Universidade Federal de São Carlos

Centro de Ciências Exatas e de Tecnologia

Programa de Pós-Graduação em Ciência da Computação

"Projeto do Subsistema de Comunicação e Distribuição e da Camada de Serviços da Arquitetura OpenReality para Suporte à Criação de Aplicações de Visualização Distribuída"

BRUNO DO AMARAL DIAS BAPTISTA

São Carlos
Fevereiro/2004

Ficha catalográfica elaborada pelo DePT da Biblioteca Comunitária da UFSCar

B222ps

Baptista, Bruno do Amaral Dias.
Projeto do subsistema de comunicação e distribuição e da camada de serviços da arquitetura OpenReality para suporte à criação de aplicações de visualização distribuída / Bruno do Amaral Dias Baptista. -- São Carlos : UFSCar, 2009.
136 f.

Dissertação (Mestrado) -- Universidade Federal de São Carlos, 2004.

1. Sistemas distribuídos. 2. Framework (Programa de computador). 3. Interatividade. 4. Plataforma JAMP. I. Título.

CDD: 005.43 (20ª)

Universidade Federal de São Carlos

Centro de Ciências Exatas e de Tecnologia

Programa de Pós-Graduação em Ciência da Computação

"Projeto do Subsistema de Comunicação e Distribuição e da Camada de Serviços da Arquitetura OpenReallity para Suporte à Criação de Aplicações de Visualização Distribuída"

BRUNO DO AMARAL DIAS BAPTISTA

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de São Carlos, como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.

Membros da Banca:

Prof. Dr. Luis Carlos Trevelin
(Orientador - DC/UFSCar)

Prof. Dr. Hélio Crestana Guardia
(DC/UFSCar)

Prof. Dr. Jauvane Cavalcante de Oliveira
(LNCC/RJ)

São Carlos
Fevereiro/2004

"Damos nome de destino à força hipotética e incompreensível que, imprevisivelmente, determina todos os eventos."
Gabriel Dupont

"Deus ajuda o marinheiro na tempestade, mas o timoneiro deve estar ao leme."
Provérbio alemão

"Há indivíduos que jamais erram, pois não pretendem fazer nada certo."
Goethe

"Pois então, seja o mundo a minha ostra, que abrirei com a espada."
William Shakespeare

Agradecimentos

Gostaria de compartilhar este espaço entre os companheiros que fizeram parte do cotidiano deste trabalho, os amigos, os *hardwares* e os *softwares*.

Aos amigos, aos novos, aos antigos, aos que achavam que sabiam jogar futebol, aos que continuam achando, valeu! Para os desprovidos de categoria que só sabiam dar pontapé, meu mais profundo obrigado, pois foi sempre um prazer retribuir os pontapés.

Aos *hardwares*, e foram vários (perguntem para o Darli), vocês conseguiram, pois entre mortos e feridos, salvaram-se todos. Diversas vezes vocês esquentaram, saíram do controle, pararam de funcionar, brigamos, mas sempre, após uma longa e exaustiva conversa, conseguíamos terminar o trabalho.

Aos *softwares*, tenho um recado para cada um. Java, obrigado pela sua portabilidade. Seu slogan, "*Write once. Run anywhere.*", é verdade. Meu mais sincero obrigado por nunca me mostrar o erro *UnknownException*.

Linux, obrigado por libertar-me das "janelas" que me prendiam. Como forma de agradecê-lo, passei a usar o seguinte rodapé em minhas mensagens eletrônicas: "*Slackware, because it works. XFCE, because it rocks!*"

XMMS, obrigado por oferecer-me "silêncio" nos momentos que precisei.

C/C++, hummm..., obrigado.

Aos sempre presentes, root@{200.18.98.123,127.0.0.1}, eugeni@200.18.98.120 e erico@200.18.98.113. Agradeço pelos tão freqüentes momentos de conhecimentos.

Ao desconhecido desenvolvedor do *exploit* que tanto utilizei durante as frias e silenciosas noites no laboratório durante a realização dos testes, obrigado, Amigo!

Bellezi, idealizador deste projeto, saiba que deu trabalho, mas valeu a pena.

Para aqueles que me obrigam a sintetizar as palavras, visando restar o espaço necessário para a apresentação do trabalho, Geraldo(Papai), Honória(Mamãe), Cíntia(Mana) e Dustin* (Pintado), deixo por ora registrado que, agora, compreendo os demais autores quando fazem uso das palavras "apoio incondicional".

E finalmente, mas não menos importante, ao Programa de Pós-Graduação em Ciência da Computação, em especial ao Prof. Dr. Luis Carlos Trevelin(mestre, professor, doutor, orientador, pai, amigo), e à Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES), por fomentar este projeto, obrigado.

Resumo

Este trabalho apresenta o modelo de distribuição de informação adotado na arquitetura OpenReality (OR), bem como as questões envolvidas na integração das estruturas nela adotadas. Aplicações de Visualização Distribuída podem ser criadas a partir do uso do *framework* OR, que aceita redefinições, especializações e personalizações. Durante o desenvolvimento de uma aplicação de visualização distribuída, é possível a escolha do protocolo de comunicação a ser utilizado, bem como suas parametrizações. Os conceitos de consistência e de interatividade e seus comportamentos diante de cada modelo de distribuição de informação também são discutidos.

Entre as principais estruturas utilizadas, encontra-se a Plataforma JAMP, que sofreu significativas expansões para suportar os requisitos impostos pela arquitetura OR. O *framework* de domínio de aplicação JNDS e a ferramenta de auxílio ao desenvolvimento JAMP2C foram adicionados ao pacote que compõe a JAMP.

De maneira geral, a contribuição do projeto do Subsistema de Comunicação e Distribuição e da Camada de Serviços da arquitetura OR é a disponibilização de um novo mecanismo para a criação de aplicações de visualização distribuída. Além disso, este trabalho avança mais um passo no processo de amadurecimento da Plataforma JAMP, que passa a contar com mais um *framework* e uma ferramenta para auxílio no desenvolvimento de suas aplicações.

Abstract

This work presents the information distribution model adopted within OpenReality architecture (OR), and the related issues of its structures integration. Distributed Visualization Applications can be created using OR framework, which permits redefinitions, specializations and customizations. During distributed visualization application development, it's possible to choose the communication protocol to be used and its parameters. The consistency and interactivity concepts and their behavior for each information distribution model are also discussed.

Among the main used structures is JAMP Platform (Java Architecture for Media Processing), which was upgraded to support OR requirements. JNDS application domain framework and JAMP2C development tool were added to JAMP development package.

In a general way, the contribution of this project is to offer a new method for creating distributed visualization applications. Besides, this work improves the maturity of JAMP Platform, which now has a new framework and a new development tool available.

Lista de Figuras

Lista de Tabelas

Sumário

Lista de Abreviaturas

ANSA – *Advanced Network Systems Architecture*

ANSAware – *Middleware ANSA*

AOIM – *Area of Interest Management*

API – *Application Programming Interface*

ASA – Árvore de Sintaxe Abstrata

AST – *Abstract Syntax Tree*

AVD – Ambiente de Visualização Distribuída

CASE – *Computer Aided Software Engineering*

CAVE – *Cave Automatic Virtual Environment*

CAVERN – CAVE *Research Network*

CCITT – *Comité Consultatif International Téléphonique et Télégraphique*

CORBA – *Common Object Request Broker Architecture*

CPU – *Central Processing Unit*

DCOM – *Distributed Component Object Model*

DIS – *Distributed Interactive Simulation*

DIVE –*Distributed Interactive Virtual Environment*

DNS – *Domain Name System*

DoD – *Department of Defense*

FEC – *Forward Error Corrected*

FIFO – *First In First Out*

GC – Grafo de Cena

GPL – *General Public License*

GPU – *Graphic Processing Unit*

IA – Inteligência Artificial

IDL – *Interface Definition Language*

IEC – *International Engineering Consortium*

IEEE – *Institute of Electrical and Electronic Engineers*

IGMP – *Internet Group Management Protocol*

IP – *Internet Protocol*

IRB – *Information Request Broker*

ISO – *International Organization for Standardization*

ITU – *International Telecommunications Union*

ITU-T – *Telecommunication Standardization Sector of the ITU*

JAMP – *Java Architecture for Media Processing*

JVM – *Java Virtual Machine*

LAN – *Local Area Network*

MASSIVE – *Model, Architecture and System for Spatial Interaction in Virtual Environment*

MBONE – *Multicast Backbone*

NPSNET – *Naval Postgraduate School Networked Vehicle Simulator*

OMA – *Object Management Architecture*

OMG – *Object Management Group*

OpenGL – *Open Graphical Library*

OR – OpenReality

ORB – *Object Request Broker*

PDU – *Protocol Data Unit*

PPG-CC – Programa de Pós Graduação em Ciência da Computação

QoS – *Quality of Service*

RAID – *Redundant Array of Independent Disks*

RMI – *Remote Method Invocation*

RPC – *Remote Procedure Call*

RT – Repositório de Travas

SIMNET – *Simulator Networking*

SMP – *Symmetric MultiProcessor*

TCP – *Transmission Control Protocol*

UDP – *User Datagram Protocol*

UFSCar – Universidade Federal de São Carlos

UML – *Unified Modeling Language*

VRML – *Virtual Reality Modeling Language*

X3D – *Extensible 3D*

1. Introdução

Este é um dos trabalhos que faz parte do conjunto que compõe a proposta de desenvolvimento da arquitetura OpenReality (OR)[1]. OR é um *framework* que vem sendo desenvolvido por alunos do Programa de Mestrado PPG-CC da Universidade Federal de São Carlos para ser utilizado no desenvolvimento de Ambientes de Visualização Distribuída.

Nos últimos anos, o interesse pelo projeto de Ambientes de Visualização Distribuída vem crescendo rapidamente. Existem muitas áreas de pesquisa atualmente, tanto acadêmicas quanto comerciais, nas quais a utilização desses ambientes pode ser de grande auxílio ou mesmo imprescindível para seu desenvolvimento.

Os Ambientes de Visualização Distribuída consistem em aplicações que buscam permitir a visualização de um determinado tema por múltiplos usuários, distantes fisicamente ou não. No decorrer do texto, muitas vezes o termo Ambiente de Visualização Distribuída também será referenciado por Aplicações de Visualização Distribuída. Essas aplicações devem se preocupar com a coerência e a fidelidade do conteúdo que cada usuário recebe a todo momento. Elas podem abranger desde grandes ambientes colaborativos de projetos de Engenharia, em que vários projetistas e engenheiros se reúnem para o desenvolvimento de um trabalho, até o campo do Entretenimento com Realidade Virtual, em que vários jogadores encontram-se para disputar as mais diversas modalidades de jogos. Embora todas elas sejam aplicações promissoras, as aplicações voltadas para o Entretenimento são o exemplo mais comum de Ambientes de Visualização Distribuída que pode ser encontrado.

Os ambientes que permitem a colaboração em projetos costumam ser construídos de forma a resolver os problemas específicos de cada caso. Diferentemente do que ocorre no campo do Entretenimento, que devido à grande demanda por novos jogos, suas empresas produtoras desenvolvem uma arquitetura genérica e, para cada novo exemplar, a especializam.

Outro campo que utiliza Ambientes de Visualização Distribuída é composto pelas aplicações de simulação e treinamento militar, que, de maneira similar a alguns jogos de batalha, possuem um ambiente de guerra que é simulado. Geralmente, buscam possibilitar o

treinamento e o exercício de táticas planejadas. Nessas aplicações, além da lógica das táticas envolvidas, o realismo proporcionado aos participantes é outro requisito fundamental para atingirem seus objetivos. Técnicas de Realidade Virtual são muito utilizadas na tentativa de construção de ambientes virtuais realistas[2].

Segundo Sementille[3], os termos Ambiente Virtual e Ambiente Virtual Distribuído são recentes e ainda não possuem uma definição que os caracterize de forma única. Assim, neste trabalho, assume-se o termo Ambiente Virtual Distribuído da mesma forma como Singhal e Zyda[2] o fazem: um sistema através do qual diversos usuários interagem entre si e em tempo real, podendo estar situados em localidades geograficamente distintas.

Sendo assim, pode-se resumidamente caracterizar os Ambientes Virtuais Distribuídos como sistemas de visualização interconectados, em que algum ambiente real ou imaginário é sintetizado, através de sons e de imagens, e que permitem a interação em tempo real entre as entidades neles presentes. Eventualmente, podem também prover certo grau de imersão. É claro que Ambientes Virtuais Distribuídos possuem características próprias, além das que são estudadas neste trabalho, como interatividade, sensação de imersão e outras. Porém, por ora, o interesse é conhecer seus mecanismos e requisitos de comunicação e distribuição das informações, sempre levando em consideração a consistência destas no ambiente como um todo.

Embora o número de Ambientes Virtuais Distribuídos existentes atualmente seja grande, seus projetos e implementações tornam-se uma tarefa difícil. As ferramentas para construção e manutenção desses ambientes ainda são bastante deficitárias, além da maioria possuir código fechado[1], tender a ser voltada para tipos específicos de aplicações e, geralmente, possuir restrições significativas quanto ao tipo de ambiente e ao número de participantes que suportam. Adicionalmente, as interfaces de programação de aplicações (APIs) abertas existentes para geração e síntese de gráficos tridimensionais são de baixo nível para esses sistemas, como OpenGL[4] e DirectX[5], ou estão em um nível de abstração alto e sofrem um pouco em desempenho, como o *Visualization Toolkit*[6] (VTK).

Diante deste cenário, este trabalho apresenta as estruturas da arquitetura OpenReality que permitem a construção de Ambientes de Visualização Distribuída, visando oferecer os recursos que auxiliem o desenvolvedor a criar aplicações das mais diversas finalidades. Também são apresentadas as novidades disponibilizadas juntamente com a Plataforma JAMP, que recebeu um novo *framework* e uma nova ferramenta de desenvolvimento.

O texto encontra-se dividido em onze capítulos, além da introdução, estudo de caso e

1 Código fechado: O código fonte da aplicação não é publicamente disponível.

conclusão. A introdução contextualiza o trabalho, além de definir seus objetivos.

O capítulo 2 oferece uma visão geral dos Ambientes Virtuais Distribuídos, partindo de sua definição, passando pela sua caracterização e seus componentes e finalizando com uma listagem dos desafios inerentes ao seu projeto. Seu objetivo é apresentar as analogias que podem ser assumidas ao serem comparados às Aplicações de Visualização Distribuída.

O capítulo 3 apresenta outras arquiteturas de desenvolvimento disponíveis para a construção de Ambientes Virtuais Distribuídos. Suas características principais, formas de distribuição e aplicabilidade são apresentadas, assim como uma breve análise comparativa, que serve de base para futuras comparações com a arquitetura OpenReality.

O capítulo 4 descreve alguns dos mecanismos mais comuns para comunicação e manutenção da consistência das informações do ambiente. Algumas relações entre consistência, largura de banda, escalabilidade e interatividade também são abordadas.

O capítulo 5 oferece uma visão geral da arquitetura OpenReality, suas características e aplicabilidade. São apresentadas as camadas e suas principais funcionalidades.

Os capítulos 6, 7, 8 e 9 apresentam, respectivamente, os componentes Grafo de Cena, Gerenciador de Distribuição, Gerenciador de Comunicação e Diretório de Aplicações. Sobre o primeiro, é apresentada a estrutura presente na arquitetura OR e, em seguida, as alterações ocorridas para permitir sua distribuição. Os demais são componentes novos que foram adicionados ao *framework* que compõe a arquitetura OR.

No capítulo 10 é apresentada a Arquitetura Java para Processamento de Mídia (JAMP) e uma breve formalização sobre como ela foi concebida e utilizada neste trabalho.

No décimo primeiro capítulo, é apresentado o *framework JAMP Networked Directory System* (JDNS), resultante de requisitos impostos pela OpenReality, assim como em seu subseqüente, o capítulo 12, que apresenta a ferramenta JAMP2C.

2. Ambientes Virtuais Distribuídos

Apesar da quantidade de aplicações de visualização distribuída existente atualmente ser relativamente alta, a maioria de seus exemplares é composta por projetos comerciais, que possuem pouca documentação disponível. Fazem parte desta maioria, as empresas que produzem jogos multijogadores, que utilizam modelos tridimensionais em projetos de engenharia, que oferecem treinamento sob situações simuladas, enfim, que fazem uso dos recursos que a Realidade Virtual tem a oferecer.

Ao buscar documentações e pesquisas científicas acerca do assunto, o foco volta-se, quase que em sua totalidade, para os Ambientes Virtuais Distribuídos. Grandes instituições de pesquisa investem seus potenciais nesse tema, pois os resultados podem trazer contribuições para as mais diversas áreas, como Medicina, Educação, Treinamento, Entretenimento, Engenharia e Militar, entre outras.

Embora os Ambientes Virtuais Distribuídos envolvam conceitos próprios, principalmente os relacionados à interação entre os participantes, pode-se generalizar as Aplicações de Visualização Distribuída como uma macro-arquitetura contendo, como uma de suas especializações, os Ambientes Virtuais Distribuídos.

Sendo assim, segue uma breve apresentação dos Ambientes Virtuais Distribuídos, cujos conceitos guiaram a especificação da arquitetura OpenReality.

2.1 O que são Ambientes Virtuais Distribuídos

Ambientes Virtuais Distribuídos são sistemas de *software* que utilizam as técnicas de computação gráfica, síntese de áudio e comunicação para criar um ambiente sintético, que pode basear-se em algo real ou fictício. Múltiplos usuários, também chamados de participantes, podem interagir através dele, independentemente de suas reais localizações. Geralmente sintetizam ambientes que os participantes possam explorar.

Existem, ainda, os ambientes que possuem apenas um usuário. Estes apenas recebem esta classificação no caso de utilizarem recursos distribuídos. Segundo Singhal[7], os

Ambientes Virtuais Distribuídos geralmente incorporam uma série de características importantes:

1. **Sensação de Espaço Compartilhado:** Todos os participantes devem ter a sensação de compartilhar o mesmo espaço. Mesmo que o ambiente virtual seja grande, deve haver a possibilidade de dois ou mais participantes eventualmente encontrarem-se. O ambiente deve ainda apresentar as mesmas características para os participantes que estejam num mesmo local, como temperatura, condições climáticas, ruídos, etc. Supondo que os participantes que estejam num mesmo local representem entidades com capacidades sensoriais semelhantes (ex.: dois humanos), eles devem ter uma visão muito semelhante do espaço que compartilham.

2. **Sensação de Presença Compartilhada:** No momento em que um participante ingressa no ambiente, ele passa a ser representado por um *avatar*, uma representação gráfica que normalmente está vinculada ao tipo de ambiente sintetizado e à forma de comunicação pretendida. Através dela, os participantes podem notar a presença uns dos outros.

3. **Sensação de Tempo Compartilhado:** Os participantes do ambiente devem perceber o fluir do tempo da mesma maneira. Isso requer um mecanismo de sincronização no ambiente. As ações executadas pelos participantes devem ser processadas o mais rápido possível para não comprometer a interação entre eles.

4. **Formas de Comunicação:** Mesmo contando com gráficos sintetizados, os ambientes devem possuir formas alternativas de comunicação entre seus participantes, como a digitação de textos, gesticulação, fala, etc. A capacidade de comunicação torna o ambiente mais realístico.

5. **Formas de Interação:** A interação é um elemento chave para os Ambientes Virtuais Distribuídos, pois os torna distintos de outros sistemas de comunicação, como os de teleconferência. Além da interação entre os participantes, também deve existir a interação entre esses e o próprio ambiente.

2.2 Os Componentes dos Ambientes Virtuais Distribuídos

Os Ambientes Virtuais Distribuídos são compostos por cinco componentes básicos, os mecanismos gráficos (*graphical engines*) e dispositivos de exibição, dispositivos de comunicação e controle, sistemas de processamento, rede de dados e representação das informações.

2.2.1 Mecanismos Gráficos e Dispositivos de Exibição

Os mecanismos gráficos e dispositivos de exibição são componentes essenciais para um ambiente, pois cuidam de toda a representação visual exibida ao seu conjunto de participantes. O mecanismo gráfico opera transformando as representações de informações em imagens, que são direcionadas aos dispositivos de exibição.

Os dispositivos de exibição, por sua vez, podem variar desde simples monitores e óculos de visualização estereoscópica, até capacetes com dispositivos multimídia acoplados e cavernas de imersão. A figura 2.1 mostra um desses capacetes, chamados HMD (*head mounted display*), capazes de gerar imagens ocupando todo o campo de visão do participante, reproduzir sons e, em alguns casos, informar qual o seu posicionamento no ambiente.

FIGURA 2.1: *HEAD MOUNTED DISPLAY*

A figura 2.2, por sua vez, mostra a estrutura de uma caverna de imersão. Inicialmente desenvolvidas na Universidade de Illinois em Chicago[8], a CAVE (*Cave Automatic Virtual Environment*) é um sistema de projeção de imagens em várias telas, dispostas de modo a

formarem um cubo ou parte dele. Em cada uma dessas telas, ou faces, são exibidas imagens diferentes, porém sincronizadas pelo mecanismo gráfico. Os participantes ficam posicionados no interior desse cubo.

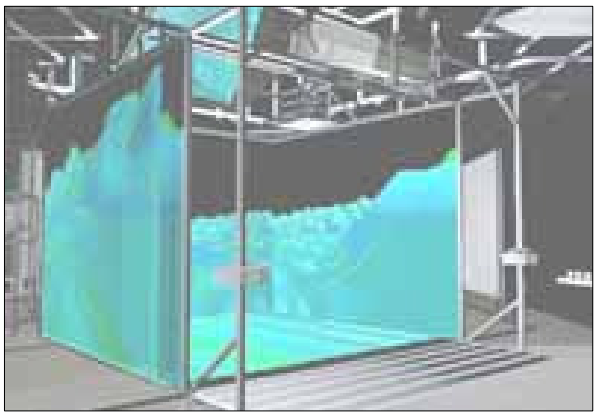

FIGURA 2.2: FOTO DE UMA CAVE

2.2.2 Dispositivos de Comunicação e Controle

Os dispositivos de comunicação e controle são os responsáveis pela tradução dos movimentos e comandos executados pelos participantes em informações que possam ser convertidas em eventos no ambiente virtual. O controle pode ser realizado, além do teclado e do mouse, por dispositivos sensores de movimento, como luvas de dados (figura 2.3), microfones e outros.

FIGURA 2.3: LUVA DE DADOS (NINTENDO *POWER GLOVE*)

2.2.3 Sistemas de Processamento

Os Ambientes Virtuais Distribuídos são grandes consumidores de processamento, pois a quantidade de tarefas a serem executadas tende a ser grande. Entre as tarefas executadas

pelo sistema de processamento, estão a análise dos dispositivos de entrada, a checagem por comandos e eventos de movimentação do participante, a conversão das modificações encontradas em informações de representação interna do ambiente, a análise da semântica das alterações, a propagação dessas alterações para os demais participantes, a reavaliação da semântica das informações após o recebimento de alterações oriundas de outros participantes e o acionamento do mecanismo gráfico, podendo inclusive auxiliá-lo em parte de seu trabalho.

Levando-se em conta todas essas tarefas e todos os requisitos para que se proporcione uma boa experiência de imersão[9], pode-se notar que o sistema de processamento tende facilmente se tornar um gargalo.

2.2.4 Rede de Dados

As redes de dados representam um componente de suma importância nos Ambientes Virtuais Distribuídos, pois é através delas que são transmitidas todas as informações referentes às modificações neles ocorridas.

Devido às limitações das redes de dados das décadas de oitenta e início de noventa, os ambientes virtuais distribuídos existentes suportavam um número limitado de participantes simultaneamente. A utilização pela Internet era inviável, devido à largura de banda disponível e à latência. Com o passar dos anos e a evolução das tecnologias de comunicação ocorridas, as redes locais e a Internet tiveram suas larguras de banda aumentadas significativamente, 1Gbps e ligações ATM formando troncos de 155Mbps, respectivamente. Ao mesmo tempo, técnicas de transmissão mais econômicas passaram a ser adotadas nos Ambientes Virtuais Distribuídos, como a comunicação de grupo (*multicasting*), por exemplo.

Atualmente, com a proliferação do acesso à Internet via provedores de banda larga, as barreiras para a massificação dos Ambientes Virtuais Distribuídos ficam cada vez mais tênues. O crescente número de jogos que simulam ambientes tridimensionais pela Internet e os que suportam uma quantidade massiva de jogadores (*Massive Multi-player Games*) confirmam isso.

2.2.5 Representação das Informações

A representação das informações é o componente afetado e utilizado por todos os anteriores. É utilizado pelo mecanismo gráfico, sofre todas as alterações através do sistema de

processamento, que utiliza as notificações provindas do sistema de comunicação e controle, e é enviado pela rede de dados quando as alterações devem ser notificadas aos demais participantes.

Geralmente, as estruturas de dados utilizadas na representação das informações dos ambientes virtuais assumem uma forma hierárquica. A forma mais utilizada para organizar as informações da cena do ambiente virtual é o Grafo de Cena.

O Grafo de Cena é uma técnica de organização da estrutura de cena que foi amplamente utilizada no conjunto de ferramentas IRIS Inventor[10], da *Silicon Graphics*, que tornou-se a ferramenta aberta chamada OpenInventor[11]. Devido à sua flexibilidade, ele foi utilizado na criação da linguagem VRML 1.0[12] e continua sendo no padrão que a substitui, o X3D[13].

Os nós e os arcos são as formas que compõem um grafo de cena. Os nós representam elementos de dados, enquanto que os arcos, as possíveis relações entre dois nós. Existem apenas dois tipos de relações, as do tipo pai-filho e as de referência.

As relações pai-filho especificam que um determinado nó é filho de outro, o nó-pai. Um nó pode ter no máximo um nó-pai e possuir qualquer número de nós-filhos. Já as relações de referência especificam a associação entre os nós do grafo de cena e nós de descrição. Os nós de descrição possuem atributos adicionais aos elementos de dados presentes nos nós comuns do grafo, como geometria, cor, espessura e outros.

A cena do ambiente virtual é construída utilizando nós e relações pai-filho, formando uma árvore (grafo acíclico e conexo). Entre os nós utilizados, estão o nó raiz, os nós locais, os nós grupo e os nós de descrição.

O nó-raiz deve ser o único que não possui um nó-pai. Ele é passado para o mecanismo gráfico, que gera as imagens através das primitivas da biblioteca de gráficos tridimensionais. Os nós locais especificam o posicionamento de sua sub-árvore de nós no ambiente virtual. Os nós-grupo possuem a função de agrupar vários nós, seus nós-filhos. Por último, os nós de descrição caracterizam-se por serem as folhas da árvore que representa o grafo de cena, pois não possuem nós-filhos. A figura 2.4 apresenta um exemplo de grafo de cena. No exemplo, pode-se observar o nó-raiz Universo posicionado no ambiente virtual através do nó Local, que possui os nós-filhos Grupo1 e Grupo2.

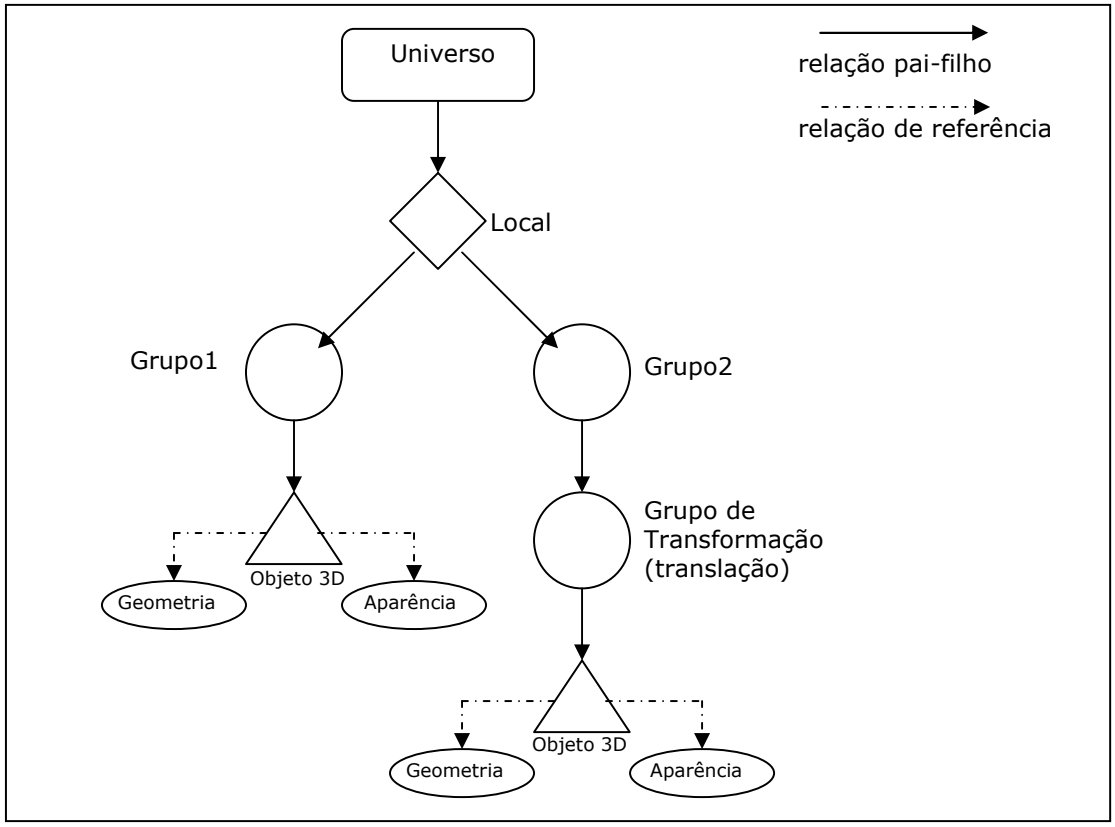

FIGURA 2.4: REPRESENTAÇÃO DO GRAFO DE CENA

2.3 Uma Taxonomia para Ambientes Virtuais Distribuídos

O projeto de arquiteturas para Ambientes Virtuais Distribuídos levanta uma série de questões que podem influir significativamente na aplicabilidade e na escalabilidade do sistema resultante. As principais questões são:

- O que deve ser distribuído?
- Por que deve ser distribuído?
- Quais são as modalidades de distribuição?

Estas questões são também o que distingue os Ambientes Virtuais Distribuídos dos simples Ambientes Virtuais, pois nestes não existem as preocupações inerentes à distribuição.

Macedonia[14] propôs, em 1997, uma série de parâmetros para classificar os Ambientes Virtuais Distribuídos levando em conta as três questões anteriores. Sua taxonomia classifica os Ambientes Virtuais Distribuídos quanto aos seguintes aspectos:

⇨ **1. Comunicação:** Vários aspectos da comunicação são considerados na busca por respostas às três perguntas. Os principais são:

a: Largura de Banda: Influencia o tamanho e a riqueza de elementos do ambiente, além de ser decisivo no número máximo de participantes que o ambiente suporta.
b: Latência: Exerce um papel determinante na sensação de interatividade do ambiente, pois quanto maior é a latência, mais os participantes perdem a noção dos eventos estarem ocorrendo em tempo-real.
c: Confiabilidade: Influencia a escolha dos protocolos de comunicação e a consistência do ambiente. Se a comunicação for sujeita a falhas, a interatividade é prejudicada, com reenvio de informações, e a consistência abalada, caso a recuperação das informações perdidas não seja possível.
d: Distribuição: Tem um grande impacto na utilização da largura de banda. As principais formas de distribuição das informações do ambiente (grafo de cena e suas alterações) são: *unicast*, *multicast* e *broadcast*.

⇨ **2. Visões:** São as janelas para o ambiente virtual na perspectiva de usuários e processos que as utilizam. As principais variações são:

a: Síncronas: Utilizadas nos ambientes que possuem mais de um dispositivo de exibição. As janelas da cabine de um simulador de vôo são um exemplo. Obviamente deve haver sincronização entre estas vistas.
b: Assíncronas: Utilizadas em ambientes genéricos. Geralmente representam a visão de um participante, associada ao seu dispositivo de exibição.

⇨ **3. Modelos de Dados:** A decisão da maneira adotada para armazenar os dados relevantes ao estado do ambiente virtual influencia diretamente em sua escalabilidade, seus requisitos de comunicação e sua confiabilidade. As variações mais comuns são:

a: Mundo Homogêneo e Replicado: Todos os participantes possuem uma cópia de toda a informação do ambiente no momento em que ingressam nele. Após o ingresso, passam a trafegar apenas as mudanças de estado. Essa variação possui uma grande escalabilidade, mas é inflexível e possui problemas de consistência com o decorrer do tempo.

b: Banco de Dados Centralizado e Compartilhado: Um servidor central é responsável pelas informações de estado do ambiente. Apenas um participante modifica uma informação num determinando instante. Essa abordagem possui um grau de consistência excelente, porém o servidor torna-se rapidamente um gargalo conforme o número de participantes aumenta. Obviamente a escalabilidade é baixa.

c: Banco de Dados Distribuído e Compartilhado: Simula uma arquitetura de memória compartilhada, dependendo de protocolos confiáveis para a manutenção do banco de dados replicado. Possui um grau de consistência excelente e escalabilidade um pouco maior que a abordagem anterior. Podem agir como clientes e servidores, dependentes de mecanismos de gerenciamento que identificam que parte do ambiente é mantida por qual participante. Nesse caso, a consistência é boa, mas existem problemas com sua escalabilidade e tolerância a falhas.

⇨ **4. Processos:** A maioria dos Ambientes Virtuais Distribuídos utiliza processos homogêneos em seus participantes, isto é, o mesmo sistema de *software* é executado por todos. Ao passo que essa homogeneidade garante respostas bastante previsíveis, o que aumenta a consistência, também tende a deixar o ambiente menos flexível. Uma solução utilizada em alguns ambientes é a transferência de objetos e comportamentos (em linguagem de roteiro) entre seus participantes.

A figura 2.5 apresenta um resumo da classificação.

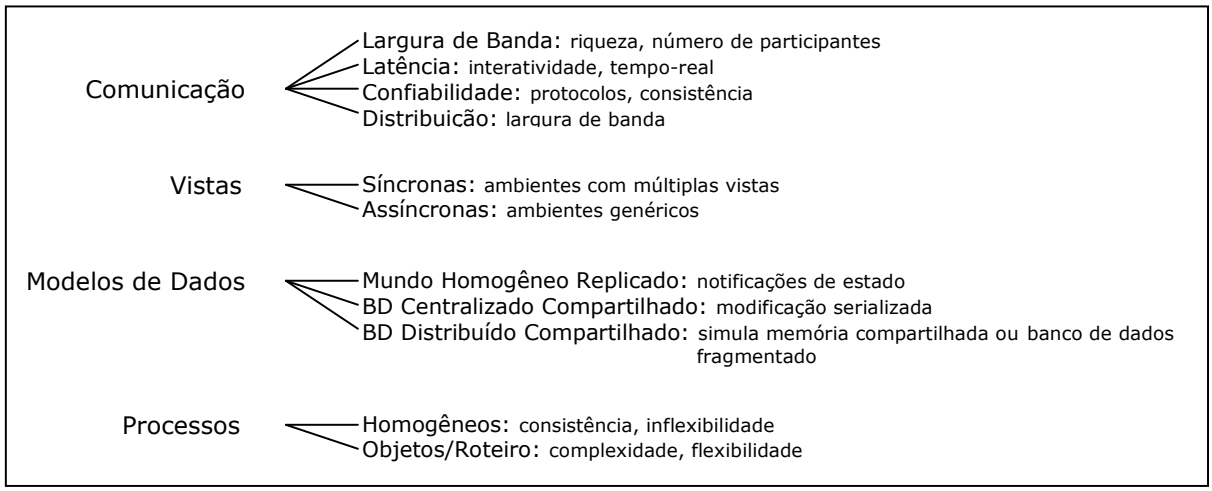

FIGURA 2.5: RESUMO DA TAXONOMIA PROPOSTA POR MACEDONIA

O próximo capítulo apresenta exemplos de arquiteturas de Ambientes Virtuais Distribuídos existentes, permitindo a verificação desta taxonomia.

2.4 Os Desafios no Projeto e Desenvolvimento de Ambientes Virtuais Distribuídos

Os Ambientes Virtuais Distribuídos são tradicionalmente sistemas de *software* de grande complexidade, pois possuem uma natureza híbrida. Ao mesmo tempo, podem agrupar as características de:

- ⇨ **Sistemas Distribuídos:** Devem lidar com os problemas como distribuição de recursos, concorrência, perda de dados e tolerância a falhas.

- ⇨ **Aplicações Gráficas:** Devem manter a qualidade gráfica e a taxa de atualizações de quadros satisfatórias, além de alocar cuidadosamente a CPU para as outras tarefas.

- ⇨ **Aplicações Interativas:** Devem responder em tempo-real aos eventos de entrada e passar a ilusão do ambiente ser um sistema local para os participantes, mesmo que eles estejam geograficamente dispersos.

Os Ambientes Virtuais Distribuídos devem ser projetados cuidadosamente, ponderando a escolha das soluções das dificuldades envolvidas com a finalidade de atingir um equilíbrio, no qual os níveis consistência e de interatividade estejam em situações aceitáveis pela aplicação alvo. A seguir, são descritas algumas dessas dificuldades, também chamadas de forças no trabalho de Singhal[2], que podem influir no desempenho do sistema.

2.4.1 Largura de Banda

Em um Ambiente Virtual Distribuído, qualquer informação deve estar disponível aos participantes, seja de alterações no estado do ambiente ou de objetos. Assim, quanto mais populoso e interativo o ambiente, maior será a utilização da largura de banda, que é um recurso limitado nas redes de comunicação. O projetista deve levar cuidadosamente em consideração a quantidade de informações que trafegará na rede para que possa alcançar os objetivos de seu projeto.

2.4.2 Heterogeneidade

Dificilmente pode-se esperar que todos os participantes de um ambiente virtual distribuído possuam exatamente os mesmos sistemas de *hardware* e recursos de rede. Em

aplicações que utilizam a Internet, por exemplo, a heterogeneidade da velocidade de conexão é quase uma regra. Os processadores e GPUs diferentes também têm um fator decisivo na capacidade de síntese gráfica da máquina do participante. É desejável que o ambiente virtual distribuído tenha a capacidade de lidar com estas diferenças, de forma a possibilitar, de acordo com os recursos disponíveis, a melhor experiência de imersão possível para os participantes. Existem ainda situações em que a heterogeneidade pode gerar disputas sobre a justiça do ambiente. Em um jogo de combate, por exemplo, um participante que possui conexão de banda larga certamente teria vantagens em relação a outrem que utiliza modem. O projetista deve levar tudo isso em questão, dosando a aplicabilidade desejada e o grau de heterogeneidade permitido.

2.4.3 Interação Distribuída

Os Ambientes Virtuais Distribuídos são aplicações interativas e que precisam processar informações praticamente em tempo-real, para oferecer a ilusão de imersão a seus participantes. Mas, como muitas das interações ocorrem entre entidades controladas externamente, uma série de informações deve ser transferida pela rede de dados para que elas possam efetivamente ocorrer. Até mesmo o simples teste de colisão entre dois participantes torna-se uma tarefa complexa, pois envolve toda a latência entre o momento gerador da informação no sistema de um participante até a sua percepção e processamento pelo sistema do outro. Além disso, o ambiente pode sofrer a ocorrência de muitos outros eventos concomitantemente. Nesses casos, operações que necessitam de confiabilidade devem empregar mecanismos de confirmação, aumentado ainda mais o tempo total da operação.

2.4.4 Requisitos de Tempo-Real

Ao contrário de sistemas convencionais, os Ambientes Virtuais Distribuídos são compostos por vários subsistemas que possuem grandes requisitos de tempo-real. Os subsistemas de entrada, por exemplo, devem ser capazes de processar quaisquer eventos oriundos de dispositivos de interação e refletir, o mais rápido possível, as modificações no ambiente. Ao mesmo tempo, informações de mudanças no estado do ambiente, que chegam pela rede de dados, também devem ser tratadas de forma imediata. O projetista deve considerar esses eventos, juntamente com as regras do ambiente a ser desenvolvido, bem como a capacidade máxima de processamento disponível pelo *hardware* a ser utilizado.

2.4.5 Gerenciamento de Falhas

De maneira análoga aos Sistemas Distribuídos, os Ambientes Virtuais Distribuídos necessitam de mecanismos gerenciadores de falhas. Desligamento de equipamentos, suas falhas ou as das redes de comunicação são fatores comuns nos sistemas computacionais e que devem ser considerados durante o projeto. O sistema deve ser capaz de lidar com estas situações e, caso seja possível, contorná-las. Mesmo que a falha seja grave e irrecuperável, o sistema deve possuir alternativas para um desligamento normal e, preferencialmente, salvando as informações relevantes da seção a ser interrompida.

2.4.6 Escalabilidade

A escalabilidade de um Ambiente Virtual Distribuído normalmente é medida através da quantidade de entidades que ele suporta. As entidades indicam objetos que são modelados por máquinas diferentes na rede, podendo ser controladas por humanos, possuir Inteligência Artificial ou, ainda, representar estados, como o clima do ambiente. O número máximo de entidades suportado depende de uma série de fatores, como capacidade de processamento das máquinas, largura de banda da rede de dados e capacidade de atendimento de servidores compartilhados. À medida que o número dessas entidades aumenta, maiores são as possibilidades de interações entre elas, gerando assim uma maior demanda para o sistema como um todo.

2.5 Considerações Finais

O capítulo apresentou os conceitos e os componentes dos Ambientes Virtuais Distribuídos, assim como as dificuldades em projetá-los. Entre os conceitos apresentados, estão as sensações de espaço, presença e tempo compartilhados, que também são observadas nas aplicações de visualização distribuída. Os componentes, por sua vez, contribuem para uma concepção melhor organizada e estruturada das Aplicações de Visualização Distribuída, pois suas apresentações esclareceram os papeis que cada um exerce no sistema. Por fim, os desafios apresentados deixam claros quais são os problemas e preocupações a serem considerados durante os seus projetos.

Portanto, o capítulo 2 trouxe, além da apresentação dos conceitos relacionados com o presente trabalho, uma metodologia para a o seu desenvolvimento.

3. Visão Geral de alguns Ambientes Virtuais Distribuídos

Durante as duas últimas décadas, surgiram vários projetos de arquiteturas para o desenvolvimento de Ambientes Virtuais Distribuídos. Esses projetos foram desenvolvidos visando diferentes aplicações e possuindo diferentes requisitos de interatividade e de número de usuários. Neste capítulo, são apresentados os principais projetos de arquiteturas para Ambientes Virtuais Distribuídos, assim como uma comparação entre as suas características.

3.1 A Arquitetura SIMNET

A arquitetura SIMNET[15] (*Simulator Networking*) é um ambiente virtual distribuído que foi desenvolvido entre 1983 e 1990 sob o financiamento da DARPA (*Defense Advanced Research Projects Agency*) americana. Seu objetivo principal era oferecer um ambiente de simulação distribuído, a um custo não muito elevado, no qual táticas de combate e batalha pudessem ser simuladas. As unidades simuladas incluíam vários veículos de batalha do exército americano, como tanques M1, helicópteros AH-64 e postos de comando. O projeto desta arquitetura baseou-se em três características principais:

- ⇨ Uma arquitetura orientada a eventos de objetos
- ⇨ Uma noção de nós de simulação autônomos
- ⇨ Um conjunto de algoritmos de modelagem preditiva conhecidos como *Dead Reckoning*[7]

No ambiente simulado, todas as interações entre as entidades são feitas através de notificações de eventos, que são mensagens transmitidas pelo meio de comunicação e descrevem o estado instantâneo de uma entidade em particular. Cada participante da simulação usualmente controla uma entidade do ambiente (tanque, helicóptero, etc.) e é responsável pela manutenção do seu estado.

Existe certa autonomia nesta forma de controle, pois o participante não interage com aqueles que recebem as suas notificações, simplesmente as envia sem receber resposta alguma. Aqueles que recebem as notificações, por sua vez, procedem com as modificações de

estado das suas cópias locais das entidades. Nessa arquitetura, cada participante é, ao mesmo tempo, origem e destino de mensagens de notificação. Mesmo entidades que estejam estáticas devem enviar regularmente (a cada cinco segundos) uma mensagem de aviso indicando que continuam em operação (*heartbeat*).

