UNIVERSIDADE DE SÃO PAULO
INSTITUTO DE FÍSICA DE SÃO CARLOS
DEPARTAMENTO DE FÍSICA E INFORMÁTICA

**A UTILIZAÇÃO DA ARQUITETURA CORBA NA CONSTRUÇÃO DE
AMBIENTES VIRTUAIS DISTRIBUÍDOS**

ANTONIO CARLOS SEMENTILLE

Tese apresentada ao Instituto de Física de São Carlos, Universidade de São Paulo, para obtenção do título de Doutor em Ciências "Física Aplicada –
Opção: Física Computacional"

04

Orientador: Prof. Dr. Claudio Kirner

São Carlos – São Paulo
1999

IFSC-USP SERVIÇO DE BIBLIOTECA
INFORMAÇÃO

Sementille, Antonio Carlos

A Utilização da Arquitetura CORBA na Construção de Ambientes Virtuais Distribuídos/Antonio Carlos Sementille.--São Carlos, 1999.
186p.

Tese (Doutorado)--Instituto de Física de São Carlos, 1999.

Orientador: Prof. Dr. Cláudio Kimer

1. Sistemas Distribuídos. 2. Realidade Virtual.

I. Título.

IFSC**UNIVERSIDADE
DE SÃO PAULO****Instituto de Física de São Carlos**

Av. Dr. Carlos Botelho, 1465
CEP 13560-250 - São Carlos - SP
Brasil

Fone (016) 273-9333

Fax (016) 272-2218

**MEMBROS DA COMISSÃO JULGADORA DA TESE DE DOUTORADO DE ANTONIO
CARLOS SEMENTILLI APRESENTADA AO INSTITUTO DE FÍSICA DE SÃO
CARLOS, DA UNIVERSIDADE DE SÃO PAULO, EM 17 DE SETEMBRO DE 1999.**

COMISSÃO JULGADORA:

Prof. Dr. Claudio Kirner/UFSCar

Prof. Dr. Gonzalo Travieso/IFSC-USP

Prof. Dr. Marcelo Knorich Zuffo/EP-USP

Prof. Dr. Sergio Takeo Kofuji/EP-USP

Prof. Dr. Luis Carlos Trevelin/UFSCar

Esta tese é dedicada:

- ❖ À minha esposa Luciane;
- ❖ Aos meus pais: Adelina e Antonio;
- ❖ Aos meus irmãos: Adelaine e Adilson;
- ❖ À memória de meus avós: Angelina, Gentil e Antonio.

AGRADECIMENTOS

Ao Prof. Dr. Cláudio Kirner, pela sua orientação, incentivo, apoio e paciência para comigo, durante todo o desenvolvimento deste trabalho.

Ao amigo Prof. Dr. José Remo Ferreira Brega, pelo contínuo incentivo e ajuda durante todos esses anos.

Ao Prof. Dr. Durval Makoto Akamatu, pela amizade e confiança depositada em mim, desde o início de meus estudos ao nível de pós-graduação.

Ao Felício Vanderlei Deriggi Júnior e Mario Massakuni Kubo, pelo trabalho em conjunto.

À Wladerez Aparecida G. Caiado, pelo apoio fornecido através da secretaria do Programa de Pós-Graduação do Instituto de Física da USP de São Carlos.

À Leonilda Lopes, pela eficiência no tratamento dos detalhes burocráticos, na secretaria do Departamento de Computação – UNESP – Bauru.

À Coordenadoria de Aperfeiçoamento de Pessoal de Nível Superior - CAPES, pela bolsa deslocamento concedida.

Ao CNPq, pelo auxílio oferecido através do PROTEM – CC - Fase III.

Aos meus familiares, em especial à minha esposa Luciane, pela compreensão, paciência e incentivo incondicionais.

Sumário

Lista de Figuras	vi
Lista de Tabelas	viii
Lista de Abreviaturas	ix
RESUMO	xii
ABSTRACT	xiii
Capítulo 1 – Introdução	1
1.1 Objetivos	2
1.2 Organização do Trabalho	3
Capítulo 2 - Ambiente Virtual Distribuído	5
2.1 Ambientes Virtuais	5
2.1.1 Caracterização	5
2.1.2 Ambientes Virtuais: Uma Visão Geral de suas Partes e Necessidades Atuais	8
2.2 Ambiente Virtual Distribuído	17
2.2.1 Conceituação e Caracterização	17
2.2.2 Os Componentes de um AVD	18
2.2.3 Taxonomia para Ambientes Virtuais Distribuídos	22
Capítulo 3 – Visão Geral dos Principais Ambientes Virtuais Distribuídos	34
3.1 Metodologia Utilizada	34
3.2 Ambientes Virtuais Distribuídos Levantados	35
3.2.1 Sistema dVS, versão 2.0 (<i>Division Ltd.</i>)	35
3.2.2 Sistema MR - <i>The Minimal Reality Toolkit (University of Alberta - Canada)</i>	37
3.2.3 Sistema VEOS - <i>The Virtual Environment Operating Shell</i>	38
3.2.4 Sistema DIVE - <i>Distributed Interactive Virtual Environment (Swedish Institute of Computer Science)</i>	40
3.2.5 Sistema BrickNet Toolkit	42
3.2.6 Sistema: EM - <i>Environment Manager</i>	44
3.2.7 Sistema: NPSNET (<i>Naval Postgraduate School - USA</i>)	46
3.2.8 Sistema AVIARY (<i>University of Nottingham - University of Manchester - UK</i>)	47
3.2.9 Sistema SIMNET - <i>Simulation Network (DARPA/BBN Laboratories)</i>	49
3.2.10 Sistema MASSIVE - <i>Model, Architecture, and System for Spatial Interaction in Virtual Environment. (University of Nottingham - UK, University Manchester - UK)</i>	50
3.2.11 Sistema RAVEN - <i>Remote Access Virtual Environment Network (NASA)</i>	51
3.2.12 Sistema PARADISE (<i>Distributed Systems Group at Stanford University</i>)	52

3.2.13 Sistema WAVES (<i>Waterloo Virtual Environment System</i>)	53
3.2.14 Sistema SPLINE (MERL – <i>Mitsubish Eletric Research Laboratory</i>)	54
3.2.15 Sistema VERN – <i>Virtual Environment RealTime Network (Institute for Simulation and Training – University of Central Florida)</i>	55
3.2.16 Sistema WorldToolkit (<i>Sense8 Corporation</i>)	57
3.3 Modelos e Características Principais	58
3.4 Uma Outra Abordagem de Classificação	58
Capítulo 4 - Computação Distribuída Orientada a Objeto	62
4.1 Objetos Distribuídos e Processamento Distribuído	62
4.2 Objetos Interoperáveis	64
4.3 Arquitetura de Gerenciamento de Objetos da OMG: CORBA	66
4.3.1 O Cliente ORB	68
4.3.2 O Servidor ORB	69
4.3.3 A Interoperabilidade da Arquitetura CORBA	70
4.3.4 Os Serviços de Objeto da Arquitetura CORBA	72
4.3.5 O Adaptador de Objetos da Arquitetura CORBA	74
4.4 Implementações CORBA Estudadas: ILU e ORBIX	76
4.4.1 O <i>Inter-Language Unification</i> (ILU)	76
4.4.2 A Arquitetura ORBIX	84
4.4.3 Uma Comparação entre a Especificação CORBA, ILU e ORBIX	96
Capítulo 5 – A Utilização da Arquitetura CORBA como Suporte para Aplicações RV Distribuídas	98
5.1 Considerações Iniciais	98
5.2. A Utilização da Arquitetura CORBA	100
5.2.1. A Motivação para o Uso da Arquitetura CORBA	100
5.3. O Modelo Proposto	100
5.4. Metodologia de Desenvolvimento Utilizada	102
5.4.1 Princípios Fundamentais de Desenvolvimento da Arquitetura CORBA	102
5.4.2. Descrição da Metodologia	102
Capítulo 6 - Implementações	115
6.1 Recursos Utilizados na Implementação dos Protótipos	116
6.1.1 O WorldToolkit	116
6.1.2 ORBIX e OrbixTalk	117
6.1.3 Linguagem de Programação	118
6.2 A Aplicação RV Distribuída: A Cidade Virtual	119
6.3 Protótipo 1: Usando Apenas o WorldToolkit e suas Facilidades de Comunicação	121
6.3.1 O <i>Loop</i> de Simulação	122
6.3.2 Os Recursos de Comunicação do WorldToolkit	123
6.3.3 A Estrutura da Aplicação do Protótipo 1	124
6.4 Protótipo 2: Usando o WorldToolkit e o ORBIX	134
6.4.1 Definição da <i>Interface</i> com o GFRV	134
6.4.2 Compilação e Implementação da <i>Interface</i> com o GFRV	139
6.4.3 Implementação da Aplicação Cliente	141
6.4.4 Implementação do Programa GFRV	144
6.5 Protótipo 3: Usando o WorldToolkit, o ORBIX e o OrbixTalk	144

6.5.1 Definição das <i>Interfaces</i> do Protótipo 3	146
6.5.2 Compilação e Implementação das <i>Interfaces</i> do Protótipo 3	148
6.5.3 A Aplicação Cliente do Protótipo 3	151
Capítulo 7 - Comparação dos Protótipos e Análise dos Resultados Obtidos	155
7.1 Ambiente Experimental	155
7.2 Latência da Comunicação	156
7.2.1 Análise dos Resultados Obtidos	157
7.3 Escalabilidade	159
7.4 Análise da Geração de Imagens	160
Capítulo 8 - Conclusões	166
8.1 Aspectos Gerais do Trabalho	166
8.2 Contribuições do Trabalho	168
8.3 Pesquisas Futuras	169
8.3.1 Aperfeiçoamentos ao Nível da Utilização da Arquitetura CORBA	169
8.3.2 Aperfeiçoamentos ao Nível da Aplicação de Realidade Virtual Distribuída	169
Referências Bibliográficas	171
Glossário	177

Lista de Figuras

Figura 1 - Estruturação de uma Aplicação RV Imersiva	7
Figura 2 - Estrutura de um HMD para Realidade Aumentada	8
Figura 3 - Os Seis Graus de Liberdade	11
Figura 4 - Comunicação <i>Unicast</i>	25
Figura 5 - Comunicação <i>Broadcast</i>	26
Figura 6 - Comunicação <i>Multicast</i>	27
Figura 7 - Um Ambiente com Visão Síncrona	28
Figura 8 - Um Ambiente com Visão Assíncrona	28
Figura 9 - Arquitetura CORBA	67
Figura 10 - Uma Ponte Genérica entre ORBs	69
Figura 11 - Estrutura de Interoperabilidade da Arquitetura CORBA	71
Figura 12 - Serviços de Objetos da Arquitetura CORBA	73
Figura 13 - A Estrutura de um Típico Adaptador de Objetos	75
Figura 14 - Arquitetura ORBIX	86
Figura 15 - A Arquitetura do Compilador IDL	95
Figura 16 - Modelo Proposto	101
Figura 17 - Um Canal de Eventos com Múltiplas Aplicações Produtoras e Consumidoras	106
Figura 18 - Visão Parcial do Arquivo Clientes.idl	109
Figura 19 - Visão Parcial do Arquivo Servidor.idl	109
Figura 20 - Estrutura de Compilação das <i>Interfaces</i> IDL	110
Figura 21 - Visão Parcial do Arquivo Clientes_i.h	111
Figura 22 - Visão Parcial do Arquivo Clientes_i.cpp	112
Figura 23 - Visão Parcial do Arquivo ClienteRV.cpp	114
Figura 24 - Grafo de Cena Simplificado	116
Figura 25 - Arquitetura OrbixTalk	118
Figura 26 (a) - Visão Geral de um Trecho da Cidade Virtual	119
Figura 26 (b) - Visão do Motorista, no Interior do Veículo (Visão da Esquerda)	119
Figura 27 - Visão da Planta da Cidade Virtual	121
Figura 28 - Um <i>Loop</i> de Simulação Padrão do WTK	122
Figura 29 - Grafo de Cena da Cidade Virtual	125
Figura 30 - Formato de Mensagem no Protótipo 1	126
Figura 31 - Algoritmo do Programa Principal	127
Figura 32 - Algoritmo da Função <i>setup_visao_dos_outros()</i>	128
Figura 33 - Algoritmo da Função <i>actions()</i>	129
Figura 34 - Algoritmo da Função <i>add_node()</i>	131
Figura 35 - Algoritmo da Função <i>delete_node()</i>	132
Figura 36 - Algoritmo da Função <i>manipula_tecla()</i>	133
Figura 37 - Visão Geral da Estrutura do Protótipo 2	135
Figura 38 - Definição da <i>Interface</i> IDL VirCity	136
Figura 39 - Comunicação entre um Cliente e o GFRV, no Protótipo 2	138
Figura 40 - O Arquivo <i>header</i> Cidade_i.h	139
Figura 41 - O Arquivo de Implementação Cidade_i.cpp	140
Figura 42 - Algoritmo do Programa Principal do Cliente	141

Figura 43 - Algoritmo da Função actions()	142
Figura 44 - Algoritmo da Função setup_net()	143
Figura 45 - O Programa Fonte GFRV.cpp	144
Figura 46 - Interação entre Clientes e Servidor (GFRV), no Protótipo 3	145
Figura 47 - A Definição da <i>Interface</i> InterfClient	147
Figura 48 - A Definição da <i>Interface</i> InterfServer	148
Figura 49 - Estrutura de Comunicação do Protótipo 3	149
Figura 50 - Arquivo <i>header</i> Clientes_i.h	150
Figura 51 - Arquivo <i>header</i> Servidor_i.h	151
Figura 52 - Algoritmo do Programa Principal do Cliente	152
Figura 53 - Algoritmo da Função actions()	153
Figura 54 - Algoritmo da Função setup_net()	154
Figura 55 - Latência para 10 Mensagens	157
Figura 56 - Latência para 100 Mensagens	158
Figura 57 - Latência para 1000 Mensagens	158
Figura 58 - Estimativa da Escalabilidade dos Protótipos	159
Figura 59 - Desempenho dos Protótipos com 2 ODU's Renderizados	162
Figura 60 - Desempenho dos Protótipos com 3 ODU's Renderizados	163
Figura 61 - Desempenho dos Protótipos: Renderização x <i>Wireframe</i>	164

Lista de Tabelas

Tabela I (a) - Principais Características dos AVDs Levantados	60
Tabela I (b) - Principais Características dos AVDs Levantados (continuação)	61
Tabela II - Alocação das Tarefas ao Cliente e Servidor	108
Tabela III - Polígonos Existentes na Cidade Virtual	162
Tabela IV - Classificação dos Protótipos Implementados	167

Lista de Abreviaturas

2D - Bidimensional
3D - Tridimensional
AIX - *Advanced Interactive eXecutive* (uma versão do UNIX produzido pela IBM)
AK - *Acknowledgement*
AOI - *Area of Interest*
API - *Application Programming Interface*
AV - *Ambiente Virtual*
AVD - *Ambiente Virtual Distribuído*
AVVIC - *Ambiente Virtual para Visualização Interativa*
BBN - *Bolt, Beranek and Newman, Incorporated*
BD - *Base de Dados*
BOA - *Basic Object Adapter*
CAVE - *Cave Automatic Virtual Environment*
COM - *Common Object Model*
CORBA - *Common Object Request Broker Architecture*
COSS - *Common Object Service Specifications*
DARPA - *Defense Advanced Projects Research Agency*
DCE - *Distributed Computing Environment*
DCE/CIOP - *DCE Common Inter-ORB Protocol*
DEC - *Digital Equipment Corporation*
DII - *Dinamic Invocation Interface*
DIS - *Distributed Interactive Simulation*
DIVE - *Distributed Interactive Virtual Environment*
DIX - *DEC/Intel/Xerox*
DLL - *Dynamic Link Libraries*
DSI - *Defense Simulation Internet*
DSI - *Dinamic Skeleton Interface*
DSOM - *Distributed System Object Model*
E/S - *Entrada/Saída*
EM - *Environment Manager*
ESIOP - *Environment-Specific Inter-ORB Protocol*
FDDI - *Fiber Distribution Data Interface*
GFRV - *Gerente de Facilidades de Realidade Virtual*
GIOP - *General Inter-ORB Protocol*
GUI - *Graphical User Interface*
HMD - *Head Mounted Display*
HP - *Hewlett-Packard*
HP/UX - *UniX da Hewlett-Packard*
HTTP - *Hyper Text Transfer Protocol*
Hz - *Hertz*
IBM - *International Business Machines*
ID - *Identification*
IDL - *Interface Definition Language*
IOP - *Internet Inter-ORB Protocol*
ILU - *Inter-Language Unification*

IP – *Internet Protocol*
ISL – *Interface Specification Language*
MASSIVE – *Model, Architecture, and System for Spatial Interaction in Virtual Environment*
MERL – *Mitsubishi Electric Research Labs*
MIDI – *Musical Instrument Digital Interface*
MIP – *Million Instructions Per Second*
MMX – *MultiMedia eXtensions*
MR – *Minimal Reality Toolkit*
ms - milissegundo
MVS – *Memory Virtual System (IBM)*
NAK – *Negative Acknowledgement*
NASA – *National Aeronautics and Space Administration*
NFS – *National Science Foundation*
NIS – *Network Information Service*
NMS – *Name-and-Maintenance-Server*
NPSNET – *Naval Postgraduate School Network*
ODU – *Objeto Dirigido pelo Usuário*
OE – *Objeto Estático*
OLE – *Object Linking and Embedding*
OMA – *Object Management Architecture*
OMG – *Object Management Group*
OML – *Object Modeling Language*
ORB – *Object Request Broker*
OSF – *Open Software Foundation*
OSI – *Open System Interconnect*
OT – *Object Table*
OTRMP – *OrbixTalk Reliable Multicast*
PC – *Personal Computer*
RAVEN – *Remote Access Virtual Environment Network*
RI – *Repositório de Interfaces*
RPC – *Remote Procedure Call*
RMP – *Reliable Multicast Protocol*
RV – *Realidade Virtual*
SBH – *String Binding Handle*
SII – *Static Invocation Interface*
SIMNET – *Simulation Network*
SKS – *Surrogate Kernel Server*
SOM – *System Object Model*
SPLINE – *Scalable Platform for Large Interactive Networked Environments*
SQL - *Structured Query Language*
SSI – *Static Skeleton Interface*
TCP – *Transmission Control Protocol*
TKS – *True Kernel Server*
UDP – *User Datagram Protocol*
UUID – *Unique User IDentifier*
VEOS – *Virtual Environment Operating Shell*
VERN – *Virtual Environment RealTime Network*
VESIV – *Virtual Environment for Shared Interactive Visualization*

VMS – *Virtual Memory System* (DEC)

VR – *Virtual Reality*

VRML – *Virtual Reality Modeling Language*

WAVES – *Waterloo Virtual Environment System*

WTK – WorldToolkit, da Sense8

WWW – *World Wide Web*

RESUMO

As aplicações de Realidade Virtual imergem o usuário em um ambiente virtual simulado, denominado Ambiente Virtual. A simulação de Ambientes Virtuais é um processo intensivo, o qual pode ser drasticamente limitado se for restrito a apenas um computador. É possível, através da distribuição, aumentar-se o tamanho e a abrangência destes sistemas, permitindo que múltiplos usuários interajam entre si e com o ambiente. Estes ambientes virtuais, são conhecidos como Ambientes Virtuais Distribuídos. A construção de Ambientes Virtuais Distribuídos é uma tarefa complexa, principalmente quando são levados em consideração, aspectos como a estruturação da comunicação ao nível de processo, a escalabilidade, a interoperabilidade e o reuso de seus componentes. Tais aspectos são, também, enfatizados, através da Tecnologia de Objetos Distribuídos, cujo maior representante atual é a Arquitetura CORBA (*Common Object Request Broker Architecture*). Assim, neste contexto, este trabalho apresenta um estudo e uma metodologia para a construção de Ambientes Virtuais Distribuídos que utilizem a Arquitetura CORBA como infra-estrutura de alto nível para a comunicação e sincronização entre seus objetos. Buscou-se, também, apresentar os elementos teóricos e práticos desta abordagem, através da implementação de três protótipos, o quais formaram uma base comparativa para o estudo da viabilidade das idéias usadas.

ABSTRACT

Virtual Reality applications immerse the user in a simulated virtual environment entitled Virtual Environment that is an intensive process which can be drastically limited if restricted to a single computer. It is possible to enlarge the size and reach of these systems through distribution to allow multiple users to interact among themselves and with the environment. These virtual environments are known as Distributed Virtual Environments. Their construction is a complex task, mainly when aspects such as communication structuring at the process level, scalability, interoperability and reuse of their components are taken into consideration. Such aspects are also emphasized through Distributed Objects Technology, being CORBA (Common Object Request Broker Architecture) currently the biggest representative. Thus, in this context, this thesis presents a study and methodology for the construction of Distributed Virtual Environments which utilize CORBA architecture as a high-level structure for the communication and synchronization among their objects. Last but not least, the aim of this thesis was also to present theoretical and practical elements of this approach, by implementing three prototypes, which formed a comparative basis to study the viability of the ideas hereby used.

Capítulo 1 – Introdução

O tema principal desta tese enquadra-se em uma área comumente descrita como Realidade Virtual (RV). Esta área é uma das mais atuais, em termos de pesquisa em Computação, e como tal, não se encontra, até o momento, bem delineada. O próprio termo “Realidade Virtual” foi originalmente criado para descrever um sistema que utilizava tecnologia imersiva, tais como capacetes de visualização (*Head Mounted Display* - HMD) e *data glove*. Desde então, a percepção do que é RV tem mudado para acompanhar as combinações de novos dispositivos de entrada e saída.

O termo “Ambiente Virtual” (AV) é usado para descrever o ambiente com o qual se interage quando se utiliza um sistema RV. Mais recentemente, os AVs têm sido expandidos para permitir a participação de diversos usuários, conectados através de um sistema distribuído. À estes sistemas, denominados Ambientes Virtuais Distribuídos (AVDs), é que esta tese especificamente se refere.

A grande maioria dos atuais desenvolvedores e pesquisadores de AVDs preocupa-se mais com os níveis inferiores de comunicação (níveis de rede e de transporte), do que com os níveis mais elevados (por exemplo, o nível de processo/aplicação). Várias são as soluções atualmente em uso e em pesquisa, podendo-se citar, principalmente, o protocolo *Distributed Interactive Simulation* (DIS) (IEEE (1993)), o IP *multicast* (Comer (1988)) e o *Real-Time Transport Protocol* (RTP) (Schulzrinne; Casner (1993)). Esta preocupação é, em grande parte, justificada pelo enquadramento dos AVDs como sistemas distribuídos de tempo real

(Hawkes (1996)), e devido a isto, os esforços de pesquisa tem se concentrado, especialmente, na minimização da latência de comunicação.

Sem a intenção de desprezar estes aspectos, realmente muito importantes, o interesse desta tese está voltado aos níveis mais próximos da aplicação de realidade virtual. Nestes níveis, o desenvolvedor encontra-se mais preocupado com a estruturação da comunicação ao nível do processo, a escalabilidade, a portabilidade, a extensibilidade e o reuso em grande escala. Tais preocupações são, também, promessas de uma tecnologia de *software* emergente conhecida como Tecnologia dos Objetos Distribuídos, cujo maior representante atual é a arquitetura CORBA (*Common Object Request Broker Architecture*).

Neste contexto, a principal motivação para esta tese é o estudo e a elaboração de uma metodologia para a utilização da arquitetura CORBA como suporte de alto nível, para a comunicação e sincronização de AVDs.

Cabe ressaltar que a metodologia, e as implementações desenvolvidas, também são aplicadas na elaboração do Módulo de Suporte do Sistema Distribuído do Projeto AVVIC (Ambiente Virtual para Visualização Interativa) (Kirner et al. (1995 a)). O Projeto AVVIC é uma pesquisa que envolve a parceria direta de várias instituições: a UFSCar (que participa como instituição principal), a PUCRS, a EMBRAPA-CNPDIA, a UNESP – Bauru e a Fundação Eurípides Soares da Rocha, e é financiada pelo CNPq – PROTEM – CC - Fase III. O AVVIC visa a construção de um AVD capaz de suportar visualização e animação em espaços tridimensionais. Além disso, o projeto também está relacionado com o projeto “VESIV: *Virtual Environment for Shared Interactive Visualization*” (Kirner et al. (1995 b)), aceito no Programa de Cooperação Internacional Alemanha-Brasil, promovido pelo GMD e CNPq.

1.1 Objetivos

Portanto, de acordo com o exposto, são objetivos deste trabalho:

- (a) Estudar a utilização da plataforma CORBA como suporte de comunicação de alto nível, para Ambientes Virtuais Distribuídos.

- (b) Elaborar uma metodologia para desenvolvimento de aplicações de Realidade Virtual que utilizem a plataforma CORBA como suporte de comunicação.
- (c) Realizar a implementação de AVDs, segundo a metodologia elaborada no item anterior, visando a comprovação e validação da mesma.
- (d) Verificar a aplicabilidade e a abrangência dos AVDs desenvolvidos.

1.2 Organização do Trabalho

O trabalho está dividido em oito capítulos, descritos resumidamente a seguir.

O capítulo 2 apresenta a caracterização de Ambientes Virtuais e Ambientes Virtuais Distribuídos, seus tipos mais conhecidos, uma visão geral de suas partes e suas necessidades atuais.

No capítulo 3, é apresentada uma metodologia para classificação de Ambientes Virtuais Distribuídos. São levantados, também, os Ambientes Virtuais Distribuídos mais importantes atualmente e classificados segundo a metodologia proposta.

No capítulo 4, são abordados os aspectos importantes da computação distribuída orientada a objeto e, em especial, as características da arquitetura CORBA. Ao final do capítulo, discute-se profundamente duas das implementações CORBA estudadas, realizando-se uma comparação entre estas e a especificação CORBA original.

O capítulo 5 descreve e discute a metodologia proposta nesta tese para a utilização da arquitetura CORBA como suporte para as aplicações distribuídas. Primeiramente são levantados os benefícios esperados com a utilização da arquitetura CORBA e, a seguir, é mostrada em detalhes, a abordagem desenvolvida.

As implementações dos protótipos, criados para testar e validar a metodologia de integração proposta no capítulo anterior, são descritas no capítulo 6. Todos os protótipos referem-se à mesma aplicação de realidade virtual distribuída, permitindo a criação de uma base comparativa adequada, para o estudo da viabilidade das idéias usadas na metodologia proposta.

O capítulo 7 mostra uma análise dos protótipos desenvolvidos, comparando-se os aspectos representativos em seu desempenho.

A tese é concluída no capítulo 8, com o levantamento de aspectos gerais do trabalho e das vantagens da abordagem utilizada. Os protótipos implementados são classificados segundo a mesma taxonomia apresentada no capítulo 3, são enfatizadas as contribuições do trabalho e listadas as sugestões para pesquisas futuras.

Capítulo 2 - Ambiente Virtual Distribuído

Os termos “Ambiente Virtual” e “Ambiente Virtual Distribuído” são extremamente recentes e, como tal, permanecem sem uma definição que os caracterizem de forma única. Assim, almejando um melhor delineamento, apresenta-se, neste capítulo, uma visão geral dos dois tipos de ambientes, objetos de estudo deste trabalho.

Inicialmente são discutidos os sistemas de Realidade Virtual e os Ambientes Virtuais, apresentando-se os tipos mais conhecidos, suas partes integrantes e suas necessidades atuais.

Ao final do capítulo, são discutidos os Ambientes Virtuais Distribuídos através da apresentação de suas características principais, seus componentes e requisitos.

2.1 Ambientes Virtuais

2.1.1 Caracterização

O crescente aumento da velocidade de processamento dos computadores, de suas capacidades gráficas, das redes, e dos periféricos especializados têm permitido a criação de aplicações de Realidade Virtual (RV), as quais imergem o usuário em um ambiente tridimensional simulado, denominado Ambiente Virtual (AV). Ainda em

sua infância, muito tem que ser feito antes que o potencial da RV seja alcançado, mas esta área oferece uma grande promessa.

Devido à sua natureza inter-disciplinar, existem diversas definições de Realidade Virtual, porém, a maioria delas sugerem a combinação de três idéias: imersão, interação e envolvimento (Morie (1994) e Kirner (1998)).

A idéia de imersão representa a sensação de se estar dentro de um ambiente. A interação, por sua vez, reflete a capacidade de se modificar o ambiente, segundo a vontade do usuário. E finalmente, o envolvimento, representa o grau de engajamento do ser humano em determinada tarefa.

Em uma implementação RV imersiva, os usuários usam óculos especializados que apresentam pequenos *displays* de vídeo no lugar das lentes, um para cada olho. Os *displays* tornam-se “lentes” para o mundo simulado. O HMD contendo estes óculos detectam o movimento da cabeça e atualizam a visão do usuário simultaneamente; o usuário tem a impressão de que pode olhar em volta e mover-se dentro do ambiente simulado. A experiência sensorial pode ser, ainda, enriquecida com som ou uma luva (*data glove*) para simular o contato entre o usuário e os objetos simulados.

Em uma implementação RV não imersiva, o *display* de vídeo de um computador torna-se uma janela para o mundo simulado (Locke (1995)).

Um AV também pode ser entendido como um sistema de *software* que cria a ilusão de um mundo que não existe na realidade. Este tipo de aplicação requer, como mostra a Figura 1, uma combinação da interação do usuário (entrada), a simulação de processos no ambiente (computação), e estímulos multi-sensoriais (saída).

A principal vantagem da RV sobre os convencionais gráficos gerados por computador é o sentimento de imersão criado pelos convincentes estímulos em tempo real apresentados ao usuário. A imersão, porém, é também muito difícil de se obter e sustentar: é uma tarefa muito custosa configurar um sistema para as diversas necessidades da aplicação (gerenciamento de entrada, simulação e exibição) de maneira a suprir o adequado desempenho exigido pela manutenção da ilusão.

Figura 1 – Estruturação de uma Aplicação RV Imersiva

2.1.1.1 Tipos de Sistemas de Realidade Virtual

A maior distinção entre os sistemas de RV pode ser considerada a forma como implementam a interface entre o usuário e o mundo virtual. Segundo este aspecto, as formas mais comuns são:

- **Sistemas “Window on World”:** estes sistemas, nitidamente não imersivos, utilizam um monitor de computador convencional para exibir o mundo virtual.
- **Sistemas de Mapeamento de Vídeo:** representa uma variação da abordagem anterior, realizando uma fusão entre uma entrada de vídeo da silueta do usuário com um gráfico bidimensional, gerado por computador. O usuário observa um monitor que mostra a interação de seu corpo com o mundo virtual.
- **Sistemas Imersivos:** como já mencionado, representam a última palavra em sistemas RV, pois devem ser capazes de imergir completamente o usuário no mundo virtual. Normalmente estes sistemas são equipados com HMD e luvas especiais. Uma variação destes sistemas usa múltiplos *displays* para criar uma “caverna” ou sala, através da qual os observadores podem passar (Cruz-Neira et al. (1993)).
- **Telepresença:** é uma variação da visualização completa gerada por computador. Esta é uma tecnologia que liga sensores remotos no mundo real, com o usuário humano, por intermédio de um ambiente virtual. Pode ser implementada por

mecanismos de teleoperação, uma interface homem-máquina, um telerobô e um ambiente remoto (Boman (1995) e Durlach; Manor (1995)).

- **Realidade Aumentada:** este tipo de sistema é resultante da combinação da telepresença e dos sistemas de Realidade Virtual (Azuma (1993), Boman (1995) e Azuma (1997)). Aqui, as entradas geradas por computador são combinadas com as entradas de telepresença e/ou a visão do usuário do mundo real, através de um HMD especial (Figura 2). A finalidade deste tipo de tecnologia é usar os objetos virtuais para auxiliar o usuário na compreensão de seu ambiente.

Figura 2 – Estrutura de um HMD para Realidade Aumentada

2.1.2 Ambientes Virtuais: Uma Visão Geral de suas Partes e Necessidades Atuais

Neste item é apresentado um levantamento das características importantes das partes integrantes de um ambiente virtual, sob a ótica de seu estado e de suas necessidades atuais. Os pontos de vista considerados são: as ferramentas de *software*, o *hardware* e as aplicações de realidade virtual (NFS (1992)).

2.1.2.1 Ferramentas de *Software*

As ferramentas de *software* para AVs devem suportar dispositivos especiais de *hardware* (usados nos AVs e no desenvolvimento destes). As ferramentas desenvolvidas atualmente são produtos comerciais ou resultado de projetos de

pesquisa. Com relação às ferramentas disponíveis no mercado, na sua maioria, suportam apenas uma configuração de *hardware* particular, vendida pelo produtor do *software*. Fornecem apenas um nível básico de suporte de *software* para o *hardware* do vendedor. Já as ferramentas de *software* de pesquisa, tendem a ser mais gerais, uma vez que não foram feitas para suportar uma plataforma de *hardware* particular. Suportam várias configurações de *hardware*, o que facilita o compartilhamento dos resultados da pesquisa e o código resultante.

Muitas destas ferramentas endereçam somente questões de suporte de dispositivo, mas de uma maneira independente do *hardware*. Algumas ferramentas estão sendo desenvolvidas para suportar a produção de ambientes virtuais, em adição ao suporte de *hardware* básico.

Em termos de necessidades, uma questão chave é a modelagem. Em muitas aplicações de realidade virtual, existe uma forte conexão entre o usuário e o modelo subjacente. O usuário deve ser capaz de se mover através do modelo, interagir com os objetos no modelo, e mudá-lo interativamente. O modelo se refere, geralmente, às estruturas de dados que são usadas para guardar a **informação geométrica** para o ambiente. Esta informação inclui: a forma dos objetos, suas partes móveis e propriedades físicas, e os comportamentos que podem executar (como eles interagem com outros objetos no ambiente e com o usuário). Esta informação não é usada apenas para produzir a apresentação visual do ambiente, mas também é usada para produção de som e realimentação tátil.

Em uma aplicação de realidade virtual podem existir múltiplos usuários compartilhando o mesmo ambiente, são os já mencionados Ambientes Virtuais Distribuídos, e serão discutidos em maiores detalhes no item 2.2. Eles podem ter diferentes necessidades ou executar tarefas diferentes, as quais podem resultar em uma diferente visão do modelo para cada usuário. Em adição, cada usuário pode ter preferências pessoais que ditam uma diferente visão do modelo. O *software* de modelagem deve ser capaz de suportar as diferentes visões do usuário.

As aplicações onde existam múltiplos usuários no mesmo ambiente, ao mesmo tempo, implicam que o modelo deve ser compartilhado entre os usuários. Cada usuário deve ter uma visão consistente da aplicação. Devem existir métodos para manter estes modelos atualizados e, ao mesmo tempo, permitir aos usuários modificar as cópias individuais do modelo.

O modelo poderá ser usado para incorporar diversas mídias, tais como gráficos e som. Neste caso, deve conter a informação requerida para todas estas mídias, mais a informação necessária para sincronizar esta apresentação (o mesmo problema existe nas aplicações multimídia).

Os objetos no AV podem corresponder a objetos que existem no mundo real. Quando isto acontece, esperamos que estes objetos reajam como os objetos reais (devem seguir as tradicionais leis da Física). Portanto, há a necessidade de se modelar, no mínimo, uma parte da Física (movimento e detecção de colisão entre objetos).

Alguns objetos no AV devem ser capazes de responder as ações do usuário e as ações de outros objetos: devem existir meios de especificar estes comportamentos e associá-los aos objetos no modelo. Alguns objetos podem ter comportamentos autônomos, como andar, por exemplo. Isto envolverá interações entre os objetos e o tempo simulado dentro do ambiente. Existirá alguma sobreposição entre as técnicas requeridas aqui e as técnicas que estão sendo desenvolvidas em animação por computador e simulação.

Uma das questões na modelagem para AV é se esta deve ocorrer dentro ou fora do ambiente. Existem vantagens para as duas abordagens: modelar dentro do ambiente fornece ao projetista uma boa visão do que o usuário verá quando estiver no ambiente; modelar fora, por outro lado, permite ao projetista usar múltiplas visões do ambiente e utilizar técnicas mais tradicionais. Atualmente, os sistemas de modelagem de dentro do ambiente não são tão bons quanto os de modelagem fora do ambiente. Isto se deve ao fato dos primeiros utilizarem dispositivos de E/S que ainda estão em aperfeiçoamento.

Um dos problemas principais com os grandes modelos está na “navegação” através deles. Em ambientes 2D, mapas gerais do modelo podem ser apresentados em telas separadas, mas técnicas similares não têm sido desenvolvidas para ambientes 3D.

Em um AV o usuário deve poder “navegar” em três dimensões, ou seja, com “seis graus de liberdade”. Cada grau se aplica a uma direção do movimento. Na verdade, conferem ao usuário a possibilidade de se movimentar em seis direções simultâneas: uma combinação de posição espacial (eixos x , y e z) e orientação espacial (*pitch*, *roll* e *yaw*), como mostra a Figura 3.

Figura 3 - Os Seis Graus de Liberdade

A modelagem é um processo consumidor de tempo, exigindo, portanto, de melhores ferramentas para suportá-la.

Uma vez que a produção de bons modelos exige considerável esforço, deve existir algum mecanismo para o compartilhamento dos modelos existentes. Existem dois aspectos para este compartilhamento: o primeiro é a determinação de um formato padrão para a codificação e a transmissão dos modelos (o formato deve ser capaz de codificar toda a informação de modelagem, incluindo estrutura hierárquica, propriedades físicas dos objetos, som e comportamento); o segundo aspecto, deve existir um ou mais lugares que armazenem, mantenham e distribuam estes modelos.

Existe uma necessidade para uma certa funcionalidade de tempo real em sistemas AVs. Em particular, existe uma necessidade de uma baixa latência no processamento de pedidos, a habilidade de associar prioridades aos processos, e o gerenciamento do tempo. Existem sistemas operacionais de tempo real que atingem estes objetivos, porém seria interessante que estas facilidades de tempo real fossem incorporadas a um maior número de sistemas operacionais.

2.1.2.2 O Hardware para AVs

O *hardware* para ambientes virtuais é uma área de rápido progresso porque, muito desta tecnologia tem aplicações fora do AV, como por exemplo em áreas como a robótica, entretenimento e instrumentação.

O *hardware* pode ser dividido em diversas categorias, sendo que as principais são: os sistemas de rastreamento, sistemas *haptic* (realimentação de força e toque), geradores de imagens, sistemas de áudio e sistemas de entrada por voz (NFS(1992)).

Sistemas de Rastreamento

São responsáveis pelas técnicas de localização e orientação da cabeça e mãos do usuário, bem como de qualquer objeto do mundo real de interesse do usuário. Existem 5 categorias:

- Sistemas magnéticos (exemplificados pelos rastreadores Polhemus e Ascension), utilizam campos magnéticos emitidos por pequenas antenas e detectadas por uma ou mais antenas receptoras. Não são confiáveis na presença de objetos de metal. Os sistemas atuais têm um alcance de menos de 1m do emissor. A latência nestes sistemas é da ordem de 50 ms. A resolução atualmente é de 1mm e 0,03 graus e a precisão é de aproximadamente 3mm e 0,1 grau.
- Sistemas acústicos (exemplificado pelo *Mouse* Logitech), usam pulsos ultrasônicos. O alcance e a latência são aproximadamente as mesmas dos sistemas magnéticos, sem a susceptibilidade à interferência magnética.

- Sistemas inerciais antigamente eram pesados, caros e tendenciosos. Todavia, avanços tecnológicos prometem para breve a suficiente sensibilidade. Estes dispositivos não estão disponíveis comercialmente.
- Sistemas mecânicos, como o observador BOOM, usa uma rígida estrutura para suportar o dispositivo de observação e medição de posição. Sua latência é pequena (abaixo de 1 ms).
- Sistemas ópticos (como o Optotrack 3000) determinam a posição do alvo via técnicas de triangulação através de câmeras em locais conhecidos. Eles determinam a orientação observando múltiplos alvos. Uma medição mais precisa para a orientação é difícil, e fornecem um pequeno volume de trabalho (1 metro cúbico).

Necessidades

Para suportar muitos dos objetivos considerados, um sensor de posição deverá ter as seguintes características:

- operação sem fio;
- habilidade de rastrear múltiplos usuários no mesmo espaço, sem interferência;
- alcance acima de 10m x 10m x 5m, com referência a uma unidade base;
- nenhuma antena para longo alcance, de teto ou sensor de campo deve ser requerido;
- latência abaixo de 5ms;
- resolução de 1mm e 0,01 grau;
- precisão de 1mm e 0,01 grau (aplicações mais exigentes) e 1cm e 0,1 grau (aplicações normais);
- taxa de coleta ≥ 60 Hz; e
- sensoriamento direto do estado.

Sistemas *Haptic*

São responsáveis por fornecer a ilusão da exploração manual ou manipulação de objetos nos mundos virtuais. Estes dispositivos empregam

o tato humano, cinemática e sistemas de motores para a interação com o ambiente virtual. Eles executam duas funções básicas: a medição das posições e forças da mão do usuário ou outras partes do corpo para controlar as entradas para o AV; e manifestação das forças e posições e/ou suas distribuições temporais para o usuário.

Existem 4 áreas de interesse:

- Posição da mão/medição do ângulo de junção: bastões instrumentados, *joystics*, luvas e exoesqueletos;
- Aplicação de forças e torques: exoesqueletos tentam tanto medir os ângulos de junção da mão/braço e carregar estas junções com os torques necessários;
- *Displays* táteis: aplicação de força sobre a pele para simular o toque;
- Distribuição de outros estímulos: incluem diversos tipos de dores, temperatura, e possíveis estímulos elétricos.

