UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

FÁBIO REIS CECIN

FreeMMG: uma arquitetura cliente-servidor e par-a-par de suporte a jogos maciçamente distribuídos

Dissertação apresentada como requisito parcial
para a obtenção do grau de
Mestre em Ciência da Computação

Prof. Dr. Cláudio Fernando Resin Geyer
Orientador

Porto Alegre, março de 2005

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Cecin, Fábio Reis

FreeMMG: uma arquitetura cliente-servidor e par-a-par de suporte a jogos maciçamente distribuídos / Fábio Reis Cecin. – Porto Alegre: PPGC da UFRGS, 2005.

101 f.: il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2005. Orientador: Cláudio Fernando Resin Geyer.

1. Jogos maciçamente multijogador. 2. Simulação interativa distribuída. 3. Sistemas peer-to-peer. 4. Jogos de estratégia em tempo real. I. Geyer, Cláudio Fernando Resin. II. Título.

AGRADECIMENTOS

Agradeço a todos os colegas e amigos que ajudaram diretamente ou indiretamente na realização deste trabalho. Rodrigo Real, primeiramente por ter me ajudado em várias situações enquanto colega de mestrado, em segundo lugar por ter topado participar do projeto FreeMMG onde, entre outras realizações, gerou os testes de escalabilidade que aqui são apresentados. Luciano Silva, por ter me dado várias dicas durante o mestrado, como mexer com CVS, Cygwin e vi, além de outras incontáveis dicas e sugestões em vários contextos. Agradeço ao Rafael Jannone, ao Paulo Zaffari e ao Márcio Martins, por terem topado participar de mais uma empreitada maluca comigo. Mário Goulart, pela sua companhia e pelo seu entusiasmo contagiante, e pelo *azile*. Alex Zilermando, por oferecer um ombro amigo e compreensível via ICQ (228340152), escutando as minhas lamentações mesmo enquanto a sua casa caía aos pedaços, em tempo real. Agradeço especialmente ao colega Renato Hentschke, cuja amizade, incentivo, dicas, discussões, caronas, Groo e Age em geral, além de auxílio na revisão do texto, foram decisivos na conclusão deste trabalho.

Agradeço ao professor Jorge Barbosa, Holo-man e "co-orientador não-oficial" deste trabalho. Grande amigo e parceiro em várias empreitadas, inclusive do projeto de pesquisa e desenvolvimento do FreeMMG, que "alavancou" esta dissertação. Agradeço ao professor e amigo Adenauer Yamin, pelos ensinamentos, pela companhia (e fotografias) nos eventos, e pela inspiração como pesquisador e pessoa.

Agradeço ao meu orientador e amigo, o professor Cláudio Geyer, que me ouviu pacientemente enquanto babulciava e rabiscava incoerências relacionadas com uma "solução para jogos distribuídos peer-to-peer" no bar da informática, lá pelos idos de 2002 (ainda não era o FreeMMG). Obrigado por todo o incentivo, toda a paciência e todo o conhecimento passado. Teu apoio foi o componente determinante para a conclusão de mais esta etapa.

Agradeço à CAPES, pelo apoio financeiro concedido como bolsa de mestrado. Ao CNPq, pelo apoio financeiro concedido ao projeto "FreeMMG: Um Framework Livre para Desenvolvimento de Massively Multiplayer Games". À PROPESq/UFRGS, pelo axílio financeiro concedido para a apresentação do artigo do FreeMMG no workshop WJogos, em novembro de 2003. Ao Instituto de Informática da UFRGS, pela infra-estrutura e pelo excelente ambiente de ensino e pesquisa que propicia aos seus "habitantes".

Agradeço à minha família, em especial aos meus pais, Carlos e Márcia, e aos meus avós, Miguel, Maria, Veny e Wally, por tudo.

E, por fim, reservo um agradecimento especial para a minha esposa, Ane. Obrigado, por ter me agüentado até aqui! Só não te agradeço ainda por ter me agüentado por todo o trabalho pois ainda não defendi, e isso pode dar azar...

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

ADI	*Área de Interesse*
API	*Application Programming Interface*
CVE	*Collaborative Virtual Environment*
CPU	*Central Processing Unit*
DHT	*Distributed Hash Table*
DIS	*Distributed Interactive Simulation*
FPS	*First-Person Shooter*
IP	*Internet Protocol*
J2SE	*Java 2 Standard Edition*
MMG	*Massively Multiplayer Game*
MMOFPS	*Massively Multiplayer Online First-Person Shooter*
MMORPG	*Massively Multiplayer Online Role-Playing Game*
MMORTS	*Massively Multiplayer Online Real-Time Strategy*
NVE	*Networked Virtual Environment*
RTP	*Real-time Transport Protocol*
RTS	*Real-Time Strategy*
TCP	*Transmission Control Protocol*
TSS	*Trailing State Synchronization*
UDP	*Unreliable Datagram Protocol*
URL	*Uniform Resource Locator*

LISTA DE FIGURAS

RESUMO

A popularização das tecnologias de acesso à Internet por “banda larga” suportam a crescente proliferação de aplicações do tipo “par-a-par” (peer-to-peer), onde o usuário doméstico, tipicamente consumidor de informação, passa também a atuar como provedor. De forma simultânea, há uma popularização crescente dos jogos em rede, especialmente dos “jogos maciçamente multijogador” (MMG ou *massively multiplayer game*) onde milhares de jogadores interagem, em tempo real, em mundos virtuais de estado persistente. Os MMGs disponíveis atualmente, como *EverQuest* e *Ultima Online*, são implementados como sistemas centralizados, que realizam toda a simulação do jogo no “lado servidor”. Este modelo propicia controle de acesso ao jogo pelo servidor, além de ser muito resistente a jogadores trapaceiros. Porém, a abordagem cliente-servidor não é suficientemente escalável, especialmente para pequenas empresas ou projetos de pesquisa que não podem pagar os altos custos de processamento e comunicação dos servidores de MMGs centralizados. Este trabalho propõe o FreeMMG, um modelo híbrido, cliente-servidor e par-a-par, de suporte a jogos maciçamente multijogador de estratégia em tempo real (MMORTS ou *massively multiplayer online real-time strategy*). O servidor FreeMMG é escalável pois delega a maior parte da tarefa de simulação do jogo para uma rede par-a-par, formada pelos clientes. É demonstrado que o FreeMMG é resistente a certos tipos de trapaças, pois cada segmento da simulação distribuída é replicado em vários clientes. Como protótipo do modelo, foi implementado o jogo *FreeMMG Wizards*, que foi utilizado para gerar testes de escalabilidade com até 300 clientes simulados e conectados simultaneamente no mesmo servidor. Os resultados de escalabilidade obtidos são promissores, pois mostram que o tráfego gerado em uma rede FreeMMG, entre servidor e clientes, é significativamente menor se comparado com uma alternativa puramente cliente-servidor, especialmente se for considerado o suporte a jogos maciçamente multijogador de estratégia em tempo real.

Palavras-chave: Jogos maciçamente multijogador, simulação interativa distribuída, sistemas peer-to-peer, jogos de estratégia em tempo real.

FreeMMG: A Client-Server and Peer-to-Peer Support Architecture for Massively Distributed Games

ABSTRACT

The increasing adoption of “broadband” Internet access technology fosters the increasing development of networked “peer-to-peer” applications, where the end-user, usually the consumer of information, begins to act also as a producer of information. Simultaneously, there is a growing popularity of networked games, especially “massively multiplayer games” (MMGs) where thousands of players interact, in real time, in persistent-state virtual worlds. State-of-the-art massively multiplayer games such as EverQuest and Ultima Online are currently implemented as centralized, client-server systems, which run the entire game simulation on the “server-side”. Although this approach allows the development of commercially viable MMG services, with game access control and player cheating prevention, the processing and communications costs associated with running a MMG server are often too high for small companies or research projects. This dissertation proposes FreeMMG, a mixed peer-to-peer and client-server approach to the distribution aspect of MMGs. It is argued that the FreeMMG model supports scalable, cheat-resistant, massively multiplayer real-time strategy games (MMORTS games) using a lightweight server that delegates the bulk of the game simulation to a peer-to-peer network of game clients. It is shown that FreeMMG is resistant to certain kinds of cheating attempts because each segment of the distributed simulation is replicated by several clients. A working prototype game called FreeMMG Wizards is presented, together with scalability test results featuring up to 300 simulated game clients connected to a FreeMMG server. The obtained results are promising, by showing that the generated server traffic in a FreeMMG network, between the server and the clients, is substantially lower if compared with purely client-server alternatives, especially if the support for massively multiplayer real-time strategy games is considered.

Palavras-chave: massively multiplayer games, distributed interactive simulation, peer-to-peer systems, real-time strategy games.

1 INTRODUÇÃO

1.1 Tema

O tema central deste trabalho compreende os sistemas de suporte (motores ou *middleware*) para uma classe de aplicação denominada *massively multiplayer game* (abreviadamente, MMG). Os MMGs podem ser considerados como derivados dos jogos "multi-jogador" (ou *multiplayer*), que são essencialmente simulações interativas em tempo real que permitem que uma pessoa com um computador conectado a alguma rede (local ou Internet) interaja, em um ambiente virtual, com outros jogadores. O aspecto "tempo real", neste contexto, é uma restrição sobre simulações interativas. Como discutido em (CECIN, 2003), jogos interativos em tempo real são simulações onde o avanço do tempo de simulação ocorre em pequenos passos discretos (geralmente abaixo de 1s) e não depende das ações dos jogadores, como num jogo baseado em turnos.

Os jogos distribuídos chamados simplesmente de "multi-jogador" suportam um número relativamente limitado de participantes simultâneos no "ambiente virtual" onde se desenvolve o jogo. Os limites típicos para jogos como Quake (ID SOFTWARE, 2004) e Half-Life (VALVE, 2004) variam entre 8 e 64 jogadores. Outra característica dos jogos multi-jogador é que estes tem uma duração relativamente curta. Partidas típicas podem durar alguns minutos ou algumas horas, e o resultado de uma partida não afeta o estado da partida seguinte.

Os Massively Multiplayer Games são jogos multi-jogador com duas características adicionais. Primeiro, os MMGs envolvem uma quantidade relativamente grande de jogadores simultâneos interagindo em tempo real. Esta quantidade pode variar entre algumas centenas de jogadores até centenas de milhares ou mesmo milhões de jogadores (KENT, 2004; IGDA, 2004, 2003). Segundo, os MMGs suportam uma simulação de estado persistente. Em outras palavras, o jogo não é dividido em partidas de curta duração: as ações que um jogador realiza afetam de forma duradoura o "mundo virtual" onde este jogador interage. Vários MMGs comerciais estão em funcionamento há anos, onde os participantes continuam jogando a mesma "partida". Exemplos clássicos são Ultima Online (ORIGIN, 2004), EverQuest (SONY, 2004a) e Lineage (NCSOFT, 2003).

Como será discutido no texto deste trabalho, a maioria dos MMGs disponíveis comercialmente utilizam modelos centralizados como suporte à distribuição do jogo. Esta dissertação propõe o FreeMMG, um modelo descentralizado (ou peer-to-peer) de suporte a MMGs. Como será visto, existem várias propostas de modelos descentralizados de suporte a MMGs. Neste contexto, a dissertação posiciona o FreeMMG não só como uma arquitetura descentralizada, mas também resistente a jogadores trapaceiros e que suporta jogos de estratégia em tempo real. Estas três características não foram encontradas, de forma simultânea, em nenhuma das propostas pesquisadas durante a realização deste tra-

balho.

1.2 Contexto de pesquisa

Este trabalho está inserido no projeto “FreeMMG: Um Framework Livre para Desenvolvimento de Massively Multiplayer Games”, de execução em 2004 e de fomento do CNPq (Edital MCT/CNPq/CT-INFO 01/2003). O objetivo do projeto é implementar em Java o modelo FreeMMG, disponibilizando um conjunto de interfaces (que abstraem as dificuldades do modelo) para o desenvolvimento de jogos “massively multiplayer” com servidor de baixo custo e clientes que realizam simulação cooperativa (peer-to-peer).

A implementação do framework FreeMMG é disponibilizada como software livre (licença GNU LGPL), no site do projeto [1]. Esta implementação é baseada no modelo FreeMMG e na implementação do seu protótipo (jogo FreeMMG Wizards), apresentados nos capítulos 3 e 4 desta dissertação, respectivamente.

Este trabalho teve apoio financeiro de uma bolsa de mestrado CAPES concedida entre 2002 e 2003. Adicionalmente, teve apoio financeiro do CNPq oriundo do projeto FreeMMG no ano de 2004.

1.3 Motivação

A indústria do entretenimento eletrônico produz jogos para computadores pessoais, consoles caseiros, aparelhos móveis, etc. Atualmente, esta indústria movimenta mais verba do que a indústria cinematográfica. Dentre os vários tipos de jogos, os jogos *on-line*, especialmente os jogos *on-line* em tempo real, como corridas de carros e expedições em mundos fantásticos, vem ganhando cada vez mais espaço. Este crescimento é parcialmente devido ao crescimento constante do público com acesso à Internet e do sucesso das tecnologias de banda larga.

Atualmente, os MMGs de sucesso comercial são implementados utilizando alguma variante do paradigma cliente-servidor. No paradigma cliente-servidor, o servidor possui a cópia “oficial” do estado e é responsável pela computação sobre a mesma, enquanto o cliente geralmente atua apenas na camada de apresentação. Adicionalmente, o servidor é responsável por atualizar os clientes, em tempo real, sobre as alterações que ocorrem no mundo de jogo simulado. Esta centralização do jogo no “lado servidor” minimiza a ocorrência de trapaças, pois os clientes não podem alterar diretamente a cópia “oficial” do estado do jogo. Porém, devido ao grande número de participantes, o custo de manutenção dos servidores de um MMG é bastante elevado.

Neste contexto, é desejavel a redução do custo da plataforma de execução de um MMG. Trabalhos recentes propõem a distribuição do servidor em várias máquinas interconectadas por redes de alta velocidade, ou seja, resolver o problema através de *cluster computing*. Apesar de *cluster computing* ser uma excelente solução para a distribuição do custo de processamento (CPU), estas propostas, de forma isolada, geralmente não são suficientes para aumentar a escalabilidade do servidor MMG. Isto ocorre porque ainda resta o problema do custo de comunicação entre o *cluster* servidor e os clientes. Devido ao modelo de distribuição adotado por praticamente todas as arquiteturas de MMGs implementadas comercialmente (IGDA, 2004), o servidor é responsável por enviar, continuamente e em tempo real, um fluxo de informação para que o cliente tenha sempre

[1] http://sourceforge.net/projects/freemmg/

uma versão atualizada do “mundo virtual”. Este tipo de atualização de estado, quando utilizado em um contexto de jogo maciçamente multijogador, resulta na necessidade da alocação de uma ligação de rede dedicada, com baixo atraso, poucas perdas de pacotes e largura de banda elevada para que o servidor possa atualizar a “visão do jogo” de milhares de jogadores simultaneamente (IGDA, 2004). Este aspecto da motivação é discutido em mais detalhes no capítulo 2.

Esta dissertação propõe o modelo FreeMMG como uma alternativa às arquiteturas centralizadas que são empregadas atualmente na grande maioria dos MMGs. O modelo FreeMMG propõe a delegação do processo simulador para os próprios clientes do jogo, em uma abordagens baseada em peer-to-peer. Desta forma, o equipamento dos próprios jogadores é utilizado para fornecer o poder de processamento e de comunicação necessários para executar a simulação.

Porém, isto não é algo essencialmente novo: na área de simulação distribuída e interativa (???, ????) já existem trabalhos tradicionais que tratam do problema da distribuição de uma simulação em uma rede sem recair em uma hierarquia do tipo cliente-servidor entre os participantes da simulação. Porém, os MMGs, que são sistemas que se utilizam da Internet como meio de comunicação entre jogadores anônimos, apresentam uma peculiaridade: a necessidade de proteção contra participantes trapaceiros. Isto foi discutido por Cunnin (CUNNIN, 2000) em sua dissertação de mestrado. Cunnin argumenta que uma abordagem como DIS (TARR; JACOBS, 2002) não pode ser empregada para jogos *on-line* pois é muito vulnerável ao ataque de jogadores trapaceiros.

