Particionamento Transparente de Ambientes Virtuais Distribuídos

Marcos Alves

DISSERTAÇÃO APRESENTADA
AO
INSTITUTO DE MATEMÁTICA E ESTATÍSTICA
DA
UNIVERSIDADE DE SÃO PAULO
PARA
OBTENÇÃO DO GRAU DE MESTRE
EM
CIÊNCIA DE COMPUTAÇÃO

Área de Concentração: **Ciência da Computação**
Orientador: **Prof. Dr. Markus Endler**

Durante a elaboração deste trabalho o autor recebeu apoio financeiro do CNPq

-São Paulo, abril de 2000-

Particionamento Transparente de Ambientes Virtuais Distribuídos

Este exemplar corresponde à redação
final da dissertação devidamente corrigida
e defendida por Marcos Alves
e aprovada pela comissão julgadora.

São Paulo, 25 de Abril de 2000.

Banca Examinadora:

Prof. Dr. Markus Endler (orientador) – IME-USP

Prof. Dr. Marco Dimas Gubitoso – IME-USP

Profa. Dra. Tereza Kirner – UFSCar

Resumo

Esta dissertação trata de uma técnica para a construção de Ambientes Virtuais Distribuídos (*AVDs*) compartilhados por muitos usuários que visa reduzir o número de interações entre os processos distribuídos. A principal contribuição deste trabalho consiste do desenvolvimento de protocolos distribuídos para uma arquitetura de servidores descentralizados que permitem um particionamento de um ambiente virtual em regiões (domínios) regulares, de maneira que a migração de elementos (atores) de um domínio para outro seja praticamente imperceptível para o usuário de um *AVD*.

Abstract

This master's thesis describes a technique for constructing Distributed Virtual Environments (*DVEs*) shared by many users, which aims at reducing significantly the amount of interaction among the distributed processes. The main contribution of this work is the development of distributed protocols for an architecture based on decentralized servers, which allows the partitioning of the virtual environment into regular regions (domains) and the transparent migration of actors between domains.

À minha família, em especial, aos meus pais Zeca e Beth e minha noiva Sil.

Agradecimentos

Agradeço primeiramente a Deus pela proteção e por nunca me ter desamparado.

Ao meu orientador, Markus, pela infinita paciência, dedicação e sabedoria com que sempre me orientou.

Aos meus pais que, sempre presentes, me acompanharam e me incentivaram a levar em frente meus ideais.

À minha amada Sil, fiel companheira que sempre esteve ao meu lado me ajudando a enfrentar todos os desafios aos quais fui submetido. Obrigado minha vida pelo teu amor que me fortalece cada dia mais.

Aos meus amigos e colegas pelo apoio e companheirismo.

Sumário

1 Introdução **5**

1.1 Modelos de Comunicação 6

1.1.1 Modelo Centralizado 6

1.1.2 Modelo Distribuído 7

1.1.2.1 *Broadcast e Multicast* 8

1.1.2.2 *Dead Reckoning* 9

1.2 Modelo Conceitual 11

1.3 Motivação para Nosso Trabalho 11

2 Trabalhos Relacionados **14**

2.1 MASSIVE 14

2.2 DIVE 17

2.3 AGORA 18

2.4 Comparação entre os Sistemas 22

3 Principais Conceitos, Arquitetura e Problemas **25**

3.1 Domínio 25

3.2 Servidor de Pertinência 26

3.3 Servidor de Domínio 26

3.4 Região de Fronteira e Aura 27

3.5 Comunicação entre Atores 27

3.6 Comunicação entre Visualizadores e Servidores de Pertinência 28

3.7 Problemas 28

3.7.1 Migração Transparente entre Domínios 28
3.7.2 Detecção de Colisão entre Atores e com Região de Fronteira 28
3.7.3 Falhas em Servidores de Pertinência 29
3.7.4 Superpopulação de um Domínio 30

4 Protocolos usados no *AVD-PTD* **31**

4.1 Protocolo para Inclusão de um Novo Ator 33
4.2 Protocolo para Exclusão de um Ator 34
4.3 Protocolo para Migração entre Domínios 34
4.3.1 Protocolo para Entrada na Região de Fronteira 35
4.3.2 Protocolo para Mudança de Domínios 36
4.3.3 Protocolo para Saída da Região de Fronteira 37
4.3.4 Identificação do Domínio Destino 37
4.4 Discussão 38

5 O Protótipo do *AVD-PTD* **39**

5.1 Javaroids 39
5.1.1 Classe *Game Manager* 42
5.1.2 Classe *Ship Manager* 42
5.2 O Protótipo 43
5.2.1 Configuração Inicial 44
5.2.2 Fase de Preparação do Ambiente 45
5.2.3 Fase de Jogo 45
5.3 Implementação do Protótipo 46
5.3.1 Interação Entre Elementos 46
5.3.2 Componentes de Software do Protótipo 47
5.3.2.1 *GameManager* 48
5.3.2.2 *SPManager* 48
5.3.2.3 *SuperServerManager* 49
5.3.2.4 *ShipManager* 49
5.3.2.5 *ControlShipManager* 49
5.3.2.6 *Domínio* 50

5.3.2.7 *Comunicação* ... 50
5.3.2.8 *SendDomainServer* ... 50
5.3.2.9 *SendDataServer* ... 51

6 Testes **52**

6.1 Teste n.º 1: Efeito visível da migração para o usuário ... 54
6.2 Teste n.º 2: Tempo gasto por ações executadas durante o protocolo de migração entre domínios ... 56
6.3 Teste n.º 3: Influência do compartilhamento dos servidores comuns na visão do usuário ... 58

7 Conclusão **61**

Apêndice A – Hierarquia de Classes – Protótipo **64**

Apêndice B – Descrição de Classes e Métodos **67**

B.1. Classe *Ator* ... 67
B.2. Interface *Bounds* ... 67
B.3. Classe *ContainerSprite* ... 68
B.4. Classe *ControlShipManager* ... 68
B.5. Classe *DataShipClass* ... 69
B.6. Classe *DataSS* ... 69
B.7. Classe *Dimension* ... 69
B.8. Classe *Dominio* ... 70
B.9. Classe *EffManager* ... 70
B.10. Classe *Explosion* ... 71
B.11. Classe *Fire* ... 71
B.12. Classe *GameManager* ... 72
B.13. Classe *GameMath* ... 73
B.14. Classe *GhostMovPol* ... 73
B.15. Classe *GhostRotPol* ... 74
B.16. Classe *GhostShip* ... 74
B.17. Classe *Jogo* ... 75

B.18. Classe *MoveablePolygon* 75
B.19. Classe *MulticastGroup* 76
B.20. Classe *NeighborView* 76
B.21. Classe *Objeto* 77
B.22. Classe *PolygonSprite* 78
B.23. Interface *Protocol* 78
B.24. Classe *PrototypeFrame* 79
B.25. Classe *ReceiveThread* 79
B.26. Classe *RotateablePolygon* 80
B.27. Classe *SendData* 80
B.28. Classe *SendDataServer* 81
B.29. Classe *SendDomain* 81
B.30. Classe *SendDomainServer* 81
B.31. Classe *Sender* 82
B.32. Classe *SenderDSS* 82
B.33. Classe *Ship* 83
B.34. Classe *ShipManager* 83
B.35. Classe *Sp* 86
B.36. Classe *SPManager* 86
B.37. Classe *SPProxy* 87
B.38. Classe *Sprite* 88
B.39. Classe *Sprite2D* 88
B.40. Classe *SuperServerManager* 88
B.41. Classe *TCPConnection* 89
B.42. Classe *Vizinho* 89

Referências **91**

Capítulo 1

Introdução

Um *Ambiente Virtual* (*AV*) é um programa que simula em tempo real um mundo (ou espaço) imaginário, no qual usuários se movem em um *mundo virtual* (*MV*) e interagem com objetos representados neste mundo por elementos gráficos. Um *Ambiente Virtual Multi-usuário* (*AVM*) é uma aplicação onde vários usuários estão simultaneamente presentes interagindo em um mesmo espaço simulado. Em *Ambientes Virtuais Distribuídos* (*AVDs*) estes usuários acessam um *AVM* através de computadores dispersos espacialmente, mas interconectados através de uma rede (por exemplo, uma *LAN* ou *WAN*). Uma característica de *AVs* é a habilidade de oferecer modelos intuitivos de interação, similares às formas com que seres humanos se comunicam ou manipulam objetos no mundo real [7].

Um mundo virtual é tipicamente povoado por objetos animados ou não. Objetos inanimados são representações de artefatos tais como quadros, paredes, mesas, etc. Os objetos animados são chamados de *atores*. Esses atores são controlados por usuários ou por um programa (robôs). Atores podem executar ações no *MV*, como por exemplo modificar objetos e/ou enviar mensagens para outros atores dentro do *MV*. No *AV* mantém-se em uma base de dados todas as propriedades correntes de cada objeto (animados ou não), tais como sua posição, velocidade, forma, cor, etc.

Na próxima seção apresentaremos possíveis modelos de comunicação para *AVDs*, destacando as características mais relevantes de cada um dos modelos.

1.1 Modelos de Comunicação

A escolha adequada de um modelo de comunicação para um *AVD* é um fator muito importante em seu projeto. É necessário estabelecer um compromisso entre a latência da comunicação e a confiabilidade do sistema, que tem influência significativa sobre o tempo de resposta a uma ação do usuário. Se a latência não tiver grande influência sobre o realismo da simulação, então o modelo que provê as melhores garantias de confiabilidade deverá ser escolhido. Mas os *AVDs*, como jogos interativos, deixam a desejar se não garantirem tempos de resposta curtos, de forma a simular de maneira realista uma interação do mundo real. Este requisito sugere o uso de protocolos menos confiáveis e uma estrutura de comunicação onde a perda de mensagens não se torna uma catástrofe, mas pode ser compensada por mensagens subseqüentes.

Gossweiler [6] descreve dois métodos para a implementação de um *AVD*, apresentados a seguir.

1.1.1 Modelo Centralizado

Neste modelo, um computador (servidor centralizado) coleta as atualizações referentes aos movimentos e interações de atores controlados por usuários em diferentes computadores, armazena as alterações em uma estrutura de dados (Base de Dados Centralizada), e envia as atualizações de volta para cada computador participante, que por sua vez interpreta os dados recebidos. A figura 1.1 ilustra o processamento típico feito no servidor centralizado.

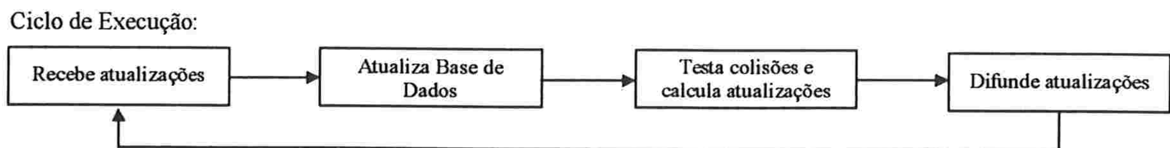


Figura 1.1 – Ciclo de Execução de um Servidor Centralizado

A figura 1.2 ilustra um modelo centralizado.

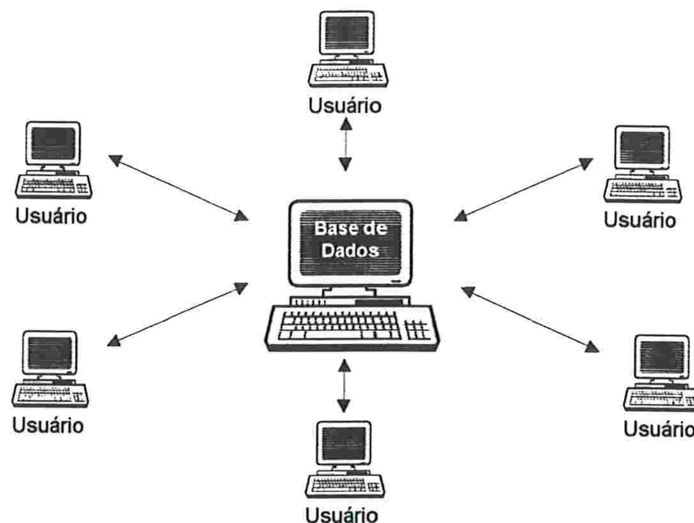


Figura 1.2 – Modelo Centralizado

Embora este modelo permita ao programador desenvolver uma estrutura de dados e protocolos simples para representar e manipular objetos no *MV*, o mesmo não é escalonável. Com a entrada de muitos usuários na simulação, o computador centralizado torna-se um gargalo no sistema. Isto faz com que, à medida que cresce o número de usuários, estes tenham que esperar cada vez mais tempo para receber as informações da base de dados do *MV*. Além disso, este modelo causa uma latência maior para a propagação de uma atualização entre os usuários participantes da simulação, pois cada atualização precisa percorrer dois caminhos: do usuário para o servidor centralizado, e do servidor centralizado para os demais usuários.

1.1.2 Modelo Distribuído

O modelo distribuído apresenta uma abordagem mais escalonável do que o modelo centralizado. Neste modelo, o *AV* é um programa distribuído, no qual cada processo controlado por um usuário[1] mantém localmente uma cópia completa da base de dados, e também é responsável pela manipulação de objetos animados ou não. Sempre que uma destas cópias locais é alterada, por exemplo devido a uma movimentação do ator, o visualizador correspondente envia atualizações para todos os demais visualizadores do *AV* a fim de manter um estado consistente da base de dados. Diferentemente do modelo com servidor centralizado, aqui o gargalo passa a ser a

[1] Daqui para frente usaremos o termo visualizador com o significado de processo controlado por um usuário.

rede, devido à existência de muitas conexões e/ou troca de mensagens. A figura 1.3 ilustra um modelo de comunicação distribuído.

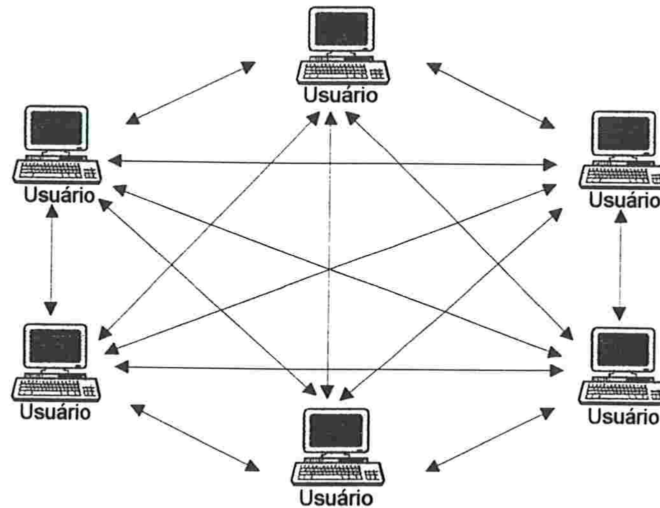


Figura 1.3 – Modelo Distribuído

Para reduzir o número de conexões e o número de mensagens enviadas, duas técnicas podem ser empregadas: *broadcast network* (*multicast*) e *Dead Reckoning*. Estas técnicas são apresentadas nas próximas seções.

1.1.2.1 *Broadcast e Multicast*

Broadcast é uma difusão de mensagens pela rede, a qual permite que todos os computadores na rede recebam cada mensagem enviada. Isto evita que, em uma simulação com *n* usuários, seja necessário estabelecer (*n-1*) conexões ou enviar *n* mensagens a fim de informar cada visualizador associado a usuário de modificações na base de dados local; a alteração é transmitida uma única vez e todos os visualizadores podem obtê-la diretamente, como visto na figura 1.4.

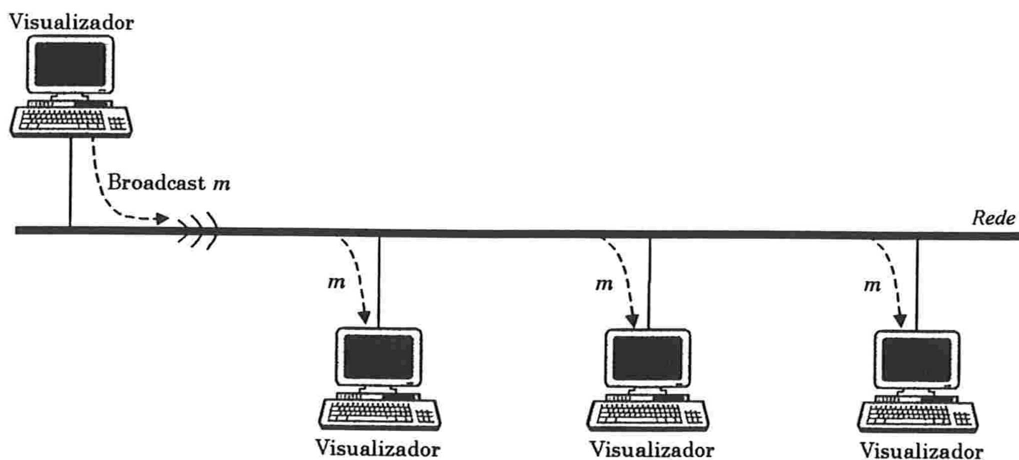


Figura 1.4 – Comunicação distribuída com uso de *broadcast*

No caso específico de *AVDs* utilizando o modelo distribuído, o uso de *broadcast* reduz significativamente o número de conexões e mensagens necessárias para manter atualizadas as cópias locais da base de dados. A programação também é simplificada, pois o visualizador de um usuário que entra para participar da simulação não precisa estabelecer uma comunicação com os visualizadores de outros usuários participantes. Em vez disso, basta que o mesmo seja informado sobre qual canal de *broadcast* ele deve usar.

Multicast é simplesmente um caso particular de *broadcast*, no qual é possível formar grupos de usuários que devem interagir por difusão de mensagens. Desta forma é possível enviar pacotes de dados para um conjunto específico de destinatários em uma única operação [13]. A família de protocolos *TCP/IP* possibilita o uso de endereços *multicast* para redes locais [2]. O protocolo *multicast* (*IGMP*) é comumente utilizados em *AVDs* com dois objetivos principais:

- Limitar o número de participantes em um mesmo escopo de recepção de mensagens, eliminando, desta forma, as trocas de informações irrelevantes, ou seja, entre visualizadores de usuários que não estão interagindo.
- Economizar largura de banda passante, uma vez que o fluxo de mensagens é reduzido.

1.1.2.2 *Dead Reckoning*

Dead Reckoning é uma técnica adicional usada em *AVDs* com modelo de comunicação distribuído para reduzir o número de mensagens de atualização. Esta técnica é baseada em uma projeção do movimento futuro de cada ator associado a um usuário. Mesmo utilizando *broadcast* ou *multicast* pode ocorrer que, se houverem muitos usuários em uma simulação e o visualizador de cada usuário enviar uma mensagem toda vez que o ator a ele associado sofrer uma alteração, isto gere um número muito grande de mensagens na rede. Ou seja, por exemplo em um jogo distribuído com n usuários, cada um controlando seu próprio ator. A cada movimento de um ator, os visualizadores dos outros (n-1) usuários são informados da mudança de posição através de um *broadcast*. Se todos os atores se moverem simultaneamente, teremos então um total de (n-1) mensagens geradas para cada um dos n usuários. Portanto, com n grande, o sistema pode ser inundado por mensagens.

Para reduzir a quantidade de mensagens enviadas, o visualizador de cada usuário poderia enviar, além da posição atual de seu ator, um vetor velocidade (e possivelmente um vetor aceleração[2]). Então, se o ator mantiver seu movimento na direção e sentido estabelecidos previamente por sua posição e velocidade, não há necessidade de enviar qualquer alteração aos visualizadores dos demais usuários participantes, pois os respectivos visualizadores podem calcular a posição do ator em movimento a partir da posição anterior e da velocidade atual. Este cálculo de posição baseado na última velocidade conhecida é chamado de *Dead Reckoning* [6]. A figura 1.5 mostra a defasagem entre as trajetórias real e projetada de um ator (representado na figura por um triângulo) em um espaço bidimensional.

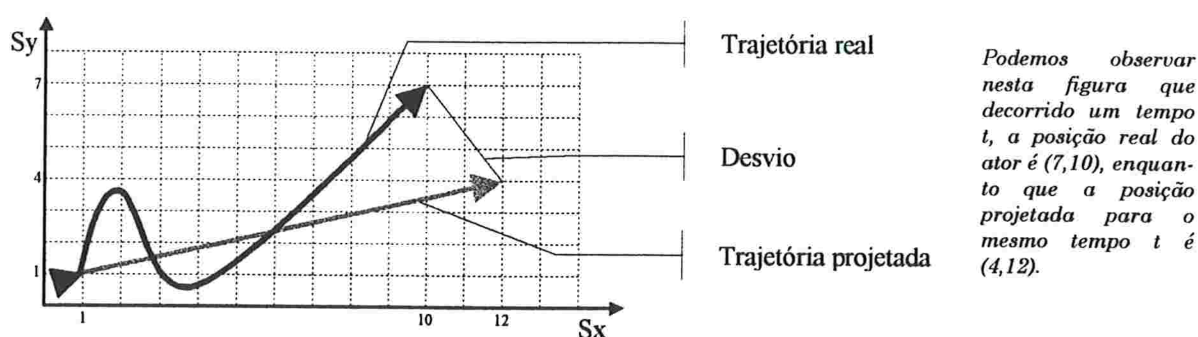


Figura 1.5 – Trajetória real *versus* trajetória projetada

Para determinar se um ator controlado por um usuário teve uma mudança significativa de posição, o visualizador do próprio usuário aplica o algoritmo de *Dead Reckoning* sobre seu ator, e compara a diferença entre a posição projetada e a posição real. Se a diferença ultrapassa um certo limite considerado significativo, então o visualizador envia uma mensagem de atualização para que os visualizadores dos demais usuários apliquem a nova posição e velocidade às suas cópias locais do ator (também denominado de *ator-fantasma*).

Para efetivar este modelo distribuído de simulação, assume-se que cada usuário controla um único ator através de um programa (visualizador). Este visualizador realiza a simulação para todos os objetos em sua cópia da base de dados. Um visualizador pode controlar um ator, e deverá usar *Dead Reckoning* para representar a movimentação dos demais *atores-fantasmas*, que são controlados pelos visualizadores de outros usuários.

---

[2] Este torna-se necessário quando o *AVD* permite alterar a velocidade de atores.

1.2 Modelo Conceitual

Nesta seção apresentaremos o modelo conceitual para nosso trabalho. O objetivo deste modelo é apresentar os termos básicos e conceitos usados no nosso trabalho, como por exemplo usuários, atores, etc. Como podemos observar na figura 1.6, um usuário está sempre associado a um ator no mundo virtual. O *visualizador*, representado na figura por um retângulo, é um programa usado para controlar a movimentação de um ator e exibir a visão deste do mundo virtual e dos objetos nele presentes. O *visualizador* é controlado pelo usuário e implementa rotinas de gerenciamento do ator associado ao usuário e dos atores-fantasmas (correspondentes aos atores dos outros usuários remotamente presentes no mundo virtual), e rotinas para comunicação com os demais *visualizadores*. O mundo virtual (*MV*) é representado por estruturas de dados replicadas em cada um dos *visualizadores*. E o *AVD* é o programa distribuído que implementa o *MV*, constituído do conjunto de *visualizadores*, que se comunicam através da rede.

