RODRIGO BARROCA DIAS FERRAZ

**Um Modelo de Objetos e Arquitetura para
Aplicações Distribuídas de Realidade Virtual**

São Paulo
2018

RODRIGO BARROCA DIAS FERRAZ

**Um Modelo de Objetos e Arquitetura para
Aplicações Distribuídas de Realidade Virtual**

Tese apresentada à Escola Politécnica da
Universidade de São Paulo para obtenção
do título de Doutor em Ciências.

São Paulo
2018

RODRIGO BARROCA DIAS FERRAZ

**Um Modelo de Objetos e Arquitetura para
Aplicações Distribuídas de Realidade Virtual**

Tese apresentada à Escola Politécnica da
Universidade de São Paulo para obtenção
do título de Doutor em Ciências.

Área de Concentração:

Sistemas Eletrônicos

Orientadora:

Profa. Dra. Roseli de Deus Lopes

São Paulo
2018

Este exemplar foi revisado e corrigido em relação à versão original, sob responsabilidade única do autor e com a anuência de seu orientador.

São Paulo, _____ de _____ de _____

Assinatura do autor: _____

Assinatura do orientador: _____

Catálogo-na-publicação

Ferraz, Rodrigo Barroca Dias

Um Modelo de Objetos e Arquitetura para Aplicações Distribuídas de Realidade Virtual / R. B. D. Ferraz — versão corrigida — São Paulo, 2018.
108 p.

Tese (Doutorado) – Escola Politécnica da Universidade de São Paulo. Departamento de Engenharia de Sistemas Eletrônicos.

1. Realidade Virtual 2. Sistemas Distribuídos I. Universidade de São Paulo. Escola Politécnica. Departamento de Engenharia de Sistemas Eletrônicos. II.t.

Dedico este trabalho ao meu filho Guilherme, como
um incentivo à suas futuras conquistas acadêmicas.

AGRADECIMENTOS

À Profa. Dra. Roseli de Deus Lopes, minha orientadora, pela dedicação, apoio e por acreditar neste trabalho.

Ao meu pai, Carlos Alberto Medon Dias Ferraz, por me apoiar e ajudar, mesmo fisicamente distante, durante todo este trabalho, principalmente nos momentos finais, transmitindo sua experiência profissional e acadêmica.

À minha mãe, Rita de Cássia, pelo apoio, e por se deslocar milhares de quilômetros para estar presente e apoiar os momentos finais deste trabalho.

À minha esposa Daniela e ao meu filho Guilherme por todo apoio, paciência e compreensão ao longo deste trabalho, principalmente nos momentos mais difíceis.

Ao Prof. Dr. Marcelo Knörich Zuffo, pelo incentivo e disponibilização de recursos e tempo para a realização desta pesquisa.

Aos colegas do Núcleo de Realidade Virtual do Laboratório de Sistemas Integráveis da USP, e aos colegas de pós-graduação Marcio Cabral e Olavo Belloc, pelas longas discussões sobre idéias para o projeto, pela contribuição com suas experiências e pelo incentivo.

RESUMO

Técnicas e ferramentas (*middlewares* para desenvolvimento de aplicações distribuídas são utilizadas no caso dos recursos de um único computador não serem suficientes, ou para atender requisitos de qualidade de serviço, ou mesmo para abranger uma maior quantidade de usuários. A Realidade Virtual é uma das áreas da computação que, devido a sua complexidade, se beneficia de tais técnicas e *middlewares*, seja para proporcionar ambientes mais imersivos e/ou possibilitar a interação de múltiplos usuários.

A Realidade Virtual engloba uma ampla diversidade de aplicações, incluindo infraestruturas de alta imersão, jogos multi-jogadores, e simuladores de voo, e cada classe de aplicação têm demandas específicas de distribuição, como escalabilidade, ou requisitos de tempo-real, entre outros. Devido à complexidade e a essa diversidade de aplicações de realidade virtual, as alternativas de *middlewares* disponíveis atendem requisitos específicos de uma única classe, ou, as genéricas, não fornecem modelos de objetos, comunicação ou execução favoráveis às aplicações de realidade virtual.

Esse trabalho tem o objetivo de propor um novo modelo de objetos e arquitetura de um *middleware* para desenvolvimento de aplicações distribuídas de realidade virtual suficientemente flexível para atender diversas classes de aplicações. O modelo de objetos proposto une os modelos publicador/assinante e requisição/ resposta em uma única entidade similar aos objetos do paradigma de programação orientada a objetos, que, além de empregar conceitos intuitivos para desenvolvedores, possibilita o desacoplamento lógico entre os processos, simplificando o projeto e a implementação das aplicações distribuídas. O principal diferencial desta abordagem é a possibilidade de distribuir a implementação de cada membro dos objetos em diferentes processos.

Palavras-chave: Aplicações distribuídas. Programação orientada a objetos. Realidade Virtual. Modelos de objetos. Modelos de comunicação. Arquitetura de *middleware*.

ABSTRACT

Techniques and middlewares for distributed application development are used in cases the resources available in a single computer are not enough for the demanded complexity, to meet the required quality of services, or even to cover a greater number of users. The Virtual Reality is one of the computer fields that, because of its complexity, benefits from such techniques and tools, either to provide more immersive virtual environments, or to enable multi-user interaction.

The Virtual Reality covers a wide range of applications, including high immersion infrastructures, multiplayer games, and full mission flight simulators. Each application class has specific distribution demands, such as high scalability, or real-time requirements. Because of complexity and the diversity of virtual reality applications, the distribution solutions available meet the specific requirements of a single class, or the generic ones do not provide object, communication, or execution favorable for virtual reality applications.

This work proposes a new model of objects and architecture of a middleware for distributed virtual reality applications that is flexible enough to suit several classes of applications. The proposed object model joins the publisher/subscriber and request/response models to a single logic entity similar to the objects of the object-oriented programming paradigm. Besides being an intuitive concept for developers, this entity enables logical decoupling between processes, simplifying the design and implementation of distributed applications. The main feature of this approach is the possibility to distribute the implementation of each member of the objects in different processes.

Keywords: Distributed applications. Object-oriented programming. Virtual reality. Object models. Communication models. Middleware architecture.

LISTA DE FIGURAS

3.1	Componentes estruturais do HLA.	27
3.2	Visão geral do CORBA.	30
3.3	Interfaces de Requisição.	32
3.4	Interoperabilidade entre ORBs.	33
3.5	Espaço Global de Dados do DDS.	34
3.6	Agrupamento de objetos DDS em tópicos.	35
3.7	Camadas de Software do DDS.	37
3.8	Aplicação utilizando a <i>Test and Training Enabling Architecture</i> (TENA).	38
3.9	Interação entre Servente e <i>Proxy</i>	39
4.1	Visão Geral do Modelo de Objetos.	44
4.2	Estrutura dos Objetos Distribuídos.	47
4.3	Exemplo de relação entre Participantes e Objetos Distribuídos.	49
5.1	Camadas de software no Participante.	54
5.2	Comunicação entre Participantes.	54
5.3	Interação entre membros e Participante.	59
5.4	Relação entre <i>Proxy</i> , Adaptador e Canal.	60
5.5	Visão geral da dinâmica das interações.	61
5.6	Diagrama de sequência de modificação de valores de atributos.	62
5.7	Diagrama de sequência de disparo de eventos.	62
5.8	Diagrama de sequência da invocação de métodos.	63
5.9	Diagrama de Sequência da Localização de Membros.	65
6.1	Diagrama de pacotes da implementação.	68
6.2	Diagrama de Classes Gerenciador de Aplicação.	69

6.3	Diagrama de Classes do Gerenciamento de Tipos.	70
6.4	Diagrama de Classes do Gerenciamento de Instâncias.	72
6.5	Diagrama de Classes das Interações.	75
6.6	Diagrama de Classes do Subcomponente Canal.	78
6.7	Formato geral de um Valor.	80
6.8	Formato das Cadeias de Caracteres.	81
6.9	Formato das Sequências de Elementos.	81
6.10	Formato Geral de uma Interação.	82
6.11	Formato dos parâmetros de Invocação de Método.	83
6.12	Formato dos parâmetros de Retorno de Invocação de Método.	83
6.13	Diagrama de Classes do Exemplo de Aplicação Distribuída.	84
6.14	Topologia do Exemplo de Aplicação Distribuída.	85
7.1	Diagrama de Classes do exemplo de jogo.. . . .	90
7.2	Diagrama de Instâncias do exemplo de jogo.. . . .	92

LISTA DE TABELAS

3.1	Comparação entre os principais <i>middlewares</i>	42
4.1	Comparação entre a proposta e os principais <i>middlewares</i>	52
5.1	Interações com membros.	59
6.1	Identificadores de Tipos (Hexadecimal).	81
6.2	Identificadores de Interações (Hexadecimal).	82
7.1	Distribuição dos membros da instância da classe <i>GerenciadorJogo</i>	94
7.2	Distribuição dos membros do objeto distribuído das instâncias da classe <i>Jogador</i>	94

LISTA DE ABREVIATURAS

ALSP	<i>Aggregate Level Simulation Protocol</i>
AOS	Arquitetura Orientada à Serviços
API	<i>Application Programming Interface</i>
ARPANET	<i>Advanced Research Projects Agency Network</i>
CAVE	<i>Cave Automatic Virtual Environment</i>
CCGE	Canal Com Garantia de Entrega
CORBA	<i>Common Object Request Broker Architecture</i>
CPR	Chamada de Procedimentos Remotos
CSGE	Canal Sem Garantia de Entrega
DCPS	<i>Data-Centric Publish-Subscribe</i>
DDS	<i>Data Distribution Service</i>
FOM	<i>Federation Object Model</i>
DIS	<i>Distributed Interactive Simulation</i>
DLRL	<i>Data Local Reconstruction Layer</i>
DoD	<i>US Department of Defense</i>
DSM	<i>Distributed Shared Memory</i>
GIOP	<i>General Inter-ORB Protocol</i>
HLA	<i>High-Level Architecture</i>
HPC	<i>High-Performance Computing</i>
IDL	<i>Interface Definition Language</i>
IEEE	<i>Institute of Electrical and Electronics Engineers</i>
IMR	Invocação de Métodos Remotos
OD	Objeto Distribuído
OMG	<i>Object Management Group</i>
OMT	<i>Object Model Template</i>
ORB	<i>Object Request Broker</i>

RMI *Remote Method Invocation*

RPC *Remote Procedure Call*

RTI *Run-Time Infrastructure*

RV *Realidade Virtual*

SDO *Stateful Distributed Object*

SOA *Service Oriented Architecture*

SOM *Simulation Object Model*

TENA *Test and Training Enabling Architecture*

TDL *TENA Definition Language*

SUMÁRIO

1	Introdução	15
1.1	Objetivos	17
1.2	Motivação	18
1.3	Materiais e Métodos	19
1.4	Contribuições	19
1.5	Estrutura	20
2	Conceitos Básicos	21
2.1	Terminologia	21
2.2	Requisitos	22
2.2.1	Requisitos Comuns	22
2.2.2	Infraestruturas de Alta Imersão	23
2.2.3	Ambientes Multiusuários	23
2.2.4	Simuladores de voo	23
2.3	Diretrizes Gerais	24
2.4	Síntese	25
3	Estado da Arte	26
3.1	High-Level Architecture	26
3.2	CORBA	29
3.3	Data Distribution Service	33
3.4	TENA	37
3.5	Comparação Entre <i>Middlewares</i>	41
3.6	Síntese	42

4	Proposta de um Novo Modelo de Objetos para Aplicações Distribuídas	43
4.1	Visão Geral	43
4.2	Objetos Distribuídos	45
4.3	Participante	47
4.4	Modelo de Comunicação e Declarações	48
4.5	Gerenciador de Aplicação Distribuída	50
4.6	Comparação	51
4.7	Síntese	51
5	Arquitetura de Software	53
5.1	Visão Geral	53
5.2	Camada Núcleo	55
5.2.1	Tipos	56
5.2.2	Instâncias	58
5.2.3	Canais	60
5.2.4	Dinâmica das Interações com Membros	61
5.3	Camada Serviços Básicos	63
5.3.1	Registro de Tipos	64
5.3.2	Localização de Objetos	64
5.3.3	Gerenciador de Canais	65
5.4	Síntese	65
6	Prova de Conceito	67
6.1	Visão Geral	67
6.2	Implementação do Núcleo	67
6.2.1	Tipos	69
6.2.2	Instância	71
6.2.3	Interação	74

6.2.4	Canal	77
6.3	Protocolos e Canais de Comunicação	79
6.3.1	Protocolo de Aplicação	80
6.3.2	Implementação do Canal TCP/IP Simples	83
6.4	Exemplo de Utilização da Camada Núcleo	84
6.5	Síntese	86
7	Exemplo de Aplicação Distribuída	88
7.1	Dinâmica do Jogo	88
7.2	Classes e Estruturas de Dados	89
7.3	Organização das Instâncias	91
7.4	Participantes e Canais de Comunicação	92
7.5	Distribuição dos Membros dos Objetos	93
7.6	Síntese	95
8	Conclusão	96
8.1	Principais Contribuições	97
8.2	Trabalhos Futuros	97
	Referências	99
	Apêndice A – Código Fonte de Exemplo de Participante	105

1 INTRODUÇÃO

Na computação, uma aplicação distribuída é um sistema de software cujos componentes (processos) localizados em diferentes computadores conectados através de uma rede de dados atuam de forma coordenada de forma a alcançar um objetivo comum [1].

Aplicações distribuídas apresentam um grau maior de complexidade pois, além da lógica particular da aplicação, é necessário lidar com outros desafios como a concorrência, comunicação e coordenação da execução dos processos. Apesar da sua maior complexidade, existem diversas razões para se optar pela distribuição de uma aplicação, entre as quais:

- Os recursos (processamento, memória, armazenamento etc.) disponíveis em um computador não atendem aos requisitos da aplicação;
- Colaboração entre diversos usuários (jogos, por exemplo);
- Distribuição geográfica de informação (Web);
- Elevar a disponibilidade e tolerância a falhas da aplicação.

A computação distribuída é tema de pesquisa desde a década de 1960 e neste período surgiram as primeiras redes locais de computadores, e consequentemente, as primeiras aplicações distribuídas. No final desta década surgiu a *Advanced Research Projects Agency Network* (ARPANET), uma grande rede de computadores em escala geográfica, que posteriormente se tornou a atual internet, servindo como base para inúmeras aplicações distribuídas.

Mesmo com a existência do conceito de computação distribuída, até meados de 1990 muitos sistemas de alto desempenho eram baseados em supercomputadores monolíticos, de custo elevado e arquitetura proprietária. Desde então, muitos destes sistemas foram migrados para aglomerados de computadores conectados através de uma rede de dados local (*clusters*). Neste modelo são utilizados computadores de prateleira (geralmente topo

de linha), de arquitetura padronizada, reduzindo significativamente os custos de aquisição e manutenção, mas, em contrapartida, com uma elevada complexidade do software para permitir a execução em múltiplos computadores.

Nas décadas de 1990 e 2000 foram realizadas muitas pesquisas com objetivo de propor e projetar protocolos, modelos de comunicação, objetos e arquiteturas para atenderem aos requisitos de classes de aplicações específicas.

Para facilitar o desenvolvimento de aplicações distribuídas, estes protocolos e modelos são geralmente implementados como *middlewares*, apresentando à aplicação entidades lógicas que abstraem os detalhes de baixo nível. Atualmente a maioria destes modelos são baseados na disseminação ativa e assíncrona de dados, como o modelo de comunicação publicador/assinante, ou seguem um padrão requisição/resposta, como os modelos Chamada de Procedimentos Remotos (CPR) e Arquitetura Orientada à Serviços (AOS).

A Realidade Virtual é uma das áreas que pode se beneficiar da computação distribuída, seja para possibilitar a interação de múltiplos usuários em diferentes localidades, seja para obter maior desempenho por meio de um aglomerado de computadores e atingir maiores níveis de imersão. O espectro de aplicações de RV é bastante abrangente, incluindo infraestruturas de alta imersão, como CAVE (*Cave Automatic Virtual Environment* (CAVE)) [2, 3], jogos multi-jogadores e simuladores de voo, cada uma com requisitos de comunicação específicos.

Outra particularidade da realidade virtual o envolvimento de diversas áreas da computação em uma única aplicação para a simulação em tempo-real do mundo virtual, interação com os usuários, exibição das imagens, reprodução de sons entre outros. Devido a esta complexidade, as aplicações distribuídas de realidade virtual podem se beneficiar simultaneamente de diversos padrões e modelos de comunicação e objetos. Pode-se citar como exemplo o seguinte cenário:

- Emprego do modelo publicador/assinante para replicar periodicamente os estados das entidades do mundo virtual entre processos e disseminar eventos aperiódicos,
- Utilização do modelo requisição/resposta para sincronização e coordenação da execução e transferências sob-demanda de recursos como imagens, modelos 3D, áudio entre outros.

Devido à complexidade e à diversidade de aplicações distribuídas de realidade virtual, observou-se uma escassez de *middlewares* flexível suficiente para atender um espectro

abrangente de requisitos, e que fornece modelos de alto nível direcionados para a multidisciplinaridade das aplicações de RV.

As deficiências dos modelos e *middlewares* existentes podem ser minimizadas de diversas formas, como:

- Emprego de diversos protocolos e/ou *middlewares* para cada parte da aplicação;
- adotar uma única solução de distribuição e adaptar o restante do software;
- implementar toda a infraestrutura necessária para distribuir a aplicação.

Estas alternativas implicam na maior complexidade final da aplicação, impactando negativamente o tempo de desenvolvimento e a dificuldade na manutenção, além da maior dificuldade em otimizar o desempenho do software.

1.1 Objetivos

Dada a relevância das pesquisas de técnicas e ferramentas para aplicações distribuídas, e a escassez de modelos híbridos que atendam casos complexos, como aplicações distribuídas de Realidade Virtual (RV), o principal objetivo deste trabalho é propor um novo *middleware* para aplicações distribuídas.

O principal diferencial desta proposta é a introdução de um novo modelo híbrido que reúna conceitos consolidados, tais como publicador/assinante e requisição/resposta, em uma estrutura denominada Objetos Distribuídos.

O desenvolvimento desta proposta será direcionado para atender as demandas e requisitos de aplicações distribuídas de Realidade Virtual, porém, espera-se que os resultados deste trabalho possam ser extrapolados para outras categorias de aplicações distribuídas tais como, *High-Performance Computing* (HPC), aplicações base para a computação em nuvem, e outras aplicações distribuídas em rede local.

Para apresentação de um novo *middleware*, este trabalho contempla os seguintes objetivos específicos:

- Realizar um estudo dos modelos e *middlewares* existentes;
- Conceber um novo modelo de alto nível para aplicações distribuídas;
- Conceber a arquitetura de software para o *middleware* proposto;

- Projetar, implementar e avaliar uma prova de conceitos do *middleware* baseado no modelo concebido.

1.2 Motivação

Apesar das primeiras pesquisas sobre sistemas distribuídos terem sido realizadas há mais de cinco décadas, este tema ainda é de bastante relevância pois ainda é a melhor abordagem para se reunir recursos computacionais suficientes para executar aplicações de qualquer complexidade. Por este motivo a computação distribuída está atualmente presente em muitos ramos da computação, sendo, por exemplo, a base da internet e da computação em nuvem.

Mais especificamente, a idéia deste trabalho surgiu a partir das experiências adquiridas ao longo da última década com os projetos de pesquisa e desenvolvimento realizados no Núcleo de Realidade Virtual do Laboratório de Sistemas Integráveis da Universidade de São Paulo.

No início da década de 2000, o laboratório foi um dos pioneiros a utilizar a computação distribuída para substituir os supercomputadores gráficos por aglomerados de computadores comuns, alcançando desempenho superior com um custo inferior nas aplicações de visualização em tempo real [3, 4, 5]. Nesta época, o laboratório contribuiu com técnicas e modelos para distribuição de aplicações gráficas [6, 7].

Outros projetos mais complexos na área de RV, já no final da década, demandaram a distribuição não somente da visualização, mas de toda a simulação do ambiente virtual. Alguns exemplos são os projetos Passeio Digital (instalado no museu Catavento Cultural) [8] e o simulador de ponte de manobras de embarcações navais (Passadiço), desenvolvido junto com a Marinha do Brasil [9, 10]. Observou-se então que as técnicas desenvolvidas para computação gráfica distribuída não atendiam às demandas do restante da aplicação. Foi constatada também uma escassez de um modelo unificado e/ou *middlewares* que suprissem as dificuldades da distribuição destas aplicações de forma satisfatória e com desempenho suficiente. Para cada um destes projetos foram experimentadas abordagens diferentes [11, 12], servindo como base para a concepção do modelo e *middleware* proposto neste trabalho.