A SIMNET utiliza extensivamente as técnicas de *Dead Reckoning*, como predição e convergência. Cada participante do ambiente virtual deve acompanhar a atualização do estado da sua entidade, simulando localmente o desvio de comportamento desta em relação à última notificação de estado enviada. Caso o desvio entre a predição e o comportamento real esteja fora de uma faixa aceitável, o participante deve gerar uma nova notificação de estado para efetuar a correção.

A tabela 3.1 ilustra o formato de um dos quadros de informação (PDUs) utilizados no controle de estado das entidades SIMNET. As informações como tipo de veículo, vetor de velocidade e velocidade do motor são utilizadas nas técnicas de predição de movimento.

Tabela 3.1: Quadro de Notificação de Estado do protocolo SIMNET

Nome do Campo	**Conteúdo**	**Tamanho(bytes)**
ID Veículo	Local, Máquina, Veículo	6
Classe de Veículo	Tanque, Simples, Estático, Irrelevante	1
ID da Força	-	1
Aparência	Tipo de Objeto: distinto / outro	8
Coordenadas	Localização: *x, y, z*	24
Matriz de Rotação	-	36
Aparência	-	4
Marcas	Texto	12
Timestamp	-	4
Capacidades	-	32
Velocidade do Motor	-	2
Estacionária	-	2
Veículos com Aparência Variável	Vetor de Velocidade, Azimute da Torre, Elevação da Arma	24

As mensagens de atualização de estado geradas pelos participantes podem possuir uma periodicidade bastante variável, pois é influenciada pelo tipo da entidade. Um helicóptero de combate, por exemplo, gera um número maior de atualizações por unidade de tempo que um veículo terrestre, uma vez que suas velocidade e dirigibilidade são muito maiores. Humanos articulados também geram um grande volume de atualizações, pois possuem muitas partes que se movem constantemente.

A SIMNET não possui servidores centralizados, operando inteiramente através da regeneração de estado por parte dos participantes. Sua escalabilidade provou-se boa quando foram realizados testes automatizados e de simulações com até 850 objetos[16], em 1990.

3.1.1 O Padrão DIS

O projeto SIMNET foi considerado um grande sucesso pelo Departamento de Defesa americano (DoD), o que incentivou o melhoramento da simulação através da criação de novas unidades e formas de interação. No entanto, o protocolo original não foi planejado de forma a ser facilmente extensível e genérico. Deficiências como essas levaram a uma tentativa de formalizar e generalizar um protocolo baseado no SIMNET, culminando na criação do protocolo DIS (*Distributed Interactive Simulation*)[17], que foi padronizado pela norma IEEE 1278, em 1993.

O propósito geral do protocolo DIS é proporcionar a quaisquer participantes, independentemente da classe de veículos ou entidade que utilizam, a interação em um ambiente virtual. A arquitetura por trás do protocolo DIS é muito similar à da SIMNET e, da mesma forma que esta, possui arquitetura orientada a eventos de objetos, noção de nós de simulação autônomos e algoritmos de modelagem preditiva. No entanto, o protocolo DIS é mais genérico e possui uma grande quantidade de campos descritivos, facilitando a introdução de novas entidades na simulação.

O padrão DIS define vinte e sete quadros de informações (PDUs) diferentes, dos quais apenas quatro são amplamente utilizados pela maioria das aplicações: notificação de estado, disparo, detonação e colisão. Os vinte e três quadros restantes nem sempre são implementados pelas aplicações.

A tabela 3.2 ilustra o formato de um dos quadros de informações utilizados no controle de estado das entidades DIS. Como pode ser observado, as PDUs apresentam uma maior quantidade de campos descritivos, detalhando melhor os parâmetros utilizados na predição de movimento. Elas ainda possuem parâmetros variáveis que são a base para a especialização dos tipos de entidades.

Tabela 3.2: Quadro de Notificação de Estado do protocolo DIS

Nome do Campo	**Conteúdo**	**Tamanho(bytes)**
Cabeçalho da PDU	Versão do protocolo, ID do exercício, tipo da PDU, família de protocolo, *timestamp*, tamanho	12
ID da entidade	Local, aplicação, entidade	6
ID da força	-	1
Num. de parâm. da articulação	-	1
Tipo da entidade	Tipo, domínio, país, categoria, subcategoria, extra	8
Tipo da entidade alternativo	Tipo, domínio, país, categoria, subcategoria, extra	8
Velocidade linear da entidade	x, y, z	12
Localização da entidade	x, y, z	24
Orientação da entidade	φ, θ, ϕ	12
Aparência da entidade	-	4
Parâmetros de *Dead-Reckoning*	Algoritmo, outros parâmetros, aceleração linear da entidade, velocidade angular	40
Marcação da entidade	Conjunto de caracteres, marcas	12
Capacidades	-	4
Parâmetros de articulação	Tipo do parâmetro, mudança, ID associado, tipo do parâmetro, valor do parâmetro	n*16

Uma arquitetura que utiliza o padrão DIS pode possuir maior heterogeneidade entre seus participantes, pois o único requisito para aplicações entrarem no ambiente virtual é que implementem a manipulação dos quadros definidos pelo protocolo. O padrão DIS incorpora ainda outras melhorias em relação ao SIMNET. Além da maior quantidade de entidades possíveis, existem também o suporte a efeitos de ambiente e a modelagem dinâmica de terreno. Os efeitos de ambiente permitem um maior grau de imersão no ambiente virtual, simulando fumaça, luz intensa e escuridão, entre outros. A modelagem dinâmica de terreno é realizada geralmente por um servidor de terreno, pois demanda por alto processamento. Sua utilização é justificada somente em alguns tipos de simulação de escala reduzida, pois gera um grande impacto na utilização de largura de banda.

3.2 A Arquitetura NPSNET

A NPSNET[18] (*Naval Postgraduate School Networked Vehicle Simulator*) é um ambiente virtual distribuído orientado à simulação, desenvolvido pelo grupo de pesquisas *The MOVES Institute* da escola naval de pós-graduação de Monterey, Califórnia. Destacou-se por

ser o primeiro Ambiente Virtual Distribuído a utilizar o padrão DIS na sua comunicação e a ser capaz de utilizar comunicação *multicast*, utilizando a rede *Multicast Backbone* (MBone). Seu projeto levou à implementação de várias arquiteturas NPSNET, sendo a NPSNET-V a mais recente.

A arquitetura NPSNET é capaz de simular humanos articulados, objetos invisíveis, além de veículos terrestres, aéreos e marítimos. Também permite a inclusão de novos modelos de entidades, como símbolos, terrenos, marcas de terreno e modelos definidos pelo usuário. A distribuição é alcançada utilizando-se a Regeneração Freqüente de Estado[7] em conjunto com técnicas de *Dead Reckoning*[7]. A comunicação utiliza técnicas *multicast*, permitindo a criação de ambientes virtuais distribuídos de larga escala em redes de comunicação heterogêneas.

Por ser a primeira arquitetura a utilizar o padrão DIS, a NPSNET identificou vários problemas na escalabilidade das aplicações que fazem uso do mesmo:

- ⇨ **Utilização de Largura de Banda:** Muito alta para ambientes virtuais de larga escala.

- ⇨ **Capacidade de Processamento:** As técnicas de *Dead Reckoning*, detecção de colisões e modelagem de interações físicas podem drenar toda a capacidade de processamento dos participantes quando a quantidade de entidades no ambiente virtual é grande.

- ⇨ **Manipulação de Entidades Estáticas:** O padrão DIS exige o envio regular de quadros de notificação de estado, mesmo quando se trata de entidades estáticas.

Para resolver alguns desses problemas relacionados à escalabilidade dos ambientes, a NPSNET utilizou dois elementos chave:

- ⇨ ***Comunicação Multicast***: Utilizando técnicas de *multicast* no lugar do *broadcast* é possível reduzir a utilização de largura de banda e de processamento nos nós.
- ⇨ **Gerenciador de Áreas de Interesse (AOIM):** Gerenciadores que particionam o ambiente virtual em um grupo de ambientes de menor escala, reduzindo as necessidades computacionais e de largura de banda dos participantes. Esse particionamento pode ser feito com base em propriedades espaciais, funcionais ou temporais. As partições são sempre associadas a grupos *multicast*.

3.3 A Arquitetura DIVE

O principal objetivo da arquitetura DIVE[19] é de suportar ambientes virtuais com um número elevado de participantes, além de oferecer uma alta sensação de interatividade. Desenvolvido no *Swedish Institute of Computer Science*, em 1992, a arquitetura DIVE utiliza um banco de dados de mundo distribuído e compartilhado, que é a sua principal característica de projeto. Todas as interações entre os participantes ocorrem neste mundo compartilhado, que é definido como um banco de dados hierárquico de estruturas denominadas entidades.

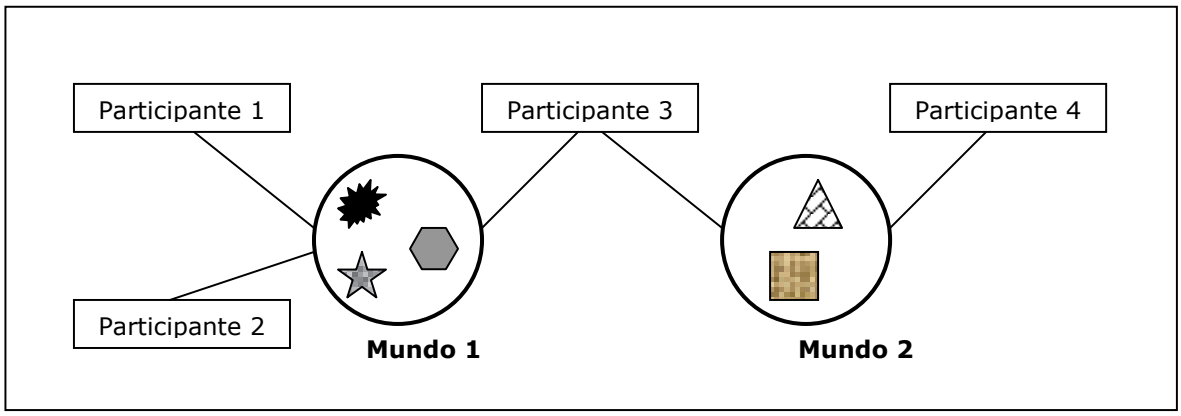

FIGURA 3.1: INTERAÇÃO ENTRE PARTICIPANTES E OS MUNDOS DIVE

A figura 3.1 mostra quatro participantes interagindo com dois mundos DIVE. Os participantes 1 e 2 interagem com o mundo 1, enquanto o participante 4 interage com o mundo 2 e o participante 3 interage simultaneamente com os mundos 1 e 2.

As entidades do banco de dados são objetos que lembram bastante os elementos de um grafo de cena, pois possuem dados associados e podem executar um conjunto de ações pré-definidas. Suas persistências são garantidas apenas enquanto houver participantes no ambiente, pois este deixa de existir quando o último participante o abandona.

A arquitetura DIVE utiliza replicação ativa do banco de dados de mundo, sendo que cada participante possui uma cópia residente na memória da sua aplicação. Este modelo permite às aplicações acessarem o banco de dados do mundo diretamente da memória, reduzindo assim a latência nas interações.

Tipicamente, as adições, remoções e modificações de entidades são feitas primeiramente na cópia local do banco de dados, sendo distribuídas e aplicadas às demais cópias posteriormente. Conceitualmente, as aplicações vêem o mundo como um único banco de dados, mas ele está completamente distribuído, não existindo um servidor central do

mundo. A figura 3.2 ilustra a distribuição do banco de dados do mundo, onde cada participante possui a sua própria cópia.

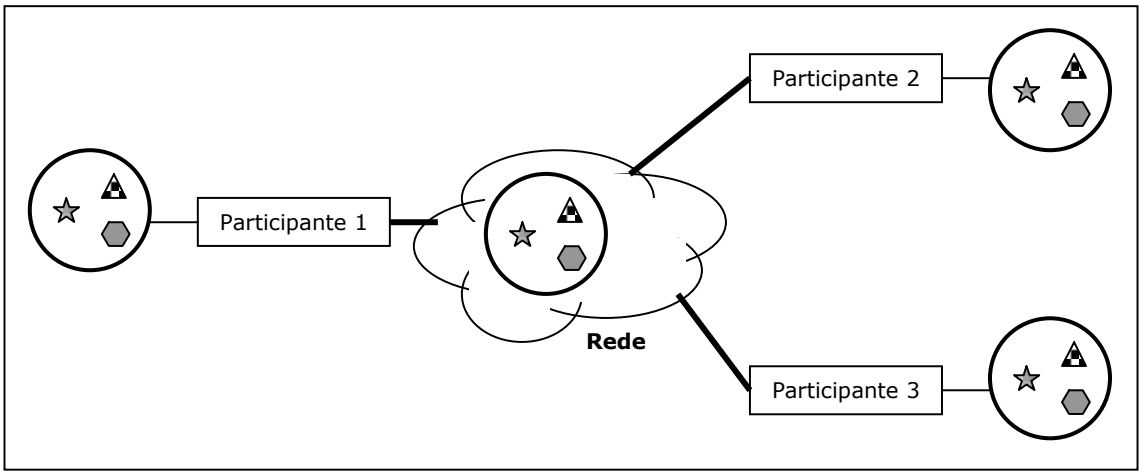

FIGURA 3.2: DISTRIBUIÇÃO DO BANCO DE DADOS DO MUNDO DIVE

Os sistemas de banco de dados distribuídos tradicionais geralmente fazem com que todos os participantes concordem com a informação de estado antes de perpetuarem quaisquer mudanças, o que provoca atrasos diminuidores da sensação de imersão. A arquitetura DIVE utiliza uma forma de consistência menos rígida para alcançar uma maior interatividade:

- ⇨ DIVE tolera que as cópias do mundo estejam ligeiramente diferentes e implementa serviços para garantir a consistência ao longo do tempo.

- ⇨ DIVE pode dividir o mundo em sub-hierarquias que são replicadas e utilizadas apenas por pequenos grupos de participantes que possam se interessar por elas.

A comunicação entre os participantes é feita através de uma organização em grupos. DIVE utiliza um protocolo *multicast* confiável[20] para a transferência das atualizações das entidades, o que garante a entrega das informações ao longo do tempo. Já para dados de mídia contínua, como vídeo e áudio, é utilizado o protocolo *multicast* tradicional.

3.4 A Arquitetura BrickNet

A arquitetura BrickNet[21] foi desenvolvida pelo *Institute of Systems Science* (ISS) da Universidade Nacional de Singapura, em 1991. Seu objetivo principal é possibilitar a criação de ambientes virtuais distribuídos que não se apresentam totalmente idênticos para seus participantes. A aplicação alvo dessa arquitetura é geralmente o projeto cooperativo de

ambientes, onde cada participante é responsável pela sua parte no projeto e por compartilhar informações com os outros participantes.

A rede BrickNet é usualmente composta por alguns servidores e clientes. Cada cliente pode conectar-se a um servidor e requisitar alguns objetos do seu interesse. Esses objetos são depositados no servidor por outros clientes conectados ou por outro servidor. Dependendo da disponibilidade do servidor e dos direitos de acesso do cliente, a requisição é satisfeita. A figura 3.3 ilustra essa forma de organização.

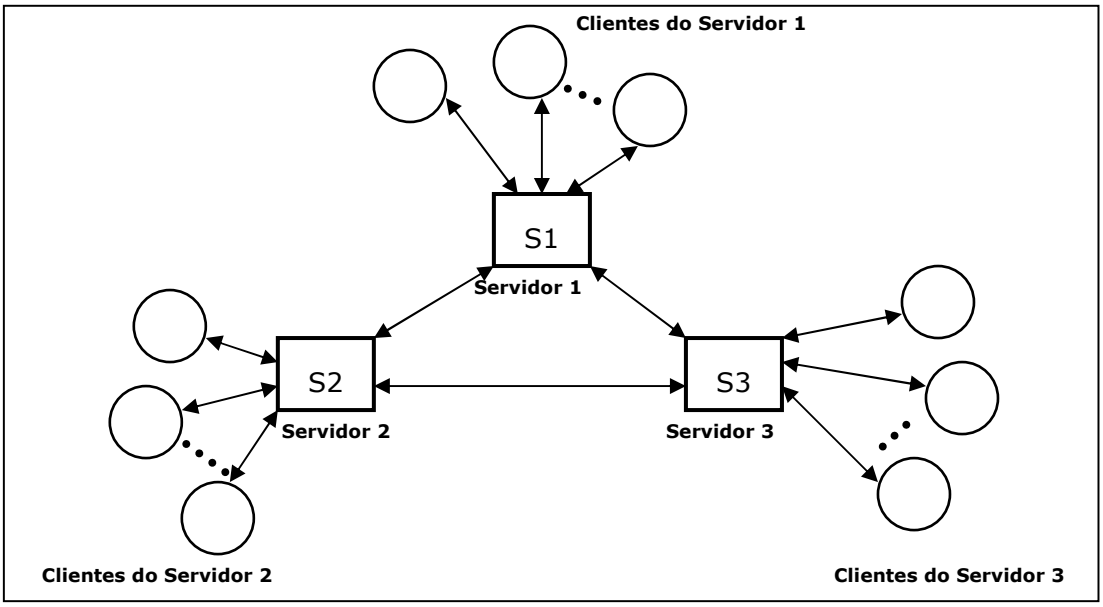

FIGURA 3.3: ORGANIZAÇÃO DA REDE BRICKNET

A arquitetura BrickNet foi implementada na forma de um conjunto de ferramentas (*toolkit*) que oferece suporte para as modelagens gráfica, comportamental e de comunicação de ambientes virtuais distribuídos. Esse ferramental permite ao desenvolvedor construir um ambiente virtual básico, que pode então ser particularizado para algum uso através da criação de objetos de interesse. Os ambientes virtuais construídos utilizando-se a BrickNet possuem algumas características em comum:

- ⇨ **Ambiente Virtual orientado a objetos:** O ambiente virtual da BrickNet é composto por objetos que encapsulam suas propriedades gráficas, comportamentais e de comunicação. Estes ambientes utilizam um interpretador de comandos que possibilita a criação, destruição e modificação de atributos de objetos em tempo de execução.
- ⇨ **Conteúdos Diferentes do Ambiente Virtual:** Os ambientes virtuais da BrickNet não

se restringem em compartilhar um conjunto idêntico de objetos. Cada participante gerencia o seu próprio ambiente virtual e o seu conjunto de objetos, dentre os quais alguns podem ser compartilhados com outros participantes.

⇨ **Organização Cliente-Servidor:** Os servidores BrickNet atuam como *object request brokers* (ORBs) e servidores de comunicação para os seus clientes. Esses servidores comunicam-se entre si para oferecerem serviços de forma transparente aos seus clientes, que não possuem qualquer conhecimento sobre a localização e a forma de organização dos servidores.

A arquitetura BrickNet é organizada na forma de camadas, como mostra a figura 3.4. Com exceção da camada de comunicação, os clientes e servidores possuem camadas diferentes em suas implementações.

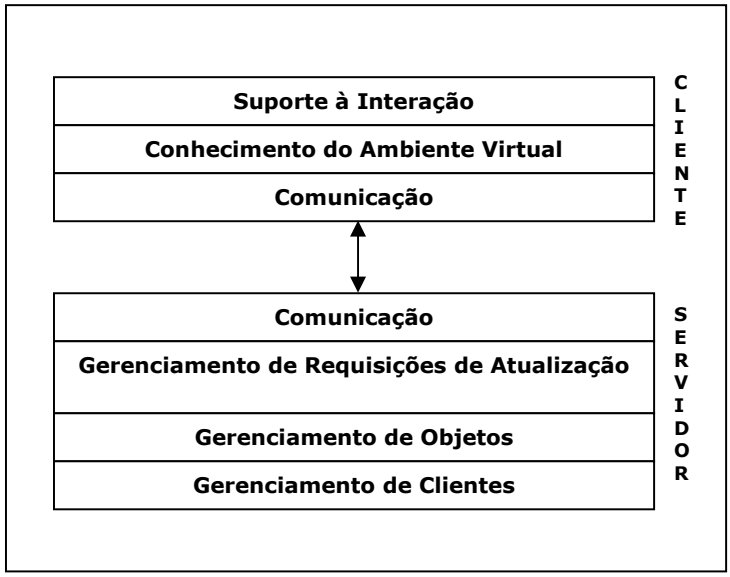

FIGURA 3.4: A ARQUITETURA BRICKNET

A camada de Comunicação é responsável por isolar o cliente e o servidor dos detalhes da comunicação de baixo nível, como gerenciamento de conexões e confiabilidade.

A camada de Conhecimento do Ambiente Virtual gerencia os objetos que compõem o ambiente virtual. Esses objetos podem possuir gerenciamento local ou serem obtidos de servidores BrickNet.

A camada de Suporte à Interação é a interface com o usuário do ambiente virtual. Tem

a função de gerenciar os dispositivos de entrada e os efeitos causados pelas ações dos participantes no ambiente.

A camada de Gerenciamento de Requisições de Atualização é responsável por receber as requisições de atualização de objetos dos demais clientes. Essas requisições são refletidas no banco de dados de objetos e atualizações são enviadas para os clientes neles interessados. Além disso, assegura que as restrições de segurança e acesso sejam respeitadas.

A camada de Gerenciamento de Objetos mantém o registro de todos os objetos depositados pelos vários clientes e as informações sobre seus direitos de acesso.

A camada de Gerenciamento de Clientes é responsável por manter informações específicas dos clientes, como o número das portas de comunicação utilizadas, estado dos clientes e requisições pendentes, além de gerenciar a conexão e desconexão deles.

3.5 A Arquitetura CAVERNsoft

A CAVERN (*CAVE Research Network*) é um grupo de pesquisa formado por empresas e instituições de pesquisa, equipadas com CAVEs e recursos computacionais de alto desempenho interconectados por redes de comunicação de alta velocidade. Criada em 1995, seu objetivo principal é promover a pesquisa nas áreas de Colaboração, Engenharia, Projeto, Educação, Treinamento e Visualização. O CAVERNsoft[22] é um ferramental desenvolvido pelo grupo CAVERN e utilizado como base para todo o *software* colaborativo e de visualização desenvolvido pelo grupo.

A arquitetura CAVERNsoft é centralizada numa estrutura chamada *Information Request Broker* (IRB). O IRB é o núcleo de toda aplicação cliente e servidora CAVERNsoft. Consiste em um repositório de dados persistentes que oferece capacidades básicas de comunicação e de banco de dados. A figura 3.5 ilustra a utilização do IRB pelas aplicações.

Toda aplicação CAVERNsoft utiliza uma interface IRB (IRBi) para a sua comunicação, que, quando acessada, lança no espaço de memória da aplicação um IRB pessoal. O IRB lançado passa então a ser encarregado de toda a comunicação com os outros IRBs e a fazer *cache* de dados. Utilizando a IRBi, um cliente pode formar uma conexão arbitrária com qualquer outro cliente ou servidor para acesso aos seus recursos.

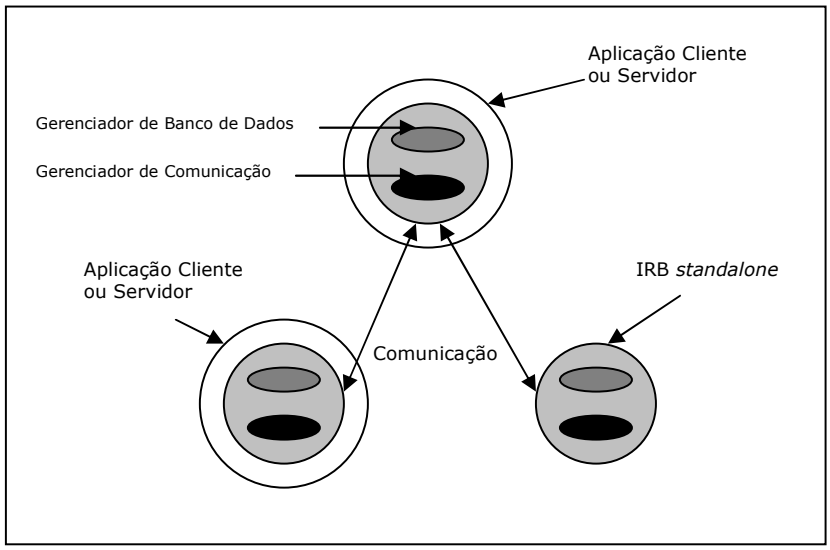

FIGURA 3.5: UTILIZAÇÃO DO *INFORMATION REQUEST BROKER* (IRB)

O IRB é lançado sempre como um processo leve (*thread*) da aplicação. Suas capacidades de comunicação e de banco de dados suprem tanto as necessidades de aplicações clientes como servidoras. Toda aplicação CAVERNsoft é automaticamente cliente e servidora. Como cliente, ela pode conectar-se a outras aplicações para acessar seus recursos. Já como servidora, ela pode aceitar conexões de outros clientes. Essa dupla capacidade, além de ser transparente para o desenvolvedor da aplicação, ainda permite a construção de topologias variadas de comunicação.

A primeira etapa na comunicação entre dois IRBs é a criação de um canal de comunicação entre eles, através do qual é possível o IRB declarar várias chaves locais e remotas utilizadas pela aplicação.

Uma chave é uma alça de manipulação para informações armazenadas no banco de dados do IRB. As chaves são definidas de forma única entre todos os IRBs e podem ser organizadas hierarquicamente, como uma estrutura de diretórios. Podem também ser interligadas entre os IRBs, fazendo com que uma chave local possua o conteúdo de uma chave remota, desde que cada chave local seja ligada a apenas uma chave remota. No entanto, as chaves locais podem receber ligações de múltiplas chaves remotas. Todo esse sistema de chaves simula uma arquitetura de memória compartilhada. A figura 3.6 ilustra a utilização das chaves pelos IRBs. Nesse exemplo, o IRB2 é cliente da Chave1, que se encontra no IRB1. O IRB1, por sua vez, é cliente de Chave2 e Chave3, que se encontram no IRB2.

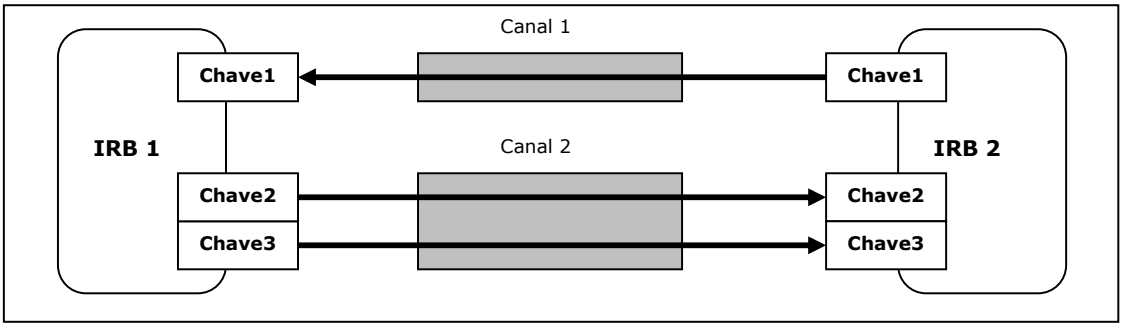

FIGURA 3.6: UTILIZAÇÃO DAS CHAVES PELOS IRBS

A arquitetura CAVERNsoft utiliza essa técnica híbrida de memória compartilhada e banco de dados distribuído para a construção de Ambientes Virtuais Distribuídos. Assim, o grafo de cena do ambiente é construído e armazenado no banco de dados do IRB, tornando possível a sua distribuição entre os participantes, que controlam partes do ambiente virtual e geralmente são os verdadeiros donos das chaves destas regiões. A figura 3.7 ilustra a distribuição do grafo de cena pelas aplicações.

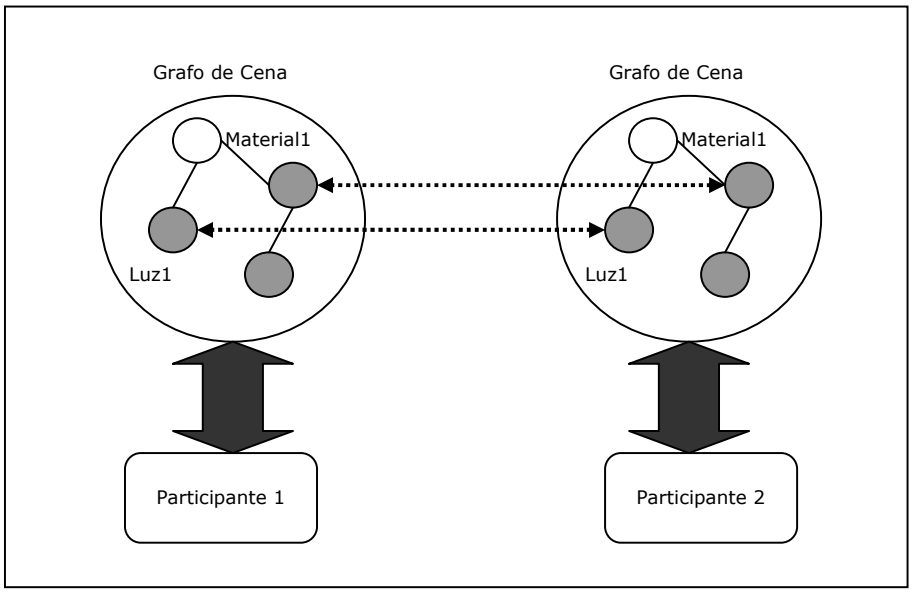

FIGURA 3.7: DISTRIBUIÇÃO DO GRAFO DE CENA PELAS APLICAÇÕES

A arquitetura CAVERNsoft utiliza várias otimizações no processo de acessos às chaves, como *caches* locais e modalidades de distribuição ativa e passiva. A qualidade de serviço dos canais de comunicação e o protocolo utilizado também podem ser especificados em detalhes. A persistência dos Ambientes Virtuais Distribuídos geralmente é tarefa de um servidor dedicado, que se liga a todas a chaves e que faz uma cópia em disco dos seus conteúdos periodicamente.

3.6 A Arquitetura MASSIVE

Criado em 1995 por pesquisadores das universidades de Nottinghan e Manchester, o MASSIVE[23] (*Model, Architecture and System for Spatial Interaction in Virtual Environments*) tem como objetivo principal ser um sistema de Realidade Virtual (RV) Distribuído capaz de suportar centenas e até milhares de usuários simultaneamente. Além disso, propõe-se a ser um ambiente de colaboração, onde os mais diferentes tipos de *hardware* podem interagir.

Sua arquitetura é baseada e totalmente dependente de uma estrutura chamada *Aura Manager*, que é a responsável por gerenciar as *auras* dos objetos. Embora não seja explicitado o modelo de dados adotado, existe o conceito de objetos. Cada objeto existente possui uma *aura* e sua interface. Todas as manipulações e os gerenciamentos dos objetos são feitos por meio de suas interfaces. Além dos objetos, existem ainda as definições de processos e a forma como se comunicam. Os processos visam tirar vantagem do paralelismo e tratam de problemas de granularidade grossa. Cada processo computacional pode suportar qualquer número de interfaces, sendo caracterizado pela combinação de chamadas remotas de procedimento (RPC), atributos e fluxos de dados (*streams*). Essa tripla combinação existe para suportar funções de RV distribuída e transmissão de mídias contínuas, como áudio e vídeo, por exemplo.

A comunicação da arquitetura MASSIVE é baseada na comunicação *unicast*, realizada sobre protocolos de transporte da Internet. A figura 3.8 mostra as entidades da arquitetura MASSIVE.

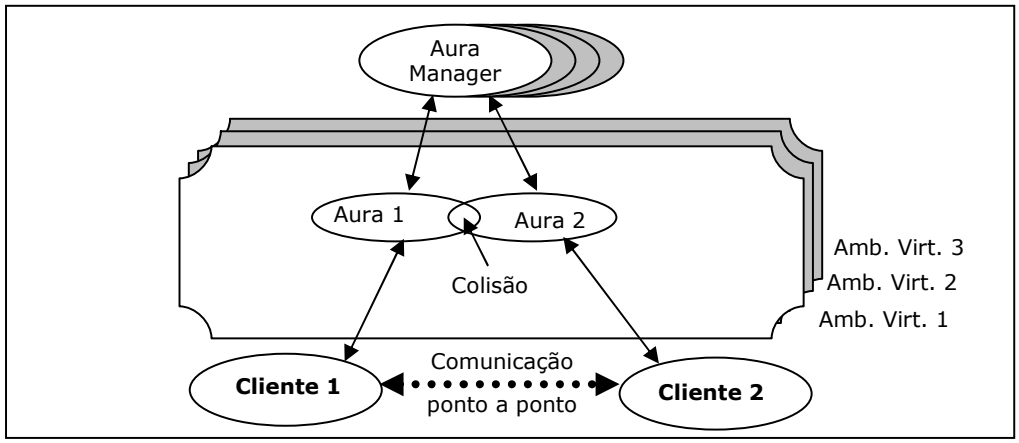

FIGURA 3.8: AS ENTIDADES DA ARQUITETURA MASSIVE

Múltiplos gerenciadores de *auras* podem coexistir. Cada um deles está relacionado a um ambiente virtual distinto. O gerenciador é responsável por identificar colisões entre as *auras* dos objetos, considerando algumas características e atributos do objeto, que podem ser obtidos a partir de suas interfaces. Alguns exemplos de atributos são o foco de atenção (para onde o cliente está olhando), velocidade, direção e sentido do movimento. Sempre que uma colisão é detectada, os clientes estabelecem uma conexão ponto a ponto e iniciam sua interação.

Como essa arquitetura suporta múltiplos ambientes virtuais simultâneos, existe a possibilidade de um objeto realizar a troca de ambiente. Nesse caso, o gerenciador de *auras* anuncia para todas as demais presentes no ambiente que determinada *aura* as está deixando. Após todas as devidas providências terem sido tomadas, o objeto retirante pode ingressar no outro ambiente.

Embora o mecanismo de ingresso em um ambiente seja realizado a partir de um *trader*, o *Spatial Trader*, ele não representa um gargalo para o sistema, pois não interfere no desempenho do ambiente após o ingresso dos clientes. Por outro lado, o fato da arquitetura MASSIVE possuir um gerenciador de *auras* centralizado pode causar sobrecarga de processamento, diminuindo o desempenho do sistema como um todo.

3.7 Uma Comparação entre as Arquiteturas

As arquiteturas analisadas neste capítulo apresentam várias diferenças nos seus projetos e implementações. Essas diferenças são geralmente o resultado do tipo de aplicação alvo para que a arquitetura foi planejada. A tabela 3.3 apresenta uma comparação entre as principais características das diversas arquiteturas analisadas.

As arquiteturas SIMNET e NPSNET são voltadas inteiramente para a simulação de batalha. Nessas aplicações, a movimentação de entidades e a simulação de colisões e interações físicas são os elementos mais importantes. Os cenários das batalhas são geralmente fixos, não havendo a necessidade de modelagem dinâmica. Para esses requisitos, a utilização da técnica de regeneração freqüente de estado é eficiente, pois somente as informações de estado das entidades estão sendo modificadas. As aplicações para estas arquiteturas são geralmente homogêneas e monolíticas, sendo que cada participante possui toda a informação descritiva (geometria, organização, etc.) do ambiente virtual.

A arquitetura BrickNet foi projetada especificamente para o projeto colaborativo de ambientes, onde cada participante trabalha em um determinado espaço deste ambiente. Como

a interação entre os participantes é pequena, geralmente restrita apenas à visualização de modelos, a utilização de um banco de dados particionado é uma abordagem adequada para suas necessidades. Além disso, a capacidade de modelar o comportamento dos objetos do mundo virtual através de roteiros é um diferencial poderoso encontrado nessa arquitetura.

Já a arquitetura MASSIVE, que tem por objetivo, além da colaboração, criar um ambiente que suporte tantos usuários quanto forem possíveis, a interação entre os participantes é algo fundamental. Isso justifica a forma de comunicação adotada, a comunicação *unicast* apenas entre os participantes interagentes.

As arquiteturas DIVE e CAVERNsoft são voltadas para ambientes virtuais distribuídos mais genéricos. Nessas arquiteturas, o grafo de cena do ambiente virtual é diretamente distribuído e manipulado pelas aplicações. Essas arquiteturas utilizam uma versão simulada de memória compartilhada para conseguir a distribuição das informações do grafo de cena. A arquitetura DIVE utiliza o seu protocolo de distribuição para garantir a consistência do ambiente, enquanto que a CAVERNsoft utiliza os IRBs. A arquitetura DIVE é ainda um pouco menos restritiva, permitindo que os ambientes virtuais possuam pequenas discrepâncias temporais. A arquitetura CAVERNsoft é mais rígida, utilizando esquemas de trava de chaves para acessos de escrita. Por esta razão, a arquitetura DIVE tende a possuir maior escalabilidade que a CAVERNsoft, que possui um maior grau de consistência do ambiente.

Tabela 3.3: Comparação entre as Arquiteturas de Ambientes Virtuais Distribuídos

	Ambiente Virtual Distribuído					
	SIMNET	**NPSNET**	**DIVE**	**BrickNet**	**CAVERNsoft**	**MASSIVE**
Propósito	treinamento militar	treinamento militar	geral / colaboração	colaboração	geral / colaboração	geral / colaboração
Número de usuários	850	> 1000	20	n/d	n/d	> 1000
Modelo de dados	mundo homogêneo e replicado	mundo homogêneo e replicado	BD distribuído e compartilhado	BD centralizado e compartilhado	BD distribuído e compartilhado	mundo homogêneo e replicado
Protocolo de Comunicação	*broadcast*	*multicast*	*multicast*	*unicast*	*multicast*	ponto a ponto, *multicast*
Nível de Controle sobre o ambiente	grosso / estado dos objetos	grosso / estado dos objetos	fino / grafo de cena	médio / ações dos objetos	fino / grafo de cena	médio / ações dos objetos
Organização	ponto a ponto	ponto a ponto	ponto a ponto	cliente-servidor	ponto a ponto	ponto a ponto
Replicação de dados	total	total	parcial	parcial	parcial	parcial
Consistência	fraca / regen. freq. de estado	fraca / regen. freq. de estado	média / prot. de distribuição	forte / BD centralizado	forte / IRB	média / *Aura Manager*
Dead-Reckoning	sim	sim	sim	não	não	não

Como pode-se observar, as arquiteturas possuem pontos positivos e gargalos, já que são voltadas para domínios específicos de aplicações. No capítulo 5 é apresentada a arquitetura OpenReality, que busca unir os pontos interessantes já vistos em um *framework* para a construção de Ambientes de Visualização Distribuída de propósito geral, incluindo os Ambientes Virtuais Distribuídos. A conclusão deste trabalho, aliada às dos possíveis trabalhos futuros, tem como um de seus objetivos a inserção de uma nova arquitetura na tabela anterior, contendo os dados exibidos na tabela 3.4.

Tabela 3.4: Objetivos técnicos da Arquitetura OpenReality

Ambiente Virtual Distribuído	**OpenReality**
Propósito	Geral
Número de usuários	n/d
Modelo de dados	mundo homogêneo e replicado
Protocolo de Comunicação	*broadcast, multicast, unicast*
Nível de Controle sobre o ambiente	fino / grafo de cena grosso / estado dos objetos
Organização	ponto a ponto
Replicação de dados	Total
Consistência	forte / prot. de distribuição fraca / reg. freq. de estado
Dead-Reckoning	Sim

3.8 Considerações Finais

Ao realizar a comparação entre os projetos de arquiteturas estudadas, seus objetivos, metodologias e resultados oferecem uma gama de informações que influenciam na escolha das metodologias a serem utilizadas neste trabalho. Além disso, também cria a possibilidade de se avaliar as vantagens e desvantagens de cada arquitetura, permitindo, assim, a escolha das metodologias que oferecem as funcionalidades desejadas para o projeto.

Portanto, baseando-se nos resultados obtidos pelas arquiteturas apresentadas, esse trabalho pode julgar a viabilidade da adoção de cada uma das metodologias estudadas.

4. Mecanismos de Manutenção da Consistência e de Comunicação

Neste capítulo são apresentados os mecanismos de manutenção da consistência de informações e de comunicação utilizados nas aplicações de visualização distribuída. Alguns deles serviram de base para a definição e implementação dos serviços de comunicação e distribuição da arquitetura OpenReality, enquanto que outros foram deixados para trabalhos futuros.

A escolha da forma como será realizada a comunicação das aplicações influencia diretamente no mecanismo a ser utilizado e na consistência permitida. A seguir são apresentadas algumas das formas de comunicação e de manutenção da consistência mais utilizados.

4.1 Formas de Comunicação

Um dos objetivos básicos de uma Aplicação de Visualização Distribuída é dar aos seus participantes a ilusão de que estes estão partilhando o mesmo ambiente. Se um dos participantes modifica em algum momento o estado do ambiente (movendo-se, manipulando um objeto, disparando uma arma, etc.), as mudanças têm que ser observadas praticamente em tempo-real pelos demais participantes. Essa percepção de mesmo ambiente leva à definição de estado compartilhado.

O Estado Compartilhado é o conjunto de todas as informações variáveis sobre o ambiente da aplicação que múltiplas máquinas devem manter para poder sintetizá-lo. Quanto mais similares essas informações se encontrarem nas máquinas dos participantes, maior a consistência ou coerência da aplicação.

Sendo assim, além da consistência, outro fator diretamente ligado à escolha da forma de comunicação é a utilização da largura de banda disponível. Retomando-se os desafios do projeto de Ambientes Virtuais Distribuídos citados por Singhal[2], pode-se resumi-los na série de dependências ilustrada na figura 4.1. A direção das setas utilizadas na figura indica o efeito de influência, por exemplo, a utilização da largura de banda influencia a escalabilidade do ambiente. Estas relações também são válidas para as Aplicações de Visualização Distribuída.

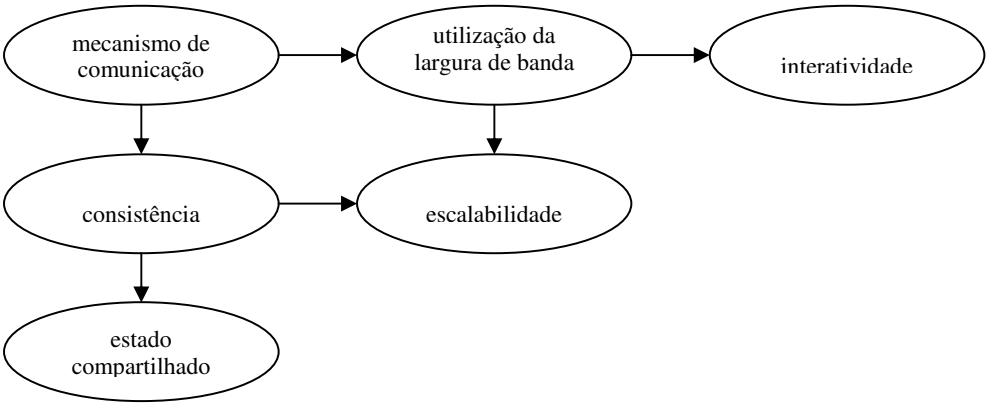

FIGURA 4.1: REPERCUÇÃO DA ESCOLHA DO MECANISMO DE COMUNICAÇÃO

O caso mais simples de distribuição é quando apenas dois participantes interagem na aplicação de visualização. A comunicação é feita diretamente entre as duas máquinas, mesmo que exista uma rede heterogênea entre elas. Nessa configuração, os participantes podem utilizar a largura de banda disponível no meio de comunicação que os une para a troca de informações de atualização de estado. Do ponto de vista lógico, as duas máquinas possuem as mesmas atribuições, controlando suas entidades locais e enviando as informações de atualização de estado para manter os dados da aplicação coerentes em ambos os lados. A figura 4.2 ilustra esta situação.

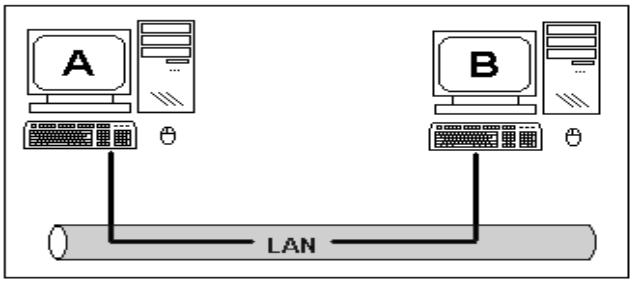

FIGURA 4.2: APLICAÇÃO COM APENAS DOIS PARTICIPANTES

Contudo, raramente esta situação ocorre. Geralmente, a quantidade de participantes tende a ser maior, aumentando a complexidade da aplicação de visualização distribuída. Questões como forma de organização, distribuição de informação e uso da largura de banda são essenciais para determinar a sua escalabilidade.