A motivação principal deste trabalho é propor uma solução para a redução tanto do custo computacional (CPU) quanto do custo de comunicação (rede) associado à entidade servidora nas arquiteturas cliente-servidor tradicionais de MMGs. Porém, apesar de reduzir custos, deve ser mantido o mesmo nível de proteção contra jogadores trapaceiros. Isto é importante pois, em um jogo, um jogador trapaceiro pode arruinar a experiência de muitos jogadores. Em outras palavras, o grau de vulnerabilidade a trapaças determina a viabilidade de uma arquitetura para jogos *on-line*.

1.4 Objetivos

O objetivo geral deste trabalho consiste na criação de um novo modelo de suporte à distribuição de jogos *massively multiplayer*. Neste contexto, destacam-se como objetivos específicos:

- Descrever os temas de pesquisa envolvidos na criação do modelo, discutindo o estado da arte no escopo pesquisado (trabalhos relacionados);
- Propor um modelo de suporte a jogos distribuídos (Modelo FreeMMG) que atenda às seguintes exigências:
  - Que seja capaz de executar sobre a Internet;
  - Que dependa de servidores dedicados de baixo custo (CPU e rede);
  - Que seja escalável;
  - Que seja resistente a jogadores trapaceiros;
  - Que seja tolerante a certos tipos de falhas, como desconexão de jogadores;
  - Que garanta a consistência global da simulação;

- Implementar um protótipo do modelo (Protótipo FreeMMG);
- Propor e implementar o protótipo de um jogo *massively multiplayer* (FreeMMG Wizards) sobre o protótipo do modelo, e executar testes de escalabilidade e usabilidade sobre o protótipo combinado;

1.5 Organização do texto

Esta dissertação está organizada em cinco capítulos. O capítulo 2 apresenta os conceitos básicos em jogos distribuídos, discute outros trabalhos relacionados com o FreeMMG, e compara diferentes abordagens de suporte a jogos distribuídos. O capítulo 3 descreve o modelo FreeMMG, empregando um jogo de estratégia em tempo real como caso de uso. Ao final do capítulo são discutidos os pontos fortes e fracos do modelo, justificando oportunidades para trabalhos futuros. O capítulo 4 apresenta o jogo FreeMMG Wizards, que é o protótipo do modelo. Resultados de testes de escalabilidade do FreeMMG são apresentados e comparados com outros trabalhos. O capítulo 5 destaca as contribuições, e conclui discutindo trabalhos em andamento e possibilidades de trabalhos futuros.

2 SUPORTE A JOGOS DISTRIBUÍDOS

O objetivo principal deste capítulo é mostrar o problema dos jogos distribuídos e o estado-da-arte em soluções para este problema. Primeiramente, serão estabelecidos os conceitos básicos. Estes conceitos (jogos distribuídos, MMGs, simulação, etc.) caracterizam as áreas de aplicação em jogos distribuídos. A seguir, são analisados alguns trabalhos representativos do estado-da-arte em jogos distribuídos, incluindo modelos aplicados na prática da indústria de jogos, bem como modelos acadêmicos. O capítulo conclui identificando a lacuna no estado-da-arte do suporte a jogos distribuídos onde o modelo FreeMMG se insere.

2.1 Conceitos básicos

Nesta seção alguns conceitos básicos serão discutidos. As definições fornecidas para os termos, alguns bastante genéricos, como “simulação”, “jogo” e “tempo real”, não são absolutas e apenas caracterizam uma base para a argumentação presente no restante do texto.

2.1.1 Centralização e distribuição

Um sistema pode ser considerado ao mesmo tempo “centralizado” e “distribuído”, dependendo do ponto de vista. Por exemplo, um sistema cliente-servidor pode ser considerado tanto um exemplo de computação centralizada, quanto de computação distribuída (COULOURIS, 2000). Se o sistema for considerado como um todo, ou seja, incluindo os clientes e suas requisições, então o sistema poderia ser considerado um sistema distribuído. Por outro lado, em vários sistemas cliente-servidor, geralmente a plataforma servidora é que possui o “estado-mestre” da computação. Se for considerado apenas este aspecto (distribuição ou centralização do repositório de dados, serviço, etc.), um sistema cliente-servidor pode ser considerado como centralizado.

Para tornar a questão ainda mais complexa, muitos sistemas cliente-servidor empregam vários equipamentos servidores, que compartilham de forma transparente a responsabilidade de prover o serviço aos clientes. Neste caso, o “servidor” pode ser considerado como um sistema distribuído, mas, do ponto de vista dos clientes, o serviço é provido por uma plataforma central. Além disso, o controle administrativo deve ser levado em conta. Por exemplo, do ponto de vista administrativo um *cluster* pode ser considerado como um recurso centralizado caso o acesso ao mesmo seja controlado por uma única entidade.

2.1.2 Simulação e jogo

Neste trabalho, o conceito de "jogo" é um refinamento do conceito de "simulação" (FUJIMOTO, 2000). Simulações são aplicações que computam a variação do estado de um sistema qualquer com base em parâmetros como "tempo". Jogos são simulações de ambientes onde cada jogador (uma entidade participante) busca atingir um certo objetivo através da interação com outros jogadores e com o ambiente.

Desta forma, considera-se que um sistema de suporte para jogos possui restrições adicionais. Um sistema de suporte para jogos deve evitar que os jogadores possam alterar o estado do jogo de forma indevida, ou seja, de forma que as regras do jogo sejam burladas. Em outras palavras, o sistema deve ser tolerante à presença de jogadores dispostos a trapacear. Um sistema de suporte para jogos também deve prover um serviço de igual qualidade para todos os participantes, de forma que todos tenham a mesma chance no jogo. Estas são duas preocupações fundamentais para a área de jogos distribuídos, que geralmente não caracterizam o foco principal de trabalhos posicionados na área de simulação distribuída e interativa. Esta distinção é importante para este trabalho pois as preocupações adicionais dos programadores de jogos justificam em parte o modelo apresentado no capítulo 3.

2.1.3 Jogos distribuídos de tempo real ou multijogador

Um jogo de tempo real é um jogo onde o jogador envia comandos de forma assíncrona à passagem do tempo da simulação. Uma corrida de carros, onde o jogador é o piloto, é um exemplo de jogo de tempo real. Em contraste, um jogo baseado em turnos é um jogo onde o jogador envia comandos de forma síncrona à passagem do tempo da simulação. Um jogo de xadrez, onde a partida não avança até que o jogador da vez realize a sua jogada, é um exemplo de jogo baseado em turnos.

O foco deste trabalho compreende os jogos de tempo real distribuídos sobre Internet. O termo "jogo multi-jogador" será utilizado com este sentido. Neste contexto, os jogos multi-jogador geralmente impõem restrições quanto ao atraso máximo entre o envio de um comando pelo jogador e a efetivação do comando no jogo. O atraso máximo tolerável varia de pessoa para pessoa (PANTEL; WOLF, 2002), mas geralmente pode-se estabelecer um valor representativo. O atraso máximo tolerável também depende do tipo de jogo. Por exemplo, em um jogo de corrida de carro, se o jogador é um piloto e comanda que o seu carro vire para a esquerda, este comando deve ser reconhecido pelo simulador do jogo o mais rapidamente possível, pois se o atraso for muito grande, o carro do jogador corre o risco de sair da pista. Neste tipo de jogo (jogos de "ação"), um atraso aceitável geralmente fica em torno de 100ms. O atraso aceitável também depende do tipo de comando (PANTEL; WOLF, 2002). Mesmo em jogos de ação que exigem reflexos rápidos, alguns comandos podem possuir requisitos de tempo mais relaxados e podem ser processados com vários segundos de atrasos. Porém, quando se propõe sistemas de suporte para jogos de ação, a preocupação central geralmente é com os comandos que possuem fortes restrições de tempo.

2.1.4 Jogos maciçamente multijogador (MMGs)

Exemplos clássicos jogos maciçamente multijogador ou *massively multiplayer games* (MMGs) compreendem Ultima Online (ORIGIN, 2004), EverQuest (SONY, 2004a) e Lineage (NCSOFT, 2003). No contexto deste trabalho, MMGs são jogos distribuídos que envolvem uma grande quantidade de jogadores simultâneos interagindo, em tempo

real, em um mundo virtual de estado persistente. Porém, o termo MMG é largamente utilizado em uma série de contextos. Um jogo que suporta uma quantidade de jogadores simultâneos realmente muito grande como, por exemplo, milhares ou mesmo milhões de jogadores, e que suporta um ambiente de interação de estado persistente, pode ser considerado um "MMG" em revistas especializadas ou relatórios da indústria (IGDA, 2004). Por este critério, jogos baseados em turnos ou jogos baseados em ambientes textuais também são considerados MMGs.

No contexto deste trabalho, um MMG é um jogo com um elevado número de participantes simultâneos, que suporta interações em "tempo real" e que simula um ambiente virtual complexo, visualizado graficamente pelos jogadores. A representação gráfica, enviada aos jogadores, pode ser tanto 3D quanto 2D, desde que precise ser constantemente atualizada (dez vezes por segundo, por exemplo), ou seja, seja um ambiente gráfico atualizado em "tempo real". Em geral, este trabalho não considera um jogo baseado em eventos, ou em "turnos" de simulação muito grandes (dezenas de segundos ou minutos) como um MMG. Em outras palavras, para que um jogo seja considerado um MMG no contexto deste trabalho, os tempos de resposta às interações, entre os vários jogadores, devem ser compatíveis com os tempos de resposta esperados, em geral, pelos jogadores de jogos tipicamente classificados como *real-time* (PANTEL; WOLF, 2002).

2.1.5 Jogos de estado persistente

Jogos de estado persistente (*persistent-world games*) suportam uma grande quantidade de jogadores simultâneos interagindo em um mundo virtual persistente, assim como os MMGs, mas a interação não é, necessariamente, em tempo real. Um exemplo deste tipo de jogo é Eixo do Mal (???, 2004). Neste jogo, cada jogador possui territórios e exércitos e deve conquistar os territórios dos adversários. Para interagir com o mundo, cada jogador possui uma certa quantidade de turnos, que podem ser trocados por ações (movimentar exército, explorar território, etc). O jogo executa em um servidor *web*, que mantém o estado do mundo virtual, bem como o perfil de cada jogador, como a quantidade de exércitos, territórios, e a quantidade de turnos que cada jogador possui. A cada quatro minutos (tempo real), o servidor atualiza o perfil de cada jogador, acrescentando mais turnos.

É interessante observar que neste tipo de jogo a passagem do tempo é independente das ações dos jogadores, pois os jogadores continuarão ganhando turnos e estado do jogo continuará em evolução, mesmo que um jogador não realize nenhuma ação. Porém, no contexto deste trabalho, estes jogos não são considerados como de tempo real, pois o tamanho do passo da simulação é muito grande (ordem de minutos) e não há nenhuma restrição do tempo de resposta aos comandos do jogador. Para os MMGs, tanto o tempo de resposta aos comandos do jogador quanto o passo da simulação são da ordem de milisegundos (PANTEL; WOLF, 2002) de forma a garantir a imersão do jogador na simulação de um mundo virtual gráfico. Em contraste, os chamados "jogos de estado persistente" são freqüentemente implementados em plataformas *web*, textuais.

2.1.6 Jogos de estratégia em tempo real (RTS)

Jogos de estratégia em tempo real (abreviados para RTS ou "real-time strategy") são jogos multi-jogador onde o jogador atua como um estrategista, controlando vários personagens independentes ao mesmo tempo. Warcraft (BLIZZARD, 2004a) e Age of Empires II (ENSEMBLE, 2003) são exemplos de jogos RTS. Em ambos os exemplos (e na maioria dos jogos RTS), o jogador é um estrategista militar que busca a vitória através do

gerenciamento do seu exército, geralmente composto por vários tipos de personagens distintos como cavaleiros, arqueiros, engenheiros, catapultas, etc. Dois aspectos importantes devem ser ressaltadas sobre os jogos RTS. Primeiro, os jogos RTS são mais tolerantes ao atraso entre o envio e o processamento dos comandos do jogador. Um atraso constante de 200ms passa praticamente desapercebido aos jogadores (BETTNER; TERRANO, 2001). E segundo, jogos RTS distribuídos são difíceis de serem implementados utilizando-se uma topologia cliente-servidor (BETTNER; TERRANO, 2001; CECIN, 2003).

2.1.7 Jogos baseados em avatares

Neste texto, iremos classificar certos jogos como "baseados em avatares". O termo *avatar* significa algo como *encarnação*. Nos jogos baseados em avatares, cada jogador controla apenas uma entidade (a sua *encarnação*) de cada vez. Por exemplo, em um MMG como EverQuest, cada jogador controla um personagem humanóide, comandando-o para, por exemplo, se movimentar em uma certa direção. O oposto dos jogos baseados em avatares são os jogos RTS, onde cada jogador pode comandar não apenas um, mas vários personagens ou objetos de cada vez.

Apesar de não ser comum, esta distinção é importante no contexto deste trabalho, pois a grande maioria dos MMGs em funcionamento atualmente são jogos baseados em avatares. Como será visto a seguir, o modelo cliente-servidor, empregado pela grande maioria dos jogos *on-line* e pela quase totalidade dos MMGs em funcionamento, não é adequado para jogos de estratégia em tempo real. Uma das vantagens do modelo proposto neste trabalho (FreeMMG) é o suporte a jogos Massively Multiplayer de estratégia em tempo real.

2.1.8 Ambientes virtuais colaborativos (CVEs)

Os "ambientes virtuais colaborativos" (CVEs ou *collaborative virtual environments*) são, como aplicações, uma tecnologia de telecomunicação capaz de reunir participantes fisicamente distantes em um ambiente simulado, onde um contexto compartilhado, social e espacial, é estabelecido entre eles (ROBERTS; WOLFF, 2004). Os CVEs podem ser considerados uma especialização dos "ambientes virtuais em rede" (NVEs ou *networked virtual environments*), apesar destes termos serem freqüentemente utilizados como sinônimos. Exemplos de CVEs incluem NPSNET (MACEDONIA et al., 1994), DIVE (CARLSSON; HAGSAND, 1993), PaRADE (ROBERTS, 1996), MASSIVE (GREENHALGH; PURBRICK; SNOWDON, 2000) e SOLIPSIS (KELLER; SIMON, 2002).

A pesquisa no contexto de CVEs e NVEs herda, em grande parte, os resultados da pesquisa em simulação militar (SINGHAL; ZYDA, 1999) obtidos no desenvolvimento de sistemas de simulação interativa e distribuída como DIS (TARR; JACOBS, 2002) e SIMNET (MILLER; THORPE, 1995). Por sua vez, a pesquisa no contexto de jogos distribuídos, em especial jogos maciçamente multijogador ou maciçamente distribuídos, herda em grande parte os resultados obtidos no contexto da pesquisa em CVEs. Apesar de os CVEs antecederem a explosão de popularidade dos jogos multijogador e da Internet comercial, muitos problemas e soluções são compartilhados entre CVEs e MMGs, relativos a, por exemplo, gerência da consistência, escalabilidade e responsividade (ROBERTS; WOLFF, 2004).

Apesar das fortes similaridades, existe pelo menos uma diferença importante entre CVEs e MMGs. Os CVEs suportam aplicações cooperativas, onde a prevenção de "trapaças" (KARLSSON; FEIJÓ, 2004; CRONIN et al., 2003; BAUGHMAN; LEVINE, 20., 2002), ou seja, ações ilegais realizadas por jogadores maliciosos, não é uma das principais

preocupações. Os MMGs, por outro lado, são sistemas de suporte a jogos competitivos via Internet. Como os MMGs simulam ambientes virtuais de estado persistente, os efeitos das alterações ilegais podem afetar o jogo de forma irreversível, arruinando a reputação do jogo perante milhares de jogadores. Portanto, é muito importante que os seus sistemas de suporte possuam mecanismos para prevenção, detecção ou correção de trapaças (KARLSSON; FEIJÓ, 2004; CRONIN et al., 2003; BAUGHMAN; LEVINE, 20., 2002; CUNNIN, 2000). Esta única diferença é suficiente para que muitos sistemas de suporte a CVEs não sejam aplicáveis, pelo menos diretamente, para o suporte a MMGs.