Necessidades

- Estabilidade mecânica;
- Controle de vibração para os sistemas de realimentação de força.

Geradores de Imagens

Uma das tarefas que mais consomem tempo em um AV é a geração de imagens. A indústria de computadores gráficos vem melhorando rapidamente sua capacidade de renderização de texturas e modelos de iluminação, e o custo dos sistemas visuais continuam a cair, a uma taxa aproximada de 50% ao ano.

Necessidades

Muitos usuários gráficos não precisam de baixa latência (nos níveis requeridos pelos AVs), e então, muitas arquiteturas gráficas são estruturadas em *pipeline*. Pesquisas sobre estas arquiteturas necessitam explorar modos de reduzir a latência.

Sistemas de Áudio

Os interesses atuais dividem-se, principalmente, em duas categorias: o som espacial e o áudio sintetizado. Resumidamente, a abordagem usada para o tratamento do som espacial é o tratamento do som tridimensional utilizando-se dispositivos de processamento de sinais de alta velocidade, geralmente direcionado a *headphones*. Para o áudio sintetizado, são utilizados atualmente dispositivos geradores de sons, quase que exclusivamente com tecnologia MIDI (*Musical Instrument Digital Interface*).

Necessidades

A maioria dos sistemas de áudio geralmente realiza amostragens (*sampling*) de suas entradas com, no mínimo, 16 bits de resolução a uma taxa de 40 a 50 KHz. Para executar as funções de síntese espacial, é requerido um poder computacional de pelo menos 300 MIPs (NFS(1992)).

Sistemas de Entrada por Voz

Atualmente, encontram-se disponíveis, a preços acessíveis, dispositivos capazes de reconhecer palavras individuais provenientes de usuários específicos.

Necessidades

Muitas tarefas de seleção seriam melhor efetuadas pelo uso da voz, do que pelo uso dos mecanismos de entrada convencionais. Por isso, necessita-se um maior avanço na tecnologia de reconhecimento de voz.

Aplicações de Realidade Virtual

Para sistemas e tecnologias AV, com respeito a aplicações particulares, existem duas questões básicas:

1. Uma dada aplicação pode ser executada com a tecnologia AV? Muitas aplicações estão atualmente além do estado da arte.

2. Se uma aplicação puder ser implementada, ela se provará superior a outras tecnologias para a execução da mesma tarefa?

Atualmente existem diversas aplicações de qualidade que são regularmente usadas para resolver problemas do mundo real. Porém, muitas daquelas que são factíveis, precisam ainda ser implementadas.

As seguintes áreas de aplicação têm muito a ganhar com o desenvolvimento da tecnologia de AV:

- **Visualização de Modelo ou Dados:** as técnicas de visualização científicas são amplamente aceitas como um meio de extrair um entendimento de uma grande quantidade de dados.
- **Projeto e Planejamento:** qualquer método que ajude o projetista durante o ciclo de análise/refinamento melhorará a atividade inteira.
- **Educação e Treinamento:** modelos computacionais podem ser apresentados através de um ambiente virtual. Uma vez que o nível de fidelidade apropriado possa ser provido, o usuário pode interagir com o sistema virtual como se ele fosse real.
- **Teleoperações:** o AV pode ser aplicado em situações onde um ambiente é muito perigoso para um humano, mas um tele-sensor/operador pode entrar.
- **Testes Psicológicos:** os AVs podem produzir estímulos ambientais irrealizáveis, que são úteis para o teste de teorias de percepção e controle manual.
- **Aplicações Artísticas e de Entretenimento:** dado o montante de público e mídia interessado na tecnologia de AV, existe um potencial econômico para entretenimento com o AV.
- **Aplicações Artísticas:** como a tecnologia de AV está se tornando mais acessível, é razoável esperar que os artistas começarão a explorar as possibilidades únicas para a expressão que os ambientes virtuais oferecem.
- **Colaboração e Comunicação:** os ambientes virtuais acoplados com redes de alta velocidade e computação distribuídos podem fornecer um

espaço comum onde a comunicação, negociação e colaboração pode tomar lugar.

2.2 Ambiente Virtual Distribuído

2.2.1 Conceituação e Caracterização

Um Ambiente Virtual Distribuído, ou AVD, pode ser melhor definido como um sistema através do qual diversos usuários interagem entre si e em tempo real, sendo que estes podem estar situados em localidades diferentes. Tipicamente, cada usuário acessa seu próprio computador, usando-o para fornecer uma interface para o ambiente virtual. Estes ambientes geralmente oferecem aos usuários uma sensação de realismo através da incorporação de gráficos 3D e som estéreo, a fim de criar uma experiência imersiva (Singhal; Zyda (1998)).

Um AVD pode ser evidenciado por cinco características comuns:

- **Sensação de Compartilhamento de Espaço:** todos os participantes têm a ilusão de estarem localizados no mesmo lugar, tais como na mesma sala, prédio ou região. Este espaço compartilhado representa um local comum, no qual outras interações podem acontecer. Este local pode ser real ou fictício. O local compartilhado deve apresentar as mesmas características a todos os participantes.
- **Sensação de Presença:** quando entra em um local compartilhado, cada participante torna-se uma “pessoa virtual”, chamada de *avatar*, o qual inclui uma representação gráfica, um modelo de estrutura corporal (por exemplo, a presença de braços, cabeça, juntas, etc.), um modelo de movimento (por exemplo, o grau de movimentação das juntas, etc.), um modelo físico (por exemplo, peso, altura, etc.), e outras características. Uma vez dentro de um AVD, cada participante pode ver os outros avatares localizados no mesmo espaço. Similarmente, quando um usuário deixa o AVD, os outros participantes veem seu avatar partir. Nem todos os participantes precisam ser controlados por seres humanos. Alguns participantes podem ser entidades sintéticas controladas por modelos de simulação dirigidos por eventos.

- **Sensação de Tempo Compartilhado:** os participantes devem ser capazes de ver o comportamento uns dos outros em tempo real.
- **Comunicação entre os Participantes:** embora a visualização seja a base para um AVD efetivo, muitos AVDs também empenham-se em permitir que algum tipo de comunicação ocorra entre os participantes. Esta comunicação pode acontecer através de gestos, escrita ou voz. Esta comunicação adiciona uma sensação maior de realismo a qualquer ambiente simulado, e é fundamental para sistemas de engenharia e treinamento.
- **Forma de Compartilhamento:** os elementos mencionados anteriormente efetivamente oferecem um sistema de videoconferência de alta qualidade. O poder real de um AVD deriva da habilidade dos usuários em interagir uns com os outros e com o próprio ambiente virtual. Em uma simulação de combate ou jogo, por exemplo, os usuários precisam atirar ou colidir uns nos outros. Estas interações precisam ser modeladas realisticamente. Os usuários podem ser capazes de escolher, mover, e manipular objetos que existem no ambiente, e até, entregá-los a outros participantes.

A importância por trás dos AVDs reside na integração dos gráficos imersivos com a habilidade de possuir diversos usuários compartilhando informação e manipulando objetos no ambiente. A presença de diversos usuários independentes diferencia os AVDs dos sistemas de realidade virtual tradicionais. Os AVDs são mais apropriados para aplicações que demandam a criação de telepresença (item 2.1.1.1), ou seja, a ilusão que outros usuários estão visíveis, mesmo que situados fisicamente em lugares remotos. Nestas aplicações, os usuários necessitam uma sensação de realismo próxima da alcançada em um contato face a face.

2.2.2 Os Componentes de um AVD

Um sistema de AVD consiste de quatro componentes básicos: um *display* gráfico, dispositivos de comunicação e controle, um sistema de processamento, e uma rede de comunicação. Estes componentes trabalham juntos para fornecer a sensação de imersão em diferentes localidades.

2.2.2.1 *Displays* Gráficos

Os *displays* gráficos oferecem ao usuário a visão tridimensional do ambiente virtual. Tradicionalmente, estes *displays* somente eram disponíveis estações de trabalho gráficas. Porém, atualmente, os preços do *hardware* têm caído ao ponto de permitir que estas capacidades gráficas estejam presentes nos computadores pessoais (PCs). Hoje, processadores gráficos de alta velocidade que podem interpretar a linguagem gráfica OpenGL são acessíveis e podem dar aos PCs um poder de renderização que rivaliza com o de algumas estações de trabalho de baixo e médio custo.

Apesar de fornecerem imagens 3D de alta qualidade, os *displays* tradicionais oferecem uma capacidade limitada de imersão aos usuários. Para uma experiência de alta fidelidade, alguns AVDs utilizam dispositivos gráficos que envolvem mais completamente o usuário, bloqueando toda a entrada visual de fora do ambiente virtual. Por exemplo, pequenos *displays* gráficos são frequentemente embutidos em óculos que podem ser utilizados pelos usuários. Estes capacetes de realidade virtual, ou HMDs, apresentam imagens diretamente a frente dos olhos do usuário, bloqueando toda a luz. Um sensor magnético dentro do HMD detecta o movimento da cabeça do usuário e transfere esta informação ao processador embutido. Consequentemente, quando o usuário movimenta a cabeça, os *displays* podem refletir a mudança do ponto de visão.

Um outro *display* gráfico imersivo é conhecido como CAVE, e foi desenvolvido pela Universidade de Illinois (Cruz-Neira et al (1993)). A CAVE é essencialmente um cubo de cinco lados. O usuário se posiciona no meio do cubo, e imagens são projetadas nas paredes do lado, a frente, acima e abaixo do participante. O usuário tem um campo de visão de 270 graus completos em todas as direções. Quando o usuário caminha através do ambiente virtual, imagens atualizadas são projetadas nas paredes da CAVE, dando ao mesmo a sensação de realismo.

2.2.2.2 Dispositivos de Comunicação e Controle

Os usuários necessitam escolher e manipular objetos, e se comunicar com outros participantes no ambiente virtual. Estas tarefas são executadas através do uso de diversos dispositivos de entrada. Os dispositivos de entrada mais comuns são o teclado e o *mouse*. Com o *mouse*, o usuário é capaz de navegar pelo ambiente, mudando sua direção e rotação. O teclado fornece suporte para a comunicação textual digitada.

Porém, apesar de comuns, o *mouse* e o teclado nem sempre são muito eficientes. Para tarefas de manipulação mais precisas, uma luva (*dataglove*) pode ser usada. Esta luva possui sensores que detectam e capturam o movimento da mão e dedos.

A comunicação textual, por sua vez, distrai o usuário da completa imersão que alguns AVDs buscam atingir. Nos AVDs mais sofisticados, os usuários comunicam-se verbalmente através de microfones.

2.2.2.3 Sistema de Processamento

Os AVDs demandam uma considerável capacidade de processamento. A unidade de processamento é responsável por receber os eventos dos dispositivos de entrada do usuário e computar o efeito destas entradas sobre a posição do usuário e de outros objetos no ambiente virtual. O processador deve determinar como e quando notificar os outros participantes das mudanças iniciadas pelo usuário local, e vice-versa. Finalmente, o processador deve animar o *display* gráfico a manter uma atualização do ponto de vista do ambiente.

Portanto, um dos desafios para os projetistas de AVDs é alocar o tempo do processador entre as tarefas requeridas para manter a ilusão de presença do usuário no ambiente.

2.2.2.2 Dispositivos de Comunicação e Controle

Os usuários necessitam escolher e manipular objetos, e se comunicar com outros participantes no ambiente virtual. Estas tarefas são executadas através do uso de diversos dispositivos de entrada. Os dispositivos de entrada mais comuns são o teclado e o *mouse*. Com o *mouse*, o usuário é capaz de navegar pelo ambiente, mudando sua direção e rotação. O teclado fornece suporte para a comunicação textual digitada.

Porém, apesar de comuns, o *mouse* e o teclado nem sempre são muito eficientes. Para tarefas de manipulação mais precisas, uma luva (*dataglove*) pode ser usada. Esta luva possui sensores que detectam e capturam o movimento da mão e dedos.

A comunicação textual, por sua vez, distrai o usuário da completa imersão que alguns AVDs buscam atingir. Nos AVDs mais sofisticados, os usuários comunicam-se verbalmente através de microfones.

2.2.2.3 Sistema de Processamento

Os AVDs demandam uma considerável capacidade de processamento. A unidade de processamento é responsável por receber os eventos dos dispositivos de entrada do usuário e computar o efeito destas entradas sobre a posição do usuário e de outros objetos no ambiente virtual. O processador deve determinar como e quando notificar os outros participantes das mudanças iniciadas pelo usuário local, e vice-versa. Finalmente, o processador deve animar o *display* gráfico a manter uma atualização do ponto de vista do ambiente.

Portanto, um dos desafios para os projetistas de AVDs é alocar o tempo do processador entre as tarefas requeridas para manter a ilusão de presença do usuário no ambiente.

2.2.2.4 Rede de Comunicação

Os participantes de um AVD necessitam da rede de comunicação a fim de trocarem informações, como por exemplo a mudança de posicionamento de avatares, ou a movimentação de algum objeto. A rede também é responsável pela sincronização do estado compartilhado no AVD, tais como tempo e visibilidade. Deve também suportar a comunicação por texto e áudio entre os usuários.

Por anos, os AVDs podiam somente ser usados em universidades e instituições militares ou industriais dotadas de redes locais velozes, tais como Ethernets e *Token Ring*. Devido ao fato destas redes possuírem uma capacidade limitada e os AVDs não gerenciarem cuidadosamente o uso desta capacidade, a maioria podia suportar pouco mais de uma dúzia de usuários simultâneos. Porque a capacidade da Internet era limitada, os AVDs não podiam utilizá-la. Para suportar usuários em múltiplos locais, um AVD necessitava ser suportado por redes privadas com grande largura de banda, como foi o caso da *Defense Simulation Internet* (DSI).

O mundo das redes de computadores sofreu significativas mudanças nos últimos anos. A capacidade das redes locais foi multiplicada pelo fator 100, como as Ethernets que cresceram de 10 Megabits por segundo para suportar 1 Gigabit por segundo. Sobre estas redes locais, portanto, os AVDs rudimentares podem agora suportar centenas de usuários. Ao mesmo tempo, novos AVDs gerenciam a capacidade de rede mais cuidadosamente, tornando-se capazes de suportar milhares de participantes simultaneamente.

Os avanços na tecnologia de redes têm permitido aos AVDs saírem do confinamento das redes privadas. Como a velocidade dos modems foi quadruplicada de 14,4 para 56 kilobits por segundo, os usuários podem, agora, participar mais realisticamente dos AVDs, conectando-se, de casa, a um serviço de provedor Internet.

A Internet também tem se tornado uma plataforma viável para o desenvolvimento de AVDs. As capacidades das redes a longa distância têm aumentado consideravelmente, bem como o número de usuários da Internet. Simultaneamente, algumas das funções do AVD estão sendo passadas para o *browser Web*. Por exemplo, a linguagem *Virtual Reality Modeling Language*

(VRML) (Hartman; Wernecker (1996)) permite aos usuários receberem modelos tridimensionais interativos a partir da *World Wide Web*.

2.2.3 Taxonomia para Ambientes Virtuais Distribuídos

Os mais importantes aspectos quanto ao desenvolvimento de *software* para AVDs, envolvem a comunicação em rede, a sincronização das visões do ambiente virtual, os tipos de modelos de dados, os modelos de gerenciamento de computação, e o comportamento dos objetos (Macedonia; Zyda (1995)).

2.2.3.1 Comunicação em Rede

Diversas características da comunicação em rede devem ser consideradas para se entender os aspectos citados anteriormente. As características principais são: a largura de banda (*bandwidth*), a latência, a confiabilidade, e os esquemas de comunicação.

2.2.3.1.1 Largura de Banda

A largura de banda é a taxa na qual a rede pode entregar um dado a um computador (*host*) destinatário. A disponibilidade da largura de banda é determinada pelo tipo do meio utilizado para o transporte do dado, e é limitado pelo *hardware* usado para transmiti-lo. Por exemplo, uma rede Ethernet tradicional pode manipular 10Mbps, e a nova tecnologia de Ethernet pode suportar 100Mbps ou até 1Gps. A maior largura de banda que os cabos de fibra ótica podem transferir informações é de 10 Gps ou mais.

2.2.3.1.2 Latência

A latência de uma rede é o tempo requerido pela rede para transferir um bit de dado de um ponto a outro. A latência aparece por muitas razões. Primeiro, a transmissão de dados é governada pelo atraso da velocidade da luz. Um dado, fundamentalmente, não pode ser transportado mais rápido do que a velocidade da luz. Segundo, atrasos são introduzidos pelos próprios computadores. Leva tempo para o dado atravessar o sistema operacional, o *hardware* de rede e, então, encontrar a rede; similarmente o dado deve atravessar os mesmos níveis, ao chegar ao computador destino. Terceiro, atrasos são introduzidos pela própria rede. Geralmente os computadores emissor e destino não estão ligados diretamente. Ao invés disto, o dado deve passar por diversas intersecções de rede, ou roteadores.

A latência de rede pode variar muito. Se o dado for enviado através de uma rede local Ethernet, a latência tipicamente gira em torno de poucos milissegundos (10 a 20 ms). Porém, se a comunicação for feita usando-se a Internet, através de um modem e do sistema telefônico, sua latência será, no mínimo, 100 milissegundos. (Singhal; Zyda (1998)).

Do ponto de vista do ambiente virtual distribuído, é a latência que controla a natureza interativa e dinâmica do sistema. Se o ambiente distribuído existe para emular o mundo real, deve operar em tempo real, em termos da percepção humana (Macedonia; Zyda (1995)). Segundo Wloka (1995), sistemas que envolvem operadores humanos devem entregar pacotes com a mínima latência e gerar gráficos texturizados 3D a uma frequência de 30 a 60Hz, para garantir a ilusão de realidade. Este desafio torna-se maior em sistemas que utilizam redes a longa distância, conforme mencionado anteriormente. A latência da rede pode ser reduzida, até certo ponto, através da utilização de ligações (*links*) dedicadas (ou virtuais, usando protocolos como o *Reservation Protocol* (Deering (1993))), melhoramentos no roteador e tecnologias de *switching*, interfaces e computadores mais rápidos. Todavia, uma maior largura de banda não é necessariamente a melhor solução. A operação em velocidade de gigabits gera um novo conjunto de problemas. Novos métodos para a manipulação de congestionamento são requeridos devido a grande velocidade de propagação.

Os gargalos encontram-se, mais comumente, nas interfaces de rede, arquiteturas de memória e sistemas operacionais. O lento progresso no aumento do desempenho das interfaces de redes FDDI é um exemplo. Atualmente a velocidade de operação das memórias não acompanham o desempenho atingido pelos processadores e rede. Ao nível de sistema operacional, muitas aplicações de realidade virtual são construídas em versões comerciais do UNIX, as quais não foram projetadas para desempenho em tempo real.

2.2.3.1.3 Confiabilidade da Rede

A confiabilidade da rede é uma medida que reflete a quantidade de dado que é perdido pela rede durante o trajeto do *host* fonte para o destino. Este dado perdido pode ser dividido em duas categorias: dado interrompido e dado corrompido. Por dado interrompido pode-se entender que o dado não chegou completo ao *host* destino. Por dado corrompido entende-se que o conteúdo dos pacotes de dados é mudado durante a transmissão e o que chega ao *host* destino é basicamente inútil. Muitos pesquisadores consideram a interrupção e a corrupção de dado essencialmente equivalente, isto porque, em ambos os casos, o dado não atinge efetivamente a aplicação destino (Singhal; Zyda (1998)).

A confiabilidade da rede frequentemente força um compromisso entre a largura de banda e a latência. Isto porque, para atingi-la, os sistemas devem assumir que o dado enviado é sempre recebido corretamente, portanto existe a necessidade de periodicamente reenviar a informação. Infelizmente, para garantir a entrega, a arquitetura de rede deve usar esquemas de reconhecimento (*acknowledgment*) e recuperação de erros que introduzem grandes atrasos – principalmente em redes a longa distância e grandes sistemas distribuídos. Adicionalmente, alguns protocolos de transporte, como o *Transmission Control Protocol* (TCP) (Comer (1988)), usam mecanismos de controle de congestionamento que são inadequados para tráfego em tempo real, pois diminuem a taxa de transmissão de pacotes, se um congestionamento é detectado.

No entanto, pesquisadores estão tentando desenvolver um serviço de *multicast* que seja confiável e escalável. Como exemplo tem-se o *Reliable Multicast*

Protocol (RMP) (Whetten; Kaplan (1994)), o qual é baseado em um protocolo *token ring* situado no topo do *Internet Protocol* (IP) *multicast*. Este método usa seqüenciação e reconhecimentos negativos (NAKs). O problema com este método é a potencial implosão de NAKs, na qual um grupo de receptores enviam simultaneamente NAKs, causando congestionamento e, conseqüentemente, provocando perda de pacotes com a introdução de mais NAKs. Por isso, sistemas confiáveis normalmente não operam muito bem em tempo real.

2.2.3.1.4 Esquemas de Comunicação

O modelo de comunicação tem muito haver com a escalabilidade de um sistema. A escalabilidade pode ser mensurada pela quantidade de entidades que podem, simultaneamente, participar do sistema. Estas entidades podem ser objetos controlados pelo usuário, objetos controlados por simulação ou objetos estáticos (Araújo (1996)). Alguns esquemas escalam melhor do que outros. Entre os principais modelos de comunicação tem-se: o *broadcast*, *unicast*, *multicast* (Kirner; Mendes (1988)).

A comunicação *unicast* (*peer-to-peer* ou ponto a ponto) ocorre quando estabelece-se uma comunicação entre dois *hosts*. Este tipo de comunicação requer o estabelecimento de uma conexão para cada par de computadores (ou nós) na rede, ou seja, um total de $N*(N-1)$ conexões virtuais, para um grupo de N nós (Figura 4).

Figura 4 – Comunicação *Unicast*

Na comunicação *broadcast* (por difusão, ou um para vários), o dado é enviado para todos os *hosts*, ou seja, uma mensagem enviada por um *host* é recebida por todos os outros, que poderão ignorá-la se a mesma não for de seu interesse, conforme ilustrado na Figura 5 (Tanenbaum (1995)). Esta característica, presente no protocolo IP, é altamente custosa, em termos de desempenho, uma vez que o *host* deve interromper sua operação e executar esta verificação ao nível do sistema operacional.

Figura 5 – Comunicação *Broadcast*

Uma outra forma de comunicação é a *multicast*, na qual grupos de *hosts* de tamanhos arbitrários se comunicam pela rede, via uma única transmissão de um *host* fonte. A comunicação *multicast* fornece os serviços de transmissão um-para-muitos e muitos-para-muitos para aplicações do tipo teleconferência e simulação distribuída, nas quais existe uma necessidade de comunicação com diversos *hosts* (mas não necessariamente todos) simultaneamente (Figura 6).

Diversos pesquisadores têm utilizado *multicast* no suporte de ambientes virtuais, como por exemplo a arquitetura SPLINE, da *Mitsubishi Electric Research Labs* (Barrus et al.(1996)).

Figura 6 – Comunicação *Multicast*

2.2.3.2 Visões do Usuário de um AVD

A visão (ou perspectiva) que um usuário ou processo tem do ambiente virtual são geralmente representadas através de janelas. Existem dois paradigmas principais para visão em ambientes distribuídos: a visão síncrona e a visão assíncrona.

- **Visão síncrona:** neste tipo de paradigma, tem-se várias máquinas controlando visões diferentes de um mesmo ambiente. As imagens exibidas por estas máquinas devem ser sincronizadas de forma a dar a ilusão de conjunto (Figura 7). Como por exemplo, um simulador de voo, onde máquinas diferentes controlam a visão da frente, da direita e da esquerda da cabine. As imagens são coordenadas de maneira a parecer que todas fazem parte de uma única visão da cabine. Alguns AVDs que usam a visão síncrona são: RAVEN (Cater; Huffman (1995)) e o CAVE. Por exigir alta confiabilidade e menor latência, a visão síncrona é restrita a redes locais.

Figura 7 – Um Ambiente com Visão Síncrona

- **Visão Assíncrona:** este paradigma é mais geral, e permite que múltiplos usuários tenham controle individual sobre “quando” e “o que” podem ver, concorrentemente, no ambiente virtual (Figura 8). Os usuários podem estar fisicamente separados por uma rede local ou uma rede a longa distância. Sua consciência da presença dos demais participantes, se são representados por um objeto, é executada dentro do ambiente virtual. Como exemplo de AVDs que utilizam a visão assíncrona, tem-se o NPSNET (Mackey (1991)).

Figura 8 – Um Ambiente com Visão Assíncrona

2.2.3.3 Modelos de Dados

A determinação do modelo de dados a ser utilizado em um AVD é, talvez, a decisão de construção mais difícil de ser tomada, pois afeta o escalonamento, os requisitos de comunicação e a confiabilidade do sistema.

Existem diversos conceitos para a distribuição persistente ou semi-persistente de dados. As consideradas mais relevantes são:

- **Modelo da Base de Dados Homogênea Replicada:** neste modelo tem-se uma base de dados homogênea que contém todas as informações sobre a geometria, texturas, e comportamento de todos os objetos que integram o mundo virtual. Esta base de dados é, então, replicada em todos os *hosts* que participam do AVD. Uma vantagem deste modelo é que as mensagens são geralmente pequenas, uma vez que os participantes só são informados sobre as mudanças de estado dos objetos, como por exemplo, mudanças de posição e colisões entre objetos. As desvantagens são que a base de dados de cada usuário tende a crescer com o aumento do conteúdo do mundo virtual.
- **Modelo da Base de Dados Centralizada e Compartilhada:** um outro modelo que pode ser utilizado é o da base de dados centralizada e compartilhada. Neste modelo, as máquinas dos participantes comunicam-se com um único servidor central, o qual mantém a base de dados com todas as informações e estado do mundo virtual. A principal vantagem deste modelo reside na facilidade de manutenção da consistência da base de dados. A principal desvantagem é que o uso de um servidor centralizado para mundos virtuais 3D, limita o número de participantes devido à contenção de E/S e a manutenção de uma base de dados de objetos, em tempo real. O servidor central torna-se rapidamente um “gargalo” do sistema, já que deve retransmitir o estado de cada participante à todos os demais.
- **Modelo da Base de Dados Distribuída, Compartilhada, Ponto a Ponto:** a idéia por trás deste modelo é fazer com que o sistema distribuído simule as arquiteturas de memória compartilhada. A base de dados é homogênea, distribuída e completamente replicada entre as máquinas dos participantes. Como exemplo tem-se o sistema DIVE (Carlsson; Hagsand (1993)), no qual a base de dados inteira é dinâmica e utiliza protocolos *multicast* confiáveis para replicar novos objetos. A desvantagem com o método é que este escala com dificuldade,

devido aos custos de comunicação associados com a manutenção da confiabilidade e da consistência dos dados, principalmente sobre redes a longa distância.

- **Modelo da Base de Dados Distribuída, Compartilhada, Cliente/Servidor:** outro modelo que pode ser utilizado é uma variação do modelo cliente/servidor, no qual a base de dados é particionada entre os clientes e a comunicação é mediada por um servidor central. Neste sistema, servidor central possui o conhecimento de qual parte do mundo virtual reside em cada cliente. Toda vez que um objeto se move através do mundo virtual, a base de dados é atualizada por intermédio do servidor. Todavia, em um mundo virtual dinâmico de grande escala, os servidores tendem a se tornar os “gargalos” de E/S do sistema, aumentando a latência inerente ao ambiente.

2.2.3.4 Gerenciamento da Computação

A computação, tanto quanto os dados, também pode ser distribuída. Como computação, no caso, entende-se qualquer tarefa envolvendo um objeto específico. Os modelos de gerenciamento mais importantes são:

- **Distribuição Completa (ou Total):** neste modelo, todas as operações sobre um objeto devem ser executadas no mesmo nó que mantém os dados sobre aquele objeto. Se um objeto deseja executar uma operação sobre outro, deve, então, enviar uma mensagem para o outro processo. A alocação de processos aos nós pode ser otimizada através da utilização de algoritmos para balanceamento de carga. Monitorando-se o consumo de recursos e padrões de comunicação, pode-se derivar uma alocação ótima. Isto permitirá que muitos objetos que se comunicam com frequência entre si, possam ser colocados no mesmo nó – esta movimentação de processos é comumente conhecida como *migração*. Esta abordagem funciona bem em sistemas fortemente acoplados, porém a latência e a baixa largura de banda encontrada nos sistemas fracamente acoplados podem reduzir sua eficiência.
- **Distribuição Parcial:** este método é similar ao da distribuição completa, exceto que o estado do objetos é normalmente adquirido usando-se técnicas de

replicação parcial de dados e as mudanças são feitas localmente. Porém, não existe duplicação de esforço computacional.

- **Replicação Parcial:** para compensar ligações de comunicações mais lentas, por exemplo, como nos sistemas fracamente acoplados, é possível replicar o estado da computação em alguns ou todos os nós. Estes processos-réplica são geralmente usados para aproximar o comportamento do objeto usando-se um método conhecido como *dead-reckoning*. O processo que executa a completa simulação daquele objeto também processa este modelo em paralelo e quando os dois diferem de uma quantidade pré-definida, uma cópia das variáveis de estado reais daquele objeto é enviada para todos os processos-réplica. As subseqüentes aproximações são, então, baseadas na última atualização. O *dead-reckoning* usa um modelo simplificado do comportamento do objeto. Geralmente as variáveis-chaves típicas usadas neste modelo são posição e velocidade, as quais podem ser linearmente extrapoladas para fornecer uma aproximação de baixa fidelidade. Uma aproximação de maior fidelidade pode ser alcançada através da incorporação de outras variáveis, tais como aceleração linear e velocidade angular, as quais são frequentemente utilizadas em objetos com comportamento altamente dinâmico, como por exemplo, aeronaves (Harvey et al. (1991) e McCarty et al. (1994)). Esta técnica é muito efetiva para a redução da quantidade de largura de banda requerida, mas o comportamento produzido no avatar remoto do objeto pode ser suficientemente diferente do normal, ao ponto de ser notado pelo usuário. Isto pode, é claro, ser melhorado pelo aumento da complexidade da aproximação, porém existe a necessidade de se equilibrar cuidadosamente a simulação completa e a aproximada.
- **Replicação Total:** a simulação de cada objeto em cada nó pode ser requerida se a simulação está sendo executada em uma rede a longa distância (fracamente acoplada). Neste modelo, somente a informação que muda o comportamento dos objetos espelhados é enviada, fazendo com que todos os cálculos sejam feitos localmente. Por conseguinte, o comportamento parece correto em todo lugar (apesar de poder não ser exatamente o mesmo, no tempo), mas ao custo da duplicação de computação.

2.2.3.5 Comportamento dos Objetos

Atualmente ainda é tema de discussão o que consiste o “comportamento” do objeto, e como este deve ser modelado. No estrito senso da orientação a objetos, os dados são atributos, e os métodos manipulam os atributos de uma forma pré-definida, por exemplo, mudando a posição a todo instante. Portanto, combinando-se dados e métodos, consegue-se a impressão de “comportamento”. No entanto, a carga computacional requerida para suportar este comportamento do objeto pode ser muito alta.

É possível classificar o comportamento do objeto como *determinístico* ou *não-determinístico*. Em geral, objetos que não necessitam de entradas (via dispositivos de entrada) são determinísticos, ao passo que aqueles objetos que necessitam, inclusive aqueles sob o controle de humanos, são não-determinísticos. Por exemplo, as decisões tomadas por um carro-robô pode ser determinado durante seu avanço, porém um carro virtual sendo dirigido por um humano, em um simulador de direção, não pode ser previsto (Hawkes (1993)). A capacidade de prever o comportamento significa que é possível melhorar a comunicação e a latência do sistema.

Segundo Roehl (1995), o comportamento determinístico pode ser subdividido em duas sub-categorias: *estática* e *animada*. Similarmente, o comportamento não-determinístico pode ser *Newtoniano* ou *inteligente*. O estado de um objeto estático é constante e, portanto, 100% previsível, a todo momento; já um objeto animado muda seu estado a todo instante, mas ainda é previsível. Um objeto Newtoniano interage com seu ambiente, mas não de uma maneira direta, enquanto um objeto inteligente pode ter um comportamento complexo e tão imprevisível quanto o de um ser humano.

De maneira similar, Roehl apresenta 4 níveis de comportamento, que podem ser usados para classificar o tipo de distribuição usada:

- **Nível 0:** Modificação direta dos atributos de um objeto (estático).
- **Nível 1:** Mudança nos atributos de um objeto, no tempo (animado).
- **Nível 2:** Séries de chamadas ao nível 1, de maneira a se conseguir realizar uma tarefa (Newtoniano).
- **Nível 3:** Tomada de decisão de alto nível (inteligente).

A forma mais básica de distribuição do comportamento está situada entre os níveis 0 e 1, quando informações como posição e orientação são transmitidas a toda atualização da simulação. O método do *dead-reckoning* situa-se entre os níveis 1 e 2. A tentativa de distribuir o comportamento ao nível mais elevado é problemática, a menos que se garanta que o estado da simulação em cada nó seja exatamente o mesmo, a todo instante. De fato, os níveis 2 e 3 não podem ser implementados em *software*, devem ser providos através da interação humana.

Capítulo 3 – Visão Geral dos Principais Ambientes Virtuais Distribuídos

Neste capítulo são levantados os AVDs considerados mais representativos encontrados na literatura, na atualidade. O objetivo deste levantamento é oferecer uma visão geral dos AVDs atuais, segundo suas características mais importantes. Levando em consideração este objetivo, concentrou-se a observação destes sistemas nas suas soluções para dois problemas: como estes resolveram o problema da modelagem e como executam determinado ambiente virtual. O primeiro problema é uma área mais abstrata e, na teoria, pode ser independente do mecanismo usado para a distribuição e do *software* de suporte ao AV. Porém, isto raramente ocorre na realidade. Algumas vezes a implementação dirige o sistema de modelagem usado e vice-versa. Seria ideal tratar-se destes aspectos separadamente, todavia isto é muito difícil, uma vez que descrevendo-se um aspecto pode implicar na referência as características de outro.

3.1 Metodologia Utilizada

Dada a grande quantidade de Ambientes Virtuais Distribuídos em uso e em desenvolvimento atualmente, e a diversidade de suas características e objetivos, considerou-se adequado fazer este levantamento segundo os aspectos descritos a seguir. Esta metodologia é a mesma utilizada em (Snowdon; West (1994)).

- **Características Relacionadas a Distribuição e Concorrência:** em um ambiente distribuído, a forma com que o paralelismo e a distribuição são abordados tendem a determinar a arquitetura do sistema. As características destacadas foram: o modelo adotado para a estruturação do sistema; se existe algum tipo de replicação (quanto a base de dados, simulação, etc); o modelo de comunicação usado no sistema; se existe suporte para computação distribuída; a granularidade do paralelismo; a unidade de paralelismo; se existe suporte ao balanceamento de carga; e o ambiente de rede.
- **Características dos Componentes Intrínsecos dos Ambiente Virtuais:** o mundo virtual e seus artefatos (ou objetos), ou seja, as características dos artefatos que existem no mundo virtual. As características destacadas foram: se o sistema permite a criação de múltiplos mundos virtuais; se permite a criação de tipos de artefatos dinâmicos; se permite a criação dinâmica de artefatos; se os artefatos criados podem ser ativos (ou seja, podem ter um comportamento independente da aplicação); se permite a edição de geometria.
- **Características Relacionadas ao Tratamento Dado ao Usuário e as Aplicações:** se o usuário e aplicação são tratados da mesma forma pelo sistema; a interação do usuário (ou do programador) com o sistema; e aspectos de E/S.

3.2 Ambientes Virtuais Distribuídos Levantados

3.2.1 Sistema dVS, versão 2.0 (*Division Ltd.*) (Snowdon; West (1994))

A) Distribuição e Concorrência

- **Modelo utilizado:** Usa o da base de dados compartilhada, com dados replicados.
Existe um processo central que é utilizado para efetivar as mudanças no banco de dados compartilhado. Este processo é conhecido como “Diretor”.
- **Replicação:** Parcial. Somente os processos que explicitamente registram interesse em algum tipo de dado ou evento recebe cópias deste. Os processos são livres para modificar seus dados locais sem consulta ao processo “Diretor”, porém, para alterar os dados da base de dados compartilhado devem fazê-lo por intermédio dele.

- **Comunicação:** Segue o modelo cliente/servidor e ponto a ponto.
- **Computação Distribuída:** Não há suporte para a distribuição de uma aplicação específica entre diversos processadores.
- **Paralelismo:** Granularidade grossa. O sistema possui processos que executam tarefas padrão (como E/S, detecção de colisão, etc.), e que podem executar em processadores separados.
- **Unidade de Paralelismo:** Processos paralelos independentes, chamados de “atores”.
- **Balanceamento de Carga:** Não existe na forma dinâmica. O programador deve fazer qualquer arranjo para o balanceamento de carga que seja necessário.
- **Ambiente de Rede:** Heterogêneo. (Testado principalmente em estações de trabalho Silicon *Graphics*).

B) Mundo Virtual e seus Artefatos

- **Múltiplos Mundos Virtuais:** Não permite a construção de múltiplos mundos virtuais. O modelo do mundo virtual consiste de um conjunto de processos agindo sobre uma base de dados, a qual contém os artefatos (ou objetos) presentes naquele mundo.
- **Tipos de Artefatos Dinâmicos:** Não permite.
- **Criação Dinâmica de Artefatos:** Permite.
- **Artefatos Ativos:** Não permite. O comportamento de um artefato não é independente da aplicação que o utiliza.
- **Edição de Geometria:** Permite, e é feita *off-line*.

C) Usuários e Aplicações

Existe uma separação explícita entre o usuário e a aplicação. Esta separação é útil em ambiente multi-aplicações, pois diversas representações de usuários diferentes podem ser contruídas e reusadas por várias tarefas independentes da aplicação. O sistema permite, portanto, a existência de múltiplas aplicações.

Quanto a E/S, existe um processo padrão, conhecido como *body actor*, que se comunica com os dispositivos de E/S.

3.2.2 Sistema MR - *The Minimal Reality Toolkit* (University of Alberta - Canada) (Snowdon; West (1994) e Wang et al. (1995))

A) Distribuição e Concorrência

- **Modelo utilizado:** Utiliza o modelo de estruturas de dados compartilhadas. Estas estruturas podem ser implementadas em processadores separados, com espaços de endereçamento distintos. Ponteiros são proibidos, e portanto, o compartilhamento de estruturas dinâmicas de dados é dificultada.
- **Replicação:** Parcialmente replicadas.
- **Comunicação:** Usa os modelos mestre/escravo e ponto a ponto. Uma aplicação MR é arranjada como uma hierarquia de processos, sendo que cada processo comunica-se diretamente com seus pais. O processo no topo da hierarquia é responsável pelo controle de todo o sistema. A comunicação é feita através de estruturas de dados compartilhadas, com um processo agindo como produtor de dados e outro como consumidor.
- **Computação Distribuída:** Permite. Uma aplicação pode ser arranjada, como já mencionado, como uma hierarquia de processos. A decomposição da aplicação em processos é estática e fixada em tempo de compilação.
- **Paralelismo:** Granularidade grossa.
- **Unidade de Paralelismo:** Processos (dependentes de uma aplicação).
- **Balanceamento de Carga:** Não existe.
- **Ambiente de Rede:** Heterogêneo (redes de estações de trabalho Silicon *Graphics* e DEC).

B) Mundo Virtual e seus Artefatos

- **Múltiplos Mundos Virtuais:** Não permite.
- **Tipos de Artefatos Dinâmicos:** Permite.
- **Criação Dinâmica de Artefatos:** Permite.
- **Artefatos Ativos:** Não permite. O comportamento do artefato é completamente dependente da aplicação.
- **Edição de Geometria:** Permite. No MR, o componente conhecido como “modelo geométrico” representa o mundo virtual.

C) Usuários e Aplicações

No MR não existe uma representação explícita para o usuário - a natureza do usuário é determinada pela aplicação.

Quando os dados da aplicação estão em um estado consistente, o “modelo geométrico” é atualizado e mostrado ao usuário através de outro componente chamado “componente de apresentação”.

3.2.3 Sistema VEOS - *The Virtual Environment Operating Shell (University of Washington's Human Interface Technology Laboratory)* (Bricken; Coco (1994) e Singh et al. (1995))

A) Distribuição e Concorrência

- **Modelo utilizado:** Utiliza o modelo de programação orientado a entidades. No VEOS, tudo é uma “entidade” (o ambiente, o participante, os dispositivos de *hardware*, os programas, e todos os objetos dentro do mundo virtual). Uma entidade pode ser definida como sendo uma coleção de recursos que exibem um comportamento dentro de um ambiente. Em um ambiente composto de entidades, qualquer entidade pode cessar de funcionar (por exemplo, se o nó que está suportando aquela entidade pára de funcionar) sem afetar o resto do ambiente. Todas as entidades são organizacionalmente idênticas. Somente sua estrutura, seus detalhes internos, diferem. As entidades são processos multiplexados em um único nó.
- **Replicação:** Parcial. Cada estação de trabalho que pertence ao sistema distribuído executa um Núcleo (*Kernel*) VEOS. Fazendo parte de cada *Kernel* existe um componente chamado NANCY, que é responsável pelo gerenciamento de uma base de dados distribuída direcionada a padrões.
- **Comunicação:** Utiliza o modelo ponto a ponto. No VEOS o componente responsável pela comunicação inter-nó é o *TALK*. O *TALK* conta com chamadas ao sistema operacional UNIX para a passagem de mensagem. Ele também conecta os processos UNIX, que estão distribuídos nas redes de estações de trabalho, para formar um multiprocessador virtual. A passagem

de mensagem é somente um dos tipos de entidade de comunicação suportados pelo *TALK*, dependendo do contexto, este mecanismo pode ser configurado para se comportar como memória compartilhada ou outro regime de comunicação. O *TALK* utiliza duas primitivas assíncronas ponto a ponto para a passagem de mensagem: *send* e *receive*. Também usa as funções LISP *throw* e *catch* para processar o compartilhamento em um único nó.

