecimento prévio da local onde reside o servidor no sistema distribuído. Para achar o servidor Orbix utiliza o serviço de localização.

A função **_bind()** também pode ser chamada especificando-se somente o nome do servidor, sem determinar o *marker*. Neste caso Orbix escolherá, dentro do servidor especificado, qualquer objeto que implementa a interface solicitada.

5.2.2 Construção do cliente

O programa cliente recebe como parâmetros as características do canal a ser criado, entre elas estão um número variável de nomes de nós onde devem ser ativados os servidores de áudio. Com base nestas informações são montadas as estruturas do tipo channel_t para as requisições de criação de canal.

O fragmento de código abaixo exemplifica o uso da função **_bind** para obtenção das referências das fábrica e a criação do canal através da referência de objeto.

```
fabrica1 = SrcChannelFactory::_bind(":AudioProducer", canal1.src_host);
fabrica1->initChannel(canal1);
fabrica2 = SnkChannelFactory::_bind(":AudioConsumer", canal2.snk_host);
fabrica2->bindChannel(canal2);
```

Depois de criado o canal, o cliente obtém as referências de todos os objetos do tipo controlador, Microphone e Speaker instanciados. O cliente atua sobre o canal exclusivamente através do controlador, a interface dos *binders* não é usada. A última etapa para estabelecer a áudio-conferência é a iniciação dos objetos ativos[11] da transmissão, conforme pode ser visto no código a seguir.

```
control->startChannel();
mic->Start();
spk->Start();
```

[10] Uma explicação mais detalhada é encontrada na Subseção 4.1.6
[11] Objetos que instanciam e controlam threads

A interface do cliente permite a gerência sobre o canal. Pode-se obter informações de áudio ou de monitorar parâmetros de QoS. Também é possível alterar algumas características dos dispositivos de áudio e do canal ou, ainda, parar e reiniciar o canal.

```
lista = control->getBinders();
relatorio = control->getReport();
control->pauseChannel();
```

O código acima apresenta a maneira de obter a lista de *binders* que participam do canal (**getBinders**) e o relatório de parâmetros de QoS (**getReport**) e, também, como parar o transmissão de fluxo no canal (**pauseChannel**). Ao final da áudio-conferência o canal é eliminado.

O programa cliente depois de compilado é ligado a uma biblioteca específica do canal. O desenvolvimento de um cliente para servidores que utilizam o canal requer o conhecimento das interfaces dos *binders*, do controlador e dos objetos criados pelo programador dos servidores.

5.3. Cenários de áudio-conferência

Os servidores de áudio e o programa cliente permitem a montagem de diferentes cenários de áudio-conferência. A seguir são apresentados algumas configurações dos servidores que serviram para avaliar a implementação de canal ODP.