Assim como mencionado anteriormente, apesar da motivação inicial e os requisitos que balizam este projeto serem provenientes das aplicações distribuídas de Realidade Virtual, acredita-se que os resultados a serem alcançados beneficiarão outras classes de aplicações.

1.3 Materiais e Métodos

Para realizar o objetivo de propor um novo middleware destinado as aplicações distribuídas mais complexas, como Realidade Virtual, foram realizadas as seguintes ações e procedimentos:

Inicialmente foi realizada uma pesquisa bibliográfica para avaliar os modelos e middlewares existentes para aplicações distribuídas com a apresentação dos detalhes de arquitetura, modelos de objetos e protocolos de comunicação finalizando com uma tabela comparativa das principais características desses modelos e middlewares.

A partir das citadas avaliações e dos requisitos de projeto foram elaboradas as diretrizes gerais para a concepção do novo middleware que atenda aos requisitos de aplicações de Realidade Virtual.

A concepção foi baseada no modelo de objetos e nesse sentido foi detalhado os entes desse modelo, como Objetos Distribuídos, Participantes, Gerenciador de Aplicação e modelos de comunicação. A concepção foi finalizada com uma tabela comparativa entre os middlewares existentes e o middleware proposto.

Com base no modelo de objetos proposto foi desenvolvida uma arquitetura de software com detalhes estruturais e dinâmicos do novo middleware, em conformidade com os requisitos de projeto e das diretrizes gerais.

Como prova de conceito do modelo de objetos e arquitetura de softwares apresentados foi realizada a implementação de um protótipo da camada núcleo do middleware na qual se demonstrou a viabilidade técnica da proposta.

Para comprovar também a aplicabilidade da proposta foram implementados um protocolo simples de comunicação e um canal para troca de dados entre os Participantes.

1.4 Contribuições

A principal contribuição deste trabalho é apresentar um conjunto de técnicas, conceitos e ferramentas, na forma de um *middleware*, que facilitem o projeto e implementação de aplicações distribuídas, em especial, as da área de Realidade Virtual. As contribuições específicas são as seguintes:

- Um novo modelo conceitual híbrido para aplicações distribuídas de realidade virtual, que une os padrões publicador/assinante e requisição/resposta;

- Uma arquitetura de software de um *middleware* que concretize o modelo conceitual;
- Protótipo de um *middleware* que utilize o modelo elaborado.

1.5 Estrutura

O restante deste trabalho está estruturado da seguinte maneira:

O capítulo 2 apresenta as definições básicas de sistemas software distribuídos, os requisitos para aplicações distribuídas de realidade virtual e as diretrizes gerais da proposta.

A revisão da literatura sobre modelos e *middlewarees* para sistemas distribuídos relevantes para esta pesquisa é apresentada no capítulo 3.

No capítulo 4 é apresentado um novo modelo conceitual de objetos para aplicações distribuídas.

A arquitetura de software do *middleware* proposto é detalhada no capítulo 5

O capítulo 6 apresenta os detalhes de implementação de uma prova de conceitos para demonstrar a viabilidade técnica da proposta.

No capítulo 7 é apresentado um exemplo de aplicação distribuída utilizando o *middleware* proposto.

As considerações finais e trabalhos futuros são apresentados no capítulo 8.

O apêndice A apresenta o código fonte de um exemplo de aplicação utilizando o protótipo do *middleware*.

2 CONCEITOS BÁSICOS

Este capítulo apresenta os conceitos básicos de computação distribuída empregados neste trabalho e os requisitos e diretrizes que guiaram a elaboração do modelo e da arquitetura propostos.

2.1 Terminologia

Neste trabalho foram consideradas as definições:

- *Middleware*: Camada de software entre a aplicação e o meio de comunicação, que fornece entidades de alto nível para facilitar a distribuição em diversos computadores em uma rede de dados.
- *Modelo de Objetos*: Conjunto de conceitos e entidades lógicas fornecidas pelo *middleware*, e a relação entre elas e com a aplicação.
- *Modelo de Comunicação*: Padrão alto nível de troca de dados entre entidades e membros de uma aplicação distribuída. Exemplos: Publicador/Assinante, Requisição/Resposta
- *Modelo de Execução*: Descrição da execução do componente de software, incluindo configuração de *threads*, ordenação, sincronismos entre outros.
- *Modelo de Arquitetura*: Descrição alto nível dos componentes de software de uma aplicação ou *middleware*, e como se relacionam estaticamente e dinamicamente. É um passo intermediário para a implementação dos modelos de objetos, comunicação e execução.
- *Publicador/Assinante*: Modelo de comunicação no qual as entidades produtoras de informação (publicadoras) disseminam de forma ativa os dados produzidos para as entidades consumidoras (assinantes), que recebem passivamente.

- *Requisição/Resposta*: Modelo de comunicação no qual entidades clientes enviam ativamente requisições de serviço ou informação para entidades provedoras passivas, e aguardam a resposta de tais serviços.

2.2 Requisitos

No contexto deste trabalho foram considerados alguns requisitos de alto nível e características desejáveis em *middlewares* e/ou modelos para serem empregados em aplicações distribuídas de realidade virtual. Alguns destes requisitos são comuns à diversas classes de aplicações, enquanto outros são específicos e, podendo ser, inclusive, conflitantes. Com o intuito de maximizar a abrangência, foram consideradas algumas classes de aplicações bastante distintas como referência de requisitos específicos.

2.2.1 Requisitos Comuns

Foram considerados os seguintes requisitos comuns às diversas classes de aplicação:

- Possibilitar alta taxa de atualização. Para maior imersão do usuário, as aplicações de RV devem garantir frequências mínimas de atualização. Atualmente o mínimo aceitável é 30Hz, porém o ideal é estar acima do 60Hz;
- Possibilitar baixa latência. O tempo entre uma ação do usuário e o instante em que ela é processada deve ser mínimo para não prejudicar a imersão. Atualmente o mínimo aceitável é 100ms
- Facilidade em replicar estados das entidades do mundo virtual entre processos. A simulação do mundo virtual constantemente atualiza o estado de suas entidades (posição, rotação etc.), que devem ser parcialmente ou totalmente replicadas em outros processos;
- Facilidade em difundir interações e eventos assíncronos do mundo virtual entre os processos;
- Facilidade em sincronizar e coordenar a execução dos processos;
- Flexível ao tipo de aplicação, possibilitando atender requisitos específicos;
- Simples e intuitivo de ser empregado;

2.2.2 Infraestruturas de Alta Imersão

Ambientes de alta imersão, como CAVEs, são geralmente compostas por diversos aparatos de interação, exibição de imagens, reprodução de áudio e outros geradores de estímulos ao usuário. O propósito de tais infraestruturas é proporcionar um alto grau de imersão do usuário ao ambiente virtual, e geralmente empregam um aglomerado computacional para a execução da aplicação.

Os principais requisitos de distribuição desta classe de aplicação são:

- Escalabilidade local, permitindo alcançar mais recursos computacionais adicionando novos computadores ao aglomerado;
- Baixo *overhead*, possibilitando a aplicação alcançar elevadas taxas de atualização, e baixa latência, promovendo um alto grau de imersão.

2.2.3 Ambientes Multiusuários

Ambientes multiusuários, como jogos multi-jogadores, são aplicações nas quais diversos usuários interagem colaborativamente com o mundo virtual. Nessa classe de aplicações a maior parte dos cada usuário geralmente utiliza um terminal computacional, conectado aos demais em uma rede local, ou por meio da internet.

Os requisitos de distribuição representativos desta categoria são:

- Escalabilidade de usuários por meio da internet;
- Suportar meios de comunicação heterogêneos, uma vez que os usuários podem estar conectados à internet por meio de diferentes tecnologias e qualidade de conexão;
- Suportar protocolos adaptativos. Como a qualidade de conexão nem sempre é garantida, a aplicação deve suportar desconexões e instabilidades, minimizando o impacto na qualidade da aplicação.

2.2.4 Simuladores de voo

Simuladores de voo são ambientes virtuais empregados para o treinamento de tripulação para operação de aeronaves. Existem diversas categorias de simuladores de voo variando o grau de realismo. Os simuladores de voo completos possuem a certificação máxima pelas

autoridades locais, e permitem o treinamento completo de pilotos para um determinado tipo de aeronave, sem a necessidade de treinamento em aeronaves reais.

Essas aplicações passam por um rigoroso processo de verificação e validação antes de serem certificados para treinamento. Para maior fidelidade, além da simulação precisa, estes simuladores utilizam alguns equipamentos eletrônicos de uma aeronave real (aviônicos), além do aglomerado computacional para a simulação aerodinâmica, de ambiente e dos sistemas da aeronave.

Desta forma, os principais requisitos específicos dos simuladores de voo são:

- Execução em Tempo-real (*soft real-time*), devido à integração com componentes reais de uma aeronave que não são tolerantes à atrasos na comunicação;
- Taxa de atualização fixa, para evitar variações na taxa de execução (*jitter*), prejudicando a imersão dos usuários. Os módulos críticos geralmente são executados à taxas divisores inteiros de 120Hz;
- Determinismo. Para a verificação e validação do simulador, a simulação deve ser previsível e reproduzível (para as mesmas condições de entrada, e o mesmo estado, as saídas devem ser sempre as mesmas).
- Elevado controle dos meios de comunicação, para possibilitar os requisitos de tempo-real e determinismo.

2.3 Diretrizes Gerais

Com base nos requisitos apresentados na seção 2.2, foram estabelecidas as diretrizes gerais guiaram a proposta de um novo *middleware*:

- Multi-paradigma (híbrido);
 - Comunicação Síncrona (Requisição/Resposta);
 - Disseminação de estados (publicador/assinante)
 - Disseminação de eventos/mensagens (publicador/assinante)
- Camada de software mínima;
- Não impor topologia baixo nível de comunicação;

- Suportar múltiplos protocolos de comunicação;
- Não impor modelo de execução;

O propósito de ser multi-paradigma, ou híbrido, é atender de forma simples os principais tipos de comunicação em uma aplicação distribuída de realidade virtual. O padrão publicador/assinante pode ser empregado tanto para sincronizar os estados das entidades da aplicação, bem como para disseminar eventos pontuais, como interações com o usuário, entre os processos. O modelo de requisição/resposta pode ser aplicado para a manutenção e sincronização temporal da execução.

As demais diretrizes foram definidas para minimizar *overhead* introduzido pelo *middleware*, e possibilitar a utilização em diversos tipos de aplicações de realidade virtual com diferentes requisitos específicos. Sendo uma camada de software mínima e não impondo protocolos e topologias de comunicação, e nem modelos de *threads*, o *middleware* poderá ser empregado desde aplicações que demandem maior escalabilidade, como sistemas de multi-projeção, até as que tenham requisitos de tempo-real, como os simuladores de voo.

2.4 Síntese

Este capítulo apresentou a definição dos termos básicos utilizados neste trabalho, os principais requisitos comuns e específicos de aplicações distribuídas de realidade virtual, e as diretrizes gerais adotadas durante a elaboração da proposta.

3 ESTADO DA ARTE

Esse capítulo apresenta um estudo de alguns dos principais *middlewares* para aplicações distribuídas. Foram selecionados *middlewares* que possam ser utilizados em aplicações distribuídas de realidade virtual, que apresentam um elevado grau de maturidade, e que influenciaram diretamente este trabalho.

O estudo apresenta a visão geral e os detalhes da arquitetura e modelo de comunicação de cada um dos *middlewares* selecionados. Ao final do capítulo é apresentada uma comparação entre as alternativas, utilizando critérios relevantes para aplicações distribuídas de realidade virtual e outras classes de alta complexidades.

3.1 High-Level Architecture

O *High-Level Architecture* (HLA) [13, 14] é uma arquitetura para reutilização e interoperabilidade de simulações [15]. Baseado no *Distributed Interactive Simulation* (DIS) [16] e no *Aggregate Level Simulation Protocol* (ALSP) [17], o HLA foi inicialmente desenvolvido em 1995 pelo Departamento de Defesa dos Estados Unidos da América - *US Department of Defense* (DoD) com a finalidade de suportar reutilização e interoperabilidade entre um grande número de simulações de diferentes tipos e, portanto, reduzir os custos de projetos. Atualmente o HLA é aprovado como um padrão aberto pelo *Institute of Electrical and Electronics Engineers* (IEEE).

O padrão HLA define regras, modelos e serviços que auxiliam o desenvolvimento de simulações distribuídas. O atual padrão (IEEE 1516) [18] é composto por três partes:

- IEEE Std. 1516-2010 - Arcabouço e Regras: Apresenta os componentes do HLA e as regras que devem ser seguidas pelas aplicações;
- IEEE Std. 1516.1-2010 - Especificação de Interface dos Federados: Especifica a Interface de Programação de Aplicação – *Application Programming Interface* (API) – dos serviços providos pelo HLA;

- IEEE Std. 1516.2-2010 - Especificação do Meta-Modelo de Objetos: Define o formato e a sintaxe para descrição dos objetos de simulação.

O HLA utiliza o conceito de Federação e Federado. A federação é o conjunto de simulações que interagem entre si. Os federados são os participantes da federação, ou seja, uma simulação individual. A interação entre federados de uma federação é realizada por meio de serviços providos pelo componente de software denominado *Run-Time Infrastructure* (RTI).

A API fornecida pelo RTI é especificada de acordo com a norma IEEE 1516.1, enquanto que os detalhes internos do RTI como protocolos, arquiteturas, algoritmos e outros são específicos de cada implementação. A figura 3.1 apresenta a visão geral de uma simulação distribuída utilizando o HLA.

Figura 3.1: Componentes estruturais do HLA.

Fonte: [15].

A comunicação entre federados ocorre por meio de objetos e interações. No contexto do HLA, objetos são elementos compostos por atributos correlatos, persistentes ao longo do tempo, utilizados para descrever o estado de cada parte da simulação. Já as interações são mensagens compostas por parâmetros, não persistentes com informações que são significantes apenas no instante de tempo em que elas foram enviadas.

A estrutura de atributos de um objeto e o conjunto de parâmetros de uma interação são definidos por classes de objetos e classes de interação respectivamente. Desta forma,

os objetos criados em uma federação são instâncias de classes de objetos, e as interações enviadas são instâncias de classes de interações.

Para permitir a interoperabilidade, o modelo de objetos do HLA determina que cada federado especifique todas as classes de objetos e interações que irá produzir e consumir. Estas especificações constituem o documento denominado *Simulation Object Model* (SOM).

Durante a fase de projetos de uma federação, os SOMs dos federados participantes são utilizados para constituir o *Federation Object Model* (FOM) específico, contendo a descrição de classes de objetos e interações que irão trafegar durante a execução da federação. Tanto o SOM quanto o FOM são formatados de acordo com o formato *Object Model Template* (OMT), especificado na parte três do padrão IEEE 1516.

O modelo de comunicação utilizado pelo HLA é o publicador/assinante. Dessa forma, para poder enviar e receber informações na federação, os federados devem informar antes quais as classes de objetos e interações pretende enviar (publicar), ou deseja receber (assinar). Como a declaração de intenção de envio e recebimento é por classe, e não instância, os federados que assinam uma determinada classe de objetos irão receber atualizações de todas as instâncias de objetos desta classe.

Portanto, cabe à implementação do federado filtrar as instâncias de interesse. No caso das classes de objetos, os federados podem se declarar como publicadores ou assinantes de atributos específicos de uma determinada classe, evitando o tráfego de atributos que não sejam de interesse do federado.

Toda a interação do federado com a federação, e outros federados, é realizada por meio dos seguintes serviços fornecidos pelo RTI.

- *Gerenciamento de Federações*: Possibilita aos federados criarem e finalizarem federações, e entrarem e saírem de federações já existentes.
- *Gerenciamento de Declarações*: Define meios para os federados declararem quais classes de objetos e interações pretendem publicar, ou desejam receber.
- *Gerenciamento de Objetos*: Interface para gerenciar as instâncias de objetos em uma federação.
- *Gerenciamento de Posse*: Oferece meios para os federados cederem ou solicitarem a posse de objetos e atributos, e negociarem em casos de conflitos.
- *Gerenciamento de Tempo*: Gerencia o avanço temporal da federação.

- *Gerenciamento de Distribuição de Dados*: Gerencia a segmentação do espaço de objetos em regiões, permitindo aos federados filtrar os objetos e interações a serem recebidas.
- *Serviços de Suporte*: Fornece serviços de suporte aos federados, como a consulta de classes e instâncias disponíveis na federação, entre outros.

O HLA foi elaborado para atender simulações analíticas e interativas. Desta forma a federação pode executar tanto no modo "o mais rápido possível", ideal para simulações analíticas; quanto em tempo-real, ideal para simulações interativas. O passo global pode ser ajustado para fixo, guiado por eventos discretos, ou sem passo global, ou seja, sem sincronização temporal entre os federados.

Apesar de ser um padrão aberto e mantido pelo IEEE, o HLA define conceitos apenas em alto nível e uma API. Algoritmos, métodos e protocolos não fazem parte do padrão e acabam sendo específicos de cada implementação. A vantagem desta abordagem é que cada implementação pode explorar livremente técnicas e protocolos mais otimizados, sem quebrar a compatibilidade com os federados. Além disso, uma federação pode adotar qualquer implementação do RTI.

Por outro lado, a desvantagem é que as implementações do RTI geralmente são incompatíveis, ou seja, todos os federados da federação devem utilizar a mesma implementação do RTI, ou utilizar federados tradutores para fazer a ponte entre uma implementação e outra. Como as licenças das implementações comerciais são por federado, grandes federações acabam tendo um custo elevado em licenças. Além disso, por ser um padrão complexo, existem poucas implementações disponíveis, sendo que as gratuitas são incompletas ou pouco maduras, e as comerciais apresentam um custo muito elevado. Assim, mesmo sendo um padrão aberto e abrangente, o emprego do HLA em aplicações de RV que visem o baixo custo ou para fins acadêmicos, acaba sendo proibitivo.

3.2 CORBA

O *Common Object Request Broker Architecture* (CORBA) [19, 20, 21] é um padrão definido pela *Object Management Group* (OMG) criado para facilitar a comunicação entre processos em uma aplicação distribuída. Por se tratar de uma especificação, atualmente existem diversas implementações do CORBA, tanto comerciais quanto não comerciais. O CORBA pode ser empregado em ambientes homogêneos ou heterogêneos, ou seja, as

partes da aplicação distribuída podem ser implementadas em diferentes linguagens, e executar em diferentes sistemas operacionais.

O modelo do CORBA é baseado no conceito de orientação a objetos cuja principal entidade são os objetos distribuídos. Assim como nas linguagens orientadas a objetos, um objeto distribuído é uma estrutura de dados composta por atributos e operações (métodos) relacionados entre si. A diferença é que no CORBA os objetos podem estar em processo diferente do utilizador do objeto. A implementação do CORBA (*middleware*) é responsável em localizar e encaminhar de forma transparente as operações em um objeto para o processo que o mantém.

Os processos que utilizam um determinado objeto são denominados clientes deste objeto. Já a entidade que implementa as funcionalidades de um objeto são denominados servidores. Um processo pode ser cliente e conter servidores simultaneamente.

A comunicação entre cliente e servente é tratada pelo componente de software denominado *Object Request Broker* (ORB). Este componente abstrai os detalhes de baixo nível do restante da aplicação. A figura 3.2 ilustra a arquitetura básica do CORBA, com os principais componentes, e as interações entre eles.

Figura 3.2: Visão geral do CORBA.

Fonte: [21].

Um cliente interage com um objeto através de uma referência que contém informações sobre a localização do servente do objeto, incluindo um endereçamento do processo, e a identificação do objeto. O ORB utiliza estas informações para encaminhar as requisições para o ORB associado com o servente.

O modelo de comunicação entre cliente e servente é baseado no *Remote Procedure*

Call (RPC), no qual os clientes têm acesso somente aos métodos dos objetos. Quanto um método é invocado pelo cliente, é criada uma requisição contendo a referência para o objeto, o método invocado e os parâmetros de invocação. Esta requisição é despachada para o ORB local que encaminha para o ORB remoto. O ORB remoto envia para o servente associado com o objeto. Este então trata a requisição, retornando o resultado para o cliente utilizando o caminho inverso.

Para ser compatível com várias linguagens de programação, o padrão CORBA define uma linguagem de declaração de interfaces denominada *Interface Definition Language* (IDL), utilizada para descrever as classes de objetos distribuídos utilizados por uma aplicação. O padrão especifica como é realizado o mapeamento dos elementos em uma IDL para diversas linguagens de programação como C, C++ e Java.