Alguns ambientes virtuais distribuídos utilizam um modelo de organização do tipo cliente-servidor[21], possuindo um ou mais servidores responsáveis por manter e organizar as informações do estado corrente do ambiente. Cada participante comunica-se diretamente apenas com o servidor ao qual está conectado, que, por sua vez, é responsável por efetuar as

mudanças necessárias no estado do ambiente e propagá-las através de cópias das informações de atualização aos outros participantes.

A utilização de servidores centrais favorece o emprego de várias técnicas que visam à economia de largura de banda total do sistema. Uma das técnicas utilizadas é a compressão ou agregação de pacotes. Nessa técnica, um servidor pode analisar o tamanho das unidades de informação (PDUs) que estão pendentes para envio e verificar se é possível agregar duas ou mais PDUs em uma única mensagem, buscando maximizar a utilização dos pacotes de dados na rede de comunicação. Com isso, pode-se conseguir uma economia no meio de comunicação. Além disso, os servidores podem ainda levar em conta as áreas de interesse [20] dos participantes, evitando, assim, que participantes recebam informações não relevantes para sua visão no momento. Tomando um jogo como exemplo, um participante que se encontra no último andar de um edifício, não precisaria receber as informações da movimentação de outro participante localizado em uma das salas do primeiro andar desse edifício. Essa informação poderia ser irrelevante naquele momento.

A utilização dessas técnicas abre espaço para a criação de aplicações mais complexas ou com um número maior de participantes. É claro que, mesmo com estes cuidados, os próprios servidores podem se tornar o gargalo do sistema, pois podem não ter o poder de processamento suficiente para a demanda. A figura 4.3 apresenta um exemplo de interligação lógica com múltiplos servidores.

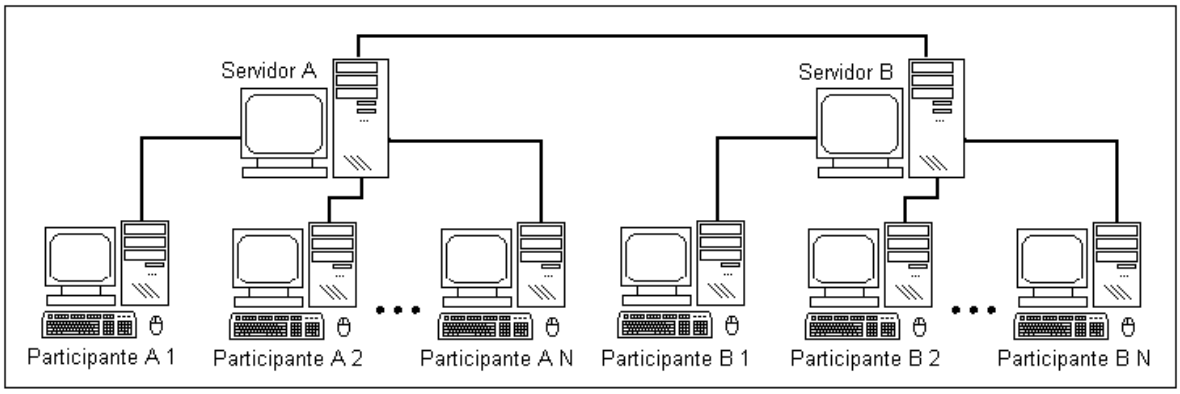

FIGURA 4.3: INTERLIGAÇÃO LÓGICA DE APLICAÇÕES COM MÚLTIPLOS SERVIDORES

Quando é esperado das aplicações de visualização distribuída um grande potencial de escalabilidade, não é conveniente ficar à mercê da capacidade limitada de processamento dos servidores. Diante disso, uma forma de organização inerentemente distribuída pode ser mais eficiente[16]: a organização ponto a ponto.

Aplicações organizadas desta forma não possuem uma figura centralizadora. Os participantes devem ser capazes de comunicar-se para manter o ambiente coerente. A modalidade distribuição de informação utilizada varia de acordo com a infra-estrutura de

comunicação em uso pelos participantes. Em um ambiente limitado a uma rede local, é possível utilizar comunicação *broadcast* ou *multicast*. Já em um ambiente heterogêneo, só é possível a utilização de *multicast*, pois os roteadores normalmente bloqueiam todo o tráfego *broadcast*. A comunicação *multicast* é mais eficiente em qualquer um dos casos, pois permite a utilização de algoritmos[24] que separam os participantes em grupos de interesse, baseados na sua localização física no ambiente e em alguns outros fatores, o que reduz a quantidade de informação trafegando na rede.

A figura 4.4 apresenta um exemplo de interligação entre vários participantes sem a presença de servidores, em uma rede heterogênea, utilizando *multicast*. No exemplo, os participantes A, C e D fazem parte do mesmo grupo de comunicação. Caso o participante A necessite enviar mensagens para o seu grupo, ele enviará apenas uma PDU, que será então passada adiante pelos roteadores até chegar à rede local destino, onde será captada apenas pelos participantes C e D.

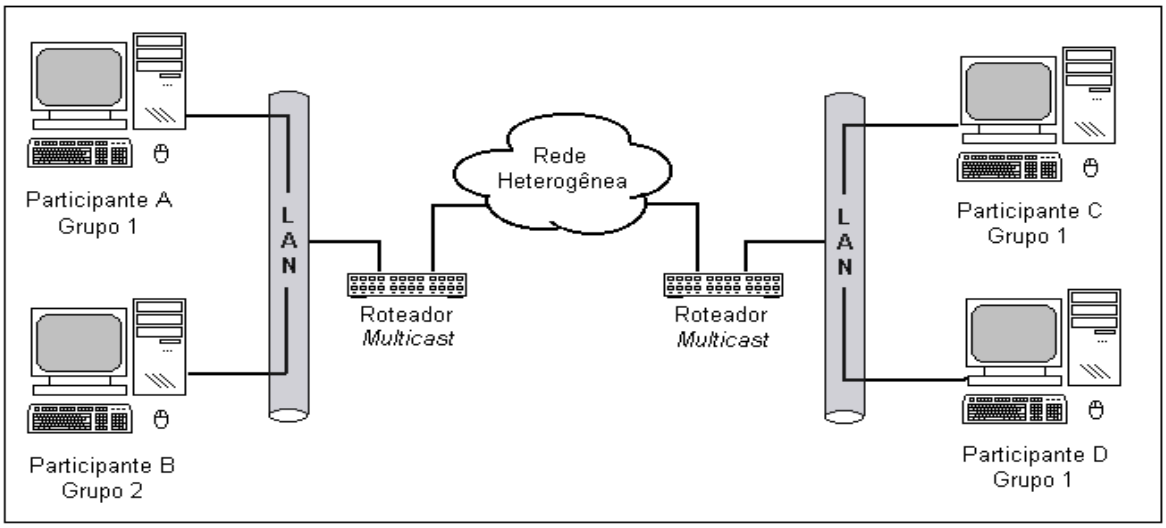

FIGURA 4.4: INTERLIGAÇÃO DE APLICAÇÕES EM UMA CONFIGURAÇÃO PONTO A PONTO

Tendo apresentado algumas das formas de comunicação mais utilizadas, a seção seguinte apresenta alguns dos mecanismos de manutenção de consistência existentes.

4.2 Formas de Manutenção da Consistência

Em uma Aplicação de Visualização Distribuída, as informações (como modificações de estado) geralmente são produzidas individualmente nas máquinas dos participantes, sendo propagadas para todos os demais participantes posteriormente. Elas são usualmente conseqüências das ações por eles tomadas. Devido aos atrasos criados durante o processamento e à própria natureza das redes de comunicação, que possuem uma latência

inerente, essas informações sempre levam algum tempo para serem absorvidas pelas aplicações locais de todos os participantes. Se não houver um esquema de trava, que inibia o participante que produziu as mudanças de executar novas ações, será alta a probabilidade de que ele faça novas modificações no ambiente, antes mesmo das anteriores serem completamente espelhadas. A figura 4.5 apresenta um exemplo desta situação.

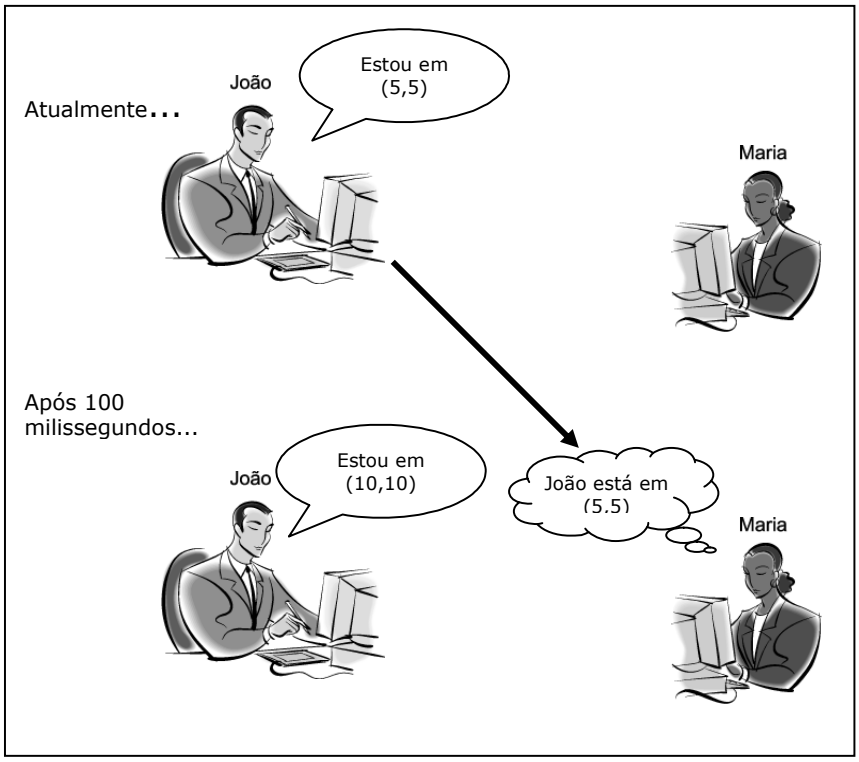

FIGURA 4.5: EXEMPLO DE INCONSISTÊNCIA DO CONTEXTO DA APLICAÇÃO

No exemplo ilustrado acima, o participante João encontra-se andando pelo ambiente. Seu sistema, em um determinado momento, gera uma mensagem de atualização de estado para os demais participantes (Maria), indicando a sua posição atual. Devido ao atraso no processamento e à latência da rede, essa informação chega até Maria somente após 100 milissegundos. Mas, durante este intervalo, João continuou a mover-se, e, no momento em que Maria recebe sua posição e atualiza seu ambiente, ele já se encontra em outra posição. Esse é um exemplo típico da dificuldade em manter-se a consistência nos ambientes que compartilham informações, como os de visualização distribuída.

Por outro lado, caso seja exigida uma consistência total das informações do ambiente da aplicação, então toda modificação de estado deve ser seguida de um período em que novas modificações não são permitidas, até que as mensagens de confirmação cheguem, indicando que todos os participantes já fizeram suas atualizações. A figura 4.6 apresenta um exemplo desta situação.

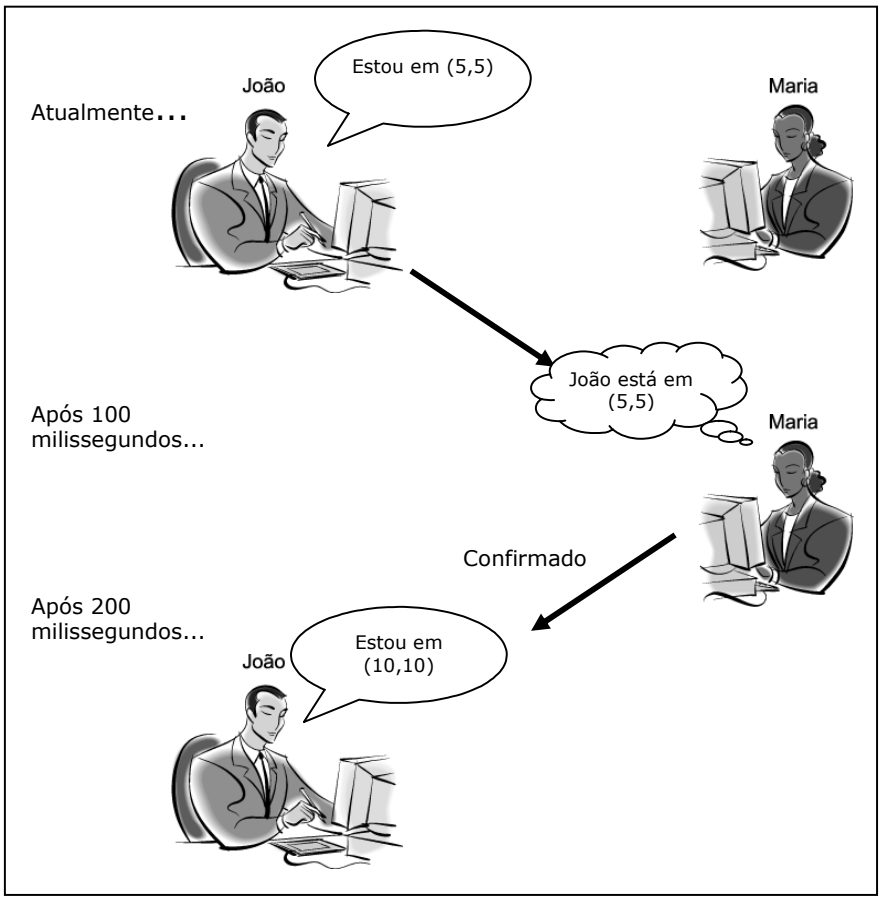

FIGURA 4.6: EXEMPLO DE CONSISTÊNCIA TOTAL

Nesse exemplo, o participante João envia sua mensagem de atualização de estado para Maria, mas o sistema o impede de continuar movendo-se enquanto não receber dela uma mensagem de confirmação. Depois dos primeiros 100 milissegundos, João ainda encontra-se na posição que indicou na sua mensagem. Somente após os 200 milissegundos, quando recebe a confirmação, é que ele pode continuar a mover-se. Pode-se concluir facilmente que uma arquitetura, ao fazer uso da consistência absoluta, terá sua interatividade sensivelmente abalada. Se toda ação ou mudança de estado necessitar de uma confirmação, haverá grandes intervalos, onde o sistema estará paralisado esperando confirmações. Entretanto, caso o sistema não exija uma consistência absoluta, mas existirem entidades que sofram mudanças freqüentes de estado, é necessário que o sistema gere mensagens, regularmente, indicando essas mudanças, de modo a manter a inconsistência do ambiente dentro de uma faixa tolerável.

A tabela 4.1 apresenta uma comparação entre ambientes com consistência absoluta e variável.

Tabela 4.1: Comparação entre Consistência Absoluta e Variável

	Consistência Absoluta	**Consistência Variável**
Consistência Experimentada	Idêntica para todos	Determinada pelos dados recebidos em cada participante
Suporte a dados dinâmicos	Baixo: O protocolo de consistência limita	Alto: Limitado apenas pela largura de banda e capacidade de processamento
Exigências de comunicação	Latência baixa, alta confiabilidade	Possibilita a utilização de diversos tipos de rede
Número máximo de participantes	Baixo	Potencialmente alto

Uma Aplicação de Visualização Distribuída que possui uma grande quantidade de entidades mudando freqüentemente de estado, ocupará toda a largura de banda disponível enviando mensagens para a manutenção da sua consistência. Nesse caso, deve haver um balanceamento entre o número de entidades que geram atualizações freqüentes, a consistência mínima desejada e a largura de banda disponível.

Entre as maneiras de garantir a consistência das informações das aplicações de visualização distribuída está a utilização de repositórios centralizados de informação. Nesse modelo, todas as informações de estado do ambiente devem ficar armazenadas em um repositório centralizador. Todo o acesso às informações de estado é monitorado e operações de escrita são sujeitas a mecanismos de trava e sincronização. Para minimizar o acesso ao repositório, *caches* locais podem ser empregados, porém mecanismos de invalidação de seus conteúdos devem ser aplicados para que a consistência continue sendo mantida.

Os dois tipos mais comuns de repositórios centralizados são os repositórios em servidor e os repositórios virtuais.

4.2.1 Repositórios em Servidor

Esse é o modelo mais comum de repositório centralizado[25]. Nele, uma máquina denominada servidor, é responsável por manter todas as informações a respeito do estado compartilhado do ambiente. Ela geralmente mantém um processo em execução, capaz de armazenar e recuperar informações, baseadas em uma chave de acesso dada, de forma similar a um sistema de arquivos distribuído. Os participantes usualmente mantêm uma conexão com o servidor através da qual realizam as operações de leitura e escrita de informações. A figura 4.7 apresenta uma representação deste modelo.

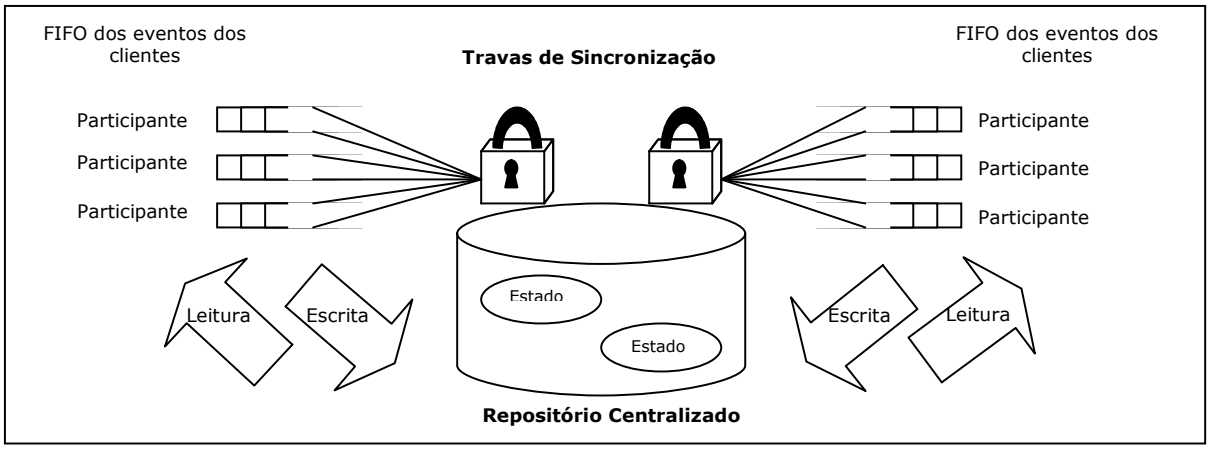

FIGURA 4.7: REPOSITÓRIO EM SERVIDOR

Obviamente, este modelo apresenta algumas desvantagens:

1. Se houver algum problema de travamento do servidor e não existir algum tipo de redundância do repositório, toda informação de estado do ambiente será perdida;
2. A manutenção de uma conexão TCP para cada participante pode gerar problemas, pois geralmente os sistemas operacionais limitam o número máximo de conexões por processo;
3. O servidor pode facilmente tornar-se um gargalo.

As desvantagens resultantes da utilização de um servidor central podem não ser aceitáveis para algumas Aplicações de Visualização Distribuída.

4.2.2 Repositórios Virtuais

O modelo de repositórios virtuais tende a reduzir a dependência gerada por um servidor central. A centralização das informações de estado da aplicação de visualização distribuída ainda existe, mas, nesse caso, é apenas uma centralização lógica.

Macedonia e Zyda[14] afirmam que isso pode ser alcançado utilizando-se um protocolo de consistência distribuída, no qual existe a ilusão de um repositório centralizado. Assim, com esse protocolo, os participantes podem trocar mensagens entre si diretamente, disseminar as atualizações de estado, garantir que todos recebam tais atualizações e determinar a ordem global das atualizações.

Uma vez que a centralização das informações é apenas lógica, conseguem-se algumas vantagens em relação ao modelo anterior. A distribuição física das informações elimina os possíveis gargalos de processamento no servidor central, pois o acesso a elas é feito em todos os participantes. Os possíveis gargalos na conexão de rede do servidor também são eliminados, pois, agora depende-se das conexões de rede individuais dos participantes. Finalmente, a distribuição permite uma melhor tolerância a falhas, possibilitando ao ambiente continuar em operação, mesmo na falta de uma das máquinas participantes. A desvantagem desse modelo é a complexidade presente no protocolo de consistência distribuída.

A consistência absoluta em um ambiente de visualização distribuída nem sempre é um requisito necessário para as suas aplicações. Como bem observaram Pantel e Wolf[26] em seus experimentos, nos jogos multi-usuários 3D, por exemplo, pequenas variações na consistência do ambiente são completamente aceitáveis. Existem ainda os ambientes que, devido ao número máximo de usuários planejado, não podem suportar os requisitos de comunicação e processamento exigidos pela consistência absoluta (repositórios centralizados). Nesses casos, é comum a utilização de uma abordagem diferente para a atualização de estado, a regeneração freqüente de estado.

4.2.3 Regeneração Freqüente de Estado

Nesse modelo, cada entidade do ambiente está sob o controle de uma aplicação participante específica. Cada aplicação é responsável por enviar, periodicamente, informações sobre o estado completo das entidades que controla a todas as demais participantes. Diferentemente do modelo de repositório centralizado, se uma aplicação precisa saber o estado de uma entidade que não controla, ela simplesmente acessa o registro da última atualização de estado recebida, referente àquela entidade. Não existe a requisição explícita de informações de estado pelas aplicações, o que certamente simplifica bastante o protocolo de comunicação e diminui a sua sobrecarga. Os primeiros ambientes de visualização, como a SIMNET[15] e os jogos multi-usuários 3D de computador, como Doom ou Quake, foram os precursores na utilização desta técnica.

Embora a sua utilização seja simplificada, existem alguns problemas inerentes à regeneração freqüente de estado. Como cada aplicação é responsável por um conjunto limitado de entidades, deve haver algum mecanismo para a atualização do estado de entidades que não estejam na aplicação local. Se cada aplicação participante simplesmente enviasse uma atualização de estado referente a qualquer entidade, poderia haver uma grave inconsistência no ambiente, pois, simultaneamente, várias aplicações poderiam enviar atualizações do estado

de uma mesma entidade. Agravando ainda mais a situação, algumas aplicações poderiam receber essas notificações em ordens diferentes. Para evitar esse tipo de inconsistência, existe o conceito de propriedade, no qual cada entidade do ambiente possui uma única aplicação proprietária e esta é a responsável pelo envio das informações de atualização de estado das entidades que possui.

Uma solução comumente adotada é a utilização de um gerenciador de propriedade que autorize a transferência de propriedade de entidades de uma aplicação para outra no ambiente. Esse gerenciador pode ser implementado na forma de um servidor centralizado ou, ainda, através de um protocolo de trava distribuído. Quando uma aplicação necessita alterar o estado de uma entidade cuja propriedade não lhe pertence, ela envia uma mensagem ao atual proprietário dessa entidade. Esse, por sua vez, aciona o gerenciador de propriedade, que então autoriza a transferência da entidade. A partir desse momento, a aplicação requisitante passa a ser responsável pelas atualizações de estado daquela entidade.

A figura 4.8 ilustra duas formas de atualização de estado de entidades remotas. Na primeira (a), a propriedade da entidade é simplesmente transferida de uma aplicação para outra, que passa então a gerar as atualizações. Na Segunda (b), é utilizada uma forma indireta de atualização de estado, onde a aplicação requisitante pede à proprietária para modificar o estado da entidade e propagar as mudanças. Essa segunda forma pode ser utilizada quando a aplicação a requisitante não necessita fazer alterações na entidade com grande freqüência, não justificando, assim, uma transferência de propriedade.

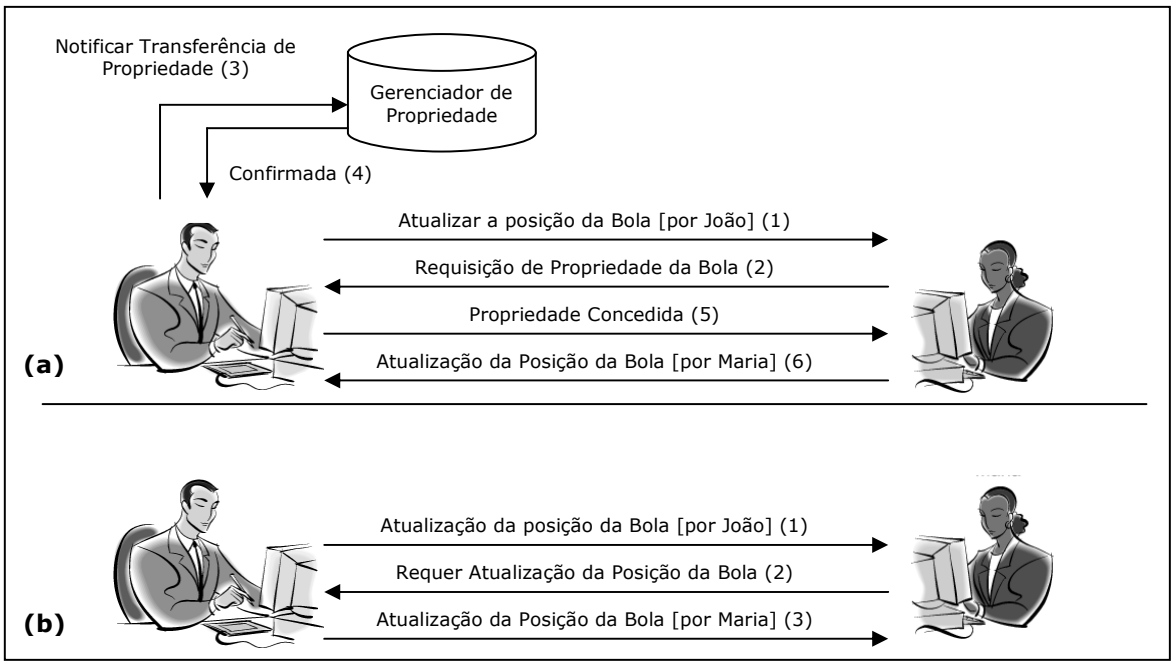

FIGURA 4.8: ATUALIZAÇÃO DO ESTADO DE ENTIDADES REMOTAS

Nos dois modelos vistos até agora, os sistemas dos participantes obtêm os estados dos seus ambientes através das informações recebidas pela rede, tanto consultando um repositório como recebendo regularmente notificações de mudança de estado. Para amenizar a necessidade do envio tão freqüente de atualizações de estado, as aplicações dos participantes podem fazer uso de técnicas preditivas, que irão oferecer uma aproximação do estado das entidades nos períodos entre as chegadas de atualizações. Essas técnicas foram intituladas de *Dead-Reckoning*.

Uma Aplicação de Visualização Distribuída, como um ambiente imersivo, por exemplo, pode sintetizar imagens para o seu participante a uma taxa de trinta quadros por segundo, enquanto recebe notificações de mudança de estado a uma taxa de apenas uma atualização por segundo. Nesse caso, se não for utilizada uma técnica de predição, este participante irá ver as entidades sendo alteradas apenas a cada segundo, o que reduziria sensivelmente as sensações de realismo e interatividade deste ambiente.

4.2.4 Predição e Convergência

As técnicas de *Dead-Reckoning* são compostas basicamente por dois elementos chave: a predição e a convergência. A predição é utilizada para aproximar o estado de uma entidade, com base nas últimas informações de estado recebidas e no intervalo de tempo transcorrido desde a última atualização. A convergência é utilizada para corrigir a possível discrepância entre o estado previsto e o estado real, no momento da chegada da nova notificação de estado.

A figura 4.9 apresenta um problema típico de predição e convergência. Nesse exemplo, uma bola encontra-se em movimento em um ambiente de uma aplicação de visualização distribuída. Regularmente, a aplicação proprietária da entidade bola envia atualizações de estado contendo a posição atual da bola e um vetor de velocidade. No instante 3, por exemplo, é emitida uma notificação indicando que a bola encontra-se na posição (4, 5) e possui velocidade (3, 2). Com base nessa informação, uma aplicação remota de um participante consegue predizer o movimento da bola e, até o instante 4, atualiza sua posição. No instante 4, chega uma nova notificação, indicando que a posição real da bola, na verdade, é outra. A aplicação remota deve então adotar um novo caminho de predição, com base na posição onde está a bola no seu ambiente e o novo vetor de velocidade real, fazendo com que essa discrepância de localização desapareça com o tempo.

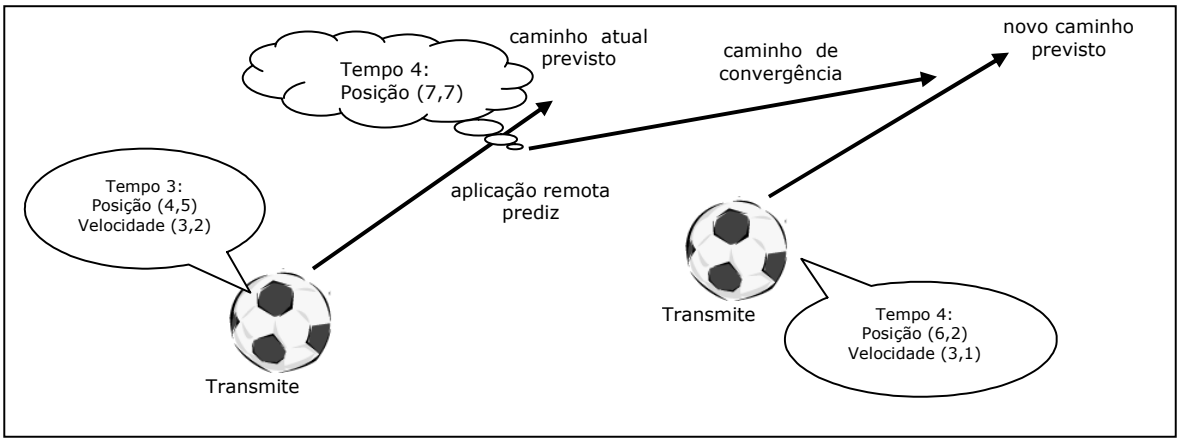

FIGURA 4.9: EXEMPLO DE PREDIÇÃO E CONVERGÊNCIA

4.3 Protocolos e Mecanismos de Distribuição

Independentemente da forma de comunicação adotada pela aplicação, os recursos de distribuição e protocolos de comunicação que a implementam geralmente são os descritos a seguir. Juntamente com suas respectivas descrições estão algumas de suas vantagens e desvantagens, derivadas de suas características particulares.

4.3.1 Transmission Control Protocol

O Protocolo de Controle de Transmissão (TCP)[27], embora seja o protocolo mais comum na Internet hoje em dia, não é um protocolo ideal para certos tipos de Aplicações de Visualização Distribuída. A transparência oferecida pelo TCP, no que diz respeito ao seu controle de fluxo, garantia de entrega e controle de seqüência dos dados, é muito importante nos sistemas que exigem consistência absoluta, pois facilita a implementação e o controle da comunicação por parte da aplicação. Porém, nos sistemas que exigem uma comunicação mais intensa, requisitando um maior grau de interatividade entre seus participantes, o TCP não se adequa perfeitamente. Isso se deve à questões próprias do protocolo, como, por exemplo, seu mecanismo de reconhecimento (ACK) e de controle de fluxo. O primeiro, embora não aumente muito a utilização da banda com o envio dos ACKs[28], retarda o envio dos próximos segmentos de dados, o que aumenta a latência entre as atualizações de estado no caso das aplicações de visualização distribuída.

Por sua vez, o mecanismo de controle de fluxo pode causar a redução na taxa de envio de informações de atualização de estado pela metade em determinados instantes.

Ambos os problemas podem causar certo atraso na aplicação e resultar na perda da sensação de realismo do ambiente e comprometer sua interatividade.

4.3.2 User Datagram Protocol

O Protocolo de Datagramas do Usuário (UDP)[27], ao contrário do TCP, é um protocolo não orientado à conexão. Possui toda a sua semântica baseada nos pacotes, isto é, não é armazenado e nem realizado qualquer tipo de controle a respeito do estado da sessão da comunicação.

Devido ao fato de não possuir mecanismos de controle de fluxo, de seqüência e nem garantir a entrega dos pacotes, o UDP é um protocolo que possui uma ocupação relativamente menor de largura de banda quando comparado ao TCP. Isso faz com que a utilização do UDP torne-se uma forma de comunicação interessante em Aplicações de Visualização Distribuída, principalmente para os que se propõem a suportar um grande número de interações, pois pode reduzir a latência da comunicação que existe ao se utilizar o TCP.

Porém, embora seja um protocolo bastante interessante em termos de economia no consumo da largura de banda, existem as aplicações que, além da alta interatividade, necessitam de grande consistência. Segundo Tanenbaum[28], embora as taxas de perdas e erros de transmissão nas redes locais (LANs) sejam baixas, essa não é a realidade para a Internet. Sendo assim, nos casos de ambientes que necessitam de altas interatividade e consistência, a utilização do UDP não é a mais apropriada.

4.3.3 IP *Broadcasting*

O IP *Broadcasting*[29] é um mecanismo proposto para resolver um problema presente tanto no TCP quanto no UDP. Embora o UDP não utilize tanta banda quanto o TCP, ambos fazem com que o desempenho do sistema seja reduzido ao ser aumentado o número de comunicações ponto a ponto entre os participantes. Isso se deve ao fato do número de pacotes trafegando nos segmentos da rede também aumentar.

Utilizando-se IP *Broadcasting*, apenas um pacote trafega pelo segmento de rede e todos os participantes nele localizados o recebem. Dessa forma, um aumento no número de participantes não deterioraria o desempenho do ambiente da forma como TCP e UDP o fariam.

Apesar de ser uma solução eficiente, o IP *Broadcasting* também possui efeitos colaterais próprios que podem não ser interessantes para alguns tipos de aplicações. Em algumas aplicações, nem sempre é desejado que todos os participantes recebam as informações transmitidas na rede. Assim, com o IP *Broadcasting*, deve ficar a cargo da

aplicação o tratamento de cada pacote de informação recebido, avaliando se este contém alguma informação relevante ou não para o participante local. Esse processamento extra de avaliação da relevância da informação recebida pode reduzir o desempenho do sistema, caso o número de pacotes irrelevantes seja elevado.

Além disso, o IP *Broadcasting* geralmente não é permitido na Internet, sendo sua utilização restrita apenas às redes locais.

4.3.4 IP *Multicasting*

O IP *Multicasting*[27], diferentemente do TCP e UDP, que transmitem dados de um emissor para um único receptor, e do IP *Broadcasting*, que transmite dados de um emissor para todos os outros participantes, apresenta-se como um meio termo na comunicação de aplicações: a possibilidade de enviar dados de um emissor para um grupo seleto de receptores, que pode conter entre zero e até mesmo todos os participantes. Essa possibilidade de comunicação de grupo possibilita uma economia na utilização da banda, principalmente quando comparada às taxas de utilização *unicast* e *broadcast* para várias máquinas. Na prática, é transmitido apenas um datagrama por segmento de rede, que pode ser recebido pelas máquinas com as aplicações dos participantes do grupo quando atinge o segmento em que elas se encontram.

A associação a um grupo é realizada de forma dinâmica, isto é, as estações podem iniciar e encerrar sua participação em um grupo a qualquer momento. Não há restrições quanto ao número de participantes de um grupo e nem a respeito de sua localização geográfica. Além disso, uma aplicação pode ser participante de mais de um grupo *multicast* ao mesmo tempo.

Essas e outras características fazem do *multicasting* um protocolo muito interessante para ser utilizado nas Aplicações de Visualzação Distribuídas. Elas permitem que o ambiente seja particionado em grupos de interesse (AOIM), de forma a hierarquizar toda a comunicação entre esses grupos, da forma como Singhal e Zyda[7] propõem.

4.3.5 Mecanismos de RPC e Objetos Distribuídos

Da mesma maneira como os mecanismos anteriores são freqüentemente encontrados em aplicações de visualização distribuída, também existe uma certa tendência pela utilização das soluções de *middleware* automatizadas[3]. Essas soluções, geralmente possibilitadas e auxiliadas por geradores automáticos de *stubs* cliente e servidor, encapsulam toda a complexidade da comunicação de dados e tornam esse processo transparente para o desenvolvedor. Sementille[3], por exemplo, utilizou CORBA[30] como ferramenta de

middleware em seu trabalho, pois seu interesse era realizar uma modelagem orientada a objetos durante a construção de Ambientes Virtuais Distribuídos. Isso fez com que sua atenção ficasse focada na aplicação e distante das questões de comunicação em si.

No modelo de RPC proposto por Coulouris[31], um procedimento, que executa em uma máquina remota, pode ser chamado a partir de outra qualquer, desde que ambas possuam os *stubs* necessários. Isso torna simples a utilização da rede em aplicações, porém, é possível que a sobrecarga de informações trocadas entre os *stubs* consuma mais largura de banda que o necessário.

Um outro modelo, que evoluiu do RPC, é o de Objetos Distribuídos. Nesse modelo, não temos mais o conceito de procedimentos remotos, mas sim, de objetos que possuem métodos e atributos remotos. A arquitetura organizacional desses objetos segue o modelo cliente-servidor[31], formado pelos objetos que servem suas funcionalidades e pelos objetos que as requisitam.

Como no modelo de Objetos Distribuídos toda a comunicação também ocorre de forma transparente para o desenvolvedor, geralmente existe um objeto centralizador, utilizado para fins de registro, atualização, armazenamento e localização dos objetos distribuídos. Esse objeto centralizador recebe o nome de *broker* em muitas das arquiteturas de *middleware*.

Porém, embora seja um modelo bastante interessante para a construção de aplicações orientadas a objeto, as arquiteturas de comunicação baseadas em objetos distribuídos geram uma sobrecarga na utilização da largura de banda. Da mesma maneira como ocorre com o modelo de chamadas de procedimentos remotos, o preço pago pela transparência na comunicação é o aumento da taxa de dados trocados entre os *stubs* cliente e servidor.

4.3.6 Protocolos com QoS

Além dos objetivos alvo das aplicações de visualização distribuída já citados, pode ser desejada a criação de ambientes que permitam a reprodução de mídias como sons e vídeos. Para esses casos é necessária a existência de mecanismos que ofereçam suporte aos requisitos de QoS, como o controle de atrasos e de suas variações, de consumo da largura de banda, entre outros. Um protocolo para a transmissão de *streams* de mídia que vem ganhando destaque é o FEC-UDP (*Forward Corrected Error*)[32], bem como algumas de suas variações, pois buscam minimizar a taxa de erros percebida pelas aplicações que utilizam transmissão de *streams* de dados contínuos. Embora existam outros protocolos até mais conhecidos, a maioria trata da transmissão de tipos específicos de *streams*, que não serão abordados por fugirem do escopo deste trabalho.

4.4 Considerações Finais

Este capítulo apresentou as diversas técnicas existentes para a implementação das metodologias que tratam da distribuição de informações em aplicações de visualização distribuída. De maneira geral, apresenta as técnicas comumente adotadas para superar os desafios específicos do projeto de um sistema de comunicação para essas aplicações, além das vantagens e desvantagens inerentes a cada uma.

Portanto, ao apresentar as possíveis técnicas utilizadas na superação dos desafios relacionados especificamente a este trabalho, deixa claro que a maior dificuldade a ser enfrentada será a escolha do mecanismo de comunicação, pois este influencia diretamente na interatividade, na escalabilidade e na consistência da aplicação de visualização distribuída.

5. A Arquitetura OpenReality

A arquitetura OpenReality[1] foi projetada para dar suporte à construção de Ambientes de Visualização Distribuída heterogêneos, nos quais o nível de controle sobre o ambiente pode variar, deste o mais detalhado, ou de granularidade fina, como o controle direto do Grafo de Cena compartilhado, até uma granularidade mais grossa, como a definição e controle de objetos compartilhados. Esta arquitetura utiliza uma estrutura de comunicação ponto a ponto e oferece várias modalidades de distribuição de informações, que podem variar, deste a regeneração freqüente dos estados dos objetos, até a replicação garantida do Grafo de Cena. Para a construção dos ambientes, ela oferece ainda objetos 3D primitivos, que podem ser utilizados e especializados pelas aplicações.

O parágrafo anterior é uma cópia de um dos trechos que apresentaram a proposta OpenReality no início de seu projeto. Projetada em três camadas distintas e independentes, a arquitetura OR foi e continua sendo fonte de pesquisa de três projetos-piloto, cada um deles cuidando de uma das suas camadas. A figura 5.1 apresenta a estrutura organizacional da OR.

As três seções seguintes descrevem sucintamente cada uma das camadas da arquitetura OpenReality e suas funcionalidades.

5.1 A Camada de Suporte aos Ambientes de Visualização Distribuída

A Camada de Suporte aos Ambientes de Visualização Distribuída é o núcleo da arquitetura da OR. Ela provê às aplicações um conjunto de recursos necessários para a criação e manutenção de ambientes. Entre eles, pode-se destacar a síntese de imagens, o gerenciamento dos dispositivos de entrada, o gerenciamento da comunicação e a manutenção da consistência do ambiente entre os participantes.

Cada aplicação que utiliza o *framework* OR possui ligada a si a Camada de Suporte aos Ambientes de Visualização Distribuída. Essa camada é executada no mesmo processo da aplicação e faz uso de *threads* para um maior desempenho e tratamento de eventos

assíncronos. Toda a Interface de Programação de Aplicação (API) da OR é disponibilizada por esta camada.

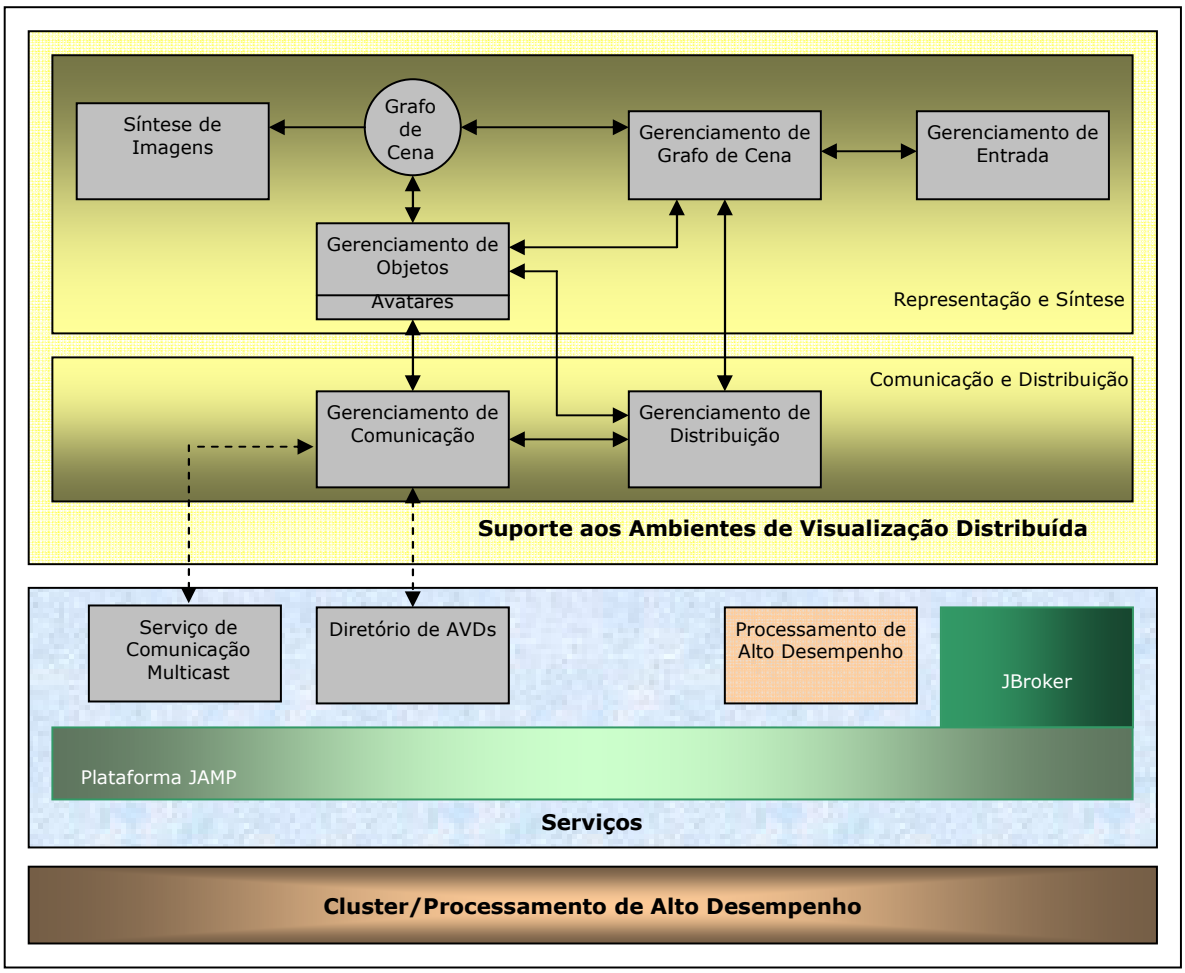

FIGURA 5.1: ORGANIZAÇÃO DAS CAMADAS DA OPENREALITY

A Camada de Suporte aos Ambientes de Visualização Distribuída é subdividida em dois subsistemas: o Subsistema de Representação e Síntese e o Subsistema de Comunicação e Distribuição. Esta subdivisão é apenas lógica, pois ambos operam no mesmo processo e espaço de memória da aplicação.