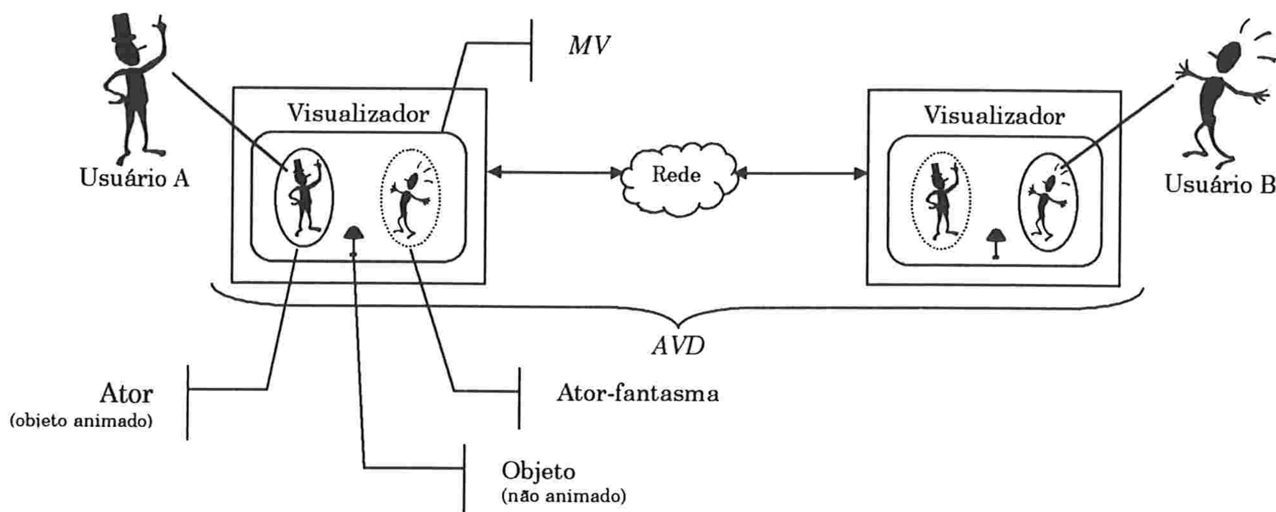


Figura 1.6 – Modelo Conceitual

1.3 Motivação para Nosso Trabalho

Quando o número de usuários participantes em uma simulação torna-se muito grande, pode surgir o problema de aumento considerável de mensagens na rede, mesmo utilizando-se comunicação *multicast* e *Dead Reckoning*. As mensagens às quais nos referimos são aquelas que tratam da atualização da visão que cada usuário tem do

MV. Além disso, as rotinas gráficas necessárias para a animação de todos os objetos do *MV* tipicamente consomem grande parte do tempo de processamento da aplicação.

Para melhor situar o problema, suponha que o número de usuários participantes da simulação na figura 1.6 seja muito grande. Manter consistente todas as visões do *MV* para todos os usuários pode tornar-se um problema, devido à latência na comunicação para a atualização das réplicas do *MV* e ao grande número de objetos. Esta situação nos leva a pensar em formas mais eficientes de atualização em um *AVD*, através da redução ainda maior do fluxo de mensagens entre os usuários participantes do *MV*. A idéia é a de limitar o conjunto de usuários informados de uma atualização, de acordo com a "proximidade" de seus atores do ator sendo atualizado. Em outras palavras, o objetivo é excluir da atualização todos aqueles usuários para os quais a atualização não tem conseqüências imediatas. Uma possível maneira de efetivar tal agrupamento é por um particionamento estático do *MV* em regiões regulares (*domínios*), onde a percepção mútua só ocorre entre os membros presentes em um mesmo domínio. Para manter uma visão consistente entre os elementos integrantes de um mesmo domínio é interessante o uso de grupos *multicast* e da técnica de *Dead Reckoning*.

Uma possível forma de particionamento do conjunto de atores em grupos poderia ser aquela determinada pela relação de vizinhança entre os atores. Esta relação de vizinhança pode ser definida diferentemente para cada meio de interação (áudio, vídeo, etc.), e no caso geral pode variar muito freqüentemente, dificultando a implementação da percepção mútua entre atores. Em nossa abordagem adotamos um agrupamento simples, baseado em um particionamento estático do *MV* em domínios e em uma co-localização de atores em um domínio. Por exemplo, poderia-se usar tal particionamento em um mundo virtual representando uma casa com vários cômodos. A interação entre os atores seria então restrita aos atores que se encontrassem em um mesmo cômodo. Quando estes atores se aproximassem de uma porta ou janela, os mesmos passariam a perceber a movimentação de atores no outro cômodo.

Estamos cientes que este agrupamento de atores de acordo com sua pertinência a um domínio pode não ser adequado para determinados ambientes simulados, especialmente quando deseja-se simular a interação através de vários meios. No entanto, um dos objetivos do nosso trabalho é o de mostrar que tal particionamento

transparente de um *MV* é factível, e pode ser implementada de forma distribuída, e que de fato reduz o fluxo de mensagens em um mundo virtual com muitos atores.

Desenvolvemos então uma arquitetura distribuída que implementa esta abordagem, a qual denominamos Ambiente Virtual Distribuído com Particionamento Transparente em Domínios – *AVD-PTD*. No próximo capítulo veremos os trabalhos relacionados estudados, e nos capítulos seguintes apresentaremos em detalhes os principais conceitos, problemas e propostas de solução para estes problemas.

Capítulo 2

Trabalhos Relacionados

Vários *AVDs* têm sido propostos e implementados nos últimos anos, tais como MASSIVE [5], DIVE [7], AGORA [9], Diamont Park and Spline [14], PARADISE [11], etc. Apresentaremos agora os trabalhos mais relacionados com o nosso trabalho. São eles os sistemas MASSIVE, DIVE e AGORA. A última seção (2.4) deste capítulo apresenta um quadro comparativo entre estes trabalhos e a nossa abordagem.

2.1 MASSIVE

MASSIVE é um sistema experimental de Realidade Virtual Distribuída [5] desenvolvido por Chris Greenhalgh e Steve Benford no Departamento de Ciência da Computação da Universidade de Nottingham.

MASSIVE ("Modelo, Arquitetura e Sistema para Interação Espacial em Ambientes Virtuais") dá uma ênfase particular a ambientes virtuais multi-usuários de grande escala. Ou seja, ambientes que permitem seu uso por centenas ou milhares de usuários simultaneamente.

Apresentamos, a seguir, os aspectos centrais do modelo espacial de interação do MASSIVE[3]. Visando facilitar a escalabilidade do *AVD*, utiliza-se o conceito de *aura* de um objeto que caracteriza a sua vizinhança imediata. Cada objeto[4] em um mundo virtual possui uma aura específica de interação com outros objetos, que depende da

---

[3] Visto em [5] e em [12].

[4] No MASSIVE, objeto significa o mesmo que ator (descrito na seção 1.2 do capítulo anterior).

mídia utilizada (por exemplo, gráficos, textos, áudio, etc.). Esta aura define uma vizinhança espacial na qual é possível a interação com outros objetos. Ou seja, uma aura determina quais outros objetos um determinado objeto poderá perceber visualmente, auditivamente, por meio de colisão, etc. A dimensão e forma de uma aura em princípio podem variar dinamicamente e podem ser função da posição e, possivelmente, de outros atributos do objeto, tais como o grau de visão do objeto. Considere por exemplo, um objeto que se move para uma região do mundo virtual que supostamente tem menos luminosidade. Espera-se que a dimensão de sua aura diminua, pois o grau de visão do objeto também diminui. Define-se que a interação entre dois objetos é possível somente quando existe uma intersecção de suas auras ou um objeto se encontrar dentro da aura do outro. Neste caso, ocorre o que é denominado *colisão de auras*.

O emprego de auras facilita a implementação de mundos virtuais com muitos usuários, dado que o número de interações (entre objetos) a serem tratadas é bem menor, pois só é necessária para os objetos próximos uns dos outros. O número de interações a serem tratadas dependerá somente da dimensão da aura de cada objeto e da densidade de objetos no mundo virtual. Assim, o mundo virtual pode ter um número arbitrário de objetos sem que isto aumente necessariamente a quantidade de processamento.

A interação entre objetos é implementada através da combinação da técnica de detecção de colisões usada em realidade virtual com o conceito de serviço de nomes baseado em atributos (*trader*) [1]. Esta forma de comunicação é denominada de *negociação espacial.*

Para entender como funciona a negociação espacial em MASSIVE, considere 2 objetos em um mundo virtual. Ao entrar em um mundo, o primeiro objeto contacta o *trader*, também chamado de *Gerenciador de Aura* (*GA*), para registrar neste gerenciador suas interfaces para interação (por exemplo: áudio, vídeo, texto, etc.) como sendo serviços implementados por ele, cada qual definindo sua própria aura. Cada *GA* manipula um ou mais mundos, cada qual com seu próprio conjunto de mídias (por exemplo, áudio, vídeo, texto, etc.). Como objetos podem mudar de mundo, estes podem ser passados de um *GA* para outro. Um segundo objeto que esteja entrando no mundo, passa pelo mesmo processo de registro em um *GA*. A cada instante o *GA* monitora a

posição e as auras relevantes de todos os objetos conhecidos por ele. Quando o *GA* detecta uma colisão de auras, ele passa referências de interfaces mútuas aos objetos envolvidos, fazendo com que estes sejam capazes de estabelecer uma conexão para interação direta (por exemplo, um UNIX ***stream*** para comunicação de áudio). Esta situação é ilustrada na figura 2.1. Nesta figura, podemos observar dois usuários e seus respectivos objetos. No momento em que o Gerenciador de Auras detecta que houve uma colisão entre suas auras, uma conexão de áudio é estabelecida entre os dois usuários para que os mesmos iniciem a interação.

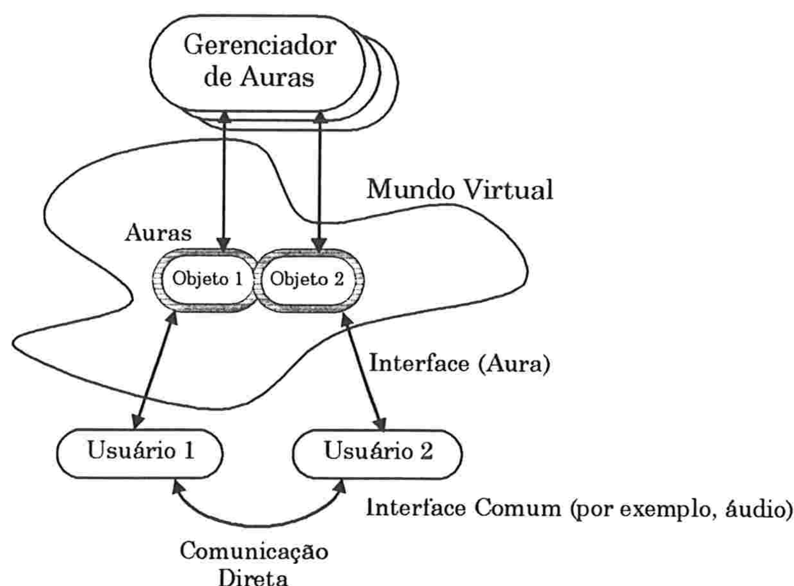


Figura 2.1 – Negociação Espacial envolvendo pares de objetos e *GA*

A passagem de objetos de um mundo para outro é feita através de *portais*. Portais, também denominados *gateways*, são pontos de passagem entre diferentes mundos, ou diferentes localizações dentro do mesmo mundo. Portais implementam uma interface especial com o usuário, através da qual o mesmo pode escolher se deseja passar para o outro mundo. Os portais incluem um atributo de localização do destino. Esta localização do destino pode incluir a identificação de um novo mundo. Quando um objeto (*OBJ*) declara para o seu Gerenciador de Auras que ele deseja mudar para um novo mundo, o Gerenciador de Auras primeiro notifica os outros objetos que *OBJ* está deixando o mundo atual, e então este passa o controle do *OBJ* para outro Gerenciador de Auras, responsável pelo novo mundo. O tratamento dado ao *OBJ* na chegada ao novo mundo é o mesmo de quando um novo objeto ingressa no sistema.

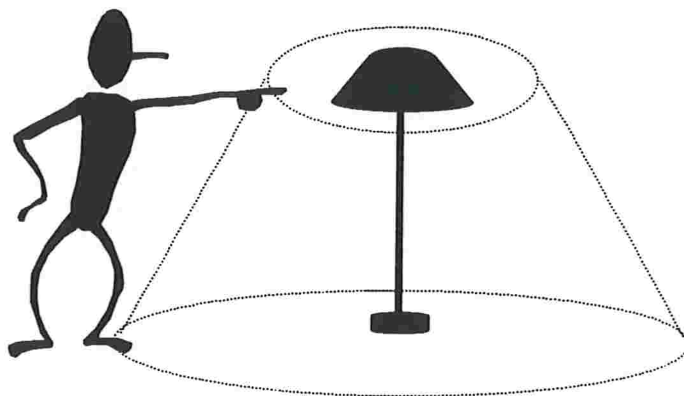
Portanto, em MASSIVE, o problema da escalabilidade é aliviado através da distribuição da responsabilidade pela detecção de colisão de auras entre todos os Gerenciadores de Auras. No entanto, se caso um número excessivo de objetos estiver em um mesmo mundo, MASSIVE poderá apresentar problemas de contenção no Gerenciador de Auras do respectivo mundo.

2.2 DIVE

DIVE (Ambiente Virtual Distribuído Interativo) [7] é uma plataforma experimental para o desenvolvimento de ambientes virtuais em três dimensões, interfaces com o usuário e aplicações baseadas em ambientes compartilhados. DIVE é voltado principalmente para aplicações multi-usuário, onde vários participantes interagem entre si através da rede Internet.

DIVE foi desenvolvido por Olot Hagsand e Christer Carlsson do Instituto de Ciência da Computação da Suécia.

Mundos virtuais em DIVE dão suporte para uma interação dinâmica. Ou seja, os objetos podem reagir à estímulos, mover-se, transformar-se e adaptar-se à mudanças de ambiente, como no exemplificado na figura 2.2.



Um ator se aproxima de uma lâmpada e intercepta sua aura. Como resultado, a lâmpada se acende automaticamente.

Figura 2.2 – Interação dinâmica em um mundo virtual

Sendo a detecção de colisão um serviço central para a maioria dos ambientes virtuais, a detecção de colisão também forma a base de muitas funções em DIVE, como por exemplo a interação entre objetos. A detecção de colisão é similar à usada no sistema MASSIVE, onde um objeto registra interesse em certos objetos (neste caso,

atores), e o gerenciador de colisões gera sinais para os atores que tiveram sua aura interceptada.

A interface com o usuário é implementada através de um *visualizador*. Um visualizador é um programa com um módulo para síntese gráfica em 3D que representa a visão atual do mundo virtual da perspectiva de um ator. Além de fornecer uma representação gráfica do mundo virtual, o visualizador do DIVE também permite que atores interajam entre si, por exemplo enviando mensagens para outros atores, ou estabelecendo conexões de áudio.

Em DIVE a comunicação direta entre atores pode ocorrer através de texto ou áudio, quando o equipamento possui recursos de multimídia, tais como microfone e alto-falantes. DIVE é baseado em uma arquitetura de comunicação direta entre pares de clientes (*peer-to-peer*) sem servidor centralizado, onde os pares se comunicam através de *multicast* confiável e *multicast* não confiável, baseado em *IP-multicast*. Conceitualmente, o estado do mundo virtual compartilhado entre atores pode ser visto como uma memória compartilhada em rede, onde um conjunto de processos interage no acesso concorrente à memória e todos tem exatamente a mesma visão do estado. Para tal, todo usuário DIVE armazena localmente uma cópia completa do mundo virtual com seu estado atual. O controle de concorrência e consistência de estado (na forma de objetos comuns) é feito através de Replicação Ativa [3] e protocolos *multicast* confiáveis. Ou seja, todos os objetos que compõe o mundo virtual são replicados em todos os programas usuário, onde a réplica é mantida consistente com as demais por meio de um protocolo de *multicast* atômico e ordenação total de mensagens. Através deste protocolo todas as réplicas executam exatamente a mesma seqüência de atualizações. Por exemplo, se dois usuários movem os seus respectivos atores simultaneamente, então as cópias locais do mundo virtual serão atualizadas, refletindo os dois movimentos com um mínimo de defasagem.

2.3 AGORA

AGORA [9] é um projeto dos Laboratórios Fujitsu do Japão. Seus principais colaboradores são Hiroaki Harada, Naohisa Kawaguchi, Akinori Iwakawa, Kazuki Matsui e Takashi Ohno. O objetivo do AGORA é prover uma tecnologia de comunicação para comunidades através de espaços compartilhados em equipamentos

disponíveis aos usuários, tais como PCs, modems, linhas telefônicas, etc. AGORA apresenta uma arquitetura para uma comunidade virtual tridimensional, que emprega uma técnica de divisão e gerenciamento do espaço virtual, um modelo de controle do fluxo de mensagens de atualização da visão, e uma técnica de compartilhamento do espaço virtual entre clientes[5] (*space-sharing*). O objetivo do projeto foi desenvolver um sistema de comunicação flexível e generalizado combinando o acesso a *sites* na *World Wide Web* (WWW), e a comunicação em tempo real, como por exemplo salas de bate-papo.

O projeto AGORA teve como motivações: a) permitir que usuários se comuniquem livremente em um *AVD*; b) prover *AVDs* com suporte para PC, e comunicação via modems; c) fazer um balanço de cargas para equilibrar as funções usadas para criar comunidades e a diminuição da carga da rede.

AGORA pode ser classificado como um sistema de banco de dados centralizado e compartilhado, que foi construído como um modelo cliente-servidor tipo mestre-escravo (*master-slave*). Neste modelo, um servidor gerencia a comunidade (*servidor da comunidade* – CS), e clientes podem navegar pelo espaço virtual através de um *navegador da comunidade* (CB). O protocolo de comunicação usado entre servidor e clientes é chamado *protocolo da comunidade* (COMMP). A base de dados que representa o *MV* não reside somente no CS, existe também uma cópia em cada cliente. Contudo, somente o CS mantém a consistência dos dados. Ou seja, toda vez que um cliente deseja modificar a base de dados, ele tem que enviar um pedido ao CS, que realiza a modificação e a difunde para todos os clientes, para que os mesmos possam atualizar suas cópias locais da base de dados.

[5] No AGORA, cliente significa o mesmo que ator (descrito na seção 1.2 do capítulo anterior).

O modelo de troca de mensagens entre um CS e um CB é apresentado na figura 2.3 e os tipos de mensagens apresentados nesta figura são explicados no quadro 2.1. Uma característica importante do modelo de mensagens do AGORA é que a troca de mensagens só ocorre entre clientes localizados em um mesmo espaço comum, denominado *região*.

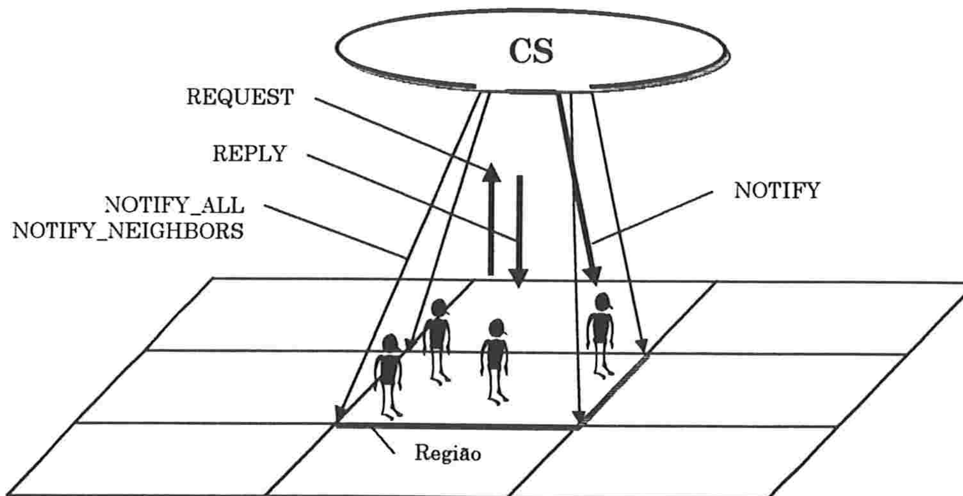


Figura 2.3 – Ilustração de *região* e modelo de mensagens

REQUEST	Pedido de um CB para o CS. CS responde ao CB.
REPLY	Resposta do CS para o CB.
NOTIFY NOTIFY_ALL NOTIFY_NEIGHBORS	Notifica um CB específico. Notifica todos os CBs na região. Notifica todos os CBs na região, com exceção do CB que enviou um pedido.

Quadro 2.1 – Tipos de Mensagens

AGORA utiliza a técnica de *space-sharing* (*SSh*) para manter cópias locais em clientes consistentes. Através do *SSh* é possível fazer uma notificação imediata de qualquer evento em um espaço virtual para todos os clientes. AGORA apresenta uma técnica que divide o espaço de comunicação em tamanhos fixos; e faz uso de um servidor VRML interativo para manter o espaço virtual de comunicação. Veremos a seguir, estas técnicas em maiores detalhes.

- *Divisão do Espaço Virtual em Regiões*: AGORA não usa o conceito de aura, como nos sistemas MASSIVE e DIVE, por assumir que em uma comunidade virtual, ao contrário de um *AV*, o número de clientes participantes é pequeno e a chance de colisão entre os mesmos é bastante remota. Assim, o conceito de comunidade não requer o uso de uma área controlada por uma aura. Em vez disso, o AGORA aplica a técnica de particionamento em regiões, que são áreas de tamanho fixo, ao invés da técnica dinâmica de aura, por considerar que é muito difícil estabelecer o tamanho ideal de auras de tal maneira que colisões ocorram mas não haja uma sobrecarga excessiva na detecção de colisões. Uma região corresponde a uma *URL* é a menor unidade disponível para espaços compartilhados e comunicação. O espaço da comunidade engloba várias regiões, como pode ser visto na figura 2.3.