2.2 Jogos cliente-servidor (centralizados)

O paradigma cliente-servidor é, sem dúvida, o mais utilizado atualmente na implementação de jogos distribuídos ("multi-jogador", em tempo real) para a Internet. Exemplos comerciais destes jogos incluem Doom, Quake e Half-Life, todos jogos de ação 3D em tempo real. A seguir, serão discutidas as vantagens e desvantagens do paradigma cliente-servidor quando aplicado a jogos, bem como alguns detalhes de implementação de protocolos cliente-servidor para jogos de tempo real.

2.2.1 Utilizando cliente-servidor em jogos distribuídos de tempo real

Em qualquer jogo cliente-servidor, os jogadores interagem com o ambiente e com os outros jogadores através de um programa cliente. Cada cliente possui uma conexão de rede apenas com o servidor. O servidor é responsável por receber a informação de cada cliente e repassar atualizações para os mesmos. Existem várias maneiras diferentes de implementar o protocolo entre os clientes e o servidor. Algumas são mais seguras, outras mais eficientes em determinados casos. A seguir, serão apresentadas duas técnicas básicas de implementação para o protocolo de um jogo cliente-servidor de tempo real:

- **Clientes enviam atualizações:** Uma técnica possível, que é empregada no jogo Half-Life (VALVE, 2004), é deixar os clientes arbitrarem sobre parte do estado do jogo. Por exemplo, um processo cliente pode atualizar periodicamente a posição atual do personagem que o jogador local controla e informar o servidor da nova posição do personagem. O servidor então "corrige" a sua cópia do personagem com base nesta informação, quando for recebida. É evidente que isto facilita o desenvolvimento de programas clientes "trapaceiros", pois o cliente pode, a princípio, "posicionar" o personagem do jogador aonde bem entender. Mesmo que o servidor realize testes para detectar alguns tipos óbvios de manipulação (por exemplo, verificando se um personagem não percorreu vários "quilômetros virtuais" em alguns poucos segundos), geralmente fica difícil para o servidor diferenciar entre uma informação correta e uma manipulação cuidadosa, que altera levemente a posição de um personagem para desviar de um projétil, por exemplo.

- **Clientes enviam comandos:** Outra técnica possível, que é empregada nos jogos Quake (ID SOFTWARE, 2004) e Outgun (CECIN et al., 2004), consiste em não confiar em atualizações dos clientes, fazendo com que o servidor aceite apenas "requisições" dos clientes. Ou seja, o servidor não consulta os clientes quanto ao estado atual de algum objeto de jogo ou à ordem em que certos eventos ocorrem. Neste contexto, o cliente envia ao servidor apenas requisições, que podem ser atendidas ou não. Estas requisições podem ser, por exemplo, comandos para movimentar um personagem ou disparar um projétil. Cada vez que um jogador emite um comando

relevante ao jogo (evento de mouse, teclado, joystick, ...) este é empacotado pelo cliente e enviado ao servidor.

Independente da técnica utilizada, o servidor é o responsável por manter o estado do jogo atualizado conforme a passagem do tempo de jogo. Como se trata de uma simulação de tempo real, é necessário que o estado do jogo seja recalculado com freqüência. Em geral, o servidor é configurado para atualizar o estado utilizando um "passo" fixo de tamanho relativamente pequeno, por exemplo, de 50ms. O "passo" da simulação constitui em calcular o novo estado do jogo partindo de regras de atualização (por exemplo, equações de movimento para atualizar posição de objetos baseado em velocidade) e da informação recebida dos clientes desde o último passo (atualizações e/ou comandos).

O servidor também é programado para enviar, periodicamente, mensagens de atualização para todos os clientes. Isto é necessário para que um cliente consiga visualizar localmente o que os outros clientes estão "fazendo" no jogo, já que não há conexão de rede entre os clientes. Estas mensagens permitem aos clientes atualizarem as suas cópias locais do estado do jogo, que são utilizadas para alimentar as rotinas que desenham a tela do jogador, emitem sons, etc.

Como o recurso de rede, tanto dos clientes, quanto do servidor, é precioso e geralmente limitado, uma mensagem de atualização geralmente não contém o estado completo do jogo. Tipicamente se envia a um cliente apenas atualizações referentes aos objetos de jogo que são do seu "interesse" (MORSE, 1996). Imaginando um "mundo virtual" de jogo em 3D, os objetos de interesse são, por exemplo, objetos que estão diretamente na linha de visão do jogador ou que estão potencialmente visíveis. Há também outras maneiras de reduzir o tamanho de uma mensagem de atualização, como filtragem (MORSE, 1996; MORSE et al., 1999; VAN HOOK; CALVIN, 1994) por prioridade (importância do objeto para a tomada de decisão do jogador), por distância (objetos "distantes" podem necessitar de atualizações de posição menos freqüentes ou de menor resolução), etc. Além da limitação de banda, reduzir o tamanho das mensagens aumenta as chances destas serem rapidamente transmitidas pela Internet (BASS, 1997), o que é muito importante para a implementação de jogos de ação.

2.2.2 Exemplo: o jogo Outgun

A seguir será detalhado o protocolo utilizado no jogo Outgun (CECIN et al., 2004), um jogo distribuído, cliente-servidor, de ação simples em um ambiente 2D, projetado para a Internet. Neste jogo dois times de jogadores controlam personagens representados por círculos que disparam projéteis uns contra os outros enquanto tentam capturar a bandeira do time adversário. Cada partida se desenvolve em um "mapa" (o "mundo virtual" do jogo) que é composto de um número variável de salas. Cada jogador só enxerga a posição exata dos jogadores que estão mesma sala. O jogo oferece também um "radar", localizado no canto superior direito da tela, que exibe todas as salas do mapa, a posição global do jogador, a posição dos jogadores aliados, e a posição de outros objetos (bandeiras e jogadores inimigos) que estejam no campo de visão de pelo menos um aliado. A figura 2.1 exibe uma tela do jogo.

Historicamente, os jogos de ação como Doom, Quake e Half-Life tiveram seus protocolos desenvolvidos para modems com capacidade de transmissão entre 14KBPS e 56KBPS, sendo 2Kbytes/s a 5Kbytes/s (por conexão cliente-servidor) uma linha-guia geral para um protocolo destes. O protocolo do jogo, via de regra, é implementado sobre UDP, pois as mensagens de atualização enviadas pelo servidor, em geral, não precisam ser

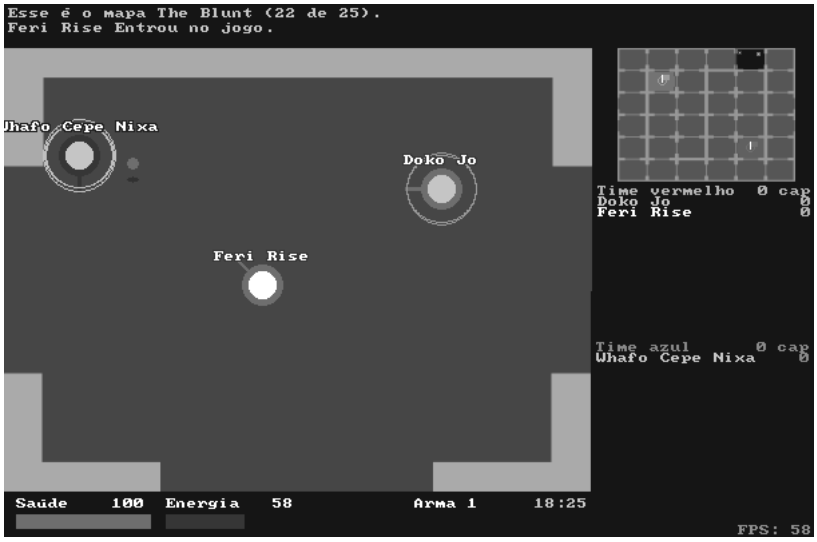


Figura 2.1: Captura de tela do jogo Outgun

retransmitidas quando perdidas, pois cada atualização enviada torna as atualizações anteriores obsoletas. Para satisfazer as necessidades de garantia de entrega e ordenamento do protocolo do jogo, pode-se utilizar outra conexão TCP em conjunto com a conexão UDP, ou, o que é geralmente o caso, é utilizado um protocolo que suporta retransmissão e ordenamento seletivos sobre UDP (por exemplo, RTP (SCHULZRINNE et al., 1996) ou outro desenvolvido especificamente para a aplicação).

O jogo Outgun foi desenvolvido com o objetivo de utilizar poucos recursos de rede. A versão 0.5.0 do jogo utiliza 0,5Kbytes/s entre o cliente e o servidor na maioria das situações, mas pode chegar a picos de 2Kbytes/s por jogador durante as batalhas mais acirradas. O jogo foi projetado para suportar até 32 clientes simultâneos (até 16 jogadores em cada um dos dois times de jogadores). Um protocolo específico de retransmissão e ordenamento seletivos foi implementado sobre UDP e foi batizado de RUDP (Reliable UDP). A tabela 2.1 exibe o formato de um pacote RUDP. O pacote consiste de quatro partes básicas: um cabeçalho que inclui o identificador do pacote, uma lista de mensagens de entrega garantida (“mensagens confiáveis”) que estão sendo transmitidas (ou retransmitidas) neste pacote, uma lista de “acks” que confirmam os identificadores dos pacotes recebidos, e um bloco de entrega não garantida (“mensagem não-confiável”). Cada mensagem confiável enviada por um lado da conexão recebe um identificador único e sempre crescente. Quando um pacote com mensagens confiáveis é recebido, o identificador deste pacote é incluído na lista de “acks” do próximo pacote enviado. Desta forma são detectadas, indiretamente, as mensagens confiáveis que já foram recebidas. Enquanto uma mensagem confiável não é confirmada, ela é constantemente retransmitida. Isto é feito para minimizar o atraso na entrega de mensagens, e não caracteriza um problema visto que o protocolo foi projetado para aplicações que enviam mensagens confiáveis muito pequenas. As mensagens confiáveis em Outgun tem tipicamente menos de 50 bytes, e a maioria não ultrapassa 10 bytes.

Utilizando RUDP, foi implementado o protocolo do jogo em si, o protocolo Outgun. O protocolo Outgun determina que os clientes enviam ao servidor apenas requisições. Desta forma, é dificultado o surgimento de programas clientes com suporte a trapaças, pois o cliente pode apenas enviar “pedidos” ao servidor, que tem autonomia para decidir se a

Tabela 2.1: Estrutura do pacote RUDP utilizado pelo jogo Outgun

Tipo	**Descrição**
int (32 bits)	"Id" do pacote
short (16 bits)	Nº de confirmações de pacotes (nack)
int [nack]	lista de "Id"s de pacotes confirmados
byte (8 bits)	Nº de mensagens confiáveis no pacote (nmsg)
estrutura [nmsg]:	Lista de mensagens confiáveis
– int	"Id" da mensagem
– short	Tamanho da mensagem (msize)
– byte [msize]	Dados da mensagem
byte [...]	Dados não-confiáveis (qualquer tamanho)

"jogada" enviada pelo jogador é válida ou não. Neste sentido, o cliente Outgun envia, no bloco "não confiável" do pacote, apenas um único byte onde é codificado o estado atual das teclas do jogador (tabela 2.2). As teclas determinam a direção e velocidade em que o jogador quer se movimentar, e também determina se o jogador quer atirar ou não.

Tabela 2.2: Estrutura do bloco não-confiável enviado pelo cliente do Outgun

Tipo	**Descrição**
byte (8 bits):	Estado do dispositivo de entrada do jogador
– bit 0	"mover-se para a esquerda" (left)
– bit 1	"mover-se para a direita" (right)
– bit 2	"mover-se para cima" (up)
– bit 3	"mover-se para baixo" (down)
– bit 4	"esquivar" (strafe)
– bit 5	"correr" (run)

O servidor Outgun atualiza o estado da simulação a cada 100ms. A atualização é feita utilizando o estado anterior do jogo e o último estado conhecido das teclas dos jogadores. Após recalcular o estado, o servidor envia um pacote RUDP de atualização para cada cliente. Áreas de interesse são contempladas à medida que cada cliente é atualizado apenas sobre os objetos (personagens, projéteis, etc) localizados na sala em que o personagem do jogador em questão se encontra. O servidor envia os disparos (criação dos projéteis dos jogadores) como mensagens confiáveis.

Como mensagem "não confiável", o servidor envia em cada pacote um registro de atualização para cada avatar que está na área de interesse do cliente em questão. A tabela 2.3 mostra a estrutura do bloco "não confiável" enviado pelo servidor aos clientes, que inclui a lista de registros de atualização de avatares. É importante que estas atualizações de posição e direção do personagem não sejam retransmitidas em caso de perda, pois estas informações se tornam obsoletas muito rapidamente (pacotes com informação mais atual são enviados a cada 100ms) e a retransmissão da mesma apenas usaria mais recursos de rede, sem nenhum ganho adicional.

Uma observação interessante é a de que o tamanho do pacote enviado do servidor para um cliente é diretamente proporcional ao número de "avatares" na linha de visão deste jogador, e não ao número total de jogadores conectados no servidor. Esta mesma

Tabela 2.3: Estrutura do bloco não-confiável enviado pelo servidor do Outgun

Tipo	**Descrição**
int (32 bits)	Tempo ou relógio de simulação (timestamp)
int	Jogadores presentes (0 a 31, 1 bit por jogador, bit ligado significa que o respectivo jogador está presente no jogo)
byte (8 bits):	Miscelânea
– bit 0	Bit extra para saúde do jogador (+256)
– bit 1	Bit extra para energia do jogador (+256)
– bit 2	Skip frame (fim do pacote)
– bits 3..7	Identificador do jogador (0..31)
byte [2]	Coordenada (x,y) da sala do jogador
int	Jogadores presentes na sala (0 a 31) (pfield) pcount = Quantidade de bits “1” em (pfield)
estrutura [pcount]:	Lista de atualização de jogadores (ordem crescente)
– byte [3]	Coordenada (x,y) do jogador (na sala)
– byte [2]	Vetor de velocidade (sx,sy) do jogador
– byte:	Modificadores do jogador
— bit 0	morto?
— bit 1	possui “deathbringer”?
— bit 2	atingido por “deathbringer”?
— bit 3	possui “escudo”?
— bit 4	possui “turbo”?
— bit 5	possui “quad damage”?
– byte:	Comandos do jogador
— bit 0	movendo-se para a esquerda
— bit 1	movendo-se para a direita
— bit 2	movendo-se para cima
— bit 3	movendo-se para baixo
— bit 4	correndo
— bits 5..7	direção do jogador (0..7)
– byte	Grau de opacidade do jogador (alpha)
short (16 bits)	Inimigos visíveis (0 a 15, 1 bit por jogador)
byte	Índice do jogador atualizado no mini-mapa (jogmm)
byte [2]	Coordenada (x,y) do jogador[jogmm] no mini-mapa
byte	Saúde do jogador (8 bits menos significativos)
byte	Energia do jogador (8 bits menos significativos)
short	Latência de rede do jogador[timestamp módulo 32]

observação vale para os jogos “massively multiplayer”, cliente-servidor. Mesmo que existam milhares de jogadores conectados a um servidor de MMG, cada cliente tipicamente só será atualizado sobre os avatares ou objetos dinâmicos do jogo que sejam do seu interesse, ou seja, que estejam próximos do avatar controlado por aquele cliente.