Apesar da IDL permitir a declaração de atributos em uma interface, eles são mapeados para métodos de obtenção (*getters*) e de alteração (*setters*), devido ao modelo de comunicação baseado em RPC adotado pelo padrão CORBA.

As implementações do CORBA normalmente fornecem ferramentas que processam os arquivos IDL e geram objetos nativos de acesso, denominados *stubs*, e esqueletos nativos para a implementação dos serventes.

Os *stubs* de uma determinada classe contêm os métodos especificados na IDL. Porém, a implementação destes métodos consiste em criar uma requisição para a operação invocada em uma determinada instância, despachar esta requisição através do ORB, aguardar e retornar o resultado da operação. Desta forma, o cliente utiliza o objeto, através de um *stub*, da mesma forma que utiliza a implementação do objeto.

Já o esqueleto gerado pelos compiladores de IDL recebe as requisições dos clientes, extraí o método invocado e os argumentos, invoca a implementação, empacota o resultado e retorna ao cliente.

Este método de mapeamento, utilizando *stubs* nos clientes e esqueleto no servente é denominado estático, pois o mapeamento é definido em tempo de compilação. Existe também o método dinâmico de acesso e implementação cujo mapeamento da interface para a linguagem de programação é realizado em tempo de execução.

O CORBA oferece para os clientes meios para que a aplicação obtenha informações sobre a interface dos objetos e crie requisições para invocação de métodos. As requisições podem ser feitas de forma síncrona (a execução da aplicação é interrompida até obter a resposta da requisição) ou assíncrona (a aplicação continua a execução, e a resposta é

obtida posteriormente).

Para o mapeamento dinâmico no servente o CORBA fornece meios para a aplicação servente receber e processar diretamente as requisições recebidas. A aplicação poderá então obter informações sobre a instância, o método e os parâmetros da invocação e realizar o processamento adequado. Os métodos estáticos e dinâmicos podem ser utilizados simultaneamente no mesmo objeto, e os clientes e servente podem utilizar métodos diferentes, conforme mostra a figura 3.3.

Figura 3.3: Interfaces de Requisição.

Fonte: [21].

O CORBA possibilita que a aplicação ajuste o funcionamento interno do ORB em diversos níveis (ORB, objetos, requisições etc) tanto no cliente quanto no servente, através de políticas. Além das políticas definidas no padrão, as implementações e extensões podem introduzir políticas específicas.

Para promover a interoperabilidade entre implementações do CORBA, o OMG definiu diversos protocolos de comunicação entre ORBs. Estes protocolos são baseados na especificação abstrata *General Inter-ORB Protocol* (GIOP). Alguns dos protocolos concretos são: IIOP – que utiliza diretamente o TCP/IP; SSLIOP – utiliza autenticação e encriptação através do SSL; HTIOP – sobre o HTTP; e ZIOP – versão com compressão. A figura 3.4 ilustra a arquitetura a interoperabilidade entre ORBs.

Figura 3.4: Interoperabilidade entre ORBs.

Fonte: [21].

O OMG também padroniza diversos serviços e extensões do CORBA, entre os quais, Serviços de Nomeação, Gerenciamento de Tempo, Serviço de Persistência, Serviço de Distribuição de Eventos, Serviço de Segurança. Um exemplo de extensão padronizada é o CORBA para tempo-real [22], voltada para sistemas embarcados e de tempo-real, introduzindo: políticas específicas para ajustes de parâmetros e disciplinas de controle de qualidade; meios para maior controle de prioridade das requisições; e mecanismos para controlar os *threads* do sistema, e disciplinas de escalonamento.

Apesar do modelo orientado a objetos facilitar a estruturação de muitas classes de aplicações, a interação com os objetos baseado em RPC apresenta algumas desvantagens como o forte acoplamento entre cliente e servente, além de não ser ideal para aplicações com elevado fluxo de dados entre os módulos.

Para minimizar estas deficiências, existe o serviço de eventos, padronizado pelo OMG, que introduz uma classe de objeto do tipo Canal de Eventos, e interfaces para objetos consumidores e fornecedores de eventos se conectarem a um ou mais canais de eventos, semelhante ao modelo publicador/assinante. A transmissão de um evento pode ser tanto iniciado pelo produtor (*push*) quanto pelo consumidor (*pulling*). Como este serviço é implementado através de objetos distribuídos, a transmissão de um evento para múltiplos consumidores envolve diversas requisições, sendo uma desvantagem quando se deseja difundir muitas informações entre os processos da aplicação.

3.3 Data Distribution Service

O *Data Distribution Service* (DDS) [23, 24, 25] é um padrão definido também pelo OMG para troca de dados em um ambiente distribuído e foi projetado para ser escalável, inte-

roperável, de alta performance e compatível com ambientes embarcados e de tempo-real. Atualmente o DDS é empregado em aplicações de diversas áreas, tais como financeira, aeronáutica, defesa, energia, entre outras.

Este padrão utiliza um modelo publicador/assinante centrado no dado (*Data-centric*), ou seja, a interação entre participantes acontece através do dado em si, em contraste ao modelo de requisição. Neste modelo, o *middleware* é responsável pelo gerenciamento do estado dos dados e pela transmissão das atualizações. Para isso, a infraestrutura tem conhecimento da estrutura dos dados compartilhados entre processos.

O padrão DDS especifica a API que deve ser fornecida pela implementação, denominada *Data-Centric Publish-Subscribe* (DCPS); e o protocolo de comunicação, denominado *Real-Time Publish-Subscribe DDS Interoperability Protocol* [26], que deve ser utilizado para a comunicação entre processos.

O DDS cria um espaço global de dados compartilhados, denominado Domínio, no qual os processos podem enviar ou receber dados conforme mostra a figura 3.5.

Figura 3.5: Espaço Global de Dados do DDS.

Fonte: Autor.

Este espaço é particionado de acordo com o escopo da informação, e os dados compartilhados no espaço global são separados em entidades denominadas objetos.

A estrutura de cada objeto, denominada "Tipo de Dado", é composta por um conjunto de campos de informações que podem ser de tipo simples (número inteiro, número real, caractere, cadeia de caracteres, octetos etc), ou coleções (listas, mapas, vetores etc).

Objetos são agrupados em tópicos de acordo com o tipo de dado (estrutura do objeto) e a afinidade. Um tópico está associado a um único tipo de dado, porém, diversos tópicos podem estar associados com o mesmo tipo.

Em uma analogia com o modelo relacional de banco de dados, os tópicos equivalem às tabelas, os objetos são as linhas, e os campos de informações são as colunas, conforme ilustrado na figura 3.6.

Figura 3.6: Agrupamento de objetos DDS em tópicos.

Fonte: Autor.

Na declaração de um tipo de dado, um ou mais campos podem ser definidos como chave do objeto. Estes campos podem ser utilizados para identificar instâncias únicas dentro de um tópico, possibilitando manipular (assinar ou publicar) instâncias específicas.

Cada modificação em um ou mais campos de uma determinada instância resulta em uma nova versão do objeto, denominada amostra. O DCPS pode ser configurado para manter um histórico destas amostras, permitindo o acesso a participantes que se juntarem posteriormente ao domínio.

Os participantes podem publicar ou assinar tópicos inteiros, ou instâncias específicas. Um participante pode assinar um tópico com filtro de conteúdo. Estes filtros são expressões aplicadas sobre os campos de um determinado tópico.

A interação dos processos com o espaço global de dados ocorre através da API fornecida pelo DCPS. As principais entidades desta API são as seguintes:

- *Participante de Domínio*: É o meio de entrada para o participante interagir com o domínio. Este componente fornece a interface para a criação das outras entidades associadas com o domínio.
- *Tópico*: Representa um tópico no domínio.

- *Tópico Filtrado por Conteúdo*: Tópico virtual cujos objetos são selecionados a partir do filtro de conteúdo.
- *Multi Tópico*: Tópico virtual que combina os objetos de diversos tópicos (inclusive tópicos filtrados por conteúdo).
- *Publicador*: Entidade responsável pela transmissão de dados em um domínio.
- *Escritor de Dados*: Interface para aplicação publicar um dado em um determinado tópico em um determinado domínio. Existe uma especialização desta entidade para cada tipo de dado, permitindo que o *middleware* consiga serializar corretamente a estrutura nativa.
- *Assinante*: Entidade responsável pela recepção de dados em um domínio.
- *Leitor de Dados*: Interface para recepção em um determinado tópico. Pode receber qualquer objeto, ou instâncias específicas (através dos campos marcados como chave). A leitura pode ser iniciada pela aplicação (*pulling*), ou pelo *middleware* (*push*). Assim como o Escritor de Dados, existe uma especialização desta entidade para cada tipo de dado.

O funcionamento destas entidades pode ser ajustado individualmente por meio de políticas de qualidade de serviço que permitem o controle de parâmetros como tamanho do histórico de amostras em um determinado tópico, volatilidade das amostras (contexto de persistência das amostras), garantias de entrega, limites de tempo de entrega, entre outros.

Para facilitar a utilização do DCPS, o OMG define em um padrão separado, uma camada do software de mapeamento de dados para objetos nativos. Esta camada, denominada *Data Local Reconstruction Layer* (DLRL) [27], fica acima da camada DCPS (Figura 3.7), e tem o propósito de permitir a manipulação dos objetos do DDS através de objetos nativos da linguagem utilizada.

O DLRL faz o mapeamento dos atributos destes objetos nativos, e as relações entre eles para diversos tópicos no domínio do DDS, de forma semelhante ao modelo relacional de banco de dados. Cada instância nativa nos assinantes é uma réplica da instância nativa no publicador de um determinado objeto. Diferente de outros sistemas de objetos distribuídos, no DLRL somente os atributos são distribuídos e as operações são executadas localmente, ou seja, não ocorre uma chamada remota (RPC).

Figura 3.7: Camadas de Software do DDS.

Fonte: Adaptado e traduzido de [27].

Apesar do DDS não suportar o RPC, existem algumas implementações e um esforço para criar um padrão de uma nova camada acima do DCPS para introduzir o modelo requisição resposta. Esta camada mapeia as operações para tópicos dedicados. Uma invocação de operação é então traduzida para publicação e assinatura nos tópicos relacionados.

O DDS é um padrão recente, que atende muitas classes de aplicações, incluindo as críticas e de tempo real. O modelo de objetos é ideal para aplicações com grande fluxo de distribuição de informação.

No entanto, algumas aplicações, como as de Realidade Virtual, além da necessidade da distribuição de dados, também necessitam se comunicar de forma síncrona em algumas situações, ou precisam fazer uma requisição de processamento remoto. Apesar do esforço de padronização de um mecanismo de RPC, o modelo de objetos original não foi projetado para acomodar esta demanda e, além disso, o mapeamento de operações para tópicos resulta em um *overhead* maior no *middleware*.

3.4 TENA

A *Test and Training Enabling Architecture* (TENA) [28, 29, 30, 31] foi projetada para suportar o desenvolvimento rápido, confiável, descentralizado e colaborativo, de aplicações em ambientes distribuídos, de tempo-real, embarcado, de alta escalabilidade e alta performance [30].

O principal objetivo deste projeto é fornecer a arquitetura e *middleware* necessários

para facilitar a interoperabilidade em ambientes de testes e treinamento de diversos dispositivos, componentes e simulações do Departamento de Defesa dos Estados Unidos da America (*US DoD*) [29].

A arquitetura bem como a implementação do *middleware* e ferramentas mantidos pelo *TENA Software Development Activity* (TENA-SDA), permitem o desenvolvimento de ambientes virtuais para testes e treinamento compostos por sensores e equipamentos reais, e elementos simulados por software.

A figura 3.8 apresenta a visão geral de uma aplicação utilizando a TENA.

Figura 3.8: Aplicação utilizando a TENA.

Fonte: [28].

Esta arquitetura emprega diversos modelos de comunicação, tais como *RPC*, publicador/assinante e *Distributed Shared Memory* (DSM), de uma forma integrada, explorando as vantagens de cada um. Estes modelos são explorados através de mensagens e objetos distribuídos.

As principais entidades do TENA utilizadas pela camada de aplicação são: *Stateful Distributed Object* (SDO), Mensagens e Objetos Locais. Os SDOs são estruturas que representam entidades da aplicação que contenham estado próprio e que persistem ao longo da execução da aplicação.

Assim como nas linguagens orientadas a objetos, os SDOs são compostos por atributos e métodos, e são descritos na forma de classes. Os atributos e métodos de um SDO são acessíveis por todos os processos da aplicação como se fossem locais. Internamente o *middleware* do TENA utiliza o modelo RPC para distribuir as chamadas de métodos de um SDO, e o modelo publicador/assinante para transmitir as atualizações dos atributos.

Semelhante ao CORBA, uma classe de SDO é composta por um componente servente e um componente *proxy*. O servente contém a implementação dos métodos da classe, e fornece uma interface para aplicação atualizar (publicar) os atributos da instância.

Desse modo, cada SDO instanciado está associado com apenas uma instância do servente, ou seja, o processo que contém um determinado servente irá executar os métodos invocados remotamente e será o publicador dos atributos do SDO associado.

O componente *Proxy* (nome proveniente do padrão de projeto [32] empregado) provê a interface, com métodos e atributos, para os demais processos da aplicação que interagem com o SDO associado. Por meio desta interface, os processos clientes podem invocar métodos e acessar os atributos de forma transparente, isto é, como se o objeto fosse local.

Desta forma, para cada SDO podem existir diversas instâncias do *proxy* que internamente transformam as chamadas de métodos em requisições para o servente (RPC), e mantêm uma réplica local dos atributos do SDO. Quando o servente publica uma atualização de atributos, os *proxies* (assinantes) atualizam a réplica local. A figura 3.9 ilustra a interação entre servente e *proxy* para um determinado SDO.

Figura 3.9: Interação entre Servente e *Proxy*.

Fonte: [28].

Outra forma de propagação de dados no TENA é por meio de Mensagens, definidas como estruturas compostas por campos de dados descritas por classes, e são distribuídas utilizando o modelo publicador/assinante. Diferentemente dos SDOs, que representam entidades com estados que persistem ao longo da execução, as mensagens representam eventos com significado apenas no instante no qual foram gerados.

Os Objetos Locais também são estruturas com métodos e atributos, porém, diferente do SDO, existem apenas no escopo do processo, e não se comunicam com instâncias remotas. Os métodos são executados e os atributos atualizados localmente.

O Objeto Local tem o mesmo comportamento que um objeto nativo da linguagem de programação, a diferença é que é possível transmitir uma instância "por valor", ou seja, um objeto local pode ser usado como parâmetro de métodos remotos, atributos de SDOs ou parâmetros de Mensagens. Nesta situação será criada uma instância clone no escopo de destino.

Os SDOs, Objetos Locais e Mensagens são descritos na linguagem *TENA Definition Language* (TDL), baseada na IDL do CORBA. Ferramentas processam as definições de classes no TDL e geram classes nativas para os *Proxys*, serventes, Objetos Locais e Mensagens.

Estas classes geradas abstraem os detalhes das requisições de chamadas, transmissão e recebimento de parâmetros e gerenciamento do *cache* local de atributos. A ferramenta de processamento do TDL faz verificações de compatibilidade de tipos de dados em tempo de geração, evitando a propagação de erros e reduzindo o *overhead* em tempo de execução.

Toda a comunicação e implementação dos elementos da arquitetura são implementados no *middleware*. O TENA também contém um repositório de utilidades (Repositório TENA) que não pertencem ao domínio específico da aplicação, mas que auxiliam o desenvolvimento, reduzindo tempo e custo através da reutilização.

Definições padronizadas de interfaces de objetos e mensagens, componentes (implementações de SDOs, Objetos Locais e Mensagens) de uso genérico, e ferramentas fazem parte deste repositório.

A arquitetura também prevê um elemento, denominado *Local Range Data Archive*, para armazenamento e fornecimento de dados específicos da aplicação. A implementação deste elemento não é especificada, podendo ser a que melhor se adequar à aplicação.

A arquitetura TENA foi projetada para aplicações de testes e treinamento, integrando simulações por software e equipamentos reais. O principal diferencial é reunir em um

único elemento do modelo de objeto os padrões de comunicação por publicador/assinante e RPC, somando as vantagens de cada um, e compensando as desvantagens.

Como desvantagem da arquitetura TENA pode-se citar que o SDO é a menor granularidade alcançada nesta arquitetura, ou seja, um SDO está associado a um único servente, em um único processo. Este modelo de objeto atende topologias de objetos planas, ou seja, com poucas composições e agregações e pouco acessos indiretos (invocar um método de um objeto através de outro objeto).

Nesta situação, geralmente, as classes de objetos representam elementos reais do domínio da aplicação. Topologias de objetos mais complexas, com composições, agregações e encapsulamentos, não são adequadas nesta arquitetura apresentando elevado *overhead* devido aos acessos indiretos, e, conseqüentemente, baixo desempenho e escalabilidade.

3.5 Comparação Entre *Middlewares*

A tabela 3.1 apresenta uma comparação sistemática entre os *middlewares* apresentados neste capítulo. A análise comparativa foi realizada sobre os seguintes aspectos:

- *Modelo de Objetos*: As principais entidades arquiteturais do *middleware* utilizados pela camada de aplicação para estruturar os dados e as interações entre os processos participantes.
- *Membros*: Elementos que compõem as entidades principais do modelo de objetos.
- *Acoplamento*: Grau de dependência entre os elementos arquiteturais e participantes.
- *Granularidade*: Menor elemento que pode ser distribuído.
- *Comunicação*: Modelo de comunicação utilizado.
- *Topologia Lógica*: Modelo de arquitetura de conexão entre processos do ponto de vista da aplicação.
- *Topologia Física*: Modelo de arquitetura de conexão entre processos implementado no baixo nível.
- *Protocolo*: Protocolo de comunicação entre processos utilizado.

¹Abreviação de Publicador/Assinante.

Tabela 3.1: Comparação entre os principais *middlewares*.

	HLA	CORBA	DDS	TENA
Modelo de Objetos	·Objetos ·Interações	Objetos	Objetos (agrupados por tópicos)	·SDO ·Objetos Locais ·Mensagens
Membros	·Atributos ·Parâmetros de Interação	Métodos	Atributos	·Atributos e Métodos ·Parâmetros de Mensagens
Acoplamento	Fraco	Forte	Fraco	Híbrido
Granularidade	·Atributo ·Interações	Objeto	Objeto	·Objeto ·Mensagem
Comunicação	Pub./Ass. ¹	RPC	Pub/Ass. ¹	·RPC ·Pub./Ass. ¹
Topologia Lógica	Barramento	Ponto a Ponto	Barramento	Ponto a Ponto
Topologia Física	Geralmente Cliente/Servidor	ORB	·Ponto a Ponto · <i>Multicast</i>	ORB
Protocolo	Não especificado	Baseado no GIOP	DDSI-RTPS	Utiliza o CORBA internamente

3.6 Síntese

Este capítulo apresentou um estudo sobre os principais *middlewares* para aplicações distribuídas, apresentando a visão geral e detalhes da arquitetura, modelos de objetos e de comunicação.

Os próximos capítulos são dedicados à elaboração de uma proposta de modelo de objetos e arquitetura de software para um novo *middleware* para aplicações distribuídas de realidade virtual.

4 PROPOSTA DE UM NOVO MODELO DE OBJETOS PARA APLICAÇÕES DISTRIBUÍDAS

O capítulo 3 apresentou os principais *middlewares* e arquiteturas disponíveis. Apesar de serem alternativas maduras e amplamente empregadas em diversos tipos de aplicações distribuídas, os modelos de objetos e comunicação não são adequados para algumas classes de aplicações de alta complexidade, como as de realidade virtual, tornando seu uso difícil, custoso e pouco eficiente.

Nesse sentido, os próximos capítulos são dedicados na apresentação dos detalhes de um novo *middleware* para aplicações distribuídas de realidade virtual. A elaboração desta proposta foi guiada pelas diretrizes gerais apresentadas na seção 2.3. Este capítulo apresenta o modelo conceitual de objetos do *middleware*, enquanto que os detalhes da arquitetura de software são descritos no capítulo 5.

4.1 Visão Geral

O *middleware* proposto foi projetado para facilitar o desenvolvimento de aplicações distribuídas de realidade virtual. As principais entidades do modelo de objetos são Participante e Objeto Distribuído (OD).

Participantes são linhas de execução que realizam uma tarefa específica e que cooperam entre si, constituindo a aplicação final. Eles contêm todo o código em execução da aplicação, e podem estar localizado em qualquer parte da rede de dados. Na prática, os Participantes são os processos que compõem a aplicação. Se forem bem projetados, os Participantes podem ser reutilizados em outras aplicações.