- *Servidor VRML interativo*: para assegurar a consistência do espaço virtual como um todo, o AGORA inclui um *Gerenciador de Objetos* (*I-VRML*) que mantém o estado efetivo do *MV*, gerenciando o espaço compartilhado na forma de um banco de dados no qual é possível acrescentar e remover objetos. A figura 2.4 ilustra a seqüência de interações entre CBs, CS e *I-VRML* quando um CB deseja mudar o seu estado no espaço virtual. Os eventos ou mensagens (interações), representados na figura 2.4 por flechas, indicam:
  - (1): Um CB que mudou seu estado no espaço virtual envia uma mensagem para o CS, generalizando o evento.
  - (2): O CS envia esta mensagem para o servidor *I-VRML*.
  - (3): O *I-VRML* dispara um processo correspondente à mensagem que altera o estado do espaço virtual.
  - (4): O CS é então notificado do resultado deste processo.
  - (5): O CS transmite através de um *broadcast* a mensagem para todos os clientes (CBs).
  - (6): O servidor *I-VRML* apresenta uma cópia (*snapshot*) do estado para todos os clientes.

Quando um CB quer apresentar seus objetos para a comunidade (7), o CB coloca os objetos no servidor *I-VRML* diretamente através de um ftp, após o CB ter sido

certificado pelo CS. O resultado deste processo é transmitido pelo CS para todos os clientes na comunidade.

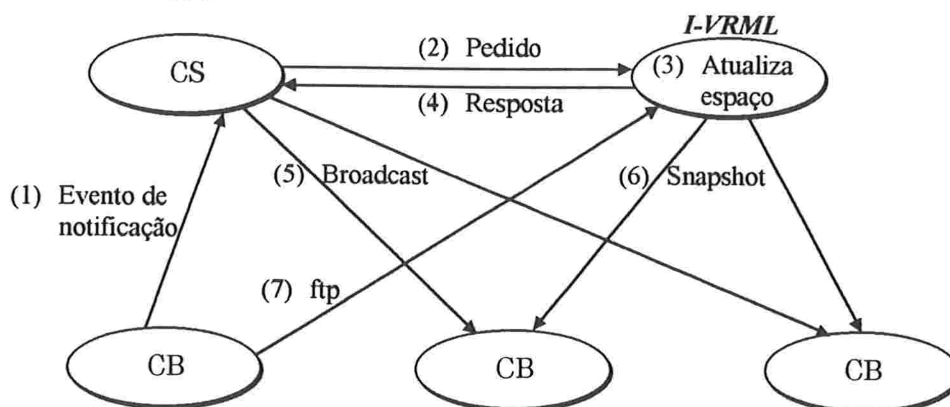


Figura 2.4 – *Space-sharing* com servidor *VRML* interativo

Para resolver os problemas de latência na entrega de mensagens que é feita pelo CS, o AGORA faz uso da técnica de *Dead Reckoning*, descrita na seção 1.1.2.2 do capítulo 1. Cada CB realiza *Dead Reckoning* para decidir quando notifica os demais CBs. Esta técnica diminui a carga da rede e sincroniza todos os CBs.

2.4 Comparação entre os Sistemas

Nesta seção apresentamos uma comparação entre os sistemas MASSIVE, DIVE, AGORA e nossa abordagem (*AVD-PTD*), com relação a algumas de suas características relativas à percepção mútua entre agentes (tais como atores, objetos, etc.), sua migração, a forma de implementação e a sincronização entre visualizadores.

	MASSIVE	DIVE	AGORA	*AVD-PTD*
Diversos meios de interação	Sim	Sim	Não	Não
Detecção de colisão	Sim	Sim	Não	Sim
Particionamento /gerenciamento de mundos	Sim	Não	Sim	Sim
Migração transparente entre regiões	Não	-	Não	Sim
Estado do mundo virtual	Distribuído	Distribuído	Servidor *I-VRML*	Distribuído
Percepção mútua entre atores	Colisão de Auras	Colisão de Auras	Pertinência à região	Pertinência à região
Sincronização de visão	-	*Multicast* para todos	*Multicast* Seletivo	*IP-multicast*

Quadro 2.2 – Comparação entre MASSIVE, DIVE, AGORA e *AVD-PTD*

Enquanto nos sistemas MASSIVE e AGORA visualizadores podem utilizar diversos meios de interação (tais como áudio, texto, etc.) para estabelecer uma comunicação entre atores, em nossa abordagem e no sistema AGORA tal facilidade não existe. Apesar de sabermos que a definição de diversos meios de interação entre atores pode aumentar o grau de realismo em um *AV* (por exemplo, atores próximos podem também ouvir uns aos outros), decidimos que em nosso trabalho adotaríamos somente um tipo de aura para atores e usaríamos a técnica de ***multicast*** como único mecanismo para interação entre os atores presentes em um mesmo domínio. Esta restrição se deve ao fato de que simular a comunicação entre atores através de diversos meios de interação aumentaria consideravelmente a complexidade dos protocolos de comunicação necessários para implementar tal interação multimídia entre atores.

Assim como no MASSIVE e DIVE, nossa abordagem implementa a detecção de colisão entre atores usando a técnica de aura. Já o AGORA assume que colisão entre atores é muito remota e não precisa ser tratada. Como veremos nos próximos capítulos, em nossa abordagem, também utilizamos a aura para detectar a aproximação de um ator da fronteira de uma região do mundo virtual.

No sistema MASSIVE e em nossa abordagem o objetivo do particionamento e gerenciamento descentralizado de mundos é garantir a escalabilidade dos mesmos. No AGORA, apesar da presença de um servidor centralizado (*I-VRML*), também existe um particionamento do mundo virtual em regiões como forma de garantir a escalabilidade do sistema. Em DIVE não há particionamento do mundo virtual e por isto o sistema é pouco escalonável.

No MASSIVE, a passagem de um mundo para outro é explicita, pois a presença de um portal de transferência para outro mundo é representado explicitamente e o usuário tem a possibilidade de escolher se deseja ou não usar o portal para se transferir para outro mundo. No AGORA a migração não é transparente pois um ator pode num instante estar interagindo com atores em um região e, no instante seguinte, pode passar para outra região, onde começa a interagir com outros atores. Em nossa abordagem a migração é transparente para o usuário, que não precisa saber da existência de várias regiões (*domínios*) no mundo virtual.

O AGORA implementa o mundo virtual através de um servidor *I-VRML* centralizado e o estado do mundo virtual fica distribuído nos clientes (usuários). No caso do MASSIVE, DIVE e em nossa abordagem o estado do mundo virtual é distribuído.

A percepção mútua entre atores em MASSIVE e DIVE ocorre somente quando há colisão de auras específicas para cada meio de interação. No sistema AGORA e em nossa abordagem a percepção mútua só é possível entre atores presentes em uma mesma região. Em nossa abordagem, no entanto, quando um ator se encontra na região de fronteira entre dois domínios, este "percebe" a presença dos outros objetos em ambos os domínios que compartilham a fronteira. Desta forma, é possível realizar uma passagem "suave" de um domínio para outro.

A sincronização do estado do mundo virtual nos visualizadores em DIVE ocorre através de mensagens *multicast* para todos os atores presentes no mundo. No AGORA a atualização da visão só ocorre entre clientes dentro de uma mesma região. Em nossa abordagem, todos os atores dentro de um mesmo domínio mantêm suas visões sincronizadas, através da técnica de *Dead Reckoning* e do envio de mensagens *multicast* de atualização.

Capítulo 3

Principais Conceitos, Arquitetura e Problemas

De forma parecida com o AGORA, descrito no capítulo anterior, usaremos a técnica de divisão do mundo em regiões fixas como forma de reduzir o fluxo de mensagens entre atores. Como um dos objetivos é o de obter uma divisão transparente do mundo virtual (*MV*) em domínios, foi necessário desenvolver um conjunto de protocolos baseados em conceitos que descreveremos neste capítulo. Ao final do mesmo discutiremos ainda alguns problemas específicos decorrentes de nossa abordagem.

3.1 Domínio

Chamaremos de *domínio* uma região regular e fixa (por exemplo, retângulo em 2D, cubo em 3D, etc.) que faz parte do *MV*. A divisão de domínios em princípio é ortogonal à qualquer subdivisão "conceitual" do mundo virtual em partes ou regiões com atributos específicos (ambientes diferentes), tais como um ambiente simulado externo e um ambiente simulado interno, diferentes lojas de um shopping, etc. Ou seja, o conceito de domínio, tal qual usado neste trabalho, serve única-e-exclusivamente para agrupar os objetos (atores) que têm uma percepção mútua, e que portanto precisam ter visões constantemente atualizadas com a movimentação dos atores presentes no mesmo domínio. Este conceito de domínio é independentes de possíveis

domínios das aplicações. Um exemplo de subdivisão em domínios de um *MV* bidimensional pode ser visto na figura 3.1.

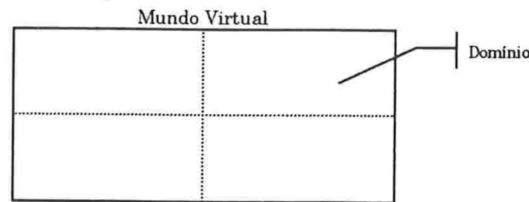


Figura 3.1 – Mundo Virtual subdividido em Domínios

3.2 Servidor de Pertinência

A cada domínio do mundo virtual *MV* está associado um servidor de pertinência (*SP*). O *SP* é responsável por gerenciar a informação sobre quais atores estão localizados no seu respectivo domínio. Na figura 3.2 podemos observar que cada domínio tem associado a si um *SP*, que mantém a lista dos atores atualmente presentes no domínio.

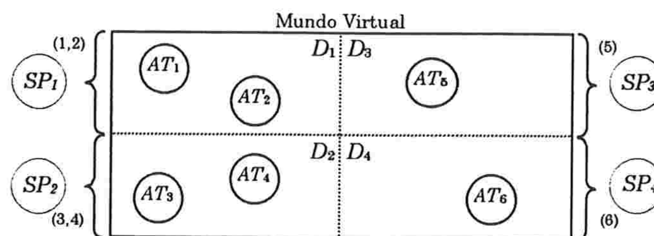


Figura 3.2 – Representação da associação de SPs com domínios no *MV*

Um servidor de pertinência mantém informações sobre os limites (fronteiras) do seu domínio. Além disso, cada servidor de pertinência mantém também as informações sobre pontos de acesso (endereços) dos servidores de pertinência responsáveis por domínios vizinhos. Estas informações são necessárias para permitir uma migração transparente de um ator de um domínio para outro.

3.3 Servidor de Domínio

A cada domínio do mundo virtual *MV* está associado também um servidor de domínio (*SD*). O *SD* é um visualizador, associado a um ator do domínio, responsável por enviar o estado atual do domínio, como por exemplo, o conjunto de objetos não animados presentes no domínio, ou outros atributos específicos usados na representação da respectiva parte do *MV*, para o visualizador de cada novo ator que entra no domínio. Quando o ator associado ao *SD* sai do domínio, cabe ao servidor de

pertinência eleger um outro ator que terá seu respectivo visualizador como o novo *SD*. Deve-se observar que qualquer um dos visualizadores dos atores presentes no domínio tem a capacidade de assumir o papel de *SD*, dado que já dispõem do estado atual do mesmo.

3.4 Região de Fronteira e Aura

A região de fronteira (*RF*) é uma faixa de largura maior que zero que percorre os limites da cada domínio. Esta região é útil para detectar quando um ator está mudando de um domínio para outro. Isto é feito, detectando-se uma "colisão" entre sua aura e a *RF*. Durante o período em que se encontra na *RF*, o visualizador do ator envia e recebe atualizações para/de todos os outros visualizadores dos atores no domínio origem e também recebe atualizações dos visualizadores dos atores participantes do domínio destino. Ao entrar em um domínio, todo ator recebe, através de seu visualizador, informações do servidor de pertinência sobre os limites do domínio. Assim, ele próprio pode detectar a colisão de sua aura com uma *RF*.

A figura 3.3 ilustra a aura (nos atores) e a região de fronteira dos domínios do mundo virtual, no momento de uma intersecção entre a aura do ator AT_5 e a *RF* entre D_3 e D_4.

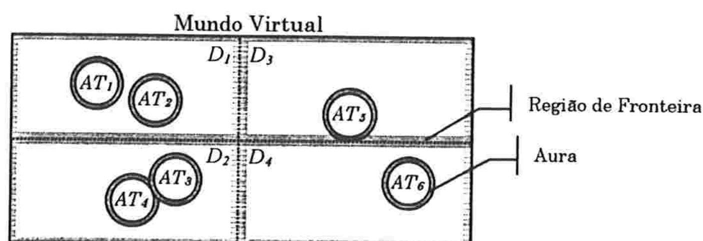


Figura 3.3 – Representação da região de fronteira de domínios e aura de atores

3.5 Comunicação entre Atores

A comunicação entre atores (através de seus visualizadores) em um mesmo domínio precisa ser eficiente e do tipo um para todos. Por isto optou-se por utilizar o *IP-multicast* para esta comunicação. Os protocolos *IP-multicast* permitem a transmissão de pacotes de dados através da rede. O *IP-multicast* possibilita o envio de um datagrama *IP* para um grupo de *hosts*, um conjunto de um ou mais *hosts* identificados por um único endereço *IP-multicast* [10].

Em nossa abordagem, cada domínio está associado a um único endereço *IP-multicast*, que é utilizado por todos os visualizadores dos atores no domínio para as atualizações mútuas. A troca de mensagens *multicast* entre eles no entanto, ocorre somente na difusão de um desvio de trajetória significativo com relação à projeção do movimento feita usando a técnica de *Dead Reckoning*.

3.6 Comunicação entre Visualizadores e Servidores de Pertinência

A comunicação entre servidores de pertinência e visualizadores deve ser orientada à conexão e ser confiável, uma vez que ela será usada para troca de dados essenciais para o gerenciamento correto de pertinência. Portanto, utilizamos *TCP/IP*. Os servidores de pertinência devem informar aos visualizadores dos atores situados no seu domínio qual porta *multicast* usar.

3.7 Problemas

Nesta seção discutiremos alguns dos problemas relacionados com a subdivisão do mundo virtual (*MV*) em regiões fixas, tal como a migração de um ator de um domínio para outro, a detecção de colisão entre atores e com a região de fronteira, possíveis falhas em servidores, superpopulação dos domínios, etc.

3.7.1 Migração Transparente entre Domínios

Com o particionamento do *MV* em domínios gerenciados por servidores distribuídos, surgiu a necessidade de definirmos protocolos de comunicação entre os visualizadores e os servidores, que tratam da passagem de um ator de um domínio para outro, além da entrada e saída de um ator de um domínio. Os principais requisitos impostos foram que estes protocolos permitissem uma subdivisão em um número arbitrário de domínios e que possibilitassem uma troca eficiente de servidor, e que esta não fosse perceptível pelo usuário.

3.7.2 Detecção de Colisão entre Atores e com Região de Fronteira

A detecção de colisão entre atores foi solucionada através do uso da técnica de *Dead Reckoning* e da definição de uma aura para cada ator. Através do *Dead Reckoning* cada visualizador controlado por um usuário fica responsável por detectar

uma potencial colisão do seu ator com outros atores, isto é, da interceptação das auras deste com a dos atores-fantasmas no mesmo domínio.

Para a detecção de colisão com a região de fronteira também se utilizou a aura nos atores, e definiu-se uma *região de fronteira* em torno de cada fronteira. Com a informação sobre os limites do domínio em que o ator se encontra no momento, da "largura" da região de fronteira, e da própria posição, pode-se detectar uma possível colisão da aura do ator com a região de fronteira.

3.7.3 Falhas em Servidores de Pertinência

Uma vez que cada domínio do *MV* é gerenciado por um servidor de pertinência (*SP*), uma eventual falha de um dos servidores representa um grande problema para um *AVD* utilizando esta técnica de particionamento. No entanto, a falha de um servidor não impede que os visualizadores cujos atores já se encontram no domínio continuem interagindo, através da porta *multicast*. O problema maior está na passagem de atores de um domínio para outro, já que o *SP* com falha será incapaz de indicar qual deverá ser o novo *SP* a ser contactado pelo visualizador de um ator que está saindo de sua região e será incapaz também de fornecer a porta *multicast* usada em seu domínio, para que o visualizador de um novo ator ingressando no domínio, possa interagir com os demais visualizadores dos atores presentes no domínio. Ou seja, a falha de um *SP* deixaria os atores presentes no domínio correspondente isolados dos demais atores no *MV*.

Portanto, só se a falha for temporária (por exemplo, omissão de mensagens) e o *SP* for capaz de recuperar o seu estado, existe a chance do *AVD* poder continuar sem maiores problemas. A maneira clássica de se contornar tal problema causado pela falha de um elemento chave como o *SP* é o de replicá-lo. Uma replicação do *SP*, no entanto, demandaria protocolos de comunicação ainda mais complexos do que os implementados neste trabalho. Assim, decidimos não tratar dos problemas de falha em qualquer um dos servidores (*SPs*, *SDs* e *Super-servidor*, que é descrito na seção seguinte).

3.7.4 Superpopulação de um Domínio

A divisão do *MV* em regiões fixas (domínios) não garante que a todo instante os canais de *IP-multicast* associados a cada um dos domínios estarão sendo utilizados de forma balanceada ou por um número razoável de visualizadores. Pode muito bem acontecer que por acaso, ou devido a algum "evento virtual" ocorra uma migração em massa para um determinado domínio. Denominamos tal situação de "superpopulação de um domínio".

Nesta situação, o ideal seria que se alocasse mais um canal de *multicast* para o domínio e que se agrupasse os atores de acordo com a sua proximidade, deixando um canal para cada uma das regiões dentro do domínio. Ou seja, isto no fundo representa um reparticionamento dinâmico de um *MV* em domínios.

Tal reparticionamento necessariamente seria coordenado por um *Super-servidor* (*SServ*), que seria responsável por gerenciar a informação sobre a vizinhança entre os servidores de pertinência. No entanto, o reparticionamento de um domínio faria necessário também uma interação do *SServ* com os visualizadores para que estes pudessem ser informados do novo canal de *multicast* a ser utilizado. Como no caso do problema de falha de servidores de pertinência, neste caso também os protocolos ganhariam em complexidade, e portanto neste trabalho não trataremos de uma divisão dinâmica do domínio.

Capítulo 4

Protocolos usados no *AVD-PTD*

Neste capítulo, apresentamos o conjunto de protocolos necessários para a implementação do particionamento transparente do mundo virtual em domínios, citados no capítulo anterior. Os elementos do mundo virtual *MV* que participam destes protocolos são:

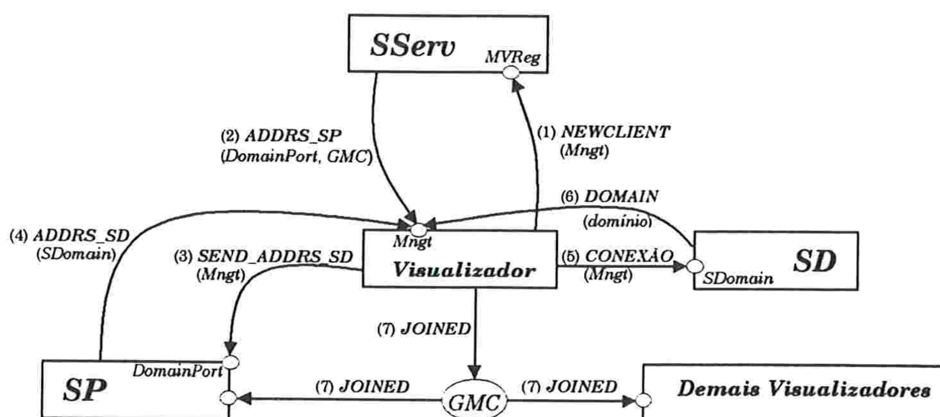
- Ator: representa um ator participante do *MV*.
- Servidor de Domínio (*SD*): mantém uma cópia atualizada do estado da parte do *MV* relativa a um domínio, como por exemplo, aspectos gerais, objetos inativos, cor de fundo, etc.
- Servidor de Pertinência (*SP*): responsável por gerenciar a pertinência de atores em um domínio.
- Super-servidor (*SServ*): responsável por manter e informar sobre a relação de vizinhança entre os *SPs*.

A figura 4.1 mostra a seqüência de interações entre o *SServ*, um servidor de pertinência *SP*, um servidor de domínio *SD*, o *visualizador* de um novo ator (*NovoAtor*), e demais visualizadores dos atores no domínio. Estas interações ocorrem durante e após a inclusão do *NovoAtor* em um domínio no mundo virtual *MV*. Podemos observar ainda que a comunicação entre os visualizadores dos atores e o *SP* ocorre através de uma porta *multicast* (*GMC*). O fluxo de mensagens é representado pelas setas com rótulos indicando a ordem de entrega das mensagens, os tipos e parâmetros

das mensagens (como por exemplo, referências à portas transferidas entre *SServ*, *SP*, *SD* e visualizador do *NovoAtor*).

Em linhas gerais, a seqüência (e o tipo) das mensagens (figura 4.1) é a seguinte:

1. ***NEWCLIENT***: pedido de registro do visualizador do *NovoAtor* (junto ao *SServ*);
2. ***ADDRS_SP***: *SServ* envia referências para endereços (*DomainPort* e *GMC*) do *SP* para o visualizador;
3. ***SEND_ADDRS_SD***: visualizador solicita o envio de referência para a porta *TCP* do *SD*;
4. ***ADDRS_SD***: *SP* envia referência para a porta do *SD*;
5. ***CONEXÃO***: visualizador se conecta ao *SD* para que este possa enviar o estado atual do domínio;
6. ***DOMAIN***: *SD* envia o estado atual do domínio através da conexão *TCP* entre o *SD* e o visualizador do *NovoAtor*;
7. ***JOINED***: visualizador se registra no *GMC* e passa a escutar e atualizar sua posição através do mesmo.



Legenda:

○ – porta de comunicação.		*MVReg* –	porta para registro do *NovoAtor*.
GMC – endereço do grupo *IP-multicast*.		*DomainPort* –	porta *TCP* do *SP* do domínio.
Mngt – porta *TCP* para comunicação com *SServ* e o *SP*.		*SDomain* –	porta por onde é enviado estado atual do domínio.

Figura 4.1 – Linhas de comunicação no *MV*

Nas próximas seções, os protocolos são descritos através de diagramas de eventos no tempo, utilizando as referências à portas apresentadas na figura 4.1.

4.1 Protocolo para Inclusão de um Novo Ator

O protocolo para inclusão é executado somente quando um ator (*NovoAtor*) entra na simulação do *MV* (em um domínio controlado por *SP*). Neste caso, a tarefa do super-servidor (*SServ*) é estabelecer a associação entre o *NovoAtor* e *SP* responsável pelo domínio no qual o *NovoAtor* está ingressando. A partir daí, o visualizador do *NovoAtor* passa a se comunicar diretamente com o *SP* e os visualizadores do *NovoAtor* e dos demais atores presentes no mesmo domínio.