Em resumo, Outgun é um exemplo de jogo distribuído, cliente-servidor, de ação em tempo real. É um jogo baseado em “avatares”, ou seja, onde cada jogador controla diretamente um objeto dinâmico (o “avatar” ou personagem) que interage diretamente com os outros objetos dinâmicos do jogo (outros avatares, projéteis, etc). Neste tipo de jogo, é muito importante que os jogadores tenham um tempo de resposta o mínimo possível aos seus comandos, ou seja, o atraso entre o envio de um pacote de comando (por exemplo, “mover personagem para frente”) e o recebimento de um pacote de atualização que é influenciado pelo comando (por exemplo, atualização onde o avatar começou a mover-se para frente). O protocolo foi desenvolvido especificamente para atender a este requisito. O protocolo também foi desenvolvido para oferecer um bom nível de proteção contra trapaças. Por exemplo, o cliente não decide a posição final do seu avatar, apenas envia requisições de movimentação para o servidor.

2.2.3 Cliente-servidor aplicado para jogos maciçamente multijogador

Teoricamente, um jogo distribuído pode utilizar a técnica de simulação centralizada do paradigma cliente-servidor para atender a milhares de clientes simultâneos. Para que isto ocorra, o servidor deve possuir poder de processamento (CPU e, indiretamente, memória) e taxa de transmissão (rede) suficientes. Os primeiros jogos comerciais considerados “massively multiplayer” para a Internet como Meridian 59 executavam o jogo a partir de um único servidor físico (uma única máquina).

Porém, na prática, pode ser difícil (IGDA, 2004) hospedar em uma única máquina todo o poder de processamento (CPU e memória) necessário para executar a simulação, em tempo real, de um “mundo virtual” habitado por milhares ou milhões de jogadores. Uma solução para este problema, comumente empregada nos MMGs comerciais como EverQuest e PlanetSide, consiste em particionar o mundo virtual e fazer com que cada partição seja atualizada por uma máquina individual, por exemplo, um nodo de um *cluster*. Como sempre haverão dependências de dados entre as diferentes partições do mundo virtual, será sempre necessário sincronizar as tarefas dos nodos responsáveis pelas partições através de troca de mensagens. Por isso, via de regra emprega-se uma rede confiável, de boa velocidade e largura de banda entre os nodos do *cluster*, o que não deixa de ser comum na maioria das aplicações de *clusters*. Existem várias maneiras de particionar um mundo virtual e várias maneiras de mapear as partições nos nodos disponíveis. Este problema é discutido em detalhe por Cunnin (CUNNIN, 2000), que também propõe novos algoritmos de balanceamento de partições em nodos e sincronização entre os nodos de um *cluster* que atua como um servidor de MMG.

Mesmo utilizando-se *clusters* para atenuar o problema do custo de processamento de um servidor MMG, ainda resta o problema do consumo de rede. O servidor MMG, seja uma única máquina ou um *cluster*, necessita de uma conexão com a Internet de baixa latência e alta largura de banda, de onde irá se comunicar em tempo real com as centenas, milhares ou milhões de clientes. Tomando como exemplo o jogo Outgun descrito na seção anterior, onde cada jogador pode utilizar até 2Kbytes/s por cliente, um servidor de Outgun que atendesse a 5.000 jogadores, por exemplo, consumiria aproximadamente 80 megabits/s de rede. Esta extrapolação é válida pois um protocolo de MMG cliente-servidor é, em essência, idêntico ao protocolo cliente-servidor para jogos de tempo real

distribuídos em pequena escala. Na verdade, o protocolo de um MMG cliente-servidor tende a ser mais pesado, pois o número de objetos dinâmicos que podem ser do interesse de um cliente MMG, a princípio, tende a ser maior. Por exemplo, no MMG PlanetSide, é possível que mais de uma centena de personagens, que movimentam-se dinamicamente em um ambiente 3D, estejam na linha de visão de um jogador. Outro dado que reforça a extrapolação apresentada acima é que o servidor do MMG Ultima Online gasta aproximadamente 2Kbytes/s por cliente servido (**?**).

Apesar dos custos de processamento e de rede da simulação centralizada de um "mundo virtual" de MMG, o paradigma cliente-servidor oferece boas garantias em relação à segurança do jogo. Por exemplo, a ocorrência de "trapaças" por parte dos jogadores é dificultada, dependendo apenas do grau de robustez do protocolo empregado entre cliente e servidor, aspecto discutido nas seções anteriores. Outra vantagem é a maior simplicidade na implementação visto que, em geral, sistemas descentralizados são mais complexos que os seus equivalentes cliente-servidor (MCGREGOR et al., 2003; SCHODER; FISCHBACH, 2003; KUBIATOWICS, 2003).

2.3 A proposta Grid: Butterfly.net

O projeto Butterfly.net (BUTTERFLY.NET, 2003) oferece uma solução de computação em grade (*Grid computing*) (FOSTER, 2002) para resolver o problema da hospedagem de servidores de MMGs. O propósito geral do Grid da Butterfly.net é substituir os servidores (*clusters*) utilizados pelas empresas que servem os MMGs. Um dos argumentos técnicos principais a favor da migração de *clusters* para o Grid refere-se ao problema do dimensionamento destes *clusters*, visto que não é possível prever exatamente qual será o sucesso do jogo, ou seja, o número de clientes que este jogo terá. Se um MMG fracassar, a empresa pode acabar com poder computacional de sobra, e se um MMG fizer muito sucesso, podem faltar máquinas e alguns clientes que compraram o jogo podem ficar de fora nos horários de pico ou a qualidade do serviço pode decair.

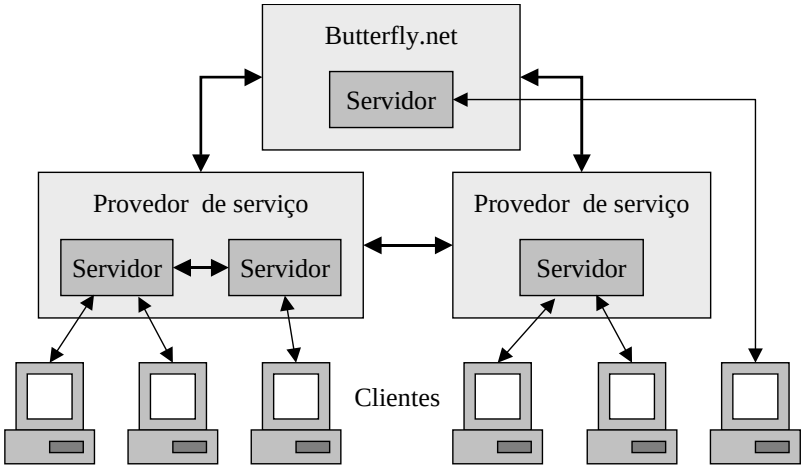


Figura 2.2: Uma visão simplificada da plataforma "Butterfly Grid" (gerenciador central, provedores de serviço e clientes)

A figura 2.2 mostra uma visão simplificada do Butterfly Grid. O Grid possui um "centro" onde ficam hospedadas as máquinas da Butterfly.net. Neste centro localizam-se servidores mestres de dados e outros serviços essenciais, além dos servidores de jogo

propriamente ditos. O servidor de jogo é o que se conecta diretamente com os clientes dos vários MMGs que são servidos pelo Grid e executa o protocolo cliente-servidor de jogo discutido anteriormente. Conectados ao "centro" do Grid, e entre si de forma dinâmica, estão servidores de jogo hospedados por provedores de serviço associados, por exemplo, provedores de acesso à Internet ou provedores de jogos e conteúdo.

Existe um problema de escalonamento envolvido (REAL, 2004), como em toda aplicação de computação em grade. O problema consiste em alocar os servidores físicos (máquinas) do Grid para hospedar um nodo virtual de um servidor distribuído de cada MMG hospedado pelo Grid. Para cada máquina do Grid, o escalonador deve decidir qual servidor de MMG deve ser servido por aquela máquina, ou seja, cada MMG necessita de vários servidores "lógicos" alocados em vários locais (físicos) distintos. Se um jogo faz sucesso e sua base de jogadores aumenta, o Grid irá alocar mais máquinas para servir este jogo. Desta forma, a adequação do poder de processamento (CPU) no "lado servidor" de um MMG cliente-servidor é resolvido.

Porém, resta verificar se o problema do consumo de rede entre os nodos servidores e os clientes ainda persiste. Apesar de a solução do Grid ser mais sofisticada do que a utilização de servidor em *cluster* próprio, o protocolo de rede entre servidor de jogo e cliente continua, a princípio, sendo o mesmo, que foi descrito anteriormente. Isto significa que o consumo de rede total entre clientes e servidores deve continuar na mesma ordem de grandeza. Uma maneira de resolver este problema, no contexto do Butterfly Grid, seria buscar alocar servidores de jogo em máquinas próximas aos clientes. Por exemplo, alocar um servidor de jogo em um nodo do Grid localizado no provedor de acesso à Internet que é utilizado pelo cliente do jogo. Mesmo que não seja sempre possível realizar esta alocação ótima, basta garantir que grande parte dos servidores estejam localizados em provedores de serviço. Desta forma ao menos, o gargalo do servidor é distribuído em várias localidades.

Uma das característica importantes do Butterfly Grid é a garantia oferecida de uma alta qualidade de serviço. Esta garantia inclui uma transparência total do mecanismo de partição do "mundo virtual" do jogo. Em outras palavras, mesmo que dois "pedaços" do mundo virtual estejam hospedados em máquinas distintas no Grid, e até distantes entre si, para os jogadores esta divisão deve ser imperceptível, mesmo que os próprios jogadores estejam distantes entre si e das máquinas servidoras que estão escalonadas para aquelas partições do mundo do jogo. Como discutido anteriormente, uma solução de *cluster* em rede local para um servidor MMG é viável visto que neste contexto é fácil instalar uma rede de alta velocidade e baixa latência, suficiente para que os servidores se sincronizem durante a simulação das partições do mundo do jogo. Isto implica, a princípio, que a Butterfly.net utiliza redes de longa distância que possuem qualidade similar para interligar as máquinas do Grid.

Em resumo, a tecnologia de Grid aplicada para MMGs cliente-servidor apresenta algumas vantagens sobre a utilização de *clusters* de servidores. O Butterfly Grid emprega uma infra-estrutura de alta qualidade (nodos e rede). Os nodos servidores, que são alocados dinamicamente, são posicionados nos provedores de serviço, próximo aos clientes. Apesar da infra-estrutura ser razoavelmente distribuída, o paradigma da aplicação ainda é jogo cliente-servidor, onde os clientes fazem apenas requisições e os servidores atuam como sincronizadores das ações dos clientes, garantindo a princípio uma proteção contra clientes potencialmente desonestos ("trapaceiros").

2.4 Jogos *peer-to-peer* de ação em pequena escala

É possível implementar jogos distribuídos de "ação", em tempo real, de uma forma mais descentralizada do que a empregada pelo protocolo cliente-servidor. Por exemplo, pode-se utilizar uma abordagem *peer-to-peer*, onde os nodos dos jogadores se comunicam entre si diretamente sem depender de um nodo sincronizador central (GAUTIER; DIOT, 1998). A figura 2.3 ilustra uma visão abstrata de uma sessão de jogo *peer-to-peer*, comparada com uma sessão de jogo cliente-servidor. Pode-se observar que há um problema de escalabilidade, pois cada participante do jogo deve possuir uma conexão com cada outro participante, aumentando o custo de rede de forma linear em todos os nodos, para cada participante adicional. Isto pode não ser problemático para um pequeno número de participantes (jogos em "pequena escala" com 8, 16, ou 32 participantes), mas torna-se inviável para aplicação em jogos maciçamente multijogador, especialmente se a rede utilizada para a troca de mensagens não possuir suporte a multicast como, por exemplo, a Internet em geral.

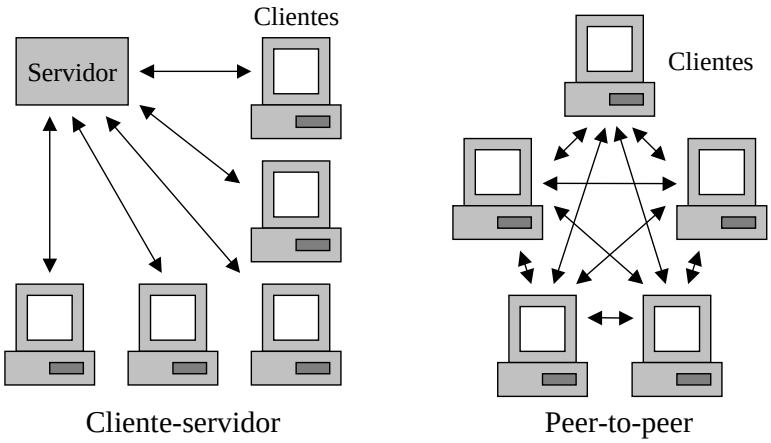


Figura 2.3: Uma sessão de jogo interativo, em cliente-servidor e *peer-to-peer*

O funcionamento básico de um jogo *peer-to-peer* determina que:

- Não há um servidor central para manter o estado global do jogo;
- Cada cliente deve enviar atualizações para cada outro cliente;
- O estado do jogo em cada cliente é computado com base nas atualizações recebidas de outros clientes;
- Clientes podem entrar e sair do jogo dinamicamente após o início do mesmo, se o protocolo do jogo permitir.

Assim como no paradigma cliente-servidor, o protocolo específico empregado por um jogo *peer-to-peer* pode ser implementado utilizando diferentes técnicas, dependendo de fatores como o tipo de jogo que se pretende suportar e as características da rede (multicast ou apenas unicast). Como exemplo, considere-se um jogo de ação baseado em "avatares", ou seja, um jogo onde cada jogador controla um único personagem em tempo real. Um jogo que se encaixa neste perfil é MiMaze (GAUTIER; DIOT, 1998), um jogo de ação *peer-to-peer* projetado para um *backbone* experimental da Internet com suporte a multicast, o MBone (ALMEROTH, 2000; HANDLEY, 1997; ALMEROTH; AMMAR,

1997). Em MiMaze, cada jogador controla um personagem dentro de um labirinto 3D. Cada cliente MiMaze envia, periodicamente, uma mensagem de atualização que contém, entre outras informações, a posição atual do personagem controlado pelo jogador. Como a atualização de posição do personagem precisa ser enviada para vários (potencialmente todos) os jogadores ao mesmo tempo, o suporte a multicast da rede pode ser explorado, reduzindo de forma significativa a quantidade total de mensagens enviadas.

Porém, como discutido anteriormente, deixar um nodo decidir a posição do seu personagem em um jogo de ação caracteriza um ponto fraco do protocolo. A vantagem de enviar atualizações de posição é que estas podem ser perdidas sem necessidade de retransmissão, simplificando o protocolo e as tarefas de manter o jogo responsivo, minimizando o efeito do atraso da rede. Se cada cliente enviasse apenas comandos como "mover personagem para frente", com o objetivo de diminuir a vulnerabilidade do protocolo à trapaças, estes teriam de ser enviadas de forma confiável, com retransmissão em caso de perda. Isto seria necessário pois, como não há um nodo central para sincronizar os comandos ou "eventos", cada cliente necessitaria da lista completa destes eventos para poder manter a sua cópia do jogo sincronizada com as demais. Esta abordagem (*peer-to-peer* com envio de comandos) será discutida em detalhe na seção seguinte.

Em resumo, jogos de ação em tempo real podem ser implementados como sistemas *peer-to-peer*. Uma vantagem dos jogos *peer-to-peer* é não necessitar de um servidor dedicado para sincronizar os eventos dos jogadores. Outra vantagem bastante significativa é o menor atraso experimentado pelos jogadores, pois os eventos gerados pelos jogadores são enviados diretamente de um cliente a outro, ao passo que em um jogo cliente-servidor, os eventos são primeiramente enviados ao servidor para depois serem repassados aos clientes. Outra importante característica do jogo *peer-to-peer* é que a escalabilidade ou quantidade de jogadores simultâneos suportados é limitada em todos os nodos, pois cada nodo, a cada "passo" da simulação, deve enviar e receber mensagem para todos os outros nodos. Um jogo cliente-servidor é limitado, a princípio, apenas pelo servidor, que é o único nodo cuja demanda cresce à medida em que mais jogadores são adicionados. Isto permite que a escalabilidade do sistema seja proporcional ao investimento de recursos no servidor, enquanto os clientes podem ser de baixa capacidade. Se a rede oferece suporte a multicast, o gargalo de escalabilidade do jogo *peer-to-peer* pode ser significativamente reduzido.