Os Objetos Distribuídos são entidades abstratas que descrevem os elementos da aplicação, seguindo o princípio do encapsulamento. No caso de aplicações de realidade virtual os ODs podem representar tanto os elementos simulados, como os objetos que constituem o mundo virtual, quanto elementos internos da aplicação, como dispositivos de interação,

componentes do sistema gráfico e de áudio, entre outros.

Análogo ao paradigma de programação orientada à objetos, os ODs possuem estado interno e comportamento. O estado interno é representado através de um conjunto de atributos do objeto, enquanto que o seu comportamento é decorrente da execução interna (*i.e.* ocorre de forma espontânea), e/ou de operações iniciados por participantes, através de métodos.

Os aspectos comportamentais e os estados de um OD são providos por um ou mais Participantes, e acessado por outros. Desta forma a interação entre Participantes ocorre de forma indireta, por meio dos ODs da aplicação. Esta característica reduz o acoplamento entre Participantes, resultando em maior flexibilidade e escalabilidade do *middleware*, e permite a fácil reutilização de Participantes em outras aplicações.

O conjunto de ODs e Participantes definem uma Aplicação Distribuída que atua como uma camada lógica sobre a rede física, conforme mostra a figura 4.1.

Figura 4.1: Visão Geral do Modelo de Objetos.

Fonte: Autor.

Esta arquitetura lógica, semelhante a um barramento de dados, possibilita que os processos Participantes comuniquem-se entre si independentemente de sua localização física. Desta forma, os Participantes podem ser livremente distribuídos nos computadores da rede física sem afetar o funcionamento da aplicação.

A Aplicação Distribuída é mantida pelo Gerenciador de Aplicação que coordena a comunicação entre Participantes, o tempo de vida e localização dos ODs e a execução da aplicação. Apesar do Gerenciador de Aplicação centralizar todo o gerenciamento, do ponto de vista arquitetural; a implementação deste componente pode ser distribuída, sem a necessidade de um processo central.

O modelo de objetos proposto utiliza um modelo de comunicação híbrido baseado no padrão publicador/assinante e RPC, dependendo do dado e da operação realizada. Os Participantes utilizam o Gerenciador de Aplicação para declarar classes de ODs e quais elementos irá fornecer ou utilizar.

4.2 Objetos Distribuídos

Objetos Distribuídos (ODs) são entidades dotadas de estado interno e comportamento, os quais são acessados pelos Participantes por meio dos elementos membros dos objetos, que podem ser atributos, métodos ou eventos. Com exceção dos eventos, os atributos e métodos são conceitos semelhantes aos encontrados nas linguagens orientadas a objetos.

Um atributo é um campo no Objeto Distribuído que contém um dado específico persistente ao longo do tempo, durante o tempo de vida do objeto ou até que um Participante o altere. O conjunto de atributos representa o estado interno de um Objeto Distribuído em um dado instante de tempo.

Qualquer Participante de uma aplicação pode ler o conteúdo de qualquer atributo, utilizando-o como entrada para seu próprio processamento. Porém, somente um Participante (declarado como publicador do atributo) pode modificar o valor deste atributo.

Os métodos são membros que fornecem um meio para os Participantes invocarem um processamento específico (sub-rotina) em outro Participante, dentro do contexto de um Objeto Distribuído. Conceitualmente, uma chamada de método representa uma mensagem destinada ao Objeto Distribuído com significado apenas no instante em que foi enviada (*i.e.* o método foi invocado). Os métodos estão associados ao aspecto comportamental do Objeto decorrente da interação com outro Participante (o invocador).

Durante a invocação, o método pode receber argumentos e pode retornar um valor no final da execução da sub-rotina. Assim como os atributos, vários Participantes podem invocar um determinado método, porém, somente um será o receptor da mensagem e executor da sub-rotina.

Eventos são mensagens não persistentes enviadas pelos ODs para os Participantes e com um significado apenas no instante em que foi enviada. Assim como os métodos, os eventos estão associados ao aspecto comportamental do Objeto Distribuído, porém, no sentido contrário, ou seja, do objeto para os Participantes.

Diferente dos atributos e métodos, qualquer Participante pode enviar ou receber um determinado evento (relação muitos para muitos). Os eventos podem conter argumentos, porém, por representar a comunicação de uma via, não existe retorno associado.

A granularidade de distribuição do Objeto Distribuído é o membro, ou seja, o estado e o comportamento de um Objeto Distribuído não precisa estar em um único Participante. Os membros podem ser distribuídos em diversos Participantes. Esta característica implica em maior flexibilidade para o projeto da aplicação, pois diferentes aspectos podem ser fornecidos por diferentes Participantes. Desta forma é possível existirem objetos complexos sem sobrecarregar um único Participante, ou sem a necessidade de separar em diversos outros objetos, resultando em composições desnecessárias e chamadas indiretas.

Toda a aplicação distribuída deve ser modelada por meio de Objetos Distribuídos. Por exemplo, uma simulação em tempo real de um campo de batalha pode conter objetos distribuídos para veículos, soldados, armamentos, equipamentos e elementos do ambiente. Um Objeto Distribuído que representa um veículo terrestre pode conter atributos para posição, orientação e velocidade; métodos para partida do motor e ligar e desligar faróis; e eventos para sinalizar o disparo de armas.

Estes membros podem estar distribuídos da seguinte forma: Os atributos são publicados por um Participante responsável pelos cálculos de física; o método de partida está associado a uma sub-rotina em outro participante responsável pela simulação de um motor a combustão; e o evento de disparo associado à um terceiro Participante responsável pela simulação bélica. Cada um destes Participantes não tem conhecimento dos demais. Os Participantes utilizadores interagem com um único objeto, sem ter relação direta com os Participantes provedores.

Os Objetos Distribuídos pertencem a uma determinada classe (ou tipo). Isso permite que os ODs sejam categorizados, provendo mais flexibilidade no acesso de atributos, métodos e eventos (mais detalhes na seção 4.4).

Uma classe pode ser derivada de outra classe (herança), estabelecendo uma relação hierárquica entre eles. No exemplo da simulação de campo de batalha, os Objetos Distribuídos das classes "Aeronave", "Tanque", "Embarcações" e "Carro" podem ser herdeiras da classe "Veículo".

Os Objetos Distribuídos também são organizados em hierarquia, na qual cada objeto é associado a um objeto pai e pode ter zero ou mais objetos filho, conforme ilustrado na figura 4.2.

Figura 4.2: Estrutura dos Objetos Distribuídos.

Fonte: Autor.

Esta organização pode ser utilizada para agrupar Objetos Distribuídos de acordo com critérios específicos, ou até mesmo criar composições sem a necessidade de utilizar atributos ou métodos para referenciar outros objetos.

No caso do exemplo do simulador de campo de batalha, diversos ODs que representam soldados podem ser agrupados sob um Objeto Infantaria. Ou alguns mísseis serem filho de um objeto do tipo "Avião Caça". Neste último caso, quando um míssil for disparado, este será desacoplado do objeto pai.

4.3 Participante

Participantes são as unidades de execução da aplicação distribuída que são executados em paralelo, cada um processando uma fração específica da aplicação. Neste modelo de objetos, os Participantes implementam todo o código executável da aplicação, e são representados por processos no sistema operacional.

A troca de dados entre Participantes ocorre somente por meio de interações com os Objetos Distribuídos, não havendo acoplamento entre eles. Este desacoplamento eleva

a flexibilidade do modelo de objetos proposto, permitindo que Participantes sejam facilmente adicionados, substituídos ou removidos em tempo de compilação, carregamento ou de execução da aplicação.

Por exemplo, um simulador de voo pode ser composto por um Participante que realiza a computação do modelo aerodinâmico da aeronave, um ou mais Participantes para a simulação de outros sistemas da aeronave, Participantes para interface com hardware dos controles e instrumentos de voo, Participantes para geração de imagens em tempo real (imagens do mundo virtual), um Participante para síntese de áudio, e um Participante para o console do instrutor.

O desacoplamento entre Participantes possibilita a substituição do modelo aerodinâmico, ou até mesmo do gerador de imagens sem a necessidade de alterar os demais Participantes, desde que as associações com membros dos Objetos Distribuídos se mantenham.

Conforme mencionado na seção 4.2, Participantes podem fornecer e utilizar atributos, métodos e eventos de qualquer Objeto Distribuído da aplicação. Durante a execução, os Participantes atualizam e acessam atributos, atendem e solicitam a invocação de métodos, e disparam e recebem eventos. Desta forma, os membros de um Objeto Distribuído podem ser distribuídos entre vários Participantes, e um Participante pode prover membros de diversos ODs, conforme ilustra a figura 4.3.

Para interagir com os Objetos Distribuídos que compõem a Aplicação, o Participante declara por meio do Gerenciador de Aplicação suas intenções de envios (atualização de atributos, invocação de métodos e disparo de eventos) e recebimentos (acesso aos atributos, recebimento de invocações de métodos e de eventos). Os detalhes das declarações e modelos de comunicação empregados são abordados na seção 4.4.

4.4 Modelo de Comunicação e Declarações

O *middleware* proposto emprega um modelo de comunicação híbrido composto pelos padrões Publicador/Assinante e RPC.

O padrão Publicador/Assinante é utilizado para a distribuição das atualizações dos atributos e disparo de evento. Os publicadores de um determinado membro são os Participantes que atualizam o conteúdo ou disparam eventos, dependendo do membro publicado. Assinantes são Participantes que recebem as atualizações de atributos e eventos.

Figura 4.3: Exemplo de relação entre Participantes e Objetos Distribuídos.

Fonte: Autor.

O padrão RPC é empregado na invocação de métodos. Cada método terá um Participante que irá receber as requisições. Os demais participantes podem fazer as requisições. Esta abordagem híbrida explora as vantagens dos dois modelos, e compensa as desvantagens.

Antes de fornecer ou utilizar um determinado membro de um determinado Objeto Distribuído, o Participante deve declarar sua intenção para o Gerenciador de Aplicação, que mantém internamente um registro dos mapeamentos entre Participantes e membros de ODs e o papel de cada um (provedor ou utilizador).

O modelo de comunicação proposto possibilita diversas formas para a declaração de intenções. A declaração de intenção de fornecer ocorre no nível dos membros de um Objeto Distribuído (*i.e.* atributos, métodos e eventos). Quando um Participante declara a intenção de fornecer um novo membro, automaticamente este membro é incorporado à estrutura do Objeto Distribuído associado.

As intenções de utilização podem ser declaradas nos níveis dos membros, no nível dos Objetos Distribuídos ou no nível das classes de objetos.

No primeiro caso o Participante declara a intenção de uso de um membro específico de um Objeto Distribuído específico.

No nível dos ODs, o Participante declara a intenção de uso de um determinado objeto como um todo, incluindo os objetos filhos. Neste caso o Participante será registrado automaticamente como utilizador de todos os membros existentes e os que forem criados posteriormente.

Na declaração no nível de classe implica no Participante ser registrado como utilizador de todos os membros de todos os objetos da classe declarada e suas sub-classes. As diferentes formas de declaração tornam o modelo de comunicação mais flexível, e facilita casos nos quais um Participante necessita utilizar informações sobre um número arbitrário de objetos de uma determinada classe.

4.5 Gerenciador de Aplicação Distribuída

O Gerenciador de Aplicação é o componente que gerencia a execução da aplicação distribuída e abstrai dos Participantes os detalhes de baixo nível, proporcionando a camada lógica acima da rede física.

Este componente fornece os seguintes serviços para os processos Participantes:

- *Gerenciamento dos Participantes:* Interface para os processos se unirem ou deixarem a Aplicação Distribuída.
- *Registro Classes de Objetos Distribuídos e tipos de dados:* Mantém o registro das classes existentes na Aplicação Distribuída e os detalhes de suas estruturas, e fornece meios para os Participantes registrarem novos tipos e consultarem o registro atual.
- *Gerenciamento das Instâncias de Objetos Distribuídos:* Mantém o registro dos ODs da aplicação e fornece interfaces para os Participantes registrarem novas instâncias, declararem a relação com membros (publicador ou assinante), e consultarem o registro.
- *Acesso aos Membros dos Objetos Distribuídos:* Fornece meios para interação com os membros dos ODs, sendo utilizado para envio e recebimento de atualizações de atributos e eventos, e invocações de métodos.
- *Gerenciamento dos Canais de Comunicação entre Participantes:* Gerencia a comunicação baixo nível entre Participantes, e a associação dos membros utilizados por um processo um ou mais canais.

Apesar do Gerenciador de Aplicação ser um componente central na perspectiva arquitetural, sua implementação poderá ser também distribuída e descentralizada, evitando gargalos de escalabilidade e pontos de falha.

4.6 Comparação

A tabela 4.1 compara a proposta com as alternativas estudadas no capítulo 3 utilizando o conjunto de aspectos descritos na seção 3.5.

4.7 Síntese

Este capítulo apresentou a proposta de um novo modelo de objetos para aplicações distribuídas de realidade virtual que utiliza uma camada lógica para abstrair a localidade de cada processo Participante e os detalhes da topologia física da aplicação. O elemento central do modelo de objetos são os Objetos Distribuídos, compostos por atributos, métodos e eventos. Os Objetos Distribuídos são utilizados para desacoplar os Participantes.

O próximo capítulo descreve a arquitetura de software com detalhes estruturais e dinâmicos do *middleware* proposto.

¹Abreviação de Publicador/Assinante.

Tabela 4.1: Comparação entre a proposta e os principais *middlewares*.

	HLA	CORBA	DDS	TENA	Proposta
Modelo de Objetos	·Objetos ·Interações	Objetos	Objetos (agrupados por tópicos)	·SDO ·Objetos Locais ·Mensagens	Objetos Distribuídos
Membros	·Atributos ·Parâmetros de Interação	Métodos	Atributos	·Atributos e Métodos ·Parâmetros de Mensagens	Atributos, Métodos e Eventos
Acoplamento	Fraco	Forte	Fraco	Híbrido	Híbrido
Granularidade	·Atributo ·Interações	Objeto	Objeto	·Objeto ·Mensagem	Membros
Comunicação	Pub./Ass. ¹	RPC	Pub/Ass. ¹	·RPC ·Pub./Ass. ¹	·RPC ·Pub./Ass. ¹
Topologia Lógica	Barramento	Ponto a Ponto	Barramento	Ponto a Ponto	Barramento
Topologia Física	Geralmente Cliente/Servidor	ORB	·Ponto a Ponto · <i>Multicast</i>	ORB	Flexível
Protocolo	Não especificado	Baseado no GIOP	DDSI-RTPS	Utiliza o CORBA internamente	Flexível

5 ARQUITETURA DE SOFTWARE

O capítulo 4 apresentou um novo modelo de objetos a ser utilizado pelo *middleware* proposto. Com base neste modelo, este capítulo apresenta a arquitetura de software com detalhes estruturais e dinâmicos do *middleware*. Assim como o modelo de objetos, o projeto desta arquitetura foi direcionado pelas diretrizes apresentadas na seção 2.3.

O *middleware* consiste em uma camada de desacoplamento entre o código específico da aplicação e o meio de comunicação, e concretiza o modelo de objetos apresentado no capítulo anterior. Para atender diversos tipos de aplicações distribuídas de realidade virtual, a arquitetura proposta do *middleware* é flexível e possibilita a customização dos protocolos, meios de comunicação e modelos de execução (modelo de *threading*).

5.1 Visão Geral

Do ponto de vista da arquitetura, os Participantes são elementos da aplicação executados em processos independentes e implementados em módulos executáveis separados. Estes processos acessam uma Aplicação Distribuída, incluindo os Objetos Distribuídos, através da implementação do Gerenciador de Aplicação, na forma de biblioteca de software.

A arquitetura proposta não impõe a existência de processos centralizadores para gerenciar a Aplicação Distribuída, ficando a cargo da implementação do Gerenciador de Aplicação.

Para maior flexibilidade o Gerenciador de Aplicação é organizado em camadas, conforme ilustrado na figura 5.1. Quanto mais elevado o nível da camada, mais fácil é a utilização do *middleware*, porém, menos customização é permitido.

A camada Núcleo proporciona meios para o código do Participante interagir com os Objetos Distribuídos, estabelecer canais de comunicação com outros Participantes remotos. Uma Aplicação Distribuída deve utilizar, no mínimo, esta camada e um ou mais canais de comunicação.

Figura 5.1: Camadas de software no Participante.

Fonte: Autor.

Os Serviços Básicos são um conjunto de Objetos Distribuídos e interfaces que proporcionam maior desacoplamento entre Participantes. Os Utilitários são Classes de Objetos Distribuídos genéricas que podem ser utilizadas por diversos tipos de aplicação.

Em uma Aplicação Distribuída, a comunicação entre Participantes é realizada por meio de diversos Canais de Comunicação, conforme mostra a figura 5.2.

Figura 5.2: Comunicação entre Participantes.

Fonte: Autor.

Os Canais de Comunicação implementam a troca de dados entre processos Participantes de modo que para dois Participante interagirem é necessário estabelecer no mínimo um canal de comunicação entre eles.

Dentro de um Participante, os membros dos ODs utilizados são associados a um ou mais Canais de comunicação. Quando o Participante invoca um método, atualiza um

atributo ou dispara um evento, esta interação é enviada para os Participantes interessados através dos canais associados.

Para suportar diversos protocolos e meios de comunicação, esta camada não faz parte do Gerenciador de Aplicação e deve ser fornecida pelo Participante.

5.2 Camada Núcleo

A camada Núcleo implementa serviços locais do Gerenciador de Aplicação e desacopla o Participante dos detalhes de comunicação e troca de dados na rede. Esta camada fornece uma API básica para declarar tipos de dados, estabelecer canais entre Participantes e manipulação dos Objetos Distribuídos.

Esta camada foi projetada para ser leve, flexível e fornecer aos Participantes elevado grau de controle sobre a topologia e canais de comunicação. Consequentemente a utilização direta desta camada é mais difícil, e requer que os Participantes tenham conhecimento sobre a topologia de rede para poderem estabelecer os canais de comunicação.

O núcleo fornece as seguintes funcionalidades:

- *Tipos*: Registro local das descrições das estruturas de dados e classes de ODs utilizados pelo Participante.
- *Instâncias*: Registro local das instâncias de Objetos Distribuídos utilizados pelo Participante, e as associações dos membros com Canais de Comunicação.
- *Canais*: Meios para o Participante estabelecer canais de comunicação com outros Participantes, definindo a topologia de comunicação da Aplicação Distribuída.

Estas funcionalidades são locais ao Participante, não havendo comunicação entre núcleos de outros Participantes. Ou seja, esta camada não é responsável pela sincronização com os registros de outros Participantes, ficando a cargo do código específico do Participante, ou das outras camadas de software acima do núcleo.

Para interagir com um determinado Objetos Distribuídos através desta camada é necessário:

1. Descrever a estrutura da classe do OD e dos tipos de dados utilizados (parâmetros de métodos, tipo do atributo etc);

2. Registrar a instância do OD e quais membros irá fornecer ou utilizar;
3. Estabelecer os canais de comunicação que irão fornecer ou utilizar os membros que o Participante irá interagir;
4. Associar os canais com os membros do OD.

Somente após estes passos o Participante poderá interagir com o Objeto Distribuído, podendo utilizar ou fornecer os membros registrados no passo 2 e associados com canais no passo 4.

A utilização direta desta camada é destinada para aplicações distribuídas que reque-rem maior controle da topologia de comunicação e execução das interações com ODs. Um exemplo são os simuladores de voo, cujo requisito de tempo-real exige protocolos de comunicação específico, controle das conexões e previsibilidade na execução, garantindo que os prazos de processamento sejam atendidos.

Os componentes da camada núcleo são descritos nas próximas seções.

5.2.1 Tipos

Este componente do núcleo fornece meios para os Participantes declararem a estrutura dos ODs e tipos de dados.

Assim como nas linguagens de programação, os Tipos são entidades que descrevem a estrutura dos Objetos Distribuídos e seus elementos. Todos os dados transferidos entre Participantes devem ser de um Tipo registrado na camada núcleo.

Os tipos podem ser:

- Classes de Objetos Distribuído;
- Primitivo;
- Estrutura de Dados.

As Classes definem os membros que compõem um determinado tipo de Objeto Distribuído. Os tipos Primitivos e Estrutura de Dados são utilizados para descreverem dados associados com os membros de uma Classe, como atributos, parâmetros e retorno de métodos, e parâmetros de eventos.