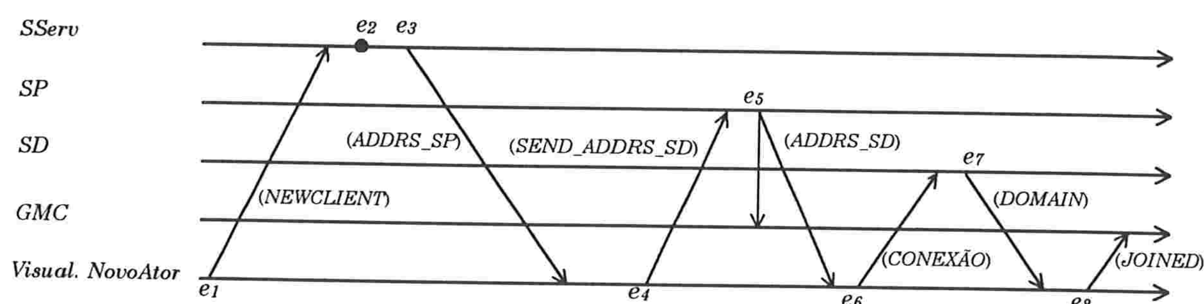


Figura 4.2 – Inclusão de um novo ator

Como se pode verificar na figura 4.2, inicialmente um pedido de registro é enviado (e_1) pelo visualizador do *NovoAtor* para o *SServ*. Este pedido é feito através de uma mensagem do tipo *NEWCLIENT*, cujo conteúdo é uma referência a uma porta (*Mngt*) do visualizador do *NovoAtor* e sua posição inicial. Ao receber esta mensagem o *SServ* determina, a partir da posição inicial do *NovoAtor*, qual é o *SP* responsável pelo domínio no qual o *NovoAtor* deseja entrar (e_2). O *SServ* então envia mensagem (e_3) do tipo *ADDRS_SP* ao visualizador do *NovoAtor*, contendo uma referência para a porta *DomainPort* no *SP* e a porta *IP-multicast* (*GMC*), bem como as coordenadas das fronteiras do domínio em questão.

O visualizador então estabelece uma conexão com a *DomainPort* e envia (e_4) a mensagem do tipo *SEND_ADDRS_SD* para o *SP*, solicitando o endereço do *SD*. Esta mensagem contém uma referência a uma porta (*Mngt*), a qual o *SP* utilizará para comunicar-se com o visualizador do *NovoAtor*.

A mensagem de resposta (e_5) do tipo *ADDRS_SD* enviada ao visualizador do *NovoAtor* pode ter um conteúdo diferente, dependendo das seguintes situações:

- Se *NovoAtor* é o único ator no domínio, a mensagem enviada pelo *SP* tem como conteúdo o endereço (*Mngt*) do visualizador *do NovoAtor*, indicando assim que o

próprio visualizador será o novo servidor de domínio (*SD*). Neste caso, o visualizador recebe o estado do domínio do próprio *SP*. A informação sobre a identificação do *SD* é guardada pelo *SP* (neste caso, os eventos (e_6) e (e_7) são desconsiderados).

- Se já existir um *SD*, o *SP* envia mensagem contendo uma referência para a porta *SDomain* do *SD*. O visualizador do *NovoAtor* estabelece então uma conexão com o *SD* (e_6), fazendo com que o *SD* envie mensagem (e_7) do tipo *SDomain*, contendo uma cópia do estado atual do domínio correspondente.

Após a obtenção do estado atual do domínio, o visualizador do *NovoAtor* então anuncia (e_8) sua entrada no grupo *multicast* através de uma do tipo *JOINED*. A partir de então, o visualizador passa a receber e enviar atualizações de posição pelo grupo *multicast*, usando a técnica de *Dead Reckoning*.

4.2 Protocolo para Exclusão de um Ator

Apresentamos, nesta seção, o protocolo para exclusão de um ator (*Ator*). A figura 4.3 ilustra a exclusão do *Ator* do domínio gerenciado pelo servidor de pertinência *SP*.

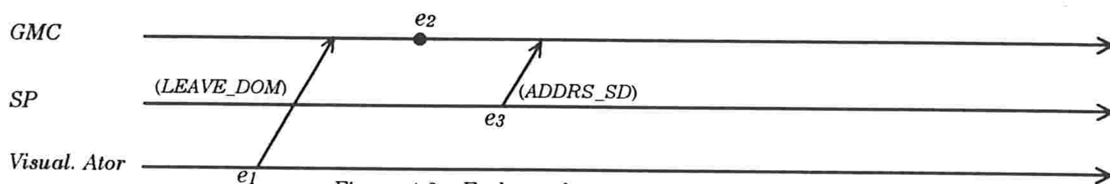


Figura 4.3 – Exclusão de um ator

Na figura 4.3, o visualizador do *Ator* informa (e_1) ao grupo *multicast*, através da mensagem do tipo *LEAVE_DOM*, que o *Ator* está deixando o domínio. No evento (e_2), todos os participantes, inclusive o próprio *SP*, excluem o *Ator* das suas cópias locais do domínio. Caso o visualizador do *Ator* seja o atual servidor de domínio, o *SP* elege outro ator que terá seu visualizador como o novo servidor de domínio. O *SP* então envia (e_3) mensagem do tipo *ADDRS_SD* ao *GMC*, informando a todos qual ator tem seu visualizador como o novo servidor de domínio.

4.3 Protocolos para Migração entre Domínios

Nesta seção, descreveremos em três etapas como se dá a migração de atores entre os domínios do mundo virtual. Na figura 4.4 podemos observar estas etapas. Na

etapa 1 ocorre a entrada de um ator na região de fronteira (*RF*) do domínio atual; na etapa 2 ocorre a passagem do ator de um domínio para outro; e na etapa 3 o ator sai da *RF* do novo domínio.

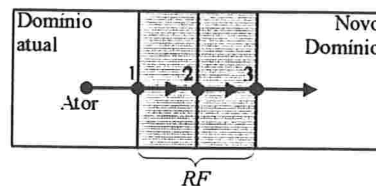


Figura 4.4 – Etapas para migração entre domínios

Nas próximas subseções veremos cada uma das etapas apresentadas na figura 4.4 em maiores detalhes.

4.3.1 Protocolo para Entrada na Região de Fronteira

Nesta seção, descreveremos as interações que ocorrem quando um ator (*Ator*) se aproxima da região de fronteira (*RF*) entre seu domínio *Atual* e um domínio *Novo*. Participam deste protocolo o servidor de pertinência do atual domínio (*SP_Atual*) e do novo domínio (*SP_Novo*), e o servidor de domínio (*SD_Viz*) do novo domínio. A interação entre estes elementos é mostrada na figura 4.5.

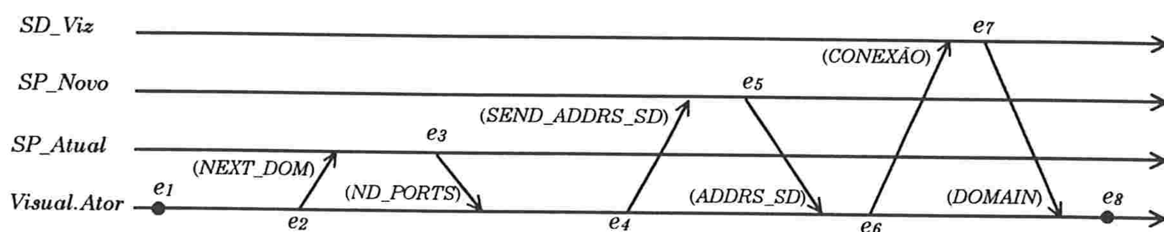


Figura 4.5 – Entrada na Região de Fronteira

Ao detectar (e_1), através de sua aura, que seu *Ator* está na *RF* entre os domínios controlados pelo *SP_Atual* e pelo *SP_Novo*, o visualizador do *Ator* envia (e_2) a mensagem do tipo *NEXT_DOM* para o *SP_Atual*, informando que poderá mudar de domínio. Esta mensagem contém uma referência para a porta (*Mngt*) do visualizador do *Ator*. O *SP_Atual* envia (e_3) a mensagem do tipo *ND_PORTS* para o visualizador do *Ator*, contendo referências para as portas *DomainPort* (do *SP_Novo*) e a porta *multicast GMC*, relativos ao novo domínio. A seguir, o visualizador do *Ator* envia (e_4) mensagem do tipo *SEND_ADDRS_SD* para o *SP_Novo*, para obter o endereço do servidor de domínio (*SD_Viz*) do novo domínio. O *SP_Novo* por sua vez envia (e_5)

mensagem do tipo *ADDRS_SD* com uma referência para a porta *SDomain* do *SD_Viz*, através da qual o visualizador do *Ator* receberá o estado atual do novo domínio.

De forma similar ao que ocorre no protocolo de inclusão, caso o novo domínio esteja vazio (ou seja, sem nenhum ator), o *SP_Novo* assume o papel do servidor de domínio. No evento (e_6), o visualizador do *Ator* estabelece conexão *TCP* com a porta *SDomain* do *SD_Viz* e este envia (e_7) mensagem do tipo *DOMAIN*, contendo o estado atualizado do domínio. O visualizador do *Ator* insere (e_8) as informações recebidas do *SD_Viz* (cópia do estado do domínio vizinho), em uma lista de domínios vizinhos.

É necessário que o visualizador do *Ator* mantenha esta informação sobre domínios vizinhos a uma fronteira, pois o *Ator* pode estar na região de fronteira com vários domínios, e nesta situação é impossível prever para qual domínio vizinho o *Ator* de fato migrará. Além disso, a manutenção simultânea das visões de domínios vizinhos é necessária para realizar a migração de forma que o usuário não tenha a sensação de mudança abrupta. As cópias de domínios vizinhos são mantidas atualizadas através da escuta por mensagens *multicast* nas respetivas portas *GMC* (*GMC_Atual*, *GMC_Novo*).

4.3.2 Protocolo para Mudança de Domínios

A segunda etapa na migração para outro domínio ocorre quando o *Ator* transpõe o limite entre os domínios gerenciados pelo *SP_Atual* e *SP_Novo*. Nesta transposição, o *Ator* usa também o grupo *multicast* atual (*GMC_Atual*) e o grupo *multicast* do novo domínio (*GMC_Novo*).

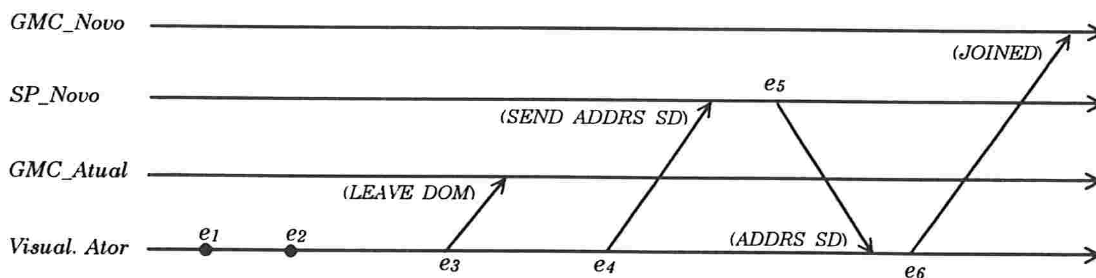


Figura 4.6 – Mudança de domínios

O protocolo (mostrado na figura 4.6) é iniciado quando o visualizador do *Ator* detecta (e_1), através da aura de seu *Ator*, que o mesmo está ultrapassando a fronteira entre os domínios. Então, o visualizador do *Ator* verifica (e_2) para qual novo domínio o

mesmo está migrando, e atualiza a informação sobre domínio principal e domínios vizinhos, tais como os respectivos endereços *multicast*, endereços dos servidores de domínios, etc. Neste momento o *Ator* já se considera integrante do novo domínio, podendo descartar a lista antiga de domínios vizinhos.

O visualizador envia (e_3) mensagem do tipo *LEAVE_DOM* para o *GMC_Atual*, e o protocolo de exclusão, descrito na seção 4.2, é executado.

Os eventos (e_4), (e_5) e (e_6) são respectivamente similares aos eventos (e_4), (e_5) e (e_8) da seção 4.1. A diferença é que neste caso o visualizador do *Ator* envia (e_5) a mensagem do tipo *SEND_ADDRS_SD* ao *SP_Novo*, com a intenção de ser declarado o servidor de domínio do novo domínio, enquanto que no evento (e_5) da seção 4.1, o objetivo era é o de obter o endereço do servidor de domínio. Os eventos (e_6) e (e_7) da seção 4.1, onde o *SD* envia uma cópia com o estado atual do domínio para visualizador do *Ator*, não são necessários pois o *Ator* já dispõe desta cópia na lista de domínios vizinhos, descrita na seção anterior.

Ao passar para outro domínio, automaticamente o protocolo de entrada na região de fronteira, descrito na seção 4.3.1, é executado, pois o *Ator* passa a estar na região de fronteira do outro domínio.

4.3.3 Protocolo para Saída da Região de Fronteira

Ao sair da região de fronteira (de qualquer domínio), percorre-se a lista de domínios vizinhos (descrita na seção 4.3) e descarta-se o elemento correspondente à fronteira que o ator correspondente está deixando para trás.

4.3.4 Identificação do Domínio Destino

Os protocolos de migração descritos na seção 4.3 também prevêem o caso em que um ator está se movimentando para uma fronteira do seu atual domínio com mais de um domínio vizinho. Esta situação é ilustrada na figura 4.7, onde um ator está na região de fronteira entre os domínios D_3 e D_2, mas onde D_3 também faz fronteira com D_1.

Neste caso, usa-se a posição de entrada na região de fronteira e o vetor velocidade para determinar se o ator deve adquirir a visão dos domínios D_1 e D_2, em vez de apenas a de D_2.

Se só considerássemos a posição do ator na *RF*, provavelmente indicaríamos que o ator estaria migrando para D_2, quando na verdade a sua trajetória indica uma migração para D_1. Uma vez recebidos os dados de D_1 e D_2, é possível ao visualizador correspondente ao ator escolher rapidamente qual será o novo domínio efetivo, assim que o ator ultrapassa efetivamente a fronteira.

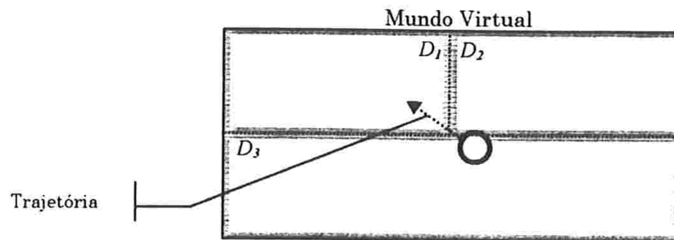


Figura 4.7 – Migração entre domínios

4.4 Discussão

Utilizando estes protocolos para migração entre domínios, de fato consegue-se obter uma transição entre domínios pouco perceptível ao usuário. Isto foi comprovado através de testes práticos com um protótipo[6], e também foi comprovado através da comparação dos tempos de deslocamento de atores intra e inter-domínios. Maiores detalhes sobre os testes realizados estão descritos no capítulo 6.

---

[6] Este protótipo é descrito no capítulo 5.

Capítulo 5

O Protótipo do *AVD-PTD*

De forma a demonstrar a factibilidade dos protocolos descritos no capítulo 4, fizemos uma implementação distribuída de um jogo simples de movimentação em um ambiente 2D. Este ambiente foi particionado em domínios, de forma transparente ao usuário.

Este protótipo é resultado da extensão de um jogo demonstrativo para Java chamado Javaroids [8]. A seguir, apresentaremos uma visão geral da estrutura do programa, com ênfase nos elementos que utilizamos para o protótipo.

5.1 Javaroids

Em sua versão original, Javaroids é um programa 2D mono-usuário. O usuário é representado por uma nave, que tem a capacidade de alterar a direção de movimentação, acelerar e disparar projéteis contra alvos asteróides e naves inimigas. O jogo se inicia com uma tela introdutória, que lista quantos pontos vale cada objeto no jogo. Depois disso, o jogador pode fazer um teste de movimentação da sua nave, e aprender os controles. A seguir, o jogador pode começar o jogo. A transição entre estas telas é realizada por um clique do mouse. Na figura 5.1 podemos observar uma tela do jogo Javaroids.

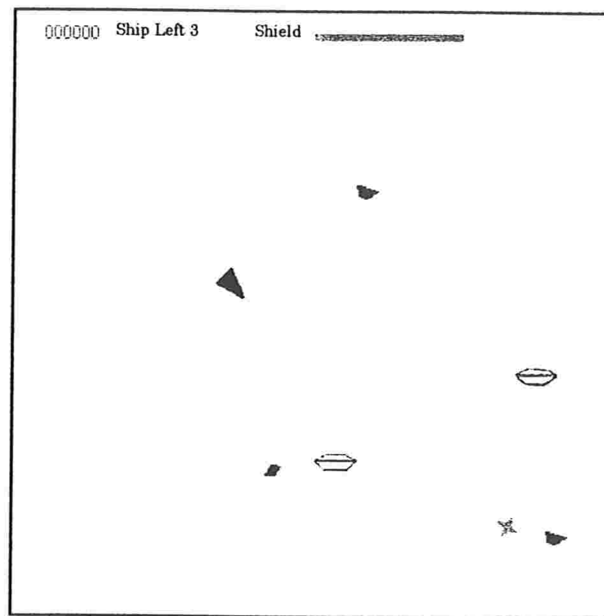


Figura 5.1 – Imagem do jogo Javaroids original

A figura 5.2 resume as funções das principais classes do Javaroids.

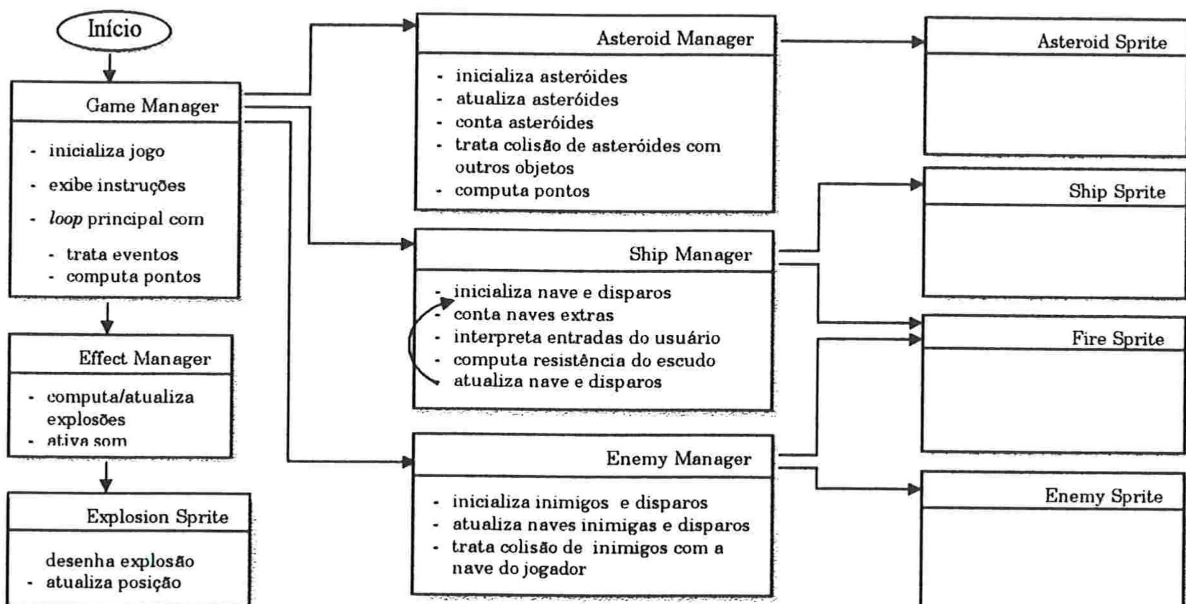


Figura 5.2 – Divisão de Responsabilidades no Javaroids

As setas na figura 5.2 ilustram o fluxo de controle entre os objetos do Javaroids. Observamos que Javaroids é formado por um conjunto de gerenciadores (*managers*), cada um controlando elementos de determinado tipo. Os tipos de objetos implementados no Javaroids são descritos pelas classes: *Asteroid sprite, Ship sprite, Fire sprite, Enemy sprite* e *Explosion sprite.*

Os gerenciadores *Asteroid Manager*, *Ship Manager* e *Enemy Manager* possuem instâncias do gerenciador de efeitos *Effect Manager*. O *Effect Manager* é responsável apenas por realizar os efeitos visuais e sonoros, causados pelos demais gerenciadores. O *Ship Manager* tem uma seta adicional, indicando que as operações descritas são repetidas cada vez que um objeto do tipo *Asteroid sprite* ou *Enemy sprite* consegue destruir o objeto do tipo *Ship sprite*, controlado pelo *Ship Manager*.

Os *sprites Asteroid*, *Ship*, *Fire* e *Enemy* têm funções comuns. Eles são responsáveis por desenhar o objeto (por exemplo, uma nave ou asteróide), computar e atualizar a posição do objeto e, com exceção do *Ship sprite*, verificar se há colisões entre os *sprites*.

Vejamos agora como é a interação dos gerenciadores no *loop* principal do programa (figura 5.3).

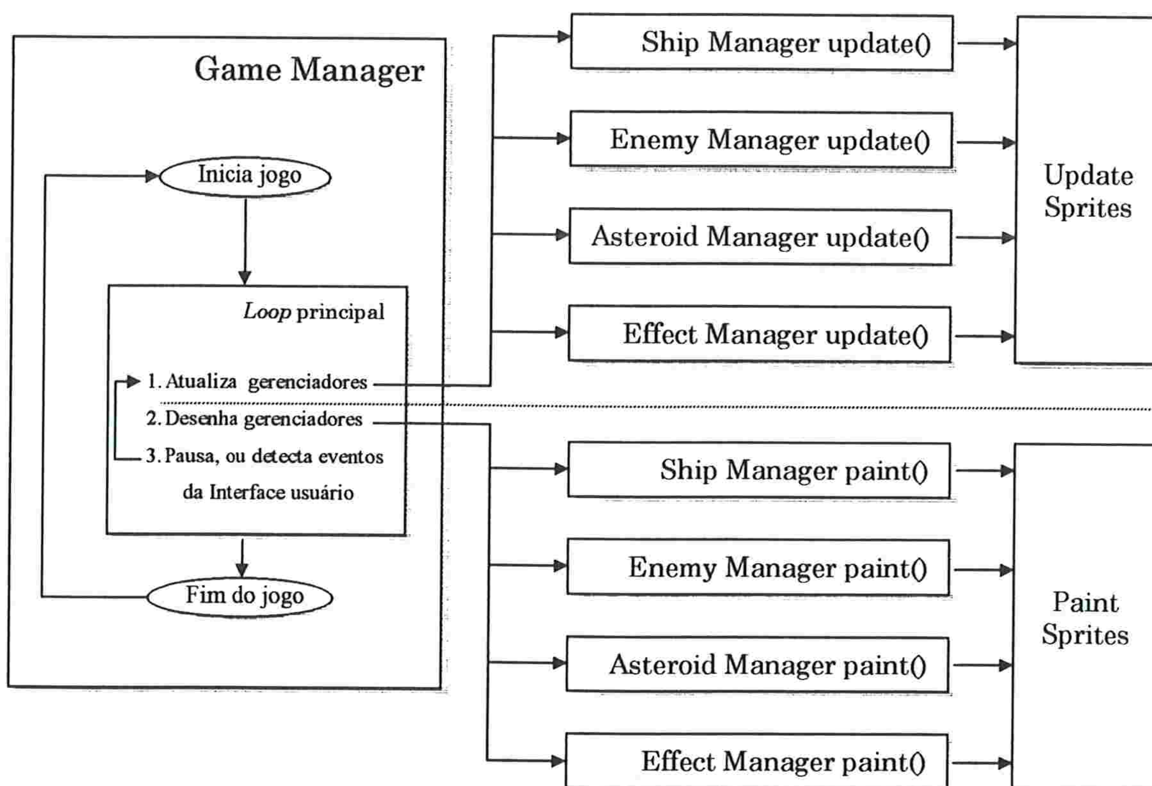


Figura 5.3 – Interação dos Gerenciadores durante *Loop* Principal

Como visto na figura 5.3, a classe *Game Manager* age como o programa principal que sincroniza as operações *update()* e *paint()* de outros quatro gerenciadores. Cada gerenciador por sua vez, invoca os respectivos métodos *update()* e *paint()* em cada um

dos *sprites* por ele controlado. Ou seja, caso existam *N* asteróides em movimento, o método *update()* do *Asteroid Manager* invoca o *update()* em cada um dos *N* objetos *sprite*. As principais funcionalidades do jogo são implementadas nos métodos *update()* dos gerenciadores. Por exemplo, o *update()* do *Asteroid Manager* invoca o método *update()* de todos os *Asteroid sprites*, onde é realizada a movimentação dos asteróides, verifica se houve colisão com outros *sprites* do jogo, e explode os asteróides quando ocorre uma colisão. Desta explosão são criados novos *sprites* que representam pedaços menores do asteróide que foi destruído. Outro exemplo é o método *update()* do *Ship Manager*, que informa ao *Ship sprite* como este deve recalcular a sua nova posição, baseado nas teclas que o usuário está pressionando.