2.5 Jogos *peer-to-peer* de estratégia em pequena escala

Esta seção descreve um protocolo de simulação interativa baseado em *peer-to-peer* onde cada participante possui uma cópia completa do estado da simulação que deve estar sempre sincronizada com a cópia de todos os outros participantes. Este protocolo, será referido ao longo do texto como "simulação *peer-to-peer* replicada", mas também é conhecido na literatura como *lockstep* (GAUTHIERDICKEY; ZAPPALA; LO, 2004; CRONIN et al., 2003) ou *stop-and-wait*. Este protocolo é utilizado pelo modelo FreeMMG, que será apresentado no capítulo 3.

Os protocolos de jogo discutidos anteriormente, tanto cliente-servidor quanto *peer-to-peer*, resolvem o problema de "jogos de ação em tempo real baseados em avatares". De "ação em tempo real", enfatizando a necessidade de priorizar um baixo tempo de resposta aos comandos dos jogadores, e "baseados em avatares", ou seja, jogadores controlando ou influenciando em tempo real, diretamente, apenas um objeto dinâmico (o personagem ou "avatar"). Os protocolos descritos anteriormente não são os mais adequados para lidar

com jogos onde cada jogador controla diretamente, em tempo real, centenas de objetos dinâmicos.

Um tipo de jogo que permite aos jogadores controlar centenas de objetos dinâmicos simultaneamente é o jogo de estratégia militar. Exemplos destes jogos incluem Age of Empires II (ENSEMBLE, 2003) e Rise of Nations (BIG HUGE GAMES, 2004). Os protocolos baseados em atualizações de *status* ou posição de objetos dinâmicos não são muito adequados para estes jogos pois a quantidade de objetos em potencial que necessitam de atualizações constantes pode ser muito grande. Por exemplo, em um MMG cliente-servidor como PlanetSide (SONY, 2004b), um jogador pode, muito raramente, visualizar batalhas envolvendo uma centena de jogadores, necessitando neste caso de até cem "registros" de atualização de objeto em cada pacote de atualização enviado pelo servidor, pois cada jogador controla um objeto dinâmico (desconsiderando mensagens de disparo de projéteis, etc). Em um jogo de estratégia como Age of Empires II, que suporta até 8 jogadores simultâneos, um encontro rotineiro entre apenas dois jogadores pode envolver até 400 objetos dinâmicos (200 de cada jogador). Ou seja, se Age of Empires II fosse implementado utilizando o mesmo protocolo de PlanetSide, e considerando o exemplo anterior, o servidor utilizaria até quatro vezes mais largura de banda (100 contra 400 objetos dinâmicos) para atualizar cinqüenta vezes menos jogadores (100 contra 2 jogadores). No exemplo os clientes também utilizariam até quatro vezes mais largura de banda.

Se considerarmos uma rede sem suporte a multicast, o protocolo *peer-to-peer* para jogos de ação descrito na seção anterior é igualmente inadequado. O protocolo especifica que cada cliente atualiza os objetos dinâmicos que controla e envia periodicamente uma mensagem de atualização para cada outro cliente. Se o cliente controla não apenas um, mas, por exemplo, cem objetos, o tamanho da mensagem a largura de banda utilizada pelo cliente são, a princípio, multiplicados por cem.

A idéia central da solução para o jogo de estratégia consiste em utilizar uma codificação diferente para o protocolo de jogo. O protocolo até aqui discutido determina que para cada objeto dinâmico deve haver um nodo "responsável" que calcula a posição do objeto e atualiza com freqüência os outros nodos, enviando a informação em pacotes sem entrega garantida. Isto faz com que o tráfego de rede seja diretamente proporcional à quantidade de objetos dinâmicos do jogo. O protocolo utilizado por Age of Empires e descrito por Bettner e Terrano (BETTNER; TERRANO, 2001) é diferente. A topologia de rede utilizada é *peer-to-peer*, com uma conexão entre cada cliente participante, como descrito anteriormente para jogos de ação, mas com a diferença de que há garantia de entrega de mensagens.

O cliente de Age of Empires utiliza estas conexões "confiáveis" para enviar listas de comandos e um comando pode ser, por exemplo, uma requisição para mover um determinado exército para certas coordenadas. É importante destacar que o tamanho (em bits) de um comando não é proporcional ao número de objetos dinâmicos envolvidos: o comando do exemplo anterior não varia de tamanho de acordo com o número de objetos que compõe o exército que está sendo comandado. As listas de comandos são despachadas periodicamente, por exemplo, a cada 100ms. A cada envio, um contador inteiro de tempo de simulação local é incrementado e associado à lista de comandos, caracterizando o "timestamp" da lista. Como a entrega de mensagens é garantida, cada cliente recebe todas as listas de comandos enviadas por cada outro cliente.

A figura 2.4 ilustra um cenário envolvendo dois simuladores (réplicas). No início de uma partida, todos os clientes participantes concordam em um estado inicial e inicializam

seus contadores de tempo de simulação local em zero. A partir desta inicialização, cada cliente fica em laço aguardando o recebimento das listas de comandos de todos os jogadores cujo "timestamp" é igual ao seu tempo de simulação local. Quando isto ocorre em um cliente, este tem informação suficiente para atualizar sua cópia da simulação, como mostra o exemplo da figura 2.4. Como todos os clientes partem do mesmo estado inicial, e recebem os mesmos comandos, os estados calculados em cada cliente serão idênticos, para qualquer tempo de simulação T.

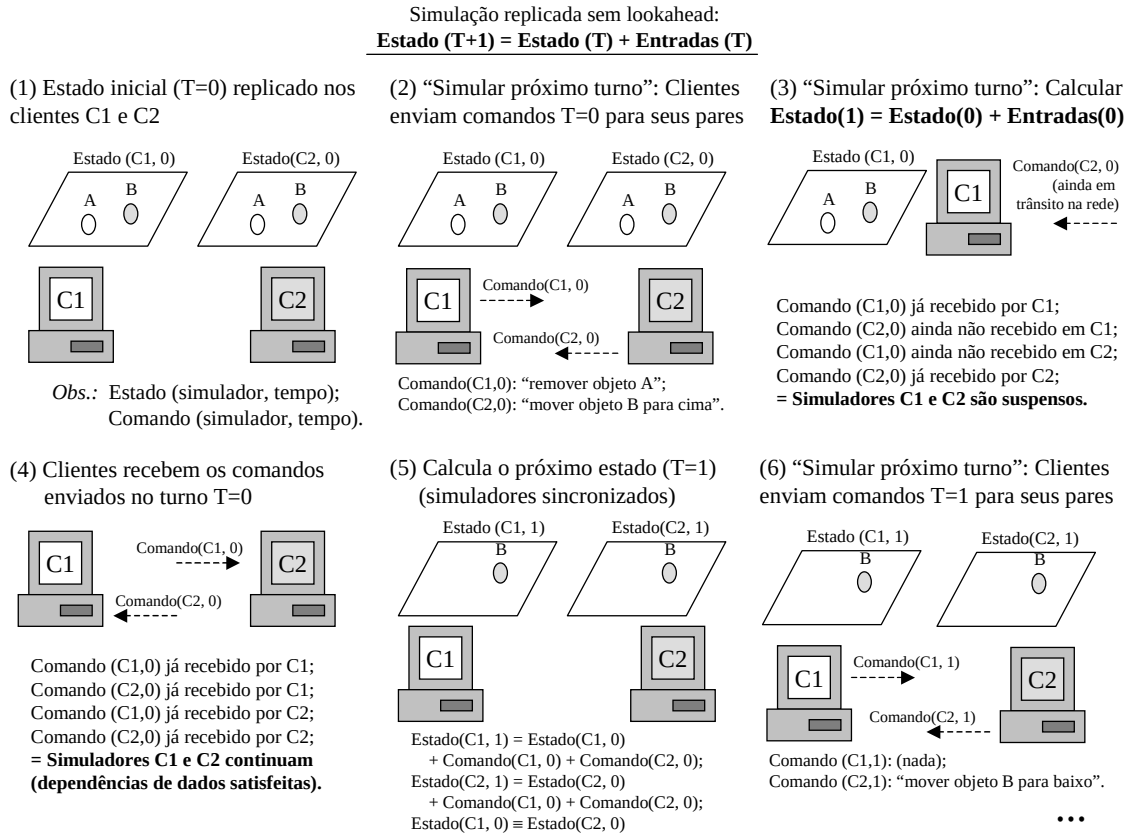


Figura 2.4: Exemplo de atualização do estado replicado, com duas réplicas e sem utilização de *lookahead*

2.5.1 Compensando atrasos de rede com *lookahead*

Apesar da maior capacidade de tratamento de grandes quantidades de objetos, a simulação replicada apresenta um problema oriundo do atraso de rede. Considerando um único cliente (réplica) do jogo, há um atraso entre coletar os comandos locais, emitidos pelo jogador (via teclado, mouse, etc), que recebem um timestamp local T qualquer, e o recebimento dos comandos dos jogadores remotos que também terão este timestamp T. O resultado é que, seguindo a regra de atualização anterior, a simulação em tempo real irá "travar" regularmente, ou seja, a atualização das réplicas não poderá prosseguir temporariamente, por um tempo proporcional ao atraso médio da rede. A tendência é a de que cada cliente do jogo colete seus comandos locais para o turno T aproximadamente ao mesmo tempo, envie este comando para os outros clientes e fique esperando (o equivalente a "um atraso" de rede) pelo recebimento dos comandos de T dos outros clientes. Este cenário é ilustrado pela figura 2.4.

A solução empregada em Age of Empires II, para evitar que os simuladores fossem suspensos enquanto aguardassem comandos, foi modificar a regra de atualização para que os comandos utilizados para atualizar uma réplica do turno T para o turno T+1 fossem os comandos coletados no turno T-2, ou seja, de dois turnos atrás. A duração de um turno em Age of Empires II pode variar, por exemplo, entre 50ms e 200ms. Considerando um turno de 200ms, atualizar uma réplica no turno T com os comandos de T-2 significa mascarar até 400ms de atraso de rede. Em outras palavras, se nenhum cliente demorar mais do que 400ms para enviar um comando para outro cliente, a simulação não irá "travar" por falta de dados. Esta solução causa um atraso mínimo e constante entre a coleta do comando do jogador e a aplicação deste comando. Em Age of Empires II, com um turno de 100ms por exemplo, isto significa que o jogador sempre irá experimentar um atraso mínimo de 200ms entre seu comando ("mover tropas" por exemplo) no dispositivo de entrada (teclado, etc) e a execução do comando no jogo (tropas em movimento). A introdução de um atraso mínimo pode ser algo inviável em alguns tipos de jogo, especialmente nos jogos em tempo real baseados em "avatares" como jogos de tiro em primeira pessoa ou jogos de corrida onde um atraso acima de 100ms já é bastante perceptível e um atraso de 200ms prejudica substancialmente a qualidade do jogo (PANTEL; WOLF, 2002; CRONIN et al., 2003). Porém, em jogos de estratégia em tempo real, onde o jogador tipicamente controla uma grande quantidade de objetos, obter um tempo de atraso constante, em contraste com um tempo de atraso que flutua de forma abrupta, é mais importante do que obter um tempo de atraso muito próximo de zero (BETTNER; TERRANO, 2001).

O restante do texto irá referir como *lookahead* (PREISS; LOUCKS, 1990; FUJIMOTO, 2000) à quantidade de turnos "no passado" do simulador que é utilizada como parâmetro para a regra de atualização do estado. A figura 2.5 ilustra um exemplo com dois simuladores, utilizando a regra de atualização com *lookahead* e um valor de *lookahead* igual a um. O cenário ilustrado assume que a latência da rede é suficientemente menor que o intervalo de tempo real entre as iterações de cada simulador local. Caso o atraso de rede seja maior que este intervalo, os simuladores irão parar enquanto aguardam os comandos necessários. Por isso, é importante configurar um valor adequado ao *lookahead*. Se o valor do *lookahead* for muito reduzido, os simuladores irão "travar" com freqüência, mas se o valor for muito elevado, os comandos dos jogadores irão demorar muito para serem executados, o que reduz a sensação de interação em tempo real com o jogo.

2.5.2 Tolerância a falhas e proteção contra trapaças

A replicação do estado garante duas propriedades: tolerância a falhas e proteção contra jogadores trapaceiros. Se um cliente conectado ao jogo falhar (por exemplo, se sua conexão com a rede é interrompida abruptamente), os outros participantes podem continuar o jogo normalmente após a detecção da falha, pois cada participante possui uma réplica do estado da simulação e não depende de informação mantida pelos outros clientes. Se um cliente resolver alterar ilegalmente sua réplica do estado (por exemplo, fabricando exércitos ou outro tipo de recurso para si mesmo), o cliente trapaceiro irá prejudicar a si mesmo pois irá fazer com que o jogo saia de sincronia sem nenhum ganho real, pois as cópias dos outros jogadores não são afetadas. Fazer com que o jogo saia de sincronia irá fazer, no máximo, com que o mesmo seja interrompido. Isto pode ocorrer se os clientes trocarem *hashes* ou *checksums* das réplicas periodicamente, uma técnica implementada em Age of Empires II. De qualquer maneira, o objetivo do trapaceiro, que é alterar o jogo e passar despercebido, não é alcançado.

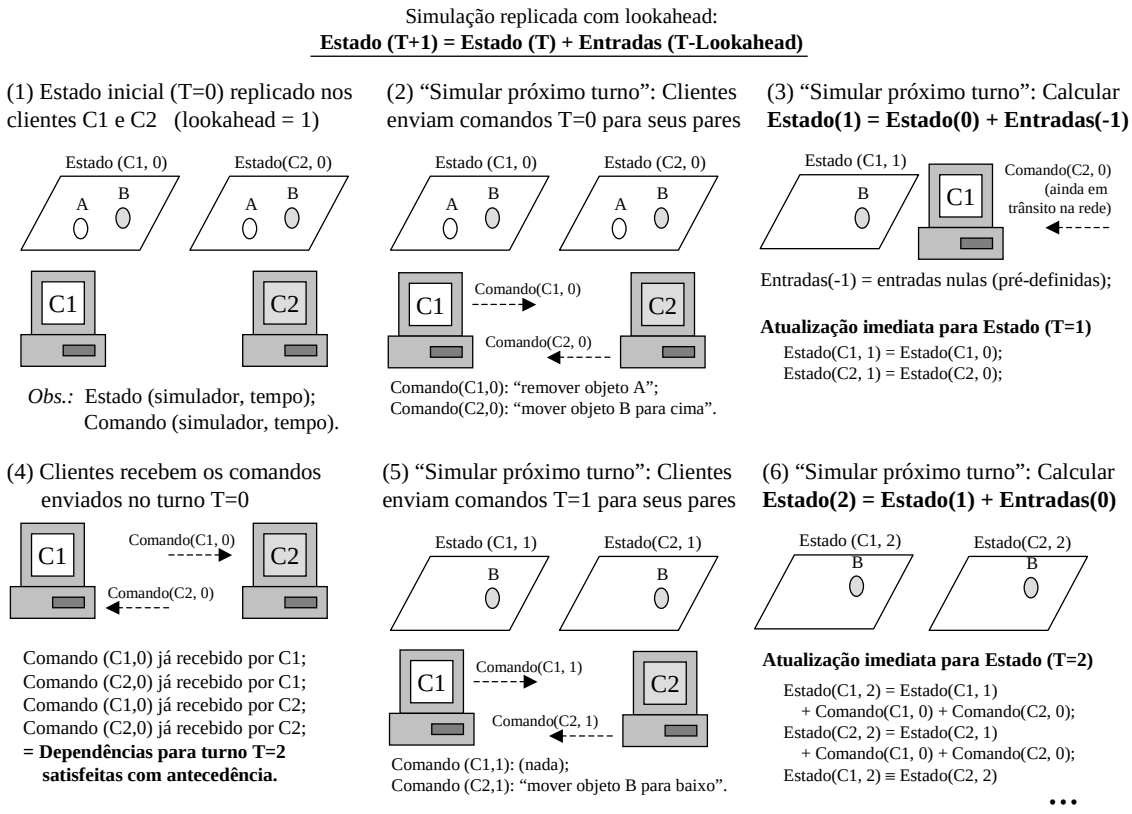


Figura 2.5: Exemplo de atualização do estado replicado, com duas réplicas e com *lookahead* igual a 1

2.5.3 O problema da ocultação parcial do estado

Porém, a replicação do estado apresenta uma desvantagem importante: não há como manter uma parte do estado inacessível a um certo cliente, pois todos os clientes tem acesso a todo o estado replicado do jogo. Uma solução possível é programar o cliente do jogo de forma a não revelar ao jogador humano certa informação. Em Age of Empires II , por exemplo, um jogador só enxerga a porção do campo de batalha que está próxima a alguma de suas tropas, apesar de cada cliente conectado ao jogo possuir informação completa do campo de batalha (posição exata de todas as outras tropas). Esta capacidade de ocultação de informação permite novas táticas de jogo e, dependendo do jogo, pode ser essencial (por exemplo, um jogo de Pôquer). Porém, se o cliente do jogo é responsável pela ocultação (e não o protocolo do jogo), então há a possibilidade de que o cliente do jogo seja adulterado por um jogador trapaceiro para revelar o jogo completo. Este tipo de trapaça ocorreu com Age of Empires II. A solução empregada foi modificar o código compilado do jogo para que se tornasse diferente do anterior, e provavelmente mais obscuro. Note-se que a indisponibilidade do código-fonte do cliente do jogo não chega a ser um problema para os trapaceiros mais perseverantes, como mostra o caso de Age of Empires II.