- **Computação Distribuída:** Permite. A unidade fundamental de organização no *Kernel* do VEOS é o nó. Cada nó, neste contexto, corresponde a exatamente um processo UNIX. Os nós são mapeados para os processos UNIX que, por sua vez, são idealmente mapeados para os processadores das estações de trabalho. Coleções de nós formam um sistema distribuído, o qual é gerenciado pelo componente VEOS chamado FERN. O FERN gerencia conjuntos de uniprocessadores (por exemplo uma rede local de estações de trabalho) como um *pool* de nós.
- **Paralelismo:** Granularidade grossa.
- **Unidade de Paralelismo:** Processos (ou seja, entidades). O processo entidade é não-preemptivo; é destinado a executar somente curtas tarefas discretas, rendendo-se rápida e voluntariamente a outras entidades que compartilham o mesmo nó.
- **Balanceamento de Carga:** Não mencionado, se existe.
- **Ambiente de Rede:** Heterogêneo. O VEOS é independente de plataforma, tendo sido testado em: DEC 5000, SUN 4, Silicon *Graphics* e Indigo.

B) Mundo Virtual e seus Artefatos

- **Múltiplos Mundos Virtuais:** Permite. O VEOS é um *framework* ao nível de usuário para a prototipagem de aplicações RV distribuídas.
- **Tipos de Artefatos Dinâmicos, Criação Dinâmica de Artefatos e Artefatos Ativos:** As entidades diferem dos objetos. Objetos consistem de dados estáticos e funções conhecidas como métodos. Objetos encapsulam funcionalidade e podem ser organizados hierarquicamente, a fim de que o esforço do programador seja minimizado. Em contraste, entidades são objetos que incluem *interface* e recursos computacionais. Cada entidade

possui os seguintes componentes: um nome único; uma partição privada da base de dados global; qualquer número de processos; e qualquer número de interações.

- **Edição de Geometria:** Permite.

C) Usuários e Aplicações

O componente do VEOS responsável pela *interface* com o usuário, iniciação dos nós e ligações, é o SHELL. O SHELL é o administrador do *Kernel* VEOS. Ele despacha iniciações, manipula interrupções e gerencia memória.

A *interface* do programador com o SHELL é a linguagem LISP, acrescida de funções de especializadas do *Kernel* para o gerenciamento da base de dados e de comunicação. O LISP permite ao usuário configurar o ambiente VEOS e todas as funções associadas.

3.2.4 Sistema DIVE - *Distributed Interactive Virtual Environment (Swedish Institute of Computer Science)* (Snowdon; West (1994), Bendford et al. (1995), Singh et al. (1994), Singh et al. (1995) e Carlsson; Hagsand (1993))

A) Distribuição e Concorrência

- **Modelo utilizado:** Usa o modelo da base de dados compartilhada. O sistema pode ser visto como uma base de dados compartilhada através da rede, com um conjunto de processos fazendo atualizações concorrentes.
- **Replicação:** Completa. A base de dados é completamente replicada.
- **Comunicação:** Utiliza o modelo *multicast*. Como o DIVE não tem noção de quais objetos da base de dados estão sendo manipulados, as atualizações e sinalizações de eventos devem ser difundidos, pois a base de dados é totalmente replicada e não há condições de se determinar quais os processos que estão atualmente interessados nestes eventos. Os tipos das mensagens utilizadas são fixos. O sistema é fracamente acoplado baseado no UNIX e nos protocolos de rede Internet.
- **Computação Distribuída:** Permite distribuição parcial.

- **Paralelismo:** Granularidade grossa.
- **Unidade de Paralelismo:** Processos. Um processo pode representar uma aplicação ou um usuário.
- **Balanceamento de Carga:** Não existe na forma dinâmica.
- **Ambiente de Rede:** Heterogêneo.

B) Mundo Virtual e seus Artefatos

- **Múltiplos Mundos Virtuais:** Permite. O DIVE suporta vários mundos virtuais concorrentes e fornece “portais”, que permitem a um usuário se mover de um mundo virtual para outro. Cada mundo possui a mesma mecânica fornecida pelo sistema, apesar de que todo o comportamento que pode distinguir os mundos virtuais é providenciado explicitamente pelas aplicações. Um mundo DIVE consiste de um conjunto de objetos com os quais os processos DIVE podem se conectar. Processos DIVE conectados ao mesmo mundo são idênticos em seus conteúdos.
- **Tipos de Artefatos Dinâmicos:** Não permite.
- **Criação Dinâmica de Artefatos:** Permite.
- **Artefatos Ativos:** Permite de forma limitada. É permitido que uma máquina de estado finito seja associada a cada objeto, fornecendo ao mesmo um comportamento simples, o qual pode ser simulado localmente sem a intervenção da aplicação. Comportamentos mais complexos, no entanto, exigirão um processo para gerenciá-lo.
- **Edição de Geometria:** Permite.

C) Usuários e Aplicações

Existe uma separação explícita entre o usuário e a aplicação. O DIVE liga a representação do usuário com sua E/S em um processo. O usuário DIVE é representado por um processo chamado “visualizador” (*visualizer*), o qual apresenta uma visão do mundo virtual em que o usuário está.

O DIVE define o conceito de “veículo” (*vehicle*), o qual mapeia os dados de um dispositivo físico de entrada para ações no mundo virtual. O usuário pode selecionar o veículo apropriado enquanto o visualizador estiver executando. Existem diversos veículos no DIVE; alguns exploram as

diferentes configurações dos dispositivos de E/S e outros usam os mesmos dispositivos para fornecer diferentes interpretações das ações do usuário.

O DIVE permite a existência de diversas aplicações independentes.

Do ponto de vista do sistema, o DIVE trata os usuários fundamentalmente da mesma forma: como processos.

3.2.5 Sistema BrickNet Toolkit (*Institute of Systems Science - National University of Singapore*) (Singh et al. (1994), Singh et al. (1995) e Macedonia et al. (1995))

A) Distribuição e Concorrência

- **Modelo utilizado:** Modelo orientado a objeto, com base de dados distribuída. Os mundos virtuais são compostos de objetos que encapsulam suas propriedades de rede, comportamento e gráficas. São criados em um ambiente interpretativo e orientado a objeto, que permite aos objetos serem adicionados, destruídos e modificados em tempo de execução.
- **Comunicação:** Modelo cliente/servidor. Com base neste modelo, a base de dados é particionada entre os clientes e as comunicações são mediadas através de um servidor central. Quando uma entidade se move através de um ambiente virtual, sua base de dados é atualizada através de um *Object Request Broker* (ORB) em um servidor que possui o conhecimento de qual cliente mantém aquela parte do mundo. Quanto ao protocolo de comunicação, a BrickNet foi implementada utilizando o UDP. Como mencionado, um servidor BrickNet age como um ORB e servidor de comunicações para seus clientes. Os servidores distribuídos pela rede comunicam-se uns com os outros para prover um serviço aos clientes, sem que estes possuam o conhecimento explícito dos múltiplos servidores ou da comunicação inter-servidores. Os clientes são incapazes de localizar os servidores que executam seus pedidos. Os clientes executam assincronamente sem ter que esperar pelos dados emitidos pelo servidor. As mensagens são enviadas apenas para os clientes que possuem o objeto afetado (esquema diferente do *broadcast* para todos os clientes).

- **Computação Distribuída:** Permite. A BrickNet é dirigida a um conjunto de projeto baseado em rede, onde uma tarefa completa é distribuída em diversos nós. Cada nó é responsável por parte do projeto e compartilha seus objetos de projeto com outros, a fim de facilitar a colaboração na execução da tarefa maior.
- **Paralelismo:** Granularidade grossa.
- **Unidade de Paralelismo:** Processos (clientes e servidores).
- **Balanceamento de Carga:** Não mencionado.
- **Ambiente de Rede:** Heterogêneo (redes de estações de trabalho Silicon *Graphics*)

B) Mundo Virtual e seus Artefatos

- **Múltiplos Mundos Virtuais:** Permite. A BrickNet foi projetada para a criação de diversos mundos virtuais que operam em estações de trabalho conectadas através da rede, formando um sistema fracamente acoplado. A BrickNet oferece aos clientes o recurso dos “portais dinâmicos”, para que estes possam acompanhar as atividades atuais no mundo virtual.
- **Tipos de Artefatos Dinâmicos, Criação Dinâmica de Artefatos e Artefatos Ativos:** O sistema permite que os objetos sejam criados e modificados em tempo de execução. Não é mencionado se permite a existência de objetos ativos. Um cliente BrickNet geralmente possui dois tipos de objetos: o local e o remoto. Os objetos locais são locais ao mundo virtual e não são compartilhados com outros clientes da rede. Os objetos remotos habilitam aos clientes a compartilharem suas informações com os demais.
- **Edição de Geometria:** Permite. Pode ser realizada através do objeto *geometry brick*, que faz parte do “nível de suporte de interação” da estrutura de um processo cliente. Existem duas formas de se definir a geometria: lendo-a de um arquivo ou usando uma forma (*shape*) pré-definida.

C) Usuários e Aplicações

Um cliente BrickNet consiste de três níveis: o nível de suporte de interação; o nível de conhecimento RV e o nível de comunicação.

O nível de suporte de interação corresponde à *interface* com o usuário para o mundo virtual. O usuário interage com o mundo virtual através de

dispositivos e técnicas de interação que fazem parte deste nível. Por exemplo, este nível decide como um objeto virtual pode ser escolhido e movido.

Os *device drivers* executam como processos separados. Os dados dos dispositivos, lidos através destes *drivers*, são coletados pelos clientes BrickNet quando necessários.

3.2.6 Sistema: EM - *Environment Manager (University of Alberta - Canada)* (Wang et al. (1995) e Singh et al. (1995))

A) Distribuição e Concorrência

- **Modelo utilizado:** Modelo orientado a objeto. O EM implementa os mesmos conceitos de compartilhamento de objetos da BrickNet. Em termos de sua arquitetura, o EM une diversos objetos *Object Modeling Language* (OML) em um mundo virtual. O OML é uma linguagem de programação procedural. Um objeto OML corresponde a uma classe C++ e contém código para gerar a geometria do objeto 3D, para controlar sua aparência e comportamento.
- **Replicação:** Completa. O EM usa a arquitetura replicada: cada mundo virtual tem uma cópia completa do conjunto de objetos descritos em OML.
- **Comunicação:** Usa o modelo ponto a ponto. O EM está encarregado da iniciação e gerenciamento da aplicação local, bem como das comunicações entre outras aplicações. No EM, as mensagens são enviadas somente para os mundos virtuais que possuem uma instância ou o objeto afetado e expressaram o interesse em receber as atualizações. Com a finalidade de reduzir a largura de banda, as instâncias transmitem apenas as variáveis compartilhadas cujos estados mudaram. As mensagens são enviadas somente para os mundos relevantes. Como no *dead-reckoning* da SIMNET, o EM suporta a simulação local do comportamento das instâncias compartilhadas. Similarmente a BrickNet, o EM também fornece um sistema de “amortecimento” de mensagens, para eliminar as comunicações desnecessárias. O EM usa o *MR Toolkit Peer Package*, que é uma extensão

do MR *Toolkit*, para fornecer facilidades ao nível de conexão e permitir às múltiplas aplicações MR independentes, trocarem dados através da Internet.

- **Computação Distribuída:** Permite. O EM é construído no topo do MR. O MR, por sua vez, possui suporte para computação distribuída.
- **Paralelismo:** Granularidade grossa.
- **Unidade de Paralelismo:** Processo.
- **Balanceamento de Carga:** Não mencionado.
- **Ambiente de Rede:** Heterogêneo.

B) Mundo Virtual e seus Artefatos

- **Múltiplos Mundos Virtuais:** Permite. O EM é uma ferramenta de alto nível para construção de ambientes virtuais mono-usuário e multi-usuário.
- **Tipos de Artefatos Dinâmicos, Criação Dinâmica de Artefatos, Artefatos Ativos e Edição de Geometria:** No EM, um compilador OML produz uma versão interpretável da especificação do objeto (chamado de protótipo de objeto), e o interpretador OML é ligado ao programa de aplicação em tempo de execução. Pode criar um número arbitrário de instâncias em tempo de execução e tem controle de alto nível sobre seus comportamentos. Diferentes instâncias de um mesmo objeto podem ter diferentes geometrias e comportamentos, uma vez que a especificação do objeto tenha sido parametrizada.

C) Usuários e Aplicações

Cada usuário no ambiente virtual multi-usuário executa um EM local, o qual se comunica com outros EMs através da rede. O programador interage com o EM através de um arquivo de descrição (para o estabelecimento da configuração do mundo EM) e programas escritos na linguagem OML.

3.2.7 Sistema: NPSNET (Naval Postgraduate School - USA) (Snowdon; West (1994) e Macedonia et al. (1995))

A) Distribuição e Concorrência

- **Modelo utilizado:** Usa o modelo da simulação distribuída. Neste sistema não existe base de dados central. Cada objeto é simulado em seu próprio *host*.
- **Replicação:** Completa. Cada *host* do usuário (jogador) executa uma cópia da simulação. Internamente a simulação em cada *host* é dividida em um conjunto de processos cooperantes especializados.
- **Comunicação:** Modelo ponto a ponto e *multicast*. Cada vez que um objeto da simulação muda de estado, ele envia um modelo *dead-reckoning* do seu movimento corrente para os outros *hosts* na simulação. O NPSNET não possui um processo central para distribuir atualizações, portanto, ele as entrega à todos os *hosts*. O NPSNET usa o protocolo DIS, o qual não é dirigido a estado, portanto, o NPSNET deve transmitir o estado completo de um objeto, ao invés de simplesmente as partes do estado do objeto que sofreram mudanças. As atualizações são transmitidas usando o IP *multicast*.
- **Computação Distribuída:** Permite.
- **Paralelismo:** Granularidade grossa.
- **Unidade de Paralelismo:** Processo.
- **Balanceamento de Carga:** Não tem. O NPSNET simula todos os objetos em todos os *hosts*. Apesar dos objetos não locais serem simulados usando o esquema do *dead-reckoning*, o sistema não possui balanceamento de carga - está implícito que cada *host* é capaz de manipular a base de dados completa.
- **Ambiente de Rede:** Heterogêneo.

B) Mundo Virtual e seus Artefatos

- **Múltiplos Mundos Virtuais:** Não permite. O NPSNET é mais uma aplicação do que um sistema de propósito geral para suportar aplicações.
- **Tipos de Artefatos Dinâmicos:** Permite.
- **Criação Dinâmica de Artefatos:** Permite.
- **Artefatos Ativos:** Não permite. O comportamento do artefato é dependente da aplicação.

- **Edição de Geometria:** Permite.

C) Usuários e Aplicações

O NPSNET é especificamente um sistema de simulação para treinamento militar.

3.2.8 Sistema AVIARY (*University of Nottingham - University of Manchester - UK*) (Snowdon; West (1994))

A) Distribuição e Concorrência

- **Modelo utilizado:** Representação orientada a objeto, com base de dados distribuída (não utiliza a representação da base de dados compartilhada). O sistema pode ser representado como uma coleção de objetos autônomos comunicantes, executando concorrentemente.
- **Replicação:** Parcial.
- **Comunicação:** Modelo ponto a ponto e cliente/servidor. O sistema de objeto do AVIARY associa cada objeto com um único ID (identificação), que é usado para toda a comunicação com este. O ID permanece constante, mesmo se o objeto é movido para outro processador, o que fornece transparência de acesso, localização e migração. Toda a comunicação é feita através da passagem de mensagem. A abordagem escolhida é a comunicação assíncrona, porém a comunicação síncrona (bloqueante com resposta associada) também é possível. Como as mensagens são enviadas para objetos que residem em diferentes processadores, todos os parâmetros são passados por valor. Os tipos das mensagens não são fixos e novas mensagens podem ser definidas em tempo de execução, quando novas classes e objetos são criados. Um objeto deve explicitamente registrar interesse em um particular objeto ou evento, de maneira a receber atualizações, reduzindo assim as comunicações.
- **Computação Distribuída:** Permite. Para distribuir a computação entre os processadores, o AVIARY suporta uma classe de objetos conhecidos como “servidores de objetos”. Um servidor de objeto fornece um ambiente para

os objetos mais “leves”, gerenciando sua criação, destruição, comunicação, gerenciamento de memória e escalonamento.

- **Paralelismo:** Granularidade grossa.
- **Unidade de Paralelismo:** O objeto. Um objeto pode representar um artefato (referenciados aqui como *daemon*), ou um *driver*.
- **Balanceamento de Carga:** Pode permitir. Como os servidores de objeto estão cientes do código e dos dados de cada objeto no *host*, o balanceamento de carga através da migração de objetos é possível.
- **Ambiente de Rede:** Heterogêneo. Executa em redes de *Transputers*, *Suns* e *Silicon Graphics*.

B) Mundo Virtual e seus Artefatos

- **Múltiplos Mundos Virtuais:** Permite. Cada mundo é representado através de um objeto chamado *World Object*. Este é usado como um ponto de conexão para um mundo particular e também armazena alguma informação global, como por exemplo, os IDs dos objetos presentes no mundo e os valores dos atributos globais específicos de cada mundo.
- **Tipos de Artefatos Dinâmicos:** Não mencionado.
- **Criação Dinâmica de Artefatos:** Permite.
- **Artefatos Ativos:** Não mencionado.
- **Edição de Geometria:** Permite.

C) Usuários e Aplicações

O AVIARY modela um usuário como uma entidade separada do sistema RV, logicamente equivalente a uma aplicação. Não são definidas formas particulares de dados que um usuário pode entrar, ou de saída que pode ser apresentada a ele. O modelo AVIARY define os conceitos de “objetos de entrada” e “objetos de saída”, que gerenciam *device drivers* e permitem a outros objetos expressar interesse por eles. Quando o estado de um dispositivo de entrada muda, o objeto de entrada informa a todos que expressaram algum interesse.

3.2.9 Sistema SIMNET - *Simulation Network* (projeto realizado pelo DARPA, projetado e contruído pela BBN Laboratories) (Blau et al. (1992) e Singh et al. (1995))

A) Distribuição e Concorrência

- **Modelo utilizado:** Utiliza os modelos de orientação a objetos e eventos, com base de dados replicada totalmente. Na SIMNET, mundos virtuais consistem de objetos que interagem através da difusão de uma série de eventos.
- **Replicação:** Completa. Cada *host* mantém sua própria base de dados, sem fazer qualquer esforço para manter sua consistência.
- **Comunicação:** Modelo ponto a ponto, *broadcast* ou *multicast*. Todos os simuladores são ligados via Ethernet e o modelo de comunicação é baseado no paradigma do *dead-reckoning*. Cada objeto na simulação possui uma máquina *host* que processará sua dinâmica. Um objeto iniciando um evento não calcula quais objetos podem ser afetados por ele. É de responsabilidade do objeto receptor determinar quais os eventos de seu interesse. Para minimizar o processamento de comunicação e os requisitos de largura de banda, os objetos transmitem somente as mudanças em seu comportamentos. Até que uma atualização seja recebida, a nova posição do objeto remoto é extrapolada, a partir do último estado reportado pelos objetos.
- **Computação Distribuída:** Permite.
- **Paralelismo:** Granularidade grossa.
- **Unidade de Paralelismo:** Processos.
- **Balanceamento de Carga:** Não mencionado.
- **Ambiente de Rede:** Homogêneo.

B) Mundo Virtual e seus Artefatos

- **Múltiplos Mundos Virtuais:** Não permite.
- **Tipos de Artefatos Dinâmicos, Criação Dinâmica de Artefatos e Artefatos Ativos:** Não mencionado.
- **Edição de Geometria:** Permite.

C) Usuários e Aplicações

Permite que múltiplos usuários interajam em simulações de pequena escala (menos de 300 jogadores interativos).

3.2.10 Sistema MASSIVE - *Model, Architecture, and System for Spatial Interaction in Virtual Environment*. (University of Nottingham - UK, University Manchester - UK) (Bendford et al. (1995))

A) Distribuição e Concorrência

- **Modelo utilizado:** Não é mencionado o modelo de gerenciamento de dados usado pelo sistema. Já o modelo de processamento usado no MASSIVE, é o dos processos computacionais independentes que se comunicam através de conexões ponto a ponto. Cada processo computacional pode suportar qualquer número de *interfaces*, sendo que cada um deles é caracterizado pela combinação de chamadas remotas de procedimento (RPC), atributos, e fluxos de dados (*streams*) para suportar uma mistura de funções RV distribuídas, e meio contínuo (por exemplo, áudio).
- **Replicação:** Não mencionado. O MASSIVE foi implementado utilizando idéias do modelo de interação espacial, sendo que todos os objetos são cercados por uma extensão espacial chamada “aura”.
- **Comunicação:** Modelo ponto a ponto, executando sobre protocolos de transporte padrão da Internet.
- **Computação Distribuída:** Permite.
- **Paralelismo:** Granularidade grossa.
- **Unidade de Paralelismo:** Processos independentes.
- **Balanceamento de Carga:** Não mencionado.
- **Ambiente de Rede:** Heterogêneo.

B) Mundo Virtual e seus Artefatos

- **Múltiplos Mundos Virtuais:** Permite. O MASSIVE suporta múltiplos mundos virtuais conectados via “portais”. Cada mundo virtual pode ser habitado por

muitos usuários concorrentes que interagem através de *interfaces* de gráficos, áudio e texto.

- **Tipos de Artefatos Dinâmicos, Criação Dinâmica de Artefatos, Artefatos Ativos:** Não mencionado.
- **Edição de Geometria:** Permite.

C) Usuários e Aplicações

Os objetivos do MASSIVE são: suportar tantos usuários simultâneos quanto for possível; suportar a interação entre usuários cujos equipamentos possuem diferentes capacidades e que, portanto, empregam estilos de *interface* diferentes.

Cada usuário pode incorporar suas próprias características gráficas através de um arquivo de configuração.

3.2.11 Sistema RAVEN - *Remote Access Virtual Environment Network* (NASA) (Cater, Huffman (1995))

A) Distribuição e Concorrência

- **Modelo utilizado:** Base de dados centralizada. Um processo central (RAVEN *main*) é responsável por coletar os dados de entrada dos sensores, bem como os dados de estado, e fornecê-los aos demais processos.
- **Replicação:** Não existe.
- **Comunicação:** Modelo ponto a ponto. A comunicação entre processos é feita via *sockets*.
- **Computação Distribuída:** Permite. Os computadores são interconectados através de uma rede Ethernet. Um arquivo de configuração, lido pelo processo controlador central, permite ao operador executar programas de visualização em qualquer computador da rede.
- **Paralelismo:** Granularidade grossa. Os processos são projetados para executar em sistema operacional UNIX.
- **Unidade de Paralelismo:** Processo.

- **Balanceamento de Carga:** Existe, porém não é dinâmico: o arranjo dos processos deve ser feito diretamente pelo usuário.
- **Ambiente de Rede:** Heterôgeneo.

B) Mundo Virtual e seus Artefatos

- **Múltiplos Mundos Virtuais:** Não permite. O sistema RAVEN foi criado com o objetivo específico de permitir o treinamento de astronautas para o reparo do telescópio espacial Hubble.
- **Tipos de Artefatos Dinâmicos, Criação Dinâmica de Artefatos, Artefatos Ativos, Edição de Geometria:** Não mencionado.

C) Usuários e Aplicações

O sistema RAVEN foi criado com o objetivo específico de permitir o treinamento de astronautas para o reparo do Telescópio Espacial Hubble. Foi projetado para permitir apenas dois usuários *trainees*.

3.2.12 Sistema PARADISE (*Distributed Systems Group at Stanford University*) (Singhal; Zyda (1998))

A) Distribuição e Concorrência

- **Modelo utilizado:** Utiliza o modelo da base de dados compartilhada.
- **Replicação:** Total.
- **Comunicação:** Modelo *multicast*, cliente/servidor. Usa o IP *multicast*, associando um diferente endereço *multicast* a cada objeto ativo. Os computadores atualizam os objetos localmente, do mesmo modo que no SIMNET e DIS. Porém, uma hierarquia de servidores de Área de Interesse (AOI – *Area of Interest*) coletam “subscrições AOI” de cada computador. Os servidores monitoram as posições dos objetos e notificam os computadores sobre aqueles objetos que foram subscritos pelos grupos *multicast*.
- **Computação Distribuída:** Distribuição parcial.
- **Paralelismo:** Granularidade grossa.
- **Unidade de Paralelismo:** Processo.

- **Balanceamento de Carga:** Não mencionado.
- **Ambiente de Rede:** Heterogêneo.

B) Mundo Virtual e seus Artefatos

- **Múltiplos Mundos Virtuais:** Permite.
- **Tipos de Artefatos Dinâmicos:** Não mencionado.
- **Criação Dinâmica de Artefatos:** Permite.
- **Artefatos Ativos:** Permite.
- **Edição de Geometria:** Não mencionado.

C) Usuários e Aplicações

O sistema PARADISE endereça questões importantes da arquitetura do *software* e de rede, para a criação de ambientes com milhares de usuários.

3.2.13 Sistema WAVES (*Waterloo Virtual Environment System*) (Hawkes (1996))

A) Distribuição e Concorrência

- **Modelo utilizado:** Utiliza o modelo de base de dados distribuída.
- **Replicação:** Parcial.
- **Comunicação:** Modelo ponto a ponto e cliente/servidor. As comunicações entre os computadores são realizadas sobre conexões virtuais, mediadas por um número de “gerentes de mensagens”. As conexões também podem ser filtradas para que somente as mensagens de interesse possam ser enviadas aos computadores. Aos gerentes de mensagens é conferida a habilidade de delegar ligações ponto a ponto entre computadores, em circunstâncias especiais, tais como a transmissão de sinal de vídeo.
- **Computação Distribuída:** Permite. Cada computador participante simula um subconjunto de objetos e provê certos serviços, como por exemplo, detecção de colisão, renderização, etc.
- **Paralelismo:** Granularidade grossa.
- **Unidade de Paralelismo:** Processos.

- **Balanceamento de Carga:** Permite o balanceamento dinâmico de carga.
- **Ambiente de Rede:** Heterogêneo.

B) Mundo Virtual e seus Artefatos

- **Múltiplos Mundos Virtuais:** Não permite.
- **Tipos de Artefatos Dinâmicos:** Permite.
- **Criação Dinâmica de Artefatos:** Permite.
- **Artefatos Ativos:** Permite.
- **Edição de Geometria:** Não mencionado.

C) Usuários e Aplicações

Os usuários são representados como objetos denominados “objóides”, os quais contém um estado encapsulado e um modelo comportamental explícito.

3.2.14 Sistema SPLINE – *Scalable Platform for Large Interactive Networked Environments* (MERL – *Mitsubish Eletric Research Laboratory*) (Barrus et al. (1996))

A) Distribuição e Concorrência

- **Modelo utilizado:** Modelo da base de dados distribuída.
- **Replicação:** Parcial. O SPLINE distribui o modelo do mundo, mantendo uma cópia parcial em cada processo. Cada cópia contém as partes do modelo do mundo que estão suficientemente próxima ao correspondente foco de atenção do processo. Para alcançar uma baixa latência, as cópias são mantidas aproximadamente, mas não exatamente, consistentes.
- **Comunicação:** Modelo *multicast*. A comunicação inter-processo é realizada através do envio de mensagens *multicast* descrevendo os objetos no modelo do mundo, e da recepção de mensagens de outros processos sobre as mudanças feitas remotamente. Como no sistemas SIMNET, cada mensagem descreve o estado corrente de um objeto no mundo, e é usada pelos processos que a recebem para atualizar sua cópia do modelo do mundo.

- **Computação Distribuída:** Permite. O SPLINE utiliza o conceito de “Locales”. Este conceito baseia-se na idéia que enquanto um mundo virtual pode ser muito grande, apenas uma pequena parte dele pode ser visualizada por um único usuário, em um dado momento. Os “Locales” dividem um mundo virtual em partes que podem ser processadas separadamente. É importante salientar que esta divisão é puramente uma questão de implementação; não é visível ao usuário.
- **Paralelismo:** Granularidade grossa.
- **Unidade de Paralelismo:** Processo.
- **Balanceamento de Carga:** Permite.
- **Ambiente de Rede:** Heterogêneo

B) Mundo Virtual e seus Artefatos

- **Múltiplos Mundos Virtuais:** Não mencionado.
- **Tipos de Artefatos Dinâmicos:** Permite.
- **Criação Dinâmica de Artefatos:** Permite.
- **Artefatos Ativos:** Permite.
- **Edição de Geometria:** Permite.

C) Usuários e Aplicações

O SPLINE permite a participação de um grande número de usuários em uma mesma simulação. Os usuários podem se comunicar utilizando a linguagem natural. Podem, também, fazer modificações temporárias ou permanentes e estender o ambiente em tempo de execução.

3.2.15 Sistema VERN –*Virtual Environment RealTime Network (Institute for Simulation and Training – University of Central Florida)* (Blau et al. (1992))

A) Distribuição e Concorrência

- **Modelo utilizado:** Modelo da base de dados distribuída.
- **Replicação:** Total.

3.2.16 Sistema WorldToolkit (Sense8 Corporation) (Sense8 (1998))

A) Distribuição e Concorrência

- **Modelo utilizado:** Base de dados visual. Uma simulação pode ser criada agrupando-se nós (*nodes*) em um grafo de cena hierárquico, o qual rege como uma simulação é renderizada e permite todas as facilidades da preservação de estado. Cada nó do grafo de cena representa uma parte da simulação.
- **Replicação:** Total. Cada computador participante de uma simulação possui sua própria cópia da simulação inteira.
- **Comunicação:** Modelo *multicast*. Utiliza os protocolos IP/UDP sobre uma rede Ethernet padrão DIX (DEC/Intel/Xerox). A comunicação entre vários computadores em uma mesma simulação é estabelecida usando-se um endereço IP de grupo válido (endereço Internet Classe D).
- **Computação Distribuída:** Permite. É possível desenvolver-se aplicações que possam, assincronamente, se comunicar sobre uma rede Ethernet. Isto possibilita a construção de simulações distribuídas que executem em microcomputadores e estações de trabalho.
- **Paralelismo:** Granularidade grossa.
- **Unidade de Paralelismo:** Processo.
- **Balanceamento de Carga:** Não existe.
- **Ambiente de Rede:** Heterogêneo.

B) Mundo Virtual e seus Artefatos

- **Múltiplos Mundos Virtuais:** Permite. Pode-se ter vários grafos de cena, cada um representando um mundo virtual completo, ou parte de um. Porém, só pode existir um Universo (*Universe*). O Universo é o *container* de todos os objetos WorldToolkit (WTK), tais como geometrias, nós, pontos de vista (*viewpoints*), sensores, etc.
- **Tipos de Artefatos Dinâmicos:** Permite.
- **Criação Dinâmica de Artefatos:** Permite. O WTK fornece funções para criação dinâmica de geometrias.

- **Artefatos Ativos:** Permite. Os objetos podem ter propriedades e comportamento semelhantes ao do mundo real.
- **Edição de Geometria:** A geometria pode ser especificada *off-line*, utilizando-se um modelador para a criação de um arquivo de geometria, o qual pode ser carregado em tempo de execução. Permite, também, a criação dinâmica de geometrias, através do oferecimento de funções específicas.

C) Usuários e Aplicações

Os usuários podem interagir com os mundos virtuais através de uma variedade de sensores de entrada, indo do mouse simples até dispositivos com “seis graus de liberdade”. Os usuários podem experimentar estes mundos com um *display* (que age como uma janela móvel para o mundo) ou usando um rastreador de posição, um HMD, ou um *display* estereoscópico.

3.3 Modelos e Características Principais

As principais características observadas nos sistemas estudados foram sintetizados na Tabela I (a) e (b).

3.4 Uma Outra Abordagem de Classificação

Segundo Bricken e Coco (1994), os Sistemas de desenvolvimento de Ambientes Virtuais podem, ainda, ser classificados como *toolkits*, sistemas integrados ou sistemas híbridos.

- **Toolkits:** consistem geralmente de um conjunto de funções para a montagem de eventos virtuais através de programação. A linguagem mais comum para os *toolkits* é a “C”; freqüentemente a funcionalidade está em um nível computacional mais baixo, como em *package device driver*. Dos sistemas levantados, os que se encaixam nesta categoria são: DIVE, MR, BrickNet, EM e WorldToolkit .
- **Sistemas Integrados:** em contraste, são direcionados a não programadores e a *experts*, como os artistas gráficos. Os sistemas integrados escondem os detalhes

computacionais de baixo nível através de opções para construção de eventos mais visuais e intuitivas. Os sistemas que se encaixam nesta categoria, daqueles estudados, são: SIMNET, NPSNET, dVS, MASSIVE, AVIARY, RAVEN, VERN e PARADISE.

- **Sistemas Híbridos:** são aqueles sistemas que podem conter características dos dois grupos descritos anteriormente. Encaixam-se nesta categoria os sistemas: VEOS, SPLINE e WAVES.

Tabela I (a) – Principais Características dos AVDs Levantados

CARACTERÍSTICAS		QVS	MR Toolkit	VLOS	DIVE	BHNet	DIS/SIMNET	NPSNET	EM
COMUNICAÇÃO	Mecanismo(s) de transporte	Ponto a ponto	Ponto a ponto	Ponto a ponto	Multicast	Ponto a ponto	Ponto a ponto, broadcast ou multicast	Ponto a ponto, ponto, multicast	Ponto a ponto, ponto, multicast
	Largura de Banda atingida	10Mbps+	10Mbps+	10Mbps+	10Mbps+	Não especificada	Não	10Mbps+	10Mbps+
	Estrutura(s)	Cliente/Servidor	Mestre/Escravo	Peer	Cliente/Servidor	Cliente/Servidor	Peer	Peer	Peer
GERENCIAMENTO DE DADOS	Organização	BD compartilhada com replicação parcial	Estruturas de dados compartilhadas com replicação parcial	BD distribuída com replicação parcial	BD compartilhada com replicação total	BD distribuída com replicação parcial	BD com replicação total	BD com replicação total	BD com replicação total
	Organização	Distribuição Parcial	Distribuição parcial/total	Distribuição total	Distribuição parcial	Distribuição total	Distribuição parcial	Distribuição parcial	Distribuição total
MODELAGEM DO AV	Permite múltiplos Mundos Virtuais	Não	Não	Sim	Sim	Sim	Não	Não	Sim
	Suporte ao Usuário	Múltiplos	Múltiplos	Múltiplos	Múltiplos	Múltiplos	Múltiplos	Múltiplos	Múltiplos
	Tipo utilizado	Heterogêneo	Heterogêneo	Heterogêneo	Heterogêneo	Heterogêneo	Heterogêneo	Heterogêneo	Heterogêneo
AMBIENTE DE REDE	Existência	Não existe	Não existe	Desconhecido	Não existe	Desconhecido	Não existe	Não existe	Desconhecido

Tabela I (b) – Principais Características dos AVDs Levantados (continuação)

CARACTERÍSTICAS		MASSIVE	WAVES	PARADISE	VERN	SPLINE	AVIARY	RAVEN	WorldToolKit
COMUNICAÇÃO	Mecanismo(s) de transporte	Ponto a ponto	Ponto a ponto	Multicast	Broadcast	Multicast	Ponto a ponto e multicast	Ponto a ponto	Multicast
	Largura de Banda atendida	10Mbps+	14Kbps+	10Mbps+	10Mbps+	10Mbps+	10Mbps+	10Mbps+	10Mbps+
	Estrutura(s)	Peer	Cliente/Servidor	Cliente/Servidor	Peer	Peer	Cliente/Servidor	Cliente/Servidor	Peer
	Organização	Não mencionada	BD distribuída com replicação parcial	BD com replicação total	BD distribuída com replicação total	BD distribuída com replicação parcial	BD distribuída com replicação parcial	BD centralizada	BD Visual com replicação total
GERENCIAMENTO DE DADOS									
GERENCIAMENTO DA COMPUTAÇÃO	Organização	Distribuição total	Distribuição completa e parcial	Distribuição parcial	Distribuição total	Distribuição total	Distribuição total	Distribuição parcial/total	Distribuição total
MODELAGEM DO AV	Permite múltiplos Mundos Virtuais	Sim	Sim	Sim	Sim	Não especificado	Sim	Não	Sim
	Suporte ao Usuário	Múltiplos	Múltiplos	Múltiplos	Múltiplos	Múltiplos	Múltiplos	2 usuários	Múltiplos
	Tipo utilizado	Heterogêneo	Heterogêneo	Heterogêneo	Heterogêneo	Heterogêneo	Heterogêneo	Heterogêneo	Heterogêneo
	Existência	Desconhecido	Permite	Desconhecido	Permite	Permite	Permite	Permite	Não existe

Capítulo 4 - Computação Distribuída Orientada a Objeto

Dentre os diversos modelos existentes atualmente, escolheu-se, para este trabalho, o do modelo da computação distribuída orientada a objeto, para a construção de Ambientes Virtuais Distribuídos. A caracterização deste modelo, bem como a apresentação da arquitetura OMG-CORBA (*Common Object Request Broker Architecture*), como uma alternativa para infra-estrutura de comunicação para AVDs, são apresentadas neste capítulo. Uma vez que a arquitetura CORBA é na verdade uma especificação padrão, fez-se necessário, na prática, utilizar-se implementações que fossem compatíveis com a mesma. Portanto, ao final deste capítulo são apresentadas as características das implementações CORBA usadas, bem como uma comparação entre estas e a especificação CORBA original.

4.1 Objetos Distribuídos e Processamento Distribuído

O processamento distribuído prevê o compartilhamento de aplicações e funcionalidade do mesmo modo que os dados são atualmente compartilhados. Porém, ao invés de se ter aplicações monolíticas, as aplicações locais em um ambiente de processamento distribuído exerce o papel de controladoras: coordenam as atividades das funções que produzem não somente dados, mas também o código

para manipulá-los. O código e os dados são produzidos de forma que possam ser acessados por múltiplos processadores em um ambiente heterogêneo, sem se preocupar com sua localização física.

Portanto, tem-se como requisitos essenciais para um ambiente de computação distribuída:

- a) A aplicação deve ser capaz de estabelecer as capacidades de processamento que ela requer;
- b) A aplicação deve ser capaz de enviar parâmetros e dados para o processo, bem como, receber resultados deste;
- c) Os parâmetros e resultados dos processos devem ser expressivos para as várias arquiteturas de máquinas dentro do ambiente; e
- d) O processo deve ser capaz de executar em diferentes implementações de ambientes e linguagens.

Cabe notar que existem ainda outros requisitos, tais como: segurança, descrição de *interface*, como a informação é transmitida, etc.

Comparando-se a idéia dos objetos distribuídos com as idéias básicas do processamento distribuído, nota-se que os primeiros consistem em uma tecnologia disponível para os sistemas distribuídos, da mesma forma que tem sido para o modelo cliente/servidor, a multimídia, o processamento de documentos, e outras aplicações. No caso dos sistemas distribuídos, preocupações como: nomeação, conversão de espaço de endereçamento, protocolos de transporte e descrição de *interface*, são maiores. Cada uma destas áreas é complexa, e os objetos podem auxiliar a abstrair e lidar com esta complexidade.

Um aspecto que causa confusão é a diferença entre as "tecnologias de componentes ao nível de aplicação" e os "modelos de objetos" que a suportam. No nível mais baixo existem os "modelos de objetos" tais como o *System Object Model* (SOM/DSOM) da IBM e o *Component Object Model* (COM) da Microsoft. Estas tecnologias ao nível de sistema são basicamente endereçadas para resolver o problema de acoplamento binário entre uma aplicação e os objetos que utilizam. Tomando como exemplo uma aplicação escrita na linguagem C++, que usa diversas classes cujos métodos estão implementados em Bibliotecas de Ligação Dinâmicas (DLLs). Convenientemente, uma DLL não faz parte do código da aplicação, e então pode ser atualizada ou alterada sem afetar a aplicação, uma vez que sua *interface*

permanece a mesma. Infelizmente, esta idéia falha se as mudanças são feitas em uma classe baseada em DLL, de forma a alterar o tamanho do objeto. Neste caso, a aplicação que executa a chamada pode necessitar ser recompilada, mesmo se seu código fonte não foi mudado textualmente.

Para servir de ligação binária entre as implementações do cliente e do objeto servidor, um modelo de objeto é definido em um nível de abstração que assume que a linguagem do modelo do objeto e sua tradução sejam transparentes. O modelo é geralmente expresso em termos de uma Linguagem de Definição de *Interface* (IDL), a qual é processada de forma independente da linguagem de implementação. Uma IDL é o caminho para se definir uma *interface* para um serviço; freqüentemente, o mecanismo gera um código *stub* (procedimento local) que pode ser chamado pela aplicação. Na implementação final, ela descreve a *interface* para o código que executa a função e gera *stubs* que podem ser usados para executar as operações (Leavens; Cheon (1993)). O mecanismo do RPC (Nelson (1981)) pode preencher a lacuna entre os dois. Desde que a IDL possa ser traduzida para a linguagem de implementação, esta abordagem preenche o requisito de independência de linguagem: os *stubs* podem ser gerados em qualquer linguagem para a qual a IDL tenha mapeamento.