Os Tipos possuem um nome e podem ser logicamente agrupados em uma hierarquia de escopos. A identificação é definida pelo caminho na hierarquia de escopos e o seu nome, e deve ser único em toda Aplicação Distribuída. Dois ou mais Tipos com o mesmo nome são permitidos, desde que estejam em escopos distintos, ou seja, a identificação é única. Esta organização é análoga aos sistemas de arquivos, no qual os Tipos equivalem aos arquivos, e os escopos aos diretórios.

Os tipos Primitivos são tipos básicos suportados nativamente pelo *middleware*. Fazem parte desta categoria tipos para representação de números inteiros, números de ponto flutuante, cadeias de caracteres e o tipo vazio (tipo *Void*).

As Estruturas de Dados são tipos complexos definidos por conjunto de campos de dados de Tipo Primitivo ou outra Estrutura de Dados. Os campos são identificados por nome e possuem multiplicidade, podendo ser unitários (apenas um elemento), ou sequências fixas (quantidade fixa de elementos) ou dinâmicas (quantidade não definida na declaração).

As Classes definem a estrutura dos Objetos Distribuídos e são compostas pelas declarações de atributos, métodos e eventos. A declaração de um atributo é definido pelo nome, Tipo e multiplicidade.

A declaração de um método contém o nome, o Tipo do valor retornado (pode ser Tipo vazio), e uma lista de declarações de argumento. A declaração de cada argumento contém o Tipo e qualificadores que indicam a direção (entrada, saída ou entrada e saída), e a multiplicidade.

No caso dos Eventos, a declaração é composta por nome e as declarações dos argumentos, com Tipo e multiplexidade.

Assim como nas linguagens de programação orientadas a objetos, as Classes de ODs podem ser derivadas de outras classes, herdando acumulativamente os membros das classes ancestrais.

As declarações de Tipos são utilizadas pela camada núcleo para assegurar a correta estrutura dos Objetos Distribuídos; e, opcionalmente, pelos canais de comunicação para auxiliar a conversão das interações com ODs em mensagens na rede.

Como mencionado na seção 5.2, a camada núcleo não sincroniza as declarações entre Participantes, sendo o software acima desta camada responsável em registrar as declarações de tipos no núcleo e garantir a consistência entre Participantes.

5.2.2 Instâncias

Este módulo é responsável pelo gerenciamento local dos Objetos Distribuídos utilizados pelo Participante e proporciona as seguintes funcionalidades ao Participante:

- Registro de Objetos Distribuídos a serem utilizados;
- Registro dos membros fornecidos (publicador);
- Registro dos membros utilizados (assinante);
- Associação dos membros com Canais de comunicação;
- Interagir com os membros registrados, como invocar e receber invocação de métodos, atualizar e receber atualização de atributos, e disparar e receber eventos;

Como mencionado no capítulo 4, Objetos Distribuídos são instâncias de uma determinada Classe e são identificados por um nome. Antes de registrar um OD, a Classe associada deve estar declarada no núcleo.

Os Objetos Distribuídos podem ter uma relação pai e filho, possibilitando a organização deles em uma hierarquia. Neste caso, a identificação completa de um OD consiste na sequência dos nomes de todos os objetos ancestrais acrescido do nome da instância. A identificação completa deve ser única no escopo da Aplicação Distribuída, ou seja, não podem existir dois Objetos com o mesmo nome e o mesmo Objeto pai.

Após registrar um determinado OD no núcleo, o Participante declara quais membros desta instância irá publicar e assinar, respeitando a estrutura da Classe do OD. A interação com estes membros ocorre através de entidades *Proxy* ou Adaptador, dependendo se o Participante utiliza ou fornece, respectivamente, o membro, conforme ilustrado na figura 5.3.

Os Adaptadores são entidades da arquitetura utilizadas pelo Participante para fornecer a funcionalidade de um determinado membro. Através destes elementos o Participante pode modificar valores de atributos, associar a implementação local à métodos e disparar eventos.

O *Proxy* é o elemento da arquitetura que fornece ao Participante a interface para utilizar membros de um OD. Através dos *Proxies* o Participante pode obter o valor de um atributo, invocar um método ou receber eventos.

Figura 5.3: Interação entre membros e Participante.

Fonte: Autor.

A tabela 5.1 sumariza as possíveis interações entre membros e Participantes através dos Adaptadores e *Proxies*.

Tabela 5.1: Interações com membros.

Membro	Adaptador	<i>Proxy</i>
Atributo	Modifica valores	Obtém valores
Método	Atende invocação	Invoca
Evento	Dispara eventos	Recebe eventos

Os Adaptadores e *Proxies* são associados com Canais de comunicação de modo que a semântica fornecedor/utilizador, ou publicador/assinante, seja respeitada. Ou seja, o Canal associado com o *Proxy* de um determinado membro em um Participante deve também estar associado com o Adaptador do mesmo membro no Participante remoto, conforme ilustrado na figura 5.4.

Os Adaptadores podem ser associados com diversos Canais uma vez que o membro associado pode ser utilizado por diversos Participantes. No caso dos *Proxies*, somente os associados a Eventos podem ter múltiplos canais, já que mais de um Participante podem disparar o mesmo evento. Os *Proxies* de método e atributos são associados a um único canal.

Figura 5.4: Relação entre *Proxy*, Adaptador e Canal.

Fonte: Autor.

5.2.3 Canais

Os Canais de comunicação são entidades da arquitetura responsáveis pela troca de dados entre Participantes, abstraindo do restante do núcleo e dos Participantes todos os detalhes baixo nível dos protocolos e meios físico de comunicação. Para atender diversas classes de aplicações distribuídas de realidade virtual, o núcleo não fornece nenhum canal concreto, ficando a cargo da aplicação a implementação de canais que atendam seus requisitos específicos.

As principais responsabilidades de uma implementação de um Canal são:

- Tradução das interações com Adaptadores e *Proxies* em mensagens de rede (Serialização/Deserialização);
- Transmissão das mensagens para o Participante remoto;
- Enviar ao Núcleo as interações recebidas de Participantes remotos;
- Garantir as Qualidades de Serviço requeridas pela aplicação distribuída (prioridades, garantias de entrega, ordenação entre outras).
- Implementar recursos como réplicas ou balanceamento de cargas.

Conforme apresentado, os canais são diretamente associados com os Adaptadores e *Proxies* dos membros dos Objetos Distribuídos. Esta associação é do tipo muitos-para-muitos, ou seja, diversos Adaptadores e *Proxies* podem compartilhar o mesmo canal de comunicação; e os Adaptadores e *Proxies* de Eventos podem estar associados a mais de um canal.

Diversos tipos de canais podem ser utilizados em uma mesma aplicação distribuída, possibilitando que cada membro de cada Objeto Distribuído possa empregar protocolos e políticas de qualidade de serviços específicos.

O *middleware* não impõe ou implementa nenhum modelo específico de execução, ficando a critério do canal adotar o que melhor atende seus requisitos específicos. Desse modo, cada implementação é livre para utilizar filas e *threads* para processar as interações com membros em ambos os sentidos (do Participante local para o remoto e do remoto para o local).

Apesar de não serem parte do núcleo, implementações de canais podem ser reutilizadas em diversas Aplicações Distribuídas, por exemplo, sendo distribuídas como bibliotecas de canais.

5.2.4 Dinâmica das Interações com Membros

Durante a execução de uma Aplicação Distribuída, a comunicação entre Participantes ocorre indiretamente através da interação com membros de Objetos Distribuídos utilizando Adaptadores e *Proxies*. Estas entidades utilizam Canais de comunicação para transmitir ou receber estas interações através de uma rede de dados. A figura 5.5 mostra o caminho percorrido por uma interação através das entidades da arquitetura.

Figura 5.5: Visão geral da dinâmica das interações.

Fonte: Autor.

A dinâmica da interação é diferente para cada tipo de membro de um OD devido à semântica e o modelo de comunicação empregado por cada um. O atributo utiliza o modelo de comunicação publicador/assinante, conforme mostra a figura 5.6.

Figura 5.6: Diagrama de sequência de modificação de valores de atributos.

Fonte: Autor.

Para modificar o valor de um atributo, o Participante publicador utiliza o Adaptador associado que irá encaminhar o novo valor para os Canais assinantes, os quais transformarão a interação em mensagens de rede e enviar para os Participantes remotos. Ao receber esta interação, o canal no Participante assinante irá decodificar a mensagem e encaminhar para o *Proxy* associado, que, por sua vez, armazenará o novo valor internamente. O Participante assinante poderá então obter o novo valor através do *Proxy*.

Os Eventos também seguem o modelo de comunicação Publicador/Assinante, conforme o diagrama na figura 5.7.

Figura 5.7: Diagrama de sequência de disparo de eventos.

Fonte: Autor.

A dinâmica de interação dos Eventos é semelhante ao atributo. As diferenças são

que Eventos podem ser publicados por múltiplos Participantes, e o *Proxy* não armazena localmente o evento disparado, mas sim notifica o Participante assinante.

A dinâmica de interação com métodos segue o modelo de comunicação Requisição/-Resposta, no qual o Participante invoca o método por meio do *Proxy* que, por sua vez, encaminha para o Canal associado, conforme mostra a figura 5.8.

Figura 5.8: Diagrama de sequência da invocação de métodos.

Fonte: Autor.

O Canal codifica a interação e envia pela rede ao Canal receptor que decodifica a invocação de método e encaminha para o Adaptador associado, que invoca a implementação concreta do método no Participante. Após concluída a execução, o Adaptador envia o resultado ao Participante requerente fazendo o caminho inverso da invocação.

Cada invocação de método está associada com um identificador único no Participante invocador que é utilizado para realizar o caminho de retorno em Participantes *multi-threads*, possibilitando a localização do *thread* invocador.

Como a implementação do Canal não faz parte do *middleware* proposto, partes das sequências podem acontecer de forma assíncrona e em *threads* separados.

5.3 Camada Serviços Básicos

Conforme apresentado nas seções anteriores, a camada núcleo fornece a funcionalidade básica do modelo de objetos e arquitetura propostos, proporcionando o desacoplamento mínimo entre o Participante e os detalhes de comunicação. Porém, a consistência dos Tipos e Instâncias de OD e o estabelecimento dos Canais é de responsabilidade das camadas

acima do núcleo. Esta abordagem possibilita maior controle para os Participantes, em troca de mais complexidade na utilização do *middleware*.

Como a maioria das aplicações distribuídas de realidade virtual não necessitam desse controle, a arquitetura fornece uma camada opcional com serviços básicos para facilitar o preparo e a execução da Aplicação Distribuída. Esta camada sincroniza os registros de Tipos e Instâncias entre Participantes, abstrai a localização dos fornecedores e utilizadores dos membros dos ODs, e abstrai a configuração dos canais de comunicação. Desta forma, esta camada proporciona um desacoplamento ainda maior entre Participantes.

Essa camada fornece os seguintes serviços:

- Registro de Tipos;
- Localização de Objetos;
- Gerenciamento de Canais.

5.3.1 Registro de Tipos

O propósito deste serviço é sincronizar a declaração de Tipos entre os Participantes de uma Aplicação Distribuída, garantindo a consistência das declarações.

A implementação deste serviço pode ser transparente ao Participante, interagindo diretamente com o núcleo para obter as declarações locais e enviar aos Participantes remotos, e inserir no registro local (núcleo) os Tipos declarados remotamente.

5.3.2 Localização de Objetos

O serviço de localização de objetos oferece meios para o Participante localizar membros de instâncias de ODs sem a necessidade de conhecer a localização dos Adaptadores, retornando *Proxies* preparados para serem utilizados.

Quando um determinado membro é solicitado, a implementação deste serviço irá verificar se existe um *Proxy* para este membro na camada núcleo. Caso não exista, o serviço irá determinar qual o Participante e quais tipos de canais estão associados ao Adaptador deste membro. Estas informações são utilizadas junto ao Gerenciador de Canais para localizar canais já existentes ou estabelecer novos canais com o Adaptador. Estes canais são associados ao *Proxy* do membro solicitado. Este processo é ilustrado na figura 5.9.

Figura 5.9: Diagrama de Sequência da Localização de Membros.

Fonte: Autor.

5.3.3 Gerenciador de Canais

A finalidade deste serviço é automatizar a criação de canais entre Participantes, contribuindo para maior desacoplamento entre o Participante e os detalhes de comunicação.

Os canais são estabelecidos com base em descritores. Estes elementos são cadeias de caracteres contendo o tipo do canal e parâmetros específicos da instância do canal, tais como endereço de rede do Participante remoto, número da porta, entre outros.

Como a implementação dos canais não faz parte do *middleware*, este serviço utiliza o padrão de projeto *Factory* [32] para construir canais, e fornece interfaces para registrar construtores para cada tipo de canal suportado pela Aplicação Distribuída.

O Gerenciador de Canais é utilizado pelo serviço de Localização de Objetos para obter ou criar Canais com os Adaptadores dos membros solicitados. Neste processo, o Localizador de Objetos é responsável em obter descritores de canais para os Adaptadores, e o Gerenciador de Objetos em estabelecer estes canais.

5.4 Síntese

Este capítulo apresentou a arquitetura de software de um novo *middleware* para aplicações distribuídas de realidade virtual. Esta arquitetura é dividida em camada núcleo e serviços básicos.

O núcleo implementa as entidades básicas do modelo de objetos proposto no capítulo 4, e os mecanismos básicos de interação entre Participantes de uma Aplicação Distribuída. Apesar da utilização direta da camada de núcleo ser mais trabalhosa e não oferecer um desacoplamento total entre Participantes, essa camada oferece alto grau de controle dos meios de comunicação e modelos de execução, possibilitando a utilização de múltiplos protocolos de rede em uma mesma Aplicação Distribuída.

A camada de Serviços Básicos é de uso opcional e oferece maior facilidade na utilização do *middleware*, e maior desacoplamento entre Participantes, atendendo a maioria das aplicações de realidade virtual.

As principais vantagens da organização em multi-camadas e a abstração dos detalhes de comunicação e modelos de execução em Canais são possibilitar à aplicação maior controle, suportar diversos meios e protocolos de comunicação, e suportar diversos modelos de execução, atendendo os requisitos específicos de diversas categorias de aplicações distribuídas de realidade virtual.

6 PROVA DE CONCEITO

Este capítulo tem os objetivos de apresentar os detalhes de implementação de um protótipo da camada núcleo do *middleware* como prova de conceito do modelo de objetos e arquitetura de software apresentados, e principalmente demonstrar a sua viabilidade técnica.

O propósito deste protótipo é implementar os conceitos principais apresentados e a arquitetura básica. Para testar a aplicabilidade da proposta, foram também implementados um protocolo simples de comunicação e um canal que utiliza o transporte TCP/IP para troca de dados entre Participantes. Não fazem parte dos objetivos desta implementação ser completa e utilizável em ambientes de produção.

6.1 Visão Geral

O protótipo é organizado em um conjunto de bibliotecas de software implementadas em linguagem C++14 seguindo o paradigma de orientação a objetos. As bibliotecas do protótipo estão relacionadas de acordo com a figura 6.1, cujo detalhamento é apresentado nas próximas seções.

Além do *middleware*, foram implementados um codificador/decodificador de interações para mensagens de rede, um canal de comunicação local, e um canal de comunicação através do protocolo de transportes TCP/IP.

6.2 Implementação do Núcleo

Esta biblioteca é o principal componente do protótipo e implementa as funcionalidades da camada núcleo, descritas na seção 5.2.

As classes do núcleo estão organizadas nos seguintes subcomponentes:

- *Tipo*: Classes para o gerenciamento local e representação das classes de Objetos

Figura 6.1: Diagrama de pacotes da implementação.

Fonte: Autor.

Distribuídos e tipos de dados utilizados pela Aplicação Distribuída;

- *Instância*: Gerenciamento local dos Objetos Distribuídos, e interfaces de interação com membros;
- *Canal*: Classes base e gerenciamento local dos canais de comunicação;
- *Interação*: Classes internas que representam as interações entre participante e membros;

O software do Participante faz interface apenas com os subcomponentes Tipo, Instância e Canal, enquanto o subcomponente Interação é utilizado internamente e pela implementação dos canais de comunicação.

A interface com o núcleo inicia pela da classe *GerenciadorAplicação* que é a responsável pela inicialização do núcleo e fornece acesso para cada um dos subcomponentes através dos seus respectivos gerenciadores, conforme o diagrama da figura 6.2.

A Aplicação Distribuída e cada Participante são identificados por nomes que deverão ser informados ao *GerenciadorAplicação* durante a inicialização.

Figura 6.2: Diagrama de Classes Gerenciador de Aplicação.

Fonte: Autor.

6.2.1 Tipos

Esse subcomponente fornece classes para descrever a estrutura de classes de Objetos Distribuídos e tipos de dados, e mantém o registro local das declarações. O diagrama de classes deste subcomponente é apresentado na figura 6.3.

A interface para declaração de novos tipos e consulta do registro local é fornecida pela classe *GerenciadorTipo* que pode ser acessada através do *GerenciadorAplicacao*.

A classe *Tipo* representa qualquer tipo declarado e é base para tipos primitivos, estruturas de dados ou classe de OD. Os Tipos são identificados por um nome, e fazem parte de um *Namespace*.

O *TipoPrimitivo* é utilizado para representar tipos nativos do *middleware* como variações de números Inteiros, números de ponto flutuante, cadeia de caracteres, entre outros. Os tipos primitivos são declarados pelo *middleware* e estão disponíveis através de instâncias globais desta classe.

Como visto na seção 5.2.1, tipos de dados complexos são definidos por uma lista de campos que armazenam valores de um determinado tipo. Estas estruturas são declaradas através das classes *TipoEstrutura* e *Campo*. Além do tipo do dado, a classe *Campo* contém informação de multiplicidade, que pode ser:

Figura 6.3: Diagrama de Classes do Gerenciamento de Tipos.

Fonte: Autor.

- 0, para sequências dinâmicas;
- 1, para valor simples;
- n ($n > 1$), para sequências com número fixo de elementos.

A classe *TipoClasse* representa classes de Objetos Distribuídos utilizados pela Aplicação Distribuída. Esta classe possui atributos para representar relação de herança com outras classes, e armazenar informações dos membros por meio das classes *Atributo*, *Metodo* e *Evento*. Os métodos de procura de membros fazem uma busca recursiva iniciando na declaração da classe filha em direção às classes ancestrais.

Os atributos são declarados com a utilização da classe *Atributo* que contém o nome, o Tipo do dado e a multiplicidade.

Eventos são declarados por meio da classe *Evento*, e os detalhes dos argumentos, através de *ArgumentoEvento*.

Instâncias da classe *Metodo* armazenam o nome, Tipo de Retorno e informações dos argumentos, como Tipo, multiplicidade e direção (entrada, saída, ou entrada e saída).

Os Tipos são agrupados em *Namespaces* que, recursivamente, podem ser agrupados em outro *Namespace*. Esta estrutura é representada pela classe *Namespace*.

Os nomes dos Tipos e *Namespaces* devem ser únicos no escopo do *namespace* em que estão contidos. A identificação única, ou nome completo, destas entidades é definida recursivamente como nome completo do *namespace* em que estão contidas, acrescido do separador "::" e o nome. Por exemplo, o nome completo de um tipo "Tipo" que faz parte do *namespace NS*, é "*NS::Tipo*".

Para evitar ambiguidades, somente os nomes completos são utilizados para busca de declarações no registro local através do *GerenciadorTipo*.

6.2.2 Instância

Este subcomponente do núcleo é responsável pelo gerenciamento local dos Objetos Distribuídos e por fornecer meios para o software do Participante interagir com os membros destes objetos, seja como utilizador ou fornecedor. As classes deste subcomponente são apresentadas no diagrama de classes da figura 6.4.

A classe de acesso a este subcomponente é o *GerenciadorInstancia*, que mantém o registro local dos ODs e fornece métodos para acrescentar novas instâncias ou buscar

instâncias já registradas.

Os Objetos Distribuídos são representados por instâncias da classe *Objeto*, e possuem um nome, um conjunto de *Proxies* e Adaptadores, e podem, opcionalmente, referências para ODs pai e filhos, possibilitando o agrupamento em hierarquias. Por meio dessa classe o Participante pode acessar *Proxies* e Adaptadores dos membros que irá utilizar e fornecer, respectivamente.

As classes *Proxy* e *Adaptador* são bases para as implementações específicas dos *Proxies* e Adaptadores de atributos, métodos e eventos; e implementam as associações com os canais de comunicação. As instâncias de cada tipo de *Proxy* e Adaptador estão associadas com a declaração do membro utilizado ou fornecido.