O Subsistema de Representação e Síntese é o responsável por manter a estrutura de dados que representa o ambiente e sintetizar, em tempo-real, as imagens que correspondem a essa representação. Ele ainda gerencia todos os objetos e entidades que fazem parte do ambiente, trata as informações originárias de dispositivos de entrada e colabora na distribuição do ambiente entre todos os participantes.

O Subsistema de Comunicação e Distribuição é responsável por gerenciar a comunicação entre os participantes e a distribuição de informações. Nesse subsistema, o Gerenciador de Comunicação oferece as primitivas básicas de comunicação para os demais

módulos, como a criação de canais de comunicação *unicast*, criação de grupos, envio e recebimento de pacotes de informação, etc. Ele também oferece um conjunto de protocolos que podem ser utilizados na comunicação, como TCP, UDP, IP *Broadcasting* utilizando UDP e IP *Multicasting*. O Gerenciador de Distribuição, por sua vez, implementa os diversos mecanismos de distribuição oferecidos pela OR. Ele permite um controle fino da distribuição, variando desde a replicação garantida das informações, utilizando um algoritmo de trava distribuído, até o controle da distribuição, através da regeneração freqüente de estado. As questões específicas do projeto do Gerenciador de Distribuição são tratadas no capítulo 7.

Por ser a modalidade que possui os maiores benefícios em relação à utilização de largura de banda e processamento, a comunicação *multicast* é geralmente adotada pelo Gerenciador de Comunicação como a modalidade padrão de comunicação. Em redes que não suportam essa modalidade de comunicação, a escolha fica a cargo do desenvolvedor, que pode escolher, dentre os protocolos de comunicação oferecidos pelo *framework* OR, os que melhor atendem os requisitos necessários à sua aplicação.

5.2 A Camada de Serviços

A Camada de Serviços é responsável por oferecer uma série de serviços à Camada de Suporte aos Ambientes de Visualização Distribuída. Inicialmente, os serviços registrados são o Diretório de Ambientes de Visualização Distribuída, o serviço de Comunicação *Multicast* e o serviço de alocação e execução de processamento de alto desempenho. Esses serviços são disponibilizados através de um *broker* (Java Broker), que faz parte da Plataforma JAMP[33], desenvolvida pelo Grupo de Sistemas Distribuídos e Redes do PPG-CC da UFSCar. Essa plataforma utiliza Java RMI para a sua comunicação. Suas estruturas são apresentadas no capítulo 10, assim como outras considerações a seu respeito.

O serviço de Diretório de Ambientes de Visualização Distribuída permite o registro de novos ambientes e a localização de ambientes que estão em andamento, incluindo seus participantes. Ele verifica, periodicamente, se os participantes ainda estão em funcionamento, através do envio regular de pacotes de checagem. Caso algum participante não esteja mais em operação, ele é removido da lista de participantes do ambiente. Se o último participante de um ambiente deixá-lo, este é removido do diretório. Geralmente, o serviço de Diretório de Ambientes de Visualização Distribuída é utilizado pelas aplicações OR somente nas suas fases de iniciação e encerramento. Os capítulos 9 e 11 tratam especificamente sobre suas funcionalidade e integração entre a JAMP e a OR.

O serviço de Comunicação *Multicast* oferece as funcionalidades de gerenciamento de grupos *multicast* para o Gerenciador de Comunicação. Detalhes sobre o Gerenciador de Comunicação e o serviço de Comunicação *Multicast* da Plataforma JAMP são apresentados no capítulo 8.

Por fim, o serviço de alocação e execução de processamento de alto desempenho permite alocar processamento junto à Camada de Processamento de Alto Desempenho e ordenar a execução de processos, repassando possíveis resultados para a aplicação OR.

5.3 A Camada de Processamento de Alto Desempenho

A Camada de Processamento de Alto Desempenho disponibiliza uma grade de processadores que pode ser alocada e utilizada para tarefas computacionalmente complexas, como simulações, criação de terreno em tempo-real, otimização de cenários e modelos, etc. Esses serviços são registrados junto à Camada de Serviços e disponibilizados para as aplicações OR.

O pacote Mosix[34] foi inicialmente escolhido para implementação dessa camada. Ele utiliza algoritmos adaptativos de gerenciamento que monitoram a distribuição de recursos entre os nós. Após a criação de um novo processo, o Mosix tenta atribuí-lo para o melhor nó disponível. Ele opera silenciosamente e suas operações são transparentes para as aplicações, isto é, os processos são executados em paralelo, simulando um Multiprocessador Simétrico (SMP). Esse pacote possui as vantagens de estar disponível para os sistemas operacionais Linux e ser gratuito, além de estar sob a licença pública de código aberto (GNU GPL). Vale ressaltar que os estudos e o desenvolvimento desta camada não fazem parte deste trabalho.

5.4 O Ciclo de Vida da Aplicação OR

Toda aplicação que utiliza o *framework* OpenReality possui a capacidade de criar um Ambiente de Visualização Distribuída ou conectar-se a um já em operação. O processo de criação de um Ambiente de Visualização Distribuída envolve o seguinte conjunto de ações que devem ser executadas pela aplicação OR:

- ⇨ Criação de um novo grafo de cena que represente o ambiente desejado;
- ⇨ Criação de novos objetos ou utilização de objetos pré-definidos disponibilizados pela API da OR, como *Avatares*, por exemplo;

- ⇨ Registro do ambiente junto ao serviço de diretório de ambientes;
- ⇨ Espera por conexões de outros participantes;
- ⇨ Interação com os outros participantes através dos protocolos de distribuição disponibilizados pela OR.

Para a conexão com ambientes já em operação, a aplicação OR deve executar as seguintes ações:

- ⇨ Busca no diretório de Ambientes de Visualização Distribuída pela referência a um ambiente de interesse, seguido do registro da aplicação como participante do mesmo;
- ⇨ Verificação do tempo de resposta de alguns participantes desse ambiente e escolha do participante com o melhor tempo;
- ⇨ Registro nos canais de comunicação utilizados pelo ambiente e armazenamento das notificações de mudanças do ambiente;
- ⇨ Requisição e transferência de uma cópia carbono do grafo de cena, informações de geometria, textura e objetos dinâmicos definidos pela aplicação;
- ⇨ Aplicação das mudanças armazenadas que ocorreram no ambiente desde o início da transferência das informações;
- ⇨ Interação com os outros participantes, através dos protocolos de distribuição disponibilizados pela OR.

Antes de iniciar a transferência da cópia carbono das informações, a aplicação OR registra-se nos canais de comunicação utilizados pelo ambiente e armazena todas as notificações de mudanças ocorridas durante a transferência das informações. Isto é necessário para manter a consistência do ambiente, pois a transferência das informações pode ser um processo demorado.

Os Ambientes de Visualização Distribuída em andamento irão continuar em operação até que seus últimos participantes deixem de utilizá-los. Nesse momento, eles serão extintos e suas referências serão retiradas do serviço de Diretório de Ambientes.

Como toda aplicação que sintetiza imagens em tempo-real, a aplicação OR está sempre executando um ciclo, no qual são computadas as mudanças nas estruturas de dados que definem o ambiente e sintetizadas novas imagens. A figura 5.2 ilustra o ciclo de execução de uma aplicação OR, no qual alguns subsistemas da Camada de Suporte aos Ambientes de Visualização Distribuída estão sendo executados concorrentemente através do uso de *threads*.

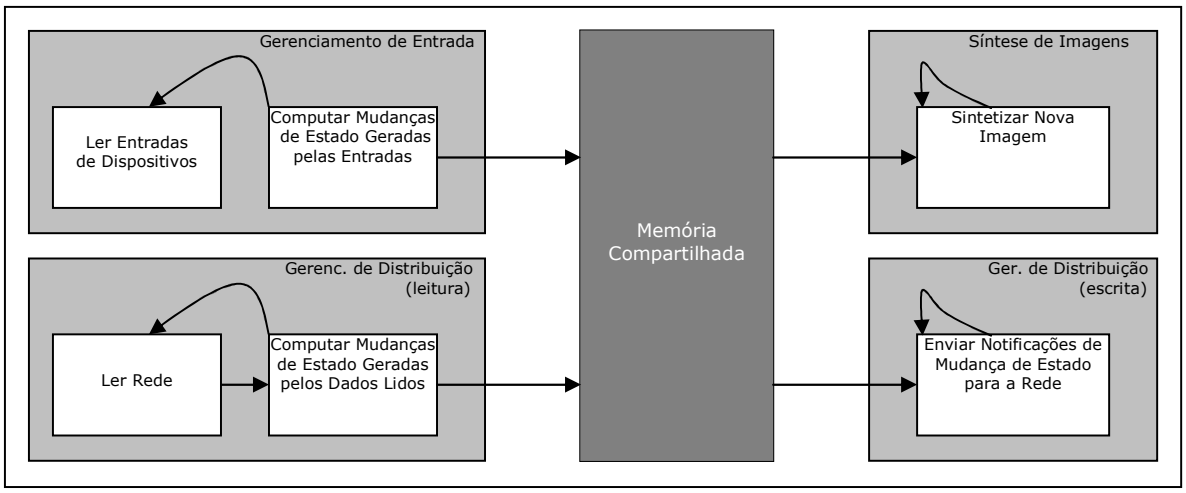

FIGURA 5.2: CILCLO DE EXECUÇÃO DA APLICAÇÃO OR

5.5 Situação Atual da Arquitetura OpenReality

O Subsistema de Representação e Síntese, presente na Camada de Suporte aos Ambientes de Visualização Distribuída, deu origem ao primeiro dos trabalhos realizados sobre a OR. Nesse trabalho, Bellezi[35] tornou possível a criação de aplicações de visualização não distribuída para os mais diversos fins.

O Subsistema de Comunicação e Distribuição, juntamente com a Camada de Serviços, é o escopo do presente trabalho.

Por fim, a Camada de Clusters e Processamento de Alto Desempenho, impulsionou o terceiro trabalho, que deve tratar dos modelos de associação de aglomerados de computadores para oferecer processamento de alto desempenho à arquitetura OR. O desenvolvimento dessa camada é tratado em outra dissertação.

5.6 Considerações Finais

Este capítulo apresentou a proposta da arquitetura OpenReality e seus objetivos. Entre as estruturas da OR apresentadas, encontram-se o Subsistema de Comunicação e Distribuição e a Camada de Serviços, cujos projetos constituem o objetivo deste trabalho.

Os capítulos apresentados até o momento ofereceram o suporte necessário para a realização dos projetos que fizeram parte deste trabalho. A divisão dos capítulos que seguem foi guiada pelas questões levantadas durante seus desenvolvimentos. Os próximos quatro capítulos tratam das estruturas diretamente envolvidas com a arquitetura OR, referentes ao

Subsistema de Comunicação e Distribuição. Os capítulos restantes discutem e apresentam o projeto das estruturas da Camada de Serviços, utilizando os recursos oferecidos pela Plataforma JAMP.

As primeiras perguntas que surgiram referentes ao que seria distribuído entre as aplicações e o porquê da distribuição são respondidas no capítulo 6, que trata da modificação da estrutura do grafo de cena.

O capítulo 7, por sua vez, trata de apresentar como as informações que devem ser distribuídas para atender os requisitos apresentados no capítulo 6. Como a proposta da arquitetura OR é oferecer flexibilidade ao desenvolvedor durante a construção de aplicações de visualização distribuída, a forma como essa distribuição ocorre foi dividida em duas estruturas. A primeira, o Gerenciador de Distribuição, é tratada nesse mesmo capítulo, enquanto a segunda, o Gerenciador de Comunicação, é apresentada no capítulo 8.

O capítulo 9 resolve o problema da organização e localização de ambientes. Ele apresenta o Diretório de Ambientes de Visualização Distribuída, encerrando o projeto do Subsistema de Comunicação e Distribuição da OR.

O capítulo 10, que é o primeiro referente ao projeto da Camada de Serviços, formaliza a Plataforma JAMP, que é utilizada para oferecer todos os serviços desenvolvidos.

O capítulo 11 apresenta o projeto de um novo *framework* para compor a Plataforma JAMP, o *framework* para a criação de Sistemas de Diretórios.

Por fim, o capítulo 12 descreve o projeto do compilador JAMP2C, criado para atender às necessidades exigidas pelos mecanismos de comunicação adotados.

6. O Grafo de Cena

Antes de iniciar o projeto de distribuição das aplicações OR, é necessário que três questões possuam respostas bem definidas: o que deve ser distribuído, o porquê da distribuição e como esta será realizada. Este capítulo apresenta as respostas encontradas, no início deste projeto, para as perguntas acima.

6.1 O que deve ser distribuído e por quê?

Analisando-se uma aplicação não distribuída OR, pode-se dividi-la nas estruturas que compõem o subsistema de Representação e Síntese da Camada de Suporte aos AVDs. Como já foram apresentados no capítulo 5, os eventos captados pelo Gerenciador de Entradas, após serem processados, modificam as propriedades do grafo de cena. Também foi visto que o Sintetizador de Imagens utiliza as informações do grafo de cena, para gerar as imagens tridimensionais que são exibidas ao usuário. Assim, fica claro perceber que o grafo de cena é a estrutura detentora de todas as informações que representam o ambiente. Portanto, pode-se afirmar que o grafo de cena é a estrutura que deve ser distribuída entre as aplicações para que seja possível a visualização distribuída.

Porém, apenas a distribuição do grafo de cena, entre as várias aplicações que desejam participar de uma sessão de visualização distribuída, não oferece recursos para que os participantes interajam e, principalmente, continuem visualizando o mesmo ambiente. Após a realização da distribuição das informações do grafo de cena, cada aplicação passa a controlar suas próprias cópias locais do grafo independentemente. Por maior que seja o esforço em manter idênticas as informações, o ambiente não oferece a interatividade necessária, pois, apenas com a distribuição das informações do grafo de cena, o que se alcança é que todas as aplicações possuam uma cópia da visão de um dos participantes. Para que os participantes possam experimentar as sensações de espaço, presença e tempo compartilhados, é necessário que haja interação entre seus participantes.

Pensando-se somente no grafo de cena, que é o detentor de toda a representação do ambiente, uma maneira simples de prover a interação é permitir que ele seja alterado por vários usuários e, somente a versão que contenha todas as alterações realizadas seja distribuída. Para isso, seria necessário que todos os participantes executassem as mesmas alterações, cada um em seu grafo local. Logo, outro requisito para a distribuição da visualização é que as aplicações sejam capazes de sincronizar seus grafos de cena, mantendo-os idênticos durante toda a experiência de visualização.

Assim, de acordo com os dois requisitos apresentados, o *framework* OR deve oferecer meios de distribuir o grafo de cena e as suas modificações entre as aplicações dos participantes.

6.2 Como distribuir?

Encontrados os itens que devem ser distribuídos, esta seção passa a apresentar as respostas de como realizar a distribuição para cada um deles. Tanto a apresentação da forma de distribuição do grafo de cena, quanto das modificações nele realizadas acompanham trechos de códigos na linguagem C++, na tentativa de esclarecer como se realizam as trocas de dados entre os participantes. Vale lembrar que as estruturas que serão apresentadas estão presentes nas aplicações de todos os participantes.

6.2.1 A distribuição do Grafo de Cena

Como toda aplicação OR é capaz de criar um grafo de cena, elas também devem passar a ser capazes de exportar e importar seus grafos de cena. Essas operações dar-se-ão através da rede de comunicação, requisitando que o grafo de cena seja enviado através dela.

Sendo assim, pode-se implementar a funcionalidade da transmissão do grafo de cena via rede através de métodos de ordenação das informações presentes nos grafos de cena. Esse recurso de programação é suportado pela maioria das linguagens de programação, incluindo C++, que é a linguagem utilizada na implementação das classes da Camada de Suporte aos AVDs da OR.

Dessa forma, todas as classes pertencentes ao grafo de cena da arquitetura OpenReality devem implementar a interface `OR::Serializable`. Nela são implementadas as redefinições dos operadores de inserção (**<<**) e remoção (**>>**) da linguagem e definidos dois métodos abstratos, que cada classe realizadora dessa interface deve implementá-los. A interface `OR::Serializable` é exibida na figura 6.1.

```
class OR_EXPORT Serializable {
  public:
    virtual DataOStream& serialize( DataOStream& os )
                                                  throw( DataStreamException ) = 0;
    virtual DataIStream& unserialize( DataIStream& is )
                                                  throw( DataStreamException ) = 0;
};

inline DataIStream& operator >>( DataIStream& src, Serializable& object )
                                                    throw ( DataStreamException ) {
  return object.unserialize( src );
}

inline DataOStream& operator <<( DataOStream& src, Serializable& object )
                                                    throw ( DataStreamException ) {
  return object.serialize( src );
}
```

FIGURA 6.1: A INTERFACE OR::SERIALIZABLE

Portanto, qualquer objeto que implementar os métodos virtuais, declarados na interface apresentada, pode ser enviado via rede através de uma *stream* de dados. As classes `DataIStream` e `DataOStream` utilizadas no código acima representam as *streams* de entrada e saída oferecidas pelas implementações dos protocolos de transporte presentes na biblioteca de classes da própria linguagem C++.

6.2.2 A distribuição das Modificações do Grafo de Cena

Além de permitir que o grafo de cena seja distribuído via rede, o segundo requisito a ser satisfeito é a disponibilização de um mecanismo que o mantenha idêntico em todas as aplicações participantes da sessão em que ele existe. Como as informações que compõem o grafo de cena são armazenadas apenas em seus nós, a consistência desejada pode ser mantida a partir do momento em que os nós também se mantenham idênticos.

Cada nó do grafo de cena é representado por algum dos objetos, disponíveis entre as classes do *framework* OR, que o representam. Esses objetos oferecem métodos para a manipulação e a alteração dos dados que armazenam. Logo, a alteração do grafo de cena é realizada através da execução dos métodos de alteração dos objetos que o compõem.

Portanto, para que os grafos de cena locais de cada aplicação sejam mantidos idênticos, os dados armazenados em seus objetos nó devem ser manipulados de maneira idêntica. Nas aplicações que desejam experimentar a visualização distribuída, isto significa que a execução de um método em um dos objetos do grafo de cena de um participante deve disparar a execução desse mesmo método do mesmo objeto presente no grafo de cena das aplicações dos demais participantes.

Diante disso, foi desenvolvido um mecanismo de notificação de execução de métodos para atender a necessidade da consistência. Esse mecanismo foi implementado em duas partes, a que altera os objetos do grafo de cena e a que cuida da distribuição da notificação. A primeira dessas partes é apresentada na próxima seção, enquanto a segunda será discutida no próximo capítulo.

6.3 O Grafo de Cena Distribuído

Esta seção apresenta a forma adotada para a distribuição do grafo de cena entre as aplicações de visualização distribuída. Utilizando alguns dos recursos de modelagem e programação orientada a objetos, pôde-se criar uma estrutura, baseada no grafo de cena já existente na arquitetura OR, que permitiu o compartilhamento e a manutenção da consistência, requisitados pelas aplicações de visualização distribuída. Essa nova estruturação do grafo de cena será denotada por grafo de cena distribuído, enquanto a estrutura do grafo já existente na OR, apenas por grafo de cena.

Antes que alguma reestruturação dos objetos do grafo de cena ocorra, deve-se ter em mente que a arquitetura OpenReality deve continuar suportando as aplicações não distribuídas. Portanto, as alterações necessárias não devem ser realizadas nas classes existentes dos objetos que formam o grafo de cena. Por outro lado, as aplicações distribuídas também realizam as mesmas operações que são realizadas pelas não distribuídas, apenas acrescentando a funcionalidade da notificação da execução dos métodos localizados nos grafos de cena distribuídos.

Essa notificação é gerada pela aplicação que possui o grafo de cena no exato momento em que ele é alterado. As demais aplicações que compartilham o grafo de cena, ao receberem as notificações de atualização, devem ser capazes de identificar qual dos objetos do grafo de cena deve ser atualizado. O objeto, por sua vez, deve saber identificar qual de seus métodos deve ser executado e com quais parâmetros.

Outro requisito para a manutenção da consistência é que todos os objetos dos grafos de cena das aplicações respeitem a mesma ordem de ocorrência das execuções. Qualquer desrespeito a essa ordem pode causar discrepâncias nas sínteses de imagens locais a cada aplicação, tornando o ambiente inconsistente.

Diante disso, foram generalizados os objetos do grafo de cena que são distribuídos em objetos que, além de possuírem os atributos e métodos iguais aos dos objetos não distribuídos, devem possuir um identificador único, informações sobre o estado de atualização em que se

encontram e saber reconhecer cada um dos seus métodos, através de informações de identificação. Na prática, podemos declarar a superclasse dos objetos do grafo de cena distribuído como é mostrado na figura 6.2.

```
typedef void (*Method)();

class DistributedSgObject : public Serializable {
  protected:
    int stampID;
    int objectID;
    Method** callMethod;
    DistributionManager* dm;

  public:
    virtual DistributedSgObject();
    virtual ~DistributedSgObject();
    int update( int stamp, int method, Params p );
    void setObjectID( int oid );
    int getObjectID();
    void addMethod( Method* m, int id );
    void setDistributionManager( DistributionManager* dm );
};

DistributedSgObject :: DistributedSgObject( int totalMethods ) {
  stampID = 0;
  callMethod = new Method*[ totalMethods ];
}

DistributedSgObject :: ~DistributedSgObject() {
  delete callMethod;
}

int DistributedSgObject :: update( int stamp, int method, Params p ) {
  if( stamp == stampID + 1 ) {
    callMethod[ method ]( p );
    return 1;
  }
  return 0;
}

void DistributedSgObject :: addMethod( Method* m, int id ) {
  callMethod[ id ] = m;
}
```

FIGURA 6.2: A CLASSE OR::DISTRIBUTEDSGOBJECT

O atributo dm, da classe **DistributionManager**, implementa as funcionalidades do Gerenciador de Distribuição e será tratado no capítulo seguinte. Por ora, assume-se que execute o papel de notificador de execuções de métodos. Para informar os demais participantes da execução, é executado seu método **notify(...)**. Já, para que a ocorrência de execuções de métodos executados no grafo de cena de outros participantes seja percebida localmente, o Gerenciador de Distribuição executa o método **update(...)** dos objetos do grafo de cena distribuído.

Generalizados os objetos que compõem o grafo de cena distribuído, pode-se passar a tratar das classes que o especializam, isto é, as classes dos objetos do grafo de cena distribuído. Como mencionado anteriormente, os objetos dessas classes, além de realizarem a herança de **DistributedSgObject**, devem implementar a interface **Serializable** e possuir as mesmas funcionalidades dos objetos que não pertencem ao grafo de cena distribuído. A figura 6.3 apresenta como devem ser os objetos do grafo de cena distribuído.

```
class DistributedSgXXX : public DistributedSgObject, SgXXX {

  /* Para facilitar a compreensão, esta classe não está separada em */
  /* .h (definição) e .cpp (implementação).                         */

  /* Redefine apenas os métodos que manipulam o grafo de cena.      */
  /* Os demais, usa da classe pai SgXXX, pois são públicos.         */

  #define TOTAL_METHODS 3
  #define METHODMMM_ID  0
  ...

  public:
    void methodMMM( Params p ) {
      if( dm -> notify( objectID, METHODMMM_ID, stampID, p ) )
        super.methodMMM( p );
    }

    DistrbutedSgXXX() : DistributedSgObject( TOTAL_METHODS ) {
      /* Adiciona dinamicamente cada um dos métodos na classe pai */
      addMethod( ( Method* ) methodMMM, METHODMMM_ID );
      ...
    }

    DataOStream& serialize( DataOStream& os ) throw( DataStreamException );

    DataIStream& unserialize( DataIStream& is ) throw( DataStreamException );

};
```

FIGURA 6.3: O MODELO DE DISTRIBUIÇÃO PARA AS CLASSES DO GRAFO DE CENA

Pode-se perceber, a partir do trecho de código anterior, que cada método possui um identificador, que é utilizado pelos métodos **notify(...)** e **update(...)**. Além desse, outros dois atributos sempre utilizados são **objectID** e **stampID**. Vale notar que ambos os métodos mencionados possuem, como valor de retorno, valores lógicos. Isso serve como forma de controle da consistência, que será tratada nos próximos capítulos. Outra característica notável é que a complexidade do processamento da decisão de qual método deve ser chamado, a partir de uma execução do **update(...)**, foi reduzida sensivelmente com a utilização de um vetor de métodos, presente na classe DistributedSgObject.

6.4 Considerações Finais

Neste capítulo, foram apresentadas quais informações devem ser distribuídas para que seja possível a visualização distribuída entre as aplicações, definindo a forma como o grafo de cena deve ser distribuído entre as aplicações e como deve ocorrer a manutenção de sua consistência. Foi também apresentada a existência da estrutura DistributionManager, responsável pelo tratamento das notificações de execução de métodos, que passará a ser apresentada no capítulo seguinte.

Por fim, a modelagem adotada para a criação do grafo de cena distribuído torna seu tratamento transparente para o desenvolvedor, que pode, inclusive, compor modelos híbridos que utilizam objetos padrões e distribuídos para compor seus grafos de cena.

7. O Gerenciador de Distribuição

O Gerenciador de Distribuição, como mencionado no capítulo anterior, realiza o papel de notificador da ocorrência dos métodos entre os participantes de uma sessão de visualização distribuída. Este capítulo apresenta seu projeto e como se dão as trocas de notificações das execuções dos métodos nos grafos de cena de outros participantes. Além disso, apresenta também outras duas tarefas não citadas, a cópia do grafo de cena via rede (exportação e importação) e a de um determinado objeto desse grafo.

7.1 O Processo de Importação e Exportação

Inicialmente, quando uma aplicação deseja criar um ambiente de visualização distribuída, ela deve tratar de várias questões além da distribuição do grafo de cena e das notificações de execução de métodos. Questões como o registro e localização do ambiente, os protocolos a serem adotados, o grau de consistência do ambiente, entre outros, são tão importantes quanto a própria distribuição. Por ora, assume-se que essas tarefas sejam realizadas por um gerenciador de comunicação, que se relaciona e oferece tais serviços para o Gerenciador de Distribuição. Esse gerenciador de comunicação será detalhado no capítulo 8.

Deixando de lado as questões acima, o Gerenciador de Distribuição preocupa-se inicialmente em oferecer o compartilhamento do grafo de cena entre as aplicações que participam da sessão de visualização distribuída.

Como o primeiro passo de uma aplicação é a construção de seu grafo de cena, as aplicações instanciam um objeto Gerenciador de Distribuição e, logo em seguida, pedem pela obtenção de uma cópia do objeto raiz do grafo de cena do ambiente. Caso a resposta seja nula, indicando que a aplicação é a primeira participante do ambiente desejado, essa passa a criar seu próprio grafo de cena. Após esse processo de criação, a aplicação informa ao Gerenciador de Distribuição a existência de um grafo de cena, passando-lhe a referência do objeto raiz do grafo criado. Assim, futuras aplicações que desejem ingressar no ambiente criado obterão

sucesso ao solicitar uma referência ao grafo de cena do ambiente. Os pedidos por essa referência serão enviados pela rede de dados e recebidos pelo Gerenciador de Comunicação, que os repassa ao Gerenciador de Distribuição. A aplicação que construiu o grafo de cena fará o papel de exportadora, enquanto as demais aplicações que ingressam no ambiente, o papel de importadoras do grafo de cena. Vale notar que, após a obtenção de um grafo de cena, seja importando-o ou criando-o localmente, toda aplicação pode exercer o papel de exportadora.

Como já foi dito, o processo de transferência dos dados do grafo de cena via rede de dados é facilitado com a utilização dos recursos de programação próprios da linguagem utilizada, como a manipulação de *streams* de dados. Porém, embora seja um processo relativamente simples, alguns problemas devem ser resolvidos para que todas as aplicações possam interagir.

O mais grave deles é, após o término da transferência dos dados do grafo de cena, a aplicação importadora deve saber como manipulá-lo. A aplicação criadora do ambiente, isto é, a que criou o grafo de cena originalmente, possui todas as referências necessárias para os objetos de controle do grafo de cena, os objetos de transformação. Por outro lado, as aplicações que apenas importaram o grafo de cena possuem apenas a referência para a sua raiz.

A solução adotada para esse problema é a utilização de nomes que definem cada um dos objetos que compõem o grafo de cena. Assim, a aplicação criadora do grafo original deve informar o nome de cada um deles no momento de suas instanciações. Já as aplicações que importam esse grafo, podem utilizar esses nomes para conseguir as referências aos objetos de controle, passando a saber como controlá-los também. Um objeto responsável pela rotação de um modelo tridimensional existente no ambiente pode receber, por exemplo, o nome de "Rotaciona_Modelo_3D". Dessa forma, uma aplicação que acabou de importar o grafo de cena desse ambiente pode passar a manipular a rotação desse modelo a partir do objeto referenciado pelo nome acima.

7.2 O Processo de Notificação de Execução

Após a obtenção do grafo de cena do ambiente, bem como o poder de manipulá-lo, a aplicação pode passar a executar os métodos desses objetos de controle da maneira que desejar. O Gerenciador de Distribuição deve reconhecer essas execuções e distinguir, para

cada uma delas, qual objeto, método, parâmetros e a ordem de ocorrência. De posse de todos esses dados, o Gerenciador de Distribuição pode notificar a ocorrência de cada alteração no grafo de cena às demais aplicações participantes. A percepção das execuções e a ordem de notificação são responsabilidades do Gerenciador de Distribuição, enquanto o envio das informações de notificação e a forma como este ocorre são tarefas do Gerenciador de Comunicação.

Como apresentado no capítulo anterior, os objetos do grafo de cena distribuído, antes de executarem seus métodos localmente, informam ao Gerenciador de Distribuição a tentativa de execução de um determinado método. Caso o Gerenciador de Distribuição aprove a execução, esta ocorre localmente. Os fatores que influenciam na permissão ou negação de execução de um método estão diretamente relacionados com o protocolo de consistência adotado. Objetos desatualizados no tempo, travas de alterações não conquistadas e outras situações que não permitem alterações no grafo de cena fazem parte de alguns desses fatores. Todos eles serão apresentados e discutidos durante o capítulo 8, que apresenta o protocolo de consistência adotado.

7.3 A Classe DistributionManager

Na tentativa de esclarecer melhor quais são os métodos que podem ser chamados pelas estruturas envolvidas e, principalmente, a forma como suas tarefas são realizadas, a figura 7.1 apresenta um trecho de código da classe DistributionManager. Nesse trecho, os atributos da classe, juntamente com os métodos existentes, tornam possível a realização das devidas tarefas.

Entre as funções apresentadas, encontram-se a geração dos identificadores únicos para cada objeto do grafo de cena, o envio e o recebimento de notificações de execução de métodos, requisições pelo grafo de cena e por objetos determinados que nele se encontrem. Vale notar que, embora essa classe ofereça todo suporte para a distribuição das informações apresentadas como requisitos para as aplicações de visualização distribuída, apenas é definida a forma como esta ocorre. Todas as trocas de dados e a execução do protocolo de consistência propriamente dito ficam a cargo do Gerenciador de Comunicação, que será apresentado no próximo capítulo.

```
class DistributionManager {

  private:
    int objectIDGenerator;
    DistributedSgObject** crossReference;
    CommunicationManager* cm;
    DistributedSgObject* root;
    char** nodeNames;
    ...

  public:
    void genObjectID( DistributedSgObject* dobj ) {
      dobj -> setObjectID( ++objectIDGenerator );
      crossReference[ objectIDGenerator ] = dobj;
      nodeNames[ objectIDGenerator ] = dobj -> getName();
    }

    DistrbutionManager( CommunicationManager* cm ) {
      objectIDGenerator = -1;
      this -> cm = cm;
      crossReference = new DistributedSgObject*[ MAX_OBJECTS ];
      nodeNames = new char*[ MAX_OBJECTS ];
      ...
    }

    int update( int objectID, int methodID, int stamp, Params p ) {
      return ( crossReference[ objectID ] -> update( methodID, stamp, p ) );
    }

    int notify( int objectID, int methodID, int stampID, Params p ) {
      return ( cm -> notify( objectID, methodID, stampID, p ) );
    }

    void setRoot( DistributedSgObject* dobj );
    DistributedSgObject* getRoot();
    DistributedSgObject* getNode( int objected );
    DistributedSgObject* getNodeByName( char* name );
    ...

};
```

FIGURA 7.1: O GERENCIADOR DE DISTRIBUIÇÃO

7.4 Considerações Finais

Este capítulo apresentou, de forma sucinta, os recursos que as aplicações de visualização distribuída possuem para distribuir as informações necessárias. De uma maneira geral, esse capítulo e o anterior definem o que deve ser distribuído e em que circunstâncias, deixando, para o capítulo seguinte, o tratamento de como ocorre a distribuição.

8. O Gerenciador de Comunicação

O Gerenciador de Comunicação é uma das estruturas centrais do Subsistema de Comunicação e Distribuição. É o responsável pela transmissão propriamente dita das informações e pela forma como ela ocorre. Seu papel na arquitetura é servir as mais diversas modalidades de comunicação para as aplicações. Na prática, o Gerenciador de Comunicação é o executor do mecanismo de consistência da arquitetura. Para realizar a comunicação propriamente dita, ele utiliza os serviços de transmissão de dados oferecidos pelos protocolos da camada de transporte do modelo TCP/IP[27].

Na maioria das vezes, esse gerenciador é utilizado pelo Gerenciador de Distribuição, oferecendo canais de comunicação para a manutenção da consistência das informações das aplicações no ambiente. Eventualmente, objetos que necessitem transmitir dados que não fazem parte de seus grafos de cena podem requisitar os seus serviços.

Segundo Singhal e Zyda[7], a escolha do mecanismo de comunicação interfere diretamente na escalabilidade, interatividade e consistência do ambiente. Como um dos objetivos do *framework* OR é proporcionar flexibilidade para a construção das mais diversas aplicações de Ambientes de Visualização Distribuída, esse gerenciador provê alguns dos protocolos adequados para cada situação.

Com o papel de executor de todo tipo de comunicação, esse Gerenciador possui mecanismos para ajudar na manutenção da consistência dos ambientes, nos casos de falha dos clientes. Além disso, otimizações na comunicação que buscam economizar processamento e largura de banda podem estar presentes.

Por fim, além da execução dos protocolos de rede, o Gerenciador de Comunicação é o responsável pela comunicação com a Camada de Serviços. Sempre que uma aplicação necessitar de algum dos serviços que esta camada oferece, toda a negociação e a troca de dados serão realizadas através dele, como ocorre na utilização do serviço de Diretório de Ambientes de Visualização Distribuída, por exemplo.

Antes de iniciar a apresentação das classes que implementam os protocolos disponibilizados para uso no *framework* OR, será apresentada a estratégia de comunicação

entre as linguagens Java e C++, visto que algumas das estruturas da Camada de Suporte aos Ambientes de Visualização Distribuída fazem uso de serviços presentes na Plataforma JAMP.

8.1 Comunicação entre Java e C++

Para realizar a comunicação entre as linguagens Java e C++, existem várias soluções existentes, que partem, desde a implementação completa da comunicação via sockets, até a utilização de ferramentas e padrões de *middleware*.

Entre as possíveis soluções analisadas, foram descartadas a utilização do *Distributed Component Object Model* (DCOM)[36] e do *Remote Object Model over Internet Inter-ORB Protocol* (RMI-IIOP)[37].

O DCOM não se enquadra nas propostas da arquitetura OR, pois, embora sua especificação afirme que seu modelo possui implementações para todas as plataformas de sistemas operacionais, na prática, apenas os ambientes Microsoft o suportam. As implementações desse modelo para ambientes Linux não são oficiais e, em sua totalidade, encontram-se incompletas.

Já o RMI-IIOP seria um protocolo interessante para ser utilizado, porém, a interoperabilidade só é possível entre objetos RMI (presentes na JAMP) e objetos CORBA. Logo, para que fosse possível sua utilização, seria obrigatória a utilização de CORBA nos objetos de comunicação da arquitetura OR. Como ambas as linguagens em questão suportam o modelo CORBA, elas podem interoperar diretamente nesse modelo, sem a necessidade do RMI-IIOP.

Logo, entre as possibilidades estudadas restantes, a comunicação entre as linguagens pode ser implementada utilizando-se CORBA ou implementado a comunicação direta via sockets. Na tentativa de ser definido qual dos mecanismos de comunicação entre as duas linguagens seria utilizado, alguns testes práticos foram realizados. As figuras 8.1 e 8.2 esquematizam a forma como as aplicações de teste foram implementadas.

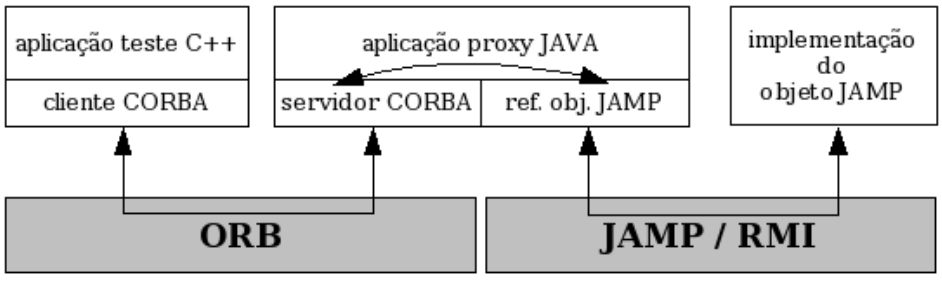

FIGURA 8.1: APLICAÇÃO TESTE USANDO CORBA

Em ambos os testes implementados, a estruturação lógica não foi alterada, apenas foram trocadas as implementações do cliente e do servidor CORBA pelos cliente e servidor socket. O objeto JAMP possui um conjunto de métodos que são chamados pela aplicação teste, escrita em C++. Essa, por sua vez, faz uma bateria de requisições, via seu cliente também implementado na linguagem C++, para o servidor que se encontra junto à aplicação *proxy*, já na linguagem Java. Ao receber uma requisição, a aplicação *proxy* encaminha o pedido para o objeto JAMP através da referência deste. A comunicação que ocorre até o recebimento do valor de retorno do método invocado se dá através do protocolo utilizado pelo mecanismo RMI. Em seguida, a aplicação *proxy* utiliza seu servidor para enviar a resposta de volta ao cliente C++, que a entrega imediatamente à aplicação teste.

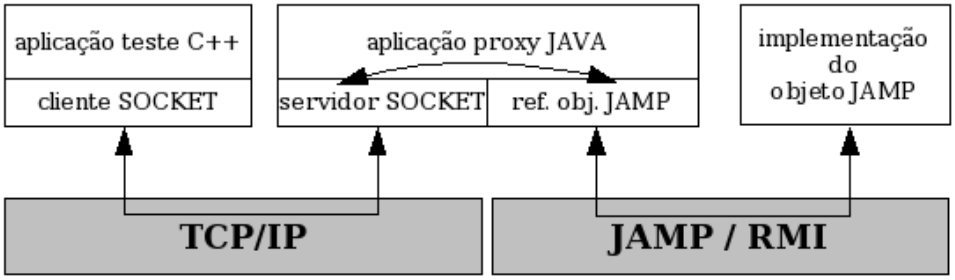

FIGURA 8.2: APLICAÇÃO TESTE USANDO SOCKET

O cliente e o servidor da aplicação teste, que utilizou sockets ao invés de CORBA, realizou a conversão dos dados das chamadas de métodos da seguinte maneira:

- cada método presente no objeto JAMP possuía um identificador numérico único;
- os parâmetros eram dispostos na cadeia de *bytes* enviados, logo em seguida a esse identificador, todos separados por um delimitador .

Assim, para um método `float getBalance( int countId, String password )` de um objeto JAMP, cujo identificador único valha 3, uma possível seqüência de bytes de uma requisição provinda do cliente e recebida pelo servidor é `"3#18880#AbCdEf"`. O servidor, por sua vez, deve converter esses bytes recebidos na chamada de método `objJAMP.getBalance( 18880, "AbCdEf" )`. De maneira análoga, a resposta do servidor para o cliente deve ser a seqüência de bytes `"3#VALOR"`, onde VALOR representa o retorno da chamada do método anterior.

A aplicação que utilizou CORBA obviamente não precisou preocupar-se com tais problemas de implementação.

A bateria de testes que foi realizada cuidou de simular apenas as possíveis situações pelas quais a aplicação real passaria. Nessas situações específicas, a abordagem que utilizou

sockets demonstrou uma sensível diferença de desempenho em relação à que utilizou CORBA. Claramente, os resultados dos testes realizados apenas podem afirmar que, para esses casos específicos e sob as mesmas condições em que foram aplicados os testes, a utilização de sockets é mais recomendada. A figura 8.3 apresenta um gráfico com a relação da quantidade média de informação e o tempo gasto para transferi-la observada durante os testes.

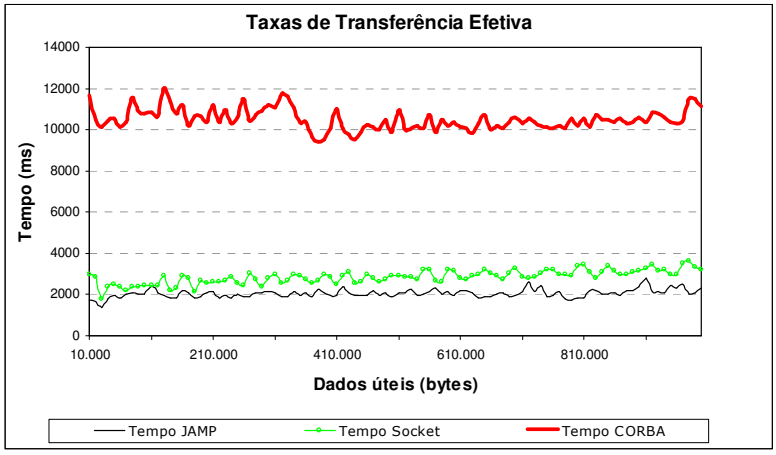

FIGURA 8.3: COMPARAÇÃO DAS TAXAS DE TRANSFERÊNCIAS EFETIVAS DOS TESTES REALIZADOS

Além do desempenho inferior observado durante os testes, a abordagem que utilizou CORBA consumiu mais recursos do sistema operacional, como memória, processamento e utilização da rede.

Sendo assim, diante dos resultados conseguidos experimentalmente, a estratégia que passará a ser utilizada para prover a comunicação entre as estruturas da arquitetura OpenReality (na linguagem C++) e os objetos de serviço presentes na Plataforma JAMP (na linguagem JAVA) fará uso diretamente de sockets e do protocolo de transporte TCP. Da mesma forma como foi apresentada a realização de uma chamada de método utilizando essa estratégia, clientes e servidores deverão concordar no que diz respeito aos valores numéricos de identificação única dos métodos do objeto JAMP. Além disso, deverão realizar as tarefas de ordenação (*marshalling*) e interpretação (*unmarshalling*) das séries de *bytes* que representam os parâmetros antes de enviá-los e recebê-los, respectivamente.