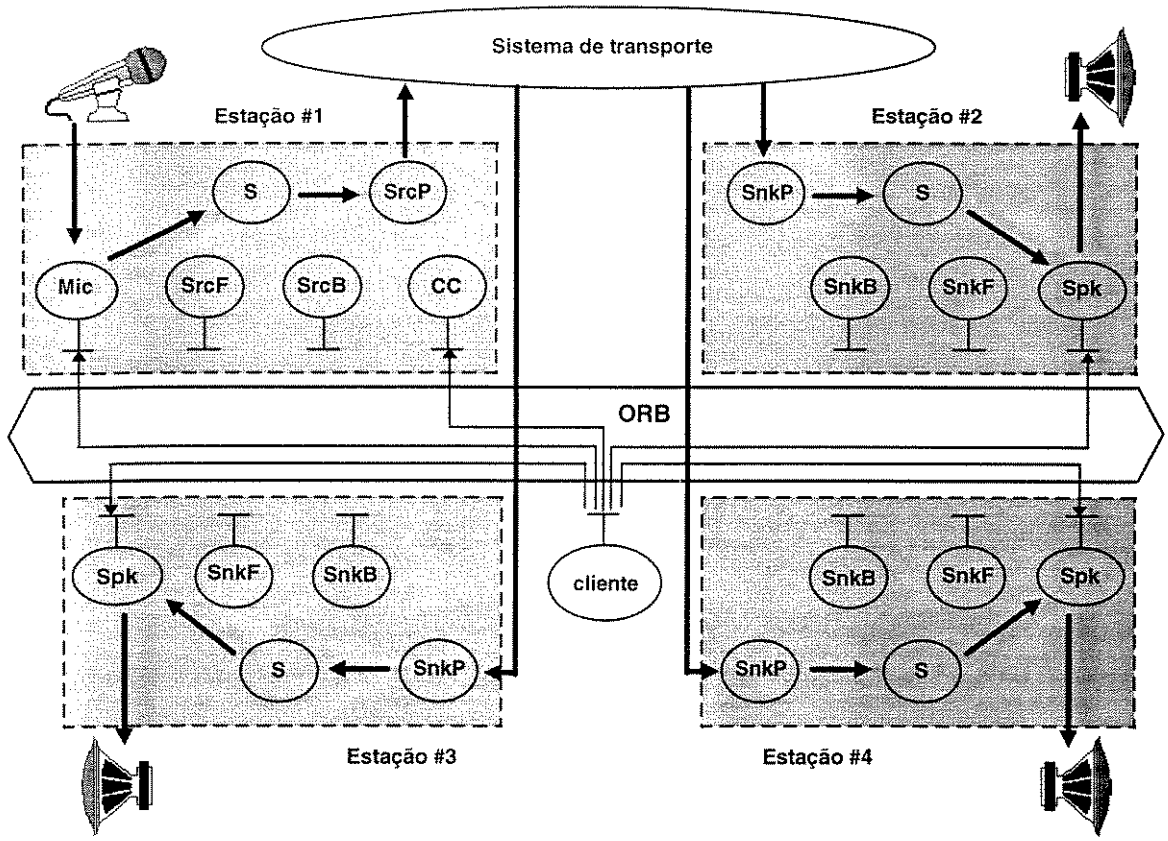

Figura 5.2: Cenário 1 de uma áudio-conferência.

A Figura 5.2 apresenta uma montagem em que três ouvintes recebem áudio de um produtor. Cada participante está utilizando uma estação diferente. O programa cliente faz requisições aos métodos das interfaces ChannelController, Microphone e Speaker através do ORB.

O cenário da Figura 5.3 representa uma situação de conversação entre duas pessoas. Em cada estação são ativados os dois servidores de áudio (AudioProducer e AudioConsumer). A representação gráfica de dois sistemas de transporte serve apenas para reforçar a independência entre fluxos de dados. Nesta situação os dois canais pode estar funcionando simultaneamente.