A classe *ProxyAtributo* armazena o valor local de um determinado atributo do OD, fornece interface para o Participante utilizador obter tal valor. Opcionalmente, o Participante utilizador pode associar um elemento Observador (*Callback*) e ser notificado de forma assíncrona quando o valor for atualizado. As atualizações de valores são recebidas através do canal associado, que é responsável em notificar o *ProxyAtributo*. O valor inicial é obtido através de uma requisição ao Participante fornecedor quando um canal é associado ao *Proxy*.

O Adaptador de atributo é implementado pela classe *AdaptadorAtributo* que fornece métodos para o Participante fornecedor modificar o valor. Esta classe então encaminha o novo valor para os canais associados, que irão enviar para os Participantes utilizadores.

A utilização de métodos de um Objeto Distribuído acontece por meio da classe *ProxyMetodo* que fornece meios para invocar o método associado, incluindo a passagem de argumentos e o recebimento do retorno. Os seguintes modelos de invocação são implementados nesta classe:

- *Invocação Síncrona*: O *thread* que inicia a invocação aguarda o retorno do método antes de executar a próxima instrução. Este modelo é semelhante à chamada de métodos nativos.
- *Invocação Assíncrona*: A invocação do método retorna imediatamente e o *thread* invocador continua sua execução. O retorno da invocação é um objeto do tipo *Futuro*¹, que permite o Participante verificar se o retorno do método está disponível, ou até mesmo aguardar o retorno em outro momento da execução.
- *Invocação de Via Única*: Esta chamada consiste somente na requisição, não havendo o retorno do resultado. A invocação retorna imediatamente, e o resultado é

descartado.

A classe *AdaptadorMetodo* associa um método de um Objeto Distribuído à implementação real do método por meio de um *Callback* instalado pelo Participante fornecedor do método. Este *Callback* é chamado pelo *middleware* quando o método é invocado por um Participante remoto. O *Callback* recebe como parâmetro os detalhes da invocação, incluindo os valores dos argumentos, e o canal de origem. Ao final da execução, os valores retornado e dos argumentos de saída, e entrada e saída, são encaminhados de volta para o canal, que é responsável por transmitir ao Participante invocador.

Os eventos são recebidos de forma assíncrona pelos Participantes utilizadores através de *Callbacks* associados ao *ProxyEvento*. Ao receber um evento disparado, o canal encaminha tal interação para os *ProxyEventos* associados, que irão notificar o Participante através do *Callback*.

O disparo de eventos ocorre por meio do *AdaptadorEvento*, que encaminha tal interação para os canais associados.

Internamente os *Proxies* e Adaptadores utilizam estruturas de dados do subcomponente Interação para representar atualização de atributos, invocação de métodos e disparo de eventos. Estas estruturas são utilizadas para encaminhar ou receber as interações dos canais associados.

Como os Objetos Distribuídos são organizados em hierarquia, o nome completo de cada instância é definido como o nome completo do objeto pai acrescido do separador "/" e o nome da instância. Por exemplo, "/obj1/obj2/obj3" é o nome completo do objeto "obj3", cujo pai é o "obj2", e o avô, o "obj1". Os nomes completos são utilizados pelo *middleware* para evitar ambiguidades.

6.2.3 Interação

Este subcomponente, estruturado de acordo com o diagrama na figura 6.5, é constituído por classes que representam as interações entre Participante e membros dos Objetos Distribuídos, e são utilizados na interface entre canais de comunicação e *Proxies* e Adaptadores.

A classe *Valor* é um tipo variante que pode armazenar dados de qualquer tipo primitivo do *middleware*, sequências de valores do mesmo tipo, ou sequências de valores

¹O par Promessa/Futuro é uma construção de programação utilizado para sincronização de dados em aplicações concorrentes. Este par compartilha um valor, que é atribuído pelo *thread* produtor através da Promessa, e acessado pelo *thread* consumidor por meio do Futuro.

Figura 6.5: Diagrama de Classes das Interações.

Fonte: Autor.

de tipos distintos. Esta classe é utilizada para abstrair o tipo dos dados, e é utilizado ao longo do *middleware* para armazenar valores como argumentos de métodos e eventos, atributos, e retorno de métodos.

É possível armazenar sequência de objetos do tipo *Valor*, possibilitando utilizar esta classe para estruturas de dados mais complexas, armazenando os valores de cada Campo como elementos da sequência de *Valores*, respeitando a mesma ordem da declaração.

As implementações de canais de comunicação devem ser capazes de codificar e decodificar objetos do tipo *Valores* em mensagens de rede.

A classe *AtualizacaoAtributo* como o nome sugere, representa a atualização de um determinado atributo, contendo além do novo valor, referências para o Objeto Distribuído e a declaração do atributo.

O *DisparoEvento* é utilizado para Eventos disparados, contendo também o OD, a declaração do Evento associado e a lista dos valores dos argumentos, respeitando a ordem da declaração.

Devido ao modelo requisição/resposta ser utilizado nas invocações de métodos, a representação desta interação está dividida nas classes *InvocacaoMetodo*, *InvocacaoSaida* e *InvocacaoEntrada*.

A classe *InvocacaoMetodo* reúne as informações da interação, incluindo o Objeto Distribuído, a declaração do método associado, a lista de valores dos argumentos e um indicador se o invocador não requer a resposta (modelo via única de invocação). Esta classe é utilizada pelos Participantes utilizadores e fornecedor do método.

O *middleware* no Participante invocador de um método utiliza também a classe *InvocacaoSaida* com informações complementares à invocação, como o canal a ser utilizado, identificador numérico da invocação, e *Promessa* de retorno associada à estrutura *Futuro* retornado ao Participante nas invocações assíncronas.

A classe *InvocacaoSaida* possibilita que o Participante e o *middleware* continuem a execução sem precisar aguardar o retorno da requisição. Durante uma invocação, o *middleware* invocador retém o objeto do tipo *InvocacaoSaida* até o recebimento do resultado. Quando a resposta chega ao canal, este localiza a *InvocacaoSaida* relacionada, e associa os valores retornados ao objeto *Promessa*, disponibilizando ao Participante e, se necessário, retomando a execução do *thread* invocador.

O identificador numérico é utilizado pelo *middleware* ao receber uma resposta de invocação, para localizar o objeto *InvocacaoSaida* pendente relacionado ao retorno. O

mesmo identificador deve ser utilizado pelo *middleware* no Participante fornecedor, sendo de responsabilidade da implementação do canal sua transmissão.

A classe *InvocacaoEntrada* é utilizada pelo *middleware* no Participante fornecedor de um método, e contém os dados da invocação, o identificador numérico, canal de origem, e espaço para reter os valores retornados pelo *Callback* associado.

6.2.4 Canal

O subcomponente Canal mantém o registro local dos canais de comunicação e fornece interfaces para as implementações de canais. As classes deste subcomponente estão relacionadas conforme o diagrama apresentado na figura 6.6.

O *GerenciadorCanais* é responsável pelo registro local dos canais estabelecidos entre Participantes, fornecendo meios para o Participante local adicionar novos canais, ou buscar canais já abertos. A localização de um canal pode ser através do nome atribuído localmente, ou pelo par *<Nome do Participante Remoto, Nome atribuído remotamente ao canal>*.

A classe abstrata *Canal* é utilizada, por meio de herança, pela implementações de canais, contendo atributos e funcionalidades comuns. Canais são identificados localmente através de um nome único, e estão associados com *Proxies* e Adaptadores de membros de ODs, cujas interações podem ser transmitidas por tal canal. Para auxiliar a interação com métodos de Objetos Distribuídos, esta classe também mantém uma lista com as invocações aguardando resposta.

A implementação de um canal deve herdar a classe *Canal*, implementar um conjunto de métodos virtuais para codificar e transmitir interações com membros de ODs iniciadas pelo Participante local, e utilizar métodos privados para encaminhar ao *middleware* as interações recebidas de outros Participantes.

Os métodos virtuais implementados pelo canal específico são:

- *EnviaAtualizacaoAtributo*: Método invocado pelo *middleware* quando um valor de atributo é modificado através do Adaptador de Atributo.
- *EnviaInvocacaoMetodo*: chamado indiretamente pelo *ProxyMetodo* quando o Participante local invoca o método associado.
- *EnviaEvento*: Chamado indiretamente pelo Adaptador de Evento quando um evento é Disparado pelo Participante local.

Figura 6.6: Diagrama de Classes do Subcomponente Canal.

Fonte: Autor.

Estes métodos recebem como argumento estruturas do subcomponente *Interação*, contendo os detalhes da interação realizada, e devem codificá-las e transmiti-las para o Participante remoto.

Ao receber uma interação proveniente de um Participante remoto, a implementação do canal é responsável pela decodificação em objetos do subcomponente *interacao*, e pelo encaminhamento ao *middleware*, que irá despachar para o Participante através de *Proxies* ou Adaptadores. Este despacho é iniciado pela implementação através dos seguintes métodos da classe base:

- *DespachaAtualizacaoAtributo*: Utilizado para despachar atualização de valor de atributo para o *ProxyAtributo* associado.
- *DespachaInvocacao*: Encaminha a invocação de método ao *AdaptadorMetodo* associado. Após o retorno deste método, os valores de resultado estão disponíveis no objeto *InvocacaoEntrada*. A implementação do canal deverá então transmitir estes valores para o Participante invocador.
- *RetornaInvocacao*: Encaminha o resultado de uma invocação para o *ProxyMetodo* invocador.
- *DespachaEvento*: Encaminha um evento recebido ao *ProxyEvento* associado.

Para possibilitar a implementação de diversos modelos de execução, o núcleo não utiliza nenhum *thread* interno, operando de forma síncrona. Dessa forma, a chamada dos métodos virtuais sempre ocorrerá no mesmo *thread* no qual a interação foi iniciada pelo Participante local, e as notificações das interações recebidas, através dos *Callbacks*, seguirão sempre no mesmo *thread* no qual a implementação do canal iniciou o despacho.

No entanto, a implementação do canal é livre para adotar o modelo de execução mais conveniente, podendo empregar diversos *threads* para otimizar as transmissões e despachos de interações, adotar políticas de prioridades, utilizar filas internas, entre outros.

6.3 Protocolos e Canais de Comunicação

O protótipo contempla a implementação de um Canal de comunicação que utiliza o protocolo TCP/IP como meio de transporte, e um protocolo binário na camada de aplicação para a codificação/decodificação das interações entre Participantes e membros de Objetos Distribuídos.

O Canal e o protocolo de aplicação estão implementados em bibliotecas de software dedicadas e separadas das bibliotecas do *middleware*. Desta forma, a implementação do protocolo pode futuramente ser reutilizada por outros meios de transmissão.

6.3.1 Protocolo de Aplicação

Este protótipo implementa um protocolo de camada de aplicação para tradução das interações entre Participantes e membros em formato transmissível pelo meio de comunicação. Como se trata de uma prova de conceitos, o projeto deste protocolo considerou a facilidade de implementação e compreensão, não sendo otimizado e robusto para aplicações reais.

Este protocolo utiliza uma codificação simplificada em binário e a implementação oferece funções para converter as estruturas do subcomponente *interação* em sequências de bytes.

A estrutura central suportada por este protocolo é a classe *Valor*, utilizada nos valores de atributos, argumentos de métodos e eventos, e valores de retorno de métodos, e cujo formato geral é ilustrado na figura 6.7.

Figura 6.7: Formato geral de um Valor.

Fonte: Autor.

O primeiro byte da sequência contém o identificador do tipo do dado, e segue os valores da tabela 6.1. Os N bytes seguintes contém o dado. Dados numéricos seguem a representação nativa, ou seja, o mesmo tamanho e formato que o número é armazenado na memória do computador.

Cadeias de caracteres (*Strings*) são formatadas de acordo com a figura 6.8.

Após o identificador de tipo (0x0C para *Strings*), segue o tamanho, em bytes, da cadeia, no formato inteiro 32 bits. Os N bytes seguintes são a sequência de caracteres da cadeia.

Tabela 6.1: Identificadores de Tipos (Hexadecimal).

Tipo Simples	Identificador	Sequência	Identificador
VOID	0x01	VALOR	0x2B
INT8	0x02	INT8	0x20
UINT8	0x03	UINT8	0x21
INT16	0x04	INT16	0x22
UINT16	0x05	UINT16	0x23
INT32	0x06	INT32	0x24
UINT32	0x07	UINT32	0x25
INT64	0x08	INT64	0x26
UINT64	0x09	UINT64	0x27
FLOAT	0x0A	FLOAT	0x28
DOUBLE	0x0B	DOUBLE	0x29
STRING	0x0C	STRING	0x2A

Figura 6.8: Formato das Cadeias de Caracteres.

Fonte: Autor.

Valores são sequências de um determinado tipo, incluindo objetos do tipo *Valor*, são formatados de acordo com a figura 6.9.

Figura 6.9: Formato das Sequências de Elementos.

Fonte: Autor.

O primeiro byte é o identificador de tipo, que deve estar entre 0x20 e 0x2B), seguido de 4 bytes contendo o número M de elementos da cadeia. Em seguida segue a sequência de M elementos. No caso de sequências de objetos do tipo *Valor*, cada elemento segue a

formatação geral de valores, contendo o identificador de tipo e dado, podendo ser utilizados recursivamente.

As interações são codificadas conforme o formato apresentado na figura 6.10. Os quatro primeiros bytes são destinados ao tamanho total da cadeia de bytes. O byte seguinte contém o identificador de interação conforme a tabela 6.2. Os próximos elementos são o nome completo do Objeto Distribuído e o nome do membro associado com a interação. Ambos os nomes são codificados como cadeia de caracteres, ou seja, 4 bytes para o tamanho da cadeia seguido pela sequência de caracteres. Os demais bytes são específicos de cada tipo de interação.

Figura 6.10: Formato Geral de uma Interação.

Fonte: Autor.

Tabela 6.2: Identificadores de Interações (Hexadecimal).

Interação	Identificador
Invocação de Método	0x00
Retorno de Método	0x01
Atualização de Atributo	0x02
Disparo de Evento	0x03

O parâmetro específico para atualização de atributos consiste apenas no novo Valor. Nas interações do tipo Disparo de Evento, o parâmetro específico é uma sequência dos valores dos argumentos do evento, seguindo a ordem em que foram declarados.

Interações do tipo Invocação de Métodos carregam como parâmetros o identificador da chamada, necessário para o retorno; o indicador se a chamada é de via única e a sequência de argumentos, conforme ilustrado na figura 6.11.

O formato do Retorno de Método é diferente das demais interações, conforme apresentado na figura 6.12.

Figura 6.11: Formato dos parâmetros de Invocação de Método.

Fonte: Autor.

Figura 6.12: Formato dos parâmetros de Retorno de Invocação de Método.

Fonte: Autor.

O Tamanho da mensagem (4 bytes) e o tipo da interação (1 byte de valor 0x01) seguem conforme as demais interações. Como o identificador de invocação é suficiente para localizar a invocação pendente, o nome completo do OD e o nome do método não são necessários.

6.3.2 Implementação do Canal TCP/IP Simples

Esta implementação de canal utiliza o protocolo de transporte TCP/IP para transmissão de interações entre Participantes.

Cada instância deste canal pode ser configurada como iniciador ou receptor de conexões. O iniciador utiliza como parâmetros o endereço de rede e número de porta do Participante receptor, enquanto que o receptor utiliza apenas o número de porta TCP/IP que irá receber a conexão. Como simplificação, cada instância deste canal suporta apenas uma única conexão com Participante remoto.

Logo após a conexão TCP/IP ser estabelecida, ocorre a fase de identificação na qual as partes envolvidas enviam as identificações locais do Participante e Canal, possibilitando os Participantes identificarem os canais. Concluído a identificação, o canal está pronto para envio e recebimento de interações, seguindo o protocolo apresentado na seção 6.3.1.

O modelo de execução implementado consiste em dois *threads* internos, de saída e de entrada, dedicados para envio e recebimento de interações, respectivamente.

O *thread* de saída é responsável pela codificação da interação em sequência de bytes, e transmissão para o Participante remoto por meio da conexão TCP/IP. Uma fila de interações é utilizada como mecanismo de sincronização entre este *thread* e o *thread* do Participante originador da interação, estabelecendo uma relação produtores-consumidor. As interações são adicionadas nesta fila através da implementação dos métodos virtuais para envio de interação. Quando a fila está vazia, este *thread* é bloqueado até uma nova interação estar disponível para consumo.

O *thread* de entrada recebe cadeias de caracteres provenientes da conexão TCP/IP, decodifica-os em interações, e inicia o despacho para os *Proxies* ou Adaptadores associados, utilizando os métodos de despacho fornecidos pela classe base *Canal*.

Este modelo foi escolhido para esta prova de conceitos para demonstrar a viabilidade de utilização de modelos *multi-threads* nas implementações nos canais.

6.4 Exemplo de Utilização da Camada Núcleo

Para demonstrar o funcionamento da implementação da camada núcleo e exemplificar o emprego do *middleware* proposto em aplicações distribuídas, foi elaborada uma Aplicação Distribuída que utiliza os principais recursos da proposta.

A Aplicação Distribuída de exemplo é composta por algumas estruturas de dados e classes de Objetos Distribuídos, conforme apresentado na figura 6.13.

Figura 6.13: Diagrama de Classes do Exemplo de Aplicação Distribuída.

Fonte: Autor.

A composição de estruturas é exemplificada por meio das estruturas *Estrutura* e *Estrutura2*. As classes de Objetos Distribuídos *ClasseBase* e *ClasseFilha* demonstram o emprego de herança entre classes.

Este exemplo é constituído por dois processos Participantes, e uma instância da classe *ClasseFilha*, relacionados conforme a topologia apresentada na figura 6.14.

Figura 6.14: Topologia do Exemplo de Aplicação Distribuída.

Fonte: Autor.

A implementação dos Participantes consiste nas seguintes etapas:

1. Inicialização local do Gerenciador de Aplicação Distribuída;
2. Declaração das estruturas de dados e classes de OD;
3. Estabelecimento dos canais de comunicação;
4. Registro de Objetos Distribuídos;
5. Preparo dos Adaptadores e *Proxies* (obtenção de referências e associação com funções locais, por exemplo);
6. Associação dos *Proxies* e Adaptadores aos canais de comunicação;

7. Interação com os membros dos ODs;
8. Finalização do Participante;

O código fonte da implementação do Participante se encontra no Apêndice A.

Os testes realizados na implementação do *middleware* consistiram em interações entre Participantes através dos membros de Objetos Distribuído, verificando o comportamento esperando durante este processo.

A interação com Atributos foi verificada atribuindo novos valores no Participante fornecedor, e constatando o correto recebimento e a notificação através dos *Callbacks*, no Participante utilizador.

Os Eventos foram testados disparando diversos eventos de ambos os Participantes, e verificando que os *Callbacks* associados ao evento nos Participantes utilizadores foram chamados corretamente pelo *middleware*, e os valores dos argumentos recebidos correspondem aos enviados.

A verificação dos métodos consistiu em invocações síncronas, assíncronas e de via única. No Participante fornecedor, verificaram-se a correta chamada do *Callback* associado, e a consistência dos valores de argumentos recebidos com os enviados. No Participante utilizador, verificaram-se os valores retornados nas chamadas síncronas e assíncronas, e o correto funcionamento dos diferentes modos de invocação.

Não foram realizados testes de desempenho nesta implementação pois a performance de uma aplicação distribuída real utilizando este *middleware* é diretamente influenciada pela implementação dos canais utilizados. Como o canal implementado nesta prova de conceitos visou apenas a avaliação das funcionalidades propostas, qualquer teste de desempenho não seria representativo.

6.5 Síntese

Este capítulo apresentou os detalhes de implementação de uma prova de conceitos da camada núcleo do *middleware* proposto. Esta implementação fornece as principais funcionalidades e conceitos do modelo de objetos proposto.

Complementando o protótipo do *middleware*, foram também implementados um canal de comunicação baseado no protocolo de transporte TCP/IP, e um protocolo de camada de aplicação para codificação das interações entre Participante e membros de Objetos

Distribuído em mensagens de rede.

As funcionalidades do protótipo foram testadas através de Aplicações Distribuídas de exemplos, demonstrando a viabilidade técnica do modelo de objetos e arquitetura de software de um novo *middleware* para aplicações distribuídas de realidade virtual.

7 EXEMPLO DE APLICAÇÃO DISTRIBUÍDA

Os capítulos anteriores detalharam o modelo de objetos, arquitetura de software e a implementação de um protótipo do *middleware* proposto. O objetivo deste capítulo é demonstrar a aplicabilidade desta proposta por meio de um exemplo de projeto de uma aplicação distribuída simples que explora e exemplifica as funcionalidades e facilidades do *middleware* proposto.