A principal diferença entre as definições de *interfaces* para os mecanismos procedurais e para os modelos de objetos, é que no modelo de objetos a *interface* é parte de uma construção semântica que representa o objeto. Dependendo do modelo específico, esta construção pode possuir algumas ou todas as características e vantagens esperadas dos objetos, incluindo o encapsulamento, herança e polimorfismo.

4.2 Objetos Interoperáveis

A interoperabilidade pode ser definida como sendo a habilidade de trocar funcionalidade e dados interpretáveis entre duas entidades de *software*. Pode ser definida em termos de quatro requisitos: comunicação, geração de pedidos, formato de dados, e semântica. As entidades de *software* requerem um canal de comunicação com um protocolo de comunicação comum. Através deste canal as entidades

necessitam ser capazes de formular e transmitir um pedido interpretável para funções ou dados. O resultado do pedido deve retornar para a entidade solicitante. O intercâmbio de dados também implica em um requisito para um formato de dados que possa ser analisado pelas entidades. As entidades devem entender o pedido e dados através de alguma forma de tradução semântica.

A principal idéia por trás dos objetos interoperáveis é transpor os limites existentes. Nos modelos de programação orientada a objetos atuais, existe um forte acoplamento binário entre uma aplicação e as classes de objetos que ela usa.

A primeira fronteira a ser transposta é a do espaço de endereçamento. Um "modelo de objeto inter-processo" permite a um processo em um espaço de endereçamento pedir serviço de um objeto em outro, ou dois objetos compartilharem um objeto em um terceiro espaço de endereçamento.

A próxima fronteira é a máquina. Qualquer modelo de objeto inter-processo deve ser capaz de traduzir os dados associados com os pedidos entre modelos de memória. Deve, ainda, ser capaz de localizar o objeto servidor, estabelecer comunicação com ele, empacotar os pedidos e parâmetros e enviá-los, esperar pelos resultados, desempacotá-los e traduzi-los, e devolvê-los à aplicação. Somente as tecnologias que atravessarem a barreira entre o modelo inter-processo/inter-processador podem ser chamadas de "tecnologias de objetos distribuídos".

As outras duas fronteiras são a linguagem de programação e o sistema operacional. Estas, por sua vez, não têm muito a haver com o fato de uma tecnologia ser distribuída ou não.

O objetivo final a ser alcançado é um modelo que permita que vários objetos possam ser escritos em qualquer linguagem, possam ser compartilhados por aplicações escritas em qualquer outra linguagem, executando em qualquer máquina em uma rede, e sobre qualquer sistema operacional (Betz (1994) e Schmidt; Vinoski (1995)).

A principal disputa pela consolidação de modelos de objetos interoperáveis está entre a OMG - *Object Management Group* (OMG (1991)) e a Microsoft. A OMG é um consórcio de mais de 300 empresas de *software* e *hardware*. Estas empresas, em companhia do Object Design, foram as autoras do *Common Object Request Broker Architecture* (CORBA). O CORBA especifica a arquitetura de um *Object Request Broker* (ORB), cujo trabalho é habilitar e regular a interoperabilidade

entre objetos e aplicações. O ORB é parte de uma visão maior chamada de *Object Management Architecture* (OMA). A Microsoft, por sua vez, desenvolveu o *Object Linking and Embedded* (OLE). O OLE é uma tecnologia de integração de componentes, ao nível de aplicação, que não possui uma contrapartida real no mundo da OMG. O OLE foi construído sobre uma base chamada *Componente Object Model* (COM), a qual executa as mesmas tarefas do ORB, porém em escala diferente, usando técnicas diferentes. E, também, as idéias da Microsoft de um modelo de objeto diferem grandemente do resto da indústria. De fato, toda a base de comparação para as duas tecnologias está na promessa de que ambas, no futuro, possuirão capacidades similares.

4.3 Arquitetura de Gerenciamento de Objetos da OMG: CORBA

A OMG prevê um barramento de interconexão comum (o CORBA), através do qual os componentes clientes utilizam os serviços necessários a todos os componentes (incluindo a nomeação, persistência, eventos e transações), e facilidades comuns para a colaboração entre os mesmos (Orfali; Harkey (1995) e Mowbray; Zahavi (1995)).

Os quatro elementos chaves da arquitetura OMG (Figura 9) são:

- **O ORB (Object Request Broker):** É o barramento de interconexão de objetos. Os clientes são separados dos mecanismos usados para a comunicação, ativação, ou armazenamento dos objetos servidores. A arquitetura CORBA define a *Interface Definition Language* (IDL) e *interfaces* para programação de aplicações (*Application Programming Interface* – API) que habilitam a interação cliente/servidor com uma implementação específica do ORB, e também, como os ORBs de diversos fabricantes interoperam.
- **Serviços de Objetos (COSS-Common Object Service Specifications):** Empacotados como componentes com *interfaces* IDL, estes serviços estendem as capacidades do ORB. A OMG adotou os seguintes serviços: nomeação, notificação de eventos, persistência, gerenciamento do ciclo de vida, transações, controle de concorrência, relacionamentos, externalização, consulta, licença, propriedades, segurança, coleção, iniciação, *trader* e tempo.

- **Facilidades Comuns (OMG Common Facilities):** Estas coleções de componentes escritos em IDL definem as regras de engajamento para os objetos de aplicação. Eles podem ser do tipo horizontal ou vertical. As facilidades horizontais endereçam quatro disciplinas: *interface* do usuário, gerenciamento de informação, gerenciamento de sistemas, e gerenciamento de tarefas. Os serviços de *interface* do usuário, como o OLE e o OpenDoc, governam as atividades de tela, tais como a edição. Os serviços de gerenciamento de informação remontam os mecanismos do OLE e OpenDoc, para o armazenamento de documentos compostos e troca de dados. Os serviços de gerenciamento de sistemas definem as *interfaces* usadas para gerenciar, instrumentar, instalar, configurar, operar e reparar os objetos distribuídos. Os serviços de gerenciamento de tarefas incluem: transações longas e *scripting*, entre outras. No domínio das facilidades verticais, *interfaces* escritas em IDL suportarão um conjunto de objetos interagentes especializados por saúde, varejo, finanças, e outros domínios.
- **Objetos de Aplicação:** Estes são objetos específicos para as aplicações do usuário final. Para participar do intercâmbio mediado pelo ORB, todos devem ser definidos usando a IDL. Objetos de aplicação, é claro, são construídos no topo dos serviços oferecidos pelo ORB, as facilidades comuns, e os serviços de objetos.

Figura 9 - Arquitetura CORBA

4.3.1 O Cliente ORB

A arquitetura CORBA permite *interfaces* estáticas e dinâmicas. Os *stubs* IDL do cliente fornecem as *interfaces* estáticas para os serviços dos objetos. Estes *stubs* pré-compilados, gerados pelo compilador IDL, definem como os clientes invocam os serviços correspondentes nos servidores.

Com as APIs de invocação dinâmicas, por outro lado, é possível (em tempo de execução) descobrir um serviço que se quer invocar, obter sua definição, emitir uma chamada parametrizada a este, e receber uma resposta dele.

Um repositório de *interfaces* APIs permite ao cliente obter e modificar as descrições de todos os componentes de *interface* armazenados, os métodos que eles suportam, e os parâmetros que eles requerem. A arquitetura CORBA chama estas descrições de "assinatura do método". O repositório de *interface* armazena, atualiza, e gerencia as definições de *interface* de objetos, e seus programas usam suas APIs para acessar e atualizar esta informação.

Os ORBs fornecem identificadores globais chamados "IDs de repositório", que única e globalmente identificam um componente e sua *interface* através de múltiplos ORBs e repositórios. Os IDs de repositório são cadeias de caracteres (*strings*) únicas geradas pelo sistema que forçam uma consistência de nomes através dos repositórios - nenhuma colisão de nomes é permitida. Eles são gerados por meio do *Distributed Computing Environment/Unique User Identifier* (DCE/UUID) ou um prefixo único fornecido pelo usuário acoplado aos nomes estabelecidos pela IDL.

A *interface* ORB, usada de forma parecida pelo cliente e servidor, oferece uma variedade de APIs úteis. Por exemplo, existem APIs que convertem uma *string* em uma referência a um objeto. Estas chamadas são convenientes no caso de armazenamento ou comunicação de referências de objetos.

Com o suporte para invocações cliente/servidor estáticas e dinâmicas e um repositório de *interfaces*, o CORBA é mais poderoso e flexível do que a RPC.

As invocações estáticas são fáceis de programar, mais rápidas, e auto-documentadas. As invocações dinâmicas, apesar de mais difíceis de se programar, oferecem um máximo de flexibilidade e são essenciais quando as aplicações devem descobrir serviços em tempo de execução (Maffeis; Schmidt (1997)).

4.3.2 O Servidor ORB

Os servidores não podem diferenciar as invocações estáticas das dinâmicas. A mesma semântica de mensagem é aplicada em ambos os casos. O ORB localiza um adaptador de objeto, transmite os parâmetros, e transfere o controle para a implementação do objeto através do *stub* IDL do servidor, chamado de "esqueleto" (*skeleton*).

Os esqueletos estáticos oferecem *interfaces* para cada serviço exportado pelo servidor. Estes *stubs*, como aqueles do cliente, são criados usando um compilador IDL. Um esqueleto estático fornece um suporte rígido para os métodos definidos em IDL de uma classe de objeto particular.

As *interfaces* de esqueletos dinâmicos oferecem um mecanismo de ligação (*binding*) em tempo de execução para os servidores (Vinoski (1997)). O esqueleto dinâmico inspeciona os parâmetros de uma mensagem recebida para determinar um objeto alvo e um método. Esta técnica é útil para construir pontes entre ORBs (Figura 10).

Figura 10 - Uma Ponte Genérica entre ORBs

Ela pode, também, ser utilizada por interpretadores e linguagens de *scripts* que têm de fabricar implementações de objetos "em tempo de execução". Os esqueletos dinâmicos podem receber invocações estáticas ou dinâmicas dos clientes.

O adaptador de objeto situa-se acima dos serviços de comunicação do núcleo do ORB, aceitando pedidos para serviços provenientes dos objetos servidores. Ele fornece o ambiente em tempo de execução para o instanciamento de objetos servidores, passando pedidos para eles, e associando a eles IDs de objetos (referências aos objetos). O adaptador de objeto também registra as classes por ele suportadas e suas instâncias de tempo de execução (isto é, objetos) com o repositório de implementação. A arquitetura CORBA especifica que cada ORB deve suportar um adaptador padrão chamado Adaptador de Objeto Básico (BOA - *Basic Object Adapter*). Os servidores podem, porém, suportar mais do que um adaptador de objetos.

O repositório de implementações lista as classes que um servidor suporta, os objetos por ele instanciados, e seus IDs. Ele também serve como um lugar comum para armazenamento de informação de rastreamento, rastros de auditoria, segurança, e outros dados administrativos.

4.3.3 A Interoperabilidade da Arquitetura CORBA

A arquitetura CORBA incorporou a interoperabilidade através da especificação do Internet *Inter-ORB Protocol* (IIOP) - isto é, um protocolo TCP/IP, com algumas trocas de mensagens definidas pela arquitetura CORBA, que serve como um *backbone* comum (Figura 11). A parte de troca de mensagem do *backbone* suporta representações de dados comum para todos os tipos IDL OMG, referência a objetos interoperáveis, e formatos de mensagens comuns e semântica otimizada para trocas de ORBs (Cottman; Cottman (1998)). Cada ORB compatível ao CORBA deve entender o IIOP nativamente ou ser capaz de se ligar, através de uma ponte, a ele (Dreyfus (1998)).

Obs: os componentes
sombreados são
essenciais
para a compatibilidade
com o CORBA 2.0

Figura 11 – Estrutura de Interoperabilidade da Arquitetura CORBA

A arquitetura CORBA especifica o DCE como o primeiro de muitos protocolos opcionais inter-ORBs, chamados *Environment-Specific Inter-ORB Protocols* (ESIOP). Ao nível semântico, as implementações DCE e Internet ORB usam os mesmos protocolos de mensagens. O DCE ESIOP fornece um ambiente mais robusto para ORBs de missão crítica, requerendo características avançadas, como a segurança Kerberos e RPC autenticado.

A arquitetura CORBA também define como construir pontes ORB-para-ORB. Quando um cliente de um ORB chama um servidor de um outro ORB, a nova *interface* de esqueleto dinâmico (*Dinamic Skeleton Interface – DSI*) transmite o pedido para o ORB alvo, e então a *interface* de invocação dinâmica invoca o objeto alvo naquele ORB. Esta tecnologia dinâmica é também bem apropriada para a construção de *gateways* para barramentos não CORBA, como o COM e o Barramento Distribuído AppWare.

4.3.4 Os Serviços de Objeto da Arquitetura CORBA

Os serviços de objeto fornecem um caminho único para a construção de um suporte de comunicação personalizado. Provedores de componentes podem usar heranças múltiplas para integrar seu próprio código com os serviços de objeto da arquitetura CORBA. Os componentes ganham persistência herdando-as das *interfaces* IDL.

O OMG define atualmente padrões para 16 serviços de objetos (Figura 12):

1. O serviço de ciclo de vida define como criar, copiar, mover e destruir componentes.
2. O serviço persistência fornece uma *interface* simples para o armazenamento de componentes em uma variedade de *containers*, incluindo bases de dados de objetos, bases de dados relacionais, e arquivos simples.
3. O serviço de nomeação habilita componentes a localizar outros componentes. O serviço permite a objetos serem ligados a diretórios de rede ou contextos de nomeação, incluindo o DCE da OSF, e o *Network Information Service* (NIS) da Sun.
4. O serviço de eventos permite aos componentes no barramento armazenarem dinamicamente eventos de seus interesses. O serviço define um objeto bem conhecido chamado de "canal de evento", que coleta e distribui eventos entre componentes.
5. O serviço de concorrência fornece um gerente de bloqueio, que pode obter o bloqueio em nome de transações ou *threads*.
6. O serviço de transação fornece uma coordenação de efetivação (*commit*) de duas fases entre componentes recuperáveis, usando tanto transações simples quanto aninhadas.
7. O serviço de relacionamento fornece uma forma de criar associações (ou *links*) entre componentes. Ele também fornece mecanismos para percorrer o *link* que agrupa estes componentes. O serviço pode ser usado para forçar a integridade referencial, e para rastrear relacionamentos contidos e qualquer outro tipo de ligação entre componentes.
8. O serviço de externalização fornece uma forma padrão de dirigir o fluxo de dados para dentro ou para fora do objeto.

9. O serviço de consulta (*Query*) fornece operações de consulta aos objetos. Este serviço é um superconjunto do *Structured Query Language* (SQL).
10. O serviço de licença fornece operações de medição do uso dos componentes, a fim de garantir uma compensação pela utilização destes. O serviço suporta qualquer modelo de controle de uso, em qualquer ponto do ciclo de vida do componente.
11. O serviço de propriedade fornece operações que permitem a associação de propriedades a qualquer componente.
12. O serviço de tempo oferece *interfaces* para a sincronização em um ambiente de objetos distribuídos. Ele também fornece operações para a definição e gerenciamento de eventos ativados por tempo.
13. O serviço de segurança fornece um conjunto completo de operações para a segurança dos objetos distribuídos. Suporta, entre outras operações, a autenticação e listas para controle de acesso.
14. O serviço de *trader* fornece um esquema de “páginas amarelas”, ou seja, permite aos objetos anunciarem seus serviços.
15. O serviço de coleção fornece *interfaces* CORBA para genericamente criar e manipular as coleções de objetos mais comuns.
16. O serviço de iniciação habilita os pedidos de iniciação, quando um ORB é invocado.

Figura 12 - Serviços de Objetos da Arquitetura CORBA

4.3.5 O Adaptador de Objetos da Arquitetura CORBA

Um adaptador de objetos registra as classes das aplicações, instancia novos objetos, associa-os com um ID único, anuncia sua existência, invoca seus métodos quando os clientes o solicitam, e gerencia os pedidos correntes para seus serviços. Os adaptadores mais sofisticados também proverão o gerenciamento de transação, balanceamento de carga e segurança de granularidade fina.

Um adaptador de objeto define se a ativação de um objeto significa criar um novo processo, ou reusar um *thread* ou processo existente. Os servidores podem suportar uma variedade de adaptadores de objeto para satisfazer diferentes tipos de pedidos. Por exemplo, um objeto base de dados pode querer explicitamente registrar todos os seus objetos de granularidade fina, sem emitir chamadas individuais para o adaptador de objetos. Neste caso, não faz sentido para um adaptador manter o estado por objeto, então a base de dados pode prover um adaptador de objeto que faça a *interface* com o núcleo do ORB e ao mesmo tempo encontre seus próprios requisitos especiais.

A OMG prefere não ver uma proliferação de adaptadores de objetos. Então a arquitetura CORBA especifica um *Basic Object Adapter* (BOA) que pode ser usado em muitos casos.

O adaptador de objetos, conforme mostra a Figura 13:

1. Registra classes de servidores com o repositório de implementação. O repositório de implementação é um armazenamento persistente gerenciado pelo adaptador de objeto.
2. Instancia novos objetos em tempo de execução. O número de instâncias criadas é função da demanda de serviços. O adaptador é responsável pelo balanceamento do suprimento de objetos com as demandas dos clientes.
3. Gera e gerencia referências a objetos. O adaptador de objetos associa referências (IDs únicos) aos novos objetos criados por ele. É responsável pelo mapeamento entre representações específicas de implementação e específicas de ORB, das referências de objetos.

4. Difunde a presença dos servidores de objeto. O adaptador de objetos pode difundir os serviços providos através do ORB, ou pode responder ao diretório de pedidos provenientes do núcleo do ORB (*core ORB*). Seu trabalho é divulgar ao mundo exterior os serviços que gerencia.

Figura 13 - A Estrutura de um Típico Adaptador de Objetos

5. Manipula as chamadas provenientes dos clientes. O adaptador de objetos interage com o topo do nível de mensagens da pilha de comunicação do ORB, retirando um pedido, e dirigindo-o para a *interface* do esqueleto. O esqueleto interpreta os parâmetros que chegam e apresenta-os de forma aceitável para a invocação do método do objeto.

6. Roteia as chamadas para cima até o método apropriado. O adaptador de objetos está implicitamente envolvido com a invocação dos métodos descritos nos *stubs* do esqueleto. Pode estar envolvido na ativação da implementação e pode autenticar os pedidos que chegam.

4.4 Implementações CORBA Estudadas: ILU e ORBIX

4.4.1 O *Inter-Language Unification* (ILU)

O *Inter-Language Unification* (ILU) pode ser caracterizado como um sistema de *interfaces* de objetos multi-linguagem. As *interfaces* de objetos fornecidas pelo ILU permitem que as diferenças entre implementações desenvolvidas, em diferentes linguagens e diferentes sistemas operacionais, sejam escondidas. Várias são as utilidades do ILU, podendo ser utilizado, entre outras coisas, na construção de sistemas distribuídos e bibliotecas multi-linguais orientadas a objetos (Janssen; Spreitzer (1996)).

Conceitualmente, uma *interface* ILU para um determinado módulo pode ser entendida como sendo um conjunto de tipos objetos, constantes, e exceções que qualquer outro módulo pode usar para se comunicar com este. O ILU permite diversos tipos de relacionamentos de ligação entre módulos, tais como:

- os módulos podem ser partes de uma única instância de um programa, todos escritos na mesma linguagem;
- os módulos podem ser partes escritas em diferentes linguagens, compartilhando o suporte de tempo de execução (*run time*) na mesma imagem de memória;
- os módulos podem ser partes processando em diferentes instâncias de programas em diferentes máquinas.

4.4.1.1 Funcionamento do ILU

Para o ILU, as *interfaces* devem ser escritas em alguma linguagem de especificação neutra, os tipos e exceções devem ser especificados e a funcionalidade é exportada através da definição de métodos nos tipos objetos. As *interfaces* são utilizadas pelas ferramentas fornecidas pelo ILU para a geração de *stubs* para as particulares linguagens de programação; estes *stubs* podem ser ligados, chamar ou serem chamados por *stubs* gerados pela mesma descrição de *interface* para uma diferente linguagem de programação. O código do *stub* é, então, ligado ao código da aplicação, ao código específico de uma linguagem de programação contendo

qualquer suporte ILU necessário para aquela linguagem de programação, e a biblioteca do núcleo do ILU, a qual é escrita em ANSI C.

Diversos módulos podem ser ligados juntos, para uso em separado. Os *stubs* ILU são gerados de forma a permitir que as aplicações que ligam um cliente a um servidor, escritos em uma mesma linguagem de programação, não sofram uma sobrecarga de chamada.

4.4.1.2 Objetos ILU

O ILU é orientado a objetos, sendo que toda sua funcionalidade é exportada de um módulo como métodos, que podem ser invocados em uma instância de algum tipo objeto, como se fossem simples procedimentos. A instância de um objeto fornece o contexto dentro do qual os métodos são executados. O sistema de tipos de objetos fornece a “herança entre objetos” para auxiliar na estruturação de *interfaces*.

Com relação a uma instância particular de um objeto ILU, um módulo é chamado de servidor, se implementa os métodos daquele objeto, ou cliente, se ele chama, mas não implementa, os métodos daquele objeto. Um objeto ILU pode ser passado como parâmetro ou como resultado de uma chamada de um método, e também pode ser um parâmetro em uma exceção. Um objeto pode ser passado de um servidor para um cliente, de um cliente para um servidor ou entre dois clientes. Podem existir múltiplos objetos ILU de um mesmo tipo, em uma mesma máquina, dentro de uma mesma instância de programa.

Para um dado objeto ILU existirão, em geral, múltiplos objetos específicos de linguagem; cada qual é um objeto em uma das linguagens de programação usadas no sistema. O objeto que fornece a implementação verdadeira de um objeto ILU é chamado de *true object* sendo, portanto, parte do módulo servidor. Os métodos dos objetos verdadeiros são escritos pelo programador, não sendo gerados pelo ILU. Os outros objetos possíveis são conhecidos como *surrogate objects*. Os métodos destes objetos são atualmente *stubs* RPC, gerados pelo ILU, que chamam o objeto verdadeiro. Um *surrogate object* é usado por um módulo cliente quando o módulo servidor está em uma diferente instância ou usa diferentes representações de dados.

O modelo de objeto especificado pelo ILU fornece múltipla herança de *interface*, ou seja, um subtipo fornece todos os métodos descritos por seus supertipos, acrescidos possivelmente de outros métodos descritos diretamente na descrição do subtipo.

Para chamar os métodos de um objeto ILU, um cliente deve primeiro obter um objeto específico de linguagem daquele objeto ILU. Isto pode ser feito de duas formas:

- (1) o cliente pode chamar um objeto específico de linguagem de um diferente objeto ILU para retornar o objeto em questão; ou
- (2) certas facilidades padrão podem ser utilizadas para adquirir o objeto, dada informação sobre seu endereçamento ou sobre seu nome. A informação de endereçamento é chamada de *String Binding Handle* (SBH), e a biblioteca de *run time* do ILU inclui um procedimento para adquirir um objeto específico de linguagem dado um SBH para um objeto ILU.

Toda a criação de uma instância *surrogate* implica em comunicação com o módulo servidor, e a ligação (*binding*) da instância substituta (*surrogate object*) com a instância verdadeira (*true object*).

O processo de criação de uma instância pode ser feito via um serviço de nomes, como por exemplo, o *PARC Name-and-Maintenance-Server* (NMS), o qual permite aos servidores registrarem instâncias em uma base de rede. Um servidor registra um mapeamento de uma informação de nomeação para um SBH. Os *stubs* do lado do cliente para uma *interface* incluem um procedimento que toma a informação de nomeação, localiza a correspondente SBH, e chama a rotina de biblioteca para mapear o SBH no objeto específico de linguagem. De forma alternativa, um cliente pode diretamente efetuar estes passos, usando um procedimento da biblioteca de *run time* do ILU para adquirir um objeto específico de linguagem para o serviço de nomeação.

4.4.1.3 Representação Baseada em Cadeia de Caracteres

O ILU utiliza uma cadeia de caracteres para referenciar um objeto. Esta cadeia de caracteres (SBH) é utilizada pelo ILU no mapeamento das referências de objetos para RPCs.

4.4.1.4 Conexões

O ILU não possui a noção de “conexões”, ou seja, o módulo chamado não aponta de volta para o chamador, e não possui qualquer outra noção além de ter que retornar uma mensagem de resultado. Informações passadas na mensagem de pedido podem identificar o chamador, mas não necessariamente sua localização. Os protocolos que necessitam desta informação, devem passá-la explicitamente como um argumento para o método.

O ILU inclui uma facilidade de ligação/nomeação experimental. Ele permite a um módulo “publicar” (*publish*) um objeto, de maneira que um outro módulo pode importá-lo conhecendo apenas seu *Object ID*.

A *interface* consiste de três operações: *Publish*, *Withdraw* e *Lookup*. O *Publish* toma como argumento um objeto ILU. O *Publish* retorna uma cadeia de caracteres (*string*) que é necessária para invocar com sucesso a operação *Withdraw*. O *Withdraw* desfaz os efeitos do *Publish*, e toma dois argumentos: (1) o objeto em questão, e (2) a *string* retornada por *Publish*. O *Lookup* toma dois argumentos: um *Object ID* e um tipo para o objeto. Se o objeto com o ID está sendo correntemente publicado, e tem o dado tipo (entre outros), o *Lookup* retorna o objeto.

As implementações disponíveis na versão do ILU utilizada podem usar tanto um Serviço ILU, quanto um sistema de arquivos compartilhados, para armazenar informações sobre os objetos publicados. A escolha deve ser feita em tempo de configuração do sistema. Se a abordagem do arquivo compartilhado é usada, este diretório deve estar disponível com o mesmo nome em todas as máquinas que desejem interoperar. A forma com que os clientes interagem com a ligação é o mesmo. Para conhecer os objetos atualmente publicados pode-se, também, utilizar um programa simples conhecido como *ihusbls*.

4.4.1.5 Ligação Através de Arquivos Compartilhados

Com este tipo de ligação, um diretório padrão é compilado para a biblioteca ILU. Este diretório pode ser explicitamente especificado no tempo de configuração do sistema e pode ser alterada em *run time*, através do uso de uma variável de ambiente. O ILU cria um arquivo neste diretório para cada objeto publicado. O arquivo contém o SBH do objeto e uma *string* aleatória, a qual serve como uma “prova” que deve ser verificada quando da remoção do registro. Usando-se a abordagem do arquivo compartilhado, o estado da proteção do diretório deve ser tal, que programas que chamam o *Publish* possam remover e criar novos arquivos, e programas que chamam *Lookup*, devem ser capazes de ler os arquivos no diretório.

4.4.1.6 Ligação Através do Serviço do ILU

Neste tipo de ligação, a idéia é que um programa chamado *ilusb* está executando em alguma máquina, e exporta o serviço de ligação via um porto especificado. Todos os clientes possuem o conhecimento (gerado em tempo de compilação) de qual computador (e porto), o serviço de ligação está executando, e que devem contatar o serviço executando as chamadas *Publish*, *Withdraw* e *Lookup*.

4.4.1.7 Sinalização de Erros

O ILU utiliza a noção de “exceção” para sinalizar erros entre módulos. Uma exceção é um modo de se passar o controle para fora do fluxo normal de execução. É uma forma tipicamente usada na manipulação de erros. A rotina que detecta o erro sinaliza uma exceção, a qual chama algum mecanismo para manipulação do erro.

4.4.1.8 O Sistema de Comunicação do ILU

O ILU permite a comunicação entre módulos através do mecanismo de troca de mensagens conhecido como protocolo RPC. Estas mensagens podem ser transportadas de diversas maneiras diferentes. O ILU é extensível, pois protocolos RPCs e transportes podem ser adicionados, tanto em tempo de compilação quanto em tempo de execução.

O Servidor do Núcleo do ILU exporta objetos através de uma ou mais “pilhas de contato”. Cada pilha tem um protocolo RPC em seu topo, formando e interpretando mensagens, e um ou mais níveis de transporte abaixo da camada de protocolo, transformando ou comunicando mensagens de várias formas.

4.4.1.9 Protocolo Abstrato de Mensagens do ILU

O protocolo abstrato do ILU está no topo de cada nível de protocolo. Utiliza dois tipos de mensagens: um para comunicar parâmetros para um método verdadeiro, e outro para comunicar resultados e/ou exceções do método verdadeiro ao chamador substituto (*surrogate*). Os parâmetros e valores são codificados de acordo com o simples valor abstrato do formato de representação externa. Este protocolo abstrato identifica qual a informação deve ser passada entre módulos, sem especificar seu exato mapeamento para os padrões de bits.

O primeiro tipo de mensagens é chamado de *request*. Cada *request* consiste de um código identificando o método sendo requisitado, um bloco de autenticação identificando o chamador, e uma lista de parâmetros de entrada para o método sendo chamado. O método é identificado passando-se um valor ordinal (ou seja, o índice do método em uma lista de métodos, iniciando com “1”). O bloco de autenticação do chamador varia de acordo com o protocolo de autenticação utilizado.

O segundo e último tipo de mensagem, chamada de *result*, é usada para levar os valores de retorno, ou de exceção, do método verdadeiro de volta ao chamador. Consiste de um valor booleano, para indicar se a chamada teve sucesso (*true*), ou não (*false*). Se houve sucesso, o valor de retorno (se existir), segue na posição que foi especificado dentro dos parâmetros. Se uma exceção for sinalizada, um valor ordinal

(identificando a exceção na lista especificada na definição do método) retornará, seguido pelo valor do tipo da exceção, se foi especificado algum para a mesma.

Entre os principais protocolos de RPC que podem ser usados pelo ILU, tem-se: o *Xerox Courier Protocol*, e o *OMG Internet Inter-Orb Protocol* (IIOP) e o *Hyper Text Transfer Protocol* (HTTP).

Quanto ao transporte das mensagens, tem-se que sua pilha de transporte é formada por uma sequência de camadas de transporte. A camada mais baixa executa algum tipo de E/S de baixo nível; as outras são “filtros” ou “modificadores” dos serviços de transporte fornecido pela parte mais baixa da pilha. Isto é, todo o final de uma pilha de transporte implementa uma abstração chamada simplesmente de “um transporte” (ou em C, *ilu-Transport*); cada camada de transporte (exceto a mais baixa) implementa um transporte em termos de um outro transporte (o qual é implementado pelo resto da pilha).

Uma pilha de transporte é especificada através de uma sequência de *strings*, cada uma especificando uma camada de transporte. Alguns transportes transferem mensagens delimitadas (onde cada mensagem é uma sequência de bytes), outros simplesmente uma sequência de bytes não delimitada. Alguns protocolos requerem um transporte do primeiro tipo, outros do segundo tipo.

Alguns transportes são confiáveis, outros não. Transportes não confiáveis são desaprovados no ILU, mas incluídos para permitir a interoperabilidade com os *softwares* existentes que usam somente protocolos não confiáveis. Com estes transportes, as mensagens podem ser entregues mais de uma vez. A implementação ILU do *User Datagram Protocol* (UDP), no lado do servidor, filtra múltiplos recebimentos da mesma mensagem. Métodos assíncronos podem não ser chamados sobre este tipo de mecanismo de transporte, bem como a entrega confiável do pacote de requisição (*request packet*) pode não ser reconhecido pelo lado do cliente. Métodos não-assíncronos usam mensagem de resposta como uma confirmação de que o pedido foi recebido.

4.4.1.10 Principais Mecanismos de Transporte Utilizados pelo ILU

Os principais mecanismos de transporte que podem ser usados pelo ILU são: o TCP, o UDP, o SunRPC *Record Marking* e o *In-Memory*. Estes mecanismos são apresentados, resumidamente, a seguir.

- **TCP - *Transmission Control Protocol*:** um nível de transporte TCP é confiável, não delimitado, e está acima do nível mais baixo de protocolo.
- **UDP - *User Datagram Protocol*:** um transporte do tipo UDP é não confiável, delimitado e está no nível mais baixo do transporte. Este tipo de transporte suporta apenas mensagens não maiores do que o tamanho máximo de um pacote UDP, sendo que tentativas de transmissão de pacotes de tamanho superior ao máximo causam um erro de E/S para o emissor.
- **SunRPC *Record Marking*:** um transporte do tipo SunRPC *Record Marking* é confiável e delimitado, e é construído no topo de algum outro transporte confiável e não delimitado.
- ***In-Memory*:** os níveis de transporte do tipo *In-Memory* não são normalmente usados ou mesmo conhecidos pelos usuários ou aplicações; são automaticamente criados e usados pelo ILU para chamadas entre representações, dentro de um espaço de endereçamento. Um transporte do tipo *In-Memory* é confiável e delimitado.

4.4.1.11 Servidores de Núcleo

Os objetos ILU são gerenciados por Servidores de Núcleo (*Kernel Servers*). Podem existir múltiplos servidores em um espaço de endereçamento. Cada servidor possui seu próprio conjunto de instâncias de objetos ILU, e gerencia certa funcionalidade para estas instâncias. Um item de funcionalidade gerenciada pelo Servidor de Núcleo é a comunicação com servidores de núcleo de outros espaços de endereçamento. Esta comunicação inter-servidor-de-núcleo é sempre executada entre um servidor de núcleo “verdadeiro” (*True Kernel Server*), ou TKS, e sua contraparte em outro espaço de endereçamento, o servidor de núcleo “substituto” (*Surrogate Kernel Server*), ou SKS. Instâncias em um TKS são aquelas que contém

implementações para as operações. Um SKS gerencia instâncias substitutas ou *proxy* para algum subconjunto das instâncias gerenciadas por seu correspondente TKS. Poderão existir diversos SKSs correspondendo-se com um único TKS, mas haverá apenas um SKS para cada TKS em um mesmo espaço de endereçamento.

4.4.2 A Arquitetura ORBIX

A IONA Technologies foi criada em março de 1991 como resultado da experiência obtida de vários projetos de pesquisa durante o período de 1985 a 1991. Estes projetos foram centrados em questões como: a programação distribuída em C++, sistemas operacionais distribuídos orientados a objetos, e persistência transparente de objetos. A IONA utilizou as lições aprendidas destes projetos para construir uma elegante implementação do CORBA.

O resultado, o ORBIX, foi o primeiro ORB a suportar tanto o padrão CORBA quanto a linguagem C++ (Iona (1997a) e Iona (1997b)).

Os três objetivos principais do ORBIX são os seguintes:

1. Facilidade de utilização: obtida através da adoção de convenções de programação que permitem a escolha do ambiente e da linguagem de programação.
2. Flexibilidade: o ORBIX possui uma arquitetura aberta a qual permite aos programadores a integração de uma ampla variedade de outras tecnologias usadas na construção de sistemas reais.
3. Alto Desempenho: As versões do ORBIX podem executar em sistemas operacionais de tempo real.

4.4.2.1 A Implementação da IONA do Padrão CORBA

O ORBIX é implementado como um par de bibliotecas - uma para aplicações clientes, e outra para servidores - e um *daemon* de ativação (*orbixd*). O *orbixd* precisa estar presente apenas nos nós que atuam como servidores CORBA, e sua função é (re)disparar processos servidores dinamicamente, quando solicitado. Um cliente não distribuído e aplicações servidoras em um mesmo espaço de

endereçamento, podem ser construídos utilizando-se apenas a biblioteca para o servidor. Neste caso, as referências de um objeto para o outro são diretas (por exemplo, são ponteiros diretos em C++ e as operações de invocações são simples funções virtuais em C++).

Quanto a localização do ORB ORBIX, devido a implementação de sua biblioteca, o ORB ORBIX é onipresente: não existe como um componente distinto identificável. Não existe um componente central através do qual todos os pedidos dos objetos tenham que passar: ao invés disso, os pedidos dos objetos passam diretamente do código do cliente para a implementação do objeto invocado.

O papel do orbixd é conectar clientes e servidores pela primeira vez. O orbixd usa uma base de dados simples, o Repositório de Implementação, para obter a informação de ativação de seus servidores; para cada servidor a informação inclui: o modo de ativação CORBA apropriado, o nome da imagem executável associada e qualquer parâmetro de linha de comando. Um componente importante do ORBIX é sua tecnologia de compilação, a qual traduz o CORBA IDL para o código da linguagem de programação (por exemplo, C++) que executa as chamadas remotas. O código gerado é suficientemente sofisticado para que os programadores não se preocupem com passos extras de programação.

O ORBIX é disponível em mais de 20 sistemas operacionais, incluindo: uma dúzia de versões do UNIX, *Windows For Workgroups* 3.11, *Windows* 95 e 98, *Windows* NT, VMS, MVS, *VxWorks* e Macintosh. O Suporte UNIX inclui: Solaris, HP/UX, AIX, OSF/1, SunOS, Solaris x86, UnixWare, Sinix e Ultrix. Ambos os lados do cliente e do servidor são suportados em todas as plataformas (incluindo o *Windows*), e é fornecida completa interoperabilidade entre todas elas.

A seguir serão descritas os diversos componentes da arquitetura ORBIX.

4.4.2.2 Componentes da Arquitetura ORBIX

A Figura 14 apresenta uma representação da hierarquia semântica da arquitetura ORBIX.

Figura 14 - Arquitetura ORBIX

4.4.2.2.1 A Máquina de Mensagens (*Messaging Engine*)

Este componente é responsável por fornecer um robusto sistema de entrega de pedidos sobre uma variedade de camadas de transporte. A Máquina é dividida em duas metades: um conjunto de características genéricas, tais como correlação de pedidos/respostas, *caching* de mensagens, gerenciamento de *heap* de mensagens e políticas de *threading*, são implementadas na própria Máquina. A Máquina também possui uma API a qual permite uma variedade de mecanismos de transporte. Esta API consiste em 4 abstrações maiores (expressas como classes abstratas C++ na base de código do ORBIX):

1. **Canal (*Channel*)**: Este componente fornece a abstração básica de uma conexão entre cliente e servidor. (Em termos do UNIX, o canal é estabelecido entre um par de espaços de endereçamento). Portanto múltiplas conversações entre um par de objetos podem ser multiplexadas em um único canal. Em princípio, diversas categorias diferentes de canais podem existir

em uma mesma aplicação. Na prática, o grau em que isto é possível é limitado pelo modelo adotado pelas camadas de transporte inferiores.

2. **Protocolo (*Protocol*):** Este componente abstrai a noção de protocolo de comunicação usado sobre um ou mais canais. A Máquina de Mensagens do ORBIX assume que todo protocolo concreto possui uma noção de pedido, resposta, e resposta excepcional. Protocolos concretos específicos podem ter, também, uma noção de estabelecimento de conexão, finalização natural de conexão, e prioridade de tráfego. Existem atualmente duas implementações concretas de protocolo:

- a classe *OrbixProtocol*, de propriedade da IONA, usado no ORBIX antes da padronização da OMG; e
- a classe *IIOP*, a qual implementa o protocolo IIOP da OMG.

3. **Codificador (*Coder*):** Esta classe abstrata provê o suporte de codificação/decodificação para todos os tipos IDL básicos (*long*, *short*, *double*, etc) para/dos *buffers* de mensagens. As funções membro também fornecem a codificação e a decodificação de vetores contínuos dos tipos IDL básicos.

Atualmente existem três classes derivadas do Codificador:

- **Codificador *OrbixProtocol*:** Esta é uma variação do protocolo usado para codificar o protocolo ORBIX proprietário (criado antes do IIOP).
- **Codificador *IIOP*:** Esta classe implementa a Representação Comum de Dados, como definido na especificação IIOP.
- **Codificador Nulo:** Esta é uma otimização do *OrbixProtocol*, pela qual se os dois lados de uma conexão suportam a mesma arquitetura, nenhuma codificação é executada.

4. **Manipulador de Eventos (*EventHandler*):** Este componente é responsável por implementar a política de sincronização entre processos (objetos clientes) e a execução de seus eventos de E/S em, no mínimo, um conjunto de canais abertos. Por exemplo, o componente impõe a política na qual o cliente irá se bloquear, até que uma resposta seja recebida de um objeto remoto. Existem diversas implementações possíveis desta classe de abstração, incluindo:

- *selectHandler*, uma simples implementação baseada em *threads*, a qual usa a chamada de *socket* padrão.
- *mtEventHandler*, a qual sincroniza os *threads* ao nível da aplicação com o *threads* internos ao ORBIX. Em particular, no ORBIX-MT um *thread* é criado para todo canal, permitindo, portanto, que a E/S continue em paralelo com cada conexão aberta com o servidor. O *mtEventHandler* usa semáforos para sincronizar o código da aplicação com os eventos provenientes da camada inferior.

4.4.2.2.2 A Máquina de Tempo de Execução (*Runtime Engine*)

O *Runtime* implementa as APIs CORBA clientes (por exemplo, a *Interface* de Invocação Dinâmica - DII) e as APIs servidoras (por exemplo, o Adaptador Básico de Objetos - BOA). Ele também implementa a funcionalidade requerida pelo código do *stub*, produzido pelo compilador IDL a partir das descrições fonte IDL.

As classes fundamentais no *Runtime* as classes *Object* e *Request*. As *interfaces* para ambas as classes estão disponíveis aos programadores de aplicações ORBIX via o módulo CORBA. Os programadores que escolherem utilizar a *interface* de invocação estática do CORBA, não precisam estar cientes destas duas classes em particular.

A classe *Request* implementa a *interface Request*, definida na especificação CORBA. É uma parte da DII e, no ORBIX, também é usada pelo código *stub* gerado. O ORBIX, além da *interface* DII padrão, estende o DII como especificado pelo CORBA, através de uma *interface* baseada em *stream*. Quando apresentado com argumentos para uma invocação, a classe *Request* internamente checa se estes estão sendo entregues como parte do DII ou através de *stubs* estaticamente gerados. Os argumentos DII são passados em uma lista *Named-Value (NVList)* e sua codificação é adiada até que *Request::invoke* seja chamado. Os argumentos estáticos são codificados diretamente.