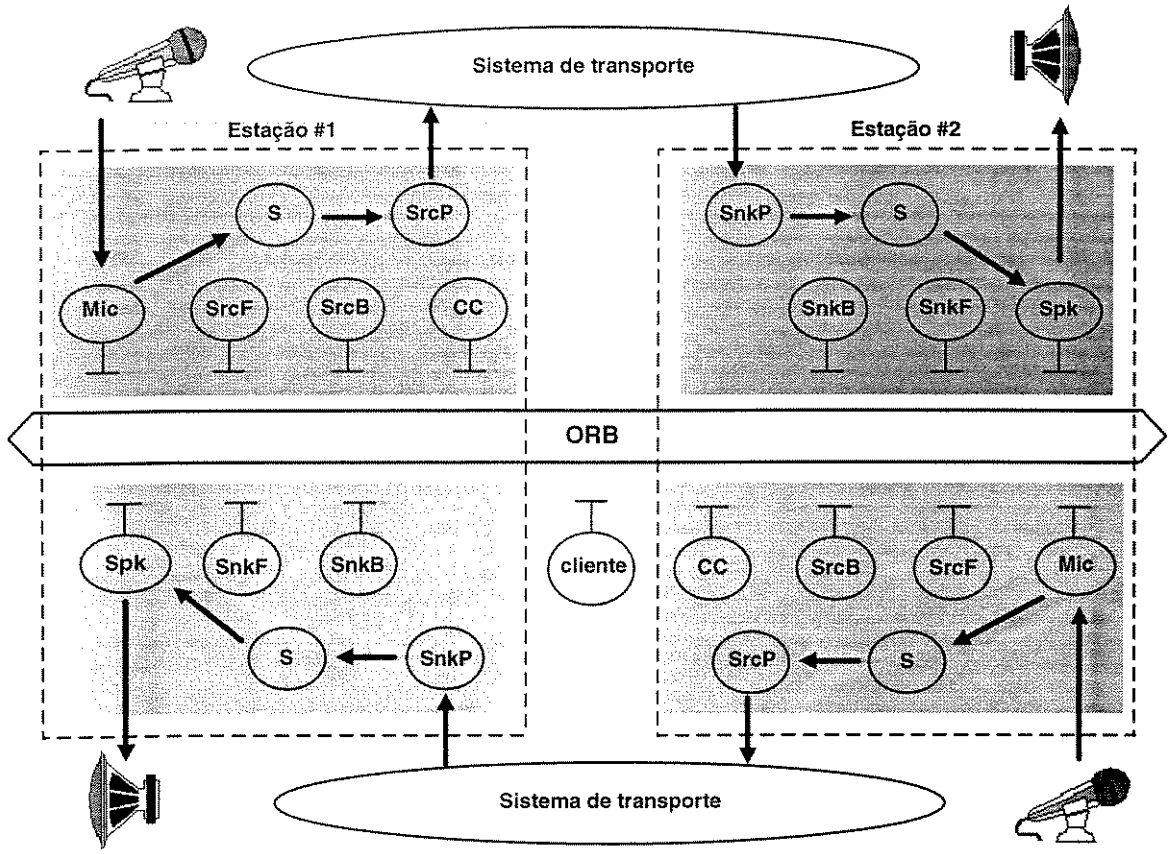

Figura 5.3: Cenário 2 de uma áudio-conferência.

A configuração da Figura 5.4 representa o caso em que um consumidor de fluxo de áudio pode escolher entre 3 fontes. Para cada fluxo é criado, no mesmo servidor, um conjunto *stub*, *binder* e *protocol*.

O cliente interagindo com os controlares, através dos métodos **pauseChannel** e **continueChannel**, determina o canal que começará a transmitir dados. A variável **TheStub** também é atualizada passando a apontar para o *stub* do canal ativado. Como o Speaker faz a leitura do *buffer* do *stub* indicado por **TheStub** garante-se que os dados corretos estão chegando ao destino (Speaker).