O método *paint()* do *Ship Manager* invoca o método *paint()* dos objetos *Fire sprite* e *Ship sprite*, que realmente desenham os objetos. Além disso, ele mantém e apresenta o *status* relativo ao número de naves restantes, e a durabilidade do escudo protetor que o *Ship sprite* possui para sua defesa. Se for detectado um evento do tipo "pressionamento da tecla *shield*", então *paint()* também desenha um escudo em volta do *Ship sprite*.

A seguir descreveremos em mais detalhes as classes *Game Manager* e *Ship Manager*, nas quais fizemos as modificações mais significativas para o desenvolvimento do nosso protótipo.

5.1.1 Classe *Game Manager*

A classe *Game Manager* contém o laço principal (*main*) do Javaroids. *Game Manager* implementa a interface *Runnable*, definindo um método chamado *run()*. O coração deste método é um *loop*, que coordena a seqüência de ações descritas na figura 5.2. *Game Manager* também define tratadores para eventos de teclas, que são passados adiante para o *Ship Manager*.

5.1.2 Classe *Ship Manager*

A classe *Ship Manager* é encarregada de inicializar a nave do jogador e o *Fire sprite*, que desenha os mísseis lançados pelo jogador. Esta inicialização ocorre no construtor da classe *Ship Manager*.

Outra função do *Ship Manager* é traduzir eventos gerados pelo usuário para mensagens enviadas ao *Ship sprite*. Para processar entradas do usuário, mantém-se um *buffer* que verifica se uma tecla foi pressionada ou não. Isto permite ao jogo reagir a múltiplas entradas como, por exemplo, disparar e avançar simultaneamente. Se não existisse tal *buffer*, o jogo só poderia responder a uma única entrada do usuário por vez.

5.2 O Protótipo

Com o objetivo de demonstrar a viabilidade e a eficácia dos protocolos descritos no capítulo 4, o jogo Javaroids foi adaptado para ser um programa distribuído a ser usado por vários usuários.

Na figura 5.4 temos uma imagem do protótipo onde o mundo virtual foi dividido em 3 domínios, representados pelos três retângulos maiores. As linhas em preto representam a região de fronteira entre os domínios. No centro da figura podemos observar um ator com sua aura navegando pela região de fronteira dos domínios. Na região de fronteira entre os domínios superiores e o domínio inferior do mundo virtual, observamos três retângulos cinzas menores, que representam a projeção dos atores que estão presentes no domínio inferior. Estas projeções foram incorporadas no protótipo a fim de mostrar que mesmo estando na RF de seu atual domínio, um ator já acompanha a movimentação dos atores no(s) novo(s) domínio(s) para os quais está migrando.

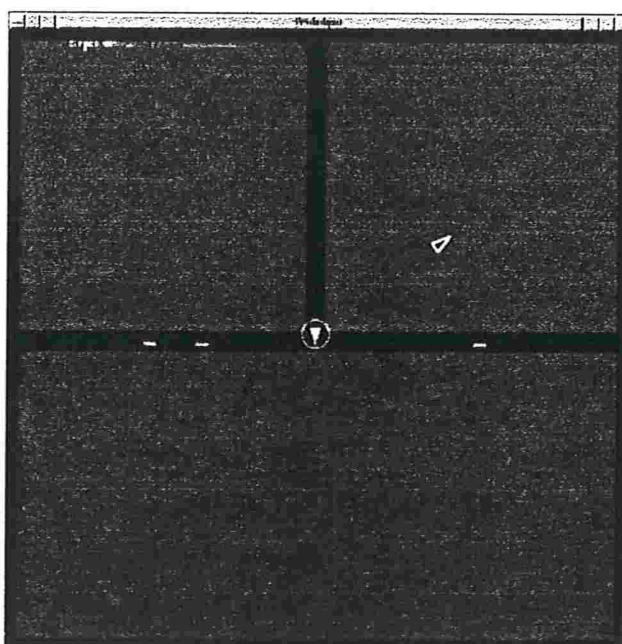


Figura 5.4 – Imagem do *MV* no visualizador do protótipo

O protótipo consiste de processos independentes de três diferentes tipos:

- *Super-servidor*: é responsável pela associação de domínios a servidores de pertinência.
- *Servidor de Pertinência*: é responsável pelo gerenciamento da porta *multicast* de cada domínio.
- *Visualizador*: é o processo que implementa a interface do usuário com o jogo.

Os servidores de pertinência foram modelados como sendo processos executando em diferentes *hosts* de uma rede local. Os visualizadores são processos que vão se conectando e desconectando dos respectivos servidores de pertinência à medida em que os atores (controlados pelo usuário) transpõem fronteiras entre domínios. O controle de movimentação definido para cada ator pode ser de dois tipos: trajetórias parametrizadas ou controladas pelo usuário.

A simulação tem como base um mundo virtual 2D, com particionamento estático de domínios. A simulação inclui a detecção de colisão entre atores e a detecção de entrada na região de fronteira (*RF*). A plataforma de implementação escolhida foi Java 1.2. A escolha foi motivada pelo fato de Java ser multiplataforma, oferecer coleta de lixo, ser a linguagem do jogo Javaroids, além de dispor de muitas classes (*java.net.**) para programação em um ambiente de rede.

A seguir, apresentamos as principais funções dos elementos que compõem o protótipo.

5.2.1 Configuração Inicial

Para a configuração inicial dos servidores de pertinência (*SPs*) e do super-servidor (*SServ*), criamos um arquivo chamado *Setup.dat*. Este arquivo contém informações sobre endereços *IP* e portas (*TCP* e *multicast*) dos servidores de pertinência, e do super-servidor.

A figura 5.5 apresenta um exemplo do arquivo de configuração inicial. A primeira linha do arquivo contém informações sobre os endereços *IP* e *multicast* do *SServ*. Cada uma das demais linhas está associada a um *SP*.

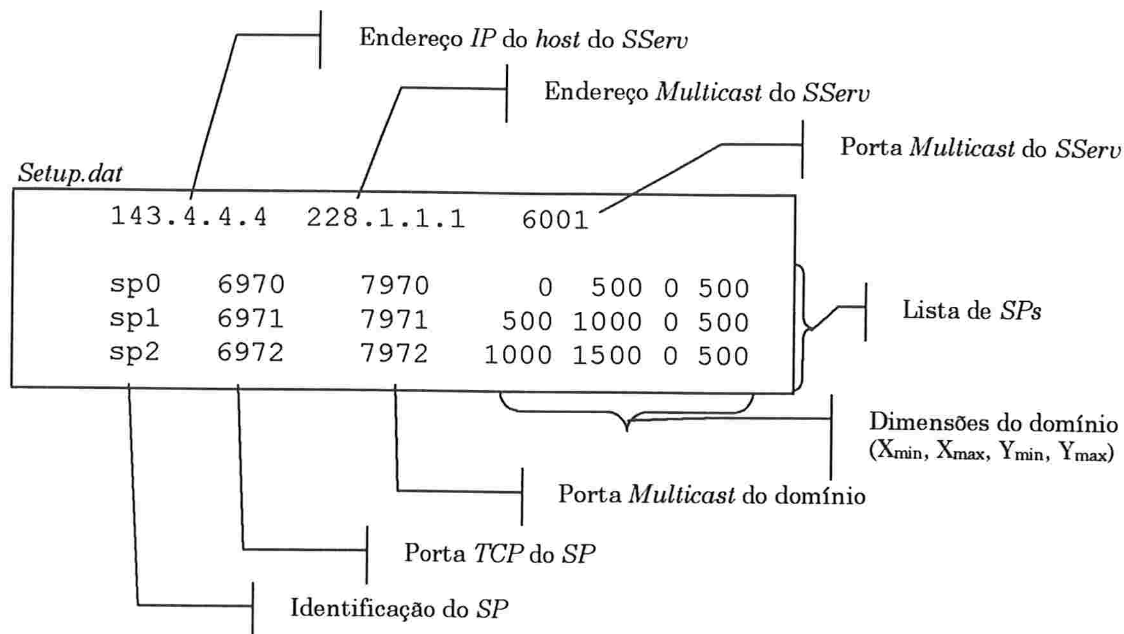


Figura 5.5 – Estrutura do arquivo de configuração inicial *Setup.dat*

5.2.2 Fase de Preparação do Ambiente

Antes de poder iniciar o jogo do Protótipo, é necessário instanciar o super-servidor e os servidores de pertinência. O primeiro passo é a criação do super-servidor *SServ* (que é um processo UNIX) que deverá executar no *host* indicado em *Setup.dat*. Para isto deve-se digitar: `java jogo`. Ao iniciar, o *SServ*: a) lê o arquivo de configuração inicial *Setup.dat*; b) atualiza este arquivo, inserindo nele o seu atual endereço *IP*; e c) cria a lista de servidores de pertinência utilizada para consultas toda vez que um novo ator entra para o mundo virtual.

Em seguida, deve-se instanciar os servidores de pertinência (*SPs*). Cada *SP* pode ser instanciado em um computador (máquina) diferente. Na linha de comando, deve-se digitar `java sp spID`, onde *spID* corresponde à identificação do *SP*. Esta identificação será usada pelo *SP* para que ele possa localizar a respectiva linha que contém seus dados iniciais no arquivo de configuração *Setup.dat*.

5.2.3 Fase de Jogo

Quando o ambiente já está preparado, o próximo passo é a entrada dos participantes (usuários) no jogo. De qualquer máquina, um novo usuário pode entrar

na simulação, digitando na linha de comando: `java GameManager`[7] `px py`, onde *px* e *py* correspondem às coordenadas da posição inicial do ator dentro do mundo virtual (*MV*). Caso esta posição não seja fornecida ou esteja além das dimensões do *MV*, uma posição inicial aleatória lhe será atribuída. Começa então a simulação para este ator, dando início à seqüência de interações, descritas no capítulo 4, entre seu visualizador, super-servidor, servidores de pertinência, servidores de domínios, e visualizadores dos demais atores.

5.3 Implementação do Protótipo

Para a implementação do protótipo, foram necessárias modificações na estrutura das classes do jogo *Javaroids*. Além disso, várias classes do jogo original foram desativadas, como por exemplo as classes *Enemy* e *Explosion*. Mecanismos para a comunicação em rede foram implementados para atender à necessidade do protótipo ser um jogo distribuído. Nas seções seguintes descrevemos tais modificações e extensões.

5.3.1 Interação Entre Elementos

A classe *Ship Manager* do jogo *Javaroids* foi renomeada, passando a se chamar *Control Ship Manager*. Uma nova classe *Ship Manager* foi criada com o propósito de implementar a interação entre os visualizadores dos atores, servidores de pertinência (*SPs*), servidores de domínios (*SDs*) e super-servidor (*SServ*).

As alterações foram realizadas visando tornar o jogo *Javaroids* distribuído, utilizando comunicação (baseada em *TCP* e *multicast*) via *sockets*. Estas alterações incluem a criação de novas classes que implementam a comunicação no protótipo, como por exemplo as classes *ReceiveThread*, *MulticastGroup*, *SendDomainServer*, *Sender*, descritas nas seções seguintes.

Alguns métodos foram introduzidos e/ou modificados em outras classes para permitir a interação entre os visualizadores dos atores e *SPs* pertencentes a um mesmo grupo *multicast*. São exemplos deste caso: a classe *Ship*, que teve alterações no método *updatePosition*() e a adição de um novo método, *checkBoundary*(). O método *updatePosition*() original tinha como funções atualizar a posição do centro do polígono

---

[7] Esta classe implementa o visualizador do ator associado ao usuário.

(que representa a nave[8] do usuário), e deslocar o centro para o outro lado da tela, caso o polígono ultrapasse os limites da tela, criando assim o efeito de um espaço circular infinito comum em jogos deste tipo. Agora, além de invocar o método *checkBoundary()*, este método também invoca o método *leaveBoundary()* da classe *Ship Manager*.

O método *checkBoundary()* verifica se o polígono está sobre a região de fronteira *RF* do domínio. E o método *leaveBoundary()* do *Ship Manager* é invocado quando o polígono sai da *RF*. Neste caso, ou ocorre a transferência do ator para um novo domínio, ou as informações sobre a respectiva fronteira são descartadas, por exemplo quando o ator (nave) sai da *RF*, mas permanece no mesmo domínio.

A classe *Control Ship Manager*, sofreu modificações nos métodos *update()* e *paint()*. O método *paint()*, descrito na seção 5.1, agora invoca o método *desenha()* da classe *Ship Manager*, que desenha os demais atores no domínio. O método *update()* invoca o método *ghostUpdate()* também da classe *Ship Manager*. Neste método utilizamos a técnica de *Dead Reckoning* para atualizar a posição dos objetos-fantasmas, que representam os demais atores no domínio. Além disso, foi criado o método *deadReckoning()* que é invocado pelo método *update()*. O método *deadReckoning()* verifica se houve mudança significativa de posição entre o *Ship* e o fantasma a ele associado. Caso isto ocorra, uma mensagem de atualização é enviada para grupo *multicast*, para que todos os visualizadores dos demais atores no domínio possam atualizar os respectivos objetos-fantasmas.

5.3.2 Componentes de Software do Protótipo

As classes do protótipo foram agrupadas, formando um conjunto de componentes. A seguir descreveremos estas componentes e suas principais funções. Para esta descrição utilizaremos os termos *SServ* como referência para o super-servidor, *SPs* para os servidores de pertinência, e *SDs* para os servidores de domínios.

- *GameManager*: implementa o visualizador, responsável por iniciar o programa do usuário.
- *SuperServerManager*: *SServ* que coordena *SPs* e a entrada de novos atores na simulação.

[8] Neste jogo, um ator é uma nave controlada pelo usuário ou um robô.

- *SPManager*: gerencia domínio, coordenando a entrada/saída de cada ator do mesmo.
- *ShipManager*: gerencia a interação do visualizador de um ator com visualizadores de outros atores, *SPs*, *SDs* e *SServ*.
- *ControlShipManager*: implementa rotinas de animação da nave controlada pelo usuário.
- *Domínio*: responsável por manter as informações sobre os atores presentes em um mesmo domínio do *MV*.
- *Comunicação*: agregado de classes que implementam os vários tipos de comunicação (*TCP*, *multicast*) citados no capítulo 1.
- *SendDomainServer*: *thread* servidor responsável por enviar cópia do estado atual do domínio para visualizadores de novos atores. Este *thread* só é instanciado em visualizadores que também são servidores de domínio, e nos servidores de pertinência.
- *SendDataServer*: *thread* servidor responsável classe responsável pelo envio, via conexão *TCP*, de dados, tais como endereço do servidor de domínio e do servidor de pertinência, para o visualizador de um novo ator entrando no domínio.

Nas próxima subseções, veremos estas componentes em maiores detalhes. Os diagramas UML [4] correspondentes a estas classes se encontram no apêndice A.

5.3.2.1 *GameManager*

A componente *GameManager* é formada pelas componentes *ShipManager*, *ControlShipManager* e pela classe *PrototypeFrame*. Estas componentes em conjunto formam o visualizador, descrito na seção 1.2 do capítulo 1.

A classe *PrototypeFrame* foi criada para tornar o visualizador uma aplicação Java em vez de uma *applet*, como era o jogo Javaroids original, e para incorporar também tarefas de comunicação.

5.3.2.2 *SPManager*

A componente *SPManager* que implementa o servidor de pertinência de um domínio é formada pelas componentes *Comunicação* e *Domínio*, e pelas classes *SP* e *Vizinho*.

Com exceção da classe *SenderDSS*, que trata do envio de datagramas via *DatagramSocket*, todas as outras classes da componente *Comunicação* são implementadas pela componente *SPManager*. A classe *SP* executa o programa responsável pela instanciação dos *SPs*. E a classe *Vizinho* é usada nos *SPs* para a criação da lista de *SPs* vizinhos.

5.3.2.3 *SuperServerManager*

A componente *SuperServerManager* é responsável por informar ao visualizador de um novo ator qual *SP* contactar para que o mesmo possa dar início à interação com outros visualizadores dos demais atores no domínio. Esta componente é formada pela componente *Comunicação* e pelas classes *Jogo* e *SPProxy*.

Com exceção das classes *DataShipClass*, *SendDataServer* e *Sender*, todas as outras classes da componente *Comunicação* são implementadas pela componente *SuperServerManager*. A classe *Jogo* executa o programa responsável pela instanciação do *SServ*. A classe *SPProxy* é um proxy criado pelo *SServ* e é utilizado para consultas sobre qual será o *SP* responsável por um novo ator que entra para a simulação.

5.3.2.4 *ShipManager*

A componente *ShipManager* implementa a comunicação entre visualizadores, e entre *SPs* e visualizadores. Esta componente é formada pelas componentes *Comunicação*, *Domínio* e pela classe *NeighborView*.

Com exceção das classes *SendDataServer* e *SenderDSS*, todas as outras classes da componente *Comunicação* são implementadas pela componente *ShipManager*. A classe *NeighborView* mantém atualizadas as informações sobre domínios vizinhos, enquanto um ator se encontra na região de fronteira. Ou seja, esta classe é responsável por manter a lista de domínios vizinhos descrita na seção 4.3.1.

5.3.2.5 *ControlShipManager*

A componente *ControlShipManager* é formada pelas componentes *Fire* e *Ship*.

A classe *Fire* realiza a animação de um disparo de uma nave. O conceito de aura é utilizado na classe *Fire* para a detecção de colisão do objeto *Fire* com outros objetos. E a classe *Ship* representa graficamente uma nave (ator) no jogo.

5.3.2.6 *Domínio*

A componente *Domínio* é responsável por armazenar a lista de atores e objetos presentes em um domínio, além de possíveis atributos específicos de cada domínio. Esta componente é formada pelas classes *Ator*, *Dimension* e *GhostShip*.

A classe *Ator* mantém informações sobre a identificação do ator enquanto que a classe *GhostShip* armazena informações sobre os objetos-fantasmas presentes no domínio.

A classe *Dimension* mantém os valores que determinam a dimensão de um domínio. Esta classe foi criada para facilitar futuras extensões do protótipo, como por exemplo para 3D.

5.3.2.7 *Comunicação*

Esta componente implementa a comunicação no protótipo. Ela é formada pelas componentes *SendDataServer*, *SendDomainServer*; pelas classes *DataShipClass*, *DataSS*, *MulticastGroup*, *ReceiveThread*, *Sender*, *SenderDSS*; e pela interface *Protocol*.

A classe *DataSS* armazena informações (tais como portas *TCP*, *multicast*, etc.) que são trocadas entre *SServ*, *SPs* e os visualizadores de atores. A classe *DataShipClass* armazena dados sobre novos atores, como por exemplo a identificação do novo ator e a posição e velocidade iniciais da nave (ator), que são enviados para grupo *multicast* do domínio. A classe *MulticastGroup* cria um grupo *multicast*. Já a classe *ReceiveThread* recebe informações enviadas ao grupo *multicast*. A classe *Sender* é responsável por enviar pacotes, utilizando o socket *multicast*, para outros membros de um grupo *multicast*. E a classe *SenderDSS* é responsável por enviar datagramas do *SServ* para o visualizador de novos atores.

A interface *Protocol* armazena constantes do tipo *JOINED*, *ADDRS_SD*, que são utilizadas nos protocolos de comunicação.

5.3.2.8 *SendDomainServer*

A componente *SendDomainServer* é formada pela classe *SendDomain*. Um objeto do tipo *SendDomain* é instanciado toda vez que uma conexão *TCP* é estabelecida entre

um visualizador de um novo ator e um servidor de domínio. Esta classe envia o estado atual do domínio, solicitado pelo visualizador (do novo ator) que criou a conexão.

5.3.2.9 *SendDataServer*

A componente *SendDataServer* é formada pela classe *SendData*. Um objeto do tipo *SendData* é instanciado toda vez que uma conexão *TCP* é estabelecida entre um *SP* e um ator. Esta classe envia informações sobre um novo domínio, solicitadas pelo visualizador que criou a conexão.

A lista completa de classes e métodos do protótipo pode ser vista em maiores detalhes no apêndice B.

Capítulo 6

Testes

Foram feitos diversos testes com o protótipo para avaliar o seu funcionamento e desempenho em diferentes situações de carga, isto é, para um número variável de atores.

O principal objetivo dos testes foi o de obter valores quantitativos que caracterizassem e confirmassem de forma precisa a impressão visual que um usuário do protótipo obtém sobre o tempo de resposta durante uma migração entre domínios. Em particular, comparou-se o período de tempo necessário para que um ator com velocidade constante leva para percorrer uma mesma distância quando se movimenta dentro de um domínio e quando ultrapassa a fronteira do mundo virtual.

Um outro objetivo foi o de identificar qual ação executada por um visualizador durante a passagem de um ator pela fronteira é a principal responsável pela degradação do desempenho do protótipo. E um terceiro objetivo foi o de medir qual é o impacto que o compartilhamento dos servidores (tais como servidores de pertinência e servidores de domínios) pelos visualizadores tem sobre a atualização da visão do mundo virtual pelo usuário.