A classe *Object* implementa a *interface Object* definida na especificação CORBA. De fato, uma instância da classe *Object* possui a informação fundamental para a comunicação com um objeto remoto. Para cada *interface* IDL compilada

através do compilador IDL, existe uma classe correspondente gerada (por exemplo, C++). A classe *Object* é a classe básica de todas as classes IDL.

Além dessas duas classes fundamentais, as classes *TypeCode* e *Any* implementam suas correspondentes especificações CORBA. Apesar de *Any* e *TypeCode* receberem uma curta descrição nas especificações CORBA, suas implementações são complexas e compreendem a maior parte do código fonte de runtime. Um valor de *Any* é fundamentalmente mapeado - em C e C++, para um *void** - que é, essencialmente, qualquer valor, incluindo complexas estruturas de dados. De maneira a identificar um tipo específico de um valor que um *Any* possui em tempo de execução, cada valor *Any* é também acompanhado com o tipo da informação. O tipo da informação, por sua vez, é interpretado pelo *TypeCode* (e pode ser gerado pelo compilador IDL).

À parte destas quatro classes, e outras, o *Runtime* inclui classes específicas da implementação ORBIX. Durante a execução, o *Runtime* constrói um *proxy*, ou *surrogate*, para cada objeto remoto usado pelo processo local. Cada *proxy* é uma instância de uma classe IDL. Se a classe IDL não está disponível ao processo local - porque a *interface* IDL era desconhecida no momento em que o mesmo foi construído, e portanto não existia código *stub* IDL disponível - então o *proxy* é usado no lugar de uma instância da classe *Object*.

O *Runtime* mantém uma tabela com todos os *proxies*, e todas as implementações das *interfaces* IDL, a qual no código fonte do ORBIX é chamada de Tabela de Objetos (*Object Table* - OT). Existe uma tabela para cada contexto de processo (por exemplo, por processo UNIX).

4.4.2.2.3 A Interface de Invocação Dinâmica (DII)

Como já mencionado, o IDL é usado para descrever *interfaces*, e o compilador IDL é usado para gerar o suporte necessário para permitir aos clientes invocarem objetos remotos. Especificamente, o compilador IDL automaticamente constrói o código apropriado para gerenciar os *stubs* (ou *proxies*), para disparar os pedidos que chegam a um servidor, e para gerenciar os serviços ORBIX de mais baixo nível.

Usando esta abordagem, as *interfaces* IDL que um programa cliente pode usar são determinadas em tempo de compilação. Infelizmente, este procedimento é muito limitante para um pequeno, mas importante, conjunto de aplicações. Estes programas e ferramentas requerem o uso de uma gama indeterminada de *interfaces*: *interfaces* que talvez não tenham sido concebidas no momento em que as aplicações foram desenvolvidas. Como exemplos, tem-se: *browsers*, *gateways*, ferramentas de suporte de gerenciamento e depuradores distribuídos.

O ORBIX suporta, portanto, uma *Interface* de Invocação Dinâmica (*Dynamic Invocation Interface* - DII), a qual permite a uma aplicação emitir pedidos para qualquer *interface*, mesmo que a *interface* não seja conhecida em tempo de compilação.

Note que a herança de *interfaces*, como suportado em IDL, fornece alguma ajuda no acesso às novas *interfaces*. Uma ferramenta pode ser escrita para utilizar uma *interface* IDL específica, mas de fato pode usar objetos que suportam qualquer *interface* derivada daquela *interface* base. Algumas vezes, porém, a herança utilizada desta forma mostra-se insuficiente: em particular, porque a ferramenta ficaria restrita à utilização das operações e atributos da *interface* base, e não àquelas adicionadas pelas *interfaces* derivadas.

Para superar este problema geral, a DII permite que as invocações sejam construídas através da especificação, em tempo de execução, da referência ao objeto alvo, dos parâmetros, e nomes das operações/atributos a serem passados. Tais chamadas são denominadas “dinâmicas” porque as *interfaces* IDL usadas pelo programa não têm que ser “estaticamente” determinadas no momento em que o programa é projetado e implementado. Em contraste, o uso dos *stubs* IDL é conhecido como *Interface* de Invocação Estática (*Static Invocation Interface* - SII).

4.4.2.2.4 *Dynamic Skeleton Interface* (DSI)

A *Dynamic Skeleton Interface* (DSI) é o equivalente ao DII, porém atua no lado do servidor. Ela permite ao servidor receber uma invocação de uma operação ou atributo para qualquer objeto, proveniente de uma *interface* IDL desconhecida em

tempo de compilação. O servidor não precisa ser ligado ao código do *skeleton* para aceitar as invocações feitas através da *interface*.

Ao invés disso, um servidor pode definir uma função que será informada sobre a chegada de uma invocação de operação ou atributo: aquela função pode identificar a identidade do objeto que está sendo invocado; o nome da operação e os tipos e valores de cada argumento devem ser fornecidos pelo usuário. Ela pode, então, levar a cabo a tarefa que está sendo requisitada pelo cliente, construir e retornar o resultado.

Desde a introdução do DSI, a maneira normal usada para se escrever um servidor ficou conhecida como *Static Skeleton Interface* (SSI).

O cliente não é capaz de saber se o servidor foi implementado usando-se o DSI; ele simplesmente faz chamadas IDL (o servidor não precisa considerar se o cliente usa o DII ou o SII). Para processar as invocações de atributos ou operações que chegam usando o DSI, um servidor deve fazer uma chamada ao ORB para indicar que deseja usar o DSI para uma *interface* IDL especificada. O mesmo servidor pode usar o SSI para manipular as invocações de atributos e operações de outras *interfaces*: mas não pode usar o DSI e o SSI na mesma *interface*.

Uma utilização muito comum para o DSI é que permite aos programadores a escrita de *gateways*. Usando o DSI, um *gateway* pode aceitar invocações de operações e atributos de qualquer conjunto de *interfaces* e passá-las a outro sistema.

4.4.2.2.5 Filtragem

O ORBIX não implementa diretamente várias funções algumas vezes requeridas pelos sistemas distribuídos. Porém estas funções podem ser adicionadas pelo usuário. Por isso um ORBIX regular corresponde apenas a um ORB.

O *runtime* ORBIX permite aos programadores ORBIX desenvolverem o código de filtragem para clientes e servidores. Os filtros são instâncias das classes de filtros, as quais são derivadas da classe abstrata *Filter*, fornecida pelo ORBIX. Os filtros são colocados em uma lista ligada: isto é, um número arbitrário de filtros podem ser instalados.

Fundamentalmente, os filtros são aplicados quando um pedido ou resposta é transmitida a partir de um contexto de processo, e quando o pedido (*request*) ou a resposta (*reply*) é recebido. A ação padrão (herdada da classe *Filter*), em cada caso, é simplesmente passar o evento ao próximo filtro da cadeia. Tendo processado um evento, um filtro pode escolher não passá-lo aos demais filtros da cadeia.

Como descrito acima, a filtragem é um mecanismo ao nível do processo, sendo aplicada a todos os pedidos e respostas que saem e chegam ao contexto do processo. Porém, o ORBIX oferece uma segunda forma de filtragem, conhecida como “filtragem por objeto”. Nesta abordagem, uma cadeia de filtros pode ser acoplada a uma instância de um objeto específico (dentro de um servidor). A cadeia pode operar independentemente de outros objetos servidores. A cadeia de filtro por objeto é aplicada após a cadeia de filtros por processo, no momento da chegada de um pedido, e antes da resposta ao pedido ser formada.

4.4.2.2.6 Carregadores

O ORBIX não possui qualquer suporte direto para a manipulação de objetos persistentes - isto é, objetos cujos estados possam ser salvos e restaurados a partir de um armazenamento não-volátil.

A união de um armazenamento persistente é claramente um requisito importante para muitas aplicações. O suporte fundamental para isto é a classe abstrata *LoaderClass* do *runtime* ORBIX. Os carregadores (*loaders*) são instâncias de *LoaderClass*, e são dispostos em uma lista ligada. Sempre que um novo objeto (descrito em IDL) é construído e registrado no ORBIX, os carregadores são notificados. Um programador de um servidor pode escolher nomear um novo objeto (usando uma cadeia de caracteres), e este nome também é passado aos carregadores, juntamente com a identidade do objeto na *interface* IDL. O nome dado ao objeto é chamado seu nome marca. Os carregadores devem ser codificados de forma a existir um carregador responsável por cada objeto. Tipicamente existe um único carregador, ou um carregador para um conjunto de *interfaces* IDL particular. O carregador responsável pelo novo objeto pode escolher adotar o nome proposto para o mesmo, ou gerar um novo nome. Um nome gerado pode ser, por exemplo, uma

chave relacional que será usada, mais tarde, como base para o armazenamento do objeto.

Alternativamente, um carregador específico pode ser nomeado quando um objeto é registrado no ORBIX. Uma classe pode ser escrita de forma que todos os seus construtores garantam que o mesmo carregador será usado para todas as suas instâncias.

Os carregadores são notificados quando o processo servidor termina, e podem escolher armazenar o estado dos objetos pelos quais são responsáveis. Um carregador também pode escolher salvar o estado de um objeto antes do processo terminar.

Quando um pedido é recebido por um servidor, os carregadores são notificados se o *runtime* ORBIX não for capaz de localizar o objeto alvo - isto é, o alvo ainda não foi registrado na Tabela de Objetos (*Object Table* - OT). O nome do objeto alvo também é passado aos carregadores, para que o carregador responsável possa ser identificado, e para que possa tentar restaurar o estado do objeto do armazenamento persistente. Se o objeto alvo é recuperado com sucesso, o OT é atualizado e o pedido é reassumido - a “falta de objeto” foi manipulada com sucesso. Se o objeto alvo não puder ser recuperado, ou nenhum carregador reconhecer o nome do objeto, então uma exceção retornará ao cliente.

4.4.2.2.7 O Serviço de Nomeação e Localização de Objetos

No ORBIX, um objeto é nomeado através da concatenação do nome do *host* com o do nó em que foi criado, e o nome que o objeto tem dentro do servidor - seu nome marca.

O nome do servidor é, por padrão, o mesmo da *interface* IDL que o implementa. Por exemplo, se uma implementação de uma *interface* de servidor for chamada “banco”, então o servidor, por padrão, também se chamará “banco”.

Um nome de servidor, no entanto, pode ser escolhido de forma independente de qualquer *interface* IDL regular. Por exemplo, se for escolhido *financialRepository* como nome do servidor de banco, então não existirá *interface*

IDL com este nome na aplicação. Os nomes de servidores podem ser estruturados hierarquicamente, de forma similar aos nomes de arquivos UNIX.

O nome escolhido para um servidor é importante porque é registrado no Repositório de Implementação, e usado pelo `orbixd` para identificar o arquivo executável que será usado para ativar o servidor. O mesmo arquivo executável pode ser registrado sob diversos nomes, isto é, diferentes servidores podem usar a mesma imagem executável.

Quando um programa cliente deseja usar um serviço particular nomeado em tempo de execução, o ORBIX deve ser instruído a ligar o cliente ao servidor mais adequado. Uma forma de obter isto é fazer com que o cliente forneça ao ORBIX uma cadeia de caracteres, a qual representa uma referência de objeto ORBIX completa: este mecanismo é prescrito pela especificação CORBA. A fonte de cada cadeia de caracteres não é prescrita: pode ser encontrada em um arquivo, ou ainda, transcrito de uma mensagem eletrônica (*email*) ou de fax.

Alternativamente, no ORBIX, um cliente se ligar a um nome de servidor específico em um *host* particular. O nome do servidor deve ser um dos nomes registrados no Repositório de Implementação daquele *host* - por exemplo, “banco” ou *financialRepository*, como descrito anteriormente. O cliente pode ainda tentar ligar-se com um objeto específico daquele servidor, utilizando apenas o nome marca do objeto. Se o cliente não especificar um nome marca, então o ORBIX ligará o cliente a qualquer objeto daquele servidor que forneça uma *interface* compatível com a esperada.

A forma mais geral de ligação do cliente é quando o cliente identifica um serviço, mas não especifica um *host* que possa prover o serviço. O serviço é nomeado pelo nome do servidor: zero ou mais *hosts* podem reconhecer o nome daquele servidor, baseado nas informações de seus respectivos Repositórios de Implementação. Neste caso, o ORBIX deve vasculhar na rede, à procura dos *hosts* adequados.

Cada procura é gerenciada pelo mecanismo de localização, implementado no *runtime* ORBIX, através da classe abstrata *locatorClass*. Esta classe é chamada com um nome de serviço (por exemplo, uma cadeia de caracteres) e é esperado que retorne uma lista dos *hosts* nos quais o serviço está presente. O *locatorClass* é normalmente usado de forma transparente ao código da aplicação.

Se o serviço de localização identifica que diversos *hosts* podem prover o serviço alvo, o ORBIX seleciona aleatoriamente qualquer um destes *hosts*. A seleção aleatória ajuda a espalhar a carga do servidor, quando múltiplos clientes estão usando o mesmo serviço.

4.4.2.2.8 O Compilador IDL e o Repositório de *Interface*

O compilador IDL é estruturado internamente para manter uma clara separação entre a análise e *parsing front-end* e a interpretação *back-end* dos arquivos fonte IDL analisados (Figura 15). Existem, atualmente, cinco *back-ends* disponíveis: os geradores de código *stub* para C++, Java e Ada; o gerador das classes cápsulas do ActiveX, para suportar a interoperabilidade COM-CORBA; e o gerador de informação para o Repositório de *Interface* (RI).

Figura 15 - A Arquitetura do Compilador IDL

No ORBIX, o compilador IDL é carregado como um executável completo para a geração de *stub*. A análise feita pelo compilador sobre os arquivos fonte IDL pode prover, direta ou indiretamente, a informação requerida pelo RI.

O compilador IDL foi integrado ao RI usando-se o mecanismo de manipulação de falta de objeto, descrito anteriormente. Isto é, o RI poderá ser

construído como um servidor de objetos ORBIX normal. Ele poderá utilizar o manipulador de falta de objeto para detectar invocações sobre objetos ainda desconhecidos por ele. Para resolver estas faltas de objetos, ele poderá usar o compilador IDL em tempo de execução para analisar os arquivos fonte IDL apropriados e gerar os objetos usados internamente pelo RI.

Como resultado, o RI não pode usar um armazenamento persistente para suas informações: os arquivos de código fonte IDL representam, por consequência, o armazenamento persistente para o RI. Um benefício imediato está no uso de utilitários padrão - no UNIX, tem-se, por exemplo, o emacs - para atualizar e prover o controle de versão na informação do RI.

4.4.3 Uma Comparação entre a Especificação CORBA e as Implementações ILU e ORBIX

Comparando-se o ILU com a especificação OMG CORBA, notou-se diversas similaridades e diferenças.

Entre as similaridades, tem-se que ambos possuem modelos de tipos e exceções parecidos e que o ILU, opcionalmente, permite a interpretação tanto da Linguagem de Definição de *Interfaces* (OMG IDL) do CORBA, quanto de sua própria linguagem de definição de *interfaces* (*Interface Specification Language* – ISL). Na verdade, o subconjunto da OMG IDL entendido pelo ILU é flexível o suficiente para permitir que módulos ILU desenvolvidos em OMG IDL sejam interoperáveis com qualquer outro sistema CORBA.

Entre as diferenças, tem-se que o ILU não fornece muitas das características requeridas por uma implementação CORBA. Principalmente não fornece implementações de: *Interface* de Invocação Dinâmica, Repositório de *Interfaces* e Repositório de Implementações. Não fornece, também, uma *interface* para um Adaptador Básico de Objeto (BOA), apesar de oferecer um adaptador de objetos que não possui as capacidades de conexão com os Repositórios de *Interfaces* e de Implementação. Vários outros conceitos definidos pelo CORBA não são ainda contemplados pelo ILU, sendo que os principais são:

- o uso do *#include*, pois o ILU utiliza o conceito mais limitado de *import*;

- a noção de se usar a *interface* IDL, tanto como um tipo de objeto, quanto para um espaço de nomeação;
- a noção de que todos os objetos BOA são persistentes (no ILU esta noção é deixada para a implementação do objeto);
- a noção de que as definições de tipo possam existir fora do escopo de qualquer módulo ou espaço de nomeação (no ILU, todas as definições ocorrem na mesma *interface*).

Por sua vez, o ORBIX da IONA, fornece um ORB bem mais completo, que implementa todos os elementos da arquitetura CORBA, incluindo a abordagem de *stub* (*Static Invocation Interface* - SII), a *Interface* de Invocação Dinâmica (*Dynamic Invocation Interface* - DII), o Repositório de Implementação (incluindo todos os modos de ativação especificados), o Repositório de *Interfaces* e a DSI (*Dynamic Skeleton Interface*).

Portanto, devido a completa aderência da arquitetura ORBIX à especificação CORBA, esta foi a implementação utilizada para a criação dos protótipos desenvolvidos nesta tese.

Capítulo 5 – A Utilização da Arquitetura CORBA como Suporte para Aplicações RV Distribuídas

Este capítulo descreve e discute, em detalhes, a metodologia elaborada para a utilização da arquitetura CORBA como suporte para as aplicações de realidade virtual distribuídas. Inicialmente são levantados os benefícios esperados com a utilização da arquitetura CORBA e, a seguir, é mostrada em detalhes, a abordagem desenvolvida.

5.1 Considerações Iniciais

A demanda por sistemas de computação distribuída de alto desempenho, no qual os AVDs se inserem, está em franco crescimento. A computação distribuída é uma tecnologia promissora pois oferece vários benefícios potenciais, entre eles: a conectividade através das redes; a disponibilidade e confiabilidade através da replicação; o aumento do desempenho através do processamento paralelo; a extensibilidade através da configuração e reconfiguração dinâmica; a escalabilidade e portabilidade através da modularidade; e a redução dos custos através do compartilhamento de recursos e sistemas abertos.

No entanto, apesar destas potenciais vantagens, o desenvolvimento do *software* de comunicação é dispendioso e sujeito a erros. Componentes e linguagens

orientadas a objeto são tecnologias amplamente mencionadas e utilizadas para reduzir o custo e melhorar a qualidade do *software*.

As técnicas de orientação aos objetos vêm sendo utilizadas há muitos anos, com sucesso, em domínios bem conhecidos. Porém, domínios mais complexos, como telecomunicações, processamento de transações *on-line* e, é claro, os AVDs, têm tradicionalmente carecido de componentes *middleware* padronizados. Como resultado, os desenvolvedores têm construído, validado e mantido sistemas de *software* praticamente a partir do nada. Nos dias de hoje, em plena competição global, este processo de desenvolvimento “caseiro” está se tornando consumidor de tempo e recursos.

Segundo Schmidt (1994), esta situação está produzindo uma “crise do *software* distribuído”, onde o *hardware* de computação e de redes está ficando cada vez mais reduzido, barato e rápido; enquanto que o *software* distribuído está ficando cada vez mais extenso, caro e lento para se manter e desenvolver.

Schmidt aponta como desafios na construção de *softwares* distribuídos, as complexidades *inerentes* e *casuais* associadas aos sistemas distribuídos.

A complexidade inerente resulta dos desafios fundamentais do domínio da aplicação distribuída, destacando: a minimização do impacto da latência da comunicação, a detecção e recuperação de falhas parciais de redes e computadores, o balanceamento de carga, o particionamento ótimo dos componentes de serviço e a consistente ordenação dos eventos distribuídos.

A complexidade casual advém das limitações das ferramentas e técnicas usadas no desenvolvimento de aplicações distribuídas. As limitações chave incluem: o suporte inadequado para depuração, a ausência de *interfaces* de chamada de sistema e componentes de bibliotecas que sejam reentrantes e portáteis, a ampla utilização da decomposição algorítmica e a contínua redescoberta e reinvenção dos conceitos e componentes básicos.

5.2 A Utilização da Arquitetura CORBA

Diante do exposto no item anterior, é objetivo desta tese propor a arquitetura CORBA como camada intermediária (*middleware*) de comunicação na construção de AVDs, bem como apresentar uma abordagem para a utilização da mesma.

5.2.1 A Motivação para o Uso da Arquitetura CORBA

No capítulo 4 descreveu-se a arquitetura CORBA e suas características principais de funcionamento. Cabe agora destacar os principais benefícios específicos esperados no uso da mesma. São eles:

- Simplificar a integração de aplicações distribuídas: a arquitetura CORBA fornece uma integração em um nível mais elevado do que os tradicionais fluxos de bytes não-tipados TCP.
- Fornecer uma base para a colaboração dos objetos distribuídos de alto nível: por exemplo, o OLE *Windows* e o OMG *Common Object Service Specification* (COSS).
- Obter as vantagens para a programação distribuída, similares as oferecidas pelas linguagens de orientação a objetos para a programação não-distribuída. Entre elas tem-se: o encapsulamento (a junção de dados e funções numa única entidade escondida), herança de *interface* (a transferência das características de uma *interface* para as suas derivadas), polimorfismo (uso de um único nome para definir formas distintas de *interfaces*) e manipulação de exceções (notificação aos objetos clientes das situações de exceção ocorridas no objeto servidor).

5.3 O Modelo Proposto

Como mostra a Figura 16, o modelo proposto consiste na utilização da arquitetura CORBA como infra-estrutura de comunicação e sincronização de nível intermediário ou *middleware* (camadas 5 e 6) para os objetos das camadas superiores. Além das operações básicas oferecidas pelo ORB *Core*, considerou-se,

também, a presença dos Serviços CORBA, em especial, o Serviço de Eventos (*Event Service*).

Utilizando os serviços do CORBA encontram-se as camadas superiores do AVD, que consistem no Gerente de Facilidades RV (camada 7) e os objetos clientes que compõem a aplicação RV distribuída (camada 8).

Cabe ressaltar que o Gerente de Facilidades RV é representado pelo servidor de objetos, responsável pelas operações comuns aos objetos RV clientes.

Nas camadas inferiores do sistema de comunicação e sincronização (camadas 1 a 4) do AVD considerou-se a existência de protocolos com capacidades de *multicast* confiável.

Figura 16 – Modelo Proposto

5.4 Metodologia de Desenvolvimento Utilizada

Neste item é apresentada a metodologia utilizada na confecção dos AVDs protótipos, cujas implementações são descritas no próximo capítulo.

5.4.1 Princípios Fundamentais de Desenvolvimento da Arquitetura CORBA

Sintetizando-se as características principais da arquitetura CORBA, tem-se:

- **A utilização do paradigma cliente/servidor:** a necessidade da separação dos objetos em objetos clientes (que requisitam serviços) e objetos servidores (que fornecem serviços).
- **A separação entre *interface* e implementação:** os objetos clientes dependem das *interfaces*, e não das implementações.
- **A transparência de localização:** o uso de um serviço é ortogonal a localização do serviço.
- **A transparência de acesso:** as operações são invocadas nos objetos.
- **A existência de *interfaces* com tipos definidos:** as referências aos objetos possuem tipos especificados na *interface*.
- **O suporte a herança múltipla das *interfaces*:** a herança estende, envolve e especializa o comportamento do objeto.

5.4.2 Descrição da Metodologia

A metodologia utilizada para a construção de AVDs com base na arquitetura CORBA é composta das seguintes etapas:

- A. Identificação dos requisitos das operações de alto nível da aplicação RV distribuída.
- B. Separação das operações que devem ser executadas nos objetos clientes, e as que devem ser oferecidas pelos objetos servidores (que compõem o Gerente de Facilidades RV – camada 7).

- C. Reunião das operações necessárias em *interfaces* bem definidas.
- D. Escrita das definições das *interfaces* através do uso do OMG-IDL (*Interface Definition Language*).
- E. Compilação das *interfaces* IDL.
- F. Implementação das *interfaces* utilizando os recursos oferecidos pela linguagem de programação escolhida.
- G. Implementação do corpo principal do Gerente de Facilidades RV.
- H. Registro do Gerente de Facilidades RV no banco de dados do CORBA, conhecido como Repositório de Implementações (*Implementation Repository*).
- I. Escrita do código da aplicação RV cliente.

A seguir estão as descrições destas etapas.

A) Identificação dos Requisitos das Operações de Alto Nível da Aplicação RV Distribuída

Esta fase é uma das mais importantes no desenvolvimento de qualquer AVD. Consiste em determinar as principais tarefas de alto nível que devem ser executadas pelo sistema.

Em um sistema RV distribuído, muitas são as tarefas a serem executadas. Como exemplo, pode-se citar:

- 1) A tarefas de iniciação do mundo virtual: carregamento das geometrias que compõem o mundo virtual em uma hierarquia lógica, posicionamento de pontos de vista (*viewpoints*), criação de janelas (se a aplicação for não imersiva), definição dos sensores usados (*mouse*, luvas, teclado, etc.), associação dos sensores a algum ponto de vista (de modo a permitir, por exemplo, que a movimentação do mouse implique na navegação do usuário no ambiente – *walk-through*), definição e posicionamento das fontes de iluminação (*lights*).
- 2) Leitura de sensores.
- 3) Chamada de funções de ação gerais, como por exemplo, verificação de entradas via teclado e mudanças de parâmetros de renderização (por exemplo, de texturizado para *wireframe*).

- 4) Atualização da situação dos objetos locais em decorrência de alguma entrada via sensor, por exemplo, a alteração da posição e orientação do(s) objeto(s), devido a um comando via *mouse*.
- 5) Execução de tarefas específicas associadas a um determinado objeto local (comportamento), por exemplo, se o objeto representa um planeta, pode ser interessante associar a ele uma função que o rotacione um certo número de graus, a cada quadro (*frame*) apresentado.
- 6) Ativação de animação: certos movimentos repetitivos podem ser pré-determinados e armazenados para serem ativados devido a determinado evento. Por exemplo, pode-se armazenar a sequência de posições de uma porta se abrindo e executá-la após um evento (apertar uma representação de um botão de elevador).
- 7) Renderização do mundo virtual: todos os elementos que compõem o mundo virtual e que podem ser visualizado através do ponto de vista selecionado são desenhados (em 3D) e apresentados no dispositivo de saída escolhido (por exemplo, um *display* ou visor de um HMD). A renderização de uma cena geralmente é executada uma vez por quadro apresentado.
- 8) A manutenção e armazenamento das informações gerais sobre os participantes (por exemplo, quais os usuários conectados no AVD).
- 9) O pedido de conexão de um novo usuário no AVD.
- 10) A comunicação da entrada de um novo usuário aos demais participantes remotos.
- 11) O envio de informações de iniciação a um novo participante (por exemplo, o posicionamento dos objetos dos usuários já conectados).
- 12) O envio de informações de interesse aos demais participantes, como por exemplo, mudança no posicionamento e orientação de um objeto manipulado por um dos participantes.
- 13) O recebimento de informações de interesse aos demais participantes, como por exemplo, mudança no posicionamento e orientação de um objeto manipulado por um dos participantes.

- 14) Ativação e verificação de algoritmos para minimizar a comunicação via rede, como por exemplo, o *dead-reckoning* (Harvey et al. (1991); McCarty et al. (1994)).
- 15) Gerenciamento de múltiplas visões do mesmo mundo virtual.
- 16) O pedido de desconexão de um usuário do AVD.
- 17) A comunicação da saída (desconexão) de um usuário aos demais participantes.

B) Separação das operações que devem ser executadas nos objetos clientes, e as que devem ser oferecidas pelos objetos servidores, levando em consideração os Serviços CORBA

A necessidade desta separação é decorrente do paradigma cliente/servidor imposta pela arquitetura CORBA.

Esta etapa também é muito importante e terá um grande impacto no desempenho do sistema. Segundo o exposto no Capítulo 2, item 2.2.3, muitos aspectos devem ser considerados, entre eles tem-se: quais as operações realizadas pelo(s) Serviço(s) CORBA escolhido(s); o modelo de comunicação; o modelo de visão; o modelo da base de dados; e o modelo de gerenciamento da computação.

Todos estes aspectos são muito amplos, porém devem ser fixados antes das etapas de implementação. Como exemplo, nas implementações realizadas nesta tese, e que são descritas no próximo capítulo, tomou-se as seguintes decisões para os aspectos levantados:

- ♦ **Serviços CORBA:** optou-se pelo Serviços de Eventos. Neste serviço, a herança é utilizada para construir um caso simples de negociação de eventos entre dois objetos, através de um canal de eventos, o qual desacopla as interações. Segundo este serviço, as aplicações podem ser classificadas como Produtoras de eventos (*Suppliers/Talkers*) ou Consumidoras de eventos (*Consumers/Listeners*). O objeto canal de eventos desacopla a comunicação entre Produtoras e Consumidoras. Uma Produtora pode também agir como Consumidora e vice-versa. Pode

também aceitar a conexão de uma Produtora para uma ou mais Consumidoras, como mostra a Figura 17.

Figura 17 – Um Canal de Eventos com Múltiplas Aplicações Produtoras e Consumidoras

- ♦ **Modelo de Comunicação:** ao se optar pelo Serviço de Eventos do CORBA, de certa maneira, já se limitou os modelos de comunicação possíveis. Sabendo-se que este serviço, em algumas implementações, pode agir em conjunto com um protocolo *multicast* confiável (como é o caso da implementação selecionada, o ORBIX da IONA – descrito no Capítulo 4), este foi um dos modelos escolhido. Na verdade, cabe ressaltar que a escolha do Serviço de Eventos deveu-se, em grande parte, a esta característica importante para o AVD em questão. Certas mensagens, no entanto, nem sempre precisam ser enviadas a todo o grupo de máquinas que participam do AVD, por isso, a comunicação *unicast* também é usada.
- ♦ **Modelo de Visão:** o modelo de visualização escolhido é o modelo assíncrono, onde os múltiplos usuários têm controle individual sobre “quando” e “o que” podem ver, concorrentemente, no ambiente virtual.
- ♦ **Modelo da Base de Dados:** o modelo escolhido é um modelo híbrido, onde a base de dados do mundo virtual é homogênea e totalmente replicada nas máquinas dos clientes (camada 8) e outra base de dados,

com informações globais do ambiente é mantida no Servidor (localizado na camada 7 – Gerente de Facilidades RV).

- ♦ **Modelo de Gerenciamento da Computação:** quanto a distribuição dos processos, o modelo selecionado é o da replicação total. Como já mencionado, neste modelo, somente a informação que muda o comportamento dos objetos espelhados é enviada, fazendo com que todos os cálculos sejam feitos localmente na máquina do cliente. Quanto ao comportamento dos objetos, nos protótipos implementados, optou-se por comportamentos que se enquadram nos níveis 0, 1 e 2, segundo a classificação de Roehl (1995).

À luz destas decisões de projeto, já é possível determinar quais tarefas levantadas no item (A) podem ser efetuadas pelos clientes e quais podem ser de responsabilidade do(s) servidor(es). A alocação usada nesta tese é mostrada na Tabela II.

C) Reunião das operações necessárias em *interfaces* bem definidas

As operações candidatas a pertencerem as *interfaces* são aquelas que envolvem a troca de mensagens, ou seja, as tarefas (9), (10), (11), (12), (13), (16) e (17), da Tabela II.

Considerando que:

- a aplicação cliente pode ser Produtora e Consumidora de eventos, e que o servidor pode ser um Produtor de eventos;
- certas operações são facilitadas pela comunicação *multicast*, e outras pela comunicação *unicast*.

Evidencia-se pelo menos a necessidade de duas *interfaces*: uma que contenha as operações *multicast* (e que usem o Serviço de Eventos) e outra que contenha as operações *unicast* (sem a necessidade do uso do Serviço de Eventos). Estas *interfaces* podem ser chamadas, por exemplo, InterfClient e InterfServer, respectivamente.

As tarefas (10), (12), (13) e (17), portanto devem pertencer a InterfClient. Já as tarefas (9), (11) e (16), pertencem a InterfServer.

Tabela II – Alocação das Tarefas ao Cliente e Servidor

	CLIENTE	SERVIDOR
Serviços CORBA (Serviço de Eventos)	<p>- Podem ser Produtores: (12) enviar informações de interesse aos demais participantes;</p> <p>- Podem ser Consumidores: (13) receber informações de interesse dos demais participantes.</p>	<p>- Pode ser Produtor: (10) comunicar a entrada (conexão) de um novo usuário aos demais participantes remotos; (17) comunicar a saída (desconexão) de um usuário aos demais participantes.</p>
Modelos de Comunicação (multicast e unicast)	<p>Multicast: (12) enviar informações de interesse aos demais participantes; (13) receber informações de interesse dos demais participantes.</p> <p>Unicast: (9) pedido de conexão de um novo usuário no AVD; (16) pedido de desconexão de um usuário do AVD.</p>	<p>Multicast: (10) comunicar a entrada (conexão) de um novo usuário aos demais participantes remotos; (17) comunicar a saída (desconexão) de um usuário aos demais participantes.</p> <p>Unicast: (11) enviar informações de inicialização a um novo participante.</p>
Modelo de Visão (Assíncrona)	(4) atualização da situação dos objetos locais em decorrência de alguma entrada via sensor; (7) renderização do mundo virtual.	Não tem.
Modelo da Base de Dados (BD Homogênea e totalmente replicada no Cliente; e BD não compartilhada no Servidor)	(1) tarefas de inicialização do mundo virtual; (2) leitura de sensores; (3) chamada de funções de ação gerais; (5) execução de tarefas específicas associadas a um determinado objeto local.	(8) manutenção e armazenamento das informações gerais sobre os participantes.
Modelo do Gerenciamento da Computação (Replicação Total)	(3) chamada de funções de ação gerais; (5) execução de tarefas específicas associadas a um determinado objeto local; (6) ativação de animação; (14) ativação e verificação de algoritmos para minimizar a comunicação via rede.	Não tem.

D) Escrita das definições das *interfaces* através do uso do OMG-IDL

Uma vez decidido o número de *interfaces* necessárias, e quais as operações (e os tipos envolvidos) que devem fazer parte destas, cabe traduzi-las para a linguagem IDL. A IDL é independente de implementação, ou seja, é uma linguagem declarativa, e não uma linguagem de programação. Ela impõe que as *interfaces* sejam definidas separadamente das implementações dos objetos. Esta independência é importante em sistemas heterogêneos (como pode ser o caso dos AVDs), onde nem todas as linguagens são suportadas ou disponíveis em todas as plataformas. Apesar disto, a IDL têm a aparência (mas não a semântica) da linguagem ANSI C++, principalmente no que diz respeito as classes e tipos suportados. Um exemplo é mostrado a seguir (Figuras 18 e 19):

```
// Arquivo Clientes.idl: contém definição de InterfClient
// Declaração de tipos e variáveis globais
...
// Declaração da Interface InterfClient
interface InterfClient{
    // operações
    oneway void NetAlive(in long Id, in string Obj); //tarefa
                                                    //(10)
    ...
};
```

Figura 18 – Visão Parcial do Arquivo Clientes.idl

```
// Arquivo Servidor.idl: declaração de InterfServer
// Declaração de tipos e variáveis globais
...
// Declaracao da Interface InterfServer
interface InterfServer{
    tableObj NetOpen(in long Id, in File Obj) //tarefa (9)
        raises (IdRejected);
    ...
};
```

Figura 19 – Visão Parcial do Arquivo Servidor.idl

E) Compilação das *interfaces* IDL

Conforme ilustra a Figura 20, os arquivos contendo as *interfaces* IDL devem ser compilados ou traduzidos, para a geração dos tipos das

linguagens de programação escolhidas, bem como os *stubs* do lado do cliente, e os *skeletons* do lado do servidor, conforme descrito no item 4.3. Um *stub* é um mecanismo que efetivamente cria e envia os pedidos no lado do cliente, enquanto um *skeleton* é um mecanismo que entrega os pedidos a implementação do objeto CORBA. Uma vez que são traduzidos diretamente das especificações IDL, *stubs* e *skeletons* são normalmente específicos para uma *interface*. O despacho de pedidos através de *stubs* e *skeletons* é chamado de invocação estática. Os *stubs* e *skeletons* IDL são construídos diretamente na aplicação cliente e na implementação do objeto. Portanto, ambos possuem um conhecimento antecipado das *interfaces* IDL dos objetos CORBA que estão sendo invocados.

Os mapeamentos de linguagem geralmente mapeiam a invocação de uma operação para o equivalente a uma chamada de função na linguagem de programação.

Figura 20 – Estrutura de compilação das *interfaces* IDL

F) Implementação das *interfaces*, utilizando os recursos oferecidos pela linguagem de programação escolhida

Como citado anteriormente, a linguagem IDL é declarativa, por isso, ela não fornece características como construções de controle, nem pode ser usada diretamente para implementar aplicações distribuídas. Ao invés disso, os “mapeamentos de linguagem” determinam como as características da IDL são mapeadas para as facilidades de uma dada linguagem de programação.

Cabe, portanto, ao desenvolvedor escrever o código correspondente a implementação das *interfaces*, respeitando este mapeamento. Várias são as linguagens para as quais este mapeamento já foi definido atualmente, entre elas tem-se: C, C++, Smalltalk, ADA e JAVA.

Naturalmente, este mapeamento é mais direto nas linguagens orientadas a objetos, como é o caso, por exemplo, da linguagem C++. No mapeamento de IDL para C++, as *interfaces* mapeiam para classes C++, com suas operações sendo mapeadas para funções membro daquelas classes. As referências aos objetos mapeiam para objetos que suportam o operador de função (->).

Como exemplo, tem-se que a *interface* InterfClient, relacionada ao cliente, possui como implementação os arquivos: clientes_i.h (arquivo header) e clientes_i.cpp (arquivo de implementação escrito em C++), como é mostrado nas Figuras 21 e 22, a seguir.

```
//Arquivo clientes_i.h: contém a classe que implementa a
//interface InterfClient

//Declaração dos tipos e variáveis globais
...
class InterfClient_i: public virtual InterfClientBOAImpl {
public:
    //construtor da classe
    InterfClient_i(const char* Name)
        :InterfClientBOAImpl(Name){}
    //protótipos das funções membro
    virtual void NetAlive (CORBA::Long Id, const char *Obj,
        CORBA::Environment&) ; //tarefa 9
    ...
};
```

Figura 21 – Visão Parcial do Arquivo Clientes_i.h

```

//Arquivo clientes_i.cpp: contém a implementação das
//                          operações da classe InterfClient_i,
//                          relativa a interface IDL
//                          InterfClient

//Declaração de tipos e variáveis globais
...
//implementação da função NetAlive(correspondente a tarefa 9)
void InterfClient_i:: NetAlive (CORBA::Long Id, const char
*Obj, CORBA::Environment&) {
    //comandos C++
    ...
}

```

Figura 22 – Visão Parcial do Arquivo Clientes_i.cpp

G) Implementação do corpo principal do Gerente de Facilidades RV

O Gerente de Facilidades RV contém o objeto servidor gerado. Como visto, o servidor age como Produtor, segundo a abordagem do Serviço de Eventos do CORBA.

Por isso, as tarefas típicas para o Gerente de Facilidades RV são:

- ◆ A criação de objetos *proxy* do tipo InterfServer_i. Um objeto *proxy* é um representante local de um objeto remoto. Do ponto de vista da utilização, pode-se interagir com um objeto *proxy*, como se fosse o próprio objeto remoto.
- ◆ A iniciação do Gerente do Serviço de Eventos do CORBA;
- ◆ O registro destes objetos como Produtores, perante o Gerente do Serviço de Eventos do CORBA, especificando o evento que produzem. Estes eventos serão levados pelo Serviço de Eventos (em conjunto com o ORB) para todos os objetos consumidores que tenham registrado interesse nestes (usando o protocolo *multicast* confiável);
- ◆ A Entrada no “*Loop* de Eventos” do Serviço de Eventos da arquitetura CORBA. Este *loop* (laço) é responsável por: manipular os pedidos para reenvio de mensagens; descartar mensagens antigas; enviar periodicamente as mensagens reportando eventos; e
- ◆ Finaliza a iniciação do servidor e avisa a arquitetura CORBA que está pronto para receber pedidos de operações em seus objetos. Este aviso é feito através da chamada a função `impl_is_ready()`, a qual não

retorna imediatamente: ela bloqueia o servidor até que um evento ocorra, manipula este evento, e então, re-bloqueia o servidor a espera de um novo evento.

H) Registro do Gerente de Facilidades RV no Repositório de Implementações

Todo servidor pertencente ao Gerente de Facilidades RV deve ser registrado no Repositório de Implementações da arquitetura CORBA. Este repositório registra o nome do servidor e o nome do arquivo executável correspondente. Através deste registro, é possível a arquitetura CORBA, disparar um servidor que não esteja em execução, quando um de seus objetos for usado. Se o servidor não for registrado no Repositório de Implementações, o servidor será desconhecido ao suporte de tempo de execução do CORBA, e portanto, deve ser executado manualmente (na chamada a função `impl_is_ready()`, deve-se especificar o nome do servidor, neste último caso).

I) Escrita do código da aplicação RV cliente

Segundo esta metodologia, uma grande parte das tarefas cabe a aplicação cliente. Isto é coerente com as decisões tomadas na fase (B), além de ser vantajoso do ponto de vista da comunicação, uma vez que uma grande desvantagem do modelo cliente/servidor é a tendência do servidor tornar-se um “gargalo” no sistema. Manter apenas as tarefas gerais no servidor (como por exemplo, o controle de conexão e desconexão), torna-o mais leve (do ponto de vista computacional) e menos solicitado, minimizando a latência da comunicação.