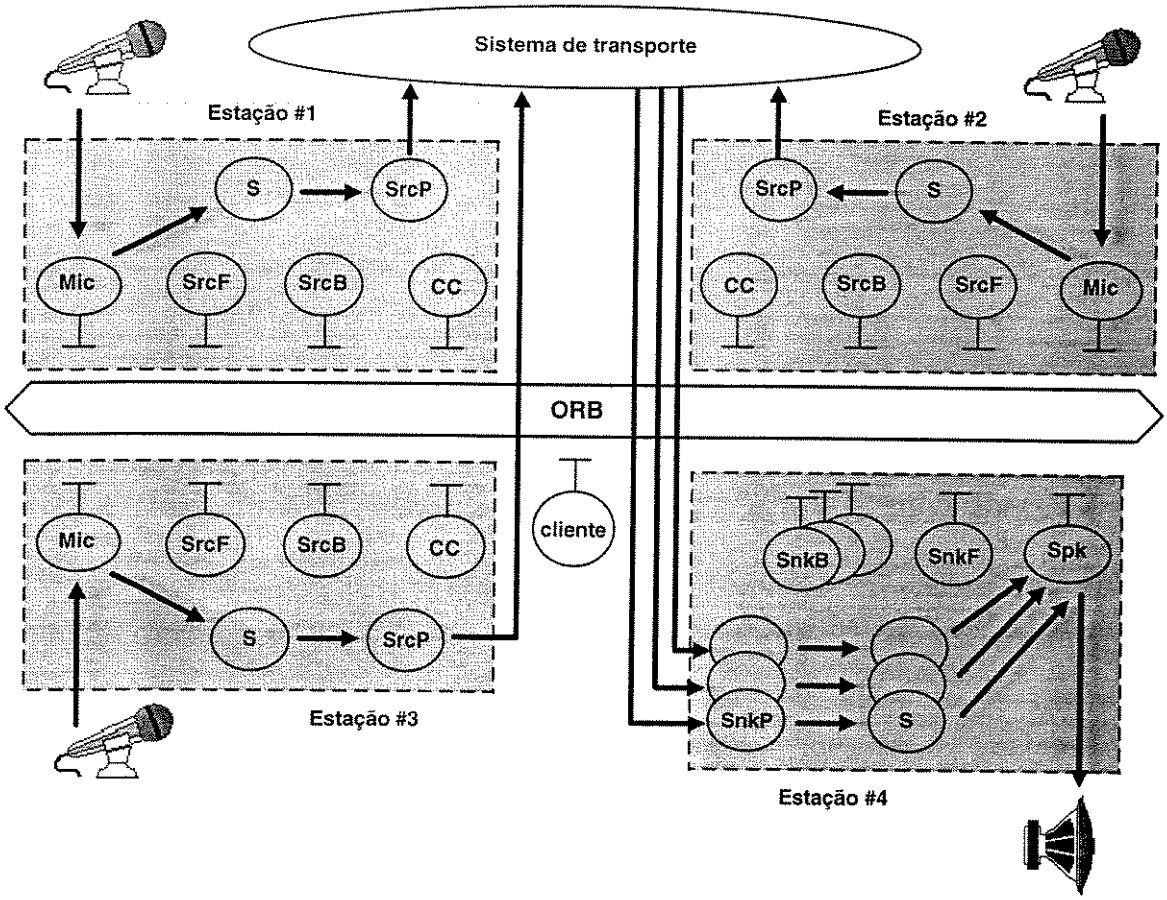

Figura 5.4: Cenário 3 de uma áudio-conferência.

Os cenários aqui apresentados não englobam todas as configurações possíveis, servem apenas para ilustrar a utilização da aplicação desenvolvida. Além destas, várias outras montagens podem ser conseguidas.

5.4 Dados experimentais

Para validar o uso do canal na transmissão de fluxo foram conduzidos alguns testes com os cenários apresentados anteriormente. Os testes foram realizados em estações SUN Sparc 5 com 64 MB de RAM ligadas por uma rede Ethernet de 10 Mbps. A rede conta com aproximadamente 60 máquinas. No horário da realização dos testes a carga da rede era média uma vez

que a maioria das máquinas estava ocupada mas as aplicações executadas não sobrecarregavam a rede em demasia.

As estações de teste rodavam o sistema operacional Solaris 2.5.1. No desenvolvimento do *software* envolvido no teste foram utilizados Orbix versão 2.1 e o compilador C++ nativo versão 4.1. Foram geradas aproximadamente 2300 linhas de código na implementação do canal, 450 linhas para os servidores de áudio e 300 linhas para o programa cliente.

A Tabela 5.1 apresenta algumas medidas de monitoramento de QoS obtidas com a configuração da Figura 5.3. Neste cenário são estabelecidos dois canais (canal A e canal B) entre os servidores de áudio. São realizadas medidas com diferentes valores de freqüência de amostragem e o número de bytes usados para representar cada amostra (precisão). Os valores dos parâmetros de áudio escolhidos correspondem aos utilizados nos telefones (8000 amostras/s e 8 bits) e CD (44100 amostras/s e 16 bits).

Tabela 5.1: Medidas de QoS para diferentes parâmetros de áudio.

Freqüência (Hz)	**Precisão (bits)**	**Canal**	**Banda (bytes/s)**	**Atraso (ms)**	**Jitter (ms)**	**PER (%)**
8000	8	canal A	7891	6,5	0,03	0
		canal B	7386	7,8	0,04	0
44100	16	canal A	39266	8,7	0,06	24
		canal B	41282	9,1	0,05	17,6

O tamanho dos *buffers* associados aos dispositivos de áudio e dos *buffers* dos *stubs* é calculado para um período de amostragem igual a 20 ms. Para uma freqüência de 8000 Hz e precisão de 8 bits o tamanho dos *buffers* é 160 bytes enquanto que para freqüência de 44100 Hz e 16 bits o valor calculado é 1764 bytes. O período de 20 ms é um valor muito comum para as aplicações de áudio.

A qualidade de áudio observada na realização dos testes foi semelhante à verificada em ferramentas que se destinam a gravar e reproduzir áudio. O atraso fim-a-fim da transmissão, percebido durante os testes, era consideravelmente inferior a um segundo e sua variação qualitativa correspondia com os valores medidos. O aumento da taxa de perdas de pacotes

com o aumento da freqüência e precisão é explicado pela maior quantidade de informação gerada e, portanto, maior taxa de utilização da rede. Nesta situação a possibilidade de perda de pacotes é maior conforme observado na prática.

Capítulo 6

Conclusão

Face à crescente demanda por sistemas distribuídos foi padronizada a especificação ODP. Com o objetivo de fazer um estudo desta especificação foi desenvolvida uma implementação de canal ODP. Neste capítulo é apresentada uma avaliação do trabalho realizado e as suas contribuições. São definidas, ainda, algumas propostas para trabalhos futuros.