Todos os testes foram executados em um conjunto de 23 máquinas executando o sistema operacional Solaris (SunOS 5.7), onde os processos (tais como super-servidor, visualizadores, etc.) foram distribuídos da seguinte forma:

- *Super-servidor*: *Sparc Station 4*, com 32MB de memória e 1 processador.

- *Servidor de Pertinência SP0*: Servidor *UltraSPARC Enterprise 3000*, com 2GB de memória e 6 processadores.
- *Servidor de Pertinência SP1*: *Sparc Station 4*, com 32MB de memória e 1 processador.
- *Servidor de Pertinência SP2*: *Sparc Station 2*, com 64MB de memória 1 processador.
- *Visualizador*[9]: *Sparc Station 20*, com 64MB de memória e 1 processador.
- *Robôs*: 1. Servidor *UltraSPARC-II Enterprise 250*, com 512MB de memória e 2 processadores (15 robôs),
  2. *Sparc Station 4*, com 32MB de memória e 1 processador (4 robôs).

O resultado de cada teste foi obtido tomando-se a média de 20 execuções para cada conjunto de robôs[10], representando os demais atores presentes no mundo virtual.

Para efeito de medições, nos dois primeiros testes todos os robôs estavam parados. Isto foi feito para evitar a influência mútua entre atores, pois o objetivo destes testes foi o de analisar isoladamente o desempenho do protocolo de comunicação para migração entre domínios, descritos na seção 4.3. Com todos os robôs parados, o ator pôde tratar exclusivamente das tarefas de mudança de domínios, sem ter que tratar também das mensagens *multicast*, contendo atualizações provenientes de outros atores.

Mediu-se portanto os tempos que um único ator precisa para percorrer a distância σ (figura 6.1) dentro de um domínio e durante a passagem pela fronteira, movendo-se em três diferentes velocidades:

- v_1: representa um ator navegando pelo mundo virtual de forma muito lenta (1 unidade de distância por segundo).
- v_2: velocidade que consideramos comum em jogos deste tipo (3 unidades/seg.).
- v_3: alta velocidade. Através dela, quisemos analisar o comportamento do protótipo em situações extremas (8 unidades/seg.).

---

9 Este visualizador foi utilizado para medição de tempos dos testes realizados.

10 Nestes testes, os robôs são simplesmente atores com um movimento pré-definido.

O primeiro teste foi executado para um ator com as três velocidades. Nos outros dois testes, achamos suficiente efetuar os testes somente com a velocidade v_2, dado que esta reflete uma situação mais comum de uso do jogo. No terceiro teste, programamos os robôs para que todos eles se movimentassem horizontalmente e verticalmente pelo mundo virtual com a velocidade v_2.

As seções seguintes descrevem os testes em maiores detalhes.

6.1 Teste n.º 1: Efeito visível da migração para o usuário

Este teste foi realizado em duas etapas. Na primeira etapa, medimos a quantidade de tempo que um ator necessita para percorrer uma distância igual a duas vezes a largura da região de fronteira mais o diâmetro de sua aura, representado na figura 6.1. Esta é a distância que um ator, passando perpendicularmente pela fronteira, percorre durante o processo de migração entre domínios.

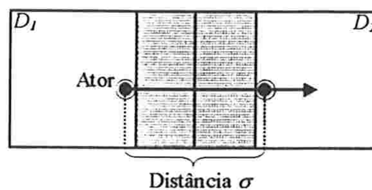


Figura 6.1 – Distância percorrida para migração entre domínios

Os dados desta primeira etapa (tempos necessários para um ator percorrer a distância σ fora da região de fronteira) podem ser visto no gráfico da figura 6.2.

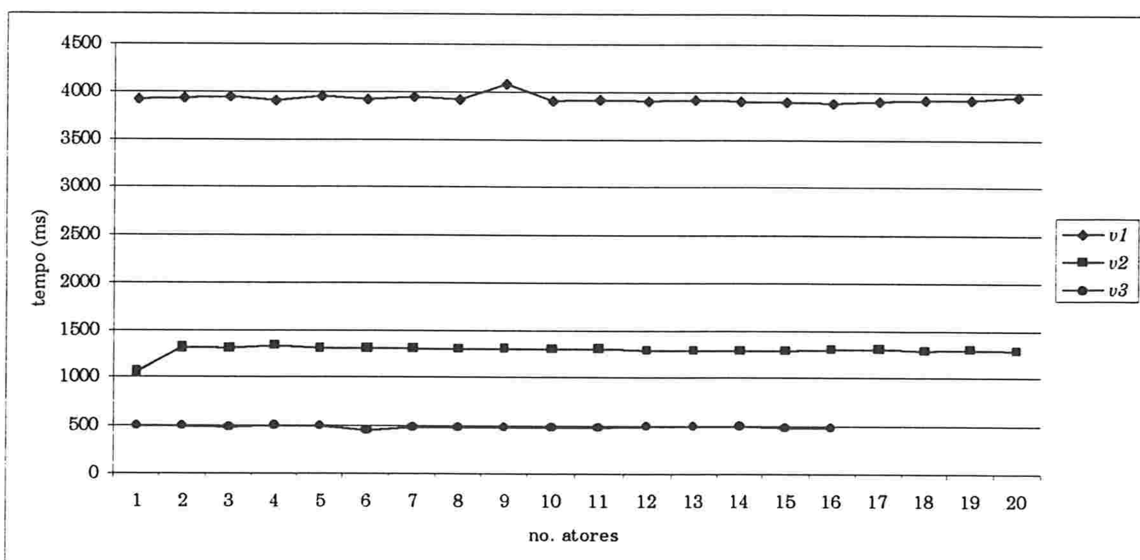


Figura 6.2 – Tempo para percorrer σ dentro do domínio

Para a segunda etapa, tomamos o tempo gasto por um ator que migra de um domínio para o outro, desde o instante em que ele entra na região de fronteira do seu domínio atual até que ele deixa a região de fronteira do domínio vizinho, o que equivale a percorrer a distância σ (gráfico da figura 6.3).

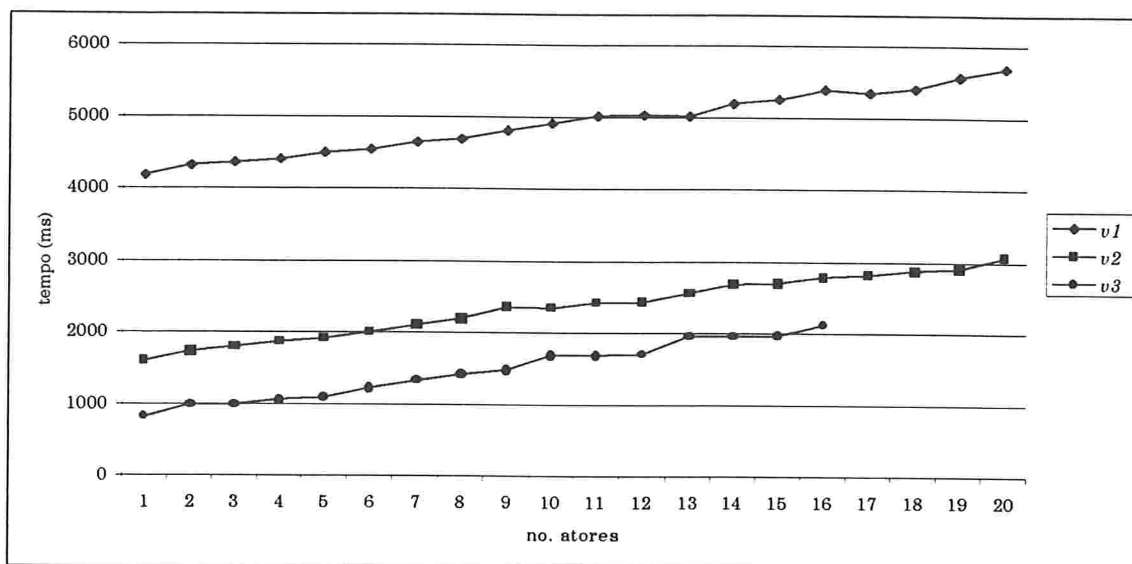


Figura 6.3 – Tempo gasto para percorrer σ na *RF*

O gráfico na figura 6.4 apresenta a relação entre os tempos obtidos nas duas etapas para as três velocidades testadas, ou seja:

$$(tempo\ RFv_i)/(tempo\ Dv_i), \text{ para } i \text{ variando de 1 até 3.}$$

O termo (*tempo* RFv_i) representa o tempo médio da segunda etapa gasto por um ator com velocidade v_i, e o termo (*tempo* Dv_i) representa o tempo médio da primeira etapa com a mesma velocidade v_i.

No gráfico podemos observar que, quanto maior a velocidade, maior é a razão entre os tempos, ou seja, maior é o efeito visual de descontinuidade de movimento notado pelo usuário. Por ser a velocidade v_2 uma velocidade normal neste tipo de jogos, consideramos que o atraso de um pouco mais que 100% é razoavelmente pequeno tendo em vista todos as ações envolvidas na migração entre domínios. Visualmente, para o usuário isto representa uma pequena parada na animação da nave por ele controlada.

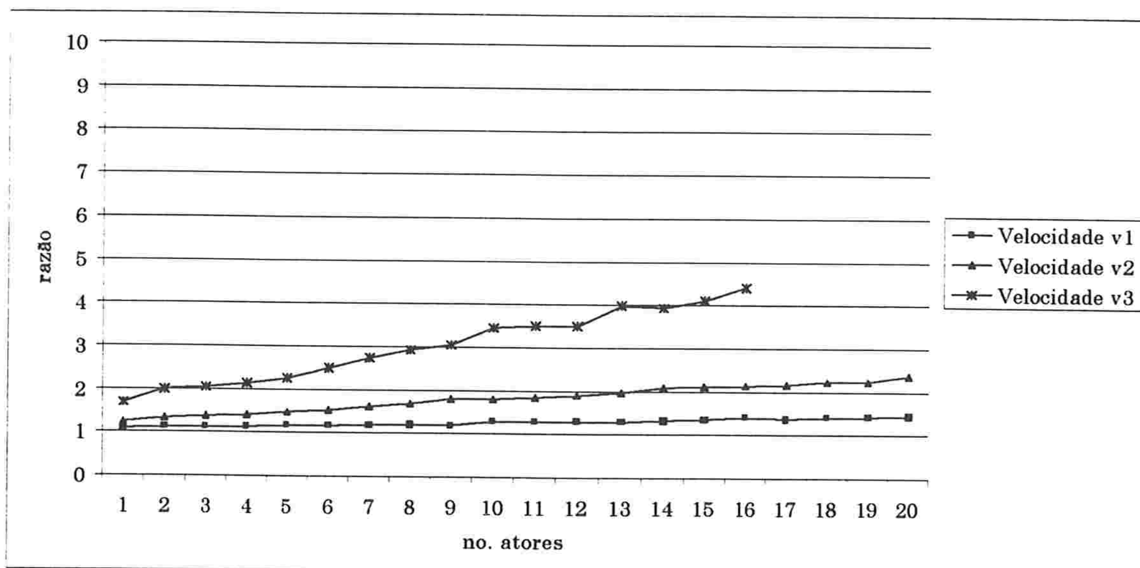


Figura 6.4 – Efeito visual da migração para o usuário

6.2 Teste n.º 2: Tempo gasto por ações executadas durante o protocolo de migração entre domínios

O objetivo deste teste foi o de identificar as ações ou tarefas que têm maior contribuição para o atraso na atualização da visão do usuário.

Durante o protocolo de mudança de domínios, descrito na seção 4.3, ocorre a seguinte seqüência de ações:

1. *Detecção da região de fronteira (RF)*: tempo necessário para a detecção de colisão do ator com a *RF*.
2. *Pedido ao servidor de pertinência (SP)*: tempo para que o visualizador do ator envie mensagem ao *SP*, pedindo a lista de domínios vizinhos relativos à região de fronteira no qual o ator entrou.
3. *Resposta do SP*: tempo de espera pela resposta do *SP* ao pedido feito no item 2.
4. *Recebimento da lista de vizinhos*: tempo de estabelecimento de uma conexão *TCP* com *SP*, e recebimento dos dados relativos aos servidores de domínios (*SDs*), correspondentes aos domínios vizinhos.
5. *Conexão com servidor de domínio (SD)*: tempo de estabelecimento de uma conexão *TCP* com *SD*, por onde será enviado o respectivo domínio.

6. *Recebimento do domínio*: tempo gasto para o recebimento completo do domínio.
7. *Join Group*: tempo de entrada em um novo grupo *multicast.*
8. *Passagem de borda*: tempo necessário para detecção de colisão com limite do domínio.
9. *Leave Group*: tempo necessário para sair dos grupos *multicast* (ao sair da região de fronteira).
10. *Remoção da lista de vizinhos*: tempo para descartar lista antiga de vizinhos.

Por representarem apenas computações locais, com tempos desprezíveis comparados aos tempos de comunicação, as ações 1, 8 e 10 não foram representadas no gráfico da figura 6.5. Através deste gráfico, pode-se verificar que a ação que mais contribui para o atraso na atualização da visão do usuário é o recebimento dos objetos contidos no domínio para o qual o ator está migrando. Isto se deve ao fato de que nesta versão do protótipo transferimos todo o conjunto de objetos, sem qualquer preocupação em restringir a quantidade de dados trocada entre o servidor de domínio e o visualizador do ator.

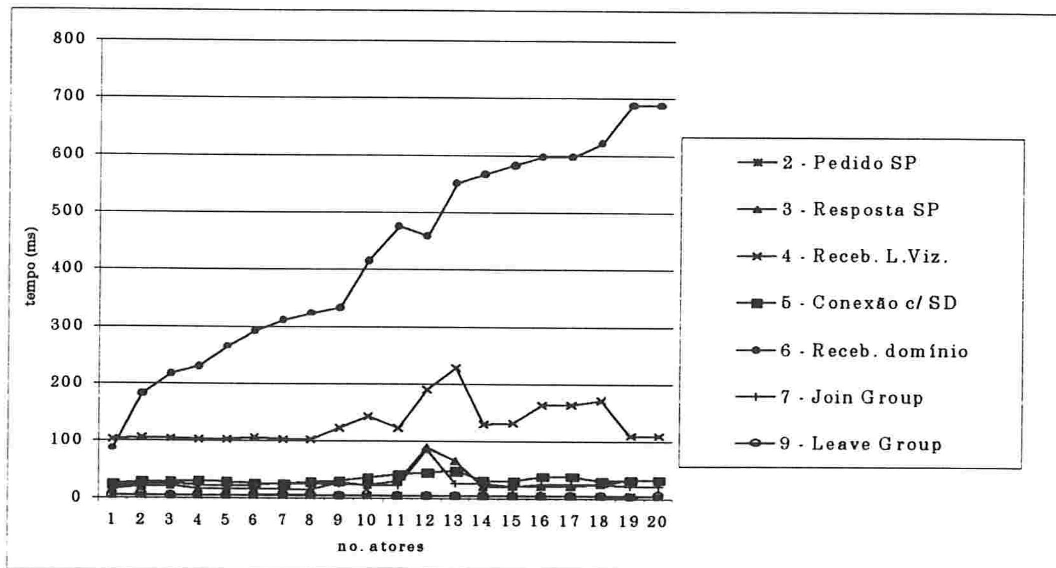


Figura 6.5 – Tempo gasto por ações durante migração

6.3 Teste n.º 3: Influência do compartilhamento dos servidores comuns na visão do usuário

Como mencionado no início do capítulo, o objetivo deste teste foi avaliar o impacto que o compartilhamento dos servidores (tais como servidores de pertinência e servidores de domínios) pelos visualizadores tem sobre a atualização da visão do mundo virtual pelo usuário.

Neste teste fizemos as medições similares às do teste anterior, mas agora para uma situação mais realista, onde todos os atores estão em movimento pelo mundo virtual.

Basicamente, repetimos o segundo teste para 2, 5, 8 e 12 atores no mundo virtual, para um ator com velocidade v_2 e robôs movimentando-se horizontalmente e verticalmente pelo mundo virtual com a velocidade v_2 (gráfico da figura 6.6).

Neste novo teste percebe-se que ainda a maior contribuição no atraso deve-se ao recebimento de domínios e ao recebimento da lista de vizinhos.

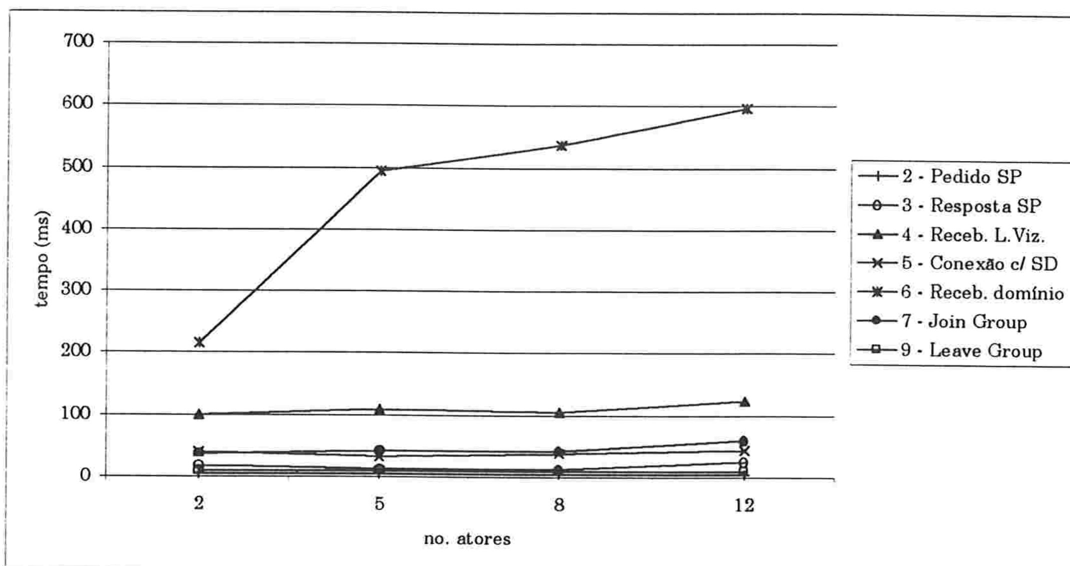


Figura 6.6 – Tempo gasto por ações com robôs em movimento

Na figura 6.7 apresentamos a razão entre o tempo gasto pelas ações durante o processo de migração descritas no segundo e terceiro testes. Comparando-se os valores obtidos no teste 2 e 3, percebe-se que o compartilhamento dos servidores de pertinência e servidores de domínios pelos demais atores praticamente só tem influência sobre o atraso no recebimento dos objetos do domínio, isso é mostrado na

figura 6.7 pelo fato de que a razão entre os dois tempos varia entre 1 e 2, aproximadamente.

Deve-se observar que testes envolvendo robôs estão sujeitos a grandes variações que são dependentes da situação particular que os atores (robôs) estão em cada momento. Por exemplo, se a maioria dos atores por acaso estiver migrando para um mesmo domínio, é de se esperar uma contenção maior no acesso ao servidor de domínio.

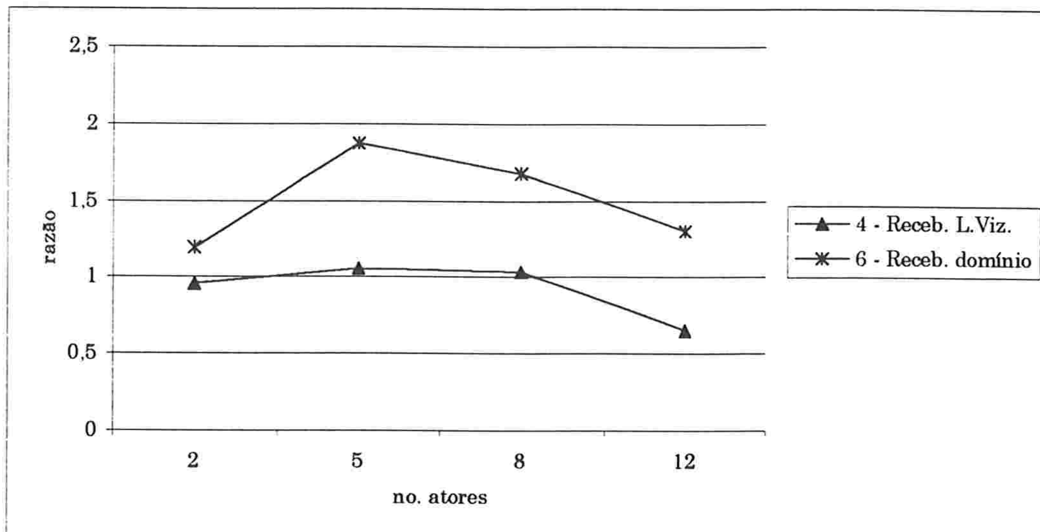


Figura 6.7 – Razão entre tempo gasto pelas ações no segundo e terceiro testes

Com este conjunto de testes pudemos verificar que, de um modo geral, os protocolos apresentados no capítulo 4 não causam uma degradação muito perceptível na visão do usuário. Concluímos ainda que o fator que mais contribui para o atraso na atualização da visão do usuário é o tempo de espera pelo envio de domínios, que cresce rapidamente quando se tem muitos atores participando de uma simulação. Uma forma de diminuir o *overhead* pelo recebimento de um domínio seria reduzir a quantidade de informações enviadas, como por exemplo: a) um visualizador poderia armazenar em um banco de dados local todas as informações relativas a todos os domínios no mundo virtual (tais como, topografia, objetos virtuais nos domínios, etc.); e b) ao invés de enviar um objeto com todos os atributos dos demais atores no domínio corrente, poderia-se enviar uma lista somente com suas respectivas posições e velocidades, enquanto que as demais informações já estariam previamente armazenadas no banco de dados local do visualizador.

Temos consciência de que os testes realizados não foram completos ou conclusivos, mas permitiram apenas comprovar a viabilidade dos protocolos desenvolvidos neste trabalho.

Capítulo 7

Conclusão

Ambientes Virtuais Distribuídos (*AVDs*) apresentam um excelente potencial como um futuro meio de interação pela rede Internet. Isto se deve às formas intuitivas com que se pode navegar, apresentar, compartilhar e interagir com informações em ambientes bi e tridimensionais. Embora *AVDs* apresentem muitas vantagens do ponto de vista de interface, ainda existem algumas limitações técnicas que impedem seu uso em larga escala.

Nesta dissertação, abordamos técnicas para a construção de *AVDs* compartilhados por muitos usuários, buscando novas formas de otimização da comunicação e redução do volume e freqüência das informações difundidas entre os elementos distribuídos.

A principal contribuição deste trabalho foi o desenvolvimento de protocolos distribuídos em uma arquitetura de servidores descentralizados que permitem um particionamento de um mundo virtual em regiões (domínios) regulares, e de tal forma que o usuário não perceba a migração de um ator de um domínio para outro. Este particionamento teve como principal motivação um particionamento do número de atores que precisam ser informados mutuamente sobre suas locomoções dentro do mundo virtual, reduzindo assim o tráfego de mensagens de atualização no sistema.