Vale a pena ressaltar que, ao escolher essa estratégia de comunicação, o custo do desenvolvimento das aplicações OR aumenta, pois passa a requisitar um maior número de implementações que o desenvolvedor deve realizar. Diante disso, o capítulo 12 destina-se a apresentar uma ferramenta de auxílio no desenvolvimento dessas aplicações que foi criada para reduzir este custo.

Definida a estratégia de comunicação entre as estruturas em diferentes linguagens de programação, o Gerenciador de Comunicação já se torna capaz de utilizar duas das três estruturas da arquitetura diretamente ligadas a ele, o Diretório de Ambientes de Visualização Distribuída e o Repositório de Travas. A primeira será discutida no próximo capítulo, enquanto o mecanismo de travas é apresentado nas próximas seções. A seguir, passa-se a apresentar as formas de comunicação entre as próprias aplicações OR, que ocorrem diretamente entre os canais de comunicação suportados pelo Gerenciador de Comunicação.

8.2 Comunicação entre as aplicações OR

Responsável pela manipulação dos dados trocados entre as aplicações OR propriamente dita, o Gerenciador de Comunicação deve oferecer as diversas formas de comunicação apresentadas. Além disso, oferece também as mesmas primitivas para a transmissão, independentemente da modalidade de comunicação adotada pela aplicação. Sua implementação busca oferecer uma classe de objetos que trate dos protocolos propostos de forma única e transparente, exportando para a aplicação todas as funcionalidades e características destes. Peculiaridades individuais de cada protocolo não devem ser tratadas pela aplicação, mas sim pela classe de objetos que o implementa. Para isso, é definida a interface OR_Protocol que generaliza todos os protocolos de comunicação inicialmente previstos na arquitetura OpenReality. Dessa forma, como a interface de manipulação dos protocolos disponíveis é única, pode-se dizer que ela também define as primitivas de comunicação utilizadas na arquitetura, que são apresentadas na próxima seção.

8.2.1 As Primitivas de Comunicação

Inicialmente, a arquitetura OpenReality propõe-se a operar em regime de consistência total, não permitindo discrepâncias entre as visões das aplicações como ocorre em algumas das arquiteturas de Ambientes Virtuais Distribuídos já vistas. Sendo assim, todas as aplicações devem possuir cópias idênticas do grafo de cena que representa o ambiente. Dado que toda aplicação é capaz de modificar seu grafo de cena local, qualquer alteração nele realizada deve ser propagada para todas as demais aplicações participantes desse ambiente também a realizarem. Logo, para que esse requisito seja atendido, os canais de comunicação devem oferecer meios para a troca de mensagem entre uma aplicação e todas as demais, tratando de prover uma estrutura totalmente conexa entre os participantes.

Como os protocolos disponibilizados oferecem diferentes formas de difusão de mensagem, *unicast*, *multicast* e *broadcast*, além das simples primitivas de envio e recebimento de dados, a interface OR_Protocol deve oferecer métodos que informam às implementações dos protocolos a situação do grupo de aplicações que compõem o ambiente. Tendo o conhecimento das possíveis situações em que o ambiente pode se encontrar, as implementações dos protocolos realizam os devidos ajustes para continuarem atendendo o requisito apresentado.

Diante disso, as primitivas básicas oferecidas pela interface OR_Protocol são as descritas a seguir.

- `int OR_Protocol :: send( Data );`
- `int OR_Protocol :: receive( &Data );`
- `int OR_Protocol :: prepareToReceive();`
- `int OR_Protocol :: stopReceiving();`
- `int OR_Protocol :: join();`
- `int OR_Protocol :: leave();`
- `Serializable OR_Protocol :: carbonCopy();`
- `Serializable OR_Protocol :: sync( int );`

As primitivas **send** e **receive** oferecem ao gerenciador de Distribuição e aos eventuais objetos da aplicação a realização da troca de informações.

As primitivas **prepareToReceive** e **stopReceiving** são utilizadas pelo Gerenciador de Comunicação para informar às implementações de OR_Protocol que devem, respectivamente, habilitar e desabilitar o recebimento de informações provindas de outras aplicações participantes do ambiente. Quando têm esses métodos chamados, os objetos que implementam OR_Protocol ajustam seus parâmetros e estruturas internas, seja para iniciar ou encerrar o recebimento de dados. Ambos os métodos cuidam do preparo da aplicação local.

As primitivas **join** e **leave** auxiliam neste processo de ajuste dos canais de comunicação, porém das demais aplicações do ambiente. A primeira informa às demais aplicações sobre o ingresso de mais uma participante, enquanto a segunda o seu abandono.

Por fim, as primitiva **carbonCopy** e **sync** requisitam a transferência de cópias de objetos do grafo de cena à alguma das aplicações ativas do ambiente e o recebem. A primeira realiza a cópia do grafo de cena completo, enquanto a segunda, executa a cópia de um objeto específico para realizar a sincronia dos dados. A operação **carbonCopy** é melhor detalhada na seção 8.2.4. Por ora, assume-se apenas que realiza a cópia do grafo de cena.

Assim, pode-se separar as primitivas em duas classes distintas, as de controle interno dos canais de comunicação e as de transmissão dos dados da aplicação. A classe das primitivas de controle contém **prepareToSend**, **stopReceiving**, **join**, e **leave**, pois tratam

apenas da manutenção dos canais de comunicação. A classe das primitivas de transmissão, utilizadas para a transferência dos dados entre as aplicações, é composta por **send**, **receive**, **carbonCopy** e **sync**.

Para que as aplicações possam receber dados e distinguir quais deles são os de controle e quais são os de transmissão, recomenda-se a utilização de um cabeçalho precedendo a *stream* dos dados enviados, embora cada implementação possa tratar esta questão da forma que melhor convier. Nas implementações dos protocolos realizadas durante este trabalho (ver seção 8.3), foram adotadas as unidades de dados (PDUs) para a diferenciação das primitivas de comunicação como mostra a figura 8.4.

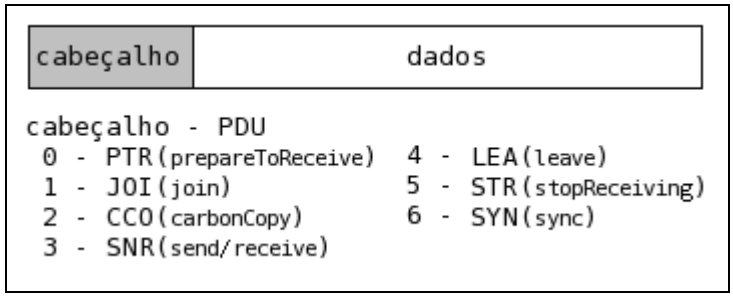

FIGURA 8.4: PDU-OR

Como pode ser observado, as PDU-OR utilizadas possuem apenas dois campos, o cabeçalho e o campo de dados. O cabeçalho possui sempre um dos valores apresentados na figura 8.4. O conteúdo do campo de dados depende da primitiva que está sendo transmitida.

Tendo apresentado as primitivas utilizadas pelos protocolos oferecidos na OR, a seção seguinte descreve as situações em que o Gerenciador de Comunicação as utiliza.

8.2.2 As Situações de Utilização das Primitivas de Comunicação

As situações em que a aplicação OR necessita das primitivas de comunicação geralmente estão ligadas a dois tipos de eventos, os de ingresso e abandono de participantes e os de modificação nos seus grafos de cena. Esta seção descreve brevemente a seqüência de operações que é realizada para cada um desses eventos.

A criação de um novo ambiente de visualização distribuída tem início com a requisição, provinda do Gerenciador de Distribuição, para a aplicação tornar-se participante de um dado ambiente. O Gerenciador de Comunicação realiza uma consulta no Diretório de Ambientes de Visualização Distribuída. Caso o diretório requisitado não exista, está caracterizado o pedido de criação de um novo ambiente. O Gerenciador de Comunicação informa ao Gerenciador de Distribuição a inexistência do diretório requisitado e inicia o

processo de criação deste junto ao Diretório de AVDs. Esse processo é simples e, para o cadastro das informações ser concretizado, necessita apenas de alguns dos parâmetros já conhecidos pelo Gerenciador de Comunicação. Finalizada a criação do novo diretório, o Gerenciador de Comunicação informa à implementação de OR_Protocol para passar a aceitar o recebimento de informações e requisições provenientes dos futuros participantes. A figura 8.4 apresenta o diagrama de seqüência do processo descrito acima.

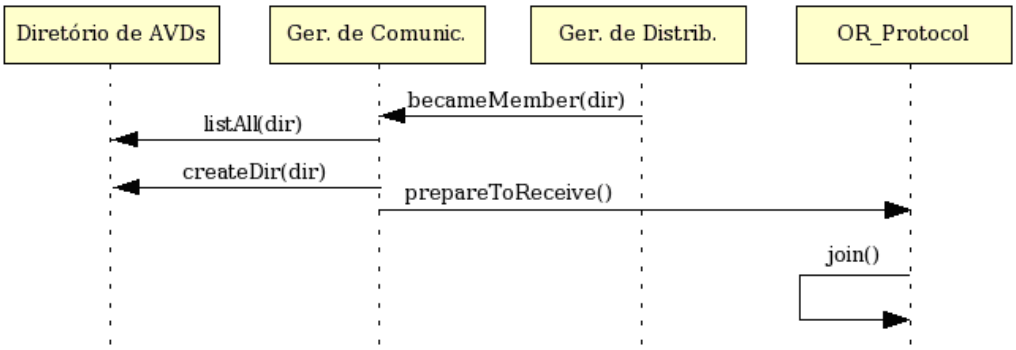

FIGURA 8.5: DIAGRAMA DE SEQÜÊNCIA PARA A CRIAÇÃO DO NOVO AMBIENTE

Depois do recebimento da requisição de participação em um dado ambiente de visualização distribuída, provinda do Gerenciador de Distribuição, o Gerenciador de Comunicação requisita a lista de todos os participantes disponíveis desse ambiente junto ao Diretório de AVDs. Após o recebimento da lista desejada, é escolhido um dos participantes que encontra-se disponível para fornecer uma cópia de seu grafo de cena. Antes de requisitar essa cópia, o Gerenciador de Comunicação deve informar à implementação de OR_Protocol para que passe a aceitar as informações provindas dos participantes do ambiente e os informe de seu ingresso. Preparado o OR_Protocol e já tendo escolhido o participante que fornecerá a matriz para a cópia do grafo de cena, a requisição da cópia é realizada. A figura 8.6 apresenta o diagrama de seqüência para as aplicações que ingressarem em um ambiente de visualização distribuída já existente.

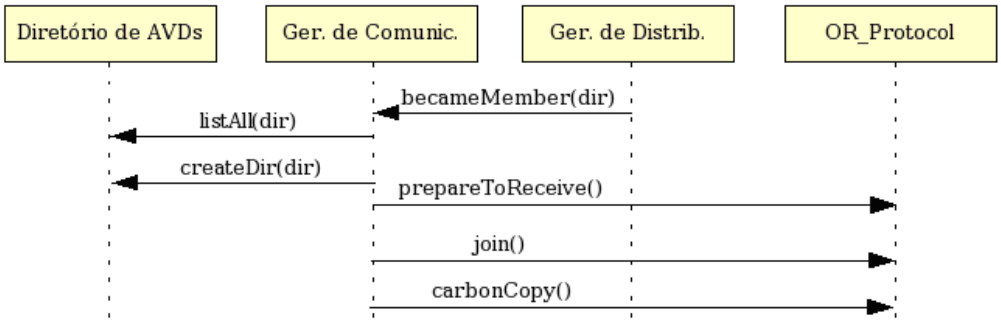

FIGURA 8.6: DIAGRAMA DE SEQÜÊNCIA PARA O INGRESSO EM UM AMBIENTE EXISTENTE

Após o ingresso dos participantes no ambiente estar completo, o envio de informações de atualização de estado ocorrerá sempre que a alteração de um objeto do grafo de cena distribuído de um participante acontecer. Como, inicialmente, o *framework* OpenReality opera apenas com a consistência total de dados, o Repositório de Travas centralizado é utilizado sempre que uma alteração nos objetos do grafo de cena ocorrer. O Gerenciador de Comunicação, ao receber um pedido de envio de atualização de estado provindo do Gerenciador de Distribuição, requisita ao Repositório de Travas pela permissão de alteração do objeto indicado. Caso seu estado temporário indique a situação bloqueada, o Gerenciador de Comunicação não realiza o envio das informações e informa o ocorrido ao Gerenciador de Distribuição. Além disso, realiza a verificação de sincronismo do objeto. Caso esse não se encontre sincronizado, é disparado o processo de sincronização do objeto e a atualização não é enviada. O processo de sincronização do objeto é apresentado na seção 8.2.5. Por outro lado, caso a permissão de alteração do objeto do grafo de cena seja conseguida, os dados da informação de atualização de estado são enviados aos demais participantes, através da primitiva **send** da implementação de OR_Protocol (PDU-OR SNR). Ao término desta operação, a trava do objeto do grafo de cena é devolvida ao Repositório de Travas. Essa é a seqüência de eventos que ocorre no Gerenciador de Comunicação quando as informações de atualização de estado são geradas localmente.

A segunda possibilidade é quando a aplicação recebe as informações de atualização de estado oriundas de modificações ocorridas nos grafos de cena de outros participantes. Nesse caso, o Gerenciador de Comunicação é informado pelo OR_Protocol da existência dessas informações, tendo que, antes de repassá-las ao Gerenciador de Distribuição, realizar a verificação de sincronismo do objeto. A figura 8.7 descreve o diagrama de seqüência para essa situação.

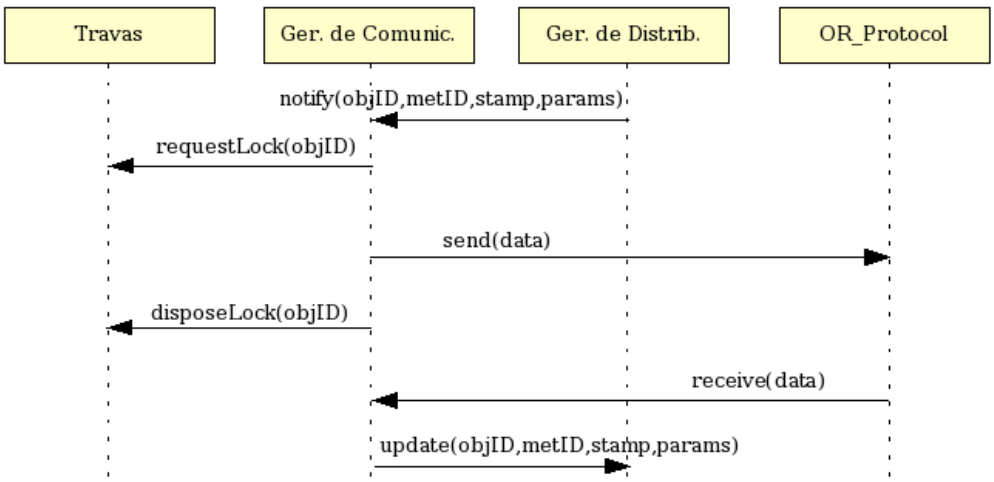

FIGURA 8.7: DIAGRAMA DE SEQÜÊNCIA PARA A TROCA DE ATUALIZAÇÕES DE ESTADO

Por fim, a situação em que um participante abandona o ambiente é apresentada no diagrama de seqüência da figura 8.8. Nesse caso, ao ser informado pelo Gerenciador de Distribuição, o Gerenciador de Comunicação apenas necessita remover sua entrada do Diretório de AVDs e encerrar as atividades do OR_Protocol.

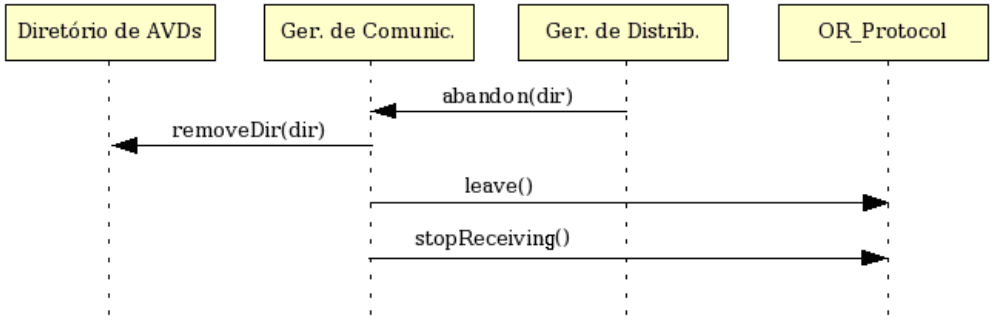

FIGURA 8.8: DIAGRAMA DE SEQÜÊNCIA PARA O ABANDONO DE UM AMBIENTE

As situações apresentadas até o momento ocorrem durante o ciclo de execução das aplicações OR. A seção 8.2.3 descreve o Repositório de Travas mencionado, enquanto a seção 8.3 apresenta as classes dos protocolos de comunicação que implementam a interface OR_Protocol.

8.2.3 O Repositório de Travas

O Repositório de Travas (RT) é implementado através de um objeto JAMP e seu acesso e utilização são permitidos através da estratégia de comunicação apresentada na seção 8.1. O objeto RT possui uma coleção de trincas de informação, compostas por dois valores numéricos inteiros e um lógico. O primeiro valor numérico inteiro, chamado de **sgObjectID**, representa um identificador único para cada objeto pertencente ao grafo de cena da aplicação OR. Já o segundo valor inteiro, chamado de **stampID**, armazena o total de atualizações de estado ocorridas no objeto identificado pelo **sgObjectID**. Por fim, o valor lógico **status**, que representa a situação da trava do objeto. As quatro operações disponibilizadas pelo Repositório de Travas que são chamadas pelo Gerenciador de Comunicação operam sobre as trincas de informação: o cadastro, a remoção, a requisição e a liberação. Todas elas recebem como parâmetro apenas o valor do identificador do objeto pertencente ao grafo de cena da aplicação. A operação de cadastro cria uma nova trinca, contendo o valor do identificador de objeto recebido por parâmetro, o valor zero e o valor falso. Essa combinação representa que nenhuma operação foi realizada sobre o objeto recém cadastrado e que este se encontra disponível para ser alterado. A remoção de uma trinca simplesmente elimina a existência

desses três valores para um dado objeto do grafo de cena. Por fim, as operações que mais interessam ao Gerenciador de Comunicação, a requisição e a liberação. A primeira realiza a checagem do valor **status** do objeto requisitado. Uma vez que este se encontre disponível para atualização, o Repositório de travas altera o valor do campo **status** para verdadeiro e o conteúdo do campo **stampID** é utilizado como valor de retorno do método. Caso o objeto não esteja disponível, nada é alterado e o método retorna um valor numérico negativo. Já a operação de liberação cuida de incrementar o valor contido no campo **stampID** e devolver o valor falso para o campo **status**. A figura 8.9 ilustra como o Gerenciador de Comunicação opera o Repositório de Travas.

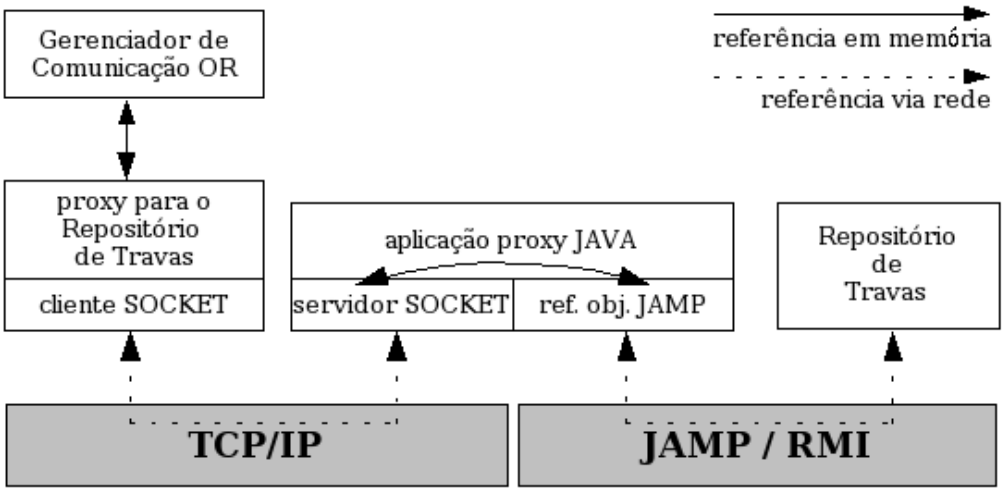

FIGURA 8.9: O REPOSITÓRIO DE TRAVAS OPERANDO COM O GERENCIADOR DE COMUNICAÇÃO

Na próxima seção é apresentada a forma como é realizado o processo de cópia do grafo de cena, que utiliza os recursos do Repositório de Travas para manter a consistência do ambiente de visualização distribuído.

8.2.4 O Processo de Cópia do Grafo de Cena

Durante o processo de cópia do grafo de cena, a implementação de OR_Protocol armazena todas as informações de atualização de estado recebidas, para que, após a conclusão da cópia, as atualizações sejam aplicadas e o grafo de cena copiado iguale-se aos dos demais participantes do ambiente. Por sua vez, a aplicação que serve a matriz do grafo de cena para esse processo suspende a realização de atualizações em seu grafo de cena durante a cópia, para que a fidelidade das informações não seja abalada.

Assim, tanto a aplicação servidora quanto a cliente nesse processo de cópia do grafo de cena provavelmente possuirão cópias desatualizadas deste, pois o processo de transferência

dos dados é relativamente longo e a chance de ocorrência de eventos de atualização de estados no ambiente é grande. Vale notar que as chances de ocorrência desses eventos de atualização de estado aumentam em função do aumento do número de participantes que interagem.

Logo, antes do processo de cópia dos dados propriamente dito ser iniciado, o cliente e o servidor da cópia do grafo de cena passam a armazenar todas as PDUs de transmissão recebidas, definidas pelo protocolo de consistência, sem que estas sejam repassadas para o Gerenciador de Distribuição durante o processo de cópia. Ao término da cópia, tanto a cópia quanto a matriz devem aplicar as atualizações, caso existam, na ordem em que ocorreram.

Para permitir que a ordem de ocorrência das atualizações seja determinada, o protocolo de consistência faz uso dos campos **stampID** armazenados no Repositório de Travas. Sempre que uma aplicação consegue a permissão de alteração de um objeto do grafo de cena, o RT a informa o valor do **stampID**, que é único e deve ser utilizado para a identificação da ordem exata que as atualizações ocorrem.

Durante a atualização de ambos os grafos de cena desatualizados, o Gerenciador de Comunicação recebe cada uma das PDU-OR que foram estocadas pelo OR_Protocol e passa a processá-las. Para cada uma delas, ele recupera o identificador de objeto **sgObjectID** e **stampID**, enviados juntamente com as informações de atualização de estado na primitiva **send**, e compara com as informações presentes em seu grafo de cena local. Sempre que uma PDU-OR SNR que contenha um valor de **stampID** igual ao valor presente no objeto nó do grafo de cena acrescido de uma unidade, as informações de atualização nela contidas são processadas. Caso esse valor seja igual ou inferior, a PDU é descartada.

Desta maneira, a consistência das informações das aplicações de visualização distribuída pode ser mantida em todos os participantes do ambiente.

8.2.5 Sincronizando Objetos do Grafo de Cena

Durante o ciclo de vida das aplicações OR, falhas na rede de dados podem corromper informações de atualização de estado trocadas pelos participantes. Como a maioria dos protocolos oferecidos pelo *framework* não possui garantia de entrega, o Gerenciador de Comunicação deve oferecer tal garantia.

O Gerenciador de Comunicação utiliza o **stampID** de cada objeto para realizar a conferência da ordem das operações informadas pelas PDU-OR. Esse controle é realizado no envio e no recebimento destas.

Caso o valor do **stampID** seja o esperado, a operação é aceita. Caso contrário, caracteriza uma situação de inconsistência. Nesse momento, o Gerenciador de Comunicação

utiliza a primitiva **sync**. A execução dessa operação funciona analogamente à da cópia do grafo de cena, porém, em dimensões menores, pois ocorre apenas no objeto desatualizado e não no Grafo de Cena todo.

8.3 As Implementações de OR_Protocol

Após as definições e especificações das tarefas realizadas pelo Gerenciador de Comunicação, passa-se a apresentar os mecanismos de comunicação, inicialmente previstos na arquitetura OpenReality, que implementam a interface OR_Protocol e oferecem as primitivas para a transmissão de dados na rede.

8.3.1 OR_TCP

O TCP, que é um mecanismo de comunicação bem conhecido, oferece a estrutura de interconexão das aplicações dos participantes da forma como apresenta a figura 8.10. Nesse modelo, todos os participantes estão conectados aos demais, formando um grafo completo de conexões. A classe OR_TCP implementa a interface OR_Protocol, possuindo uma lista de conexões estabelecidas com cada um dos participantes do ambiente. Além disso, utiliza um espaço para o armazenamento das PDU-OR de atualização de estado enquanto realiza a cópia do grafo de cena. O assincronismo dos eventos foi permitido através da utilização de *threads* responsáveis pela recepção dos dados, ou seja, no tratamento da primitiva **receive**.

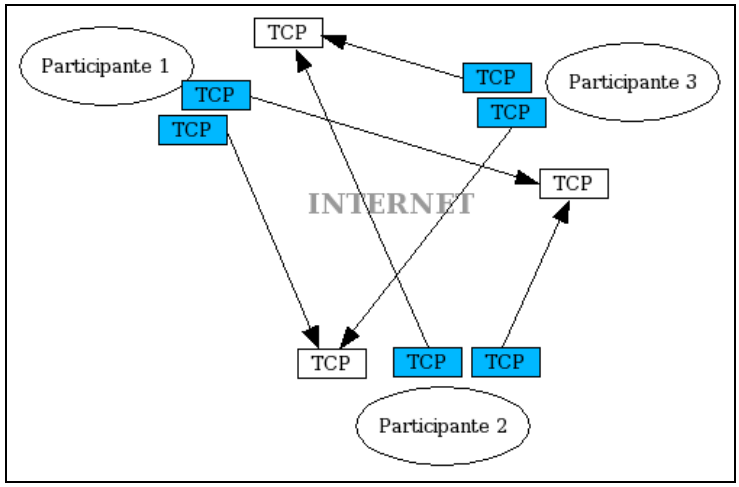

FIGURA 8.10: GRAFO DE CANAIS DE COMUNICAÇÃO OFERECIDO À OR PELO OR_TCP

A manutenção deste grafo é feita através da implementação das primitivas definidas na interface OR_Protocol. Sempre que o método **prepareToReceive()** for executado, um socket servidor deve ser iniciado. Da mesma maneira, a execução do método **stopReceiving()** faz

com que esse socket seja desativado. Já os métodos **join()** e **leave()** informam aos demais participantes para, respectivamente, abrir e encerrar conexões com a aplicação que os executa.

Quando a primitiva **send** é executada por um participante, toda a lista de conexões deve ser percorrida e a PDU-OR SNR de atualização de estado deve lhes ser transmitida.

8.3.2 OR_UDP

Da mesma forma como no TCP, o UDP também mantém sua estrutura organizacional a partir da implementação das primitivas da interface OR_Protocol. A diferença é que, na implementação da classe OR_UDP, cada participante possui apenas um socket UDP e deve manter uma lista com o endereço de todos os demais participantes do ambiente. Essa lista é consultada para endereçar o envio das PDU_OR de atualização de estado. A figura 8.11 ilustra a organização dos canais de comunicação oferecidos pelo UDP.

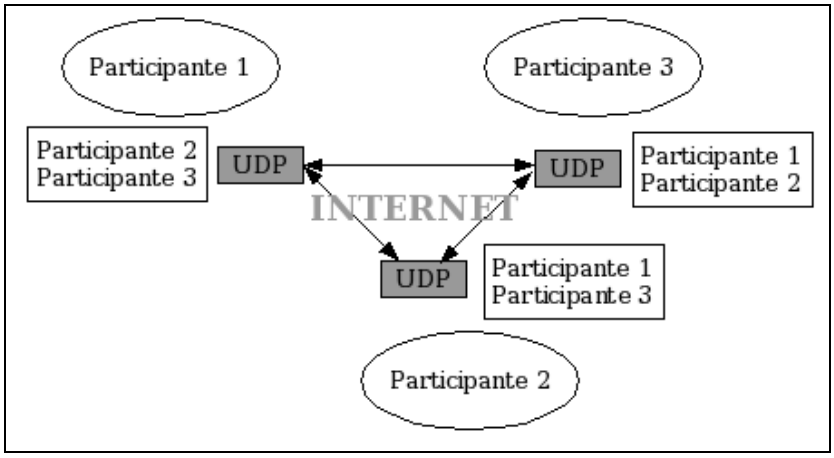

FIGURA 8.11: GRAFO DE CANAIS DE COMUNICAÇÃO OFERECIDO À OR PELO OR_UDP

8.3.3 OR_Broadcast

O protocolo IP *Broadcasting* utiliza as primitivas de serviço do UDP, utilizando como endereço de destino o endereçamento IP de *broadcast.* Como todo pacote enviado para a rede é recebido por todas as demais máquinas presentes naquele segmento, as aplicações dos participantes não necessitam armazenar uma lista de endereços, como ocorre com o UDP. Isso simplifica a implementação da classe OR_Broadcast, principalmente pela desnecessidade da implementação das primitivas **join** e **leave**. As únicas implementações que receberam atenção especial foram as das primitivas **carbonCopy** e **sync**, mas foram resolvidas da mesma forma como em todas as demais implementações de OR_Protocol.

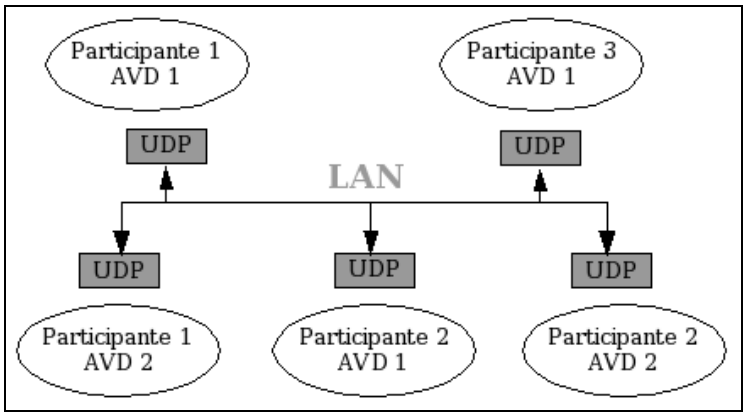

FIGURA 8.12: GRAFO DE CANAIS DE COMUNICAÇÃO OFERECIDO À OR PELO OR_BROADCAST

8.3.4 OR_Multicast

Além de se apresentar como um mecanismo interessante para as aplicações de visualização distribuída, este protocolo resolve a limitação do *broadcast* para máquinas em redes distintas. A implementação da classe OR_Multicast faz uso do serviço de comunicação *multicast* presente na Plataforma JAMP. Segundo Mendonça[38], o *framework multicast* presente na JAMP oferece as funcionalidades de um servidor de grupo *multicast*, definido pelo *Internet Group Management Protocol* (IGMP). Entre elas, estão a criação de grupos *multicast*, a adição de novos participantes ao grupo, a mudança e remoção desses e, como não poderia faltar, a extinção de grupos. Assim, a implementação desse protocolo deixa as questões referentes ao gerenciamento de grupos com o objeto de serviço JAMP que os oferece e trata apenas da criação e manutenção dos sockets para que o protocolo de consistência seja implementado sobre o grupo *multicast*. A figura 8.13 ilustra a organização dos canais de comunicação utilizando esse protocolo.

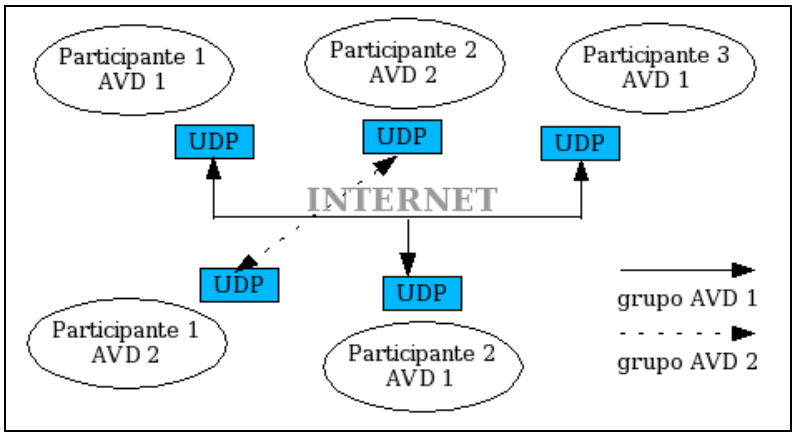

FIGURA 8.13: GRAFO DE CANAIS DE COMUNICAÇÃO OFERECIDO À OR PELO OR_MULTICAST

A implementação da classe OR_Multicast fez uso da estratégia de comunicação com objetos JAMP apresentada anteriormente. A figura 8.14 ilustra a forma como ocorre a utilização do objeto Servidor *MCommunication*[38] pela classe OR_Multicast.

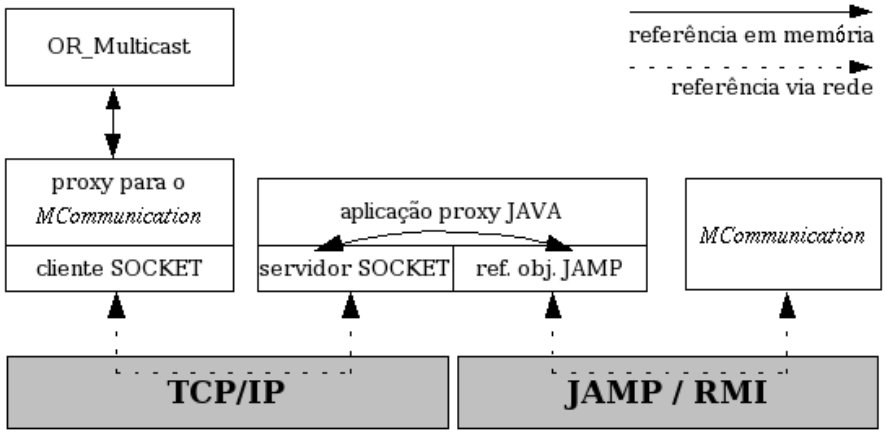

FIGURA 8.14: OR_MULTICAST UTILIZANDO OS SERVIÇOS *MCOMMUNICATION*

8.4 A Classe OR::CommunicationManager

Após apresentadas as funcionalidades e como se dão seus relacionamentos com as estruturas da arquitetura OpenReality, o Gerenciador de Comunicações foi implementado e resultou na classe CommunicationManager, disponibilizada juntamente com as demais que compõem o *framework* OR. Esta seção foi adicionada apenas com o intuito de centralizar um resumo dos métodos oferecidos por essa classe. As assinaturas dos métodos que seguem são as assinaturas de como foram implementadas as operações descritas durante este capítulo.

Os métodos presentes na classe CommunicationManager e que podem ser utilizados pelo Gerenciador de Distribuição e pelos objetos do usuário estão relacionados abaixo.

- `sgObject CommunicationManager :: becameMember( char* directory );`
- `int CommunicationManager :: abandon( char* directory );`
- `int CommunicationManager :: receive( Data& d );`
- `int CommunicationManager :: notify( int objID, int methodID, int stampID, Params p[] );`

8.5 Considerações Finais

Este capítulo apresentou o Gerenciador de Comunicação, responsável por toda a comunicação presente na arquitetura OpenReality e definiu o mecanismo como as aplicações OR interagem. Além disso, definiu também uma estratégia para a comunicação entre as aplicações implementadas na linguagem C/C++ e as implementadas em Java. Por fim, apresentou a forma como este gerenciador interage com os outros módulos da arquitetura e definiu as primitivas de serviço que devem ser implementadas por todos os protocolos que venham a ser adicionados na OR.

Portanto, o Gerenciador de Comunicação foi definido e implementado, de maneira a suprir as necessidades impostas pelo Gerenciador de Distribuição.

9. O Diretório de Ambientes

O Diretório de Ambientes de Visualização Distribuída é requisitado pelas aplicações nos momentos em que essas desejam criar um novo ambiente, ingressar em algum já existente, mudar de ambiente em que se encontram ou abandoná-lo. Sua função é organizar de forma hierárquica os ambientes existentes, facilitando a localização das informações necessárias para aplicações que desejam neles ingressar. Por tratar de uma estrutura que deve ser acessada por várias aplicações e centralizar todas as informações dos ambientes, optou-se pela criação de um serviço na Plataforma JAMP que realizasse as tarefas necessárias e que fosse utilizado pelo Gerenciador de Comunicação. O projeto do sistema de diretórios para a Plataforma JAMP é apresentado no capítulo 11.

Por ora, assume-se a existência de um sistema que permite a criação de diretórios, no qual pode-se armazenar informações de maneira hierárquica, e que encontre-se disponível na Plataforma JAMP. Assume-se, ainda, que esse sistema também é capaz de gerenciar e controlar acessos concorrentes aos seus dados.

Sendo assim, adotando-se a estratégia apresentada no capítulo 8, pode-se utilizar este sistema a partir do Gerenciador de Comunicação da arquitetura OpenReality, da forma ilustrada na figura 9.1.

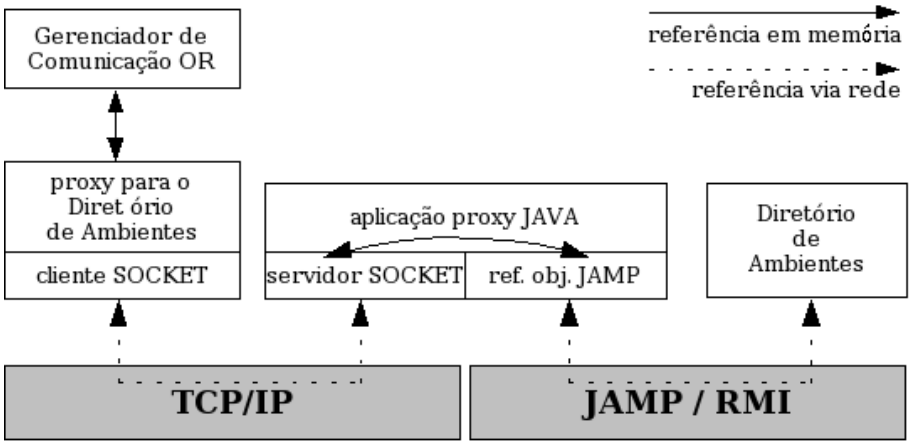

FIGURA 9.1: GERENCIADOR DE COMUNICAÇÃO UTILIZANDO O DIRETÓRIO DE AVDS

Com exceção desses momentos de acesso ao Diretório de Ambientes de Visualização Distribuída, as aplicações não o utilizam mais. Isso permite que o diretório seja um processo detalhado e, até certo ponto, despreocupado com o desempenho, pois não interferirá no decorrer da aplicação.

9.1 Os Dados do Diretório de Ambientes

Os dados que devem ser armazenados nesse sistema de diretórios deve suprir as necessidades impostas pelas situações de uso. Como visto no capítulo anterior, o diretório deve armazenar dois tipos básicos de informações, sobre os ambientes e sobre os participantes. Além disso, sua utilização restringe-se ao armazenamento dos dados referentes aos protocolos de comunicação utilizados na OR. Diante desses requisitos, a tabela 9.1 apresenta os dados que realmente necessitam ser armazenados no sistema de diretórios. A primeira coluna da tabela informa cada um dos atributos (ou propriedades) dos objetos existentes no sistema de diretórios. Enquanto isso, a segunda e a terceira coluna descrevem qual o significado desses atributos para, um ambiente e um participante, respectivamente.

Tabela 9.1: Propriedades do Diretório de Ambientes de Visualização Distribuída

Atributo do diretório	Ambiente	Participante
Tipo	Recebe o valor "GROUP"	Recebe o valor "MEMBER"
data e hora	Data de criação do ambiente	Data de ingresso no ambiente
endereço IP	Endereço de um participante que oferece o grafo de cena	Endereço do participante
número da porta	Porta que o participante aceita requisições	Porta utilizada pelo participante para receber dados
protocolo utilizado	Protocolo utilizado pelos participantes do ambiente	Protocolo utilizado pelo participante
descrição	Descrição do ambiente	Descrição do participante
Estado	Estado em que o grupo de participantes se encontra	Estado em que o participante se encontra
Função	-não utilizado-	-não utilizado-

O atributo tipo é utilizado para identificar se o diretório representa um ambiente ou apenas um participante. Caso represente o primeiro, receberá o valor literal "GROUP". Já no caso de representar um único participante, o valor será "MEMBER".

O atributo data e hora armazena as data e hora exatas que o diretório foi criado. Para ambientes, oferecerá seu tempo de vida, enquanto para os participantes, seus tempos de permanência no ambiente.

O atributo endereço IP armazena o endereço para o qual deve-se destinar as PDUs OR. Esse campo pode receber um tratamento especial para o caso do diretório representar o ambiente, armazenando o endereço IP de um dos participantes ativos do ambiente que é o responsável pelo tratamento das requisições de cópia do grafo de cena. Caso o desenvolvedor deseje aplicar algum algoritmo de balanceamento do atendimento dessas requisições, pode utilizar esse campo para isso. Bastaria a ele implementar um processo que avaliasse a carga de cada participante presente no ambiente e que mantivesse esse atributo do diretório sempre atualizado com o endereçamento do participante mais ocioso, por exemplo.

Juntamente com o atributo anterior, o atributo número da porta oferece todos os dados necessários para uma aplicação saber qual o endereçamento do outro participante.

O atributo protocolo utilizado informa como comunicar-se com o endereço descoberto a partir dos dois atributos anteriores. Vale ressaltar que esse atributo é sempre o mesmo para todos os participantes de um ambiente.

A descrição oferece um espaço para o armazenamento de texto com o intuito de possuir informações adicionais relevantes, porém não necessárias para o funcionamento do ambiente.

O atributo estado, embora não tenha sido utilizado nas implementações realizadas, oferece uma possibilidade adicional para desenvolvedores que desejem criar ambientes que a aplicação participante possa assumir estados diferentes dos costumeiros. Por exemplo, pausar a aplicação de um dos participantes para continuá-la posteriormente, porém, sem que esta abandone o ambiente. Pode ser interessante para o tratamento de situações de máquinas abandonadas repentinamente por seus usuários, que visando economizar recursos, a própria aplicação coloca-se no estado de ausente.

Por último, o atributo função, também não utilizado nas implementações reações realizadas, permite a identificação de participantes que possuem papéis específicos em um ambiente. Um exemplo disso é a presença de uma aplicação, participante de um ambiente, que não é controlada por um usuário, mas por um processo auxiliador de cópia do grafo de cena.

9.2 Os Métodos de Manipulação do Diretório de Ambientes

Diante dos atributos que devem ser armazenados no sistema de diretórios, pode-se definir uma coleção de métodos do tipo inserção, remoção e leitura para manipulá-los. A seguir, são apresentados alguns dos que foram utilizados.

- `bool Diretorio_AVD :: createDir( char* dirName );`
- `bool Diretorio_AVD :: removeDir( char* dirName );`
- `char* Diretorio_AVD :: listAll( char* dirName );`
- `void Diretorio_AVD :: setType( char* dirName, char* type );`
- `char* Diretorio_AVD :: getType( char* dirName );`
- `char* Diretorio_AVD :: getCreationDate( char* dirName );`
- `void Diretorio_AVD :: setIPAddress( char* dirName, char* inetAddress );`
- `char* Diretorio_AVD :: getIPAddress( char* dirName );`
- `void Diretorio_AVD :: setPortNumber ( char* dirName, int port );`
- `int Diretorio_AVD :: getPortNumber( char* dirName );`
- `void Diretorio_AVD :: setProtocol( char* dirName, int protocolID );`
- `int Diretorio_AVD :: getProtocol( char* dirName );`
- `void Diretorio_AVD :: setDescription( char* dirName, char* descr );`
- `char* Diretorio_AVD :: getDescription( char* dirName );`

As três primeiras são operações típicas de um sistema de diretórios. São elas que aparecem nos diagramas de seqüência das situações em que o Gerenciador de Comunicação atua. As demais operações listadas, que ajustam e recuperam os valores de cada atributo específico de um dado diretório, representam as funções dessa implementação particular de sistemas de diretórios. A utilização do termo "implementação particular" na oração anterior sugere que o Diretório de Ambientes de Visualização Distribuída pertença ao conjunto dos Sistemas de Diretórios, cujos elementos sejam especializações funcionais desse domínio de aplicações.