6.1. Avaliação

A leitura do Modelo de Referência ODP mostrou-se, por vezes, um tanto penosa, por se tratar de um documento de padronização que faz a opção pelo formalismo em detrimento da didática. A especificação para cobrir seus objetivos é bem genérica, que por um lado permite sua aplicação a diferentes domínios, mas por outro torna difícil o trabalho de implementação.

A adoção de CORBA para a gerência do canal e a transmissão de fluxo de dados "por fora", definidas na arquitetura, mostrou-se bem acertada em função dos resultados obtidos e por ser o procedimento adotado na maioria dos trabalhos publicados com objetivos semelhantes. A opção de excluir o *binder* da transmissão efetiva dos dados, outra decisão de arquitetura, também é encontrada no modelo TINA [TINA95], uma arquitetura do domínio de telecomunicações que aplica os conceitos e funções ODP.

A adoção da plataforma Orbix atendeu às expectativas com relação a transparência na comunicação cliente-servidor e a facilidade de acesso às interfaces IDL. O desenvolvimento em Orbix, depois de adquirida a familiaridade com a ferramenta, se dá sem grandes sobressaltos. Em virtude da adoção do paradigma da orientação a objeto, por parte da arquitetura CORBA, a implementação apresenta uma grande facilidade de modificação de sua estrutura.

As *threads* utilizadas na implementação apresentaram um comportamento instável, demonstrando que sua programação e funcionamento não estão completamente dominados. É necessário que se construa mais programas para entender esta instabilidade. A API de programação dos dispositivos de áudio possui algumas características que não condizem com aquelas definidas na sua documentação. Além disto, a documentação é pouco detalhada.

A construção da aplicação demonstrou a facilidade de uso da implementação de canal ODP. A programação dos dispositivos demandou mais trabalho do que as modificações em função do canal. A programação do cliente mostrou-se de grande simplicidade, sendo necessário apenas o conhecimento da ferramenta Orbix e as interfaces dos servidores de áudio.

A transmissão de dados pelo canal para diferentes parâmetros de áudio (taxa de amostragem, precisão e volume) apresentou um funcionamento similar ao de aplicações de áudio disponíveis para o sistema operacional. A gerência do canal é completa e não introduz atrasos.

Os resultados alcançados atenderam os objetivos inicialmente estabelecidos para o trabalho. A aplicação desenvolvida utilizando a implementação indica que as decisões de projeto foram acertadas.

6.2. Contribuição

Este trabalho contribuiu para a continuar e aprofundar o estudo da especificação ODP. Foram identificados alguns requisitos para a implementação dos serviços de comunicação da especificação.

A implementação demandou, também, o conhecimento da plataforma Orbix e todos os aspectos envolvidos na programação (*threads*, dispositivos de áudio, etc.).

A própria implementação se constitui em uma contribuição por fornecer uma infra-estrutura para outros trabalhos. Já foram desenvolvidos uma implementação para serviços multimídia [Prado97] e um módulo da arquitetura TINA [Araújo97].

6.3. Trabalhos Futuros

Uma sugestão na continuação deste trabalho seria o desenvolvimento de uma aplicação de vídeo que utilize a implementação do canal ODP. Esta aplicação serviria para avaliar o impacto no funcionamento do canal causado pelo aumento do fluxo de dados.

Um outro tema a ser abordado seria a utilização da versão Orbix *multi-threaded* para atendimento das requisições das interfaces de gerência do canal.

Com a adoção, por parte da OMG, de uma especificação para controle e gerência de fluxo de áudio e vídeo [OMG96], poderiam ser feitas alterações na arquitetura proposta aproveitando-se as funcionalidades criadas.

Um outro trabalho a ser desenvolvido seria modificar a implementação com a utilização de uma infra-estrutura que permita a reserva de recursos garantindo a qualidade de serviço da transmissão de dados.