Trata-se de um jogo multijogador em primeira pessoa que consiste na simulação de uma guerra de balões d'água. Neste jogo existe uma arena virtual onde todos os jogadores podem mover-se livremente, atirar balões de água e interagir com alguns objetos da arena. O objetivo é fazer o maior número de pontos possível acertando balões d'água nos demais jogadores e evitando ser atingido.

Cada usuário utiliza uma unidade computacional (computador ou infraestrutura de RV, por exemplo) que se comunica com as demais através de uma rede de dados, local ou a internet.

Um computador, dedicado ou não, é responsável pela coordenação da partida, gerenciando a entrada e saída de jogadores, e processando aspectos comuns a todos os jogadores.

Para exemplificar o processamento distribuído desta aplicação, cada unidade computacional processa os aspectos específicos do jogador associado, como a movimentação na arena, detecção de colisão e a física dos lançamentos dos balões, por exemplo. Esta topologia assume que os participantes utilizam software "honestos" às regras do jogo.

7.1 Dinâmica do Jogo

Cada jogador possui uma quantidade limitada de balões d'água, pontos de acerto e pontos de vida. Os jogadores começam com uma quantidade fixa de balões e de pontos de vida.

A medida que o jogador acerta um oponente o valor dos pontos de acerto é incremen-

tado. Se o jogador é atingido por um balão, é decrementado um ponto de vida. Quando valor de pontos de vida chega a zero, o jogador perde todos os pontos de acerto, e a quantidade de balões e pontos de vida retornam aos valores iniciais. A classificação dos jogadores é determinada pelos pontos de acerto.

A arena é composta por elementos estáticos e dinâmicos. Elementos estáticos, tais como o terreno, muros e construções, não se movem e não interagem com jogadores.

Os elementos dinâmicos são entidades as quais os jogadores podem interagir, podendo ser elementos do cenário, como uma porta que pode ser aberta ou fechada; ou modificadores do jogo, como recuperadores de pontos de vida, ou suplemento de balões, por exemplo.

7.2 Classes e Estruturas de Dados

A figura 7.1 apresenta um dos possíveis modelo de classes de objetos distribuídos e estrutura de dados deste jogo. Para manter este exemplo simples, as entidades estáticas do jogo não fazem parte do modelo de objetos distribuídos, uma vez que eles não possuem comportamentos e estados que necessitam ser transmitidos pela rede de dados.

As classes e estruturas estão agrupadas nos seguintes *namespaces*:

- *comum*: Estruturas que não são específicas ao domínio desta aplicação, contendo as seguintes estruturas:
 - *Vetor3*: Representa um vetor tridimensional;
 - *Transformacao*: Matriz 4x4 (16 elementos numéricos) para operações de translação, rotação e escala de vetores 3D;
 - *Quaternion*: Vetor de quatro dimensões utilizado para operações de rotação de vetores 3D;
- *jogo*: Classes e estruturas específicas a esta aplicação.

A classe base *Entidade* representa um elemento dinâmico do jogo cujo comportamento e/ou estado precisa ser transmitido para um ou mais computadores na rede. O atributo *transformacao* contém informações sobre o posicionamento, orientação e escala da entidade no cenário tridimensional. O atributo *visivel* indica se a entidade está presente ou não na cena. O evento *EntidadeDestruida* é disparado quando a entidade está prestes a deixar de fazer parte da cena.

Figura 7.1: Diagrama de Classes do exemplo de jogo..

Fonte: Autor.

Cada jogador da partida é representado por um objeto distribuído do tipo *Jogador*. Além dos membros herdados da classe *Entidade*, esta classe contém atributos específicos para os jogadores, tais como os pontos de acerto (atributo *pontos*), pontos de vida (*vida*) e a orientação da cabeça/mira (*orientacaoCabeca*). O evento *LancouBalao* é disparado quando o jogador lança um balão, e contém como parâmetro o vetor velocidade do lançamento. O evento *FoiAtingido* é disparado quando o jogador é atingido por um balão, e indica qual jogador fez o lançamento.

ElementoDinamico é a classe base para os elementos dinâmicos da arena. Ela não possui membros comuns aos diferentes tipos de elementos, existindo somente para manter a semântica da hierarquia de classes.

A classe *Porta* exemplifica um elemento dinâmico de cenário, e contém um atributo para indicar se está aberta ou fechada, e métodos para abrir ou fechá-la.

As classes *Medicamento* e *Municao* são exemplos de elementos dinâmicos modificadores da partida. Os medicamentos são entidades posicionadas no cenário que possibilita o jogador que o utilizar recupere uma determinada quantidade de pontos de vida. A quantidade de pontos a ser restaurado é determinado pelo atributo *pontosVida*. Ao ser utilizado, através do método *Usar*, os pontos fornecidos pelo item é adicionado aos pontos de vida do jogador que o utilizou, e o item deixa de existir.

Munições são elementos que aumentam a quantidade de balões disponíveis para lançamento. O valor a ser acrescentado é determinado pelo atributo *quantidade*. Assim como a classe *Medicamento*, o método *Usar* é invocado quando um jogador utiliza o item.

A instância única (*Singleton*) da classe *GerenciadorJogo* é responsável pelo gerenciamento da partida, sendo logicamente composto por todas as entidades do jogo, e fornecendo meios para os jogadores entrarem e saírem da partida. O atributo *jogadores* contém a lista de jogadores da partida. Os métodos *Entrar* e *Sair* são utilizados pelos jogadores para participarem ou deixarem de participar da partida, respectivamente. O evento *NovoJogador* é disparado quando um novo jogador entra na partida, sinalizando os demais jogadores.

7.3 Organização das Instâncias

As instâncias das classes desta aplicação é organizada em uma hierarquia, conforme apresentado no diagrama da figura 7.2.

Figura 7.2: Diagrama de Instâncias do exemplo de jogo..

Fonte: Autor.

A raiz desta hierarquia é a instância da classe *jogo::GerenciadorJogo*. Abaixo deste objeto estão todos os jogadores e elementos dinâmicos do jogo, concretizando a relação de composição entre as classes *GerenciadorJogo* e *Entidade*.

Para melhor organização das entidades, os objetos distribuídos dos tipos *Jogador* e *ElementoDinamico* são agrupados em sub-hierarquias separadas. Esta separação é realizado por meio de objetos distribuídos vazios, utilizados apenas como agrupadores.

7.4 Participantes e Canais de Comunicação

Esta aplicação distribuída emprega a topologia híbrida cliente/servidor e ponto a ponto, e é composta por um participante responsável pelo gerenciamento do jogo (servidor), e

diversos participantes jogadores (clientes).

O participante gerenciador coordena a entrada e saída de jogadores, processa o comportamento dos elementos dinâmicos da arena e calcula os lançamento dos balões. Cada participante jogador processa o comportamento de um jogador específico, sendo responsável pela movimentação na arena, pelo início de interações com elementos dinâmicos e iniciar o lançamento de balões.

Durante a partida cada participante se comunica diretamente com os demais através de dois tipos de canais: Canal Sem Garantia de Entrega (CSGE) e Canal Com Garantia de Entrega (CCGE).

O CSGE é baseado no protocolo de transporte UDP, apresentando pouco *overhead* e, conseqüentemente, baixa latência, porém, sem garantias de entrega e de ordenação dos datagramas transmitidos. Este canal é empregado para distribuição de membros de objetos que são atualizados constantemente e que necessitam de melhor desempenho, e/ou são tolerantes a perda de dados.

O atributo *transformacao* da classe *Entidade* é um exemplo de membro que utiliza o canal CSGE, pois, neste caso, as perdas de dados são compensadas pela alta frequência de atualização do valor deste atributo.

O canal CCGE utiliza o protocolo TCP/IP para garantir a entrega e ordenação dos dados, porém, apresentando maior *overhead* e pior desempenho devido aos mecanismos de garantia de entrega, controle de fluxo e congestionamento. Este canal é utilizado nos membros de objetos que não são tolerantes a perda de dados, degradando o correto funcionamento do jogo caso uma interação não alcance o(s) participantes de destino.

7.5 Distribuição dos Membros dos Objetos

Os membros da instância única da classe *GerenciadorJogo* são fornecidos pelo participante gerenciador, e utilizados pelos participantes jogadores, conforme apresentado na tabela 7.1.

Para entrar em uma partida, o participante jogador inicialmente estabelece os canais de comunicação CCGE e CSGE com o participante gerenciador e as associações com os membros do objeto gerenciador.

O participante jogador notifica sua entrada ao gerenciador invocando o método *Entrar*, que irá atualizar a lista de jogadores (atributo *jogadores*) e notificar os demais jogadores

Tabela 7.1: Distribuição dos membros da instância da classe *GerenciadorJogo*.

	Membro	Fornecedor(es)	Utilizadores	Canal
Atributos	<i>jogadores</i>	Gerenciador	Jogadores	CCGE
Métodos	<i>Entrar</i>	Gerenciador	Jogadores	CCGE
	<i>Sair</i>	Gerenciaidor	Jogadores	CCGE
Eventos	<i>NovoJogador</i>	Gerenciador	Jogadores	CCGE

disparando o evento *NovoJogador*. O novo participante então utiliza a lista de jogadores para estabelecer os canais CCGE e CSGE com os demais participantes.

Os membros das instâncias da classe *Jogador* são fornecidos pelo participante jogador e pelo participante gerenciador, conforme apresentado na tabela 7.2.

Tabela 7.2: Distribuição dos membros do objeto distribuído das instâncias da classe *Jogador*.

	Membro	Fornecedor(es)	Utilizadores	Canal
Atributos	<i>transformacao</i>	Jogador	Demais Jogadores e Gerenciador	CSGE
	<i>visivel</i>	Jogador	Demais Jogadores e Gerenciador	CCGE
	<i>orientacaoCabeca</i>	Jogador	Demais Jogadores e Gerenciador	CSGE
	<i>pontos</i>	Gerenciador	Todos Jogadores	CCGE
	<i>vidas</i>	Gerenciador	Todos Jogadores	CCGE
Eventos	<i>EntidadeDestruída</i>	Gerenciador	Todos Jogadores	CCGE
	<i>LancouBalao</i>	Jogador	Demais Jogadores e Gerenciador	CCGE
	<i>FoiAtingido</i>	Gerenciador	Todos Jogadores	CCGE

O participante jogador associado com uma determinada instância da classe *Jogador* processa e fornece atributos relacionados ao posicionamento e aparência do jogador, e dispara eventos *LancouBalao*. Os atributos *transformacao* e *orientacaoCabeca* utilizam canais CSGE pois eles são atualizados constantemente e demandam baixa latência na transmissão para outros participantes. Os demais membros são fornecidos pelo participante gerenciador.

Durante a partida, os participantes jogadores constantemente atualizam os atributos

transformacao e *orientacaoCabeca* com a atual posição e orientação do jogador. Quando o usuário efetua um lançamento de balão, o participante associado dispara o evento *LancouBalao*, que é utilizado pelo gerenciador para calcular o lançamento, e pelos demais jogadores para representarem graficamente e sonoramente.

Caso o balão lançado atinja outro jogador, o gerenciador dispara o evento *FoiAtingido* na instância que foi atingida, e atualiza os atributos *pontos* na instância que efetuou o lançamento, e *vida* na instância que foi atingida. O evento *FoiAtingido* é utilizado pelos participantes jogadores para a devida representação gráfica e sonora.

Os membros das instâncias das classes derivadas de *ElementoDinamico* são fornecidos pelo participante gerenciador e utilizado pelos jogadores. A interação com estes elementos ocorre por meio da invocação de métodos específicos de cada classe derivada.

Para interagir com os objetos do tipo *Porta*, os participantes jogadores invocam os métodos *Abrir* ou *Fechar*, e utilizam o atributo *aberta* para atualizar a representação gráfica da porta.

A interação com os elementos dos tipos *Medicamento* e *Municao* ocorrem por meio do método *Usar*. Quando invocado, o participante gerenciador atualiza os atributos *pontos* e *vida* do jogador que realizou a invocação, e desabilita temporariamente o elemento através do atributo *visivel*.

7.6 Síntese

Este capítulo apresentou um exemplo de projeto de uma aplicação distribuída utilizando o *middleware* proposto, demonstrando sua aplicabilidade. O exemplo consistiu em um jogo multijogadores em primeira pessoa.

O modelo de objetos proposto possibilitou a modelagem das classes e objetos da aplicação utilizando os conceitos das linguagens orientadas a objetos, e independente da topologia de baixo nível da aplicação.

8 CONCLUSÃO

Desde o surgimento das primeiras redes de computadores, técnicas e ferramentas para desenvolvimento de aplicações distribuídas são empregados seja devido aos recursos de um único computador não serem suficientes, seja para atender requisitos de qualidade de serviço, seja para atender maior quantidade de usuários. A Realidade Virtual é uma das áreas da computação que, devido a sua complexidade, se beneficia das técnicas de distribuição para proporcionar ambientes mais imersivos e/ou possibilitar a interação de múltiplos usuários.

Apesar de ser uma área específica, a Realidade Virtual inclui um espectro abrangente de classes de aplicações, como jogos multi-jogadores, infra-estruturas de alta imersão e simuladores de voo. Além de possuírem requisitos comuns, como restrições de latência e taxa de atualização, cada classe de aplicação têm demandas específicas como escalabilidades, ou requisitos de tempo-real, por exemplo.

Devido à complexidade e a diversidade de tipos das aplicações de realidade virtual, as alternativas disponíveis atendem requisitos específicos de uma única classe, ou, as genéricas, não fornecem modelos de objetos, comunicação ou execução favoráveis às aplicações de realidade virtual.

Neste contexto, este trabalho apresentou um novo modelo de objetos e arquitetura de um *middleware* para desenvolvimento de aplicações distribuídas de realidade virtual suficientemente flexível para atender diversas classes de aplicações.

O modelo de objetos proposto une os modelos publicador/assinante e requisição/resposta em uma única entidade similar aos objetos do paradigma de programação orientada a objetos. Além de de empregar conceitos intuitivos para desenvolvedores, esta entidade possibilita o desacoplamento lógico entre os processos, simplificando o projeto e a implementação das aplicações distribuídas. O principal diferencial desta abordagem é a possibilidade de distribuir a implementação de cada membro dos objetos em diferentes processos.

A arquitetura de software tem como principais características ser multi-camadas, e permitir a utilização de diversos modelos de execução, protocolos e meios de comunicação, favorecendo sua utilização em aplicações com diferentes requisitos específicos.

A viabilidade técnica desta proposta foi demonstrada por meio da implementação de uma prova de conceitos com os principais elementos do modelo de objetos e arquitetura. As funcionalidades foram testadas através de aplicações distribuídas de exemplo.

Apesar da motivação e os requisitos que guiaram este trabalho serem oriundos da área de realidade virtual, os modelos propostos são genéricos possibilitando seu emprego com sucesso também em outras áreas.

8.1 Principais Contribuições

As principais contribuições resultantes deste trabalho são:

- Um modelo de objetos intuitivo que promove o desacoplamento entre processos através de Objetos Distribuídos, compostos por atributos, métodos e eventos.
- Uma arquitetura de software flexível de um *middleware* que emprega o modelo de objeto proposto.
- Um protótipo que implementa o modelo de objetos e a arquitetura projetados neste trabalho.

8.2 Trabalhos Futuros

Como continuidade deste trabalho é sugerido as seguintes pesquisas e desenvolvimentos futuros:

- Explorar modelos de execução para diferentes tipos de aplicações distribuídas de realidade virtual;
- Explorar novos protocolos de comunicação em grupo em aplicações de realidade virtual;
- Complementar a arquitetura de software proposta com novas camadas alto-nível para tornar transparente a utilização do *middleware*

- Aperfeiçoar o protótipo, implementar bibliotecas de canais de comunicação, e implementar bibliotecas de Objetos Distribuídos que possam ser reutilizados em diversas aplicações.

REFERÊNCIAS

- [1] COULOURIS, G. et al. *Distributed Systems: Concepts and Design*. 5th. ed. USA: Addison-Wesley Publishing Company, 2011. ISBN 0132143011, 9780132143011.
- [2] CRUZ-NEIRA, C. et al. The cave: Audio visual experience automatic virtual environment. *Commun. ACM*, ACM, New York, NY, USA, v. 35, n. 6, p. 64–72, jun. 1992. ISSN 0001-0782. Disponível em: <<http://doi.acm.org/10.1145/129888.129892>>.
- [3] ZUFFO, J. a. A. et al. Caverna Digital - Sistema de Multiprojeção Estereoscópico Baseado em Aglomerados de PCs para Aplicações Imersivas em Realidade Virtual. In: *SBC 4th Symposium on Virtual Reality*. Florianópolis, SC, BR: [s.n.], 2001. (SVR '01).
- [4] KACZMARSKI, H.; ZUFFO, M. K. Commodity Clusters for Immersive Projection Environments, Course 47. In: *ACM SIGGRAPH 2002 Course Notes*. [S.l.: s.n.], 2002. (SIGGRAPH '02).
- [5] RAFFIN, B. et al. Commodity Clusters for Virtual Reality. In: *Proceedings of the IEEE Virtual Reality 2003*. Washington, DC, USA: IEEE Computer Society, 2003. (VR '03), p. 307—. ISBN 0-7695-1882-6. Disponível em: <<http://dl.acm.org/citation.cfm?id=832289.835985>>.
- [6] aES, M. P. G. et al. Synchronization and data sharing library for PC clusters. In: *Workshop on Commodity Clusters for Virtual Reality - VR-Cluster'03*. Los Angeles, CA, USA: [s.n.], 2003.
- [7] SOARES, L. P.; ZUFFO, M. K. JINX: An X3D Browser for VR Immersive Simulation Based on Clusters of Commodity Computers. In: *Proceedings of the Ninth International Conference on 3D Web Technology*. New York, NY, USA: ACM, 2004. (Web3D '04), p. 79–86. ISBN 1-58113-845-8. Disponível em: <<http://doi.acm.org/10.1145/985040.985052>>.
- [8] TEUBL, F. et al. Passeio Digital: Um Sistema de Multiprojeção de Baixo Custo Voltado às Aplicações Educacionais. In: *In XI Symposium on Virtual and Augmented Reality SVR2009*. [S.l.: s.n.], 2009.
- [9] RODRIGUES, F. L. D. *Sistema de Realidade Virtual para Simulador Visual de Passadiço*. Dissertação (Mestrado) — Escola Politécnica da Universidade de São Paulo, 2010.
- [10] RODRIGUES, F. L. D. et al. Sistema de Realidade Virtual para Simulador de Passadiço. In: *Military Simulation Workshop, XI Simpósio Brasileiro de Jogos e Entretenimento Digital - SBGames*. [S.l.: s.n.], 2012.

- [11] RODRIGUES, F. et al. Coupling Virtual Reality Open Source Software Using Message Oriented Middleware. In: LATOSCHIK, M. E. et al. (Ed.). *Software Engineering and Architectures for Realtime Interactive Systems (SEARIS), proceedings of the {IEEE} {V}irtual {R}eality 2009 workshop*. [S.l.]: Shaker Verlag, 2009. p. 17–24.
- [12] FERRAZ, R. et al. A tool to aid the development of distributed virtual reality applications. In: *2014 XVI Symposium on Virtual and Augmented Reality*. [S.l.]: IEEE Computer Society, 2014. p. 315–323.
- [13] DAHMANN, J. S.; FUJIMOTO, R. M.; WEATHERLY, R. M. The Department of Defense High Level Architecture. In: *Proceedings of the 29th conference on Winter simulation*. Washington, DC, USA: IEEE Computer Society, 1997. (WSC '97), p. 142–149. ISBN 0-7803-4278-X. Disponível em: <<http://dx.doi.org/10.1145/268437.268465>>.
- [14] TAYLOR, S. J. E.; TURNER, S. J.; STRASSBURGER, S. Guidelines for commercial off-the-shelf Simulation Package interoperability. In: *2008 Winter Simulation Conference*. IEEE, 2008. p. 193–204. ISBN 978-1-4244-2707-9. Disponível em: <<http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4736068>>.
- [15] DAHMANN, J.; FUJIMOTO, R. M.; WEATHERLY, R. M. The DoD High Level Architecture: An update. In: *Proceedings of the 30th conference on Winter simulation*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1998. (WSC '98), p. 797–804. ISBN 0-7803-5134-7. Disponível em: <<http://dl.acm.org/citation.cfm?id=293172.293308>>.
- [16] IEEE-1278. IEEE Standard for Distributed Interactive Simulation–Application Protocols. *IEEE Std 1278.1-2012 (Revision of IEEE Std 1278.1-1995)*, p. 1–747, 2012.
- [17] WILSON, A. L.; WEATHERLY, R. M. The Aggregate Level Simulation Protocol: An Evolving System. In: *Proceedings of the 26th Conference on Winter Simulation*. San Diego, CA, USA: Society for Computer Simulation International, 1994. (WSC '94), p. 781–787. ISBN 0-7803-2109-X. Disponível em: <<http://dl.acm.org/citation.cfm?id=193201.194367>>.
- [18] IEEE-1516. IEEE Standard for Modeling and Simulation (M & S) High Level Architecture (HLA)– Framework and Rules. *IEEE Std 1516-2010 (Revision of IEEE Std 1516-2000)*, p. 1–38, ago. 2010.
- [19] YANG, Z.; DUDDY, K. CORBA: a platform for distributed object computing. *SI-GOPS Oper. Syst. Rev.*, ACM, New York, NY, USA, v. 30, n. 2, p. 4–31, abr. 1996. ISSN 0163-5980. Disponível em: <<http://doi.acm.org/10.1145/232302.232303>>.
- [20] VINOSKI, S. CORBA: integrating diverse applications within distributed heterogeneous environments. *Communications Magazine, IEEE*, v. 35, n. 2, p. 46–55, fev. 1997. ISSN 0163-6804.
- [21] Object Management Group (OMG). *Common Object Request Broker Architecture (CORBA) v3.3 Specification*. 2012. \url{<http://www.omg.org/spec/CORBA/3.3/>}. Disponível em: <<http://www.omg.org/spec/CORBA/3.3/>>.
- [22] Object Management Group (OMG). *Real-Time CORBA v1.2*. [S.l.], 2005. Disponível em: <<http://www.omg.org/spec/RT/1.2/>>.