A fim de mostrar a viabilidade de nossa abordagem, implementamos um protótipo, que é extensão de um jogo tipo *Asteroids*, chamado *JAVAroids* [8] e programado em Java. A partir dos resultados de vários testes e medições de tempo

usando este protótipo, pudemos comprovar que, apesar da sobrecarga causada pelos protocolos envolvidos em uma migração, na maioria dos casos o efeito visual para o usuário é somente uma pequena retração na animação dos movimentos dos atores. Além disso, pudemos verificar que a maior parcela de atraso é devido à transmissão do estado do novo domínio para o visualizador do ator correspondente. Este fato indica um possível aspecto de nosso trabalho que poderia se beneficiar bastante de uma otimização. Em particular, em vez de transmitir todos os dados referentes a um novo domínio (incluindo a descrição completa de todos os atores presentes), poderia-se pensar em enviar somente as características do domínio que variam no tempo. No entanto, de acordo com nossos objetivos de mostrar a viabilidade da arquitetura e dos protocolos desenvolvidos, tivemos uma preocupação menor com aspectos de otimização.

Não sabemos de nenhum outro trabalho que tenha tratado do problema do particionamento de um mundo virtual e que tenha proposto protocolos para garantir uma migração transparente entre domínios. O sistema AGORA é o único a propor um particionamento de um mundo virtual 3D em regiões regulares, mas não houve uma preocupação em garantir uma migração entre domínios transparente para o usuário.

Estamos cientes de que neste trabalho abordamos apenas um problema bem específico relacionado a *AVDs*: o do particionamento regular e estático do mundo virtual. Naturalmente, seria desejável permitir um particionamento dinâmico do mundo virtual, a fim de lidar com problemas como a concentração de atores em determinado domínio, ou ser capaz de tratar de falhas espontâneas dos servidores que fazem parte de nossa arquitetura. Neste sentido, consideramos este trabalho como uma experiência inicial neste assunto e que possivelmente poderá ajudar na solução de outros problemas interessantes na área de *AVDs*.

Apesar deste trabalho ter tratado do problema da migração no contexto específico de Ambientes Virtuais Distribuídos, sabemos que problemas muito similares existem também em outras áreas de sistemas distribuídos, tais como computação móvel, agentes móveis, e outros.

Ou seja, todos os problemas nos quais:

- Elementos se movimentam de forma aleatória em um ambiente dividido em domínios, cada qual com seu servidor específico;

- Existe uma diferença entre a capacidade de comunicação intra e inter-domínios;
- A troca de um servidor para outro deve ocorrer de forma automática para o elemento em movimento, isto é, este não deve precisar executar qualquer ação para efetivar a troca; e
- A passagem de um domínio para outro (e a conseqüente mudança de servidor) deve ocorrer de forma contínua e imperceptível para o elemento

provavelmente deverão empregar protocolos parecidos com os desenvolvidos neste trabalho. Assim, esperamos que com este trabalho tenhamos contribuído também indiretamente para elucidar problemas similares em outras áreas.

Apêndice A

Hierarquia de Classes – Protótipo

Neste apêndice apresentamos os diagramas UML (Unified Modeling Language) [4] que representam a relação de herança entre as classes do protótipo. A figura A.1 apresenta uma descrição dos símbolos UML utilizados nos diagramas deste apêndice.

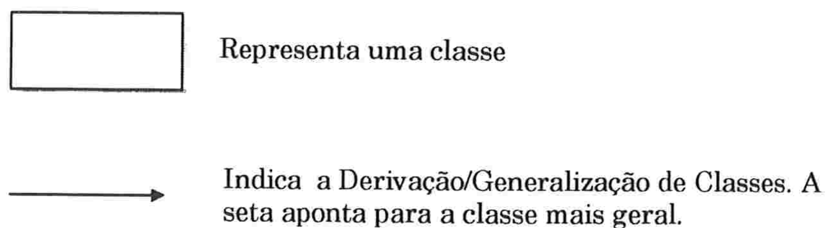


Figura A.1 – Símbolos utilizados na representação dos diagramas de classes

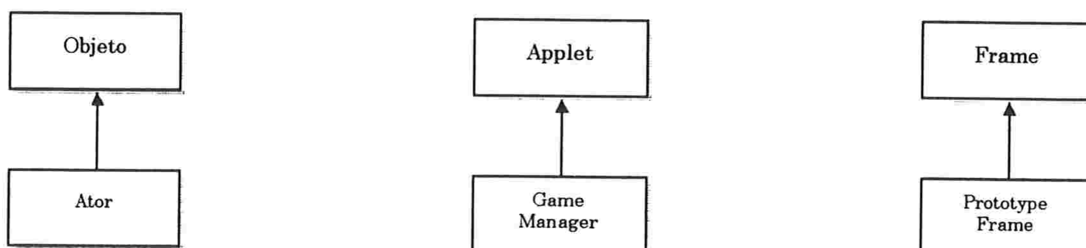


Figura A.2 – Classes derivadas das classes Objeto, *Applet e Frame*

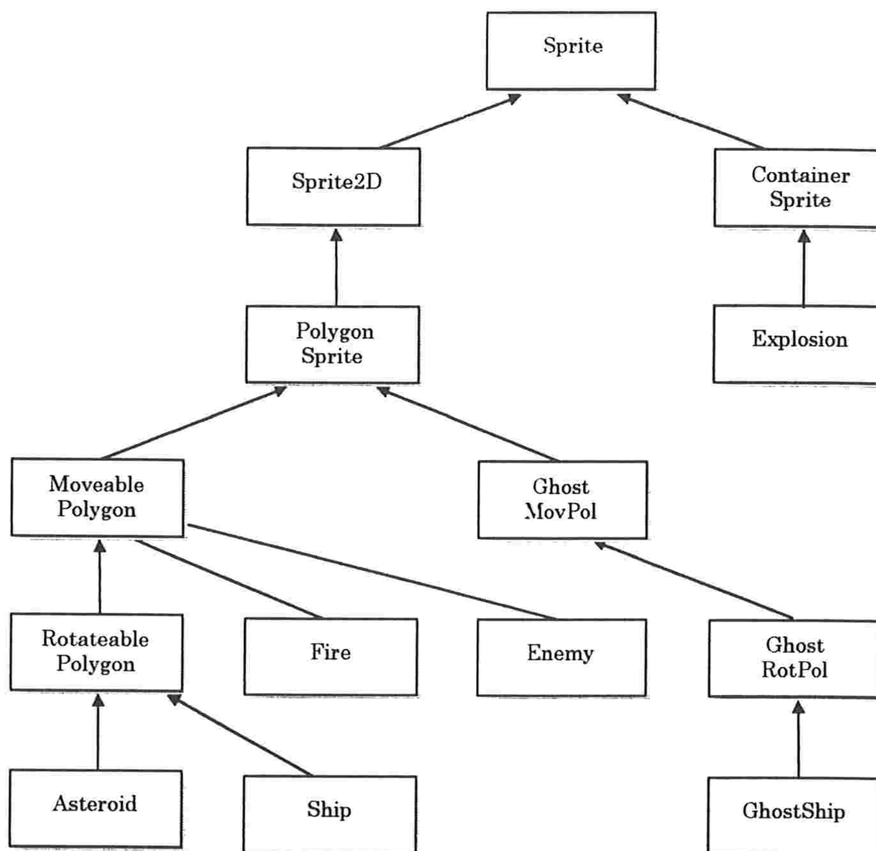


Figura A.3 – Classes derivadas da classe *Sprite*

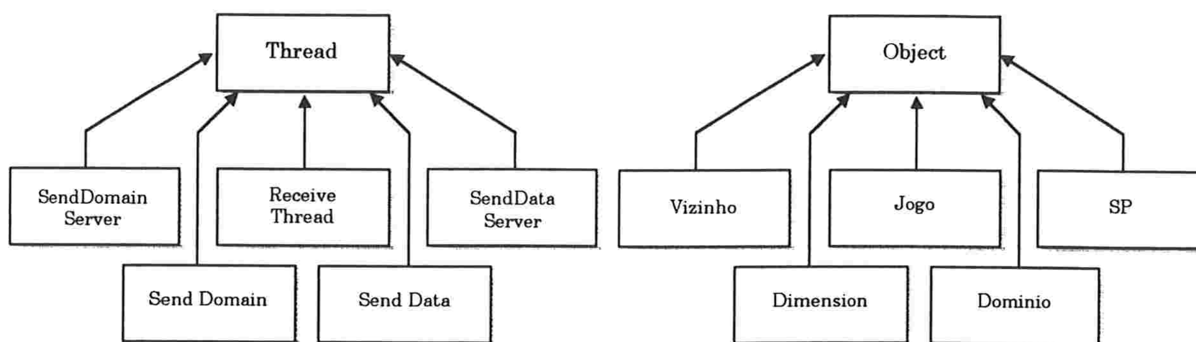


Figura A.4 – Classes derivadas das classes *Thread* e *Object*

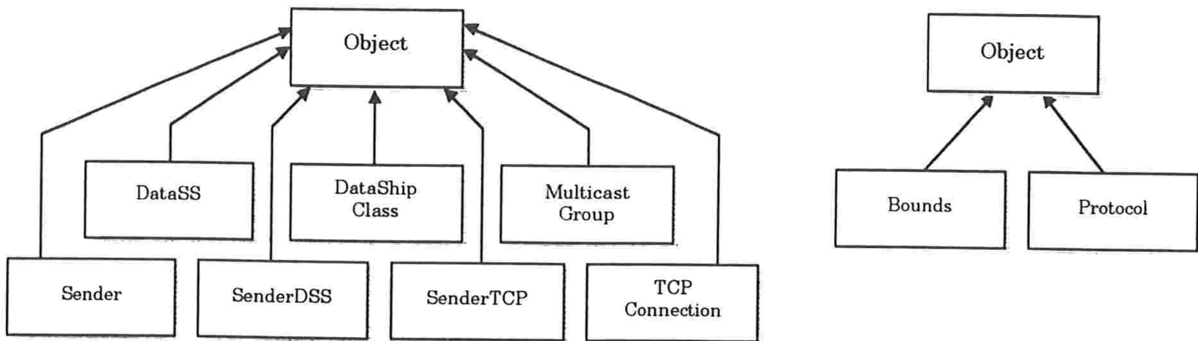


Figura A.5 – Classes derivadas da classe *Object*

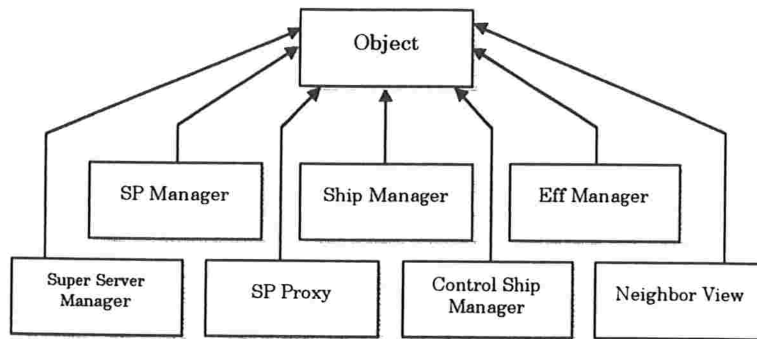


Figura A.6 – Classes derivadas da classe *Object*

Apêndice B

Descrição de Classes e Métodos

Este apêndice apresenta uma lista completa com todas as classes e métodos que integram o *AVD-PTD*.

B.1. Classe *Ator*

Classe que mantém informações sobre a identificação do ator.

***public class** Ator **extends** Objeto **implements** Serializable*

• *Construtores*

***public** Ator(**String** name)*

B.2. Interface *Bounds*

Interface utilizada para definição de constantes relativas às bordas do domínio.

***public interface** bounds*

• *Variáveis*

***public final static byte** RIGHT = 0;*
***public final static byte** LEFT = 1;*
***public final static byte** TOP = 2;*
***public final static byte** BOTTOM = 3;*
***public final static byte** largBound = 11;*

- *Largura da Região de Fronteira.*

B.3. Classe *ContainerSprite*

Mantém lista de sprites.

public class *ContainerSprite* ***extends*** *Sprite*

- *Construtores*

public *ContainerSprite()*

- *Métodos*

public void *addSprite(Sprite s)*

public void *paint(Graphics g)*

public void *update()*

B.4. Classe ControlShipManager

Classe que implementa rotinas de animação da nave (*Ship*) controlada pelo usuário, bem como trata as entradas de comandos do usuário.

public class *ControlShipManager* ***extends Object***

- *Construtores*

public *ControlShipManager(EffManager efm, GameManager game, Point posInicial, ShipManager sm)*

- *Métodos*

public *Ship getShip()*

public *Fire[] getFire()*

public void *stopFire(**int** i)*

public void *extraShip()*

public void *destroyShip()*

private void *clearKeyState()*

public String *PadNumber(**int** n,**int** l)*

public void *paint(Graphics g)*

public void *update()*

public void *newGame()*

public void *increaseShield()*

public void *keyDown(Event e,**int** k)*

public void *keyUp(Event e,**int** k)*

***private void** updateShield()*
***private void** handleFire()*

B.5. Classe *DataShipClass*

Esta classe armazena dados sobre novos atores, como por exemplo a identificação do novo ator e a posição e velocidade iniciais da nave (*Ship*), que são enviados para grupo *multicast* do domínio.

***class** dataShipClass **implements** Serializable*

- *Construtores*

***public** dataShipClass (**int** sx, **int** sy, **int** vx, **int** vy, **int** theta, Ator ator)*
***public** dataShipClass (**int** tx[], **int** ty[], **int** n, **int** centerx, **int** centery,*
*Color c, Dimension d, **int** r, Ator ator)*

- *Métodos*

***public void** store(**int** nsx, **int** nsy, **int** nvx, **int** nvy, **int** ntheta, Ator nator)*

B.6. Classe *DataSS*

A classe DataSS armazena informações (tais como portas TCP, multicast, etc.) que são trocadas entre SServ, SPs e visualizadores de atores.

***public class** dataSS **implements** Serializable*

- *Construtores*

***public** dataSS(InetAddress myAddrs, **int** myPort, **int** pID, **int** tipo)*
***public** dataSS(InetAddress myAddrs, **int** myPort, **Point** dim, **int** tipo)*

B.7. Classe *Dimension*

Armazena os limites de domínio, que neste protótipo é bidimensional.

***public class** Dimension **implements** Serializable*

- *Construtores*

***public** Dimension(**int** top, **int** bottom, **int** left, **int** right)*

B.8. Classe *Dominio*

A componente *Domínio* é responsável por armazenar a lista de atores e objetos presentes em um domínio, além de possíveis atributos específicos de cada domínio.

public class *Dominio* ***implements*** *Serializable*

- *Construtores*

public *Dominio(Dimension dDomain,* ***int*** *numserv,* ***int*** *ind)*

- *Métodos*

public synchronized void *insereObj(GhostShip obj, Ator ator)*
public synchronized void *removeObj(Ator ator)*
public synchronized void *updateObj(GhostShip obj, Ator ator)*
public synchronized *GhostShip getObj(Ator ator)*
public synchronized void *insereAtor(Ator ator)*
public synchronized void *removeAtor(Ator ator)*
public synchronized void *removeAtor(****String*** *nome)*
public synchronized void *updateAtor(Ator ator)*
public synchronized *Objeto getAtor(Objeto ator)*
public synchronized *Ator getMeuAtor()*
public String *getNomeMeuAtor()*
public String *getID()*
public int *getIndice()*
public int *getNumServidor()*
public void *setIndice(****int*** *indice)*
public void *setNumServidor(****int*** *numServidor)*
public *Hashtable getListaObj()*
public *Hashtable getListaAtores()*

B.9. Classe *EffManager*

Trata os efeitos do jogo, tais como explosões, sons, etc.

public class *EffManager* ***extends Object***

- *Construtores*

public *EffManager(AudioClip expsound)*

• *Métodos*

public void *addShipExplosion(**int** x,**int** y)*
public void *addAstExplosion(**int** x,**int** y)*
public void *addEnemyExplosion(**int** x,**int** y)*
public void *addExplosion(Color c,**int** x, **int** y, **int** u)*
public void *paint (Graphics g)*
public void *update()*

B.10. Classe *Explosion*

Realiza a animação de um conjunto de linhas, passando a impressão de ondas causadas pelo choque de dois objetos.

public class *Explosion* ***extends*** *ContainerSprite*

• *Construtores*

public *Explosion(Color color, **int** x, **int** y, **int** u)*

• *Métodos*

public void *update()*
public boolean *isDone()*

B.11. Classe *Fire*

A classe *Fire* realiza a animação de um disparo de uma nave. O conceito de aura é utilizado na classe *Fire* para a detecção de colisão do objeto *Fire* com outros objetos.

public class *Fire* ***extends*** *MoveablePolygon*

• *Construtores*

public *Fire(Color c, Dimension d)*
public *Fire(Color c,Dimension d,**int** length,**int** updates)*

• *Métodos*

public void *initialize(**int** x, **int** y, **int** angle)*
public void *update()*
public boolean *intersect(Ship s)*

B.12. Classe *GameManager*

A classe *Game Manager* implementa o visualizador, responsável por iniciar o programa do usuário.

public class *GameManager* ***extends*** *Applet* ***implements*** *Runnable, bounds*

- *Métodos*

public static void *main(**String** args[])*

- neste método lê-se o arquivo de configuração Setup.dat para obtenção dos endereços e portas do super-servidor. E também define uma posição inicial para o objeto Ship, caso esta não seja fornecida pelo usuário ou esta esteja fora das dimensões do mundo virtual.

public void *init()*

- inicializa um Applet, definindo limites da imagem.

public void *start()*

- instancia EffManager(), ControlShipManager() e ShipManager().

public void *stop()*

public void *setGameOver()*

private void *newGame()*

public boolean *mouseUp(Event e,**int** x,**int** y)*

public boolean *keyDown(Event e, **int** k)*

public boolean *keyUp(Event e, **int** k)*

public void *run()*

private void *startInstructions()*

private void *updateOpening()*

public void *updateScore(**int** val)*

public void *update(Graphics g)*

public String *PadNumber(**int** n,**int** len)*

public void *paint(Graphics g)*

public void *paintInstructions(Graphics g)*

public void *paintTestDrive(Graphics g)*

public static String *getToken()*

- *lê palavra do arquivo de configuração inicial.*

B.13. Classe *GameMath*

Contém rotinas matemáticas para cálculos dos gráficos e programas do jogo.

public final class *GameMath*

- *Métodos*

public static float *cos(****int*** *degree)*

public static float *sin(****int*** *degree)*

public static double *computeAngle(****int*** *v1x,* ***int*** *v1y)*

public static float *computeMagnitude(****int*** *v1x,* ***int*** *v1y)*

public static int *getRand(****int*** *Max)*

public static float *getRand(****float*** *Max)*

public static double *getRand(****double*** *Max)*

B.14. Classe *GhostMovPol*

Implementa métodos de movimentação de um polígono que representa um ator-fantasma.

class *GhostMovPol* ***extends*** *PolygonSprite* ***implements*** *Moveable, bounds, Serializable*

- *Construtores*

public *GhostMovPol(****int*** *tx[],* ***int*** *ty[],* ***int*** *n,* ***int*** *centerx,* ***int*** *centery,*
Color c, ***boolean*** *isActor)*

public *GhostMovPol(****int*** *tx[],* ***int*** *ty[],* ***int*** *n,* ***int*** *centerx,* ***int*** *centery,*
Color c, Dimension d, ***int*** *r,* ***boolean*** *isActor)*

- *Métodos*

public void *setPosition(****int*** *x,* ***int*** *y)*

public void *setVelocity(****int*** *x,* ***int*** *y)*

public void *scale(****double*** *factor)*

public void *updatePosition()*

public void *updatePoints()*

public void *update()*

- *Interfaces*

interface *Moveable*

***public** abstract **void** setPosition(**int** x, **int** y)*
***public** abstract **void** setVelocity(**int** x, **int** y)*
***public** abstract **void** updatePosition()*

B.15. Classe *GhostRotPol*

Implementa métodos de rotação de um polígono que representa um ator-fantasma.

***class** GhostRotPol **extends** GhostMovPol **implements** Serializable*

- *Construtores*

***public** GhostRotPol(**int** tx[], **int** ty[], **int** n, **int** centerx, **int** centery, Color c, Dimension d, **int** r, **boolean** isActor)*

- *Métodos*

***public void** setAngle(**int** a)*
***public void** setRotationRate(**int** r)*
***public void** updateAngle()*
***public void** updateAngle(**int** a)*
***public int** checkAngleBounds(**int** th)*
***public void** rotate(**int** a)*
***public void** update()*

B.16. Classe *GhostShip*

Faz a representação de um ator-fantasma através de um polígono.

***public class** GhostShip **extends** GhostRotPol **implements** Serializable*

- *Construtores*

***public** GhostShip(**int** tx[], **int** ty[], **int** n, **int** centerx, **int** centery, Color c, Dimension d, **int** r, **boolean** isActor, **String** nome)*

- *Métodos*

***public void** rotateLeft()*
***public void** rotateRight()*
***public void** update()*
***public void** thrust()*
***public String** getNome()*

B.17. Classe *Jogo*

Dá início ao programa responsável pela instanciação do super-servidor.

***public class** jogo*

- *Construtores*

***public static void** main(**String** args[])*

- neste método lê-se o arquivo de configuração Setup.dat para obtenção da lista inicial de servidores de pertinência. Este arquivo é passado para o super-servidor que tratará estas informações.

B.18. Classe *MoveablePolygon*

Implementa métodos de movimentação de um polígono que representa um ator associado a um usuário. É nesta classe que ocorre a detecção de mudança de domínios por parte dos atores. Mas esta informação só é tratada na classe ShipManager.

***class** MoveablePolygon **extends** PolygonSprite **implements** Moveable, bounds*

- *Construtores*

***public** MoveablePolygon(**int** tx[], **int** ty[], **int** n, **int** centerx, **int** centery,*
*Color c, **boolean** isActor)*

***public** MoveablePolygon(**int** tx[], **int** ty[], **int** n, **int** centerx, **int** centery,*
*Color c, Dimension d, **int** r, **boolean** isActor,*
ShipManager sm)

***public** MoveablePolygon(**int** tx[], **int** ty[], **int** n, **int** centerx, **int** centery,*
*Color c, Dimension d, **int** r, **boolean** isActor)*

- *Métodos*

***public void** setPosition(**int** x, **int** y)*

***public void** setVelocity(**int** x, **int** y)*

***public void** scale(**double** factor)*

***public void** updatePosition()*

- neste método, caso o objeto seja do tipo Ship, o método checkBoundary() é invocado para a detecção da região de fronteira. Se estiver na região de fronteira e sair dela (migrando para outro domínio), o método leaveBoundary() do ShipManager() é invocado.