A implementação do Diretório de AVDs serviu-se do reuso de projetos de *software*[39] oferecido pelo *framework* JNDS adicionado à Plataforma JAMP. Sua implementação, bem como a apresentação do próprio *framework*, é tratada no capítulo 11.

9.3 Considerações Finais

Tendo apresentadas todas as estruturas do Subsistema de Comunicação e Distribuição, presentes na camada de Suporte aos Ambientes de Visualização Distribuída, assim como algumas das estruturas da Camada de Suporte, passa-se a tratar das questões específicas da Camada de Serviços, referentes à Plataforma JAMP e aos serviços que ela oferece para a arquitetura OpenReality.

10. Formalização da Plataforma JAMP

Embora a Plataforma JAMP[33] venha sendo palco de vários trabalhos e já tenha colhido diversos frutos, não é encontrada qualquer forma de representação gráfica da arquitetura que define sua plataforma. Textualmente[33], como é encontrada, pode-se compreendê-la e até mesmo utilizá-la em várias aplicações com características distribuídas, mas nem sempre o resultado positivo de uma implementação demonstra o domínio dos conceitos, contidos na arquitetura, por parte do desenvolvedor.

Recorrendo-se à literatura, pode-se encontrar várias representações gráfico-esquemáticas que ilustram a forma como seus respectivos autores conceberam e utilizaram a JAMP em seus projetos. Algumas destas representações, muitas vezes recorrentes, aproximam-se da idéia da arquitetura, porém, em sua grande maioria, ressaltando apenas as características relevantes ao projeto em discussão. Entre os esquemas de representação não textuais encontrados, os que pretendem abordar a arquitetura de uma maneira mais generalizada, acabam tendo a clareza reduzida e não atingindo seus objetivos.

Conforme a plataforma ganha maior maturidade e novas características a ela são adicionadas, ficam melhor definidas cada uma das estruturas que compõem a JAMP. Sendo assim, antes da apresentação dos estudos restantes cobertos por este trabalho, segue uma breve formalização de como a JAMP foi utilizada e manipulada, na tentativa de suprir a necessidade encontrada.

10.1 Histórico de desenvolvimento e versões

A Arquitetura Java para Processamento de Mídia (JAMP) sofreu forte influência de ANSAWare[40], o *middleware* implementado sobre o modelo de visões proposto para a ANSA (Arquitetura Avançada para Sistemas Distribuídos)[41]. Embora o contato e a experiência com os modelos propostos pela ANSA fossem animadores, havia obstáculos em sua adoção nos projetos, como limitações de licença e de linguagem de programação. A necessidade da utilização de um *middleware* nos projetos que envolviam o processamento de

mídias, juntamente com o desejo de elaborar uma arquitetura própria, que buscasse reunir características interessantes presentes nos demais modelos de sistemas de objetos distribuídos, disparou o início dos estudos e propostas para a definição da JAMP.

Entre as tecnologias oferecidas na época de seu desenvolvimento, Java RMI[42] foi a que melhor atendeu aos requisitos da JAMP. Embora a OMG[43] já possuísse, há algum tempo, as especificações da OMA[44] e de CORBA[45], elas ainda encontravam-se nos seus primeiros conjuntos de versões e com poucas implementações não comerciais. Pouco tempo após o início de seu desenvolvimento, estava concluído o componente principal da arquitetura, o JBroker[46]. Logo seguiram os primeiros projetos de *frameworks* de domínios de aplicação e aplicações de gerenciamento que hoje compõem parte da coleção de *software* que forma a Plataforma JAMP. JBroker Manager[47], CoopWrite[48], CoopGraph[49] e ChatMulticast[38] são algumas das aplicações que utilizam a JAMP de acordo com sua primeira versão. Mais tarde, Souza[50] agregou novas características e funcionalidades à arquitetura, disponibilizando a segunda versão. Em seu trabalho, Souza[50] concebeu a JAMP de uma maneira melhor organizada, tomando por base a nova especificação de CORBA 2.0[51].

A versão 2.0 de Souza trouxe para a JAMP muitos dos novos serviços oferecidos pelo recém especificado ORB da OMG. Mesmo com todas as mudanças entre a antiga e a nova versão, a compatibilidade foi mantida.

Assim, de maneira natural, a JAMP passou a concretizar uma arquitetura muito próxima da que fora especificada pela OMG, a OMA[44], mesmo que sem as devidas formalidades. Atualmente, pode-se encontrar os projetos sobre a Plataforma JAMP divididos de acordo com as especificações nova e antiga do JBroker, embora a arquitetura não tenha sido alterada.

10.2 A Arquitetura Java para Processamento de Mídia (JAMP)

Devido aos projetos pertencentes ao MultiEng que motivaram o seu desenvolvimento, a JAMP recebeu este nome. Porém, desde o início do projeto da arquitetura, houve uma divisão de funcionalidades que resultou em um modelo para o desenvolvimento de aplicações distribuídas de propósitos gerais, não se restringindo apenas ao processamento de mídias. Vários projetos com este foco inicial trouxeram facilidades à plataforma na época, mas atualmente a JAMP vem sendo palco de pesquisas envolvendo, desde computação de grade[52], até ubíqua[53]. Neste projeto, por exemplo, ela é utilizada para prestar suporte à arquitetura OpenReality[35].

A Arquitetura Java para Processamento de Mídia é mostrada na figura 10.1. Sua organização assemelha-se ao modelo proposto pela OMG, sendo dividida em Objetos da Aplicação, Mecanismo de Distribuição de Objetos, Objetos de Serviço e *Frameworks* de Domínio.

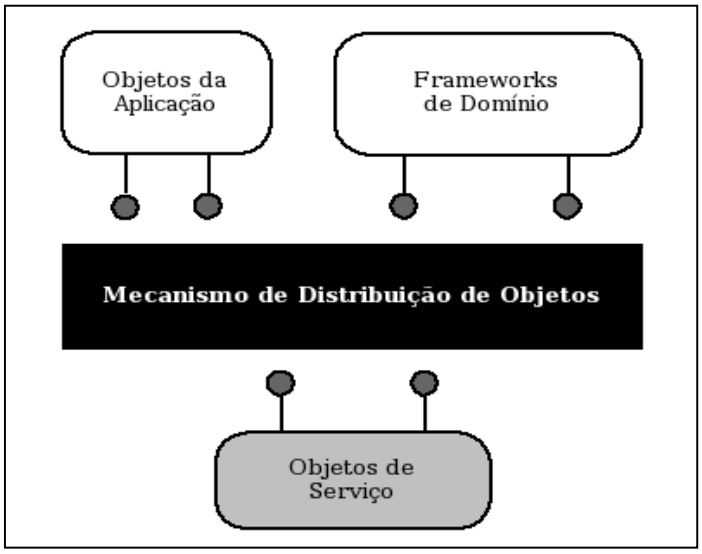

FIGURA 10.1: ARQUITETURA JAVA PARA PROCESSAMENTO DE MÍDIA

Objetos da Aplicação representa o conjunto dos objetos que pertencem à aplicação e que utilizam os serviços oferecidos pelo restante da arquitetura. A grosso modo, pode ser visto como o conjunto dos objetos que permitem o acesso aos serviços da arquitetura.

Objetos de Serviço é o conjunto composto por todos os objetos que oferecem facilidades gerais para o funcionamento da aplicação.

Frameworks de Domínio são *frameworks* presentes na arquitetura que tratam de domínios específicos de aplicações. Pode ser comparado às facilidades verticais definidas pelo modelo CORBA[54].

Mecanismo de Distribuição de Objetos é formado por um conjunto de *software*, composto por modelos e protocolos de comunicação, responsável pela troca de informações entre as aplicações.

Atualmente, as estruturas que compõem a JAMP encontram-se na seguinte situação:

Mecanismo de Distribuição de Objetos: Utiliza o modelo Java RMI[55] como principal forma de distribuição e controle de objetos. Todas as características e funcionalidades presentes nesse modelo são mantidas na JAMP, como montagem e desmontagem das séries de dados que representam implementações de classes, registro e localização via *rmiregistry*, controle de concorrência, chamadas síncronas e localização estática e dinâmica de interfaces.

Objetos de Serviço: Possui a estrutura imprescindível para o funcionamento da JAMP, o JBroker[50], que atua na arquitetura realizando o processo de *trading*. A figura 10.2 ilustra esse processo, que é a forma como devem ser iniciadas as aplicações que utilizam a JAMP.

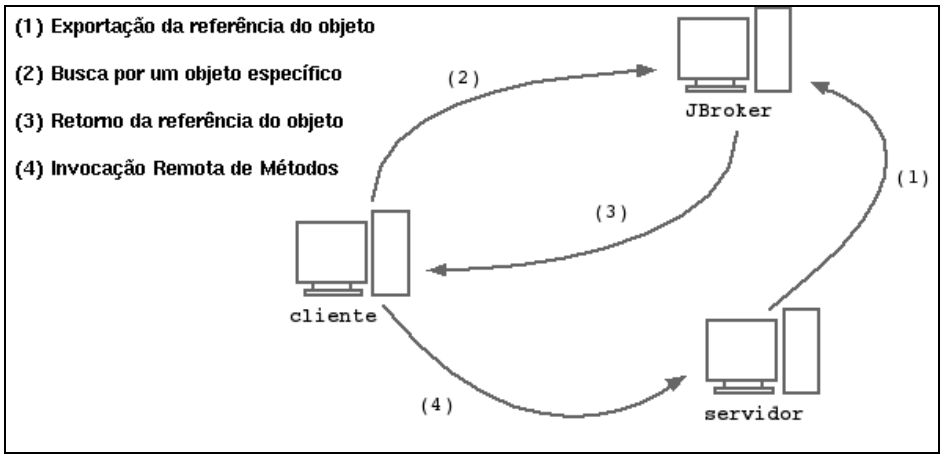

FIGURA 10.2: O PROCESSO DE *TRADING*

***Frameworks* de Domínio**: Composto pelos domínios de aplicação Áudio, Vídeo, Rope, Midi, Trabalho Colaborativo, Comunicação de Grupo e Comércio Eletrônico.

Objetos de Aplicação: É composto por todas as aplicações existentes que utilizaram a JAMP como uma plataforma de distribuição.

10.3 O pacote de desenvolvimento JAMP

Juntamente com as estruturas que definem a arquitetura e suas respectivas implementações, é disponibilizado um conjunto de ferramentas e utilitários que auxiliam no desenvolvimento e na implementação das aplicações distribuídas. A figura 10.3 representa como é composto o pacote de desenvolvimento da JAMP.

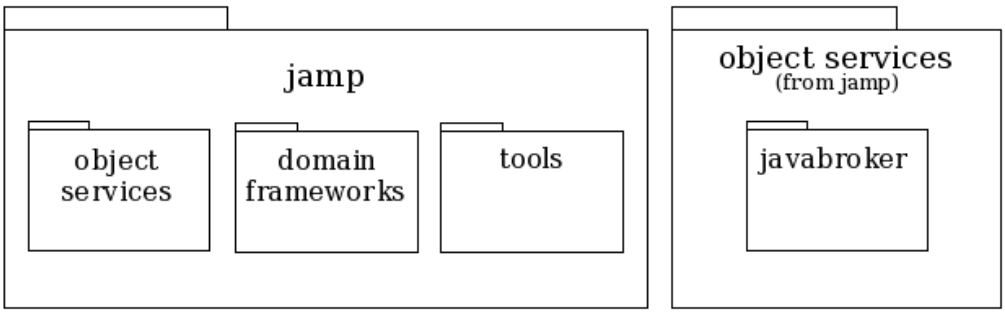

FIGURA 10.3: O PACOTE DE DESENVOLVIMENTO JAMP

10.3.1 O desenvolvimento de aplicações usando a JAMP

Durante o processo de desenvolvimento de aplicações distribuídas que utilizam as facilidades oferecidas pela JAMP, é comum a utilização do termo Plataforma JAMP. O fato se deve à forma organizacional que tomam as implementações ao utilizarem os serviços oferecidos pela JAMP.

Esquematicamente, ao utilizar a JAMP em um projeto, destacam-se três camadas bem definidas. Como bem descreve Souza[50], genericamente a plataforma é composta pelas camadas Aplicação, Serviços e Infra-estrutura. A mesma semântica de separação é relatada no estudo de caso de Prado[33], que descreve uma abordagem para o desenvolvimento de aplicações distribuídas utilizando a Plataforma JAMP.

Sendo assim, assume-se neste trabalho que o termo Plataforma JAMP será empregado sempre que uma referência ao conjunto dos *softwares* que implementam a JAMP seja necessária. A figura 10.4 representa a arquitetura da lógica estrutural de aplicações distribuídas ao fazerem uso da Plataforma JAMP.

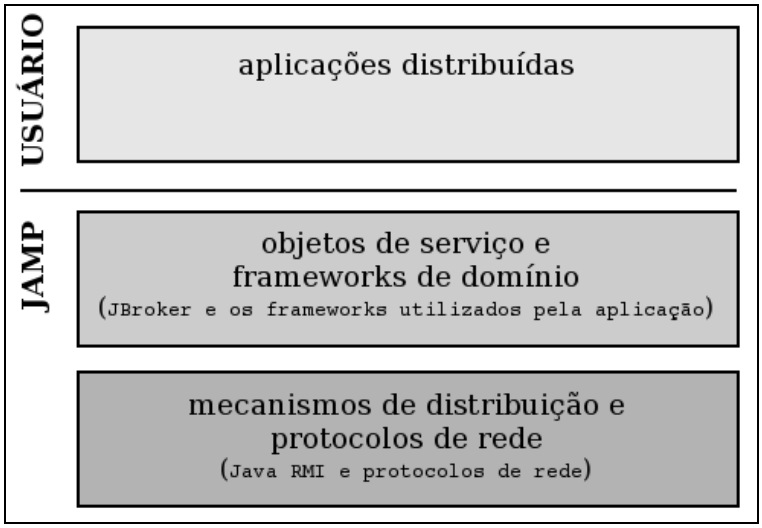

FIGURA 10.4: ARQUITETURA DAS APLICAÇÕES JAMP

Seguindo a arquitetura exibida na figura 10.4, o desenvolvedor (usuário) atém-se às questões relevantes à sua implementação particular, não sendo necessário preocupar-se com as questões referentes à distribuição da aplicação, função desempenhada pela JAMP. Na camada mais inferior, encontram-se Java RMI e seus *stubs*[55] de objetos realizando as tarefas do Mecanismo de Distribuição de Objetos. A camada intermediária sempre é composta pelo JBroker, podendo também conter quaisquer *frameworks* do conjunto *Frameworks* de Domínio, além, é claro, de implementações de objetos servidores definidos

pelo próprio usuário. Por fim, de acordo com os diagramas de casos de uso, colaboração e atividades[56], que especificam o comportamento da aplicação, encontram-se, na camada superior, as interações dos objetos de aplicação conforme a implementação do usuário.

10.4 Considerações Finais

Este capítulo apresentou uma breve formalização a respeito da plataforma de *software* que implementa a Arquitetura Java para Processamento de Mídia (Plataforma JAMP), bem como sua definição, suas versões e seu histórico de desenvolvimento. Esse conjunto de *softwares*, além de tornar a arquitetura funcional, ainda permite que desenvolvedores reutilizem seus serviços, pois, em sua maioria, estão disponíveis na forma de *frameworks*.

Apresentada a arquitetura, pode-se continuar a apresentação dos projetos realizados neste trabalho.

11. JAMP Networked Directory System: um novo framework de serviços para a Plataforma JAMP

Diante da necessidade apresentada pela arquitetura OpenReality em possuir um sistema de diretórios, juntamente com não existência deste serviço na JAMP, iniciou-se o projeto do sistema de diretórios distribuídos da JAMP, ou simplesmente JNDS. Seus objetivos são suprir a carência existente na JAMP e oferecer um *framework* para que desenvolvedores sejam capazes de utilizar o novo serviço, além de permitir que criem seus próprios sistemas de diretórios de maneira personalizada.

11.1 A Definição do *Framework*

Seguindo os passos que Teixeira[57] aconselha, o projeto deve ser capaz de definir comportamentos abstratos, para que o reuso não seja apenas do código, mas sim do projeto como um todo. A abstração dos comportamentos gera o que é chamado de domínio (ou família) de aplicação, enquanto que a especialização deles leva às aplicações do domínio. O conjunto de classes que fornece uma solução genérica para um determinado domínio de aplicação é chamado de *framework* orientado a objetos[58]. Assim, os *frameworks* distinguem-se das bibliotecas de função, pois enquanto os primeiros possuem todo o fluxo de dados e a lógica da aplicação, elas precisam ser invocadas e utilizadas segundo a lógica e o controle do desenvolvedor. A figura 11.1 ilustra a diferença apontada por Souza[59].

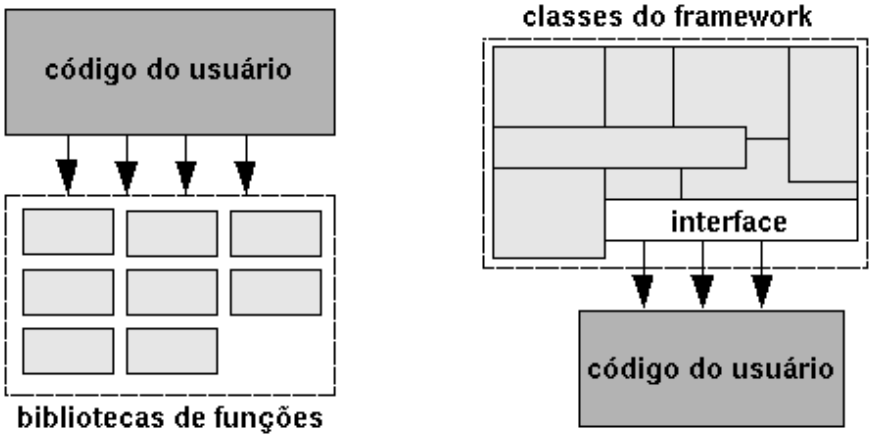

FIGURA 11.1: BIBLIOTECA DE FUNÇÕES *VERSUS* FRAMEWORKS

Um estudo detalhado sobre as diferenças entre *frameworks* e Padrões de Projeto, vantagens de sua utilização, metodologias para seu desenvolvimento e as dificuldades do seu projeto pode ser encontrado no trabalho de Mendonça[38].

Portanto, segundo Teixeira[57], para a criação do *framework* JNDS, faz-se necessária a definição do domínio das aplicações, partindo-se das definições mais abstratas possíveis dos sistemas de diretórios.

11.2 Sistema de Diretórios

Um sistema de diretório pode ser visto como um mecanismo lógico para o armazenamento de informações organizadas hierarquicamente seguindo um determinado critério[59]. As recomendações do antigo CCITT (atual ITU-T), que definiram o X.500 e tornaram-se posteriormente a série de normas ISO/IEC 9594[60], definem o que é um sistema de diretórios em sua essência. Em sua primeira recomendação, define os conceitos dos componentes de um sistema de diretórios, bem como seu modelo e serviços oferecidos. Sempre que necessário, faz analogia entre o sistema de diretório e um catálogo telefônico, pois os serviços oferecidos por ambos são os mesmos.

Embora X.500 possua propósitos próprios, pois oferece o serviço de diretório para o modelo OSI[61], define de forma genérica o termo sistema de diretórios. Alguns exemplos desses sistemas de organização de informação que podem ser encontrados facilmente são os sistemas de arquivos dos sistemas operacionais e o DNS[62] (*Domain Name System*) utilizado por diversos serviços na Internet.

Entre todos os conceitos definidos pela série de recomendações X.500, quatro deles são essenciais:

- **objeto:** é a estrutura de armazenamento dos dados
- **nome:** denota um objeto
- **endereço:** indica a localização de um objeto
- **rota:** indica como chegar até o objeto

Além dos conceitos, a recomendação X.500 define as seguintes regras sobre eles:

- *nomes possuem uma ou mais partes denominadas atributos;*

- *as partes de um nome têm um relacionamento hierárquico;*
- *o espaço dos nomes, a partir do qual os nomes são criados, é estruturado em árvore, sendo que os nomes são associados aos arcos e os objetos aos nós;*
- *todos os objetos que compartilham do mesmo nó pai possuem nomes relativos diferentes, isto é, os arcos do nó pai até eles devem receber nomes diferentes;*
- *cada nó possui um único nome absoluto, formado pela justaposição dos nomes associados aos arcos que compõem a rota da raiz da árvore até o objeto.*

A figura 11.2 resume as definições e regras apresentadas para um sistema de diretórios.

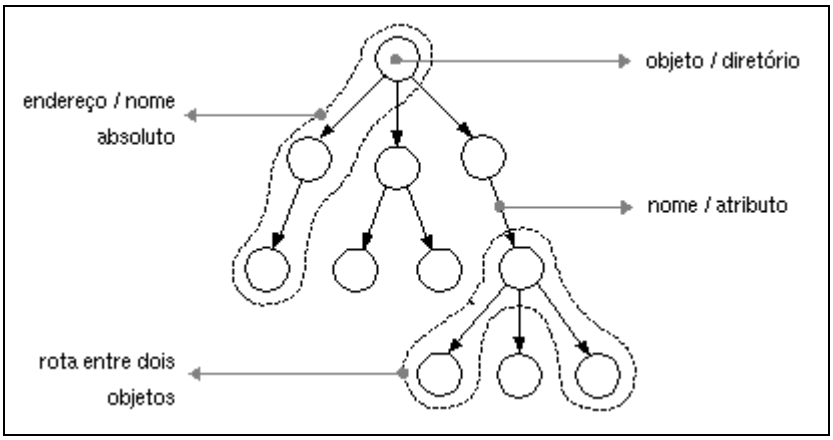

FIGURA 11.2: RESUMO DAS DEFINIÇÕES

11.3 Abstração do Sistema

Da maneira como é definido pelo padrão X.500 e como também pode ser verificado nos sistemas de diretórios existentes, o conjunto de serviços oferecidos por um sistema de diretórios deve conter a adição, a remoção, a atualização e a listagem de dados, além de prover mecanismos de autenticação, validação e definição de direitos e acesso dos usuários. Assim, estruturas de armazenamento de dados e de estruturação hierárquica devem existir para suportar as operações que um sistema de diretórios oferece.

A figura 11.3 apresenta uma abstração do sistema de diretório que foi projetado, já contendo estruturas adicionais às especificadas no padrão X.500 pelo CCITT, pois estas vieram a ser incorporadas nas normas posteriores. A representação dessa figura assume que uma dada estrutura utiliza os serviços oferecidos pelas demais estruturas adjacentes localizadas imediatamente abaixo dela, não havendo relacionamentos colaborativos entre as adjacências horizontais.

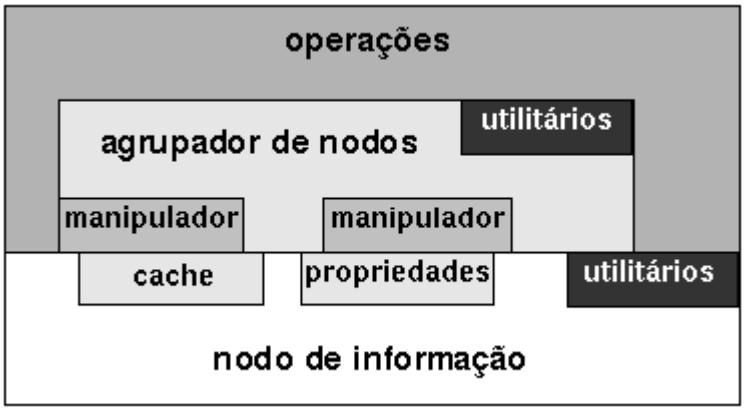

FIGURA 11.3: ARQUITETURA ABSTRATA DO SISTEMA DE DIRETÓRIOS

A estrutura básica da arquitetura apresentada é o nodo de informação, que é responsável pelo armazenamento dos dados, ou propriedades, de cada nó. As propriedades representam um conjunto de atributos que um determinado nó pode possuir.

O agrupador de nodos é a estrutura de dados que, além de organizar hierarquicamente os nós, possui primitivas de serviço que são utilizadas pelas operações.

O *cache* representa estruturas de armazenamento para agilizar o processo de localização dos nós. Cada nó possui seu próprio *cache*, que pode conter os dados dos seus nós filhos, netos, filhos dos netos, etc.

Os manipuladores de *cache* e de propriedades são estruturas que auxiliam o agrupador de nodos a realizar, respectivamente, as operações específicas do *cache* e do conjunto de propriedades.

Os utilitários são funções específicas que poupam o trabalho do desenvolvedor, sejam utilitários que operam sobre os nodos de informação ou sobre o agrupador de nodos.

Por último, as operações que são exportadas pelo sistema de diretórios para as aplicações dos usuários. Elas compõem a API do *framework*, que deve ser utilizada pelas aplicações que desejam utilizar os serviços do sistema de diretórios.

11.4 Projeto e Implementação

Tendo definidas as estruturas necessárias do sistema de diretórios, pode-se dar continuidade ao projeto do *framework*, definindo as operações que este suporta. Em seguida, também são apresentados quais são os pontos de personalização (*hotspots*) disponíveis.

11.4.1 Operações do Sistema de Diretórios

Embora a primeira aplicação beneficiada pelo JNDS seja o Diretório de Ambientes de Visualização Distribuída, o conjunto de funções oferecidas pelo *framework* deve ser mais abrangente do que o conjunto particular de operações por ela requisitado. No esforço de disponibilizar o maior número dessas operações inicialmente, o projeto do JNDS espelhou-se em alguns sistemas de arquivos presentes nos sistemas operacionais. Desta forma, as operações suportadas pelo JNDS são as que encontram-se listadas na tabela 11.1.

Tabela 11.1: Operações do JNDS

OPERAÇÃO	DESCRIÇÃO
cópia de diretórios	*copia um diretório já existente em uma nova rota*
criação de diretórios	*adiciona novos nomes e objetos em objetos já existentes*
recuperação da rota de um diretório	*informa a rota de um dado objeto desde a raiz*
recuperação de um diretório	*troca o diretório corrente de operação*
listagem do conteúdo de um diretório	*exibe os subdiretórios do diretório corrente*
listagem das propriedades do conteúdo de um diretório	*exibe as propriedades dos subdiretórios do diretório corrente*
criação de um sistema de diretórios	*prepara a raiz para receber um tipo de sistema de diretório específico*
mudança da rota de um diretório	*move um determinado diretório para uma rota diferente da que ele se encontra*
exclusão de um diretório	*Remove o(s) diretório(s) a partir da rota especificada*
exclusão de todos os diretórios	*Remove todos os diretórios existentes a partir da raiz*
ajuste de propriedades a um diretório	*troca o valor de uma determinada propriedade de um dado diretório*
atribuição de propriedades a um diretório	*insere um valor para uma dada propriedade de um determinado diretório*

Caso o projeto do JNDS visasse suportar simples sistemas de diretórios, utilizados apenas por aplicações não distribuídas, as definições e especificações apresentadas até o momento seriam suficientes para o início das implementações. Porém, deve-se ter em mente que todas as operações listadas na tabela anterior podem ocorrer simultaneamente, caso diversos usuários utilizem o mesmo sistema de diretórios.

Nestas situações, o sistema de diretórios deve oferecer mecanismos que controlem o acesso às regiões críticas, proporcionem exclusão mútua quando necessário e, principalmente, disponibilizem meios do recurso (sistema de diretórios) ser compartilhado.

11.4.2 Mecanismos de Controle e Manutenção

Várias são as maneiras de realizar a exclusão mútua para controlar o acesso às regiões críticas, desde registros centralizados que armazenam quais são as aplicações que detêm o

acesso até algoritmos distribuídos para eleger qual aplicação pode acessar o recurso[31]. A escolha do mecanismo utilizado no projeto do JNDS foi feita visando a simplificação das implementações e, como conseqüência, buscando atingir um desempenho maior em execução. Obviamente, o mecanismo foi projetado de forma a oferecer uma robusteza suficiente para garantir a consistência das informações presentes no sistema de diretórios.

Alterações nas informações contidas nos diretórios (nós da árvore), quando ocorrem desordenadamente, podem causar inconsistências irrecuperáveis. Basta imaginar uma simples situação em que dois usuários simultâneos utilizam o mesmo sistema de diretórios. Enquanto o primeiro inicia uma operação de mudança na rota de um dado diretório, o segundo começa a remover a rota de destino da primeira operação. Caso não haja um mecanismo de controle, certamente uma das operações será prejudicada. Os resultados são indeterminísticos, pois além de não ser possível prever qual das operações finalizará primeiro, ainda existe a possibilidade da corrupção dos dados e ocorrência de falhas que comprometam o sistema de forma geral. Por outro lado, não havendo intersecção entre as rotas das operações, qualquer forma de controle torna-se desnecessária.

Sendo assim, as operações oferecidas pelo sistema de diretório foram separadas em duas classes, aquelas que reduzem ou remodelam a estrutura da árvore de diretórios e as que simplesmente a aumentam. Além disso, o controle das operações foi centralizado, de forma a garantir que as informações de controle mantenham-se coerentes. As tabelas 11.2a e 11.2b apresentam as operações, realizadas nos nós da árvore de diretórios, para cada comando oferecido pelo sistema de diretórios. As operações que executam a remoção do diretório de origem pertencem à classe das operações que reduzem ou remodelam a estruturação da árvore. As demais pertencem à segunda classe de operações.

Tabela 11.2: Modificações nos nós da árvore de diretórios

operação	**origem**	**destino**
formatação	*remoção*	-
remoção	*remoção*	-
propriedades	*escrita*	-
criação	*escrita*	-
listagem	*leitura*	-

(a)

operação	**origem**	**Destino**
Mudança	*remoção*	*Escrita*
Cópia	*leitura*	*Escrita*

(b)

A política de permissão de execução adotada garante que, quantas e quaisquer que sejam, operações da classe que aumenta a árvore de diretórios podem ser executadas em um

dado nó, desde que nenhuma operação da classe de redução ou remodelagem esteja sendo executada em algum dos nós de sua rota. Já as operações de redução ou remodelagem da árvore devem ser executadas apenas quando possuírem exclusividade em um dado nó, inclusive entre elas mesmas.

A figura 11.4 apresenta um esquema de utilização do mesmo sistema de diretórios por três usuários, mostrando como as informações de controle são manipuladas. Comandos conhecidos de manipulação de diretórios são utilizados para representar as ações tomadas pelos usuários das aplicações.

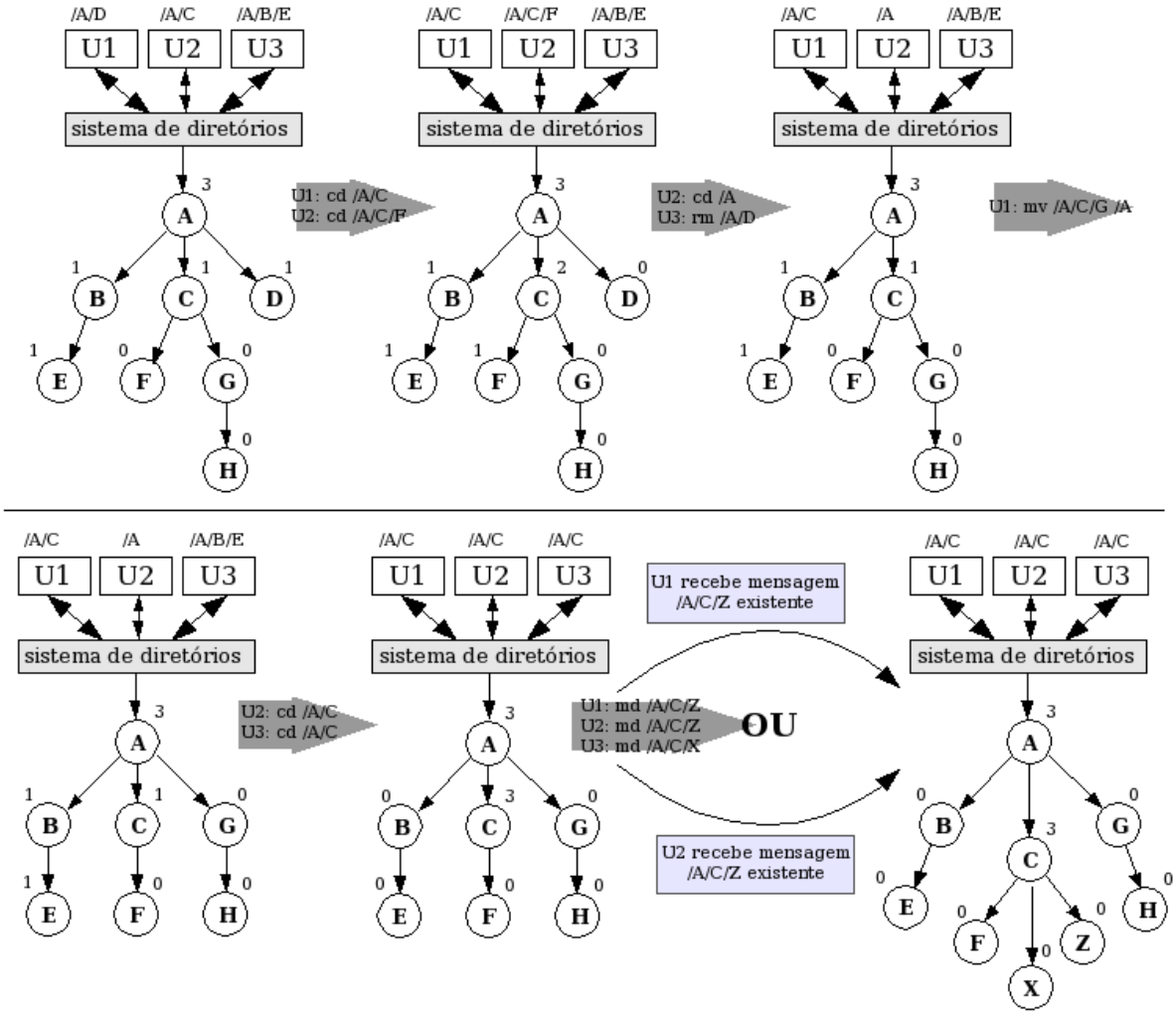

FIGURA 11.4: EXEMPLO DE MANUTENÇÃO DA CONSISTÊNCIA COM CONTADORES DE USO

Como pode ser observado na figura a anterior, cada nó (diretório) possui um contador de uso, que é incrementado para cada nova aplicação usuária do sistema de diretórios,

computando os nós desde a raiz até o nó em que a aplicação do usuário se encontra. Desta forma, antes que as operações formatação, remoção e mudança apresentadas nas tabelas 11.2a e 11.2b sejam executadas, é necessária a verificação do valor do contador de uso presente no diretório em que elas realizar-se-ão. Sempre que o valor encontrado for maior que zero, indicando que uma ou mais operações que aumentam a árvore estão ocorrendo, a operação não é continuada, pois pode causar inconsistências. Um exemplo típico é a ocorrência de erros em sistemas de arquivos causados por violações de compartilhamento. O sistema de diretórios deve sempre evitar tais situações.

Outras situações, que aparentemente também podem causar inconsistências, são perfeitamente aceitáveis em um sistema de diretórios compartilhado por múltiplos usuários. As figuras 11.5a e 11.5b retratam duas destas situações. A primeira mostra uma cópia ocorrendo ao mesmo tempo que uma criação, enquanto a segunda exibe a mesma cópia concorrendo com uma remoção.

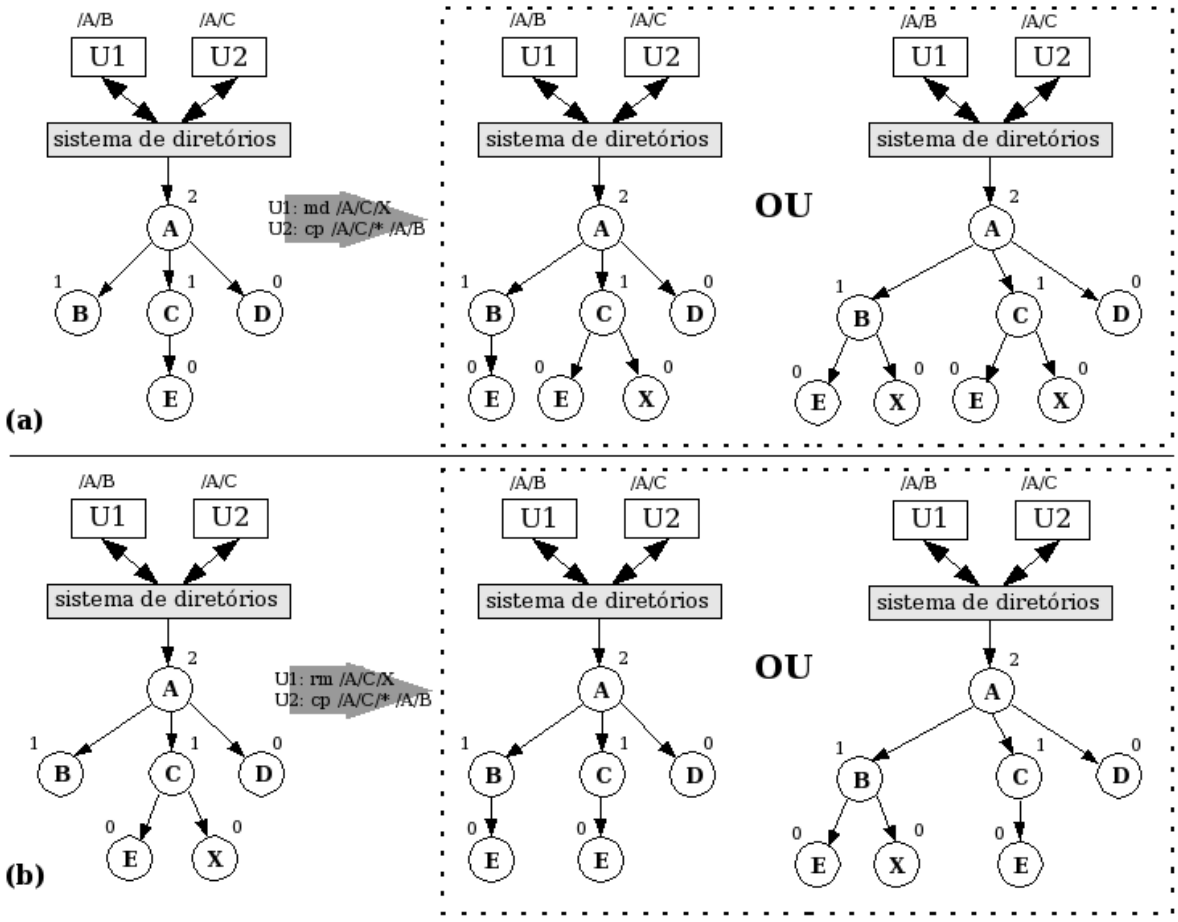

FIGURA 11.5: EXEMPLO DE INDETERMINISMO PERMITIDO NO SISTEMA

Embora os nós da árvore de diretórios possuam métodos que adicionem, recuperem e removam seus atributos, a exclusão mútua em cada um deles é garantida pelo mecanismo *synchronized* da linguagem Java. Já as operações que processam os comandos dos usuários, que manipulam os nós, necessitam seguir as políticas de permissões antes de prosseguirem.

As operações da classe de redução ou remodelagem da árvore nunca devem ser executadas de forma paralela ou concorrente. Sendo assim, só podem prosseguir suas execuções no caso do contador de uso do nó ser igual a zero. Além disso, devem atribuir um valor negativo a esse contador, para evitar que qualquer outra operação da mesma classe seja executada.

Por sua vez, as operações da classe que aumenta a árvore de diretórios devem também realizar checagens segundo a política de permissões. Sempre que o contador mencionado possuir um valor maior ou igual a zero, poderão prosseguir suas execuções. Caso contrário, alguma operação da outra classe de operações estará sendo executada e, de acordo com a política de permissões, não devem prosseguir suas execuções, a fim de evitar inconsistências.

11.4.3 Mecanismos de Distribuição e Compartilhamento

O compartilhamento do JNDS segue o mecanismo de exportação e importação. Dentro do contexto da JAMP, uma aplicação que exporta um sistema de diretórios é considerada um servidor de diretórios e a aplicação que o importa é chamada de cliente do serviço de diretórios. Todos os trâmites entre o início das aplicações e suas efetivas interações seguem o processo de *trading*, descrito na seção 10.2.

O *framework* JNDS foi desenvolvido de maneira a permitir que tanto aplicações locais quanto distribuídas pudessem fazer uso de suas funcionalidades. De uma maneira geral, é possível conceber o JNDS como um sistema de diretórios que pode ser utilizado local ou remotamente. Essa funcionalidade foi obtida acrescentando-se uma interface de acesso ao sistema de diretórios, separando assim a aplicação do sistema de diretórios propriamente dito. Através dessa interface, que define as operações do sistema de diretórios, aplicações fazem uso remoto dos serviços disponíveis nos servidores de diretórios da mesma maneira que o fazem ao instanciar um sistema de diretórios localmente.

Ao exportar um sistema de diretórios, uma referência remota do objeto que controla e mantém a estrutura de diretórios é disponibilizada aos clientes. Assim, todas as políticas de permissões que são executadas por esse objeto também as serão quando as operações a serem executadas forem disparadas por clientes do serviço de diretórios.

Embora toda a comunicação seja realizada sobre o modelo Java RMI, que implementa mecanismos de múltiplas *threads* para atender as chamadas de procedimento remoto (RPC) dos clientes de maneira mais otimizada, o objeto exportado também faz uso da exclusão mútua oferecida pela própria linguagem Java para garantir a consistência das informações de controle. Isso faz com que, independentemente de quantos forem os clientes do serviço de diretórios, sejam eles locais ou remotos, a consistência continue sendo mantida.

11.4.4 A Modelagem do *Framework* JNDS

Seguindo o modelo de arquitetura da figura 11.2, considerando as operações da tabela 11.1 e prevendo estruturas que suportem os mecanismos de manutenção de consistência e de distribuição apresentados, o *framework* JNDS foi modelado como mostra a figura 11.6. As classes e interfaces presentes no interior do retângulo tracejado compõem a essência do *framework*, que busca atingir o primeiro dos quatro objetivos propostos no guia de desenvolvimento Taligent[63], a flexibilidade. Nesse guia, as abstrações de um *framework* devem permitir seu reuso em diferentes contextos. Já as classes localizadas ao seu redor, buscam o aperfeiçoamento do segundo dos objetivos citados no guia, a máxima abrangência. Ele afirma que, além de fornecer as características desejadas pelos clientes, o projeto de um *framework* deve fornecer uma implementação padrão sempre que possível, pois ela auxilia aos clientes a utilizá-lo e melhor compreendê-lo.

A presença das dez interfaces no projeto facilita a extensão e a personalização dos comportamentos, da maneira como o guia Taligent considera ideal. Por fim, o último objetivo importante é o grau de compreensão oferecido pelo projeto. Segundo as diretrizes do guia, as interações entre o código do cliente e o *framework* devem ser claras e bem documentadas, para que o aprendizado de sua utilização seja rápido.

Como pode ser visto no diagrama de classes da figura 11.6, uma aplicação que deseja fazer uso do *framework* JNDS deve implementar cada uma das interfaces definidas internamente ao retângulo tracejado. Além disso, podem também estender ou simplesmente utilizar as classes já disponibilizadas que implementam as devidas interfaces. As classes apresentadas no diagrama anterior fazem parte da implementação padrão do JNDS. Qualquer comportamento pode ser alterado por meio de herança, polimorfismo e re-implementação das interfaces.