- [23] Object Management Group (OMG). *Data Distribution Service Version 1.2*. [S.l.], 2007. Disponível em: <<http://www.omg.org/spec/DDS/1.2>>.
- [24] PARDO-CASTELLOTE, G. OMG Data-Distribution Service: architectural overview. In: *Distributed Computing Systems Workshops, 2003. Proceedings. 23rd International Conference on*. [S.l.: s.n.], 2003. p. 200–206.
- [25] BELLAVISTA, P. et al. Data Distribution Service (DDS): A performance comparison of OpenSplice and RTI implementations. In: *Computers and Communications (ISCC), 2013 IEEE Symposium on*. [S.l.: s.n.], 2013. p. 377–383.
- [26] Object Management Group (OMG). *The Real-Time Publish-Subscribe Wire Protocol DDS Interoperability Wire Protocol (DDSI-RTPS) Version 2.2*. [S.l.], 2014. Disponível em: <<http://www.omg.org/spec/DDSI-RTPS/2.2/>>.
- [27] Object Management Group (OMG). *Data Distribution Service Data Local Reconstruction Layer (DDS-DLRL) Version 1.4 - Beta*. [S.l.], 2014. Disponível em: <<http://www.omg.org/spec/DDS-DLRL/1.4/>>.
- [28] The Foundation Initiative 2010 Program Office. *The TENA Architecture Reference Document*. [S.l.], 2002. Disponível em: <<https://www.tena-sda.org/download/attachments/6750/TENA2002.pdf>>.
- [29] NOSEWORTHY, J. R. Developing distributed applications rapidly and reliably using the TENA middleware. In: *Military Communications Conference, 2005. MILCOM 2005. IEEE*. [S.l.: s.n.], 2005. p. 1507–1513 Vol. 3.
- [30] NOSEWORTHY, J. R. The Test and Training Enabling Architecture (TENA) Supporting the Decentralized Development of Distributed Applications and LVC Simulations. In: *Proceedings of the 2008 12th IEEE/ACM International Symposium on Distributed Simulation and Real-Time Applications*. Washington, DC, USA: IEEE Computer Society, 2008. (DS-RT '08), p. 259–268. ISBN 978-0-7695-3425-1. Disponível em: <<http://dx.doi.org/10.1109/DS-RT.2008.35>>.
- [31] NOSEWORTHY, J. R. Supporting the Decentralized Development of Large-Scale Distributed Real-Time LVC Simulation Systems with TENA (The Test and Training Enabling Architecture). In: *Distributed Simulation and Real Time Applications (DS-RT), 2010 IEEE/ACM 14th International Symposium on*. [S.l.: s.n.], 2010. p. 22–29. ISSN 1550-6525.
- [32] GAMMA, E. et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. [S.l.]: Addison-Wesley, 1995.
- [33] B. Sabata M. BROWN, B. D. Transport Protocol for Reliable Multicast: TRM. Disponível em: <<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.57.5141>>.
- [34] BESSANI, A.; da Silva Fraga, J.; LUNG, L. Extending the UMIOP Specification for Reliable Multicast in CORBA. In: MEERSMAN, R.; TARI, Z. (Ed.). *On the Move to Meaningful Internet Systems 2005: CoopIS, DOA, and ODBASE*. Springer Berlin Heidelberg, 2005, (Lecture Notes in Computer Science, v. 3760). p. 662–679. ISBN 978-3-540-29736-9. Disponível em: <http://dx.doi.org/10.1007/11575771_42>.

- [35] BIRMAN, K.; JOSEPH, T. Exploiting virtual synchrony in distributed systems. In: *Proceedings of the eleventh ACM Symposium on Operating systems principles - SOSP '87*. New York, New York, USA: ACM Press, 1987. v. 21, n. 5, p. 123–138. ISBN 089791242X. ISSN 0163-5980. Disponível em: <<http://dl.acm.org/citation.cfm?id=41457.37515>>.
- [36] BIRMAN, K. et al. Scalable Multicast Platforms for a New Generation of Robust Distributed Applications. In: *2007 2nd International Conference on Communication Systems Software and Middleware*. IEEE, 2007. p. 1–8. ISBN 1-4244-0613-7. Disponível em: <<http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4268060>>.
- [37] LAMPORT, L. The part-time parliament. *ACM Transactions on Computer Systems*, ACM, v. 16, n. 2, p. 133–169, maio 1998. ISSN 07342071. Disponível em: <<http://dl.acm.org/citation.cfm?id=279227.279229>>.
- [38] LIAO, T. Light-weight reliable multicast protocol. *INRIA Technical Report*, 1998.
- [39] MOSER, L. et al. Extended virtual synchrony. In: *14th International Conference on Distributed Computing Systems*. IEEE Comput. Soc. Press, 1994. p. 56–65. ISBN 0-8186-5840-1. Disponível em: <<http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=302392>>.
- [40] OBRACZKA, K. Multicast transport protocols: a survey and taxonomy. *IEEE Communications Magazine*, v. 36, n. 1, p. 94–102, 1998. ISSN 01636804. Disponível em: <<http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=649333>>.
- [41] OSTROWSKI, K. J. *Live distributed objects*. Tese (Ph.D. Dissertation) — Cornell University, 2008.
- [42] WANG, J.; ZHENG, T. A hybrid multicast–unicast assignment approach for data distribution management in HLA. *Simulation Modelling Practice and Theory*, v. 40, p. 39–63, jan. 2014. ISSN 1569190X. Disponível em: <<http://www.sciencedirect.com/science/article/pii/S1569190X13001226>>.
- [43] BABAY, A. *The Accelerated Ring Protocol: Ordered Multicast for Modern Data Centers*. Tese (Master of Science Thesis) — The Johns Hopkins University, 2014.
- [44] BALAKRISHNAN, M.; PLEISCH, S.; BIRMAN, K. Slingshot: Time-CriticalMulticast for Clustered Applications. In: *Fourth IEEE International Symposium on Network Computing and Applications*. IEEE, 2005. p. 205–214. ISBN 0-7695-2326-9. Disponível em: <<http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1565954>>.
- [45] BALAKRISHNAN, M.; PLEISCH, S.; BIRMAN, K. Slingshot: Time-CriticalMulticast for Clustered Applications. *Fourth IEEE International Symposium on Network Computing and Applications*, 2005.
- [46] BALAKRISHNAN, M.; BIRMAN, K. Reliable Multicast for Time-Critical Systems. In: *1st Workshop on Applied Software Reliability (WASR)*. [S.l.: s.n.], 2006. p. 1–6.

- [47] BALAKRISHNAN, M.; BIRMAN, K.; PHANISHAYEE, A. PLATO: Predictive Latency-Aware Total Ordering. In: *2006 25th IEEE Symposium on Reliable Distributed Systems (SRDS'06)*. IEEE, 2006. p. 175–188. ISBN 0-7695-2677-2. ISSN 1060-9857. Disponível em: <<http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4032480>>.
- [48] BALAKRISHNAN, M. et al. Ricochet: Lateral Error Correction for Time-Critical Multicast. In: *NSDI 2007: Fourth Usenix Symposium on Networked Systems Design and Implementation*. [S.l.: s.n.], 2007.
- [49] FLOYD, S. et al. A reliable multicast framework for light-weight sessions and application level framing. *IEEE/ACM Transactions on Networking*, IEEE Press, v. 5, n. 6, p. 784–803, 1997. ISSN 10636692. Disponível em: <<http://dl.acm.org/citation.cfm?id=270856.270863>>.
- [50] GRANSART, C.; GEIB, J.-M. Using an ORB with multicast IP. In: *Proceedings of PCS99: Parallel Computing Systems Conference*. Enseada, Mexico: [s.n.], 1999.
- [51] LEVINE, B.; GARCIA-LUNA-ACEVES, J. A comparison of known classes of reliable multicast protocols. In: *Proceedings of 1996 International Conference on Network Protocols (ICNP-96)*. IEEE Comput. Soc. Press, 1996. p. 112–121. ISBN 0-8186-7453-9. Disponível em: <<http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=564920>>.
- [52] MOSER, L. et al. Multicast group communication for CORBA. In: *Proceedings of the International Symposium on Distributed Objects and Applications*. IEEE, 1999. p. 98–107. ISBN 0-7695-0182-6. Disponível em: <<http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=793993>>.
- [53] NETT, E.; KAISER, J. Verifications of RMP: an efficient reliable multicast protocol. In: *Proceedings Second International Symposium on Parallel Architectures, Algorithms, and Networks (I-SPAN'96)*. IEEE Comput. Soc. Press, 1996. p. 388–393. ISBN 0-8186-7460-1. ISSN 1087-4089. Disponível em: <<http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=509015>>.
- [54] OSTROWSKI, K.; BIRMAN, K.; DOLEV, D. Quicksilver Scalable Multicast (QSM). In: *2008 Seventh IEEE International Symposium on Network Computing and Applications*. IEEE, 2008. p. 9–18. Disponível em: <<http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4579634>>.
- [55] OSTROWSKI, K.; BIRMAN, K.; DOLEV, D. Live Distributed Objects: Enabling the Active Web. *IEEE Internet Computing*, IEEE, v. 11, n. 6, p. 72–78, nov. 2007. ISSN 1089-7801. Disponível em: <<http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=4376231>>.
- [56] OSTROWSKI, K. et al. *ECOOP 2008 – Object-Oriented Programming*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008. 463–489 p. (Lecture Notes in Computer Science, v. 5142). ISSN 0302-9743. ISBN 978-3-540-70591-8. Disponível em: <<http://dl.acm.org/citation.cfm?id=1428508.1428536>>.

- [57] OSTROWSKI, K. et al. Implementing reliable event streams in large systems via distributed data flows and recursive delegation. In: *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems - DEBS '09*. New York, New York, USA: ACM Press, 2009. p. 1. ISBN 9781605586656. Disponible em: <<http://dl.acm.org/citation.cfm?id=1619258.1619279>>.
- [58] OSTROWSKI, K.; BIRMAN, K.; PHANISHAYEE, A. *QuickSilver Scalable Multicast*. [S.l.], abr. 2006. Disponible em: <<http://www.truststc.org/pubs/161.html>>.
- [59] OSTROWSKI, K.; BIRMAN, K. P. Scalable group communication system for scalable trust. In: *Proceedings of the first ACM workshop on Scalable trusted computing - STC '06*. New York, New York, USA: ACM Press, 2006. p. 3. ISBN 1595935487. Disponible em: <<http://dl.acm.org/citation.cfm?id=1179474.1179477>>.
- [60] POPESCU, A. et al. A Survey of Reliable Multicast Communication. In: *2007 Next Generation Internet Networks*. IEEE, 2007. p. 111–118. ISBN 1-4244-0856-3. Disponible em: <<http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4231828>>.
- [61] VIGFUSSON, Y. et al. Dr. multicast: Rx for data center communication scalability. In: *Proceedings of the 5th European conference on Computer systems - EuroSys '10*. New York, New York, USA: ACM Press, 2010. p. 349. ISBN 9781605585772. Disponible em: <<http://dl.acm.org/citation.cfm?id=1755913.1755949>>.

APÊNDICE A – CÓDIGO FONTE DE EXEMPLO DE PARTICIPANTE

Listagem A.1: Exemplo de implementação de Participante.

```

1
2 //Implementacao nativa do Metodo 1
3 void Metodo1( const interacao::InvocacaoMetodo& invocacao, Valor& outResultado, std::
    shared_ptr< Canal > canal )
4 {
5     //Este metodo recebe como argumento dois valores float. O retorno e a soma destes
        dois valores
6     float arg1 = invocacao.Argumento( 0 ).Get< float >();
7     float arg2 = invocacao.Argumento( 1 ).Get< float >();
8
9     float resultado = arg1 + arg2;
10
11     outResultado = resultado;
12
13     //Aguarda 1 segundo para retornar, simulando uma requisicao lenta
14     using namespace std::chrono_literals;
15     std::this_thread::sleep_for( 1s );
16 }
17
18 //Implementacao nativa do Metodo 2
19 void Metodo2( const interacao::Invocacao& invocacao, Valor& outResultado, std::shared_ptr
    < Canal > canal )
20 {
21     //Este metodo recebe como argumento uma string, e retorna o mesmo valor
22     std::string arg1 = invocacao.Argumento( 0 ).Get< std::string >();
23
24     outResultado = arg1;
25
26     //Aguarda 1 segundo para retornar, simulando uma requisicao lenta
27     using namespace std::chrono_literals;
28     std::this_thread::sleep_for( 1s );
29 }
30
31
32 //Callbacks
33

```

```

34 //Atualizacao de atributo
35 void CallbackAtualizacaoAtributo( const Valor& valor, const ProxyAtributo& proxy, std::
    shared_ptr< canal > canal )
36 {
37     //obtem referencia da instancia Instancia
38     auto obj = proxy.Instancia().lock();
39     assert( obj );
40
41
42     std::cout << "Callback_de_atualizacao_de_atributo" << std::endl;
43     std::cout << "Instancia:" << obj->NomeCompleto() << std::endl;
44     std::cout << "Atributo:" << proxy.Atributo()->Nome() << std::endl;
45     std::cout << "Valor:";
46     ImprimeValor( valor );
47     std::cout << std::endl;
48 }
49
50
51 //notificacao de Evento
52 void CallbackEvento( const interacao::DisparoEvento& evento, const ProxyEvento& proxy,
    std::shared_ptr< canal > canal )
53 {
54     //obtem referencia da instancia Instancia
55     auto obj = proxy.Instancia().lock();
56     assert( obj );
57
58     //Imprime argumento do evento
59     std::cout << "Callback_de_notificacao_de_evento" << std::endl;
60     std::cout << "Instancia:" << obj->NomeCompleto() << std::endl;
61     std::cout << "Evento:" << proxy.Evento()->Nome() << std::endl;
62     std::cout << "Argumento:" << evento.Argumento( 0 ).Get< double >() << std::endl;
63 }
64
65
66
67 int main( int argc, char** argv )
68 {
69     //Configuracao da aplicacao
70     GerenteAplicacao* app = new GerenteAplicacao( "AplicacaoTeste", "Participante" );
71
72     //Declaracao dos tipos
73
74     //Estruturas de dados Estrutura2
75     tipo::Estrutura* e2 = new tipo::Estrutura( "Estrutura2" );
76     e2->NovoCampo( "Campo1", TipoInt32 );
77     e2->NovoCampo( "Campo2", TipoString );
78     app->GerenciadorDeclaracao().AdicionaTipo( e2, "Namespace1::Namespace2" );
79
80     //Estruturas de dados Estrutura
81     tipo::Estrutura* e = new tipo::Estrutura( "Estrutura" );
82     e->NovoCampo( "Campo1", TipoString );
83     e->NovoCampo( "Campo2", e2 );
84     app->GerenciadorDeclaracao().AdicionaTipo( e, "Namespace1::Namespace2" );
85

```

```

86 //Classe ClasseBase
87 tipo::Classe* cb = new tipo::Classe( "ClasseBase" );
88
89 cb->NovoAtributo( "Atributo1", TipoString );
90 cb->NovoMetodo( "Metodo1", TipoFloat, { &TipoFloat, &TipoFloat } );
91 app->GerenciadorDeclaracao().AdicionaTipo( cb, "Namespace1::Namespace2" );
92
93 //Classe ClasseFilha
94 tipo::Classe* cf = new tipo::Classe( "ClasseFilha" );
95
96 cf->NovoAtributo( "Atributo2", e );
97 cf->NovoMetodo( "Metodo1", TipoVoid, { &TipoString } );
98 cf->NovoEvento( "Evento1", { &TipoDouble } );
99 app->GerenciadorDeclaracao().AdicionaTipo( cf, "Namespace1::Namespace2" );
100
101
102 //Cria uma instancia da ClasseFilha
103 ObjetoPtr obj1 = app->GerenciadorInstancia().RegistraNovaInstancia( "OD1", cf );
104
105 //Membros fornecidos (adaptadores)
106
107 auto& adptAtributo1 = obj1->AdaptadorAtributo( "Atributo1" );
108 auto& adptMetodo1 = obj1->AdaptadorMetodo( "Metodo1" );
109 adptMetodo1.AssociaMetodo( &Metodo1 ); //Associa implementacao
110 auto& adptEvento1 = obj1->AdaptadorEvento( "Evento1" );
111
112 //Membros utilizados (proxies)
113 auto& prxMetodo2 = obj1->ProxyMetodo( "Metodo2" );
114 auto& prxAtributo2 = obj1->ProxyAtributo( "Atributo2" );
115 prxAtributo2.AssociaObservador( &CallbackAtualizacaoAtributo );
116
117 auto& prxEvento1 = obj1->ProxyEvento( "Evento1" );
118 prxEvento1.AssociaObservador( &CallbackEvento );
119
120
121
122 //Configuracao do canal TCP/IP
123
124 auto canal = std::make_shared< canais::CanalTCP >( "Canal1" );
125 app->GerenciadorCanal().AdicionaCanal( canal );
126
127 //Estabelece conexao com Participante remoto
128 canal->Conecta( "participante2.exemplo.com", 1234 );
129
130
131 //Associa canal com proxies e adaptadores
132 adptMetodo1.AssociaCanal( canal );
133 adptAtributo1.AssociaCanal( canal );
134 adptEvento1.AssociaCanal( canal );
135
136 prxMetodo2.AssociaCanal( canal );
137 prxAtributo2.AssociaCanal( canal );
138 prxEvento1.AssociaCanal( canal );
139

```

```

140 //Fim das etapas de configuracao
141 //-----
142
143
144 //Testes de funcionalidade
145
146
147 //Interacoes com metodos
148
149 //Invocacao de metodo sincrona
150 Valor resultado = prxMetodo2.Invoca( "Teste1" ); //resultado deve ser "Teste teste"
151
152 //Invocacao de metodo assincrona
153 std::future< Valor > f = prxMetodo2.InvocaAssinc( "Teste2" );
154
155 //Aguarda retorno
156 f.wait();
157 resultado = f.get(); //resultado deve ser "Teste2"
158
159 //Invocacao via unica
160 prxMetodo.InvocaUmaVia( "Teste3" );
161 //Retorno imediato e sem valor
162
163
164 //Interacoes com Atributos
165 //Obtencao do valor atual de atributo
166 Valor a2 = prxAtributo2.Valor();
167 //a2 deve ser uma sequencia recursiva dos seguintes valores:
168 //a2 = [ "Teste". [ 123, "Teste2" ] ]
169
170 //Atualizacao de atributo
171 adptAtributo1.AtualizaValor( "Teste□Atributo1" );
172
173
174 //Interacao com Eventos
175
176 //Disparo de Evento
177 adptEvento1.Dispara( 12.345 );
178
179
180 //... Removido codigo para sincronizar o termino do Participante...
181
182 //Desconecta canal
183 canal->Desconecta();
184
185
186 //Finaliza participante
187 delete app;
188 }

```