Uma estrutura típica para a aplicação cliente é:

- ◆ Realizar as tarefas (1) a (7), relativas diretamente a execução do mundo virtual. Para a implementação destas tarefas, pode-se fazer uso de *toolkits* para programação de aplicações RV;
- ◆ A realização das demais tarefas (9), (12), (13), (14) e (16), envolve a comunicação via CORBA, devendo o cliente:

- Localizar o objeto remoto de interesse e obter uma referência a ele.
Isto geralmente é feito pela chamada a função CORBA `_bind()`.
A função `_bind()`, retorna uma referência ao objeto *proxy*.
- Invocar as operações no objeto *proxy*.

A Figura 23 mostra uma visão das partes principais de uma aplicação cliente.

```
//Arquivo ClienteRV.cpp: contém o código fonte da aplicação
// cliente
...
//Declara uma instância para a classe InterfServer
InterfServer_var ServidorIO;
...
//Associa o proxy ao objeto servidor remoto verdadeiro
ServidorIO = InterfServer::_bind(":CidadeOT",host);
...
//Faz invocações ao objeto remoto, através do objeto proxy
ServidorIO->NetClose(netid);
...
```

Figura 23 – Visão Parcial do Arquivo ClienteRV.cpp.

Conforme foi descrito anteriormente, a aplicação cliente, neste exemplo, é tanto Produtora quanto Consumidora de eventos. Portanto, os mesmos passos quanto ao uso do Serviço de Eventos do CORBA, descritos no item (G) para o servidor, devem ser obedecidos pela aplicação cliente.

Portanto, realizando-se corretamente as etapas (A) a (I) descritas nesta metodologia, é possível obter-se um AVD suportado pela arquitetura CORBA.

No próximo capítulo são apresentados os protótipos implementados com o objetivo de validação da metodologia proposta.

Capítulo 6 - Implementações

Este capítulo apresenta as implementações dos protótipos baseados na metodologia descrita no capítulo anterior. Como mencionado anteriormente, esta tese concentra-se na tarefa de comprovar que a arquitetura CORBA é válida para fornecer o suporte intermediário (*middleware*) de comunicação e sincronização aos AVDs, bem como propor uma abordagem para a utilização da mesma.

Os protótipos foram implementados com a finalidade de testar e validar a metodologia apresentada. Contemplam aspectos de construção diferentes, visando a criação de uma base comparativa consistente com a proposta inicial. A exaustiva análise do desempenho destes protótipos está fora do escopo deste trabalho, porém são apresentados diversos resultados demonstrativos.

Ao todo, foram implementados 3 protótipos:

- ◆ **Protótipo 1:** neste protótipo foi construída a aplicação RV distribuída utilizando-se apenas os recursos da ferramenta WorldToolkit (WTK) (descrita no capítulo 3) e suas facilidades de comunicação. Não se utilizou a arquitetura CORBA, nem seus serviços.
- ◆ **Protótipo 2:** este protótipo foi construído para se verificar a viabilidade da integração da aplicação WTK e a plataforma CORBA, sem seu Serviço de Eventos. Utilizou-se, aqui, apenas o modelo cliente/servidor.
- ◆ **Protótipo 3:** este protótipo implementa completamente as tarefas discutidas no capítulo 5, integrando a aplicação RV distribuída WTK, com a arquitetura CORBA e seu Serviço de Eventos.

6.1 Recursos Utilizados na Implementação dos Protótipos

Na construção dos protótipos foram utilizados os recursos de *software* descritos a seguir.

6.1.1 O WorldToolkit

Ambientes virtuais são sistemas complexos e requerem formas de se integrar, de maneira padronizada, a descrição geométrica dos objetos com seus comportamentos programados. Isto geralmente requer a utilização de um *toolkit* de modelagem 3D. Estes *toolkits* são melhor desenvolvidos quando utilizam uma abordagem de orientação a objeto, onde cada tipo de objeto (polígono, fonte de luz, etc.) é modelada como uma classe. Instâncias dessa classe são arranjadas em um grafo de cena hierárquico. O instanciamento dos subgrafos permite ao modelo explorar similaridades na cena (por exemplo, a geometria para quatro rodas idênticas de um carro só precisa ser especificada uma vez). Cada grafo de cena é processado de forma transversal, e para cada nó visitado, o apropriado método da classe correspondente é chamado. A Figura 24 mostra um grafo de cena simplificado.

Figura 24 – Grafo de Cena Simplificado

Como a criação de um *toolkit* para modelagem e renderização é uma tarefa complexa e tediosa, e que está além do escopo deste trabalho, optou-se pela solução de utilizar-se um *software* já existente, o Sense8 WorldToolkit (Sense8 (1998)).

O WorldToolkit, descrito previamente no item 3.2.16, pode ser entendido como um ambiente de desenvolvimento de aplicações gráficas 3D de tempo real, de

alto desempenho. Possui uma biblioteca de mais de 1000 funções escritas em linguagem C que permitem ao programador desenvolver aplicações de realidade virtual de qualidade. As funções WTK são orientadas a objeto e estão organizadas em mais de 20 classes. Estas classes incluem: *Universe*, *Lights*, *Sensors*, *Paths*, *Motion Links*, e outras.

Para a modelagem propriamente dita das geometrias dos objetos envolvidos no mundo virtual escolhido, utilizou-se o WorldUp Modeler da Sense8 e o 3DStudio da Autodesk.

6.1.2 ORBIX e OrbixTalk

Como implementação da arquitetura CORBA utilizou-se o ORBIX da IONA (Iona (1997b)). O ORBIX, descrito em detalhes no item 4.4.2, foi o primeiro ORB a suportar tanto o padrão CORBA quanto a linguagem C++. Atualmente, o ORBIX fornece um ORB sofisticado que implementa todos os elementos do CORBA, incluindo a abordagem de *stub* (*Static Invocation Interface* - SII), a *Interface* de Invocação Dinâmica (*Dynamic Invocation Interface* - DII), o Repositório de Implementação (incluindo todos os modos de ativação especificados), o Repositório de *Interfaces* e a *Dynamic Skeleton Interface* (DSI).

Ainda neste trabalho, utilizou-se o OrbixTalk da IONA (Iona (1997a)), em complemento ao ORBIX. O OrbixTalk é uma implementação completa do Serviço de Eventos CORBA da OMG. Consiste em uma extensão da especificação CORBA que permite aos clientes se comunicarem com objetos da aplicação usando mensagens. O OrbixTalk possibilita que a informação compartilhada seja organizada em uma estrutura hierárquica de tópicos, cada qual identificado por um *OrbixTalk Topic Name*. Desse modo, uma aplicação pode determinar qual informação é de seu interesse e informar o OrbixTalk usando o *Topic Name*. As aplicações construídas usando o OrbixTalk podem, portanto, ser informadas dos eventos quando estes acontecem e agir de acordo com eles. O *Topic Name* é traduzido para um endereço IP *multicast* através do processo *OrbixTalk Directory Enquiries*.

No OrbixTalk, os *talkers* (ou produtores), geram mensagens destinadas aos *listeners* (ou consumidores). O *talker* não necessita conhecer a identidade dos

listeners, nem sua natureza ou quantidade. O mesmo acontece com os *listeners*, em relação aos *talkers*. Portanto, tem-se uma comunicação completamente desacoplada.

O OrbixTalk usa o *User Datagram Protocol* (UDP) baseado no IP *multicast* para compartilhar informações entre aplicações. Por si mesmo, este mecanismo não informa as aplicações quando uma mensagem é perdida ou chega fora de sequência. Para fornecer confiabilidade, o OrbixTalk usa o protocolo chamado *OrbixTalk Reliable Multicast* (OTRMP). O OTRMP garante que as mensagens enviadas de uma aplicação particular sejam confiavelmente entregues, na sequência correta, para todas as aplicações em um grupo *multicast*.

A Figura 25 sumariza a comunicação entre um *talker* e um *listener* via processo *daemon OrbixTalk Directory Enquiries* e o serviço de transporte *multicast* do OrbixTalk. O *Loop* de Eventos (citado no capítulo anterior) deve estar presente tanto nos *talkers* quanto nos *listeners*, e é responsável pela manipulação dos pedidos para reenvio de mensagens; descarte das mensagens antigas; e envio periódico das mensagens reportando eventos.

Figura 25 – Arquitetura OrbixTalk

6.1.3 Linguagem de Programação

A linguagem de programação escolhida para a implementação dos protótipos foi a Visual C++ versão 5.0, da Microsoft. Esta linguagem foi escolhida por ser orientada a objetos e, portanto, compatível tanto com a ferramenta de RV utilizada (WorldToolkit), quanto as arquiteturas ORBIX e OrbixTalk.

6.2 A Aplicação RV Distribuída: A Cidade Virtual

No capítulo 2 desta tese, foram citadas diversas áreas que podem ser beneficiadas pelo emprego das técnicas de realidade virtual. Como a implementação de todas estas possibilidades seria impraticável, escolheu-se uma que pode ser aplicada tanto na área de projeto e planejamento, quanto nas áreas de educação, treinamento e entretenimento.

A aplicação de realidade virtual distribuída selecionada é a de uma Cidade Virtual. Nesta aplicação, diversos usuários, representados por um avatar modelado como um veículo, podem percorrer a malha viária de uma mesma cidade virtual. As Figuras 26 (a) e (b) mostram uma visão geral de um trecho da cidade e uma visão do motorista (avatar), respectivamente.

Figura 26 (a) – Visão Geral de um Trecho da Cidade Virtual

Figura 26 (b) – Visão do Motorista, no Interior do Veículo (Visão da Esquerda)

A fim de facilitar a programação e o estudo do comportamento dos objetos envolvidos, os objetos foram agrupados nas seguintes classes funcionais: Objetos Estáticos (OE) e Objetos Dinâmicos Dirigidos pelo Usuário (ODU). Os OE são fixos e caracterizam o contexto da aplicação como edifícios, ruas, avenidas, placas de sinalização, etc. Os ODU representam as ações dos usuários sobre o ambiente virtual, no caso, os veículos.

Esta aplicação é extremamente versátil pois permite tanto uma abordagem não-imersiva quanto imersiva. Implementou-se uma versão não-imersiva, com a utilização de monitores de vídeo, como dispositivos padrão de saída. No entanto, o sistema pode ser facilmente modificado para exibir suas imagens através de dispositivos de saída imersivos, como é o caso dos HMDs.

A fim de conseguir-se um mundo virtual gerenciável do ponto de vista dos recursos computacionais disponíveis, restringiu-se a área da cidade à um trecho composto de 9 quadras. Como curiosidade cabe enfatizar que o trecho em questão foi inspirado em uma área existente na cidade de Bauru – SP. Uma visão da planta da cidade virtual modelada é apresentada na Figura 27.

Um mundo virtual com área geográfica maior certamente necessita de técnicas como as descritas em Macedonia et al. (1995), para ser devidamente gerenciado. No trabalho citado, os autores se preocuparam com aplicações amplamente distribuídas, e descrevem uma arquitetura constituída por partições de ambientes virtuais, às quais são associadas classes funcionais, espaciais e temporais com grupos *multicast*. Nesta arquitetura, os autores exploram as características atuais dos ambientes em grande escala do mundo real, restringindo-se o processamento de entidades e recursos de rede para suas áreas de interesse, via um Gerente Local de Área de Interesse. Um outro trabalho, com abordagem similar, pode ser encontrado em Araújo (1996).

Figura 27 – Visão da Planta da Cidade Virtual

Porém, o objetivo desta tese não está no estudo de um tipo particular de aplicação RV distribuída, e sim, na sua integração com a arquitetura CORBA. Por isso, de acordo com o apresentado no capítulo anterior, escolheu-se replicar inteiramente a base de dados do mundo virtual em todas as máquinas conectadas ao AVD, e não particioná-la.

A seguir, tem-se a descrição dos protótipos desenvolvidos.

6.3 Protótipo 1: Usando Apenas o WorldToolkit e suas Facilidades de Comunicação

Para o entendimento da implementação do protótipo 1, deve-se considerar a estrutura de aplicações WTK e seus recursos de comunicação.

6.3.1 O Loop de Simulação

No núcleo de uma aplicação escrita em WTK existe um *Loop* de Simulação, o qual lê sensores de entrada, atualiza objetos, e renderiza uma nova visão do grafo de cena no *display* (Figura 28).

Figura 28 – Um *Loop* de Simulação Padrão do WTK

A entrada no *Loop* de Simulação é conseguida através da chamada da função `WTuniverse_go()`, e a saída, é feita através da função `WTuniverse_stop()`. A criação do grafo de cena, na maioria de implementações, acontece antes da entrada no *Loop* de Simulação.

6.3.2 Os Recursos de Comunicação do WorldToolkit

As aplicações distribuídas WTK compartilham uma API comum, de modo a permitir a uma aplicação executar tanto em plataforma PC quanto UNIX, sem modificação.

Os níveis de uma rede WTK são quatro:

- ◆ **Nível de Transporte:** este é o nível mais baixo, e permite que múltiplos computadores possam ser interligados através de uma rede Ethernet. Suporta os padrões DEC/INTEL/XEROX, os mais usados pelas estações de trabalho. Não suporta os padrões 802.3, 802.5 e *Token Ring*. Os problemas de ordenação de bytes são transparentes ao WTK.
- ◆ **Nível de Protocolo:** a capacidade de rede é construída sobre os protocolos IP *multicast* e UDP.
- ◆ **Nível do WorldToolkit:** neste nível são oferecidas as primitivas de comunicação do WTK. As comunicações de rede são iniciadas através da chamada à função `Wtnet_open()`. Esta função não verifica se a rede está funcionando; assume que este é o caso. As comunicações com outras máquinas da simulação são estabelecidas através do uso de um endereço de grupo *multicast* válido (endereço Internet Classe D). Em adição ao endereço de grupo, as máquinas em uma mesma simulação somente se comunicam com outras máquinas que compartilham endereços de portas idênticos. Esta informação é passada através da chamada a `Wtnet_open()`. As informações entre máquinas são transmitidas através de itens de mensagens discretos, montados em pacotes UDPs válidos. Isto é feito por meio da função `Wtnet_additem()`, e enviados automaticamente pela rede através do WTK. A aplicação receptora obtém as informações recebidas através do uso da função `Wtnet_removeitem`.
- ◆ **Nível da Aplicação:** enquanto o WTK fornece o substrato para que a comunicação ocorra, cabe à aplicação, localizada neste nível, definir quais as informações a serem transmitidas.

6.3.3 A Estrutura da Aplicação do Protótipo 1

A estruturação de uma aplicação WTK deve ser iniciada pela definição do Grafo de Cena.

Uma cena é uma coleção de geometrias e luzes, em associação com a informação posicional destes elementos. Ou seja, uma cena WTK é construída fornecendo-se quatro elementos:

- ♦ **Geometrias:** incluem os arquivos de geometria estática que podem ser carregados durante a simulação, e as geometrias dinâmicas que podem ser criadas ao nível de polígono e vértices.
- ♦ **Luzes:** incluem as luzes que podem ser carregadas a partir de um arquivo, e as luzes criadas dinamicamente.
- ♦ **Informação Posicional:** também pode ser suprida através de arquivos ou geradas dinamicamente. A informação posicional descreve onde os elementos devem aparecer na cena, em relação a outro objeto, ou em relação a cena como um todo.
- ♦ **Fog:** é um efeito especial que simula as condições do ambiente, tais como neblina, fumaça, etc. O *fog* pode afetar parte das geometrias ou todas elas, se necessário.

De maneira a renderizar uma coleção de objetos do 3D em uma tela, o WTK precisa saber onde e como o observador está posicionado, com relação a cena. O *Wtviewpoint* é o objeto WTK que contém esta informação. Ele define a posição do observador na cena e a direção para a qual o observador está olhando.

O WTK mantém uma cena através da estrutura hierárquica de um grafo de cena. Esta estrutura mantém todos os elementos correntes na cena, tais como geometrias, luzes e informação posicional e *fog*. É ordenado como uma coleção de nós, formando um grafo direcionado acíclico.

Para a aplicação da Cidade Virtual, construiu-se o Grafo de Cena apresentado pela Figura 29.

Figura 29 – Grafo de Cena da Cidade Virtual

Explicando, sucintamente, os tipos de nós envolvidos, tem-se:

- ♦ **Root:** age como nó de topo em um grafo de cena. Cada grafo de cena possui apenas um nó *Root*, que não pode ser compartilhado com outros grafos de cena.
- ♦ **Light:** especifica uma luz WTK.
- ♦ **Transform (Xform):** fornece informação de posição e orientação. Uma vez que o “estado” pode ser propagado e acumulado em cada ponto de um grafo de cena, a informação de um nó de transformação pode afetar os nós posicionados após o mesmo.
- ♦ **Group:** tem a função de juntar nós-filhos, a fim de facilitar a construção da estrutura hierárquica.
- ♦ **Separator (Sep1):** impede que a informação de estado se propague de seus nós descendentes para seus nós-irmãos.
- ♦ **Geometry (Geom):** exibe um conjunto de polígonos associados com um material.
- ♦ **Movable (Move):** especifica um nó de geometria móvel. Este tipo especial de nó é uma entidade que facilita a construção de nós que podem alterar suas posições e direções, sem propagar suas transformações de estado para seus nós-irmãos. Incorpora em um único nó, as propriedades dos nós *Separator*, *Transform* e *Geometry*.

Conforme mostra a Figura 29, a sub-árvore mais à direita representa os nós móveis (*movable*) envolvidos na construção do objeto que representa a presença do usuário no ambiente (avatar). Portanto, no decorrer da execução da aplicação, para cada novo usuário que se conecta, será adicionada uma sub-árvore como a mostrada, ao grafo de cena.

Na implementação deste protótipo, utilizou-se a seguintes estruturas de dados principais, (supondo-se NETNODES como o número máximo de participantes):

- ◆ **Vetor de ponteiros para os objetos:** esta estrutura contém os ponteiros para a representação dos objetos móveis que representam cada avatar do ambiente. Formato: `WTnode *nodeobj[NETNODES]`
- ◆ **Vetor de tempo:** contém as informações de tempo, para cada usuário ativo no ambiente. Formato: `int nodetval[NETNODES]`
- ◆ **Vetor de tags:** esta estrutura contém os identificadores associados aos avatares ativos no ambiente. Formato: `int nodeid[NETNODES]`
- ◆ **Vetor de flags:** este vetor contém os sinalizadores usados para determinar quais usuários estão ativos na simulação, em um dado momento. Formato: `int nodeadd[NETNODES]`

O formato geral das mensagens utilizadas na comunicação entre os processos de usuário participantes do ambiente é mostrado na Figura 30.

Item	Comprimento	Tipo	Tag
------	-------------	------	-----

Figura 30 – Formato de Mensagem no Protótipo 1

O significado dos campos da mensagem são os seguintes:

- ◆ **Item:** representa a informação propriamente dita, gerada pela aplicação.
- ◆ **Comprimento:** indica o tamanho, em bytes, da área do buffer que contém o Item.
- ◆ **Tipo:** determina o tipo da mensagem. Foram usados 6 tipos de mensagens diferentes:
 - **NET_VIEWPOS:** usada para comunicar a posição atual de um avatar aos demais participantes do ambiente.

- **NET_BYE:** usada para sinalizar que determinado usuário está se retirando (desconectando) da simulação.
 - **NET_ALIVE:** usada para indicar que um novo usuário está se conectando ao ambiente.
 - **NET_REJECT:** usada na fase de conexão, durante a tentativa de se determinar uma identificação válida (*tag*) ao novo participante.
 - **NET_TIMEREQ:** usada quando determinado participante quer saber a latência da comunicação, com relação às demais máquinas integrantes do ambiente.
 - **NET_TIMEACK:** usada para indicar o recebimento de uma mensagem do tipo NET_TIMEREQ.
- ♦ **Tag:** contém a identificação válida para o participante.

As Figuras 31 a 36 mostram a estrutura geral da implementação do Protótipo 1.

Principais variáveis e constantes globais
 /* Exceto as estruturas de dados e tipos de mensagens explicados anteriormente */
 Constantes inteiras: ESPERA_NET=20; NETNODES=16;
 Variáveis Inteiras: netid=0, espera_net=ESPERA_NET, tval, nval, tag, t0, renderiza=1;
Programa Principal

- Obter o nome do modelo de background a ser carregado, a partir dos parâmetros da linha de comando;
- Criar o universo estático;
- Criar a janela inicial;
- Criar o nó *Root*;
- Associar um ponto de vista ao universo;
- Iniciar a comunicação via rede:
 - Se a rede não está ativa
 - então
 - destruir os objetos do universo e encerrar a aplicação;
 - senão
 - tval= um valor de tempo para formar a identificação (*tag*) deste novo usuário;
 - enviar uma mensagem do tipo NET_ALIVE, contendo o valor de tempo (*tval*),
 - para todos os demais participantes do ambiente;

Fim-Se

Figura 31 – Algoritmo do Programa Principal

- Carregar o modelo de background (estacionário);
- Chamar a função **setup_visao_dos_outros()**, para criar os avatares dos outros participantes do ambiente (se existirem);
- Carregar luzes padrão;
- Abrir as entradas pelo teclado;
- Usar o mouse como sensor;
- Associar a função **actions()**, como função de ação do universo;
- Adequar a velocidade do sensor às dimensões do universo;
- Acertar o ponto de vista inicial (posição e orientação) do usuário;
- Associar o sensor (mouse) ao ponto de vista;
- Preparar a aplicação para a entrada no Loop de Simulação;
- Entrar no Loop de Simulação, chamando a função **WTuniverse_go()**;
- Enviar uma mensagem do tipo **NET_BYE** aos outros participantes, sinalizando que este usuário está se retirando do ambiente;
- Excluir todos os objetos do universo.

Figura 31 – Algoritmo do Programa Principal (continuação)

Função setup_visao_dos_outros()

Variável local

Inteira: i;

Carregar o arquivo com o objeto (“carro.nff”) que representa os demais participantes;

Se o arquivo não existe

Então

/* desconectar-se da rede */

Enviar uma mensagem do tipo **NET_BYE** aos outros participantes, sinalizando que este usuário está se retirando do ambiente, com campo tag=0;

Sinalizar a desconexão da rede (**WTnet_close()**);

Destruir os objetos do universo e finalizar a aplicação;

Fim-Se

Preencher as estruturas de dados com os valores iniciais, para cada avatar:

i=0;

Enquanto i<NETNODES faça:

nodeobject[i] recebe o ponteiro para o objeto carregado do arquivo;

nodetval[i], nodeid[i], nodeadd[i], recebem zero;

incrementar i;

Fim-Enquanto

Figura 32 – Algoritmo da Função **setup_visao_dos_outros()**

Função actions()

```

Obter a próxima mensagem da fila de mensagens;
Caso o campo Tipo da mensagem seja:
  NET_VIEWPOS: /* um usuário remoto mudou sua posição e
                orientação */
    Se a identificação do emissor não é válida (tag > NETNODES)
    ou a identificação do emissor ainda não foi efetivada
    (nodeid[tag-1] é zero)
        então
            Finaliza comando Caso;
        senão
            Se é a primeira vez que o avatar é posicionado
            (nodeadd[tag-1] == 1)
                então
                    Adiciona o objeto no grafo de cena, em sua
                    posição inicial;
                    Marca o objeto como "adicionado"
                    (nodeadd[tag-1] = 0);
            Fim-Se;
        Fim-Se;
    Mudar a posição e a orientação do avatar que enviou a
    mensagem, de acordo com a informação do campo Item;
    Acertar as estruturas de dados;
    Finaliza comando Caso;
  NET_ALIVE: /* um usuário remoto sinaliza que está ativo */
    Add_node(Item, tag);
    Finaliza comando Caso;
  NET_BYE: /* usuário está se desconectando */
    Se tag < 0
        então
            Imprime que o participante com identificação no campo
            tag está saindo;
        Fim-se;
        Finaliza comando Caso;
  NET_REJECT: /* a identificação sugerida para este usuário não foi
               aceita */
    Destrói os objetos do usuário remoto, chamando
    delete_node(Item, tag);
    Finaliza comando Caso;
  NET_TIMEREQ: /* algum usuário está pedindo o tempo */
    Se o tempo foi requerido diretamente ao usuário local
        então
            Confirma o recebimento, enviando uma mensagem do
            tipo
            NET_TIMEACK;
        Fim-Se;
    Finaliza comando Caso;

```

Figura 33 – Algoritmo da Função actions()

```

NET_TIMEACK: /* obtém um tempo de resposta */
    Se a mensagem recebida é devido a um pedido de tempo feita
    por este usuário local
    então
        Obtém o tempo atual;
        Faz a diferença entre o tempo atual e o tempo em
        que a Mensagem NET_TIMEREQ foi enviada;
        Imprime o tempo de resposta quanto ao emissor da
        mensagem
        NET_TIMEACK;
    Fim-Se;
    Finaliza comando Caso;
Default: /* tipo de mensagem recebida não existe */
    Descarta a mensagem recebida;
Fim-Caso;

/* Enviar o ponto de vista deste usuário para todos os demais */
Se (netid < 0) /* se o usuário já possui uma identificação */
    então
        Mandar uma mensagem NET_VIEWPOS, com campo Item contendo a
        posição e a orientação do usuário local, no campo tag colocar identificação
        deste usuário (netid);
    senão /* verifica se pode associar uma identificação inicial ao usuário */
        Percorrer o vetor nodeid, e contar o número de nós ativos (com
        identificação);
        Se existirem usuários ativos
            então
                Decrementa contador de espera por identificação (espera_net);
                Se (espera_net == 0)
                    então
                        netid = identificação gerada;
                        enviar mensagem do tipo NET_ALIVE, com a nova
                        identificação (netid) no campo tag;
                Fim-Se;
            Fim-Se;
        Fim-Se;
    Fim-Se;
/* processa entradas pelo teclado */
Enquanto existirem valores entrando via teclado faça:
    Chamar função manipula_tecla(tecla), passando o valor da tecla
    (character) como parâmetro;
Fim-Enquanto;
Retornar;

```

Figura 33 – Algoritmo da Função actions() (continuação)

Função add_node(val,tag)

Variáveis locais

Inteiras: id, meu_id;

Se emissor não possui identificação (tag= 0)

então

id=(Item%NETNODES)+1; /*Tenta obter identificação a partir do campo Item da mensagem (contém um valor de tempo) */

meu_id=netid;

Se (nodeid[id-1]=1) /*id já existe na tabela nodeid */

então

Sinaliza ao emissor que a sugestão não é válida, enviando uma mensagem do tipo NET_REJECT a ele;

Retornar;

Fim-Se;

/* Verifica se a nova identificação coincide com a identificação deste usuário */

Se (meu_id==id)

então /*Desfazer o impasse observando o tempo de envio da mensagem*/

Se (Item == tval) /* O menor valor vence - se ambos forem iguais, ambos perdem */

então

sinaliza a rejeição da identificação do usuário remoto, enviando uma mensagem NET_REJECT;

tval=geração de novo tempo;

envia mensagem NET_ALIVE, com o tempo tval (no campo Item);

espera_net=ESPERA_NET; /*Marca que o usuário local ainda espera uma identificação */

netid=0;

Retornar;

senão

Se (Item < tval) /* usuário local perdeu */

então

tval=geração de novo tempo;

envia mensagem NET_ALIVE, com o tempo tval (no campo Item);

espera_net=ESPERA_NET; /*Marca que o usuário local ainda espera uma identificação */

netid=0;

senão /* usuário remoto perdeu*/

sinaliza a rejeição da identificação do usuário remoto, enviando uma mensagem NET_REJECT;

Retornar;

Fim-Se;

Fim-Se;

Figura 34 – Algoritmo da Função add_node()


```

/* Se não existe conflito com a identificação do usuário local, então
avisar os demais usuários sobre a identificação do usuário local */
Se (netid < 0)
    então
        enviar uma mensagem NET_ALIVE, contendo netid (no campo
        Item)
    Fim-Se;
senão /*o emissor já possui identificador */
Se (tag > NETNODES)
    então
        enviar uma mensagem NET_REJECT;
    Fim-Se;
    id = tag;
    Se (nodeid[id-1] < 0) /* Já possui uma entrada na tabela*/
        então
            Retornar;
    Fim-Se;
Fim-Se;
Adiciona a identificação do novo participante: nodeid[id-1] = id;
nodetval[id-1] = Item;
Marca o usuário como adicionado recentemente: nodeadd[id-1] = 1;
Retornar;

```

Figura 34 – Algoritmo da Função add_node() (continuação)

Função delete_node(val,tag)

Variáveis Locais

Inteiras: id, meu_id;

```

id = tag;
Se (id < 0)
    então
        id = (val % NETNODES) + 1;
        /* verificar se id é válido */
        Se ( id > NETNODES )
            então
                Retornar;
        Fim-Se;
        Se ( (nodeid[id-1] ) && nodetval[id-1] == nval) /*se id é válido*/
            Então
                nodeid[id-1] = 0;
                nodetval[id-1] = 0;
                nodeadd[id-1] = 0;
                Se ( tag < 0 )
                    então
                        Remover o objeto (nodeobj[id-1]) do grafo de cena;
                Fim-Se;
            Fim-Se;
        Fim-Se;

```

Figura 35 – Algoritmo da Função delete_node()

```

/* checa para ver se o usuário local foi rejeitado */
meu_id = netid;
Se (meu_id == 0)
    então
        meu_id = (tval % NETNODES) + 1;
Fim-Se;
Se ( (id == meu_id) && (tval == val) )
    então
        netid = 0;
        espera_net = ESPERA_NET;
        tval = geração de novo tempo;
        envia mensagem do tipo NET_ALIVE, com campo Item = tval
        e tag = 0;
Fim-Se;
Fim-Se;

```

Figura 35 – Algoritmo da Função delete_node() (continuação)

Função manipula_tecla(tecla)

Variáveis locais:

Inteiro i;

Caracter: tecla;

Caso a tecla seja:

‘i’: imprime o número de polígonos envolvidos na cena e o número de frames por segundo;

Finaliza comando Caso;

‘w’: /* comuta o modo de renderização */

Se (renderiza == 1)

então

renderiza = 0;

Mudar exibição para renderizado (cena exibida com texturas e iluminação);

senão

renderiza = 1;

Mudar exibição para wireframe (cena exibida sem texturas e iluminação);

Fim-Se;

Finaliza comando Caso;

‘q’: /* finaliza a aplicação */

Chamar a função WTuniverse_stop(), para sair do Loop de Simulação;

Finaliza comando Caso;

Figura 36 – Algoritmo da Função manipula_tecla()

```

'r': /* pede tempo de resposta*/
  Se este usuário ainda não possui uma identificação (netid= =0)
    então
      imprimir mensagem "este usuário ainda não possui identificação";
      Finaliza comando Caso;
  Fim-Se;
  t0=tempo atual;
  tag=identificação deslocada 8 bits para esquerda;
  i=0;
  Enquanto i<NETNODES faça:
    Se (nodeid[i]<>0)
      então
        enviar uma mensagem do tipo NET_TIMEREQ,
        com t0 no campo Item e (tag | nodeid[i]) no campo tag;
      Fim-Se;
      Incrementa i;
  Fim-Enquanto;
  Finaliza comando Caso;
Fim-Caso;
Retornar;

```

Figura 36 – Algoritmo da Função manipula_tecla() (continuação)

6.4 Protótipo 2: Usando o WorldToolkit e o ORBIX

Este segundo protótipo foi construído com a finalidade de se verificar a viabilidade da integração da aplicação WTK e a plataforma ORBIX. Utilizou-se, aqui, somente o modelo cliente/servidor. Nesta implementação, como na anterior, cada participante (cliente) possui uma cópia de todo o universo, ou seja, o mesmo é totalmente replicado entre os participantes. Porém, clientes não se comunicam diretamente entre si, apenas por intermédio do servidor, localizado no Gerente de Facilidades de Realidade Virtual (GFRV). A comunicação é, portanto, ponto a ponto.

6.4.1 Definição da *Interface* com o GFRV

Seguindo a metodologia proposta no capítulo anterior, exceto pelas etapas que envolviam a utilização do Serviço de Eventos, transferiu-se para o GFRV a

responsabilidade de gerenciar a comunicação, bem como a conexão e desconexão de novos participantes (clientes), conforme mostra a Figura 37.

De acordo com a abordagem da arquitetura CORBA, os clientes não devem se preocupar com detalhes de comunicação, mas sim com quais serviços podem invocar, por intermédio da *interface* implementada pelo servidor, neste caso, pelo GFRV.

Portanto, a estrutura da aplicação WTK, representada pelo protótipo 1, foi devidamente modificada, substituindo-se, principalmente, as trocas de mensagens pelas devidas invocações dos métodos providos pela *interface* com o GFRV. Após estas modificações, obteve-se uma aplicação cliente WTK-ORBIX.

Figura 37 – Visão Geral da Estrutura do Protótipo 2

Para a definição da *interface* com o GFRV, foram utilizadas as seguintes estruturas de dados principais:

- **Tabela de Objetos (tipo TableObj):** contém um *flag* (inteiro) para cada avatar (usuário) do sistema. Se o *flag* for igual a zero, indica que o usuário correspondente não está conectado ao ambiente (seu avatar não deve aparecer).
- **Tabela de Posições (tipo TablePos):** contém as informações de posição e orientação, bem como um *flag*, para cada avatar (usuário) do sistema. O campo do *flag*, se igual a um, indica que o usuário correspondente sofreu alguma alteração (pode ter se conectado ou desconectado do ambiente).

Antes da apresentação da definição da *interface* IDL criada, cabe ressaltar alguns aspectos importantes quanto a construção mesma:

- Uma *interface* IDL é composta pela especificação de seus atributos (propriedades) e de suas operações.

- Cada operação na *interface* é mapeada para uma função-membro C++, de mesmo nome, na classe C++ IDL.
- Um atributo, por sua vez, é mapeado em um par de funções-membro C++: uma função responsável por ler o valor (*get*) e outra, responsável por escrever o valor (*set*). Os atributos *readonly* são mapeados para apenas uma função que lê um valor (*get*).
- Quanto à passagem de parâmetros, existem 3 modos disponíveis:
 - ♦ Modo *in*: o parâmetro é passado do cliente para o objeto chamado.
 - ♦ Modo *out*: o parâmetro é passado do objeto chamado para o cliente.
 - ♦ Modo *inout*: o parâmetro é passado em ambas as direções.

Considerando o exposto, foi definida a *interface* VirCity, apresentada na Figura 38.

```
//Arquivo cidade.idl
//definição de tipos
typedef float WT_p3[3];
typedef float WT_q[4];
typedef struct pq{
    WT_p3 p; // Posição da carroceria
    WT_q q; // Orientação da carroceria
    WT_q qroda; //Orientação das rodas dianteiras
    WT_p3 pRodaDE, pRodaDD; // Posição das rodas dianteiras

    } WT_pq;
typedef short Flags[16];
typedef WT_pq Positions[16];
typedef WT_q Orientacoes[16];

typedef struct table1{ Flags Flag;
    } TableObj; //definição da tabela de objetos

typedef struct table2{
    Positions Pos;
    Flags Change;
    } TablePos; //definição da tabela de posições

//definição da interface
interface VirCity{
    exception IdRejected{};
    TableObj NetAlive(in long Id) raises (IdRejected);
    void NetBye(in long Id);
    TablePos NetSendPos(in long Id, in WT_pq Pos);
    TableObj NetGetObjects(in long Id);
    long NetTimeReq(in long Id);
    long NetTimeAck();
};
```

Figura 38 – Definição da Interface IDL VirCity

A seguir, tem-se a descrição das operações fornecidas pela *interface*:

- ♦ **Operação NetAlive(Id):** esta operação é usada para permitir a conexão de um usuário (cliente) ao ambiente. O parâmetro a ser passado ao GFRV é uma identificação (Id) sugerida pelo cliente. Se o GFRV constatar que já existe outro cliente com a mesma identificação, é levantada uma exceção ao cliente chamador da operação. Caso contrário, a operação retorna ao cliente chamador uma tabela indicando quais os objetos (avatars representando os outros usuários) estão ativos no momento.
- ♦ **Operação NetBye(Id):** é usada por um cliente para sinalizar ao GFRV que está se desconectando do ambiente. O único parâmetro a ser passado é a identificação do cliente (Id).
- ♦ **Operação NetSendPos(Id, Pos):** é usada por um cliente para notificar o GFRV da posição e orientação atual de seu avatar. Os parâmetros passados ao GFRV são a identificação do cliente (Id) e uma estrutura contendo as posições e orientações das partes do avatar (Pos). O GFRV, por sua vez, retorna ao cliente chamador uma tabela contendo as posições e orientações de todos os avatares do ambiente e um *flag* de alteração (TablePos).
- ♦ **Operação NetGetObjects(Id):** é usada por um cliente para solicitar ao GFRV uma tabela de objetos atualizada. O único parâmetro passado pelo cliente é sua identificação (Id). O GFRV retorna a tabela de objetos solicitada (TableObj).
- ♦ **Operação NetTimeReq(Id):** permite a um cliente solicitar ao GFRV a verificação do tempo de resposta da rede. O cliente passa como parâmetro sua identificação. O GFRV, ao receber esta solicitação, começa a contar os clientes que se comunicam com ele. Retorna o valor 1, ao cliente chamador, para indicar que recebeu a solicitação.
- ♦ **Operação NetTimeAck():** é utilizada por um cliente que já tenha, anteriormente, solicitado a verificação de tempo de rede (NetTimeReq()) ao GFRV. Retorna o valor 0, se nem todos os clientes se comunicaram com o GFRV, após o recebimento da solicitação. Caso contrário, retorna o valor 1 (sucesso).

A Figura 39 ilustra a sequência de comunicação típica entre um cliente e o Gerente de Facilidades de Realidade Virtual.

Figura 39 – Comunicação entre um Cliente e o GFRV, no Protótipo 2

Conforme mostrado na Figura 39, a comunicação entre cliente e GFRV pode ser resumida da seguinte forma:

1. Cada novo cliente pode conectar-se ao ambiente invocando a operação NetAlive(Id), e passando uma identificação inicial ao GFRV.
2. O GFRV, ao executar um NetAlive(Id), verifica se existe um outro cliente com a mesma identificação enviada pelo chamador. Em caso negativo, aceita o novo cliente, ajusta suas tabelas com a nova informação e retorna ao chamador uma tabela de objetos. Esta tabela contém *flags* que indicam quais os clientes ativos no ambiente. Caso contrário, rejeita a conexão, retornando ao chamador uma exceção (Net_Reject). O cliente, após receber uma exceção, deve gerar uma nova identificação e invocar novamente uma operação NetAlive(Id), repetindo este procedimento até conseguir uma identificação válida.
3. Os Clientes enviam constantemente (a cada iteração do *Loop* de Simulação), informações de reposicionamento ao GFRV, através da operação NetSendPos(Id,Pos).
4. O GFRV, ao executar uma operação NetSendPos(Id,Pos), atualiza suas tabelas, e retorna ao chamador uma tabela (TablePos), contendo as posições e orientações de todos os avatares, bem como *flags* de alteração.
5. Ao receber a tabela de posições, o cliente verifica os *flags* de alteração, para detectar a desconexão de um cliente antigo ou a conexão de cliente novo. Se

este for o caso, o cliente invoca a operação `NetGetObjects(Id)`, solicitando ao GFRV a Tabela de Objetos atual.

6. O GFRV, ao executar a operação `NetGetObjects(Id)`, retorna ao chamador a tabela atualizada.
7. Um cliente pode retirar-se do ambiente, avisando o GFRV através da invocação da operação `NetBye(Id)`.

6.4.2 Compilação e Implementação da *Interface* com o GFRV

O compilador IDL do ORBIX produziu, a partir do arquivo `cidade.idl` (contendo a *interface* `VirCity`), os seguintes arquivos C++:

- **Cidade.hh:** um arquivo *header* (cabeçalho) que deve ser incluído na implementação da *interface* (no GFRV) e em todos os clientes. Este arquivo define a visão C++ do IDL.
- **CidadeS.cpp:** um arquivo fonte a ser compilado e incluído na implementação da *interface* (apenas no GFRV).
- **CidadeC.cpp:** um arquivo fonte a ser compilado e incluído nos clientes.
- **Cidade.ih:** uma versão inicial do arquivo *header* da classe de implementação.
- **Cidade.ic:** uma versão inicial do arquivo fonte da classe de implementação.

A Figura 40 mostra o arquivo *header* `Cidade_i.h`, escrito em C++, a partir de `Cidade.ih`.

```
// Arquivo cidade_i.h

#ifndef cidade_ih
#define cidade_ih
#define NETNODES 16
#include "cidade.hh"

CORBA::Long FlagTime[NETNODES];
CORBA::Long TimeResp, TimeAck;
TablePos Tabela_Posicoes;
TableObj Tabela_Objeto;
```

Figura 40 - O Arquivo *header* `Cidade_i.h`


```

class VirCity_i : public virtual VirCityBOAImpl {
public:
    VirCity_i ();
    void CheckTimeResp(CORBA::Long Id);
    virtual TableObj NetAlive (CORBA::Long Id,
CORBA::Environment &IT_env=CORBA::default_environment);
    virtual void NetBye (CORBA::Long Id, CORBA::Environment
&IT_env=CORBA::default_environment) ;
    virtual TablePos NetSendPos (CORBA::Long Id, const WT_pq&
Pos, CORBA::Environment &IT_env=CORBA::default_environment) ;
    virtual TableObj NetGetObjects (CORBA::Long Id,
CORBA::Environment &IT_env=CORBA::default_environment);
    virtual CORBA::Long NetTimeReq (CORBA::Long Id,
CORBA::Environment&) ;
    virtual CORBA::Long NetTimeAck (CORBA::Environment&) ;
};
#endif

```

Figura 40 - O Arquivo *header* Cidade_i.h (continuação)

A implementação propriamente dita da *interface* está presente no arquivo Cidade_i.cpp (escrito a partir de Cidade.ic). Neste, tem-se a classe C++ VirCity_i, como mapeamento da *interface* IDL original, VirCity.