2.5.4 Resumo e considerações finais

Em resumo, a técnica empregada no jogo Age of Empires suporta jogos distribuídos em pequena escala, de estratégia em tempo real. É empregada uma topologia *peer-to-*

peer onde há uma réplica (cópia) completa do estado do jogo em cada nodo ou cliente participante. Periodicamente, cada cliente envia uma lista com os comandos de jogo acumulados até o momento, recebe as listas dos outros clientes, incrementa o seu contador de tempo de simulação local e atualiza sua réplica do estado do jogo. Como todos os clientes partem do mesmo estado inicial e a entrega de comandos é garantida, a atualização das réplicas é determinística e previsível por todos os clientes, mantendo todas as réplicas em sincronia. Como o estado é totalmente replicado, não há como haver ocultação de objetos entre os jogadores. Porém, a replicação garante tolerância a falhas e também uma forte proteção contra trapaças, duas características importantes. Esta técnica é adequada para implementar jogos *peer-to-peer* de estratégia pois permite que cada jogador controle um número arbitrariamente grande de objetos dinâmicos em tempo real, sem variar o consumo de rede do protocolo. Há um problema de atraso, onde o comando gerado localmente pelo jogador (teclado, mouse) não pode ser aplicado imediatamente pois há a necessidade de esperar os comandos dos outros jogadores para o mesmo turno, sujeitos ao atraso da rede. Este problema é resolvido com o uso de *lookahead*, onde os comandos utilizados na atualização da réplica não são os comandos mais atualmente gerados e sim comandos gerados em um "turno" passado que depende do *lookahead* utilizado pelo jogo. Esta solução é aceitável em jogos de estratégia pois estes são mais tolerantes a um atraso maior (mas fixo) entre um comando gerado localmente e a aplicação deste comando no jogo.

Uma observação interessante é que Age of Empires não possui suporte para que um jogador entre no meio de uma partida em andamento. Porém, a técnica descrita nesta seção poderia ser incrementada para permitir a entrada de novos jogadores. Para isso seria necessário enviar ao novo cliente uma cópia do estado atual do jogo. Devido à quantidade de objetos dinâmicos em jogo, e do próprio jogo em questão, uma cópia do estado pode ter um tamanho significativo, e isto significa que haveria um certo tempo mínimo necessário para a transmissão do mesmo. A transmissão da cópia do estado também precisaria ser concorrente com o envio das listas de comandos pois ela necessita de atualizações a partir do momento que começa a ser transmitida. Se as listas de comandos são enviadas a cada 100ms, por exemplo, e a transmissão de uma cópia do estado para o novo jogador demorar 1 minuto, isto significa que quando a cópia for recebida, ela já teria que ser atualizada 600 vezes.

2.6 Suporte descentralizado a jogos maciçamente multijogador

Como discutido anteriormente, os sistemas de suporte a MMGs atualmente empregados comercialmente são, em sua grande maioria, baseados em uma filosofia de execução "centralizada" da simulação (GAUTHIERDICKEY; ZAPPALA; LO, 2004), na medida em que atualizações ao estado do jogo (por exemplo, decidir qual jogador ganhou uma disputa) são computadas ou autorizadas por servidores localizados em domínios administrativos "confiáveis". Isto ocorre tanto para MMGs cliente-servidor, quanto para MMGs baseados em arquiteturas do tipo "computação em grade", como é o caso do Butterfly Grid (BUTTERFLY.NET, 2003).

Neste contexto, há um crescente interesse na pesquisa e desenvolvimento de arquiteturas mais "descentralizadas" (ou "par-a-par") de suporte a MMGs. Os modelos científicos propostos buscam a descentralização, em geral, como forma de aumentar a escalabilidade do modelo e reduzir a dependência em nodos alocados em domínios "confiáveis". Outros benefícios incluem a eliminação dos pontos centrais de falha (BHARAMBE;

RAO; SESHAN, 2002), bem como o aumento da responsividade, que pode ser obtido através da exploração da localidade nas interações entre os "avatares" dos jogadores no mundo virtual (KNUTSSON et al., ????). Neste sentido, estes modelos são diretamente relacionados com o modelo FreeMMG, proposto neste trabalho.

Algumas destas propostas não tratam o problema da proteção contra jogadores trapaceiros, como é o caso de Mercury (BHARAMBE; RAO; SESHAN, 2002) e a "arquitetura de comunicação para jogos maciçamente multijogador" proposta por Fiedler et al (FIEDLER; WALLNER; WEBER, 2002). Ambas as arquiteturas tratam o problema da escalabilidade ao propor uma solução baseada no paradigma publicador-assinante. Neste paradigma, o mundo virtual é dividido em partes menores, geralmente denominadas de "células", e cada participante pode optar por assinar (ou seja, participar) de apenas algumas células. Desta forma, cada assinante de uma célula só precisa trocar mensagens de atualização com os assinantes da mesma célula. As arquiteturas também tratam de outros problemas básicos, como o problema da responsividade aos comandos gerados por cada participante. Mercury enfatiza a robustez da arquitetura em execução, enfocando a eliminação do servidor, que é considerado um ponto central de falha. Já Fiedler et al propõem a divisão do mundo virtual não só em células geográficas como também em "canais" dentro de cada célula, sendo que em cada canal trafegam mensagens com requisitos diferentes. Por exemplo, uma célula pode possuir um canal "de ambiente", com envio de mensagens menores e de maior prioridade (mudanças no "mapa" do mundo, por exemplo) e um canal "de interações", com envio de mensagens maiores e de menor prioridade (atualizações de posição de "avatares", por exemplo).

Recentemente, observa-se que as propostas de arquiteturas de suporte a MMGs que utilizam modelos de distribuição *peer-to-peer* estão se tornando cada vez mais adequadas à realidade da Internet e às necessidades dos MMGs "reais". Um exemplo disto é o modelo NEO, publicado recentemente (GAUTHIERDICKEY; ZAPPALA; LO, 2004; GAUTHIERDICKEY et al., 2004). O modelo NEO é dividido em quatro componentes principais: autenticação, comunicação, armazenamento e computação. O componente de autenticação é responsável por controlar o acesso dos jogadores ao jogo, e deve executar em máquinas "confiáveis". O componente de armazenamento provê o armazenamento de longo prazo do estado persistente do mundo virtual através de uma tabela *hash* distribuída (DHT ou *distributed hash table*), sem a necessidade de um armazenamento persistente em máquinas confiáveis. O componente de comunicação determina como os jogadores em um mesmo "grupo NEO" enviam mensagens entre si (por multicast ou unicast, etc), utilizando o "protocolo NEO". Por fim, o componente de computação decide como distribuir a responsabilidade da atualização do mundo virtual entre os vários nodos "clientes" que participam da simulação. O componente de computação também evita a ocorrência de trapaças através da replicação de uma mesma computação em vários clientes.

Como a maioria dos sistemas de suporte a NVEs, o modelo NEO explora a localidade das interações entre os jogadores. Neste sentido, um "grupo NEO" é um grupo de simulação *peer-to-peer* em pequena escala, ou seja, um subconjunto dos participantes do mundo virtual. O "protocolo NEO" é utilizado para a comunicação entre os participantes de um grupo NEO. Este protocolo é um refinamento do protocolo *lockstep* ou "simulação *peer-to-peer* replicada", apresentado na seção anterior, sendo acrescentadas algumas características presentes em outros protocolos, como a proteção contra trapaças do protocolo "*lockstep* seguro" (BAUGHMAN; LEVINE, 20., 2002) e a redução do *jitter* (intervalo entre o recebimento de pacotes) presente no protocolo *Sliding Pipeline* (CRONIN et al., 2003).

Atualmente, a parte mais desenvolvida do modelo NEO é o componente de comunicação, e a modelagem de algumas partes importantes deste componente ainda permanecem em aberto (GAUTHIERDICKEY; ZAPPALA; LO, 2004; GAUTHIERDICKEY et al., 2004). Por exemplo, a versão atual do modelo não especifica como seria feita a formação dos "grupos NEO" (ou seja, como seria feita a inclusão ou a remoção de jogadores de um grupo) ou como o "protocolo NEO" propagaria os eventos que afetam mais de um grupo ou "célula" do mundo virtual (o que incluiria, em princípio, o suporte à interação entre "objetos de jogo" localizados em diferentes grupos). Por outro lado, o projeto NEO possui objetivos interessantes, em especial a intenção de suportar jogos maciçamente multijogador de ação, baseados em "avatares".

2.7 Considerações finais

Nos últimos anos, observa-se um aumento crescente no contexto de pesquisa e desenvolvimento de sistemas baseados em *peer-to-peer* para a Internet (ORAM, 2001). A crescente popularização do acesso de "banda larga" à Internet, bem como a proliferação de dispositivos móveis e tecnologias de acesso sem-fio, tornam os equipamentos "clientes" cada vez mais aptos a atuarem não só como consumidores de informação, mas também como produtores. Cada vez mais, problemas tipicamente difíceis de serem tratados de forma distribuída, se tornam viáveis de serem resolvidos, total ou parcialmente, através de computação colaborativa, sem a necessidade de participação de entidades centrais (CLARKE, 2002; KELLER; SIMON, 2002) ou, pelo menos, através de arquiteturas híbridas, onde a entidade central tem um papel reduzido na rede (COHEN, 2003; PELLEGRINO; DOVROLIS, 2003; ANDERSON et al., 2002).

A análise do estado-da-arte do suporte a jogos distribuídos para a Internet mostra que há uma lacuna em relação a sistemas de suporte de jogos maciçamente multijogador através de plataformas que não exijam um "lado servidor" custoso. Por um lado, as plataformas cliente-servidor e Grid, que são usadas atualmente no suporte a MMGs, exigem uma estrutura servidora custosa, tanto em relação ao poder de processamento, quanto em relação aos recursos de rede, relativos à interconexão entre os servidores e à conexão com a Internet para provimento do serviço aos jogadores. Por outro lado, existem propostas acadêmicas de modelos descentralizados, totalmente *peer-to-peer*, e também de modelos híbridos, que combinam as vantagens dos modelos cliente-servidor e *peer-to-peer*.

Porém, alguns destes modelos não oferecem garantias de proteção contra jogadores trapaceiros, o que é uma característica essencial para a implementação de soluções para a Internet. Além disso, foi verificado que há uma lacuna em relação ao suporte a jogos de estratégia em tempo real (jogos RTS), em um contexto maciçamente multijogador, justamente devido às características de centralização da simulação que são buscadas pelas implementações atuais de MMGs comerciais. Neste contexto insere-se a motivação da pesquisa e desenvolvimento do modelo FreeMMG. O capítulo 3 descreve o modelo proposto, que caracteriza uma arquitetura híbrida, cliente-servidor e *peer-to-peer*, de suporte a jogos maciçamente multijogador, que utiliza um servidor de baixo custo ao mesmo tempo em que oferece resistência a alguns dos principais tipos de trapaças em jogos distribuídos.

3 MODELO FREEMMG

Este capítulo descreve o modelo FreeMMG, que é um sistema de suporte otimizado para aplicações do tipo "jogo *massively multiplayer* de estratégia em tempo real". A seção 3.1 caracteriza o modelo de aplicação suportado pelo modelo FreeMMG. A seção 3.2 fornece uma descrição de alto nível do modelo FreeMMG. As seções seguintes descrevem em detalhe os diferentes aspectos do modelo, como a gerência de grupos de jogadores, persistência do "mundo virtual", e detecção e recuperação de falhas. Por fim, é feita uma discussão sobre a contribuição do modelo, bem como sobre os pontos em que é possível desenvolver trabalhos futuros.

3.1 A aplicação: jogo de estratégia em tempo real maciçamente multijogador (MMORTS)

Um jogo de estratégia em tempo real ou *real-time strategy game* (geralmente abreviado para "RTS") é um jogo de ação que simula um ambiente virtual onde um ou mais jogadores controlam, em tempo real, um conjunto de objetos. Em geral, os jogos RTS simulam guerras onde cada jogador controla seu próprio exército com o objetivo de derrotar os oponentes (outros jogadores conectados por rede local ou Internet). Em geral, os jogos RTS também simulam um sistema econômico simplificado, envolvendo a extração de recursos naturais (minério, combustível, alimentos, etc) e a conversão destes recursos em "objetos de jogo" (construções, soldados, tecnologia, etc) que são necessários para que um jogador supere os seus adversários. "Age of Empires II" (ENSEMBLE, 2003), "Warcraft III" (BLIZZARD, 2004a), "Starcraft" (BLIZZARD, 2004b), "Rise of Nations" (BIG HUGE GAMES, 2004) e "Rome: Total War" (TOTAL WAR, 2004) são exemplos de jogos do tipo RTS.

O típico jogo RTS não é projetado para ser escalável ou para suportar estado de jogo persistente. Os jogos RTS com suporte "multijogador" (participantes conectados por rede local ou Internet) geralmente não suportam mais do que 8 ou 16 participantes simultaneamente. Os jogos RTS são disputados em "partidas" cujo cenário inicial é escolhido de comum acordo entre todos os participantes. E, após iniciada, uma partida de jogo RTS dura apenas algumas horas (em torno de duas horas sendo uma duração típica). Após o término da partida é declarado o vencedor (ou vencedores), e novas partidas podem ser disputadas a partir de um novo cenário inicial, sem relação com o resultado da partida anterior.

O modelo FreeMMG é uma solução para o problema de suporte à execução de jogos de estratégia em tempo real maciçamente multijogador ou *massively multiplayer online real-time strategy game* ("MMORTS", para abreviar). O MMORTS é um jogo RTS

maciçamente multijogador, ou seja, projetado para suportar milhares de jogadores simultaneamente que interagem, em tempo real, em um “mundo virtual” de estado persistente. Para facilitar o desenvolvimento do modelo, foi criado um “modelo abstrato” de aplicação do tipo “jogo MMORTS”, que é descrito a seguir.

3.1.1 Modelo de aplicação do FreeMMG

A aplicação suportada pelo modelo FreeMMG é um jogo de estratégia em tempo real maciçamente multijogador (jogo MMORTS) que implementa os seguintes conceitos-chave: “jogador”, “objeto”, “mundo” e “área de interesse”. A figura 3.1 ilustra um cenário de jogo que envolve estes conceitos, que são definidos a seguir.