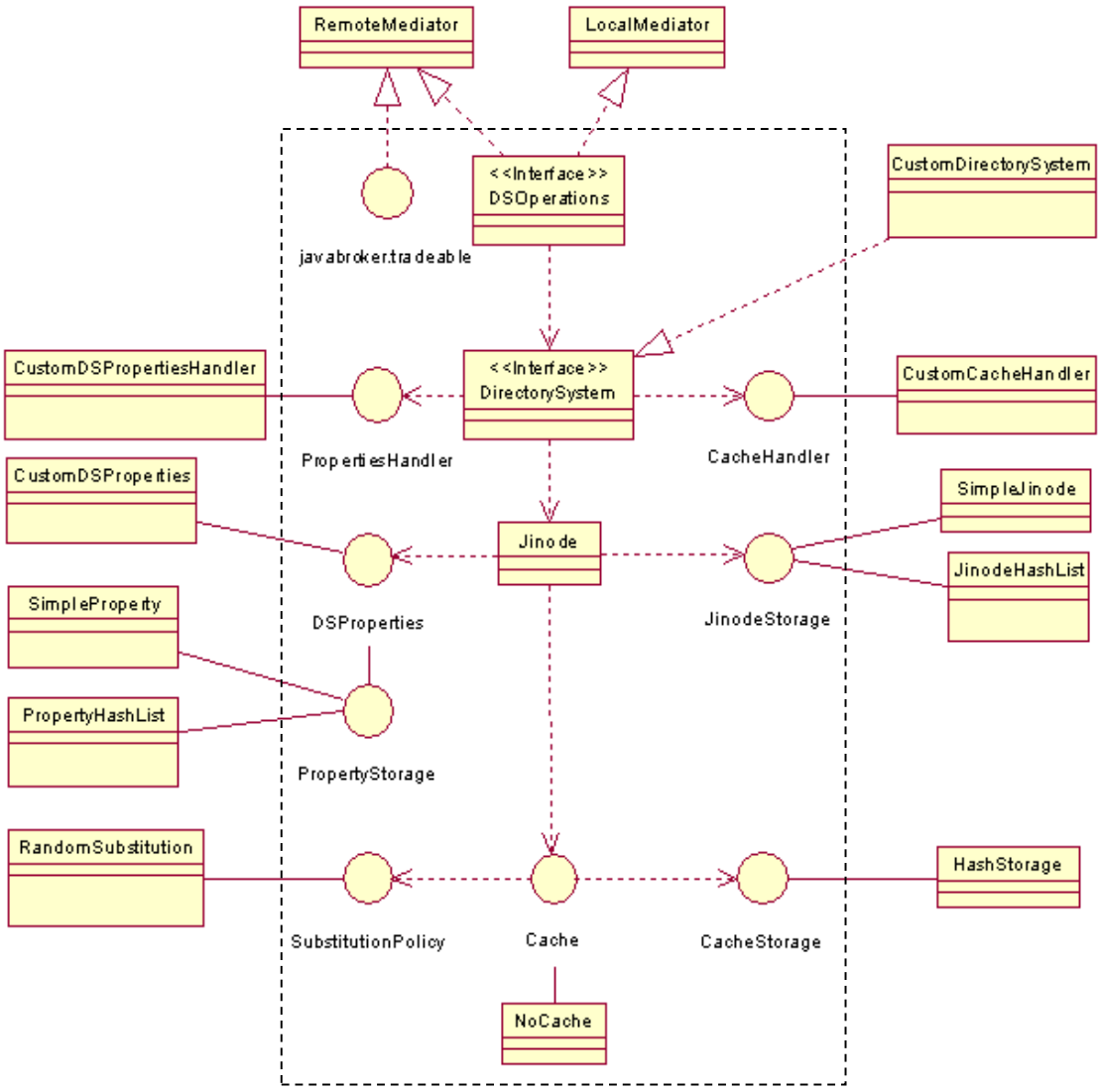

FIGURA 11.6: DIAGRAMA DE CLASSES DO *FRAMEWORK* JNDS

11.4.5 A Documentação do *Framework* JNDS

O *framework* JNDS é composto por dez interfaces e uma classe, além de utilizar uma interface do pacote jamp.objectservices.javabroker e oferecer treze outras classes que já implementam suas interfaces, tornando-o funcional. A seguir, é apresentada uma breve descrição de cada uma das estruturas que compõem o JNDS.

Jinode é a classe que representa os nós da árvore de diretórios. Seus objetos possuem duas instâncias de **JinodeStorage,** para representar seus nó pai e nós filhos, uma instância de **DSProperties** e também um **Cache**. Além das instâncias citadas, armazenam atributos como

seus nomes e contadores de uso, manipulados pelo **DirectorySystem**.

JinodeStorage é a abstração de um local para armazenamento de **Jinodes** e oferece métodos para armazená-los e recuperá-los. As classes **SimpleJinode** e **JinodeHashList** implementam esta abstração e oferecem as estruturas para armazenar, respectivamente, uma única instância **Jinode** e uma tabela *hash* deles.

DSProperties é a abstração encarregada de representar uma estrutura de armazenamento das propriedades e atributos de um diretório. É implementada pela classe **CustomDSProperties**, que possui atributos como data e hora de criação. Da mesma maneira como podem ser personalizados nos **Jinodes**, **DSProperties** permite a escolha da estrutura de dados para o armazenamento das propriedades. **PropertyStorage** define as funções básicas de uma estrutura de armazenamento para propriedades, enquanto que **SimpleProperty** e **PropertyHashList** implementam tais funções.

A outra instância presente nos **Jinodes** é o **Cache**, que auxilia nas buscas realizadas pelo sistema de diretórios. Da mesma forma como os modelos de *cache* convencionais, esta abstração possui duas outras instâncias, a **CacheStorage** e a **SubstitutionPolicy**. A primeira representa, da mesma forma como as interfaces de armazenamento anteriores, uma abstração de uma estrutura de dados para o armazenamento de referências de **Jinodes**. Já a segunda, define métodos que serão chamados quando uma referência do *cache* deve ser substituída. Vale ressaltar que o JNDS não define políticas de invalidação[64] do conteúdo do *cache*, pois como são armazenadas as referências para os objetos, sempre que um diretório sofrer uma modificação, o conteúdo do *cache* será automaticamente atualizado. As classes do *framework* que já implementam as interfaces sitadas são **HashStorage** e **RandomSubstitution**. A classe **NoCache** é uma classe interna ao *framework* que o faz operar sem a utilização de *cache*.

A interface **DirectorySystem** define a forma como **Jinode**, **Cache** e **DSProperties** se relacionam, além de realizar o controle e a manutenção da consistência da árvore de diretórios. Seu trabalho é auxiliado pelas interfaces **PropertiesHandler** e **CacheHandler**, que são, respectivamente, os manipuladores de propriedades e *caches* dos diretórios. Sempre que o sistema de diretórios necessita realizar uma operação sobre o *cache* ou sobre as propriedades de um dado diretório, faz o uso de seus manipuladores, caso existam.

CustomDirectorySystem é a implementação de **DirectorySystem** que acompanha o conjunto de classes do *framework* JNDS. Esta classe associa as implementações padrão **CustomDSPropertiesHandler** e **CustomCacheHandler** ao sistema de diretórios. A primeira

faz uso da implementação **CustomDSProperties**, que utiliza **SimpleProperty** como forma de armazenamento. Já a segunda, utiliza a implementação **NoCache**.

Por fim, a interface **DSOperations**, que define para as aplicações quais são os métodos e parâmetros aceitos pelo sistema de diretórios. A classe **LocalMediator** implementa as operações que são realizadas em instâncias locais de **DirectorySystem**, enquanto que a **RemoteMediator** cuida das implementações das operações realizadas remotamente.

11.4.6 A Disponibilização do pacote JNDS

Todas as classes e interfaces apresentadas na seção anterior encontram-se disponíveis no pacote de desenvolvimento da JAMP. A localização do pacote JNDS pode ser vista na figura 11.7.

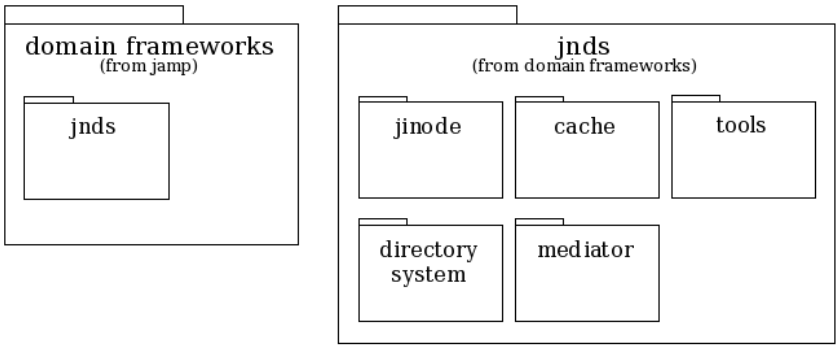

FIGURA 11.7: O PACOTE JNDS

Observado a figura anterior, além das estruturas descritas, o pacote JNDS possui um conjunto de ferramentas que auxiliam em sua utilização e execução. Entre elas, encontram-se classes que realizam a exportação e a importação das referências de objetos (do sistema de diretórios) entre as aplicações e o JBroker e vice-versa. Os demais subpacotes buscam oferecer o grau de personalização do *framework* citado por Taligent[63]. Caso o desenvolvedor deseje apenas criar uma aplicação que utilize o sistema de diretórios, precisará apenas fazer uso do pacote jamp.domainframeworks.jnds.directorysystem. Por outro lado, caso haja a necessidade de se definir políticas de *cache*, formas de armazenamento, políticas de substituição de *cache*, propriedades dos diretórios, estruturação lógica da árvore ou formas de compartilhamento diferentes das padrões, o desenvolvedor deve redefinir apenas o que lhe convier. A figura 11.8 ilustra uma simples aplicação distribuída que faz uso do pacote JNDS.

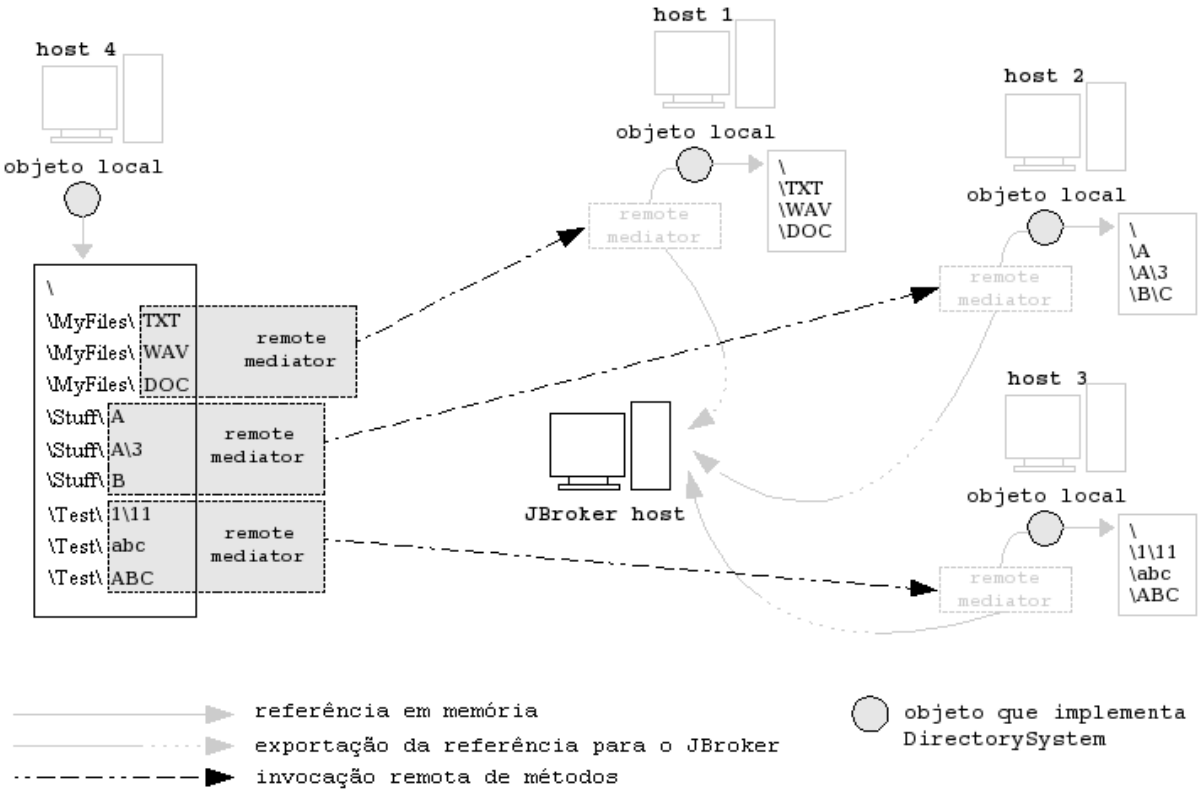

FIGURA 11.8: EXEMPLO DE UTILIZAÇÃO DO JNDS

No exemplo acima, as aplicações servidoras que executam nos *hosts* 1, 2 e 3 exportam as referências de seus objetos que implementam o sistema de diretórios para o JBroker. A aplicação cliente do *host* 4, por sua vez, faz uso das referências exportadas, construindo um sistema de diretórios próprio, porém, completamente distribuído. Vale notar que a aplicação cliente do *host* 4 também pode exportar seu objeto local para o JBroker, tornando-se um quarto servidor de diretórios.

11.5 A implementação do Diretório de Ambientes de Visualização Distribuída

A implementação desse serviço de diretórios que é utilizado pela OpenReality levou em conta as propriedades apresentadas na tabela 9.1 e o *framework* JNDS.

Utilizando-se do reuso de projetos[39], a implementação desse serviço foi facilitada, ao restringir-se apenas à modificação do comportamento do *framework* JNDS. Entre as classes e interfaces desse *framework*, como já foram vistas nas seções anteriores, encontra-se a interface **jnds.properties.directorysystem.DSPropertiesHandler**. O código fonte que implementa essa interface foi a única implementação necessária para a confecção do serviço

desejado. Facilitando inda mais o trabalho de desenvolvimento, optou-se pela redefinição da classe **CustomDSPropertiesHandler**, que já implementa todos os métodos definidos pela interface acima.

Assim, as únicas linhas de código necessárias para a implementação do Diretório de Ambientes de Visualização Distribuída foram as exibidas na figura 11.9.

```
/*
 * DVEDirectory.java
 * Created on October 27, 2003, 9:08 AM
 */

package jnds.directorysystem.properties;

/**
 *
 * @author   Bruno do Amaral Dias Baptista
 */
public class DVEDirectory extends CustomDSPropertiesHandler {
  public static String[][] PROPERTIES = {
    { "TYPE", "Entry type (MEMBER or GROUP)" },
    { "CREATION_DATE", "The entry creation date and hour" },
    { "IP_ADDRESS", "The IP address of the member" },
    { "IP_PORT", "The port number used to receive data" },
    { "PROTOCOL", "The protocol used by the members" },
    { "DESCRIPTION", "Description of the entry" },
    { "STATUS", "Status of the entry" },
    { "FUNCTION", "Not used" };
  };
}
```

FIGURA 11.9: IMPLEMENTAÇÃO DO DIRETÓRIO DE AVDS COM JNDS

Dessa maneira foi implementado o objeto de serviço JAMP que executa as tarefas do Diretório de AVDs proposto. A confecção dos *proxies* necessários para a sua integração com as estruturas da Camada de Representação e Síntese foram implementadas com a utilização da ferramenta JAMP2C, que é apresentada no próximo capítulo.

11.6 Conclusões

Este capítulo apresentou o projeto do *framework* JNDS, proposto para oferecer, além dos serviços necessários para a arquitetura OpenReality (Diretório de AVDs), mais uma gama de recursos para a Plataforma JAMP.

O JNDS, disponibilizado como um *framework* de domínio dentro da arquitetura, oferece facilidades para desenvolvedores criarem seus próprios sistemas de diretórios. A implementação do Diretório de AVDs da arquitetura OR mostrou que isso é possível. Além

disso, pôde-se perceber as vantagens que o reuso de projeto de *software* oferece. Entre elas, podem ser citadas a diminuição de complexidade, tempo de desenvolvimento, manutenção e testes, e, obviamente, do custo do desenvolvimento como um todo.

12. O Compilador JAMP2C

Diante da estratégia de comunicação entre aplicações nas linguagens Java e C/C++ apresentada no capítulo 8, a confecção de uma ferramenta de auxílio para o processo de desenvolvimento de *software* que a seguisse tornou-se vantajosa. Seu objetivo é reduzir o custo do desenvolvimento das aplicações de visualização distribuída sobre a arquitetura OpenReality, visto que a estratégia de comunicação adotada, entre as estruturas de seu *framework* e os serviços da JAMP, exige a implementação de um volume relativamente grande de código. Moldando-se nas funcionalidades oferecidas pelas ferramentas idl2java[65] e idlj[66], a ferramenta gera, a partir de uma especificação de entrada, saídas que agilizam o processo de desenvolvimento das aplicações em questão, atingindo o objetivo esperado. Este capítulo apresenta o projeto dessa ferramenta, o compilador JAMP2C.

12.1 A Tarefa do Compilador

Analisando-se as especificações e os requisitos das estruturas da arquitetura OR, nota-se a comum característica da aplicação em C/C++ sempre requisitar os recursos disponíveis na JAMP, oferecidos através de seus objetos. Desta maneira, foi possível o desenvolvimento de um *software* que, automaticamente, construísse as estruturas sempre presentes nas aplicações que seguem a estratégia apresentada, ou seja, a confecção dos *proxies* Java e C/C++. Como dados de entrada para o *software* desenvolvido, foi considerada a utilização da IDL[67] como em idl2java e idlj, ou até mesmo uma variação desta. Porém, como o escopo de utilização desta ferramenta sempre envolve a JAMP e Java RMI, torna-se mais interessante a utilização da própria linguagem Java como código de entrada.

De acordo com Aho[68], todo *software* capaz de analisar a exatidão de um código de entrada e gerar um código de saída também correto realiza o processo de compilação. Sendo assim, a ferramenta de auxílio desenvolvida é o compilador chamado JAMP2C.

Embora o nome atribuído ao compilador desenvolvido passe a impressão de transformar um código de objeto JAMP em códigos na linguagem C, também são gerados códigos em C++. A figura 12.1 apresenta as tarefas que o compilador JAMP2C realiza.

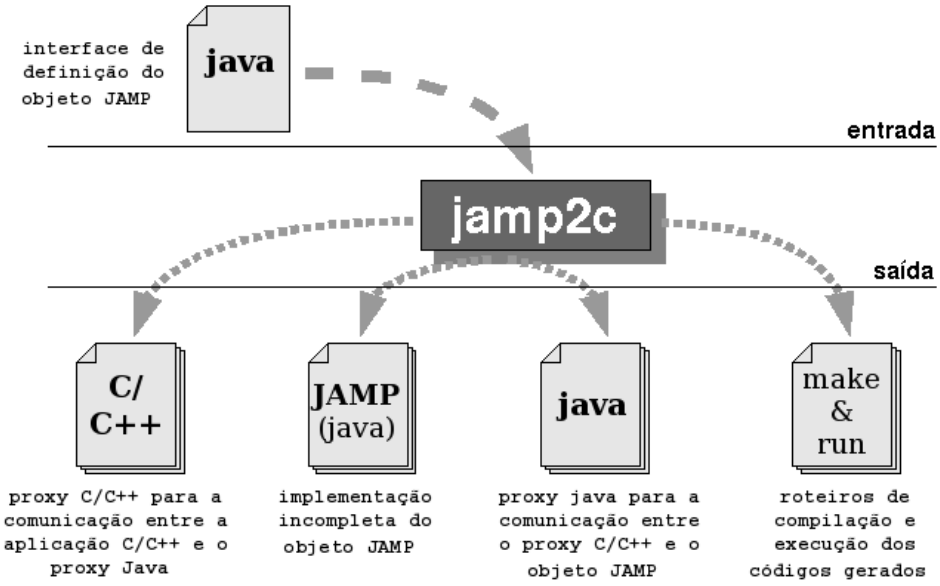

FIGURA 12.1: O TRABALHO DO COMPILADOR JAMP2C

Com a finalidade de reduzir as nomenclaturas utilizadas neste texto, passar-se-á a chamar de objeto JAMP todo objeto de serviço presente na Plataforma JAMP, seja ele um objeto de serviço ou um objeto que implemente algum *framework* de domínio. Pode-se notar que, dado um objeto JAMP, os códigos a serem gerados estão diretamente relacionados aos métodos que ele oferece, pois, de acordo com a estratégia já apresentada, os *proxies* realizam apenas as operações de montagem e desmontagem das séries de dados que devem ser trocadas entre eles. Sendo assim, buscando minimizar ainda mais o custo do processo de desenvolvimento, o código a ser utilizado como entrada pelo compilador será o mesmo código Java que define a interface de um objeto JAMP.

Tomando como base o modelo de construção de compiladores orientados a objeto proposto por Appel[69] e adaptando algumas de suas fases da análise, a estrutura do compilador foi definida como mostra a figura 12.2.

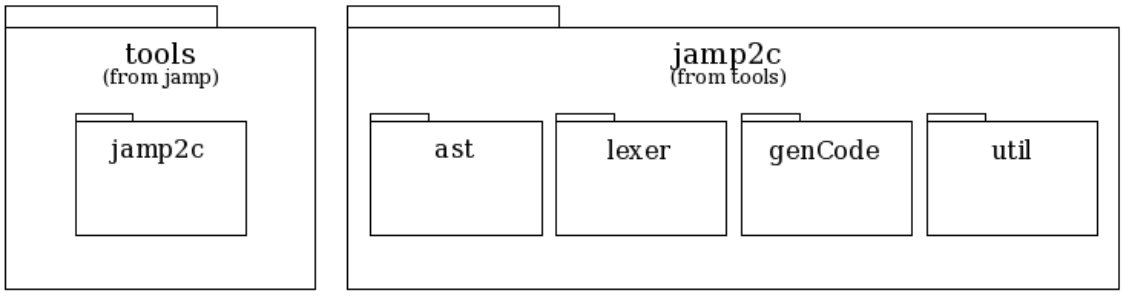

FIGURA 12.2: O PACOTE JAMP2C

12.2 Os Analisadores Léxico, Sintático e Semântico

O analisador léxico (jamp2c.lexer) criado segue a proposta de Appel[69] e realiza o reconhecimento do subconjunto, apresentado na tabela 12.1, das palavras reservadas da linguagem Java. Este subconjunto possui somente os lexemas que o compilador javac[66] aceita como corretos em uma declaração de interface válida na linguagem Java[70]. Sua tarefa é identificar cada um dos lexemas e disponibilizá-los para o analisador sintático. Além disso, realiza as tarefas de ignorar as formas sintáticas dos comentários permitidos pela linguagem Java, bem como os caracteres de tabulação de texto, como espaço em branco, paragrafação, fim de linha e fim de arquivo. O reconhecimento de comentários foi atribuído como tarefa do analisador léxico, para que a análise sintática trate única e exclusivamente da avaliação da exatidão sintática das sentenças do código de entrada.

Tabela 12.1: Lexemas reconhecidos pelo Compilador JAMP2C

boolean	int	protected
char	float	public
extends	package	String
import	private	throws

Pode-se notar, a partir dos lexemas da tabela anterior, que apenas um subconjunto dos tipos básicos da linguagem Java são considerados. Arranjos, matrizes e objetos de classes definidas pelo usuário ainda não são aceitos, pois requerem um estudo detalhado de como realizar suas conversões em tipos suportados pelas linguagens C e C++.

Já o analisador sintático, que segue a técnica *top-down*, implementando um analisador recursivo descendente[68], foi gerado manualmente, a partir do subconjunto da gramática que define a linguagem Java[70]. As regras de produção dessa gramática são apresentadas na figura 12.3. Durante sua execução, o compilador utiliza o objeto jamp2c.Compiler para montar uma representação intermediária do código analisado, a árvore de sintaxe abstrata que é representada pelo pacote jamp2c.ast. Entre as verificações que esse analisador realiza, estão as checagens de sobrecargas de métodos, de suas visibilidades e, principalmente, da declaração e da utilização dos pacotes e exceções que devem estar presentes nos códigos de declaração dos objetos JAMP.

Não há necessidade de ser realizada a análise semântica, pois as interfaces Java não possuem características semânticas quando analisadas isoladamente, embora a verificação de tipagem de dados deva ser realizada no reconhecimento dos parâmetros aceitos por cada

método. As questões referentes à coerção de tipos seguem a mesma relação adotada na linguagem Java e deve ser levada em consideração durante a verificação de sobrecarga de métodos. A declaração de métodos distintos, porém com assinaturas coercíveis, é considerada inválida, pois causa indeterminismo no sistema de tempo de execução (JVM).

```
Programa ::= Pacote Lista_de_Importações Declaração_de_Interface

Pacote ::= package Caminho_Completo ;
         | E

Lista_de_Importações ::= import Caminho_Completo ; Lista_de_Importações
                       | import Caminho_Inicial ; Lista_de_Importações
                       | E

Declaração_de_Interface ::= Visibilidade identificador Extensão { Métodos }

Caminho_Completo ::= Caminho_Completo.identificador
                   | identificador

Caminho_Inicial ::= identificador.Caminho_Inicial
                  | identificador.*

Visibilidade ::= public
               | private
               | protected
               | E

Extensão ::= extends Caminho_Completo

Métodos ::= Método Métodos
          | Método

Método ::= Visibilidade Tipo identificador ( Parâmetros ) Exceções ;

Exceções ::= throws Exceção

Exceção ::= Caminho_Completo, Exceção
          | Caminho_Completo

Tipo ::= int
       | float
       | boolean
       | char
       | String

Parâmetros ::= Parâmetro , Parâmetros
             | Parâmetro

Parâmetro ::= Tipo identificador
```

FIGURA 12.3: GRAMÁTICA VERIFICADA PELO COMPILADOR JAMP2C

Ao final da fase de análise, o compilador possui uma representação intermediária do código de entrada, a árvore de sintaxe abstrata (ASA). Foi utilizada uma abordagem orientada

a objetos para a representação das estruturas que compõem a ASA, o que facilitou sua manipulação durante a fase de síntese. A organização dos objetos que compõem a ASA pode ser vista na figura 12.4.

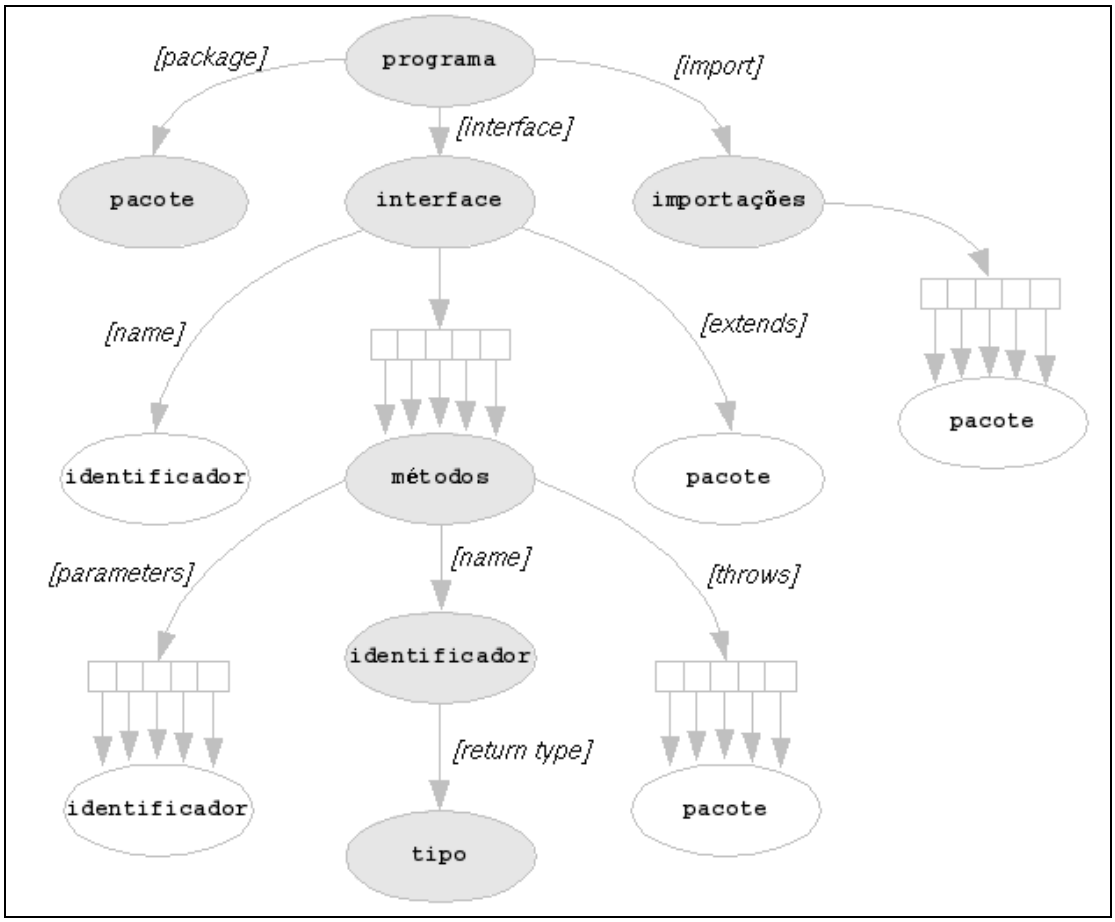

FIGURA 12.4: ÁRVORE DE SINTAXE ABSTRATA DO COMPILADOR JAMP2C

Durante a fase de análise são realizadas, além das verificações léxicas, sintáticas, de utilização dos pacotes da JAMP e declarações de exceções[70], algumas das checagens que o compilador rmic[66] também realiza. Embora possa parecer redundância, algumas dessas verificações ajudam a antecipar a detecção de erros comuns de programação, tornando o processo de desenvolvimento mais ágil.

12.3 Os Sintetizadores de Código

Encerradas as análises, é iniciada a fase de síntese do compilador JAMP2C. Percorrendo-se hierarquicamente a estrutura de representação intermediária construída na fase

anterior, os códigos dos *proxies* Java e C/C++ são criados. A figura 12.5 apresenta alguns esquemas dos trechos dos códigos gerados pela ferramenta. Os objetos das classes que compõem a ASA implementam os métodos `genC()`, `genJava()` e `genJAMP()`, já de forma hierárquica. Assim, chamando-se esses métodos a partir do objeto raiz da ASA, os códigos são completamente gerados, sem que o sistema precise percorrer a estrutura da árvore de sintaxe abstrata. Além dessas facilidades, objetos auxiliares foram criados para tratarem de questões específicas da geração de código, como a manipulação dos arquivos, a identificação dos caracteres especiais dependentes do sistema operacional e o alinhamento do texto do código gerado, permitindo, inclusive, a configuração das tabulações e espaçamentos seguindo o estilo de digitação do desenvolvedor.

Módulos para a geração de códigos adicionais e manipulação de algumas opções de geração, de forma a auxiliar ainda mais o desenvolvedor foram criados. Entre os códigos adicionais, estão a geração do próprio objeto JAMP (caso ele ainda não exista) e dos roteiros de compilação e de execução para os códigos gerados. Já entre as opções que podem modificar a forma de geração, encontram-se parâmetros referentes aos protocolos envolvidos na estratégia de comunicação e escolha do sistema operacional que receberá os códigos gerados. Além dessas, funcionalidades como realização da compilação em modo silencioso, *debug* e *verbose* também foram disponibilizadas.

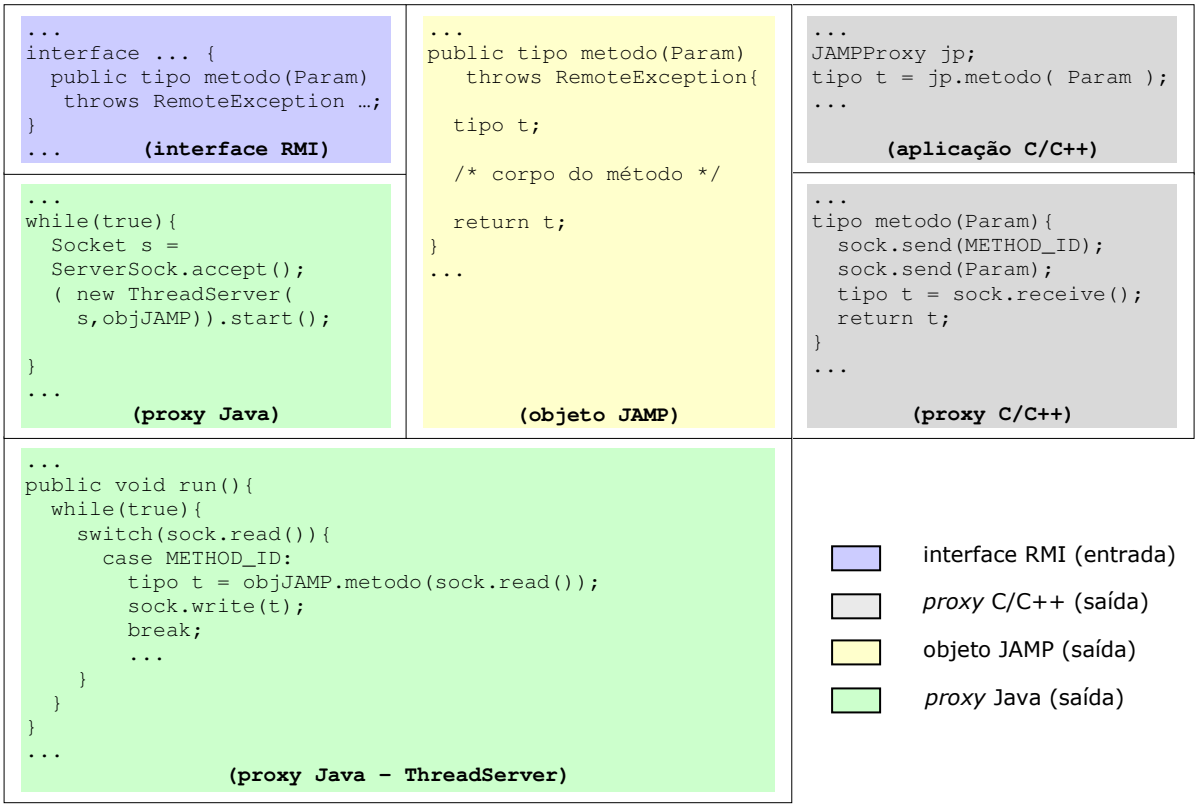

FIGURA 12.5: ESQUEMA DE GERAÇÃO DE CÓDIGO DO COMPILADOR JAMP2C

Vale notar que o *proxy* Java é gerado de forma a suportar múltiplas *threads*, o que o permite atender requisições de múltiplos *proxies* clientes simultaneamente. O esquema apresentado oculta o processo de ordenação e de decomposição da série de dados, trocadas pela rede, (*marshalling* e *unmarshalling*) dos parâmetros e retornos dos métodos criados, apresentando apenas a parte da lógica que é utilizada durante a geração de código. Cada uma das *threads* do *proxy* Java possui o mesmo tempo de vida que seu par *proxy* C/C++. Além dessas características, o código gerado também opera de forma assíncrona para os métodos que não possuem valor de retorno, já que assumem uma postura síncrona para os demais métodos.

12.4 Desenvolvimento de Aplicações com o JAMP2C

Independentemente do paradigma de desenvolvimento de *software* adotado para o processo de desenvolvimento da aplicação, o compilador JAMP2C atua a partir da especificação, já na linguagem Java, do objeto JAMP. Assim, com uma ferramenta CASE que permita a modelagem de sistemas em UML[56], o desenvolvedor obtém automatizadamente todos os códigos que definem as classes e as interfaces de seu modelo, além dos demais itens do sistema modelado. A partir da interface gerada para o objeto JAMP, o compilador JAMP2C cria a implementação do objeto JAMP e os *proxies* Java e C/C++ automaticamente. Vale notar que os objetos JAMP gerados nesse processo são implementações funcionais, porém sem a lógica desejada pelo desenvolvedor. Suas implementações possuem o corpo de cada método declarado, incluindo os comandos de retorno de valores. Isso foi assumido para que, mesmo sem a conclusão da implementação do objeto JAMP, o sistema possa ser completamente compilado e testado. Assim, o desenvolvedor pode criar protótipos funcionais das aplicações tão logo quando possuir a definição da interface do objeto JAMP. Além disso, caso faça uso dos roteiros de compilação e execução também gerados automaticamente pelo compilador, o desenvolvedor passa a ter um ambiente de desenvolvimento, ainda que não seja unificado, que lhe permite modelar o sistema em UML, preencher o corpo dos métodos da implementação do objeto JAMP e executar o sistema para realizar testes e validações, podendo, inclusive, utilizar alguma outra ferramenta CASE que os automatizem. A figura 12.6 apresenta os passos do desenvolvimento de *software* auxiliados pela atuação da ferramenta.

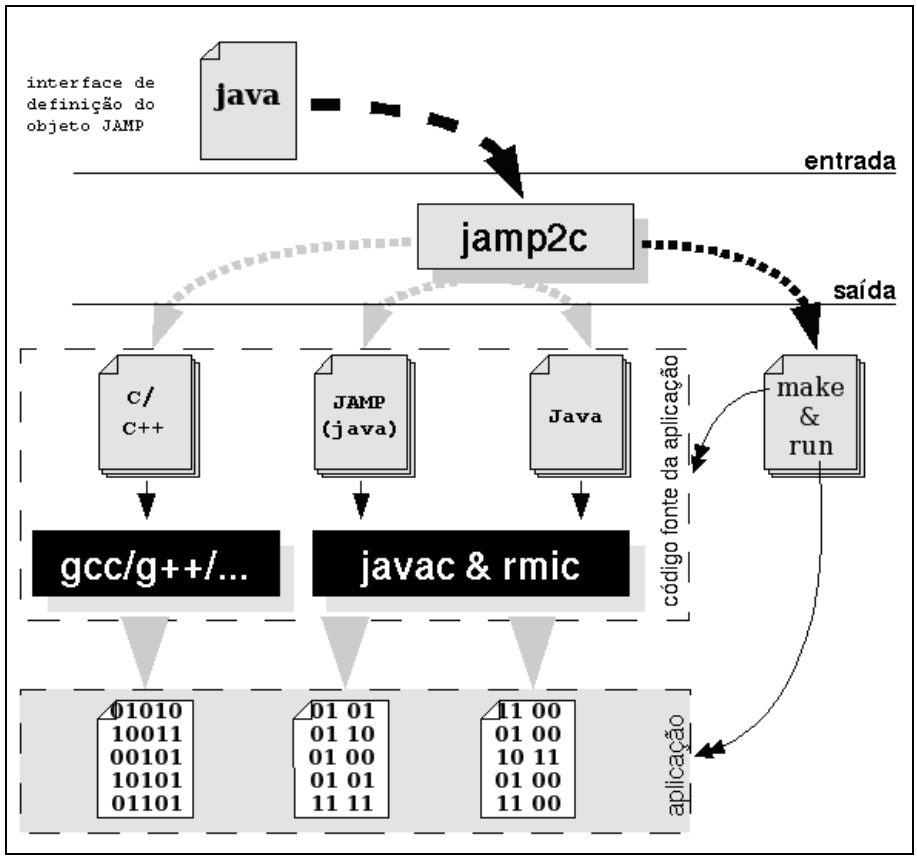

FIGURA 12.6: ROTEIROS DE COMPILAÇÃO E EXECUÇÃO GERADOS AUTOMATICAMENTE

A figura 12.7, por sua vez, mostra como o compilador JAMP2C foi disponibilizado dentro do pacote de ferramentas da JAMP. Atualmente, o pacote jamp.tools.jamp2c possui apenas o *back-end* do compilador, podendo ser executado via linha de comandos ou ser instanciado por alguma aplicação que cuide apenas da interface a ser exibida para o usuário (desenvolvedor).

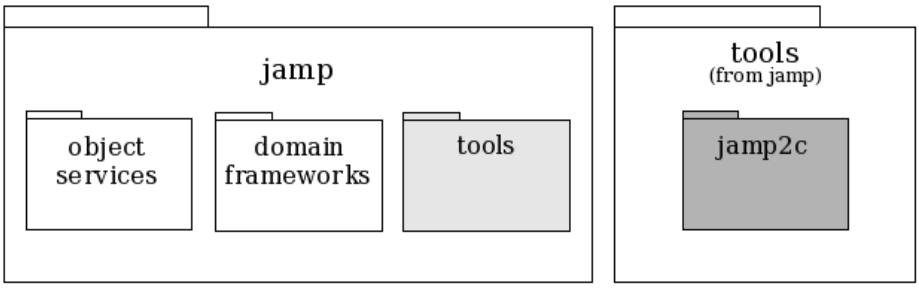

FIGURA 12.7: JAMP2C NO PACOTE DE DESENVOLVIMENTO JAMP

12.5 Considerações Finais

Este capítulo apresentou o compilador JAMP2C, desenvolvido para reduzir o custo do desenvolvimento de aplicações que utilizam a estratégia apresentada como forma de comunicação entre aplicações implementadas nas linguagens C/C++ e Java. Foi utilizada neste trabalho para auxiliar no projeto do Diretório de Ambientes de Visualização Distribuída, Repositório de Travas e do OR_Multicast. Em todas essas implementações, o compilador JAMP2C realizou plenamente as tarefas que deveria, gerando códigos corretos e funcionais para cada um dos *proxies* criados.

Durante o desenvolvimento desses projetos, ao utilizar-se o compilador JAMP2C pôde-se perceber o ganho de produtividade que essa ferramenta oferece ao processo de desenvolvimento de *software*.

13. Uma aplicação de Visualização Distribuída

Como forma de testar e avaliar o que foi proposto neste trabalho, escolheu-se uma aplicação que abrangesse o maior número de estruturas apresentadas. Com este intuito, a aplicação desenvolvida para ser utilizada como um estudo de caso distribui um modelo tridimensional para outros participantes. O modelo distribuído pode ser manipulado por qualquer aplicação participante do ambiente.

Embora seja uma aplicação relativamente simples, requer a utilização de praticamente todos os recursos oferecidos pela arquitetura OpenReality. Ao desenvolver uma aplicação deste tipo, as estruturas, desde o Diretório de Ambientes, até o Gerenciador de Distribuição podem ser testadas, além, é claro, do Repositório de Travas, do Gerenciador de Comunicação e dos protocolos OR. Vale lembrar que, para o funcionamento dos serviços presentes na Plataforma JAMP, como o Repositório de Travas, o Diretório de Ambientes e o Gerenciador de Grupos Multicast, serão utilizados os *proxies* gerados pelo compilador JAMP2C. Além disso, no caso particular do Diretório de Ambientes, o próprio *framework* JNDS estará sendo testado.

Portanto, o objetivo da implementação dessa aplicação é a validação das estruturas, modelos e estratégias adotadas durante a realização deste trabalho. Vale observar que as avaliações de desempenho e comparações com outras arquiteturas não são cobertas por esse experimento.

13.1 Apresentação

A aplicação desenvolvida possui um modelo tridimensional de uma mão humana. A aplicação que cria o ambiente possui uma luva de dados que manipula esse modelo. As demais aplicações que ingressam no ambiente também podem manipular o modelo, porém utilizando seus teclados. O objetivo desse sistema é experimentar o compartilhamento do modelo para a comprovação da interação entre os participantes.

13.2 Modelagem e Estruturação

A modelagem da aplicação descrita na seção anterior foi estruturada da forma como mostra a figura 13.1. Os retângulos com linhas contínuas representam a Camada de Suporte aos Ambientes de Visualização Distribuída, enquanto os com linhas pontilhadas, a Camada de Serviços.