A Figura 41 mostra um fragmento do arquivo Cidade_i.cpp.

```

//Arquivo Cidade_i.cpp:implementação da interface com o GFRV
#include "cidade_i.h"
VirCity_i::VirCity_i() /* Construtor - inicia as variaveis */
{
    CORBA::Short i;
    for(i=0;i<NETNODES;i++)
    {
        Tabela_Objeto.Flag[i]=0;
        Tabela_Posicoes.Change[i]=0;
        FlagTime[i]=1;
    }
    TimeResp=0;
    TimeAck=0;
}
TableObj VirCity_i::NetAlive (CORBA::Long Id, CORBA::Environment&)
{
    CORBA::Short i;
    CheckTimeResp(Id);
    if ( Tabela_Objeto.Flag[Id] == 1)/*Se outro objeto ja possui */
        throw VirCity::IdRejected(); /* este Id, envia excecao */
    else{ Tabela_Objeto.Flag[Id]=1; /* Atualiza tabelas com o */
        for(i=0;i<NETNODES;i++) /* objeto */
            Tabela_Posicoes.Change[i]=1;
        Tabela_Posicoes.Change[Id]=0;
    }
    return(Tabela_Objeto); /* Envia tabela dos objetos */
}
...
/* implementação das demais funções */
CORBA::Long VirCity_i:: NetTimeAck (CORBA::Environment&) {
    return(TimeAck);
}

```

Figura 41 - O Arquivo de Implementação Cidade_i.cpp

6.4.3 Implementação da Aplicação Cliente

A aplicação cliente segue, basicamente, a mesma estrutura apresentada no protótipo 1, exceto pela substituição da troca explícita de mensagens pela interação via chamada dos métodos implementados no GFRV.

As Figuras 42 a 44 apresentam os algoritmos das partes principais da aplicação cliente, com ênfase nos aspectos de integração com o ORBIX.

Principais variáveis e constantes globais

```
/* Exceto as estruturas de dados explicadas anteriormente */
Constante inteira: NETNODES=16;
Variáveis Inteiras: netid=0, tval;
...
/* Declaração do objeto que representa o servidor dentro do GFRV */
VirCity_var NetPosSrv;
```

Programa Principal

- Obter o nome do host que contém o GFRV, a partir da linha de comando;
- Criar o universo estático;
- Criar a janela inicial;
- Criar o nó *Root*;
- Associar um ponto de vista ao universo;
- Carregar o modelo de background (estacionário);
- Carregar luzes padrão;
- Abrir as entradas pelo teclado;
- Usar o mouse como sensor;
- Associar a função **actions()**, como função de ação do universo;
- Adequar a velocidade do sensor às dimensões do universo;
- Acertar o ponto de vista inicial (posição e orientação) do usuário;
- Associar o sensor (mouse) ao ponto de vista;
- Iniciar a comunicação via ORBIX, chamando a função **setup_net(host)** e passando o nome do host como parâmetro;
- Preparar a aplicação para a entrada no Loop de Simulação;
- Entrar no Loop de Simulação, chamando a função **WTuniverse_go()**;
- Sinalizar ao GFRV que este usuário está se desconectando, chamando **NetPosSrv->NetBye(netid)**;
- Excluir todos os objetos do universo.

Figura 42 – Algoritmo do Programa Principal do Cliente

Função actions()

```

Obter a posição e orientação do avatar local;
Enviar esta informação ao GFRV, e obter a tabela de posições como parâmetro de
retorno: Table_Pos=NetPosSrv->NetSendPos(netid,pos);
Observar a tabela de posições:
  Se houve mudança em algum cliente
  então
    Fazer uma cópia da tabela de objetos atual (Tabela de objetos antiga recebe
    a tabela atual);
    Solicitar uma tabela de objetos atualizada ao GFRV:
    Table_Obj=NetPosSrv->NetGetObjects(netid);
    Percorrer a tabela de objetos antiga e a atual, a procura de diferença:
    Se encontrou uma diferença
    então
      Se é um novo cliente no ambiente
      então
        carregar a geometria correspondente ao novo avatar
        senão /* um cliente se desconectou */
        remover os nós correspondentes a este cliente, do grafo de
        cena;
      Fim-Se;
    Fim-Se;
  Fim-Se;
Percorrer a tabela de objetos atual, e atualizar a posição e orientação de todos os
avatares;
Ler o teclado;
Caso a tecla seja:
  'i': imprime o número de polígonos envolvidos na cena e o
  número de frames por segundo;
  Finaliza comando Caso;
  'w': /* comuta o modo de renderização */
  Se (renderiza==1)
  então
    renderiza=0;
    Mudar exibição para renderizado (cena exibida com texturas e
    iluminação);
  senão
    renderiza=1;
    Mudar exibição para wireframe (cena exibida sem texturas e iluminação);
  Fim-Se;
  Finaliza comando Caso;
  'q': /* finaliza a aplicação */
  Chamar a função WTuniverse_stop(), para sair do Loop de Simulação;
  Finaliza comando Caso;

```

Figura 43 – Algoritmo da Função actions()

```

'r': /* pede tempo de resposta*/
    Obter o tempo de resposta invocando as funções NetTimeReq(netid) e
    NetTimeAck() no GFRV;
    Imprimir o tempo de resposta.
    Finaliza comando Caso;
'0': /* muda geometria da cidade – cidade 0*/
    Deletar o nó de geometria antigo;
    Carregar novo arquivo de geometria;
    Finaliza comando Caso;
...
'5': /* muda geometria da cidade – cidade 5*/
    Deletar o nó de geometria antigo;
    Carregar novo arquivo de geometria;
    Finaliza comando Caso;
Fim-Caso;
Retornar;

```

Figura 43 – Algoritmo da Função actions() (continuação)

Função setup_net(host)

Variáveis Locais

Inteiras: i, erro=0;

- try{ /*Tentar fazer a ligação entre o objeto e o GFRV, onde o objeto servidor está realmente executando*/

NetPosSrv = VirCity::_bind(":CidVir",host);
 } catch(CORBA::SystemException& se){
 Imprimir a mensagem("\nBind para Objeto CidVir, no GFRV
 falhou...\n");
 Sair da aplicação;
 }
- Repetir: /* Consegue uma identificação (Id) */

try{
 erro=0;
 tval = identificação gerada;
 netid = (tval % NETNODES);
 Table_Obj=NetPosSrv->NetAlive(netid);
 } catch(VirCity::IdRejected& r){
 erro=1;
 }

 até (erro==0);
- Inserir um avatar (automóvel) no grafo de cena, para representar o cliente local;
- Percorrer a tabela de objetos recebida (Table_Obj) e carregar a geometria de avatar (automóvel) para cada cliente ativo diferente do cliente local (Table_Obj.Flag[i] && i <> netid), para o grafo de cena;
- Retornar;

Figura 44 – Algoritmo da Função setup_net()

6.4.4 Implementação do Programa GFRV

Tendo-se implementado a classe que representa a *interface*, e suas estruturas de dados, escreveu-se o programa GFRV, o qual cria um objeto servidor e avisa ao ORBIX que a iniciação do servidor está terminada.

A Figura 45 mostra a implementação do programa GFRV.

```
// Arquivo GFRV.cpp

#include "cidade_i.h"
#include <iostream.h>

void main (void)
{
    VirCity_i ServerPosition;

    try{
        CORBA::Orbix.impl_is_ready("CidVir");
    } catch(CORBA::SystemException& se){
        cerr<<"Excecao inesperada:"<<endl << &se;
    }
    cout<<"GFRV terminando"<<endl;
}
```

Figura 45 – O Programa Fonte GFRV.cpp

6.5 Protótipo 3: Usando o WorldToolkit, o ORBIX e o OrbixTalk

Nesta terceira implementação, seguiu-se integralmente a metodologia apresentada no capítulo anterior. Resolveu-se, neste terceiro protótipo, reduzir o papel do Servidor (elemento do GFRV), restringindo suas tarefas ao controle da conexão e desconexão de clientes ao ambiente virtual distribuído. A utilização do OrbixTalk facilitou a comunicação direta entre clientes, bem como a comunicação entre grupos *multicast*.

Conforme exposto no item 6.1.2, o OrbixTalk impõe a estruturação dos objetos comunicantes segundo sua função de Produtor (*Talker*) ou Consumidor (*Listener*) de um evento (sinalizado através de mensagem). Portanto, os clientes, neste protótipo, são ao mesmo tempo *Listerners* e *Talkers* de um mesmo *TopicName*,

para poderem enviar e receber mensagens de reposicionamento, enquanto que o servidor é um *Talker*, uma vez que sinaliza eventos de conexão e desconexão de clientes. A comunicação entre *Talkers* e *Listeners* é feita via protocolo *multicast*.

Porém, nem todas as operações necessitam ser sinalizadas para todos os elementos do grupo. Um exemplo disto, é o pedido de conexão e desconexão de um cliente. Como o responsável por gerenciar estas tarefas é um servidor pertencente ao GFRV, a comunicação entre cliente e servidor, neste caso, pode ser *unicast*. Apenas a divulgação da entrada ou saída de um cliente, aos demais participantes, necessita ser feita através de *multicast*.

Por isso, neste protótipo, foram definidas duas *interfaces* (já introduzidas no capítulo anterior):

- InterfClient: a qual oferece as operações que necessitam da comunicação *multicast* e do Serviço de Eventos (OrbixTalk);
- InterfServer: a qual fornece as operações que necessitam da comunicação *unicast*, sem o Serviço de Eventos (usam apenas o ORBIX).

A Figura 46 ilustra o relacionamento entre clientes e servidor, no Protótipo 3.

Figura 46 – Interação entre Clientes e Servidor (GFRV), no Protótipo 3

6.5.1 Definição das *Interfaces* do Protótipo 3

Neste item são apresentadas as definições das duas *interfaces* do protótipo 3.

A) *Interface* InterfClient

As operações principais oferecidas pela *interface* InterfClient são:

- **Operação NetAlive(Id,Obj):** esta operação é chamada pelo servidor do GFRV para adicionar um novo avatar ao grafo de cena dos clientes. Os parâmetros passados pelo servidor são: a identificação do novo cliente e o nome do arquivo com a geometria do novo nó. Atualiza, também, a estrutura de dados que representa os clientes presentes no ambiente (Table_Obj).
- **Operação NetBye(Id):** esta operação é chamada pelo servidor do GFRV para atualizar as estruturas de dados dos clientes, devido à desconexão de um deles. O parâmetro passado pelo servidor é a identificação do cliente que se desconectou.
- **Operação NetViewPos(Id,Pos):** esta operação é chamada por um cliente para avisar, aos demais clientes, sobre sua nova posição e orientação. Os parâmetros passados para esta operação são: a identificação do cliente (Id) e uma estrutura de dados contendo sua nova posição e orientação (Pos).
- **Operação NetTimeReq(Time,message,IdReq):** usada por um cliente para solicitar a medição do tempo de resposta da rede. Os parâmetros para esta operação são: o tempo em que a operação foi disparada (Time), o número da mensagem (message) e a identificação do cliente (IdReq).
- **Operação NetTimeAck(Time, message, IdReq, IdAck):** usada para confirmar o recebimento da solicitação de tempo de resposta. Os parâmetros são os mesmos da operação NetTimeReq(), acrescidos da identificação do cliente que está confirmando (IdAck).

A Figura 47 ilustra o arquivo Clientes.idl, o qual contém a declaração dos tipos globais usados e a definição da *Interface* InterfClient, escrita em IDL.

```

// Arquivo Clientes.idl
// Declaração de tipos globais
typedef float WT_p3[3];
typedef float WT_q[4];
typedef struct pq{ WT_p3 p;
                  WT_q q;
                } WT_pq;

// Declaração da Interface InterfClient
interface InterfClient{
    oneway void NetAlive(in long Id, in string Obj);
    oneway void NetBye(in long Id);
    oneway void NetViewPos(in long Id, in WT_pq Pos);
    oneway void NetTimeReq(in long Time, in long Message, in
                          long IdReq);
    oneway void NetTimeAck(in long Time, in long Message, in
                          long IdReq, in long IdAck);
};

```

Figura 47 – A Definição da *Interface* InterfClient

B) *Interface* InterfServer

A *interface* InterfServer fornece as seguintes operações principais:

- **Operação NetOpen(Id,Obj):** esta operação pode ser invocada por um cliente, ao servidor do GFRV, para solicitar sua conexão ao ambiente. Deve passar ao servidor uma identificação (Id) e o nome do arquivo que contém a geometria de seu avatar (Obj). Esta operação deve verificar se outro cliente possui a mesma identificação sugerida. Se for este o caso, retorna uma exceção ao chamador. Caso contrário, retorna uma tabela de objetos (TableObj), a qual contém os nomes dos arquivos de geometria de todos os avatares presentes no ambiente, bem como *flags* indicando os avatares que estão ativos.
- **Operação NetClose(Id):** um cliente deve invocar esta operação, no servidor, para se desconectar do ambiente. Deve passar, como parâmetro, sua identificação.

O arquivo Servidor.idl, o qual contém a declaração dos tipos globais usados e a definição da *Interface* InterfServer, é mostrado na Figura 48.


```

// Arquivo Servidor.idl
// Tipos Globais
typedef char File[12];
typedef File Files[16];
typedef short Flags[16];
typedef struct table1{  Files Obj;
                        Flags Flag;
                        } TableObj;

// Declaracao da Interface InterfServer
interface InterfServer{
    exception IdRejected{};
    TableObj NetOpen(in long Id, in File Obj) raises (IdRejected);
    void NetClose(in long Id);
};

```

Figura 48 – A Definição da *Interface InterfServer*

A Figura 49 ilustra a comunicação entre cliente/servidor e cliente/cliente, descrita a seguir (com exceção das solicitações para tomada do tempo de resposta):

1. Cada novo cliente pode conectar-se ao ambiente chamando o método NetOpen().
2. O servidor, após receber um NetOpen(), verifica se o novo cliente pode conectar-se. Em caso afirmativo, retornará a este uma tabela de objetos (TableObj) com as posições dos outros avatares participantes do ambiente. Caso contrário, enviará um Net_Reject.
3. O servidor avisa os demais clientes do ambiente quando acontece a conexão de um novo participante, através da chamada ao método NetAlive().
4. Os clientes trocam informações de reposicionamento entre si, através da chamada ao método NetViewPos().
5. Um cliente pode retirar-se do ambiente, avisando o servidor através da chamada ao método NetClose().
6. O servidor avisa a todos os demais clientes participantes do ambiente que um cliente retirou-se da simulação, chamando o método NetBye().

6.5.2 Compilação e Implementação das *Interfaces* do Protótipo 3

A partir do arquivo Clientes.idl, que contém a definição da *interface* InterfClient, o compilador IDL produziu os seguintes arquivos C++:

- **Cientes.hh:** um arquivo *header* que deve ser incluído na implementação da *interface* *InterfClient_i* e em todos os clientes. Este arquivo define a visão C++ do IDL.
- **Cientes.ih:** uma versão inicial do arquivo *header* da classe de implementação.
- **Cientes.ic:** uma versão inicial do arquivo fonte da classe de implementação.

Figura 49 – Estrutura de Comunicação do Protótipo 3

A Figura 50 mostra o arquivo *header* *Cientes_i.h*, escrito em C++, a partir de *Cientes.ih*.

Já, a partir do arquivo *Servidor.idl*, que contém a definição da *interface* *InterfServer*, o compilador IDL produziu os seguintes arquivos C++:

- **Servidor.hh:** um arquivo *header* que deve ser incluído na implementação da *interface* *InterfServer_i* e em todos os clientes.
- **ServidorS.cpp:** um arquivo fonte que deve ser compilado e incluído no programa fonte do servidor (GFRV).

- **Servidor.ih:** uma versão inicial do arquivo *header* da classe de implementação.
- **Servidor.ic:** uma versão inicial do arquivo fonte da classe de implementação.

```
//Arquivo Clientes_i.h
#include "clientes.hh"
#include "wt.h"
#define MESSAGES 10 /*No. de mensagens na tomada do tempo de
                    resposta*/
#define NETNODES 8 /*Numero maximo de nos na rede */
#define THEMSIZE 1.0f /*Fator de Escala para os objetos
                    participantes */
unsigned short netid = 0; /* Id deste nó */

TableObj Table_Obj; /* Tabela dos arquivos .nff dos objetos */
WtObject *nodeobj[NETNODES]; /* Vetor de todos os objetos da
                    simulacao */
char *nodeformat[NETNODES]; /* Vetor dos arquivos .nff dos nós */
void quit_net(void); /* Funcao que sai do loop de simulacao */
void TimeAck(long Time, long Message, long IdReq);
float TableTime[NETNODES][MESSAGES]; /*Vetor dos tempos das
                    mensagens */
int TimeFlag[NETNODES]; /*Vetor que verifica quais obj. já
                    enviaram */
void CheckTableTime(void); /* Verifica tempo de resposta */

/* Vetor com a posicao dos carros (até 8) */
Wtpq pos_carro[NETNODES]=
{-694.8694f, -12.0224f, -82.1209f, 0.0f, 0.0f, 0.0f, 1.00f,
-496.9193f, -12.0224f, 335.6938f, 0.0f, 0.7139f, 0.0f, 0.7003f,
673.0453f, -12.0224f, 527.3203f, 0.0f, 1.0f, 0.0f, 0.00f,
-736.9567f, -12.0224f, 552.5996f, 0.0f, 1.0f, 0.0f, 0.00f,
-694.8694f, -12.0224f, -82.1209f, 0.0f, 0.0f, 0.0f, 1.00f,
-496.9193f, -12.0224f, 335.6938f, 0.0f, 0.7139f, 0.0f, 0.7003f,
673.0453f, -12.0224f, 527.3203f, 0.0f, 1.0f, 0.0f, 0.00f,
-736.9567f, -12.0224f, 552.5996f, 0.0f, 1.0f, 0.0f, 0.00f};

char *nodenames[NETNODES]; /* Vetor dos arquivos .nff dos nos */
WtNode *carro[NETNODES]; /* Vetor de todos os avatares da
                    simulacao */

WtNode *root;
class InterfClient_i: public virtual InterfClientBOAImpl {
public:
    InterfClient_i (const char* Name) : InterfClientBOAImpl
        (Name) {}
    virtual void NetAlive (CORBA::Long Id, const char *Obj,
        CORBA::Environment&) ;
    virtual void NetBye (CORBA::Long Id, CORBA::Environment&) ;
    virtual void NetViewPos (CORBA::Long Id, const WT_pq& Pos,
        CORBA::Environment&) ;
    virtual void NetTimeReq(CORBA::Long Time,CORBA::Long
        Message,CORBA::Long IdReq, CORBA::Environment&);
    virtual void NetTimeAck(CORBA::Long Time,CORBA::Long
        Message,CORBA::Long IdReq, CORBA::Long IdAck,
        CORBA::Environment&);
};
```

Figura 50 – Arquivo *header* Clientes_i.h

A Figura 51 mostra o arquivo *header* Servidor_i.h, escrito em C++, a partir de Servidor.ih.

```
//Arquivo Servidor_i.h
#include "servidor.hh"

#define NETNODES 8      /* Número máximo de nós na rede */
InterfClient_var ServidorTalker; /* cria um proxy para usar
                                operações da interface do cliente
                                (para usar NetAlive e NetBye
                                (como um talker)*/

class InterfServer_i: public virtual InterfServerBOAImpl {
TableObj Table_Obj; /* Armazena nome de arquivo .nff dos nós */
char *object;
void InicializaTabelas();
public:
    InterfServer_i();
    virtual TableObj NetOpen (CORBA::Long Id, const File Obj,
        CORBA::Environment&) ;
    virtual void NetClose (CORBA::Long Id, CORBA::Environment&) ;
};
```

Figura 51 – Arquivo *header* Servidor_i.h

As implementações das *interfaces* InterfClient e InterfServer, consistem das classes C++: InterfClient_i e InterfServer_i, cujas operações foram descritas no item 6.5.1.

6.5.3 A Aplicação Cliente do Protótipo 3

A aplicação cliente faz invocações aos métodos das duas *interfaces* implementadas.

As Figuras 52 a 54 apresentam os algoritmos das partes principais da aplicação cliente, com ênfase nos aspectos de integração com o OrbixTalk e ORBIX.

Principais variáveis e constantes globais

/* Exceto as estruturas de dados explicadas anteriormente */

Constante inteira: NETNODES=8;

Variáveis Inteiras: netid=0, tval;

...

/* Iniciação do OrbixTalk */

OrbixTalk* otalk;

InterfClient_ptr myRegisterTalker (const char* cidtk)

```
{
  CORBA(Object_ptr) obj = otalk->registerTalker(cidtk,"InterfClient");
  return InterfClient::_narrow(obj);
}
```

/* Declaração dos objetos CORBA */

InterfServer_var ServidorIO;

InterfClient_var ClientTalk;

InterfClient_ptr ClientListen;

...

Programa Principal

- Obter o nome do arquivo com o modelo estacionário e o nome do host que contém o GFRV, a partir da linha de comando;
- Criar o universo estático;
- Criar a janela inicial;
- Criar o nó *Root*;
- Associar um ponto de vista ao universo;
- Carregar o modelo de background (estacionário);
- Carregar luzes padrão;
- Abrir as entradas pelo teclado;
- Usar o mouse como sensor;
- Associar a função **actions()**, como função de ação do universo;
- Adequar a velocidade do sensor às dimensões do universo;
- Acertar o ponto de vista inicial (posição e orientação) do usuário;
- Associar o sensor (mouse) ao ponto de vista;
- Iniciar a comunicação via OrbixTalk e ORBIX, chamando a função **setup_net(host)** e passando o nome do host como parâmetro;
- Preparar a aplicação para a entrada no Loop de Simulação;
- Entrar no Loop de Simulação, chamando a função **WTuniverse_go()**;
- Sinalizar ao GFRV que este usuário está se desconectando, chamando **ServidorIO->NetClose(netid)**;
- Excluir todos os objetos do universo.

Figura 52 – Algoritmo do Programa Principal do Cliente

Função actions()

Variáveis Locais

Inteira: j;

Obter a posição e orientação do avatar local;

Enviar esta informação aos demais clientes chamando:

ClientTalk->NetViewPos(netid,pos);

Entrar no Loop de Eventos do OrbixTalk: CORBA(Orbix).processEvents(50);

Ler o teclado;

Caso a tecla seja:

'i': imprime o número de polígonos envolvidos na cena e o
número de frames por segundo;

Finaliza comando Caso;

'w': /* comuta o modo de renderização */

Se (renderiza==1)

então

renderiza=0;

Mudar exibição para renderizado (cena exibida com texturas e
iluminação);

senão

renderiza=1;

Mudar exibição para wireframe (cena exibida sem texturas e iluminação);

Fim-Se;

Finaliza comando Caso;

'q': /* finaliza a aplicação */

Chamar a função WTuniverse_stop(), para sair do Loop de Simulação;

Finaliza comando Caso;

'r': /* pede tempo de resposta */

j=0;

Enquanto (j<MESSAGES) faça:

Obter o tempo de resposta invocando a função

ClientTalk->NetTimeReq(clock(),j, netid);

Incrementar j;

Fim-Enquanto;

Finaliza comando Caso;

/* outras funções */

...

Fim-Caso;

Retornar;

Figura 53 – Algoritmo da Função actions()

Função setup_net(host)

Variáveis Locais

Inteiras: i, erro=1;

- Iniciar a comunicação via OrbixTalk:

```

try
{
    otalk=OrbixTalk::initialise();/*tenta iniciar o objeto gerente do OrbixTalk
*/
    ClientTalk = myRegisterTalker("otrmpp//posicoes"); /*registra o cliente
como talker*/
    ClientListen = new InterfClient_i("otrmpp//posicoes");
    otalk->registerListener(ClientListen); /*registra o cliente como listener */
    CORBA(Orbix).processEvents(50); /* Entra no Loop de eventos do */
                                   /* OrbixTalk */
}
catch (CORBA(SystemException) &sysEx)
{
    cerr << "Excecao Inesperada do sistema, saindo." << endl;
    cerr << &sysEx;
    Encerrar a aplicação;
}
catch(...)
{
    cerr << "Excecao Inesperada " << endl;
    exit(1);
}

```

- Iniciar a ligação entre o objeto e o GFRV, onde o objeto servidor está realmente executando:

```

try{
    ServidorIO = InterfServer::_bind(":CidadeOT",host);
} catch(CORBA::SystemException& se){
    WTmessage("\nLigacao (Bind) com objeto InterfServer falhou\n");
    Encerrar a aplicação;
}

```

Repetir: /* Consegue uma identificação (Id) */

```

try{
    erro=0;
    tval = identificação gerada;
    netid = (tval % NETNODES );
    Table_Obj=ServidorIO->NetOpen(netid, object);
} catch(VirCity::IdRejected& r){ erro=1;
}

```

até (erro==0);

- Inserir um avatar (automóvel) no grafo de cena, para representar o cliente local;
- Percorrer a tabela de objetos recebida (Table_Obj) e carregar a geometria de avatar (automóvel) para cada cliente ativo diferente do cliente local (Table_Obj.Flag[i] && i <> netid), para o grafo de cena;
- Retornar;

Figura 54 – Algoritmo da Função setup_net()

Capítulo 7 – Comparação dos Protótipos e Análise dos Resultados Obtidos

Neste capítulo é apresentada uma análise dos protótipos descritos no capítulo anterior. Esta análise têm como objetivo demonstrar a viabilidade da abordagem proposta nesta tese e dar uma idéia do desempenho dos protótipos.

A análise realizada abrange os seguintes aspectos:

- a obtenção da latência de comunicação exibida pelos protótipos, através da mensuração dos tempos de resposta dos sistemas;
- a identificação das tendências quanto à escalabilidade dos protótipos; e
- uma estimativa do desempenho exibido pelas aplicações clientes, com relação a variação da complexidade do mundo virtual gerado.

7.1 Ambiente Experimental

Como ambiente para os experimentos, foi criada uma rede local Ethernet dedicada, composta de microcomputadores compatíveis com IBM PC.

De forma a reduzir o número de variáveis independentes potencialmente não gerenciáveis, seguiu-se o seguinte procedimento para os experimentos:

- Todos os computadores utilizados possuíam idêntica configuração de *hardware* (64MBytes de memória principal, processador Pentium MMX 233MHz e placa aceleradora 3D) e sistema operacional (*Windows* NT, versão 4.0., da Microsoft).
- Todas as aplicações clientes possuíam cópias idênticas do mundo virtual.

- A aplicação servidora (GFRV – Gerente de Facilidades de Realidade Virtual), necessária para os protótipos 2 e 3, executou em computador independente das aplicações clientes.
- Todos os programas não essenciais foram desativados durante os experimentos.

7.2 Latência da Comunicação

Como descrito no capítulo anterior, todos os protótipos implementados possuem a facilidade de retornar, quando solicitado pelo usuário, o tempo requerido pela rede para transferir uma mensagem de um processo a outro (e, no caso, receber a correspondente confirmação), ou seja, a sua latência.

Porém, devido a diferença entre os modelos de comunicação usados nos protótipos, cabe ressaltar a forma como esta latência foi obtida:

- **Protótipo 1 (WTK):** nesta implementação, o modelo de comunicação utilizado foi o ponto a ponto, com uso de *multicast*. A obtenção da latência é iniciada pelo envio, a partir de uma aplicação cliente, de uma mensagem de solicitação de tempo para todas as demais aplicações participantes do sistema, via *multicast*. As aplicações receptoras, ao receberem a solicitação, retornam à aplicação emissora, uma mensagem de confirmação. O tempo é, portanto, estimado pela aplicação emissora, após o recebimento das confirmações de todas as aplicações participantes.
- **Protótipo 2 (ORBIX):** nesta implementação, o modelo de comunicação usado é o do cliente/servidor. Portanto, toda comunicação entre clientes é feita por intermédio do servidor (Gerente de Facilidade de Realidade Virtual – GFRV). Neste sistema, a obtenção da latência é iniciada pelo envio, a partir de uma aplicação cliente, de uma mensagem de solicitação de tempo para o GFRV. Este, por sua vez, se encarrega de reenviar esta solicitação às demais aplicações clientes. Ao receberem esta solicitação, as aplicações clientes enviam uma confirmação ao GFRV. Após ter recebido as confirmações de todas as demais aplicações, o GFRV envia uma mensagem de confirmação à aplicação cliente que solicitou o tempo.

- **Protótipo 3 (OrbixTalk):** este sistema combina o uso do modelo cliente/servidor com a comunicação *multicast*. Como neste sistema os clientes podem se comunicar diretamente, com uso do protocolo *multicast*, o esquema usado para a estimativa da latência é similar ao do protótipo 1.

7.2.1 Análise dos Resultados Obtidos

Para a realização desta análise, as mensagens de solicitação de tempo foram enviadas 10, 100 e 1000 vezes para os nós (computadores) participantes do sistema. Estas estimativas foram realizadas para 2 e 3 nós-clientes e 1 nó-servidor (o nó servidor, naturalmente, só foi utilizado nos protótipos 2 e 3).

Esta análise foi realizada com o mesmo mundo virtual (Cidade Virtual) para todos os protótipos. A quantidade de polígonos existentes no mundo virtual foi a máxima utilizada para os experimentos (5430 polígonos).

As Figuras 55, 56 e 57 ilustram os resultados obtidos.

Figura 55 – Latência para 10 Mensagens

Figura 56 – Latência para 100 Mensagens

Figura 57 – Latência para 1000 Mensagens

Como pode ser observado, a menor latência foi obtida pelo protótipo 1, tendo-se em segundo lugar o protótipo 3 e, por último, o protótipo 2. Estes resultados já eram esperados, uma vez que:

- O Protótipo 1, por utilizar uma comunicação direta entre clientes, não sofre o atraso introduzido pelo uso do servidor, além de utilizar primitivas de mais baixo nível (em relação aos demais protótipos) para a troca de mensagens, que é realizada através dos protocolos UDP e IP *multicast*;

- o Protótipo 2, por utilizar o servidor como elemento intermediário na comunicação entre clientes e não utilizar o IP *multicast*, obteve a mais alta latência;
- o Protótipo 3, por utilizar a comunicação direta entre clientes, via um protocolo mais confiável (OTRMP, apresentado no item 6.1.2), também baseado no UDP e IP *multicast*, apresentou a segunda melhor latência.

7.3 Escalabilidade

Nos experimentos executados, a escalabilidade foi interpretada como sendo a capacidade do sistema em aceitar novos nós clientes.

Considerando-se os resultados discutidos no item anterior, pode-se ter uma idéia quanto à escalabilidade dos protótipos verificando-se a degradação do tempo de resposta do sistema com relação ao número de nós clientes.

Os resultados obtidos são apresentados na Figura 58.

Figura 58 – Estimativa da Escalabilidade dos Protótipos

Como mostra a Figura 58, o protótipo 3 apresentou as menores degradações do tempo de resposta (de 3,6% a 4,7%), e, portanto, a melhor escalabilidade,

considerando-se até 3 clientes. Isto, provavelmente, deve-se ao fato do protótipo 3 valer-se da comunicação desacoplada fornecida pelo modelo *Talker/Listener* (item 6.1.2) do OrbixTalk e não utilizar o servidor para a comunicação entre clientes.

O protótipo 1, o qual obteve uma degradação de 14,3% a 57,3%, apresentou, também, uma maior variação do tempo de resposta com relação ao número de mensagens trocadas. Apesar de utilizar os protocolos UDP e IP *multicast*, o protótipo 1 não possui a vantagem da comunicação desacoplada oferecido no protótipo 3.

O protótipo 2, por sua vez, apresentou degradações que variaram de 41,6% a 43,9%, provavelmente devido ao “gargalo” representado pelo servidor.

Pode-se notar que os protótipos 2 e 3, que utilizam a arquitetura CORBA, apresentaram uma menor variação entre os tempos de resposta obtidos, com relação ao número de mensagens trocadas entre clientes.

7.4 Análise da Geração de Imagens

Apesar de não ser um dos objetivos principais desta tese, é importante ter-se uma idéia do desempenho, quanto a apresentação das cenas geradas pelos vários protótipos.

Segundo Kirner (1998), grande parte da ênfase do projeto de sistemas de realidade virtual tem sido estabelecida pelas restrições de geração da cena visual. Entretanto, muitos dos assuntos envolvidos na modelagem e geração das características de audição e tato/força do sistema são semelhantes ao domínio visual. Os requisitos para interação, navegação e comunicação em um mundo virtual são comuns a várias modalidades sensoriais.

No caso de geração da cena visual, o sistema requer taxas altas de quadros (*frames*) por segundo e respostas rápidas, em função de sua natureza interativa. O conceito de quadros é proveniente da animação baseada em uma sucessão rápida de uma seqüência de fotografias, como acontece em um filme de cinema. A taxa ideal da troca de fotos é de 20 quadros por segundo ou mais, para manter a ilusão de movimento. As taxas de quadros por segundo podem ser examinadas do ponto de vista gráfico, computacional e de acesso aos dados, de forma independente. A taxa

mais crítica é a do ponto de vista gráfico, pois está ligada com a sensação de presença ou imersão, sendo o mínimo aceitável da ordem de 8 a 10 quadros por segundo. Nesse caso, a taxa mínima de quadros do ponto de vista computacional e de acesso a dados deverá também ser de 8 a 10 quadros por segundo. Para sustentar a taxa de quadros do ponto de vista gráfico, para aplicações com controle interativo, são necessários tempos de resposta bem rápidos.

Os atrasos também podem ser classificados do ponto de vista gráfico, computacional de acesso a dados e não devem ser inferiores a 0,1 segundo. Quanto maior a frequência de movimento de algum objeto da cena, maior deverá se manter a taxa de quadros e menor o atraso, para manter-se a sensação confortável de animação.

A taxa de quadros do ponto de vista gráfico depende de: complexidade gráfica, iluminação, sombreamento e textura. Do ponto de vista de acesso aos dados e computacional, a taxa de quadros está relacionada com a complexidade do ambiente e com a quantidade de objetos dinâmicos. Isto vai se refletir no tempo de acesso e no tempo de simulação, definindo os atrasos.

A maneira mais comum de criar-se imagens gráficas tridimensionais por computador baseia-se no uso dos polígonos. Foi estimado que uma cena realista detalhada requer aproximadamente 80 milhões de polígonos (Durlach; Mavor (1995)). Para mostrar este mundo virtual a 10 quadros por segundo, por exemplo, seria necessário uma taxa de 800 milhões de polígonos por segundo, o que está longe da capacidade tecnológica atual. Para se ter uma idéia, as taxas recentes de máquinas *Silicon Graphics*, com aceleradores gráficos avançados estão em torno de 40 milhões de polígonos por segundo. Já um microcomputador, com acelerador gráfico, pode apresentar um desempenho de cerca de 1 milhão de polígonos por segundo.

Considerando o exposto, para se ter uma idéia do desempenho dos protótipos desenvolvidos, quanto a geração de cenas visuais, criou-se várias modelagens para o mundo virtual, cada qual com um número diferente de polígonos. Como mencionado no capítulo anterior (item 6.2), os objetos gráficos tridimensionais foram organizados nas seguintes classes funcionais: Objetos Estáticos (OE) e Objetos Dinâmicos Dirigidos pelo Usuário (ODU). Os OE são fixos e caracterizam o contexto da aplicação como edifícios, ruas, avenidas, placas de sinalização, etc. Os ODU

representam as ações dos usuários sobre o ambiente virtual, no caso, os veículos. Os ODUs também são conhecidos como avatares.

A Tabela III apresenta as geometrias usadas na construção do mundo virtual, segundo seu número de polígonos.

Tabela III – Polígonos Existentes na Cidade Virtual

Objetos Estáticos (OE)	Número de Polígonos	A. Total de Polígonos na Cena B. (OE + ODU)		
		com 1 ODU	com 2 ODUs	com 3 ODUs
Superfície	157	302	447	592
Cidade 1	980	1125	1270	1415
Cidade 2	1670	1815	1960	2105
Cidade 3	2507	2652	2797	2942
Cidade 4	3675	3820	3965	4110
Cidade 5	4995	5140	5285	5430

As Figuras 59 e 60 mostram o desempenho dos protótipos, quanto ao número de quadros apresentados por segundo, para cenas renderizadas. A renderização corresponde à aplicação de texturas, sombreamento e iluminação aos objetos tridimensionais em uma cena.

Cabe ressaltar, ainda, que para os experimentos foi utilizada a resolução padrão de 1024x768 *pixels*, com janelas de 416x312 *pixels*.

Figura 59 – Desempenho dos Protótipos com 2 ODUs Renderizados

Figura 60 – Desempenho dos Protótipos com 3 ODU's Renderizados

Conforme pode ser observado nas Figuras 59 e 60, o melhor desempenho pertence ao protótipo 1, seguido pelo protótipo 2 e protótipo 3. Apesar de ter apresentado a maior taxa de quadros por segundo, nota-se que a mesma degrada-se rapidamente no protótipo 1, a medida que aumenta o número de polígonos na cena. Esta situação já era esperada, conforme exposto no início deste item. A taxa inferior de quadros apresentada pelos protótipos 2 e 3 também era esperada, uma vez que ambos, além de utilizarem o próprio WorldToolkit (usado no protótipo 1) como ferramenta para confecção do ambiente de realidade virtual, também executam sobre a arquitetura CORBA. O protótipo 2 apresentou o segundo melhor desempenho porque utiliza apenas as APIs de biblioteca do ORBIX, na máquina da aplicação cliente. O protótipo 3, além de utilizar estas APIs, utiliza, também, os serviços oferecidos pelo OrbixTalk, o que sobrecarrega a máquina da aplicação cliente.

Todos os protótipos possuem a opção de inibição da renderização, podendo-se gerar cenas com objetos representados apenas por suas estruturas em formato de malhas (formato *wireframe*).

Na Figura 61, são apresentados os gráficos com o desempenho dos protótipos, em termos da taxa de quadros por segundo e do número de ODU's, para cenas renderizadas e não renderizadas (*wireframe*).

Figura 61 – Desempenho dos Protótipos: Renderização x Wireframe

Considerando-se os resultados mostrados na Figura 61, nota-se que os três protótipos, para a quantidade máxima de polígonos utilizada nos experimentos, apresentam uma taxa aceitável de apresentação (acima de 8 quadros por segundo), sendo que o mais baixo desempenho é o do protótipo 3, pelos motivos discutidos anteriormente.

Um detalhe interessante é que o desempenho dos três protótipos, pouco melhorou quando passou-se a inibir a renderização, ou seja, quando se utilizou a exibição *wireframe*. Isto, provavelmente, deve-se ao fato de ter se utilizado placas aceleradoras de vídeo, pois em alguns experimentos executados sem a placa aceleradora, a inibição da renderização resultou em um aumento sensível da taxa de quadros por segundo. Quando não se utiliza uma placa aceleradora, a renderização das cenas provoca uma grande sobrecarga no processador e na memória principal do computador.

Capítulo 8 – Conclusões

O capítulo final desta tese destaca, inicialmente, alguns aspectos gerais importantes do trabalho, tais como as diretrizes utilizadas em seu desenvolvimento e os benefícios obtidos no seu cumprimento. Em seguida, os protótipos implementados são classificados usando-se a metodologia apresentada no início deste trabalho. Finalmente, as contribuições do trabalho também são enfatizadas, bem como as pesquisas futuras que poderão ser executadas, e que darão continuidade ao mesmo.

8.1 Aspectos Gerais do Trabalho

Este trabalho baseou-se na seguinte premissa maior: a que a arquitetura CORBA possui as facilidades adequadas para o suporte de comunicação e sincronização dos componentes de um ambiente virtual distribuído. Para a comprovação desta proposição, efetuou-se um estudo desta plataforma e chegou-se a uma metodologia para o desenvolvimento de AVDs CORBA, a qual foi descrita e discutida no capítulo 6. Para a verificação da eficácia desta metodologia criou-se, então, três implementações que procuraram evidenciar, de maneira gradual, os aspectos de integração entre a aplicação distribuída de realidade virtual e a arquitetura de suporte alvo.

Alguns testes foram realizados no intuito de identificar-se o comportamento e o desempenho dos protótipos. Estes testes revelaram que a arquitetura CORBA, em especial com a utilização de seu Serviço de Eventos, pode fornecer um desempenho

aceitável para as aplicações distribuídas, acrescentando ainda os seguintes benefícios:

- Fornece uma base de alto nível para a colaboração dos objetos distribuídos;
- Fornece uma abordagem orientada aos objetos que é adequada aos AVDs, pois permite herança entre *interfaces*, encapsulamento, polimorfismo e manipulação de exceções;
- Separa a definição dos serviços da implementação dos mesmos, inclusive da linguagem de programação utilizada; e
- Simplifica a integração de aplicações distribuídas de realidade virtual, pelos aspectos citados anteriormente.

Para um melhor enquadramento dos AVDs implementados, é conveniente classificá-los segundo a metodologia usada no capítulo 3, o que é mostrado na Tabela IV.