O “mundo” é o sistema de regras ou “leis” que implementam o espaço virtual onde estão localizados os “objetos”. Espera-se que o mundo simule um ambiente virtual típico, onde os objetos possuem uma posição relativa aos outros objetos, tamanho, velocidade, aceleração, etc. Espera-se também que quaisquer objetos localizados no mesmo mundo possam interagir entre si (colisão, troca de informações, etc) e que as interações entre objetos ocorram com maior freqüência quando estes objetos estão “próximos”.

O “objeto” é o elemento básico que compõe o conjunto de informações dinâmicas do mundo. Ou seja, qualquer informação que pode ser adicionada, copiada, alterada ou removida do mundo virtual pode ser modelada como um objeto, ao passo que a informação estática pode ser modelada como parte do mundo. Se o “mundo virtual” do jogo é uma simulação de um campo de batalha, exemplos típicos de objetos que serão percebidos pelos usuários do jogo são os soldados, veículos, construções, projéteis, etc. que habitam o campo de batalha. Um objeto pode (opcionalmente) ser associado a um “jogador”, que é então o “dono” deste objeto.

O “jogador” representa o usuário do jogo, que interage com o mundo ao exercer controle sobre os objetos. Cada jogador, tipicamente, só poderá controlar diretamente uma certa parte dos objetos. Por exemplo, pode-se determinar que um jogador pode controlar apenas os objetos que são marcados como tendo o jogador em questão como “dono”. Como se trata de um jogo maciçamente multijogador com um mundo virtual arbitrariamente grande, cada jogador poderá interagir, em um dado momento, com apenas um subconjunto do mundo. O subconjunto do mundo que um jogador pode interagir define a “área de interesse” (MORSE, 1996) deste jogador. Por default, a área de interesse de um jogador compreende o espaço do mundo que é ocupado pelos objetos que pertencem ao jogador. Porém, o modelo FreeMMG deve suportar qualquer tipo de critério para o cálculo da área de interesse de um jogador. Ou seja, o critério de cálculo da área de interesse deve ser configurável pela aplicação.

Como será visto nas seções seguintes, o FreeMMG propõe o particionamento do mundo virtual em “segmentos”. O segmento é a menor unidade do “mundo virtual” que pode ser adicionada ou não à área de interesse de um jogador. Ou seja, em tempo de execução, a aplicação pode adicionar ou não um segmento, com todos os objetos que estão localizados naquele segmento, à área de interesse de um jogador. Mesmo que apenas um objeto em um segmento seja “de interesse” a um jogador, o mesmo terá que receber atualizações para todo o segmento.

Um exemplo de jogo implementado a partir deste modelo de aplicação é o jogo-protótipo “FreeMMG Wizards” (ou simplesmente Wizards), um MMORTS simples que foi implementado para auxiliar o processo de validação do modelo FreeMMG. No Wizards, o mundo virtual é particionado em segmentos retangulares e um segmento é considerado pertencente à “área de interesse” de um jogador se este jogador é “dono” de,

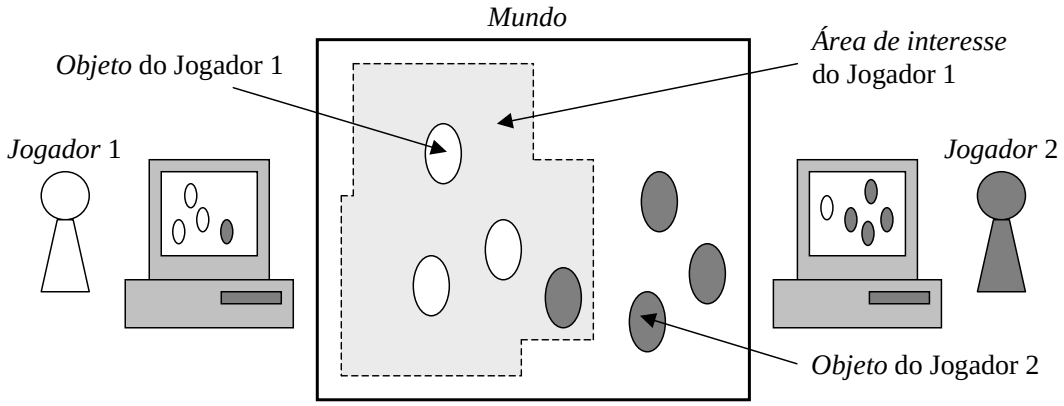


Figura 3.1: FreeMMG: modelo de aplicação suportada

no mínimo, um objeto no segmento em questão. Mais detalhes sobre o protótipo são apresentados no capítulo 4.

3.2 Modelo FreeMMG: visão geral

O FreeMMG é um modelo de suporte para a distribuição de jogos maciçamente multijogador (MMGs) otimizado para o suporte de jogos maciçamente multijogador de estratégia em tempo real (jogos MMORTS), descritos na seção anterior. O FreeMMG utiliza uma estratégia de distribuição híbrida, baseada em parte no paradigma cliente-servidor e em parte no paradigma *peer-to-peer*. Esta estratégia visa usufruir das vantagens de ambos os paradigmas. O componente "cliente-servidor" do FreeMMG caracteriza-se pela utilização de nodos ou máquinas servidoras confiáveis para a realização de tarefas críticas, como autenticação e autorização de usuários, que são difíceis de serem distribuídas de forma segura entre nodos anônimos e não-confiáveis. A versão atual do modelo FreeMMG suporta a utilização de apenas um único servidor (para cada instância de "mundo virtual" simulado). Porém já existem trabalhos que apontam para a possibilidade de extensão do modelo FreeMMG para utilização de um servidor distribuído (MENDES, 2004).

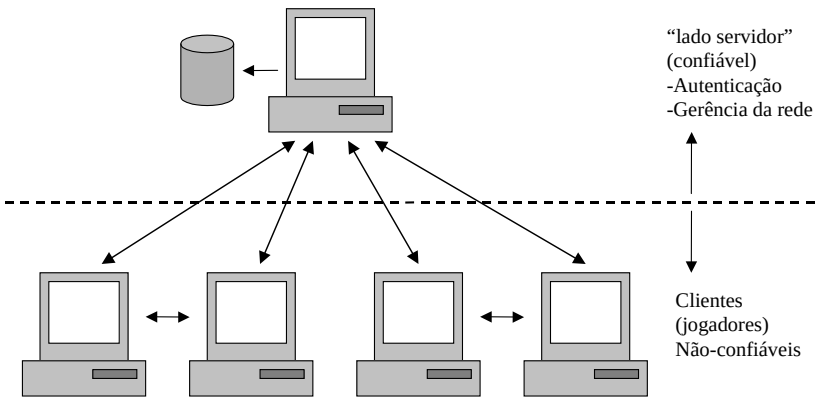


Figura 3.2: A rede FreeMMG: servidor, clientes, *peer-to-peer* entre clientes

É de interesse do provedor do jogo (ou seja, de quem possui poder administrativo sobre as máquinas servidoras) que o custo associado à implantação e à manutenção dos servidores seja o menor possível. Com a distribuição do servidor FreeMMG, um eventual gargalo de processamento (CPU) no "lado servidor" pode ser aliviado. Porém, a quantidade de dados trocados entre os servidores e os clientes permanece a mesma, pois mesmo que cada servidor fique responsável por se comunicar com apenas parte dos clientes, o custo da comunicação entre os servidores e todos os clientes ainda recai sobre o provedor do serviço.

Para evitar que o "lado servidor" se torne um gargalo, tanto de recursos de rede quanto de processamento, o FreeMMG utiliza, juntamente com a estratégia cliente-servidor, uma estratégia *peer-to-peer*, onde os nodos "clientes" da simulação, ou seja, as máquinas dos jogadores, negociam parte do jogo diretamente entre si, evitando assim a utilização do servidor como único ponto de sincronização entre os jogadores. Porém, como discutido no capítulo anterior, ao se considerar uma estratégia de execução descentralizada para jogos, especialmente jogos voltados para redes com participantes anônimos como a Internet, deve-se também considerar como será resolvido o problema oriundo da participação de nodos maliciosos (trapaceiros ou vândalos) na simulação.

3.2.1 Simulação *peer-to-peer* replicada no FreeMMG

Para executar a simulação de forma descentralizada e, ao mesmo tempo, oferecer um mecanismo de resistência contra nodos maliciosos, o FreeMMG utiliza a técnica de simulação *peer-to-peer* com replicação de estado descrita na seção 2.5 do capítulo anterior. Desta forma, os clientes podem sincronizar o andamento do jogo, em tempo real, diretamente entre si. O servidor se envolve na comunicação apenas para realizar tarefas administrativas eventuais, como inserir e retirar clientes da rede.

O principal problema desta técnica de simulação *peer-to-peer* com replicação é que esta não é uma técnica escalável, pois ela exige que cada participantes tenha uma conexão direta com cada outro participante e comunique-se em tempo real com cada um. A técnica funciona para suportar 10 ou 20 nodos de jogadores, mas não é adequada para suportar MMGs com milhares de participantes. Para resolver este problema, o FreeMMG estabelece que o "mundo virtual" do jogo deve ser particionado em "segmentos" menores. Cada segmento é simulado por um grupo de nodos conectados em uma rede *peer-to-peer* que é independente das demais redes, e cada nodo cliente irá participar apenas dos grupos que são do seu interesse em um determinado momento. Desta forma, mantém-se um número reduzido de nodos em cada grupo ou segmento, de forma que o algoritmo de simulação *peer-to-peer* replicada possa ser utilizado.

Um cliente que participa do grupo que executa a simulação de um segmento é dito ser "assinante" do segmento em questão. Cada assinante de um segmento possui uma cópia (réplica) do estado do segmento (relativo a um ponto "t" no tempo de simulação daquele segmento). Os assinantes de um segmento trocam mensagens entre si em tempo real em turnos de tempo de comprimento fixo (por exemplo, 200ms). As mensagens indicam os comandos ou ações que cada assinante está realizando sobre o estado daquele segmento.

No modelo de aplicação do FreeMMG apresentado anteriormente, um jogador pode controlar os objetos de jogo que possui, a qualquer momento. Cada objeto está localizado em um segmento específico. Portanto, a "área de interesse" de um jogador é caracterizada por todos os segmentos em que possui pelo menos um objeto. Para que o jogador possa interagir com todos os objetos dentro da sua área de interesse, este deve "assinar" todos os segmentos que estão nesta área, pois é através da assinatura de um segmento que um

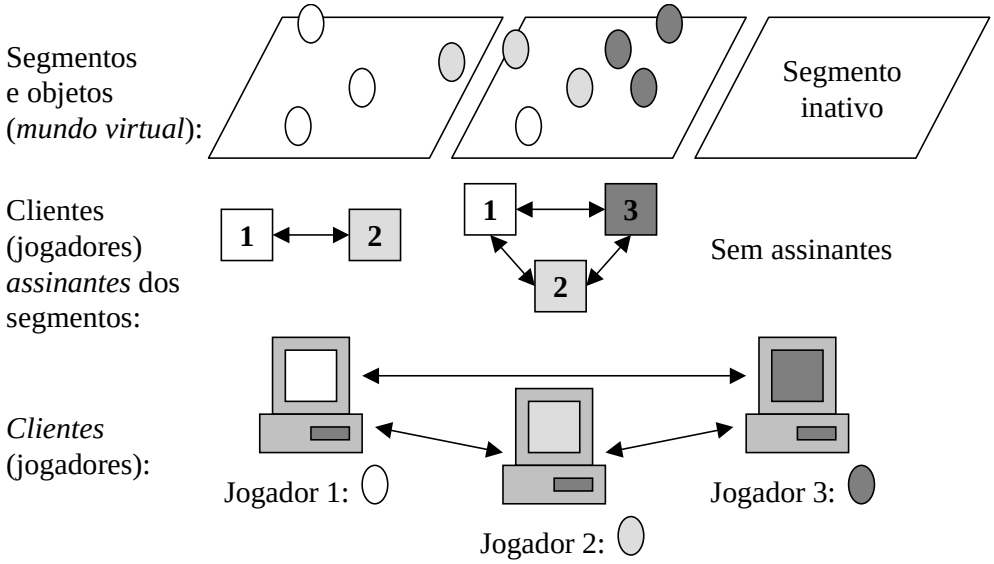


Figura 3.3: Mapeamento entre segmentos, assinantes e clientes, sendo utilizada a regra "default" para o cálculo da área de interesse dos jogadores

cliente pode interagir com os objetos que estão no segmento.

É importante ressaltar aqui que o critério de cálculo de áreas de interesse pode ser customizado pela aplicação do FreeMMG. Implementações do modelo FreeMMG podem refinar o conceito de "posse" de objetos e relações entre jogadores para implementar regras mais complexas de cálculo de áreas de interesse. Por exemplo, as regras do jogo podem determinar que um jogador pode possuir "aliados", e que todos os jogadores que são aliados devem compartilhar o mesmo campo de visão, apesar de não exercerem controle sobre os objetos dos outros aliados. Esta é uma semântica bastante empregada para implementar "áreas de interesse" em jogos de estratégia em tempo real que permitem a formação de alianças entre os jogadores.

3.2.2 Interação e sincronização do tempo entre segmentos

A técnica de partição do estado do jogo em "segmentos" garante escalabilidade (JOGALEKAR; WOODSIDE, 2000) ao modelo, mas introduz um novo problema, que é o problema do suporte à interação entre objetos de jogo localizados em segmentos distintos. Se cada segmento executar seu próprio protocolo de simulação, sem envolver os outros segmentos, de forma isolada, então não tem-se um sistema de suporte a um "mundo virtual" escalável e sim um sistema de suporte a um grande número de "pequenos mundos virtuais" que não interagem entre si.

Uma solução ideal para este problema envolveria um suporte para que os objetos de jogo pudessem interagir entre si utilizando sempre as mesmas regras, independente do segmento em que os mesmos estão localizados. Porém, o protocolo de simulação *peer-to-peer* replicada, que é utilizado pelo FreeMMG para executar a simulação em pequena escala de cada segmento, não pode ser facilmente adaptado para esta situação. Isto ocorre pois a simulação *peer-to-peer* replicada só suporta a interação entre dois objetos se eles estão na mesma réplica. Desta forma, objetos localizados em réplicas diferentes só poderiam interagir se as réplicas envolvidas forem somadas, o que resultaria, na prática, em um mundo virtual não particionado e não escalável. Portanto, o modelo FreeMMG determina

que objetos em segmentos distintos não podem interagir diretamente entre si. Para que dois objetos localizados em segmentos distintos interajam, é necessário que um deles seja transferido (pelo jogador que o controla) para o segmento onde está localizado o outro objeto. Neste sentido, o FreeMMG suporta a "transferência de objetos" entre segmentos através de um protocolo específico (figura 3.4). A transferência de um objeto entre segmentos é feita através do servidor. Os assinantes do segmento de origem enviam o objeto para o servidor, e este envia o objeto para o segmento destino.