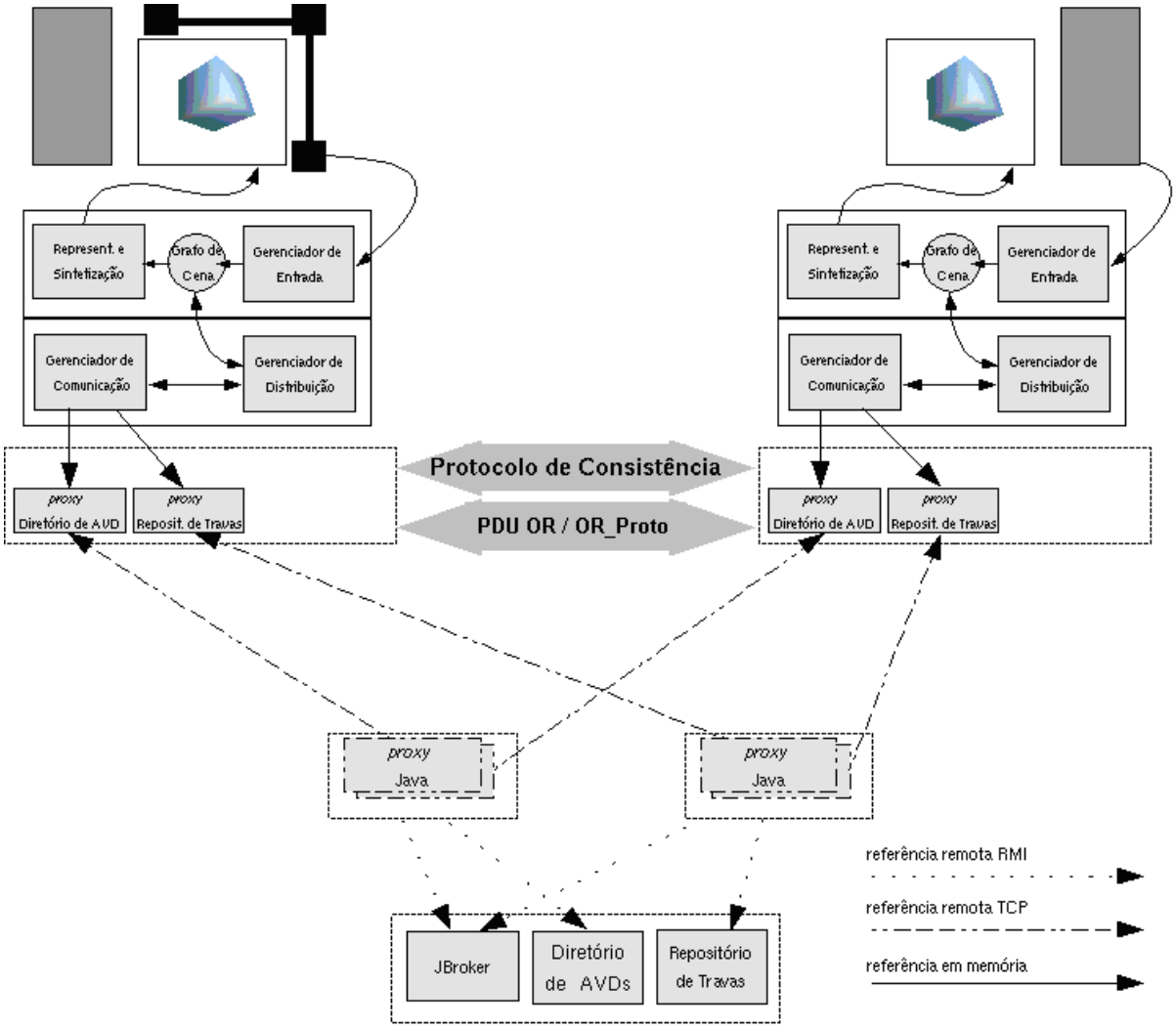

FIGURA 13.1: MODELAGEM E ESTRUTURAÇÃO DA APLICAÇÃO

13.3 Implementação

Começando pela Camada de Serviços, o Diretório de Ambientes foi implementado utilizando-se o *framework* JNDS. Já sua comunicação com a aplicação OR foi realizada através da geração automática dos *proxies*, com a ferramenta JAMP2C. O mesmo ocorreu

com o Repositório de Travas. Os *proxies* C++, por sua vez, foram compilados juntamente com todas as estruturas da Camada de Suporte aos Ambientes de Visualização Distribuída. O protocolo de comunicação adotado para transferir as PDUs OR foi o OR_TCP.

Já na Camada de Suporte aos Ambientes de Visualização Distribuída, além dos gerenciadores de Distribuição e de Comunicação, foram redefinidos todos os objetos pertencentes ao grafo de cena, como objetos das classes que herdam de DistributedSgObject. Além disso, foi implementado um InputSensor adicional, o GloveInputSensor, para a manipulação da luva de dados.

13.4 Execução

Após a compilação de todos os códigos envolvidos na aplicação, a execução iniciou-se pelo JBroker, e, em seguida, os serviços presentes na JAMP. Após o início da execução do Serviço de Diretórios de AVDs e do Repositório de Travas, foram iniciados os *proxies* Java referentes a cada um deles. Por fim, as aplicações propriamente ditas. Primeiramente, a que possuía a luva de dados criou o grafo de cena, o diretório do ambiente e cadastrou todas as travas no repositório. Em seguida, a segunda aplicação participante ingressou no ambiente, conseguindo sua cópia do grafo de cena após consultar os dados presentes no Diretório de AVDs.

13.5 Resultados

Entre os resultados alcançados, destacam-se dois pontos principais. O primeiro refere-se ao funcionamento da arquitetura, que teve seu fluxo de dados ocorrendo da maneira como foi planejado. O segundo refere-se à utilização do *framework* OpenReality, que atendeu às expectativas em relação à facilidade de utilização.

Ainda com relação ao primeiro ponto observado, pode-se concluir que o compilador JAMP2C realmente auxilia no desenvolvimento de aplicações que seguem a estratégia apresentada para a comunicação entre as linguagens C/C++ e Java. Antes de ser utilizado nessa implementação, ele foi testado em várias outras situações, que serviram como fase de testes e manutenção. Diante dos resultados apresentados com suas utilizações, pode-se afirmar que os códigos foram gerados de maneira correta, pois gerou tanto os códigos fonte quanto os scripts de compilação corretamente em todos os testes aos que foi submetido.

Da mesma forma, o *framework* JNDS também passou por uma bateria de testes antes de ser utilizado como um serviço para a arquitetura OR. Em todos os casos atendeu aos requisitos e manteve seus dados consistentes.

Quanto ao *framework* OpenReality propriamente dito, obteve um rendimento satisfatório, não aparentando atrasos durante o ingresso das novas aplicações no ambiente. Porém, embora os resultados aparentem ser animadores, testes mais rigorosos devem ser aplicados para cada um dos possíveis protocolos disponíveis na OR e para as diferentes topologias de rede, monitorando sempre o consumo de largura de banda. Dessa maneira, poderá ser estimar o número máximo de participantes que a arquitetura suporta para cada modalidade de comunicação permitida. A figura 13.2 ilustra momentos dos experimentos.

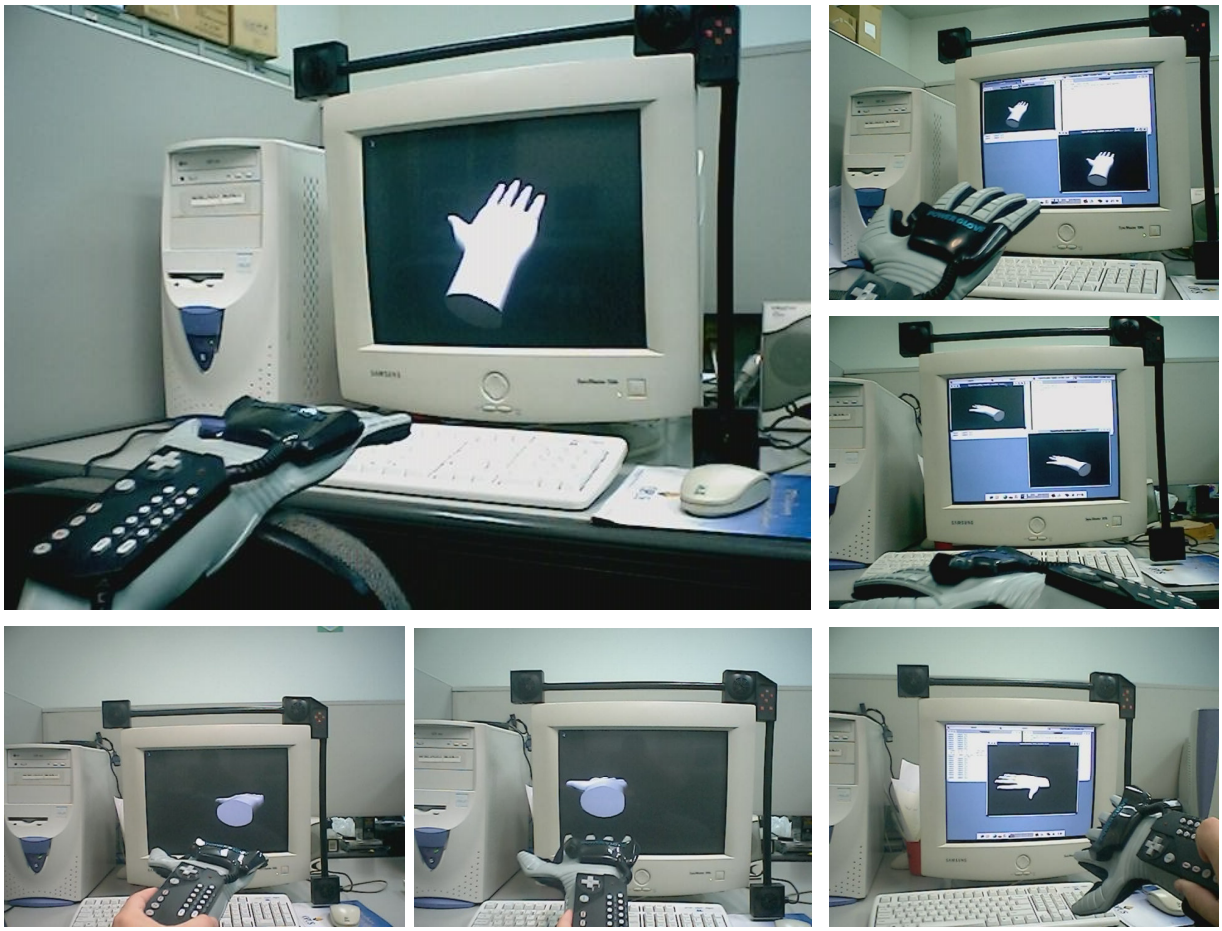

FIGURA 13.2 EXECUÇÃO DA APLICAÇÃO IMPLEMENTADA

Portanto, pode-se considerar validado o funcionamento do modelo de distribuição adotado e a forma como ele foi implementado, embora ainda seja necessária uma avaliação comparativa entre a OR e as demais arquiteturas. Somente através de uma comparação desse tipo pode-se determinar quais as reais vantagens e os pontos frágeis dessa arquitetura.

14. Conclusões

Embora os objetivos propostos por este trabalho tenham sido atingidos, este não se encontra encerrado. Durante o período dedicado ao desenvolvimento de cada um dos módulos, que cabiam a este trabalho, da arquitetura OpenReality, muitas outras questões e possibilidades de estudo foram encontradas. Algumas delas já eram previstas desde seu início, mas grande parte das restantes abriu um leque de novas possibilidades de pesquisa. Este capítulo apresenta as conclusões sobre este trabalho.

14.1 Considerações sobre o trabalho

Foram apresentadas as estruturas e técnicas utilizadas no projeto do Subsistema de Comunicação e Distribuição e da Camada de Serviços da arquitetura OpenReality. O primeiro requisitou, além da criação dos gerenciadores de Distribuição e de Comunicação, a adaptação e modificação das classes do grafo de cena, tornando possível a sua representação como série de *bytes* e seu envio pela rede de dados. Também foi necessária a especificação do mecanismo de travas, que garante a consistência absoluta do ambiente. Já a camada de Serviços foi adaptada para suportar tanto o Diretório de Ambientes de Visualização Distribuída quanto o mecanismo de travas especificado. Em ambos os casos, a interoperabilidade entre as aplicações nas linguagens Java e C++ foi imprescindível, assim como na utilização do serviço de Comunicação *Multicast* presente na JAMP.

O Gerenciador de Distribuição, maior responsável pelas alterações ocorridas no grafo de cena, definiu um mecanismo de distribuição das informações de atualização de estado muito parecido com o modelo de Chamadas Remotas de Procedimento. Isso simplificou significativamente a compreensão do *framework* OR, o que representa uma característica importante e que deve ser levada em consideração durante o projeto de um *framework*.

A estratégia adotada para a realização da comunicação entre as duas linguagens em questão foi adotada após a realização de estudos preliminares e de testes comparativos com

outros modelos. Sua definição acabou motivando o projeto e a implementação de um compilador, capaz de auxiliar no processo de desenvolvimento de aplicações distribuídas.

De uma maneira geral, o trabalho aproxima a arquitetura OpenReality da situação idealizada pela tabela 3.4 do capítulo 3. Claramente, a confecção de uma arquitetura que oferecesse todas as características presentes nesta tabela não estava entre os objetivos deste trabalho, embora ela continue sendo uma das metas dos projetos que envolvem a OR.

Neste sentido, este trabalho realizou as alterações previstas desde o início de seu projeto, que permitem a criação de Aplicações de Visualização Distribuída a partir do *framework* OpenReality. O modelo de distribuição de informações utilizado baseou-se nos estudos realizados sobre os Ambientes Virtuais Distribuídos e suas formas de comunicação. A tabela 14.1 apresenta a situação real da arquitetura OpenReality.

Tabela 14.1: Situação atual da Arquitetura OpenReality

	Atual	**Ideal**
Propósito	Geral	geral
Número de usuários	n/d	n/d
Modelo de dados	mundo homogêneo e replicado	mundo homogêneo e replicado
Protocolo de Comunicação	*Broadcast, multicast, unicast*	*broadcast, multicast, unicast*
Nível de Controle sobre o ambiente	fino / grafo de cena	fino / grafo de cena **grosso / estado dos objetos**
Organização	ponto a ponto / grupo	ponto a ponto / grupo
Replicação de dados	Total	total
Consistência	forte / prot. de distribuição com travas de objetos	forte / prot. de distribuição **fraca / reg. freq. de estado**
Dead-Reckoning	não	**sim**

Como pode ser observado na tabela anterior, o *framework* ainda não permite a utilização das técnicas de *Dead-Reckoning* e nem de regeneração freqüente de estado. Outra característica que ainda não pode ser afirmada é referente ao número de usuários que a arquitetura suporta, pois, além de ser permitida a utilização de modalidades distintas de comunicação, o que influencia diretamente na escalabilidade da aplicação, não foram realizados testes suficientes que avaliassem todas as combinações dos protocolos com as topologias de rede. Esses são possíveis temas para trabalhos futuros.

14.2 Contribuições Acadêmicas

A conclusão dos mecanismos de distribuição da arquitetura OpenReality e sua disponibilização como um *framework* de domínio público oferece uma alternativa para o desenvolvimento de Aplicações de Visualização Distribuída. Além das arquiteturas já apresentadas no capítulo 3, aplicações de Visualização Científica[71], Medicina, Simulação e Treinamento, Engenharia, Educação, Entretenimento e outras podem contar com a OR para serem desenvolvidas.

Ao mesmo tempo em que a arquitetura OpenReality foi ampliada, a JAMP também sofreu atualizações significativas. As dificuldades encontradas na integração das estruturas utilizadas também trouxeram ganhos para a JAMP, como a criação do novo *framework* de domínio JNDS e do compilador JAMP2C, que pode auxiliar no processo de desenvolvimento de *software* quando se utiliza essa arquitetura.

O compilador JAMP2C oferece à JAMP a possibilidade de suportar aplicações clientes na linguagem C e C++. Embora aparentemente isso contrarie os princípios de portabilidade da JAMP, na verdade apenas aumenta as possibilidades de utilização da plataforma, pois somente os clientes dos serviços JAMP podem ser escritos em C/C++.

Outra funcionalidade deste compilador, mesmo que secundária, é sua atuação como ferramenta de auxílio no desenvolvimento de aplicações distribuídas, sejam elas em C/C++ e Java ou puramente em Java. Como o compilador oferece a criação do objeto JAMP no caso deste ainda não existir, essa opção pode ser explorada de forma a compor um ambiente de desenvolvimento, juntamente com uma ferramenta de modelagem UML.

Essas e outras características fazem diminuir o custo do desenvolvimento de *software* orientado a objeto quando se utiliza a JAMP como plataforma de distribuição, reduzindo, assim, um pouco da sua disparidade frente à utilização de outras tecnologias de *middleware*.

O Sistema de Diretórios Distribuídos da JAMP, por sua vez, oferece um novo *framework* de domínio para a JAMP, ampliando ainda mais sua base de suporte à aplicações distribuídas. O serviço de diretórios distribuídos, que raramente é encontrado da forma como foi especificado neste trabalho, é flexível às personalizações, podendo atrair a atenção de desenvolvedores para a arquitetura em que se encontra.

Também como um efeito colateral de sua proposta, o JNDS pode ser utilizado como forma de comunicação entre processos e entre aplicações, pois permite o compartilhamento de informações de forma transparente. A figura 14.1 apresenta o estado atual da Plataforma JAMP.

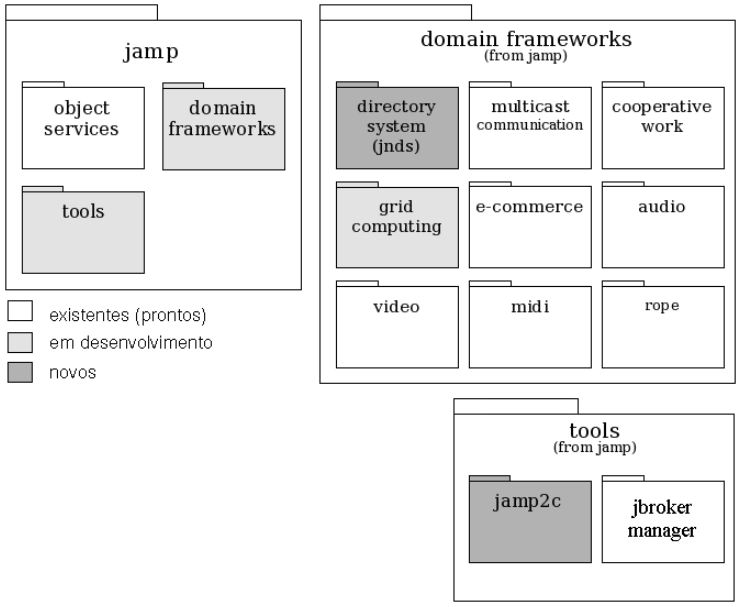

FIGURA 14.1: ESTADO ATUAL DA JAMP

Durante a realização deste trabalho, os artigos ***"OpenReality: An Architecture for Distributed 3D Visualization"***[1] e ***"Representação e Distribuição no Framework OpenReality***"[72] foram publicados. Além desses, outros que apresentam os resultados obtidos e discutem algumas das propostas de trabalhos futuros estão sendo escritos.

Por fim, dado que a arquitetura OpenReality provê os mecanismos necessários para a construção de Aplicações de Visualização Distribuída, é esperado que, a partir deste trabalho, novas oportunidades de pesquisa sejam proporcionadas nas áreas de Sistemas Distribuídos, Realidade Virtual e Processamento de Alto Desempenho dentro do Programa de Pós Graduação em Ciência da Computação da Universidade Federal de São Carlos, assim como a JAMP já o faz há algum tempo.

14.3 Trabalhos Futuros

Entre as possibilidades de trabalhos futuros a este, encontram-se vários temas e propostas de pesquisas, tanto sobre a arquitetura OpenReality quanto sobre a JAMP. A seguir, são apresentadas essas possibilidades para cada uma das arquiteturas.

14.3.1 Arquitetura OpenReality

- Desenvolvimento de mecanismos de *heartbeat* para a manutenção dos dados do Diretório de Ambientes de Visualização Distribuída.

- Criação dos módulos de distribuição das informações de atualização de estado para operarem com Regeneração Freqüente de Estado, oferecendo uma maior flexibilidade para o desenvolvedor, no que se refere ao controle do grafo de cena. Assim, ambientes de visualização distribuída que não necessitam possuir uma consistência total dos dados podem ser desenvolvidos. Jogos 3D são o exemplo mais comum de utilização dessa técnica que pode ser encontrado.

- Adição de mecanismos para auxiliar na predição e na convergência dos dados e possibilitar a utilização das técnicas de *Dead-Reckoning* em aplicações OR. Assim, ao fazer uso da Regeneração Freqüente de Estado, o desenvolvedor pode ser auxiliado pelas técnicas de *Dead-Reckoning* para aumentar a sensação de realismo do ambiente.

- Especificação e implementação de um compilador que gera os objetos do grafo de cena distribuído a partir dos objetos do grafo de cena local. Com ele, todo e qualquer novo objeto do grafo de cena criado pelo próprio usuário pode ser utilizado também nos ambientes de visualização distribuída, sem que o desenvolvedor seja obrigado a implementar também a versão distribuída desse objeto.

- Estudos de modificações na utilização dos protocolos de transporte, buscando aumentar a escalabilidade, consistência e interatividade providas às aplicações. A utilização do envio de informações redundantes, que pode reduzir o tempo gasto com a retransmissão dos dados, e o agrupamento de múltiplas PDUs OR sendo transmitidas dentro do mesmo pacote de transporte, que pode reduzir o consumo da largura de banda, auxiliam nessa busca.

- Estudo de análise dos requisitos de desempenho da OR em relação aos serviços de comunicação e distribuição propostos, visando, além de realizar uma avaliação do real estado da arquitetura, prover informações utilizadas para medir o impacto das soluções encontradas para a proposta do item anterior.

14.3.2 Plataforma JAMP

- A integração do compilador JAMP2C em um ambiente de desenvolvimento já existente, através da confecção de módulos adicionais (*plug-ins*), ou mesmo a implementação de um novo ambiente de desenvolvimento próprio para a JAMP.

Com a confecção de *plug-ins*, conhecidas ferramentas como Rational Rose, ArgoUML, MVCase, entre outras, poderiam ser expandidas de forma a oferecer, a partir da modelagem em UML, a criação dos códigos dos objetos JAMP automaticamente. Uma ferramenta capaz de oferecer esse ambiente tornaria o processo de desenvolvimento de *software* sobre a Plataforma JAMP mais ágil e econômico.

- A especificação e o desenvolvimento de um sistema de arquivos distribuídos para a JAMP, aos moldes do JNDS, o *JAMP Networked File System* (JNFS). Faria uso do JNDS e ofereceria um sistema de arquivos próprio para a Plataforma JAMP, podendo ser disponibilizado na arquitetura como um simples objeto de serviço ou até como um *framework* de domínio. Esse sistema de arquivos possuiria as propriedades herdadas do JNDS de ser exportado e importado entre as aplicações, oferecendo, entre outras facilidades, um mecanismo de comunicação entre processos. Entre as aplicações que poderiam fazer uso deste sistema, pode-se citar os projetos que tratam de adaptação de conteúdo.

- A criação de um serviço de redundância de armazenamento utilizando o JNDS (ou mesmo o JNFS) que seja disponibilizado como serviço da JAMP. O serviço oferecido por esse sistema ofereceria algumas das funcionalidades que se encontram nos RAIDs, porém utilizando-se apenas *softwares*. Diversas hierarquias de armazenamento redundante podem ser projetadas fazendo uso da personalização das classes de armazenamento e *cache* oferecidas no *framework* JNDS. Entre as aplicações que poderiam usufruir desse sistema, estão os *sites* WWW e os sistemas de difusão de mídias, como os servidores de vídeo sob demanda e outros.

- O desenvolvimento de um serviço de distribuição de conteúdo baseado no balanceamento da carga dos servidores, que utilizaria o serviço de redundância anterior, oferecendo facilidades para a estruturação de *sites* e servidores de mídias que suportem um volume muito grande de usuários.

- Estudo analítico dos requisitos de desempenho do JNDS em relação aos serviços de comunicação e distribuição propostos, realizando ensaios sobre as mais diversas formas de estruturação hierárquica de diretórios, buscando avaliar qual a significância da perda de desempenho do sistema por utilizar o modelo RMI para a comunicação.

15. Referências Bibliográficas

[1] BAPTISTA, B. A. D. et al. **Openreality: An Architecture for Distributed 3D Visualization**. ACIS International Conference on Computer Science, Software Engineering, Information Technology, e-Business and Applications (CSITeA-02), Foz do Iguaçu, PR, Brasil, jun. 2002.

[2] SINGHAL, S. K.; ZYDA, M. **A Brief History of Shared, Networked Virtual Environments**. Curso 34 do ACM SIGGRAPH 99, Los Angeles, USA, ago. 1999.

[3] SEMENTILLE, A. C. **A utilização da arquitetura CORBA na construção de Ambientes Virtuais Distribuídos**. Teste de Doutorado, Instituto de Física de São Carlos. set. 1999.

[4] WOO, M et al. **OpenGL programming guide: the official guide to learn OpenGL, version 1.2,** 3rd ed, Addison-Wesley, Massachussetts, USA, nov. 1999.

[5] BARGEN, B.; DONELLY, T.P. **Inside DirectX**, Microsoft Programming Series Microsoft Press, abr. 1998.

[6] SCHROEDER, W. J.; MARTIN, K. M.; LORENSEN, W. E. **The Visualization Toolkit,** 2nd ed, Prentice-Hall, New Jersey, jan. 1998.

[7] SINGHAL, S.; ZYDA, M.. **Networked Virtual Environments: design and implementation.** Addison-Wesley; ACM Press, Massachussetts, USA; New York, USA, set. 1999.

[8] CRUZ-NEIRA, C.; SANDIN, D. J.; DEFANTI, T. A. **Surround-screen projection-based virtual reality: The design and implementations for the CAVE.** In: KAJIYA, J. T. (Ed.). Computer Graphics (SIGGRAPH'93 Proceedings), ACM Press, New York, USA, v.27, p.135-142, ago. 1993.

[9] PARENT, R. **Computer Animation : Algorithms and Techniques**. 1st ed., Morgan Kaufmann, California, USA, set. 2001.

[10] STRAUSS, P. S. **IRIS Inventor, A 3D graphics toolkit**. In: PAEPCKE, A.(Ed.) Proceedings of the 8th Annual Conference on Object-Oriented Programming Systems, Languages and Applications, ACM Press, District of Columbia, USA, p.192-200, 1993.

[11] GROUP, O. I. A. **Open Inventor C++ Reference Manual: The Official Reference Document for Open Systems**. Addison-Wesley, Massachussetts, USA, jul. 1994.

[12] BROLL, W.; KOOP, T. **VRML: today and tomorrow.** Computer and Graphics, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, v.20, n.3, p.427-443, mai-jun. 1996.

[13] ISO/IEC. **Information Technology – Computer graphics and image processing - Extensible 3D (X3D) – Part 1: Architecture and base components**. ISO/IEC FCD 19775-1, 1st ed., stage 40.99, dez. 2003.

[14] MACEDONIA, M. R.; ZYDA, M. J. **A taxonomy for networked virtual environments**. IEEE MultiMedia, v.4, n.1, p 48-56, jan-mar 1997.

[15] POPE, A. **The SIMNET Network and Protocols**. BBN Report 7102, BBN Systems and Technologies Advanced Simulations Division, Massachussetts, USA, jul. 1989.

[16] MILLER, D.; THORPE, J. **SIMNET: The advent of simulator networking.** In Proceedings of IEEE, ago, v.83, n.8, p.1114-1123, ago 1995.

[17] IEEE COMPUTER SOCIETY. **IEEE 1278-1993, Standard for information technology – Protocols for distributed simulation applications: Entity information and interaction**. New York, USA. 1993.

[18] MACEDONIA, M. R. et al. **NPSNET: A network software architecture for large-scale virtual environment**. Presence MIT Press, v.3, n.4, p.265-287, 1994.

[19] FRCON, E.; STENIUS, M. **Dive: A scalable network architecture for distributed virtual environments**. Distributed Systems Engineering Journal (special issue on Distributed Virtual Environments), v.5, n.3, p.91-100, set 1998.

[20] FLOYD, S. et al. **A reliable multicast framework for light-weight sessions and application level framing**. IEEE/ACM Transactions on Networking, nov. 1996.

[21] SINGH, G. et al. **Bricknet: A software toolkit for network-based virtual worlds**. Presence MIT Press, v.3, n.1, p.19-34, 1994.

[22] JOHNSON, A. et al. **Cavern: the cave research network**. In: Proceedings of 1st International Symposium on Multimedia Virtual Laboratories. Tokyo, Japan, p.15-27, 1998.

[23] GREENHALGH, C.; BENFORD, S. **MASSIVE: a Distributed Virtual Reality System Incorporating Spatial Trading,** In: Proceedings of IEEE 15th International Conference on Distributed Computing Systems (DCS'95), IEEE Computer Society, Vancouver, Canada, jun. 1995.

[24] MACEDONIA, M. R. et al. **Exploring reality with multicast groups.** IEEE Computer Graphics and Applications, v.15, n.5, p.38-45, set. 1995.

[25] NAKAMURA, K. N.; SHINOHARA, K. **Distributed virtual reality system for cooperative work.** NEC Research and Development, v.35, n.4, p.403-409, out. 1994.

[26] PANTEL, L.; WOLF, L. C. **Network issues for Video and Games on impact of delay on real-time multiplayer games.** In: Proceedings of the 12th International Workshop on Network and Operating Systems support for Digital Audio and Video, Florida, USA, p. 23-21, set. 2002.

[27] COMER, D. E. **Internetworking with TCP/IP.** 3rd ed., Prentice-Hall, New Jersey, USA, mar. 1995.

[28] TANENBAUM, A.S. **Computer Networks.** 3rd ed., Prentice-Hall, New Jersey, USA, mar. 1996.

[29] MOGUL, JEFFREY. **Request for Comments 919 – Broadcasting Internet Datagrams.** Network Working Group, out 1984.

[30] OBJECT MANAGEMENT GROUP (OMG). **The Common Object Request Broker Architecture Specification.** Revisão 2.6.1, mai. 2002

[31] COULOURIS, G.F et al. **Distributed Systems: Concepts and Design.** 3rd edition, Addison-Wesley, Workingham, UK, ago. 2000.

[32] LEIGH, J. et al. **Adaptive networking for tele-immersion.** In: FRÖHLICH, B.; DEISINGER, J.; BULLINGER, H. J. (Ed.). Proceedings of the joint 5th Immersive Projection Technology and 7th Eurographics Virtual Environments Workshop (EGVE-01), Springer-Verlag, Wein, Australia, p.199-208, 2001.

[33] MENDONCA, S. P. et al. **Development of object oriented distributed systems (DOODS) using frameworks of the JAMP platform**. ICSE'99, Los Angeles, USA, mai. 1999.

[34] HADDAD, I. F.; PAQUIN, E. **MOSIX: a cluster load-balancing solution for Linux.** Linux Journal, v. 85, p. 120-122, mai 2001.

[35] BELLEZI, M.A. **Estudos de Mecanismos de Representação e Sintetização em Ambientes Virtuais Distribuídos: Projeto Inicial do Framework OpenReality**. Dissertação de Mestrado. Departamento de Computação. UFSCar, São Carlos, ago. 2003.

[36] DCOM. **Distributed Component Object Model Specification.** Microsoft press, mar 1998.

[37] RMI-IIOP. **Remote Method Invocation over Internet InterORB Protocol.** Object Management Group and Sun Microsystems, out. 1997. Disponível para consulta em http://java.sun.com/j2se/1.4/docs/guide/rmi-iiop/index.html. Consultado em 17/06/2002.

[38] MENDONÇA, S.P. **Implementação de Serviço Multicast na Plataforma JAMP para Suporte a Aplicações Colaborativas**. Dissertação de Mestrado. Departamento de Computação. UFSCar, São Carlos, ago. 2000.

[39] PRESSMAN, R.S. **Software Engineering: practitioner's approach**. 5th ed. McGraw-Hill Book, Massachussetts, USA, dez. 2001

[40] ARCHITECTURE PROJET MANAGEMENT LTD (APM). **ANSAware 4.0 Application Programmer's Manual**. Cambridge, UK, mar. 1992.

[41] ADVANCED NETWORK SYSTEMS ARCHITECTURE (ANSA). **Requirements Phase Manual.** Cambridge, UK, ago. 1985.

[42] SUN MICROSYSTEMS INC. **The Java Remote Method Specification**. ed. 1.1, California, USA, nov. 1996.

[43] OBJECT MANAGEMENT GROUP (OMG). **The OMG Hitchhiker's Guide – A Handbook for the OMG Technology Adoption Process**. ver. 6.5, 2003.

[44] OBJECT MANAGEMENT GROUP (OMG). **Object Management Architecture Guide**. OMGTC Document, rev. 2.0, New York, USA, nov. 1992.

[45] OBJECT MANAGEMENT GROUP (OMG). **Common Object Request Broker: architecture and specification**. rev. 2.0, Massachussetts, USA, jul. 1995.

[46] FERREIRA, M. M.; TREVELIN, L. C. **The Java Broker System: Concepts & Java Programming Guide**. Relatório Técnico. Departamento de Computação. UFSCar, São Carlos, mar. 1998.

[47] FERREIRA, M. M.; TREVELIN, L. C. **Especificação e Testes de Uma Interface de Alto Nível para Manipulação de Áudio no Servidor de Áudio Distribuído do SMmD**. Relatório Técnico. Departamento de Computação. UFSCar, São Carlos, 1997.

[48] VESSONI, M. D. **Design de um Ambiente de Escrita Colaborativa Baseado em uma Plataforma de Suporte para Distribuição**. Dissertação de Mestrado. Departamento de Computação. UFSCar, São Carlos, ago. 1999.

[49] GUIMARÃES, M. P. **Projeto e Implementação do Suporte para Trabalho Cooperativo na Plataforma JAMP**. Dissertação de Mestrado. Departamento de Computação. UFSCar, São Carlos, jan. 2000.

[50] SOUZA, L.F.H. **Estudos de Modelos de Serviços para middleware e proposta de extensões à Plataforma JAMP**. Dissertação de Mestrado. Departamento de Computação. UFSCar, São Carlos, fev. 2000.

[51] OBJECT MANAGEMENT GROUP (OMG). **The Common Object Request Broker: architecture and specification**. rev. 2.4, Massachussetts, USA, jul. 1996.

[52] MATTOS, E.C.T.de. **Estudo e Construção de um Ambiente de grade computacional *peer-to-peer*, com ênfase no balanceamento de carga**. Texto apresentado em exame de qualificação de mestrado. Departamento de Computação. UFSCar, São Carlos, dez. 2003.

[53] JARDIM, F.M. **Desenvolvimento de um framework preceptivo à mudança contextual de localização para implementação de serviços transmissores de vídeos voltados a dispositivos móveis.** Texto apresentado em Exame de Qualificação de Mestrado. Departamento de Computação. UFSCar, São Carlos, ago. 2003.

[54] OBJECT MANAGEMENT GROUP (OMG). **Common Object Request Broker: architecture and specification**. ver. 3.0, Massachussetts, USA, jun. 2002.

[55] SUN MICROSYSTEMS INC. **The Java Remote Method Specification**. ed. 1.4, California, USA, out. 2002.

[56] BOOCH, G.; RAMBAUGH, J. **UML, guia do usuário**. Tradução de **The Unified Modeling Language user guide.** Campus, Rio de Janeiro, 2000.

[57] TEIXEIRA, R.C.; DUARTE, O.C.M.B. **Comunicação Multidestinatárias em Ambientes Virtuais Distribuídos Escaláveis.** Grupo de Teleinformática e Automação, UFRJ, Rio de Janeiro, 1999.

[58] SOUZA, C. **Um Framework para Editores de Diagramas Cooperativos baseados em Anotações.** Dissertação de Mestrado. Universidade Estadual de Campinas, out. 1998.

[59] CCITT. **The Directory: Overview of Concepts, Models and Services.** CCITT Recommendation X.500, 1988.

[60] ISO/IEC. **Information Technology – Open Systems Interconnection – The Directory: Overview of Concepts, Models and Services**. ISO/IEC 9794-1, 2001.

[61] DAY, J.D.; ZIMMERMANN, H. **The OSI Reference Model.** Proceedings of the IEEE, v.71, p.1334-1340, dez 1983.

[62] ALBITZ, P.; LIU, C. **DNS and BIND**. 4th ed., O'Reilly, abr. 2001.

[63] TALIGENT INC. **Building object-oriented frameworks.** A Taligent White Paper, nov. 1997.

[64] TANENBAUM, A.S., **Structured Computer Organization.** 4ª ed., Prentice-Hall, New Jersey, USA, out. 1998.

[65] BORLAND SOFTWARE CORPORATION. **Visibroker for Java Developer's Guide – Entertprise Server 6.** Borland Software Corporation, California, USA, 2003.

[66] CHAN, P. **The Java™ Developers Almanac 1.4, Volume 1: Examples and Quick Reference**. 4th edition, Addison-Wesley, San Francisco, mar. 2002.

[67] FLANAGAN, D.; FARLEY, J.; CRAWFORD, W. **Java Enterprise in a Nutshell – A desktop quick reference**. O'Reilly, set. 1999.

[68] AHO, A.V.; SETHI, R.; ULLMAN, J.D. **Compilers: Principles, Techniques, and Tools**. Addison Wesley, Massachussetts, USA, jan. 1985.

[69] APPEL, A.W. **Modern Compiler Implementation in Java.** Edição revisada e estendida de **Modern Compiler Implementation in Java: Basic Techiniques**. Cambridge University Press, Cambridge, UK, 1999.

[70] GOSLING, J.; JOY, B.; STEELE, G. **The Java Language Specification**. 1st ed., Addison-Wesley, Massachussetts, USA, jun. 2000.

[71] VINCE, J. **3-D Computer Animation**. 2nd ed., Addison-Wesley, Massachussetts, fev. 1994.

[72] BELLEZI, M.A.; BAPTISTA, B. A. D.; TREVELIN, L.C. **Representação e Distribuição no Framework OpenReality**. Proceedings of SVR 2003 – VI Synposium on Virtual Reality, Ribeirao Preto, SP, Brasil, 15-18 out. 2003.

16. Glossário de Termos

Application Programming Interface – É um conjunto de rotinas, protocolos e ferramentas para a construção e o desenvolvimento de aplicações. Uma API torna o desenvolvimento de programas mais fácil, pois fornece diversos blocos de código prontos.

Broadcast – Forma de transmissão de dados que se propaga em todas as direções e pode ser recebida por todos os receptores. Nas redes de computadores, é o mecanismo utilizado em redes locais para a transmissão de dados de um emissor para todos os receptores, localizados em um mesmo segmento.

Broker – Dentro do contexto de sistemas distribuídos, é o *software* que faz o papel de um mediador para os objetos, sendo procurado pelos servidores, quando estes tornam-se disponíveis, e pelos clientes, sempre que necessitam de um determinado serviço.

Cache – Técnica de armazenamento de dados que torna o acesso aos dados presentes em dispositivos de acesso randômico à memória mais rápido e eficiente. Geralmente operam em um espaço limitado de armazenamento e utilizam políticas de manutenção dos dados.

Cluster – Aglomerado de dispositivos agrupados e concentrados na resolução de um ou mais problemas.

Cockpit – Substantivo da língua inglesa que denota cabine de pilotagem. É o ambiente que os pilotos têm para pilotar seus equipamentos. Geralmente é constituído por um acento, painéis de navegação e controles.

Dead-Reckoning – Técnica utilizada para reduzir a utilização da largura de banda da rede de comunicaçõa, contornar os problemas relativos aos seus atrasos e aumentar a sensação de realismo oferecida aos participantes. Opera juntamente com os mecanismos de

regeneração freqüente de estado e utiliza técnicas de previsão e de aproximação do comportamento de um objeto do ambiente.

Debug – Verbo da língua inglesa que significa remoção de erros.

DirectX – Grupo de técnicas desenvolvidas para fazer com que computadores desfrutem ao máximo de seus recursos multimídia, constituindo um *framework* proprietário da empresa Microsoft.

Framework – Conjunto de classes que oferecem uma solução genérica para um determinado domínio de aplicação. Esse conjunto de classes define um projeto abstrato para soluções e problemas de uma família de aplicações dentro de um domínio. Suas classes implementam o comportamento genérico do domínio da aplicação, deixando os aspectos específicos de cada aplicação serem completados por subclasses. Funciona como um molde para a construção de aplicações ou subsistemas pertencentes a um domínio de aplicação.

GNU – Este é um acrônimo para "*GNU's Not Unix*", um grupo que em 1984 iniciou o desenvolvimento de sistemas baseados em Unix. Hoje possui como seu maior representante o grupo *Free Software Foundation*.

Graphic Processing Unit – É uma unidade de processamento gráfico. Hoje pode ser encontrada como uma ou mais pastilhas de circuito eletrônico, especializadas em cálculos vetoriais junto às placas gráficas. Podem, inclusive, possuir otimizações para aplicação de texturas, rotações, translações e muitas outras transformações gráficas.

Hardware – É a parte física dos computadores, telecomunicações e outros dispositivos de Tecnologia da Informação.

Hash – Função matemática injetora utilizada nos mecanismos de armazenamento e recuperação de informação, visando reduzir a complexidade algorítmica da busca e localização dos elementos. Geralmente é utilizada para determinar a posição em que um dado elemento deve ser encontrado em uma tabela.

Heartbeat – Substantivo da língua inglesa que denota pulsação cardíaca. Usado para referenciar o mecanismo computacional de verificação freqüente da situação de determinado *software* ou *hardware*. Costuma ser empregado em situações que falhas podem comprometer todo o sistema, buscando identifica-las o mais breve possível.

Information Request Broker – Em uma arquitetura distribuída, é o *software* que atua como um *broker* entre as requisições dos clientes e as informações que os objetos servidores possuem.

Marshalling – É o processo de agrupar dados e transforma-los em um formato padrão antes de envia-los pela rede. *Marshalling* é utilizado na passagem de parâmetros de um programa escrito em uma linguagem para outro escrito em outra linguagem.

Massive Multi-Player Games – Jogos *on-line* que fazem uso de gráficos tridimensionais e que suportam a presença de um grande número de jogadores.

Middleware – Programa que conecta duas aplicações separadas. O termo *middleware* é utilizado para descrever produtos que agem como uma ligação entre duas aplicações. Algumas categorias de *middleware* incluem: ODBC (*Open DataBase Connectivity*) e JDBC (*Java DataBase Connectivity*).

Multicast – Forma de transmissão utilizada par difundir mensagens para um grupo selecionado de receptores. Requerem protocolos e redes mais robustos.

MultiEng – Plataforma distribuída que fornece diversos *frameworks* para o desenvolvimento de aplicações distribuídas multimídias e cooperativas em ambiente multiplataforma.

Object Request Broker – É um componente de *software* que atua como um *middleware* entre clientes e servidores. Os ORBs recebem requisições, repassam para os servidores apropriados e retornam os resultados para os clientes.

Peer-to-peer – Técnica de comunicação ponto a ponto, onde cada um desses pontos pode exercer tanto o papel de cliente como o de servidor.

Role Playing Games – Estilo de jogo que envolve vários participantes em uma aventura simulada, onde cada um possui uma personalidade pré-definida. As aventuras são coordenadas por um mestre que não participa do jogo, apenas cria as situações que os demais devem enfrentar.

Software – É um termo genérico para designar programas que operam computadores e os dispositivos a ele relacionados.

Stream – Seqüência de bytes que representam alguma informação quando reorganizados em sua ordem original. Por exemplo, uma *stream* de vídeo quando é decodificada, pode reproduzir o vídeo original. *Stream* de áudio pode ser nossa voz trafegando em uma linha telefônica.

Stub – Termo utilizado ao trecho de código gerado automaticamente por algum tipo de ferramenta de auxílio ao desenvolvimento. Geralmente, designa o suporte à comunicação de dados de aplicações que seguem o modelo cliente-servidor, gerado automaticamente.

Timestamp – É um selo com a estampa do tempo exato em que determinado evento foi registrado por um sistema. Esse conceito é muito utilizado em sistemas distribuídos para que os relógios possam ser sincronizados e a ordem de ocorrência dos eventos determinada.

Toolkit – Termo derivado da expressão *tool kit* da língua inglesa, que denota ferramental; conjunto ou coleção de ferramentas para um determinado fim.

Unicast – Forma de transmissão de dados dada entre um emissor e um único receptor.

Unmarshalling – É o processo de receber dados pela rede em um determinado padrão e transformá-los em valores e tipos válidos para a aplicação. Desfaz o processo de *marshalling* para que as aplicações possam realizar a comunicação.

Verbose – Adjetivo da língua inglesa que acentua a capacidade de verbalização de algo ou alguém; aquilo ou aquele que verbaliza de forma acentuada, passando mais informações do que o necessário.