Tabela IV – Classificação dos Protótipos Implementados

CARACTERÍSTICAS		Protótipo 1	Protótipo 2	Protótipo 3
COMUNICAÇÃO	Mecanismo(s) de transporte	<i>Multicast</i>	Ponto a ponto	<i>Multicast e ponto a ponto</i>
	Largura de Banda atingida	10Mbps+	10Mbps+	10Mbps+
	Estrutura(s)	<i>Peer</i>	Cliente/Servidor	Híbrida: <i>Listener/Talker</i> Cliente/Servidor
GERENCIAMENTO DE DADOS	Organização	BD visual com replicação total	BD visual com replicação total	BD visual com replicação total
GERENCIAMENTO DA COMPUTAÇÃO	Organização	Distribuição total	Distribuição total	Distribuição total
MODELAGEM DO AV	Permite múltiplos Mundos Virtuais	Sim	Sim	Sim
	Suporte ao Usuário	Múltiplos	Múltiplos	Múltiplos
AMBIENTE DE REDE	Tipo utilizado	Heterogêneo	Heterogêneo	Heterogêneo
BALANCEAMENTO DE CARGA	Existência	Não existe	Não existe	Não existe

Segundo esta classificação, no geral, os protótipos possuem características semelhantes, com exceção dos mecanismos de transporte e estruturas utilizadas na

comunicação, uma vez que os protótipos 2 e 3 utilizam a arquitetura CORBA e o protótipo 1, não.

Quanto à abrangência e aplicabilidade dos protótipos, especialmente o mais elaborado deles, o protótipo 3, nota-se que são especialmente adequados aos AVDs de pequeno e médio porte. No entanto, com os contínuos aperfeiçoamentos sofridos pelas implementações do CORBA, não se excetua sua utilização na confecção de AVDs de grande porte, em um futuro próximo.

8.2 Contribuições do Trabalho

A contribuição principal refere-se ao estudo, elaboração de uma metodologia de desenvolvimento, e implementação de ambientes virtuais distribuídos baseados na utilização da arquitetura CORBA como camada *middleware* de comunicação.

No levantamento efetuado no decorrer deste trabalho, verificou-se que, até o momento da elaboração desta tese, nenhum AVD atual utiliza a arquitetura CORBA propriamente dita. O sistema conhecido como BrickNet (Singh et al.(1994), Singh et al. (1995) e Macedonia et al. (1995)) foi o único que utilizou algum tipo de ORB para a comunicação, mas sem qualquer alusão à arquitetura CORBA.

Uma outra contribuição é a proposta de uma camada de facilidades CORBA para aplicações RV distribuídas (representada pelo GFRV), visto que, atualmente, não existe nenhuma padronização neste sentido.

Os próprios protótipos implementados consistem em uma contribuição, por serem totalmente funcionais, podendo, efetivamente ser utilizados como ferramenta de ensino, treinamento e pesquisa.

Durante o andamento deste trabalho, com o objetivo de divulgar no meio científico a proposta e as soluções encontradas, foram elaborados diversos artigos, nos quais participaram, além do autor desta tese, membros do Grupo de Realidade Virtual da Universidade Federal de São Carlos. Estes artigos são: Sementille et al. (1999), Brega et al. (1999 a), Kubo et al. (1999), Deriggi Jr. et al. (1999 a), Brega et al. (1999 b), Deriggi Jr. et al. (1999 b), Kubo et al. (1998), Deriggi Jr. et al. (1998), Deriggi Jr. et al. (1997).

8.3 Pesquisas Futuras

A seguir são listadas as pesquisas que poderão dar continuidade a este trabalho. Estas pesquisas referem-se tanto aos aperfeiçoamentos ao nível da utilização da arquitetura CORBA, quanto aos melhoramentos que poderão ser feitos ao nível da própria aplicação de realidade virtual distribuída.

8.3.1 Aperfeiçoamentos ao Nível da Utilização da Arquitetura CORBA

- Utilizar os mecanismos de invocação dinâmica da arquitetura CORBA para fornecer serviços em tempo de execução, aos clientes do AVD.
- Realizar experimentos com uma quantidade maior de clientes.
- Realizar experimentos em redes a longa distância, principalmente a Internet.
- Utilizar ORBs diferentes, tais como o OrbixWeb, da IONA, a fim de criar-se AVDs com objetos que se beneficiem da interoperabilidade entre plataformas. Tal interoperabilidade pode ser conseguida através da utilização do protocolo IIOP, mencionado no capítulo 4.
- Utilizar outros serviços CORBA, além do Serviço de Eventos, tais como o Serviço de Nomeação e de Tempo, para melhorar o desempenho do AVD.
- Analisar os recursos *multithreads* oferecidos pelas implementações CORBA mais recentes, e verificar seu impacto no aperfeiçoamento dos ORBs e servidores.

8.3.2 Aperfeiçoamentos ao Nível da Aplicação de Realidade Virtual Distribuída

- Incorporar o mecanismo de *dead-reckoning*, apresentado no capítulo 2, para melhorar o desempenho do sistema, a partir da minimização da latência da comunicação.
- Incorporar mecanismos para a detecção e o tratamento de colisões entre os objetos do mundo virtual.

- Ampliar o espaço gerado na aplicação da Cidade Virtual, incorporando mecanismos para o gerenciamento de objetos em espaços amplos, tais como a introdução de Gerentes de Área e particionamento da base de dados.
- Transformar a aplicação de não imersiva para imersiva, incorporando novos sensores para a interação com o usuário, tais como capacetes de visualização (HMD - *Head Mounted Display*) e luvas (*data gloves*).

Referências Bibliográficas

- ARAÚJO, R. B. *Especificação e Análise de um Sistema Distribuído de Realidade Virtual*. São Paulo, 1996. 149p. Tese (Doutorado) - Escola Politécnica da Universidade de São Paulo.
- AZUMA, R. A Survey of Augmented Reality. *Presence: Teleoperators and Virtual Environments*, n. 6, p355-385, August 1997.
- AZUMA, R. Tracking Requiriments for Augmented Reality. *Communications of ACM*, v. 36, n.7, p50-51, July 1993.
- BARRUS, J.W.; WATERS, R. C.; ANDERSON, D. B. Locales: Supporting for Large Multiuser Virtual Environment. *IEEE Computer Graphics and Applications*, v.16, n.6, p50-57, November 1996.
- BENDFORD, S.; BOWERS, J.; FAHLÉN, L. E.; GREENHALGH, C. Networked Virtual Reality and Cooperative Work. *Presence*, v. 4, n. 4, p364-386, 1995.
- BETZ, M. Interoperable Object. *Dr. Dobb's Journal*, p18-39, October 1994.
- BLAU, B.; HUGHES, C. E.; MOSHELL, J. M.; LISLE, C., Networked Virtual Environments. *Proceedings of 1992 Symposium on Interactive 3D Graphics*, Cambridge, Mass. 29 March-1 April, 1992, Special Issue of Computer Graphics - ACM.
- BOMAN, D.K. International Survey: Virtual Environment Research. *IEEE Computer*, v.28, n.6, p57-65, June 1995.
- BREGA, J.R.F.; SEMENTILLE, A.C.; KIRNER, C. Utilizing the CORBA Platform in the Construction of a Highly Immersive Virtual Environment. *AeroSense '99*, Orlando, 5-9/April 1999 b.
- BREGA, J.R.F.; SEMENTILLE, A.C.; SÓRIA, M.H.A.; KIRNER, C. O Emprego da Realidade Virtual como Ferramenta para Simulação de Problemas de Engenharia de Tráfego. *II Workshop de Realidade Virtual – WRV'99*. Marília, São Paulo, novembro, 1999 a. /Resumo/
- BRICKEN, W.; COCO, G. The VEOS Project. *Presence*, v. 3, n. 2, p111-129, 1994.
- CARLSSON, C.; HAGSAND, O. DIVE – A Multi-User Virtual Reality System. *Procedings of the IEEE VRAIS'93 Conference*, p.394-400, 1993.

- CATER, J.P.; HUFFMAN, S. D. Use of the Remote Access Virtual Environment (RAVEN) for Coordinated IVA-EVA Astronaut Training and Evaluation. *Presence*, v.4,n.1, p103-109, Winter 1995.
- COMER, D. *Internetworking with TCP/IP – Principles, Protocols and Architecture*. Prentice-Hall, Inc., New Jersey, 1988.
- COTTMAN, B.; COTTMAN, B. *The Future of Database Access With Netscape ONE, CORBA, and JDBC*. Documento on line. http://developer.netscape.com/viewsource/cottman_JDBC.html, 1998.
- CRUZ-NEIRA, C.; LEIGH, J.; PAPKA, M.; BARNES, C.; COHEN, S. M.; DAS, S.; ENGELMANN, R.; HUDSON, R.; ROY, T.; SIEGEL, L.; VASILAKIS, C.; DEFANTI, T. A.; SANDIN, D. J. Scientists in Wonderland: A Report on Visualization Applications in the CAVE Virtual Reality Environment. *IEEE 1993 Symposium on Research Frontiers in Virtual Reality*, San Jose California, p.59-66, October 25-26 1993.
- DEERING, S. Mbone- The Multicast Bone. *CERFnet Seminar*, March 1993.
- DERIGGI JR, F.; SEMENTILLE, A.C.; KUBO, M.M.; KIRNER, C. A Utilização da Plataforma CORBA como Suporte para Aplicações Distribuídas de Realidade Virtual. *Anais do I Workshop de Realidade Virtual - WRV'97*, UFSCar, São Carlos/SP, p01-10, Novembro, 1997.
- DERIGGI JR., F.; KUBO, M. M.; SEMENTILLE, A. C.; SANTOS, S. G.; KIRNER, C. *Distributed Interoperable Virtual Environments*. *CLEI Electronic Journal*, Chile, Issue of December 1999 a.
- DERIGGI JR., F.; KUBO, M.M.; SEMENTILLE, A.C.; SANTOS, S. G.; BREGA, J.R.F.; KIRNER, C. CORBA Platform as Support for Distributed Virtual Environments". *Proceedings of the IEEE Virtual Reality*, Houston, Texas, 13 - 17 March 1999 b.
- DERIGGI JR., F.; KUBO, M.M.; SEMENTILLE, A.C.; SANTOS, S. G.; KIRNER, C. Ambientes Virtuais Distribuídos Interoperáveis. *Anais da XXIV Conferência Latino-americana de Informática (CLEI'98)*, Quito, Equador, Outubro, 1998.
- DREYFUS, P. CORBA: *Theory and Practice*. Documento on-line. http://developer.netscape.com/viewsource/dreyfus_corba.html, 1998.
- DURLACH, N.I.; MAVOR, A.S. *Virtual Reality: Scientific and Technological Challenges*. National Academy Press, Washington, DC, 1995.
- HARTMAN, J.; WERNECKE, J. *The VRML 2.0 Handbook: Building Moving Worlds on Web*. Addison-Wesley, Reading, Massachussets, 1996.

- HARVEY, E., SCHAFFER, R., MCGARRY, S. High Performance Fixed-Wing Aircraft Simulation using SIMNET Protocols. *Proceedings of the 1991 SCS Conference*, 23rd Annual Summer Computer Simulation Conference, p965-970, 1991.
- HAWKES, R. *A Software Architecture for Modeling and Distributing Virtual Environments*. Edinburg, 1996. Tese (Doutorado) - University of Edinburg.
- HAWKES, R. Virtual Environment Laboratory, University of Edinburg. *Proceedings of Virtual Reality Systems Fall '93*, New York, October 19-21, 1993.
- INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS. *IEEE Standard for Information Technology – Protocols for Distributed Simulation Applications: Entity Information and Interaction*, IEEE Standard 1278 – 1993, IEEE Computer Society, New York, 1993.
- IONA TECHNOLOGIES LTD. OrbixTalk Programming Guide, 1997 a.
- IONA TECHNOLOGIES LTD. Orbix 2.2 Programing Guide, 1997 b.
- JANSSEN, B.; SPREITZER, M. *ILU 2.0Alpha8 Reference Manual*. XEROX Corporation. Documento on-line. ftp://ftp.parc.xerox.com/pub/ilu/2.0/20a8-manual-html/manual_toc.html, 1996.
- KIRNER, C. *Sistemas de Realidade Virtual*. Documento on-line, <http://www.dc.ufscar.br/~grv/>, 1998.
- KIRNER, C.; BATTAIOLA, A.L.; NETO, A.T.; COPSTEIN, B.; SILVA, J.A.; BUGATTI, I.G.; PINHO, M.S.; CRUVINEL, P.E.; WAGNER, P.R.; ARAÚJO, R.G.; KIRNER, T. *Projeto AVVIC*. Documento on_line. <http://www.dc.ufscar.br/~grv/avvic.htm>, 1995 a.
- KIRNER, C.; ARAÚJO, R.B.; BUGATTI, I.G.; SEMENTILLE, A.C.; DERIGGI JR., F.V.; KUBO, M.M.; SANTOS, S.G.; ENCARNÇÃO, J.L.; HILDEBRAND, A.; SIERING, F. *Virtual Environments for Shared Interactive Visualization*. Paper accepted for Presentation at the Workshop of the German-Brazilian Cooperative Programme in Informatics, Berlim, 14-15 December 1995 b.
- KIRNER, C.; MENDES, S.B.T. *Sistemas Operacionais Distribuídos – Aspectos Gerais e Análise de sua Estrutura*. Editora Campus, 184p, Rio de Janeiro, 1988.
- KUBO, M.M.; DERIGGI JR., F.; SEMENTILLE, A.C.; BREGA, J.R.F.; MARQUES, D.; KIRNER, C. Framework for CSCW Applications by Using Virtual Reality. *Anais do IEEE ICCIMA '99*, Nova Delhi, India, 1999 (aceito para apresentação e publicação).
- KUBO, M.M.; DERIGGI JR., F.; SEMENTILLE, A.C.; KIRNER, C. Suporte de Rede para as Aplicações de Realidade Virtual destinadas a Educação à Distância.

Simpósio Brasileiro de Informática na Educação - Sbie '98, Fortaleza, Ceará, 17 a 19 de novembro, 1998.

LEAVENS, G. T.; CHEON, Y. *Extending CORBA to Specify Behavior with Larch*. Technical Report, Department of Computer Science, Iowa State University, August 1993.

LOCKE, J. Applying Virtual Reality. *IEEE Potentials*, October/November 1995.

MACEDONIA, M. R.; ZYDA, M. J. A Taxonomy for Networked Virtual Environments. *IEEE multimedia*, 1995.

MACEDONIA, M. R.; ZYDA, M.J.; PRATT, D.R.; BRUTZMAN, D.P.; BARHAM, P.T. Exploiting Reality with Multicast Groups: A Network Architecture for Large-scale Virtual Environment. *Proceedings of Virtual Reality Annual International Symposium '95*, March 11-15, 1995.

MACKEY, R. L. *NPSNET: Hierarchical Data Structures for Real-Time Three-Dimensional Visual Simulation*. Monterey, Califórnia, junho, 1991. Dissertação (Mestrado) - Naval Postgraduate School.

MAFFEIS, S.; SCHMIDT, D.C. Constructing Reliable Distributed Communication Systems with CORBA. *IEEE Communications Magazine*, v.14, n.2, February 1997.

MCCARTY, W., SHEASBY, S., STYTZ, M., SWITZER, C. A Virtual Cockpit for a Distributed Interactive Simulation. *IEEE Computer Graphics & Applications*, January 1994.

MORIE, J.F. Inspiring the Future: Merging Mass Communication, Art, Entertainment and Virtual Environments. *Computer Graphics*, v.28, n.2, p135-138, May 1994.

MOWBRAY, T. J.; ZAHAVI, R. *The Essential CORBA - Systems Integration Using Distributed Objects*. John Wiley & Sons, Inc., 316p, 1995.

NELSON, V. P. *Remote Procedure Call*. Dept. of Computer Science, Carnegie-Mellon University, May 1981. Tese (Doutorado).

NFS. Research Directions in Virtual Environments. (Report of an NSF Invitational Workshop, march 23-24, 1992), *Computer Graphics*, v. 26, n. 3, August 1992.

OBJECT MANAGEMENT GROUP, INC. *The Common Object Request Broker: Architecture and Specification*. OMG document n. 91.12.1, December 1991.

ORFALI, R.; HARKEY, D. Client/Server with Distributed Objects. *BYTE*, p151-162, April 1995.

- ROEHL, B. *Some Thoughts on Behaviour in VR Systems*. Documento on-line. <http://sune.uwaterloo.ca/~broehl/behav.html>, 1995.
- SCHMIDT, D.C. The ADAPTATIVE Communication Environment – An Object-Oriented Network Programming Toolkit for Developing Communication Software. *Proceedings of the 12th Annual Sun Users Group Conference*, SUN, San Francisco, June 16-17, 1994.
- SCHMIDT, D.C.; VINOSKI, S. Object Interconnections: Introduction to Distributed Object Computing, *C++ Report Magazine*, January 1995.
- SCHULZRINNE, H., CASNER, S.. *RTP: A Transport Protocol for Real-Time Applications*. Audio-Video Transport Working Group, Internet Engineering Task Force. Documento on-line. <http://ds.internic.net/internet-drafts/draft-ietf-avt-rtp-08.txt>, October, 1993.
- SEMENTILLE, A.C.; BREGA, J.R.F.; KIRNER, C. Uma Metodologia para o Desenvolvimento de Ambientes Virtuais Distribuídos usando CORBA. // *Workshop de Realidade Virtual – WRV'99*. Marília, São Paulo, novembro, 1999. /Resumo/
- SENSE8 CORPORATION. *WorldToolkit Reference Manual – Release 7*. Mill Valley, CA, 1998.
- SINGH, G., SERRA, L., PNG, W., NG, H. Bricknet: A Software Toolkit for Network-Based Virtual Worlds. *Presence*, v. 3, n. 1, p19-34, 1994.
- SINGH, G.; SERRA, L.; PNG, W.; WONG, A.; NG, H. Bricknet: Sharing Object Behaviors on the Net. *Proceedings of Virtual Reality Annual International Symposium '95*, March 11-15, 1995.
- SINGHAL, S. K.; ZYDA, M. *Networked Virtual Environment*. Documento on-line. <http://www.npsnet.nps.navy.mil/npsnet0/publications.html>, 1998.
- SNOWDON, D. N.; WEST, A. J. AVIARY: Design Issues for Future Large-Scale Virtual Environments. *Presence*, v. 3, n. 4, p228-308, 1994.
- TANEMBAUM, A.S. *Distributed Operating Systems*. Prentice Hall International Editions, 614p, New Jersey, 1995.
- VINOSKI, S. *CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments*. *IEEE Communications Magazine*, v.14, n. 2, February 1997.
- WANG, Q., GREEN, M.; SHAW, C. EM - An Environment Manager for Building Networked Virtual Environments. *Proceedings of Virtual Reality Annual International Symposium '95*, March 11-15, 1995.

WHETTEN, B.; KAPLAN, S. A High Performance Totally Ordered Multicast Protocol. *SIGCOMM'94*, 1994.

WLOKA, M. M. Lag in Multiprocessor VR. *Presence*, v.4,n.1, Spring 1995.

Glossário

ACK – (*Acknowledgement*) É uma resposta de um receptor para um emissor, usada para indicar que a transmissão dos dados foi completada com sucesso.

Adaptador – É um componente do ORB que fornece uma referência a um objeto, a ativação, e o estado relacionado aos serviços para uma implementação de objeto. Podem existir diversos adaptadores diferentes fornecidos para diferentes tipos de implementações.

Ambiente Virtual – É um ambiente tridimensional simulado, criado por aplicações de realidade virtual, no qual um usuário pode interagir.

Ambiente Virtual Distribuído – É um sistema através do qual diversos usuários interagem entre si e em tempo real, sendo que estes podem estar situados em localidades diferentes. Tipicamente, cada usuário acessa seu próprio computador, usando-o para fornecer uma *interface* para o ambiente virtual. Estes ambientes geralmente oferecem aos usuários uma sensação de realismo através da incorporação de gráficos 3D e som estéreo, a fim de criar uma experiência imersiva.

API – (*Application Programming Interface*) *Interface* de programação de aplicativos. É um conjunto de funções definidas para permitir o acesso a um sistema. Por exemplo, o sistema operacional *Windows* 98 possui centenas de APIs, contidas em bibliotecas de ligação dinâmica (DLLs), às quais um programa tem acesso quando é executado.

Assinatura – Define os parâmetros para uma dada operação, incluindo seu número de ordem, tipos de dados e modos de passagem de parâmetros; os resultados (se existirem); e os possíveis efeitos (normais ou excepcionais) que podem ocorrer.

Ativação – Preparar um objeto para executar uma operação. Por exemplo, copiando a forma persistente dos métodos e dados armazenados para um espaço de endereçamento executável, a fim de permitir a execução dos mesmos.

Atributo – É uma associação identificável entre um objeto e um valor.

BOOM - Dispositivo que suporta monitores, sensores e a óptica necessária, de forma suspensa, sem que o usuário tenha que vestir o dispositivo pesado.

Broadcast – É um sistema que entrega um dado pacote para todos os computadores (*hosts*) que estão conectados a ele. Pode ser implementado em *hardware* ou *software*.

CAVE – (*Cave Automatic Virtual Environment*) É um ambiente virtual alternativo, que utiliza telas de projeção para a visualização, em especial, de aplicações científicas.

Cena – É o mundo virtual que é exibido, em um ambiente virtual.

Classe – É uma entidade que congrega a definição de dados e das operações (funções ou métodos) que manipulam estes dados. Uma variável de uma classe é chamada objeto e conterá campos de dados e funções. A simples definição de uma classe não cria um objeto.

Cliente – É o código ou processo que invoca uma operação em um objeto.

Cliente/Servidor – É um modelo de interação em um sistema distribuído, no qual um programa (ou objeto) em uma localidade envia um pedido a um programa (ou objeto) em outro lugar, e espera por uma resposta. O programa requisitante é chamado cliente; o programa que satisfaz o pedido é chamado de servidor.

Comportamento – No contexto da orientação a objetos, corresponde aos efeitos observáveis de um objeto executando uma operação requisitada, incluindo seus resultados.

Conexão – É o caminho entre dois módulos de protocolo, o qual fornece um serviço de fluxo de dados confiável. Na Internet, uma conexão estende-se de um módulo TCP de um computador à um módulo TCP de outro.

CORBA – *Common Object Request Broker Architecture*. É uma arquitetura criada pela *Object Management Group* (OMG), a qual especifica como objetos distribuídos podem cooperar utilizando uma rede de computadores, sem se preocupar com os sistemas operacionais de clientes e servidores, nem com linguagens de programação. A arquitetura CORBA é uma plataforma completa para objetos distribuídos. Ela estende os limites das redes, linguagens, componentes e sistemas operacionais.

Criação de um objeto – Um evento que causa a existência de um objeto que é distinto de qualquer outro objeto.

Data glove – É uma luva dotada de sensores, a qual executa um sistema de reconhecimento de movimentação de dedos e posição da mão. O sistema traduz a atividade da mão e dos dedos em sinais eletrônicos, que o computador utiliza com objetivos de rastreamento de posição e controle de entrada de dados.

Datagrama IP – É a unidade básica de informação passada através da Internet. Um datagrama IP é para a Internet o que um pacote é para a rede física. Ele contém um endereço do emissor e do destino, bem como os dados.

DCE – (*Distributed Computing Environment*) É um ambiente criado pela *Open Software Foundation* (OSF), que pode servir como plataforma para execução de aplicações distribuídas. Este ambiente é construído no topo de um sistema operacional, tais como UNIX, *Windows* e VMS. A idéia é que um usuário pode tomar uma coleção de máquinas existentes, adicionar o *software* DCE, e então ser capaz de executar aplicações distribuídas, sem perturbar as aplicações (não distribuídas) existentes.

Dead-Reckoning – É uma técnica utilizada para minimizar a latência de comunicação em AVDs. Consiste em fazer com que um processo execute a completa simulação de um determinado objeto, processando um modelo de comportamento deste objeto, em paralelo, e quando os dois diferem de uma quantidade pré-definida, uma cópia das variáveis de estado reais daquele objeto é enviada para todos os processos-réplica. As subseqüentes aproximações são, então, baseadas na última atualização.

Destruição de um objeto – Um evento que força um objeto a cessar de existir.

DIS – (*Distributed Interactive Simulation*) É um protocolo utilizado em simulações interativas distribuídas. É o resultado do esforço de generalização do protocolo SIMNET, desenvolvido pelo Departamento de Defesa dos Estados Unidos (DoD). Foi padronizado através da especificação IEEE 1278.

DLL – (*Dynamic Link Libraries*) São bibliotecas de dados ou programas, que podem ser chamadas ou acessadas por qualquer aplicativo *Windows*. Um arquivo DLL pode usar a extensão dll ou exe.

Escalabilidade – A escalabilidade, para os Ambientes Virtuais Distribuídos, pode incluir diversas medidas. Comumente pode ser mensurada pelo número de entidades que podem, simultaneamente, participar do sistema. Pode, também, ser mensurada pelo número de computadores que podem se conectar simultaneamente a um AVD.

Estado – Representa a variação das propriedades de um objeto, no tempo. Esta variação afeta o comportamento do objeto.

Exceção – Uma condição causada por uma tentativa de executar uma operação inválida.

Exoesqueleto – Em sistemas de realidade virtual, são sensores mecânicos dispostos em uma espécie de “armadura”, utilizados no rastreamento da posição do usuário.

Extensibilidade – É a característica de uma arquitetura capaz de suportar utilizações imprevistas e se adaptar aos novos requisitos do desenvolvedor.

Extrusão – Estiramento ou deformação de um objeto em uma cena tridimensional.

Facilidades Comuns CORBA – São coleções de componentes escritos em IDL, as quais definem as regras de engajamento para os objetos de aplicação. Elas podem ser do tipo horizontal ou vertical.

Filtro ORBIX - Os filtros representam uma extensão do MT-ORBIX ao CORBA, a qual fornece um padrão para desacoplar a demultiplexação das requisições CORBA, do eventual despacho da chamada do método provido pela implementação do objeto alvo.

GFRV – (Gerente de Facilidades de Realidade Virtual) É o elemento criado nos protótipos implementados nesta tese, cuja função principal é fornecer serviços

comuns aos objetos distribuídos de realidade virtual, como por exemplo, permitir a conexão e desconexão de clientes ao ambiente virtual distribuído.

Grafo de Cena – É uma estrutura hierárquica de nós (tais como, geometria, luz, fog e informação posicional) representando os objetos em uma simulação. Um Universo pode conter mais do que um grafo de Cena.

Granularidade – É o tamanho das unidades de trabalho (por exemplo, instrução, programas, etc.) submetidas aos processadores de um sistema. Está ligada ao porte e quantidade de processadores de um computador. Pode ser classificada em granularidade grossa (unidades de trabalho grandes), média (unidades de trabalho médias) e fina (unidades de trabalho pequenas).

Haptic - São dispositivos responsáveis por fornecer a ilusão da exploração manual ou manipulação de objetos nos mundos virtuais. Estes dispositivos empregam o tato humano, cinemática e sistemas de motores para a interação com o ambiente virtual. Eles executam duas funções básicas: a medição das posições e forças da mão do usuário ou outras partes do corpo para controlar as entradas para o ambiente virtual; e manifestação das forças e posições e/ou suas distribuições temporais para o usuário.

Herança – É a construção de uma definição pela modificação incremental de outras definições.

Herança de Interface – É a construção de uma *interface* pela modificação incremental de outras *interfaces*. A linguagem IDL provém a herança de *interface*.

Herança múltipla – É a construção de uma definição pela modificação incremental de mais do que uma outra definição.

Herança Simples – É a construção de uma definição pela modificação incremental de uma definição.

HMD – (*Head Mounted Display*) Visor montado na cabeça, ou capacete de realidade virtual. Geralmente possui monitores estereoscópicos, oferecendo um sentimento de presença do usuário, no mundo virtual.

IDL – (*Interface Definition Language*) É uma linguagem que permite a definição de *interfaces*. A definição de *interface* especifica as operações que um objeto está preparado para executar, os parâmetros de entrada e saída que requerem, e qualquer exceção que possa ser gerada durante sua execução.

Implementação - Uma definição que provém a informação necessária para criar um objeto e permitir ao mesmo fornecer um conjunto de serviços. Uma implementação tipicamente inclui uma descrição da estrutura de dados usada para representar o estado associado com um objeto, bem como, as definições dos métodos que acessam aquela estrutura de dados.

Instância – Um objeto é uma instância de uma *interface* se fornece as assinaturas e as semânticas das operações especificadas para aquela *interface*. Um objeto é uma instância de uma implementação, se seu comportamento é fornecido por aquela implementação.

Interface – É uma lista de operações e atributos que um objeto fornece. Isto inclui as assinaturas das operações, e os tipos dos atributos. Uma definição de *interface* idealmente inclui sua semântica. Um objeto satisfaz uma *interface* se puder ser especificado como o objeto alvo em cada potencial pedido descrito pela *interface*.

Interoperabilidade – É a habilidade de trocar funcionalidade e dados interpretáveis entre duas entidades de *software*. Pode ser definida em termos de quatro requisitos: comunicação, geração de pedidos, formato de dados, e semântica. As entidades de *software* requerem um canal de comunicação com um protocolo de comunicação comum. Através deste canal as entidades necessitam ser capazes de formular e transmitir um pedido interpretável para funções ou dados. O resultado do pedido deve retornar para a entidade solicitante. O intercâmbio de dados também implica em um requisito para um formato de dados que possa ser analisado pelas entidades. As entidades devem entender o pedido e dados através de alguma forma de tradução semântica.

Invocação dinâmica – Construir e invocar um pedido cuja assinatura não é conhecida até o momento da execução.

Invocação Estática – É a construção de um pedido em tempo de compilação. Corresponde a chamar uma operação via um procedimento *stub*.

IP – (Internet Protocol) É um protocolo padrão que define o datagrama Internet como a unidade de informação passada através da Internet e que fornece a base para o serviço de entrega de pacotes, sem conexão da Internet.

Latência - É o tempo requerido pela rede para transferir um bit de dado de um ponto a outro. A latência aparece por muitas razões. Primeiro, a transmissão de dados é governada pelo atraso da velocidade da luz. Um dado, fundamentalmente, não pode ser transportado mais rápido do que a velocidade da luz. Segundo, atrasos são introduzidos pelos próprios computadores. Leva tempo para o dado atravessar o sistema operacional, o *hardware* de rede e, então, encontrar a rede; similarmente o dado deve atravessar os mesmos níveis, ao chegar ao computador destino. Terceiro, atrasos são introduzidos pela própria rede. Geralmente os *hosts* emissor e destino não estão ligados diretamente. Ao invés disto, o dado deve passar por diversas interseções de rede, ou roteadores.

Mapeamento de linguagem ou **Mapeamento** – Corresponde aos meios e as convenções através dos quais um programador, escrevendo em uma linguagem de programação específica, acessa as facilidades do ORB.

Método – É uma implementação de uma operação. É o código que pode ser executado para realizar um serviço requisitado. Os métodos associados com um objeto podem ser estruturados em um ou mais programas.

Middleware – Em um sistema distribuído constituído de uma hierarquia de níveis de abstração, corresponde as camadas intermediárias desta hierarquia, geralmente localizadas entre a camada de transporte e a camada de aplicação.

Modelagem – Criação de uma forma geométrica representando um objeto tridimensional.

Modelagem Hierárquica – Paradigma de modelagem tridimensional, que usa hierarquias de objetos para construir conjuntos e subconjuntos complexos.

Modo de passagem de parâmetro - Descreve a direção do fluxo de informação para um parâmetro de uma operação. Os modos de passagem de parâmetro da arquitetura CORBA são: *IN*, *OUT*, e *INOUT*.

Multicast – Uma técnica que permite que cópias de um pacote possa ser passado para um subconjunto selecionado de possíveis destinos.

Multithread – Sistemas ou programas que suportam mais que uma *thread* concorrente de execução, isto é, eles são capazes de executar simultaneamente múltiplas sequências de instruções. Cada sequência de instrução tem seu próprio fluxo de controle, que é independente de todos os outros.

NAK – (*Negative Acknowledgement*) É uma resposta de um receptor para um emissor, usada para indicar que a transmissão dos dados fracassou (isto é, que o dado foi corrompido por erros de transmissão).

Nome de uma operação – É um nome usado em um pedido, para identificar uma operação.

Objeto – É uma combinação do estado e de um conjunto de métodos que explicitamente incorporam uma abstração caracterizada pelo comportamento dos requisitos relevantes. Um objeto é uma instância de uma implementação e uma *interface*. Um objeto modela uma entidade do mundo real, e é implementada como uma entidade computacional que encapsula estado e operações (internamente implementadas como dados e métodos) e responde aos serviços requisitados.

Objeto de Implementação – Um objeto que serve como uma definição de implementação. Objetos de Implementação residem em um Repositório de Implementação, na arquitetura CORBA.

Objeto distribuído – É um componente de um programa distribuído. Um objeto distribuído possui *interfaces* bem definidas, através das quais podem se comunicar com outros computadores de um sistema distribuído.

Objeto Interface – É um objeto que serve para descrever uma *interface*. Objetos *interface* residem em um Repositório de *Interface*, na arquitetura CORBA.

Objeto Persistente – É um objeto capaz de sobreviver ao processo ou a *thread* que o criou. Um objeto persistente só cessa de existir após ser explicitamente destruído.

Objeto Servidor - É um objeto que fornece respostas a um pedido para um serviço. Um dado objeto pode ser um cliente para alguns pedidos e servidor para outros.

Objeto Transiente – É um objeto cuja existência é limitada pelo tempo de vida do processo ou *thread* que o criou.

OLE – (*Object Linking and Embedding*) Significa incorporação de objetos. É uma forma de comunicação inter-processo criado pela Microsoft, que permite a um aplicativo incorporar ou vincular informações criadas em outro aplicativo. Feito isto, um documento composto é criado.

Operação – É um serviço que pode ser requisitado. Uma operação tem uma assinatura associada, a qual restringe quais parâmetros atuais são válidos.

ORB – (*Object Request Broker*) É um elemento da arquitetura CORBA, que fornece um meio através do qual clientes podem fazer e receber pedidos e respostas.

ORB Core – É o componente do ORB que transporta um pedido de um cliente para um adaptador apropriado do objeto alvo.

Pacote – É uma unidade de dado enviada através de uma rede de comutação de pacotes.

Pedido – Um cliente faz um pedido (ou requisição) quando necessita que uma operação seja executada. Um pedido consiste de uma operação e zero ou mais parâmetros atuais.

Pedido One-way – É um pedido onde um cliente não espera pelo término do mesmo, ou não pretende obter resultados de retorno.

Pedido Síncrono – É um pedido onde o cliente espera seu pelo término.

Pipeline – É uma técnica usada em processadores avançados, onde o processador inicia a execução de uma segunda instrução antes da primeira ter sido completada. Isto é, diversas instruções podem estar no *pipeline* simultaneamente, cada uma em um diferente estágio de processamento. O *pipeline* é dividido em segmentos e cada segmento pode executar sua operação concorrentemente com os outros segmentos. Quando um segmento completa uma operação, ele passa o resultado ao próximo segmento no *pipeline* e busca a próxima operação do segmento precedente. Os resultados finais de cada instrução emerge no final do *pipeline* em rápida sucessão.

Pixel – Representa o menor elemento de uma imagem. Em sistemas RGB (*Red/Green/Blue*), um *pixel* é uma tríade composta por um ponto vermelho, um ponto verde e um ponto azul.

Polimorfismo – É o uso de um único nome para definir várias formas distintas. Nas linguagens orientadas a objeto, como a C++, corresponde a criação de uma família de funções que compartilham o mesmo nome, mas cada uma possui código independente.

Portabilidade – Característica do sistema de *software* capaz de ser transferido de um computador para outro sem mudanças.

Portos - O protocolo TCP permite que múltiplos programas em uma mesma máquina se comuniquem concorrentemente e demultiplexem o tráfego TCP que chega de outros programas. Como o UDP, o TCP incorpora objetos abstratos chamados portos, que identificam o destino comum dentro de uma máquina. Cada porto é associado a um inteiro usado para identificá-lo. Como as associações do número aos portos são locais a uma dada máquina e únicos dentro daquela máquina, o destino final para o tráfego TCP é unicamente especificado pela associação do endereço IP, bem como o número do porto TCP daquele *host*.

Proxy – Na arquitetura CORBA, quando um objeto requerido por um cliente é remoto, uma referência ao objeto irá se referir a um objeto “substituto” (*proxy*) do objeto remoto. Um *proxy*, portanto, é um representante local, para um objeto remoto.

Realidade Virtual – Campo da Ciência da Computação que trata das simulações 3D em tempo real, interativas, que imitam o mundo real ou fictício, proporcionando um ambiente virtual com o qual um usuário pode interagir.

Referência a um objeto – É um valor que, unicamente, identifica um objeto. Referências a objetos nunca podem ser reusadas para identificar um outro objeto.

Renderização – Adicionar iluminação, sombreamento e cores a uma cena tridimensional.

Repositório de Implementação - (*Implementation Repository*) É um local de armazenamento para informações de implementação de objetos.

Repositório de Interface – (*Interface Repository*) É um local para armazenamento de informações de *interface*.

Resolução de um Método – É a seleção de um método para executar uma operação requisitada.

Resultados – É a informação retornada ao cliente, a qual pode incluir valores como informação de estado indicando que condições excepcionais foram levantadas, quando tentou se executar o serviço requisitado.

Run Time – Tempo de Execução. É o momento em que determinado processo, *thread* ou aplicativo está sendo executado.

Sensor – É um dispositivo que responde ao movimento físico e que transmite a informação de posição e (possivelmente) orientação resultante.

Serviço de Eventos CORBA – O serviço de eventos permite aos componentes no barramento (ORB) armazenarem, dinamicamente, eventos de seus interesses. O serviço define um objeto bem conhecido chamado de "canal de evento", que coleta e distribui eventos entre componentes.

Servidor – É um processo implementando um ou mais operações em um ou mais objetos.

Sistema Distribuído – É um conjunto de elementos de processamento autônomos, interligados através de um subsistema de comunicação de topologia arbitrária. A comunicação entre processos residentes em elementos de processamento diferentes é realizada através de troca de mensagens. Possui um sistema operacional único e distribuído entre os diversos elementos de processamento, de forma a fornecer ao usuário um único computador virtual.

Sistemas de Tempo Real – São sistemas que devem servir à aplicações que controlam processos externos, e que possuem tempos de resposta limitados. Geralmente são comandados por sinais de interrupção, provenientes de sensores.

Skeleton – É um componente específico de *interface* do ORB, o qual assiste a um adaptador de objetos na passagem dos pedidos para os métodos particulares.

Sockets - É a base para a E/S de Rede do Unix. O *socket* é uma generalização do mecanismo de acesso aos arquivos que fornece um ponto final (endpoint) para a comunicação. Como no acesso a arquivo, as aplicações pedem ao sistema operacional para criar um *socket* quando é necessário. O sistema retorna um inteiro que a aplicação usa para referenciar o recém criado *socket*. A diferença principal entre descritores de arquivos e *sockets* é que o sistema operacional liga um descritor de arquivo a um específico arquivo ou dispositivo (quando a aplicação chama um *OPEN*), mas ele pode criar *sockets* sem ligá-lo a um destino específico. A aplicação pode escolher suprir um endereço de destino cada vez que ela usa o *socket* (por exemplo, quando usa datagramas), ou pode escolher ligar (*bind*) o endereço de destino ao *socket* evitar ter que especificá-lo repetidamente (por exemplo, fazendo uma conexão TCP).

Stub – É um procedimento local que corresponde a uma operação simples, a qual invoca uma operação remota, quando chamada.

Surrogate – É um objeto substituto local para um objeto remoto. O mesmo que objeto *proxy*.

Tarefa WTK – (*task*) É uma função definida pelo usuário, a qual é associada ao WTK ou aos objetos do usuário.

TCP – (*Transmission Control Protocol*) É o protocolo padrão ao nível de transporte da Internet, que oferece um serviço confiável de comunicação bidirecional (*full duplex*), do qual muitos protocolos de aplicação dependem. O TCP permite a um processo em um computador enviar um fluxo de dados para um processo em outro.

É orientado a conexão, uma vez que antes de se transmitir os dados, os participantes devem estabelecer uma conexão. Os *softwares* que implementam o TCP, geralmente, residem no sistema operacional e usam o protocolo IP para transmitir informações através da Internet.

Textura – É um mapa de bits (*bitmap*) de imagem geralmente criado com o propósito de se aplicar imagens complexas a polígonos simples, a fim de aumentar o desempenho de aplicações gráficas tridimensionais.

Threads – Representam sequências de instruções executadas independentemente, dentro de um processo. *Threads* não são processos propriamente ditos, são fluxos de controle os quais executam em separado dentro de um processo.

Tipo de dado – É uma categorização dos valores dos argumentos de uma operação, tipicamente cobrindo o comportamento e a representação.

Tipo de Interface – É um tipo satisfeito por qualquer objeto que satisfaz uma *interface* particular.

UDP – (*User Datagram Protocol*) É o protocolo padrão da Internet, o qual permite a um processo em um computador enviar um datagrama para um processo em outro. O UDP usa o protocolo IP para entregar os datagramas. Conceitualmente, a diferença importante entre o UDP e o IP, é que as mensagens UDP incluem um número de protocolo de porto, permitindo ao emissor distinguir entre múltiplos destinos (programas de aplicação) em um computador remoto.

Universo WTK – É o receptáculo de todas as entidades consideradas pelo gerente de simulação do WorldToolkit – incluindo objetos gráficos e outras entidades: luzes, sensores, pontos de vista, etc.

Valor – É qualquer entidade que possa ser um parâmetro atual em um pedido. Os valores servem para identificar os objetos em referências a objetos.

Wireframe – É um estilo de renderização no qual texturas, materiais e sombreado não são visíveis. Isto porque, somente as linhas externas dos polígonos são renderizadas.