Referências Bibliográficas

[Araújo96] D. E. de Araújo, "Serviços de Gerenciamento ODP Utilizando a Arquitetura CORBA", Dissertação de Mestrado, Faculdade de Engenharia Elétrica e de Computação, 1996

[Araújo97] D. E. Araújo, R. C. M. Prado, L. F. Faina e E. Cardozo, "Implementação do Módulo DPE da Arquitetura TINA-C sobre CORBA", 2° Seminário Franco-Brasileiro em Sistemas Informáticos Distribuídos, Fortaleza, CE, Novembro 1997

[Comer91] Douglas E. Comer, "Internetworking with TCP/IP, Volume I: principles, protocols and architecture", Second Edition, Prentice-Hall International Editors, 1991

[CORBA95] The Common Request Broker: Architecture and Specification, Revision 2.0, July 1995

[Coulson96] G. Coulson, D. G. Waddington, "A CORBA Compliant Real-Time Multimedia Platform for Broadband Networks", Proceeedings of he International Workshop on Trends in Distributed Systems (TreDS), Lecture Notes in Computer Science Volume 1161, Aachen, Germany, October, 1996

[Desideá97] L. Desideá, "Arquitetura de Conexão de Dispositivos Multimídia em Ambientes Distribuídos", Dissertação de Mestrado, Faculdade de Engenharia Elétrica e de Computação, 1997

[ISO97] SC33 Strategy - Joint Work with OMG, SC21/WG7 ad hoc meeting on SC33 Strategic Plan, Helsinki, 14-25 July 97, disponível em ftp://ftp.dstc.edu.au/pub/arch/RM-ODP/Helsinki_Output/7N1307.ps.gz

[Keshav97] S. Keshav, "An Engeneering Approach to Computer Networking: ATM Networks, the Internet and the Telephone Network", Addinson-Wesley Publishing Company, EUA, 1997

[Lippman91] S. B. Lippman, "C++ Primer", 2nd ed, Adisson Wesley, 1991

[Murphy96] B. J. Murphy, G. E. Mapp, "Integrating Multimedia Streams into a Distributed Computing System", Proceedings of Multimedia Computing and Networking, San Jose, CA, USA, January, 1996

[ODP95a] ISO/IEC 10746-1 RM-ODP, "ODP Reference Model Part 1, Overview", June 1995

[ODP95b] ISO/IEC 10746-2 RM-ODP, "ODP Reference Model Part 2, Foundations", June 1995

[ODP95c] ISO/IEC 10746-3 RM-ODP, "ODP Reference Model Part 3, Architecture", June 1995

[ODP95d] ISO/IEC 10746-4 RM-ODP, "ODP Reference Model Part 4, Architectural Semantics", June 1995

[OMG96] Control and Management of A/V Streams, OMG RFP, telecom/96-08-01

[OMG97] Control and Management of Audio/Video Streams, OMG RFP Submission, Revised Submission, telecom/97-05-07, IONA Technologies, Plc, Lucent Technologies, Inc, Siemens-Nixdorf, AG

[ORBIX95a] Orbix 2 Programming Guide, Iona Technologies Ltd, Release 2.0, November 1995

[ORBIX95b] Orbix 2 Reference Guide, Iona Technologies Ltd, Release 2.0, November 1995

[Prado97] R. C. M. Prado, L. A. Guedes, L. F. Faina e E. Ca rdozo, "Implementação de Serviços Multimídia em CORBA", XXIII Conferência Latino-americana de Informática (CLEI), Valparaiso, Chile, Novembro 1997

[PREMO96] ISO/IEC 14478-3; "Presentation Environment for Multimedia Objects Part 3: Multimedia System Service Component", 1996, disponível em ftp://ftp.cwi.nl/pub/premo/PremoDocument/Part3/Part3.ps.gz

[Reynolds94] J. Reynolds, J. Postel, "Assigned Numbers", STD 2, RFC 1700, USC/Information Sciences Institute, October 1994, disponível em http://ds.internic.net/rfc1700.txt

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

[Schulzrinne96] H. Schulzrinne, S. Casner, R. Frederick, V. Jacobson, "RTP: A Transport Protocol for Real-Time Aplications", STD 1, RFC 1889, January 1996, disponível em http://ds.internic.net/rfc1889.txt

[Siegel94] J. Siegel, Liaison SC Statement for WG7 ODP, disponível em ftp://ftp.omg.org/pub/docs/1994/94-09-14.ps

[TINA95] Overall Concepts and Principles of TINA, Version 1.0, February 1995, disponível em http://www.tinac.com

[Trader95] Draft Rec. X9tr, ISO/IEC DIS 13235, ODP Trading Functions, June 1995

[Yun97] T. H. Yun, J. Y. Kong, J. W. Hong, "MAESTRO: A CORBA-Based Distributed Multimedia System", Proc. of the Fourth Pacific Workshop on Distributed Multimedia Systems, Vancouver, Canada, July 1997