***public void** updatePoints()*

***public void** update()*

public void *checkBoundary()*

- Se o objeto Ship estiver na região de fronteira, o método handleBoundary() é invocado, caso contrário, o método leaveBoundary() é invocado, ambos do ShipManager().

public void *updateBounds(Dimension d)*

- quando ator muda de domínio, este método é invocado para atualizar as fronteiras do domínio recebidas via conexão TCP.

- *Interfaces*

interface *Moveable*

public *abstract* ***void*** *setPosition(**int** x, **int** y)*

public *abstract* ***void*** *setVelocity(**int** x, **int** y)*

public *abstract* ***void*** *updatePosition();*

B.19. Classe *MulticastGroup*

Responsável pela criação de um grupo *multicast*.

public class *MulticastGroup*

- *Construtores*

*MulticastGroup (**int** mPort)*

*MulticastGroup (**int** mPort, **String** mIP)*

- *Métodos*

public boolean *mJoinGroup()*

public boolean *mLeaveGroup()*

public *MulticastSocket getSocket()*

public int *getPort()*

public *InetAddress getGroupAddrs()*

public void *Close()*

B.20. Classe *NeighborView*

Esta classe mantém atualizadas informações sobre domínios vizinhos, quando o ator está sobre a região de fronteira.

public class *NeighborView* ***extends Object implements*** *protocol*

- *Construtores*

public *NeighborView(InetAddress myAddrs,* ***int*** *myPort,* ***int*** *multPort,* ***int*** *edge, InetAddress spAddrs,* ***int*** *spPort)*

- *Métodos*

public synchronized void *init()*

- comunica-se com servidor de pertinência para obter domínio.

private synchronized void *initNetwork()*

- instancia MulticastGroup, ReceiverThread e Sender.

public synchronized void *recebeu(DatagramPacket recv)*

- trata dados recebidos pela porta multicast.

private void *receiveDomain()*

- cria conexão TCP por onde recebe domínio enviado pelo servidor de atores.

private Object *recebeuObjeto(**byte[]** in,**int** offset,* ***int*** *length)*

- recebe pacote enviado para grupo multicast.

B.21. Classe *Objeto*

Representa a estrutura básica de um ator.

public class *Objeto* ***implements*** *Serializable*

- *Construtores*

public *Objeto(Objeto obj)*

public *Objeto(**String** name)*

- *Métodos*

public String *getNome()*

public int *hashCode()*

public boolean *equals(**Object** comObject)*

public void *desenha(Graphics g)*

public void *update()*

- Os métodos desenha() e update() foram substituídos por outros métodos da classe Ship.

B.22. Classe *PolygonSprite*

Armazena e define um polígono.

class *PolygonSprite* ***extends*** *Sprite2D* ***implements*** *Serializable*

- *Construtores*

public *PolygonSprite(**int** x[], **int** y[], **int** n, Color c)*

public *PolygonSprite(**int** tx[], **int** ty[], **int** n, **int** centerx, **int** centery, Color c)*

- *Métodos*

public void *addPoint(**int** x, **int** y)*

public void *paint(Graphics g)*

public void *update()*

B.23. Interface *Protocol*

Interface utilizada para definição de constantes relativas aos protocolos de comunicação entre atores, servidores de pertinência, servidores de domínios e super-servidor.

public interface *protocol*

- *Variáveis*

public final static byte *ZERO = 0;*

public final static byte *JOINED = 1;*

public final static byte *GJOINED = 2;*

public final static byte *MSG = 3;*

public final static byte *INSERE_OBJ = 4*

public final static byte *UPDATE_ATOR = 5;*

public final static byte *LEAVE_DOM = 6;*

public final static byte *UPDATE_OBJ = 7;*

public final static byte *DEAD_RECKONING = 8;*

public final static byte *CLIENT = 9;*

public final static byte *NEWCLIENT = 10;*

public final static byte *ADDRS_SP = 11;*

public final static byte *SEND_TCPPORT = 12;*

public final static byte *REC_TCPPORT = 13;*

public final static byte *NEXT_DOM = 14;*
public final static byte *ND_PORTS = 15;*
public final static String *FIM = "#FIM#";*
public final static byte *SM = 16;*
public final static byte *SP = 17;*
public final static byte *SS = 18;*
public final static byte *NV = 19;*
public final static byte *ADDRS_SD = 20;*
public final static byte *SEND_ADDRS_SD = 21;*
public final static byte *NEW_SD = 22;*
public final static byte *VIZINHO = 23;*

B.24. Classe *PrototypeFrame*

A classe *PrototypeFrame* foi criada para tornar o visualizador uma aplicação Java em vez de uma *applet*, como era o jogo Javaroids original, e para incorporar também tarefas de comunicação.

class *PrototypeFrame* ***extends*** *Frame*

- *Construtores*

public *PrototypeFrame(**String** str)*

- *Métodos*

public boolean *handleEvent(Event evt)*

B.25. Classe *ReceiveThread*

Classe encarregada de receber informações enviadas ao grupo *multicast.*

public class *ReceiveThread* ***extends*** *Thread* ***implements*** *protocol*

- *Construtores*

public *ReceiveThread(MulticastSocket mSocket, ShipManager manager, **int** tipo)*
public *ReceiveThread(MulticastSocket mSocket, SPManager manager, **int** tipo)*
public *ReceiveThread(MulticastSocket mSocket, SuperServerManager manager, **int** tipo)*
public *ReceiveThread(MulticastSocket mSocket, NeighborView manager, **int** tipo)*

• *Métodos*

public void *run()*

public boolean *On()*

public boolean *Off()*

B.26. Classe *RotateablePolygon*

Implementa métodos de rotação de um polígono que representa objetos como *Ship*, *Asteroid*, etc.

class *RotateablePolygon* ***extends*** *MoveablePolygon*

• *Construtores*

public *RotateablePolygon(**int** tx[], **int** ty[], **int** n, **int** centerx, **int** centery, Color c, Dimension d, **int** r, **boolean** isActor, ShipManager sm)*

• *Métodos*

public void *setAngle(**int** a)*

public void *setRotationRate(**int** r)*

public void *updateAngle()*

public void *updateAngle(**int** a)*

public int *checkAngleBounds(**int** th)*

public void *rotate(**int** a)*

public void *update()*

B.27. Classe *SendData*

Esta classe envia informações sobre um novo domínio, solicitadas pelo visualizador que criou a conexão. A classe *SendData* é instanciada pela classe *SendDataServer*, apresentada a seguir.

public class *SendData* ***extends*** *Thread*

• *Construtores*

public *SendData(Socket sendSocket, **Object** data)*

• *Métodos*

public void *run()*

public void *manda()*

B.28. Classe *SendDataServer*

Cria um *ServerSocket* que fica esperando um pedido de conexão vindo de um visualizador de um ator.

public class *SendDataServer* ***extends*** *Thread*

- *Construtores*

*SendDataServer(****Object*** *data, InetAddress serverAddrs,* ***int*** *serverPort)*

- *Métodos*

void *setData(dataSS data)*
public void *run()*
public void *StopSending()*
public boolean *isSending()*

B.29. Classe *SendDomain*

Um objeto do tipo *SendDomain* é instanciado toda vez que uma conexão TCP é estabelecida entre um visualizador de um novo ator e um servidor de domínio. Esta classe envia o estado atual do domínio, solicitado pelo visualizador (do novo ator) que criou a conexão.

public class *SendDomain* ***extends*** *Thread*

- *Construtores*

public *SendDomain(Socket sendSocket, Dominio dominio,****int*** *novoServidor,****int*** *indice)*

- *Métodos*

public void *run()*
void *manda()*

B.30. Classe *SendDomainServer*

Cria um *ServerSocket* que fica esperando um pedido de conexão vindo de um visualizador associado a um ator.

public class *SendDomainServer* ***extends*** *Thread*

• *Construtores*

SendDomainServer(Pong pong, Dominio dominio, InetAddress serverAddrs,
__int__ serverPort, __int__ numServidor)

• *Métodos*

public void *run()*
public int *getContador()*
public void *setContador(**int** cont)*
public void *StopSending()*
public boolean *isSending()*
public void *Close()*

B.31. Classe *Sender*

A classe *Sender* é responsável por enviar pacotes, utilizando o socket *multicast*, para outros membros de um grupo *multicast*.
public class *Sender* ***implements*** *protocol*

• *Construtores*

Sender(MulticastGroup mGroup)

• *Métodos*

public void *send(**String** s)*
public void *send(**Object** objeto, **int** TIPO)*
public void *send(**Object** objeto, **int** TIPO, InetAddress mAddrs, **int** mPort)*

B.32. Classe *SenderDSS*

A classe *SenderDSS* é responsável por enviar datagramas do *SServ* para o visualizador de novos atores.
public class *SenderDSS* ***implements*** *protocol*

• *Construtores*

SenderDSS()

• *Métodos*

public void *send(**String** s)*

***public void** send(**Object** objeto, InetAddress mAddrs, **int** mPort, **int** TIPO)*

B.33. Classe *Ship*

Esta classe armazena informações, tais como posição e velocidade de um ator, para efetuar a representação gráfica do mesmo no jogo.

***public class** Ship **extends** RotateablePolygon*

- *Construtores*

***public** Ship(**int** tx[], **int** ty[], **int** n, **int** centerx, **int** centery, Color c, Dimension d, **int** r, **boolean** isActor, ShipManager sm)*

- *Métodos*

***public void** rotateLeft()*
***public void** rotateRight()*
***public void** update()*
***public void** thrust()*

B.34. Classe *ShipManager*

A classe *ShipManager* implementa a comunicação entre visualizadores, e entre *SPs* e visualizadores.

***public class** ShipManager **extends Object implements** protocol, bounds*

- *Construtores*

***public** ShipManager(**String** ssIP, **int** ssPort, **Point** posInicial)*

- inicializa variáveis e invoca métodos de inicialização do ambiente de comunicação.

- *Métodos*

***public synchronized void** SMReady()*

- método que retorna "verdadeiro" caso a classe ShipManager já tenho sido totalmente instanciada. A classe GameManager aguarda SMReady() retornar "verdadeiro" para que ela possa instanciar a classe ControlShipManager.

***public synchronized void** initServCli()*

- verifica se ator deve ser inicializado como cliente (initClient()), ou como servidor de atores (initServer()). Invoca o método init().

***public synchronized void** init()*

- cria instância do ator e envia mensagem do tipo JOINED para o grupo multicast.

***private synchronized void** initNetwork()*

- consulta ao Super Servidor para obter o endereço multicast do SP. Obtido o endereço, cria o grupo multicast e instancia as classes ReceiveThread() e Sender().

***public void** ReceivePorts(MulticastSocket mSocket)*

- recebe portas do SP, enviadas pelo SServ. Esta informação é tratada no método recebeu().

***public synchronized void** recebeu(DatagramPacket recv)*

- neste método, as informações vindas do multicast receiver são tratadas. As mensagens recebidas são dos seguintes tipos:

- ***JOINED**: mensagem contendo informações sobre novo ator, informando que este agora é integrante do domínio e que seu respectivo fantasma deve ser criado e inserido no domínio.*
- ***LEAVE_DOM**: mensagem do visualizador de um ator que está migrando para outro domínio.*
- ***DEAD_RECKONING**: mensagem de atualização de atores-fantasmas.*
- ***ADDRS_SP**: mensagem com informações sobre portas multicast e TCP do SP, vindas do SServ.*
- ***NEW_SD**: mensagem do SP informando a identificação do novo servidor de domínio.*
- ***ADDRS_SD**: mensagem contendo o endereço do atual servidor de domínio.*
- ***REC_TCPPORT**: mensagem informando porta TCP por onde o SP enviará domínio.*
- ***ND_PORTS**: mensagem contendo informações sobre domínio vizinho.*

***private synchronized void** initServer(**int** novoServidor, **boolean** primeiro)*

- este método é invocado quando ator é declarado servidor de atores do domínio. Envia mensagem solicitando ao SP informações sobre o domínio. Cria thread para tratar pedidos de envio do estado atual do domínio, vindos de novos atores.

***private void** initClient()*

- quando já existe um servidor de atores, este método é invocado pelo novo ator para que ele possa inicializar as variáveis relativas ao servidor de atores e, invocando o método receiveDomain, receber o domínio atual.

***private synchronized void** receiveDomain()*

- aqui uma conexão TCP é estabelecida com servidor de atores para o envio do estado atual do domínio.

***private void** receiveNextDom(InetAddress sAddrs, **int** sPort)*

- recebe informações sobre domínio vizinho e armazena em uma lista de vizinhos.

***public** Dominio getDomain()*

***public int** getWidth()*

***public int** getHeight()*

***public** Dimension getDimension()*

***public** Sender getSender()*

***public boolean** isServer()*

***private Object** recebeuObjeto(**byte[]** in,**int** offset, **int** length)*

- lê objeto recebido pelo multicast receiver.

***public synchronized void** handleBoundary (**int** edge)*

- ao entrar na região de fronteira (RF), contacta SP para iniciar processo de transferência para outro domínio.

***public synchronized void** leaveBoundary (**int** edge, **boolean** saiuDom, MoveablePolygon s)*

- existem duas situações possíveis quando ator sai da RF:

- *Ator permanece no domínio, e as informações sobre domínio vizinho são descartadas.*
- *Ator ultrapassa as bordas do domínio, migrando assim para outro domínio. Usando as informações sobre próximo domínio, já previamente conhecidas, armazenadas na lista de domínios vizinhos, o ator é atualizado passando a integrar agora o outro domínio.*

***private** NeighborView getNView(**int** edge)*

- varre lista de domínios vizinhos, em busca das informações sobre o domínio para o qual o ator está migrando.

***private synchronized void** removeNView()*

- remove da lista de domínios vizinhos a visão do domínio, cuja RF ator deixou.

***private synchronized void** removeAllNView()*

- remove todos os elementos da lista de domínios vizinhos. Este método é invocado quando ator migra para outro domínio.

***void** deadReckoning(Ship s)*

- verifica se houve mudança significativa da posição real do ator para o ator-fantasma que o representa nas cópias do domínio mantidas pelos outros atores. Caso haja grande

alteração de percurso, ou mesmo velocidade, o ator envia mensagem de atualização para o grupo multicast.

***public void** desenha(Graphics g, Ator meuAtor)*

- desenha todos os "ghosts" no domínio.

***public void** ghostUpdate()*

- atualiza fantamas, utilizando téc. de Dead Reckoning.

***public void** updateGhost (dataShipClass dS)*

- atualiza ator-fantasma com dados recebidos através da porta Multicast.

B.35. Classe *Sp*

Dá início à aplicação responsável pela instanciação do servidor de pertinência.

***class** sp*

• *Métodos*

***public static void** main(**String** args[])*

- neste método lê-se o arquivo de configuração inicial Setup.dat para obtenção da lista inicial de servidores de pertinência. Este arquivo é passado para o novo SP que tratará estas informações em busca dos dados iniciais a ele associado.

B.36. Classe *SPManager*

Esta classe implementa um servidor de pertinência que gerencia um domínio, coordenando a entrada/saída de cada ator do mesmo.

***public class** SPManager **extends Object implements** protocol, bounds*

• *Construtores*

***public** SPManager(**String** setup, **String** spID) **throws** UnknownHostException*

- seta variáveis iniciais e invoca métodos de inicialização do ambiente de comunicação.

***private void** InitSP(**int** myPort, **int** MCPort, Dimension dDomain)*
***throws** UnknownHostException*

- cria novo domínio e invoca métodos CriaListaViz() e initNetwork().

***private void** CriaListaViz() **throws** UnknownHostException*

- percorre arquivo de configuração inicial e cria uma lista, relacionando com cada fronteira do domínio seus respectivos domínios vizinhos.

private void *initNetwork()*

- cria o grupo multicast para este domínio e cria instâncias das classes ReceiveThread, Sender e SendDataServer.

public synchronized void *recebeu(DatagramPacket recv)*

- neste método, as informações vindas do multicast receiver são tratadas.

- ***JOINED****: mensagem contendo informações sobre novo ator, informando que este agora é integrante do domínio e que seu respectivo fantasma deve ser criado e inserido no domínio.*
- ***LEAVE_DOM****: mensagem do visualizador de um ator que está migrando para outro domínio. Caso esta ator seja o servidor de domínio, o SP escolhe outro ator como sendo o novo servidor de domínio e envia mensagem informativa para o grupo multicast.*
- ***SEND_TCPPORT****: pedido do visualizador do novo servidor de domínio para o envio de porta TCP.*
- ***SEND_ADDRS_SD****: pedido do visualizador de um novo ator para o envio do endereço do servidor de domínio.*
- ***NEXT_DOM****: recebe pedido de transferência de domínio. Caso exista um domínio para esta fronteira, faz o envio do mesmo para visualizador do ator correspondente.*

private Object *recebeuObjeto(**byte[]** in, **int** offset, **int** length)*

private *Vizinho getViz(**int** edge)*

public String *getToken()*

public String *getID()*

B.37. Classe *SPProxy*

A classe *SPProxy* é um proxy criado pelo *SServ* e é utilizado para consultas sobre qual será o *SP* responsável por um novo ator que entra para a simulação.

public class *SPProxy* ***extends Object***

• *Construtores*

public *SPProxy(**String** spID, InetAddress myAddrs, **int** myPort, **int** MCPort, Dimension d)*

B.38. Classe *Sprite*

Classe que possibilita a representação de objetos que podem ser desenhados na tela.

*abstract **class** Sprite **implements** Serializable*

- *Métodos*

abstract void *paint (Graphics g)*

abstract void *update()*

public boolean *isVisible()*

public void *setVisible(**boolean** b)*

public boolean *isActive()*

public void *setActive(**boolean** b)*

public void *suspend()*

public void *restore()*

B.39. Classe *Sprite2D*

Classe que possibilita a representação de objetos que podem ser desenhados na tela.

*abstract **class** Sprite2D **extends** Sprite **implements** Serializable*

- *Métodos*

public boolean *getFill()*

public void *setFill(**boolean** b)*

public void *setColor(Color c)*

public *Color getColor()*

B.40. Classe *SuperServerManager*

Servidor *SServ* que coordena *SPs* e a entrada de novos atores na simulação.

public class *SuperServerManager* ***extends Object implements*** *protocol*

- *Construtores*

public *SuperServerManager(**String** setup)* ***throws*** *IOException*

- seta variáveis iniciais e invoca métodos de inicialização do ambiente de comunicação.

• *Métodos*

public void *creatSPList()* ***throws*** *UnknownHostException*

- varre arquivo de configuração inicial e cria lista de SPs. Instancia classes SenderDSS e ReceiveThread.

public synchronized void *recebeu(DatagramPacket recv)*

- quando chega uma mensagem do tipo ***NEWCLIENT****, o SServ verifica, através do método LocalizaDominio, em que domínio o novo ator se encaixa, baseado na posição inicial do mesmo, e envia esta informação para que o visualizador do respectivo ator possa contactar o respectivo SP.*

private int *LocalizaDominio(****Point*** *dim)*

private Object *recebeuObjeto(****byte****[] in,* ***int*** *offset,* ***int*** *length)*

public String *getToken()*

B.41. Classe *TCPConnection*

Cria um *ServerSocket* para conexões do tipo *TCP*.

public class *TCPConnection*

• *Construtores*

public *TCPConnection (InetAddress mAddrs,* ***int*** *mPort)*

public *TCPConnection (InetAddress mAddrs)*

• *Métodos*

public boolean *mCreat()*

public *Socket getSocket()*

public int *getPort()*

public *InetAddress getAddrs()*

B.42. Classe *Vizinho*

Classe instanciada pelos *SPs* no momento da criação da lista de *SPs* vizinhos.

public class *Vizinho*

• *Construtores*

public *Vizinho(****String*** *spID,InetAddress mAddrs,****int*** *TCPPort,****int*** *MCPort,****int*** *edge)*

- *Métodos*

public void *setEdge(**int** edge)*

Referências

[1] Castle Hill. *The Advanced Network Systems Architecture (ANSA) Reference Manual*, chapter Architecture Project Management. Cambridge, England, 1989.

[2] Douglas E. Corner. *Internetworking with TCP/IP*, chapter 17, 5. Prentice Hall, 1991.

[3] Fred Shneider. Replication Management using The State-Machine Approach, In Sape Mullender, editor, *Distributed Systems*, chapter 7. Addison-Wesley, 2nd edition, 1993.

[4] M. Fowler, K. Scott. *UML DISTILLED, Applying the standard object modeling language*. Foreword by Grady Booch, Ivar Jacobson, and James Rumbaugh. Addison-Wesley, 1998.

[5] Chris Greenhalgh and Steve Benford. MASSIVE: a Distributed Virtual Reality System Incorporating Spatial Trading. In *Proc, IEEE 15th Int. Conference on Distributed Computing Systems*, 1995.

[6] R. Gossweiler, R. J. Laferriere, M. L. Keller and R. Pausch. An Introductory Tutorial for Developing Multi-User Virtual Environments. *Presence: Teleoperators and Virtual Environments*, 3(4):255-264, dec 1994.

[7] O. Hagsand. Interactive MultiUser VEs in the DIVE System. *IEEE*, 3(1), 1996. http://www.sics.se/dce/dive/dive.html.

[8] J. Fan, E. Ries and C. Tenitchi. *Black Art of JAVAtm Game Programming*, chapter Building the JAVAroids Game. Waite Group Press. 1996.

[9] H. Harada, N. Kawaguchi, A. Iwakawa, K. Matsui and T. Ohno. Space-sharing Architecture for a three-dimensional virtual community. *IOP/IEE Distributed Systems Engineering*, 5:101-106, 1998.

[10] M. R. Macedonia and D. P. Brutzman. MBone provides audio and video across the Internet. *IEEE Computer*, abr 1994.

[11] H.W. Holbrook, S.K. Singhal and D.R. Cheriton. Log-Based Receiver-Reliable Multicast for Distributed Interactive Simulation. *Proc. SIGCOMM'95*, 25(4):328-341, oct 1995. Published as Computer Communications Review.

[12] Regina Borges de Araújo. Sistemas Distribuídos de Realidade Virtual. *Minicurso do 16º Simpósio Brasileiro de Redes de Computadores*. Universidade Federal Fluminense. Rio de Janeiro, maio 1998.

[13] Renata. C. Teixeira e Otto Carlos M. B. Duarte. Requisitos de Ambientes Virtuais Distribuídos de Larga Escala. *1º Workshop de Realidade Virtual (WRV'97)*. Grupo de Teleinformática e Automação (GTA), COPPE/EE, Universidade do Rio de Janeiro, 26-35, nov 1997.

[14] R. Waters, D. Anderson, J. Barrus, D. Brogan, M. Casey, S. McKeown, T. Nitta, I. Sterns and W. Yerazunis. Diamond Park and Spline: A Social Virtual Reality System with 3D Animation, Spoken Interaction and Runtime Modifiability. Technical report, A Mitsubishi Electric Research Laboratory (MERL), nov 1996. TR-96-02a.