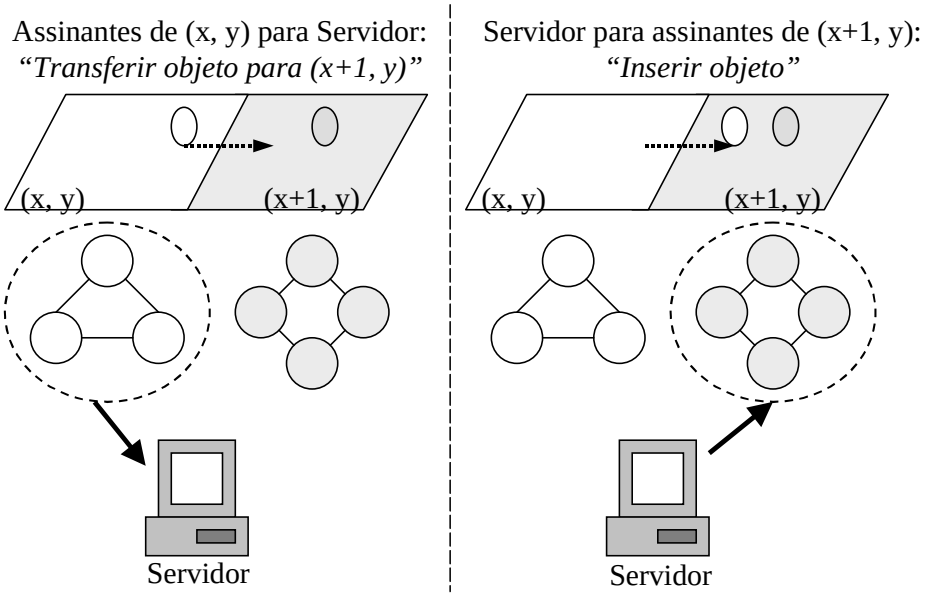


Figura 3.4: Transferência de objeto, gerenciada pelo servidor, entre assinantes de segmentos

Ao transferir objetos entre segmentos, o servidor atua como um "sincronizador" de eventos distribuídos (FUJIMOTO, 2000), que envolvem mais de um segmento. O servidor ordena estes "eventos distribuídos" de acordo com a ordem em que são recebidos. Eventualmente, se os clientes da rede FreeMMG fossem modificados para trocar eventos distribuídos diretamente entre si, sem a intervenção do servidor, seria necessária alguma forma de sincronização dos relógios entre os participantes (FUJIMOTO, 2000), por exemplo, através do relógio lógico de Lamport (LAMPORT, 1978) ou de *proper time* (HRISCHUK; WOODSIDE, 1996). Dentro de cada segmento, o ordenamento dos eventos também é total, pois é relativo ao relógio (contador de "turnos") do algoritmo de "simulação *peer-to-peer* replicada" apresentado na seção 2.5.

3.2.3 Detecção e tratamento de falhas

O sistema de suporte à distribuição de um MMG deve especificar como é feito o tratamento de falhas de clientes. Um cliente é considerado "falho" quando deixa de se comportar do modo previsto (WEBER; JANSCH-PÔRTO; WEBER, 1996). Por exemplo, o servidor pode parar de receber dados do cliente, devido à perda de conexão (falha da rede) ou queda do processo cliente devido a defeitos no programa.

O cliente pode falhar também devido à ação de clientes maliciosos. Pode-se considerar basicamente dois tipos de clientes maliciosos: o vândalo e o trapaceiro. O vândalo quer apenas prejudicar o andamento do sistema, isto é, fazer com que ele pare de funcionar.

Por exemplo, o vândalo pode propositalmente enviar mensagens mal-formadas para outros clientes ou para o servidor, visando explorar fraquezas na implementação dos mesmos e causar, por exemplo, esgotamento de recursos, caracterizando então um ataque similar a "negação de serviço". O resultado de vandalismo geralmente é facilmente detectável, apesar de que a fonte do ataque pode ser facilmente mascarada, ou seja, o servidor dificilmente vai saber qual identidade de jogador (se alguma) que está por trás dos ataques. Por outro lado, o objetivo do cliente "trapaceiro" é o de ganhar vantagem no jogo ao mesmo tempo em que sua ação passe despercebida. Uma "trapaça" bem-sucedida, como alterar o estado do jogo para conferir uma vantagem extra para algum jogador, é uma situação de falha que não foi detectada pelo sistema e, portanto, não dispara um processo de tratamento.

O FreeMMG especifica uma série de estratégias para detecção e tratamento de falhas de clientes, tanto falhas não-intencionais (clientes perdem conexão, etc) como falhas intencionais, causadas por clientes maliciosos (trapaceiros ou vândalos). A base de todas as estratégias é a replicação (ou redundância), seja replicação de estado, através do armazenamento de cópias do jogo em vários nodos, de computação, através de computação redundante em vários nodos, ou de comunicação, com o envio de mensagens redundantes por várias fontes. Por exemplo, a técnica de "simulação *peer-to-peer* replicada", que é empregada pelos grupos de assinantes de segmentos, utiliza redundância de estado, pois cada assinante mantém uma cópia do estado do segmento, e também utiliza redundância de computação, pois cada assinante deve processar o mesmo conjunto de comandos para atualizar a sua cópia. A redundância ajuda a minimizar as chances de que haja uma alteração ilegal no estado do jogo ou que haja uma perda total de parte do estado do jogo em caso de falha não-intencional de um cliente (por exemplo, por uma perda súbita de conexão).

A redundância oferecida pela replicação de estado nos clientes fornece um nível de tolerância a falhas sem aumentar o peso no lado servidor, que é o recurso da rede que precisa ser economizado. Porém, ela não garante uma segurança total contra falhas, especialmente falhas intencionais do tipo "trapaça". Por exemplo, mesmo que cada segmento tenha vinte réplicas (assinantes), nada impediria que estes vinte clientes componham um grupo pré-combinado de "trapaceiros", que irão alterar o estado de forma ilegal em comum acordo. Em outras palavras, utilizar apenas replicação de estado, onde nenhum dos clientes envolvidos é confiável, pode dificultar a execução de trapaças, mas, a princípio, não as impede. Nas seções seguintes, serão detalhados mecanismos alternativos para reduzir ou mesmo eliminar a possibilidade de que clientes (ou grupos de clientes) "trapaceiros" possam alterar ilegalmente o estado do "mundo virtual".

Os algoritmos do FreeMMG que realizam replicação, que serão descritos nas seções seguintes em maiores detalhes, são inspirados em soluções para problemas do tipo "generais bizantinos" (CASTRO; LISKOV, 1999; LAMPORT, 1983). No caso, cada "assinante" de um segmento é um componente que pode "falhar" ou agir maliciosamente, representando um "general bizantino". Porém, como será visto adiante, os limiares de aceitação do resultado após uma falha não são estabelecidos pelo modelo (por exemplo, baseados em limiares obtidos de soluções publicadas para falhas bizantinas), mas deixados para configuração pela aplicação. Um trabalho que busca valores adequados para os limiares e aprofunda a comparação dos problemas do FreeMMG com o problema do consenso distribuído dos "generais bizantinos" encontra-se em desenvolvimento (SANTOS MARTINS, 2004).

3.3 Mensagens replicadas

Como discutido na seção anterior, a replicação do estado de cada segmento é importante pois reduz a chance de que o estado do segmento seja alterado de forma ilegal por um ou mais jogadores maliciosos. Além de replicar o estado do segmento, o modelo FreeMMG também determina que os eventos distribuídos (FUJIMOTO, 2000), gerados no contexto de um segmento, devem ser enviados para os outros segmentos de forma redundante através de uma "mensagem replicada". Isto é feito para prevenir a ocorrência de trapaças, como será discutido a seguir. Existem dois tipos de "mensagem replicada" utilizados no FreeMMG:

- Mensagem segmento-servidor: mensagem enviada por todos os assinantes de um segmento para o servidor;
- Mensagem servidor-segmento: mensagem enviada pelo servidor para todos os assinantes de um segmento.

A mensagem segmento-servidor é uma mensagem enviada por várias fontes que, individualmente, não são confiáveis, ou seja, um conjunto de clientes que assinam um segmento. O receptor da mensagem replicada (o servidor) aguarda o recebimento de todas as cópias da mensagem antes de validar e processar a mesma. Mesmo que apenas um dos assinantes não envie uma cópia da mensagem, a mesma será descartada, gerando um evento de falha. O tempo limite para o recebimento da última cópia da mensagem é determinado pelo tempo em que foi recebida a primeira cópia. Por exemplo, se o "timeout" para mensagens replicadas for configurado para 10 segundos, o recebimento da mensagem replicada irá falhar se não forem recebidas todas as cópias até 10 segundos após o recebimento da primeira cópia. Isto caracteriza uma falha individual por tempo limite (seção 3.10.1) que, por sua vez, causa uma falha de grupo, cujo tratamento é discutido a partir da seção 3.11.

A exigência de que todos os assinantes enviem para o servidor uma cópia da mensagem segmento-servidor evita a ocorrência de trapaças como, por exemplo, a "fabricação" de quantidades arbitrárias de objetos de jogo através de transferências ilegais, como é ilustrado pela figura 3.5. Como será discutido em detalhes na seção seguinte, a transferência de um objeto entre dois segmentos é iniciada por uma mensagem replicada segmento-servidor que parte do segmento de "origem" do objeto. Se o servidor aceitasse um pedido de transferência de objeto informado por apenas um assinante do segmento, por exemplo, este assinante poderia gerar, com facilidade, qualquer quantidade de objetos "inexistentes", ao transferir estes objetos para os segmentos adjacentes. Com a utilização da mensagem replicada, esta trapaça só é possível se todos os assinantes de um segmento resolverem trapacear em comum acordo, o que é mais difícil de ocorrer (GAUTHIERDICKEY; ZAPPALA; LO, 2004).

A mensagem servidor-segmento é uma mensagem enviada pelo servidor para todos os assinantes de um segmento. Ao enviar a mensagem diretamente a todos os assinantes, o servidor evita a utilização de intermediários, que podem simplesmente não enviar a mensagem ao destino. Isto poderia ocorrer se, por exemplo, o servidor solicitasse a um assinante que entregasse a mensagem para outro assinante do mesmo segmento. No caso da mensagem replicada do tipo "servidor-segmento", a entrega da mensagem é garantida pelo protocolo de rede utilizado para conectar o servidor com cada cliente (por exemplo, TCP/IP). "Timeouts" podem ser detectados pelo servidor através de mensagens adi-

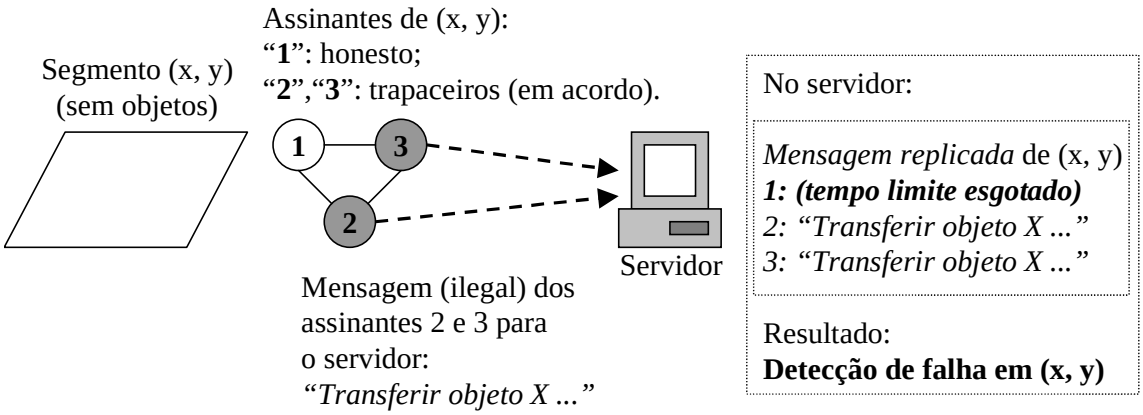


Figura 3.5: Exemplo de mensagem segmento-servidor evitando a transferência ilegal de um objeto

cionais, por exemplo, ao esperar uma mensagem segmento-servidor de resposta ou outra mensagem qualquer enviada pelos clientes.

3.4 Algoritmo de transferência de objeto entre segmentos

Um algoritmo do FreeMMG que utiliza tanto mensagens replicadas segmento-servidor quanto servidor-segmento é o algoritmo de transferência de objeto entre dois segmentos. Como citado anteriormente, a transferência de um objeto começa com o envio de uma mensagem dos assinantes do segmento de "origem" (em que o objeto está localizado originalmente) para o servidor. Para reduzir o consumo de recursos de rede, poderia se considerar que apenas um dos assinantes precisasse enviar o objeto para o servidor. Porém, isto não pode ser feito pois o cliente escolhido pode resolver "mentir" para o servidor para, por exemplo, trapacear e obter uma vantagem no jogo (por exemplo, fabricando uma quantidade qualquer de objetos no segmento destino da transferência). Para reduzir as chances de que uma transferência ilegal de objeto ocorra, o servidor exige que todos os assinantes enviem a mensagem que contém o objeto a ser transferido. Portanto, a mensagem de envio de objeto, de um segmento para o servidor, é uma "mensagem replicada".

O algoritmo de transferência de objetos consiste dos seguintes passos, onde S é o servidor, O representa o objeto a ser transferido, A é o segmento (grupo de assinantes) de origem e B é o segmento (grupo de assinantes) destino:

- Cada assinante de A detecta, individualmente, que o objeto O, que inicialmente reside no segmento A, deve ser transferido para o grupo B, e envia "mensagem coletiva" para S contendo (t, O), onde "t" é o valor do relógio (timestamp) do simulador local quando O foi removido de A para que fosse transferido;

- S recebe a mensagem de A contendo (t, O) e então envia para B uma mensagem solicitando o valor atual do relógio de simulação (de cada assinante), juntamente com uma cópia do objeto (O) e uma constante numérica C, configurada pela aplicação;

- Cada assinante de B recebe a mensagem e envia para S o valor R do relógio local ("turno" atual da simulação), bem como garante que não procederá com a simulação

localmente se o relógio local atingir (R + C);

- S recebe todos os R, escolhe o maior (R'), e envia mensagem para B solicitando a inserção do objeto em B no turno T, onde (T = R' + C);
- Cada assinante de B recebe a mensagem contendo T, "cancela" o limite local (R + C) do simulador local e, quando o simulador local atingir o turno T, insere o objeto O na sua cópia local do segmento.

A constante "C" citada acima, e que também é utilizada por outros algoritmos do modelo, é um valor fornecido pela aplicação que tenta se aproximar ao máximo do valor real do intervalo de tempo necessário para que a sincronização entre todos os assinantes seja feita. Se valor de "C" for muito baixo, os assinantes "antigos" (ou seja, que já estavam presentes no grupo quando foi iniciada a inclusão do novo assinante) irão acabar bloqueando a simulação local para que a barreira de sincronização seja satisfeita. Para evitar que, durante uma assinatura (ou desassinatura, como será visto na seção seguinte), os jogadores que já estão jogando sejam prejudicados, o valor de "C" deve ser estimado para um valor mais elevado do que o tempo médio necessário para que o servidor calcule o turno em que ocorrerá a sincronização dos relógios, o que depende principalmente do tempo de resposta das conexões entre os clientes e o servidor.

Porém, o valor de "C" não pode ser muito grande, pois isto pode resultar em um aumento sensível no tempo necessário para a execução do algoritmo. No caso de transferência de objetos, pode resultar em um intervalo muito grande para que um objeto cruze a "fronteira" entre dois segmentos. No protótipo do FreeMMG (discutido no capítulo 4), valores de C que foram utilizados variaram entre 4 e 30 turnos em uma simulação que executa 10 turnos por segundo (ou seja, entre 0,4 e 3 segundos de tempo real).

3.5 Cálculo de áreas de interesse e notificação de presença

Quem conhece e aplica as regras relativas ao cálculo da área de interesse de cada cliente é o servidor. Ou seja, o cliente não pode assinar um segmento sem a prévia autorização do servidor. Porém, um cliente pode estar sobrecarregado a ponto de não conseguir assinar um segmento, mesmo que o cliente esteja autorizado pelo servidor para assinar aquele segmento. Um cliente pode estar sobrecarregado se, por exemplo, a sua conexão de rede não suportar mais uma assinatura de segmento, visto que cada assinatura adicional impõe mais trocas de mensagens com outros assinantes.

O servidor pode implementar qualquer critério de autorização de assinatura. Porém, o servidor precisa de certas informações para que seja possível tomar decisões relativas à área de interesse de cada cliente. Por exemplo, saber se um jogador está "presente" em um segmento é uma informação essencial para que o servidor implemente a regra "default" de cálculo de área de interesse. Um jogador está "presente" em um segmento se possui, no mínimo, um objeto no segmento.

Como o servidor, a princípio, não tem uma informação atualizada sobre quais objetos estão presentes em cada segmento, esta informação deve ser fornecida pelos assinantes. O modelo FreeMMG prevê duas mensagens replicadas enviadas pelos assinantes do segmento para o servidor, para notificar o servidor sobre alterações relativas à "presença" dos jogadores no segmento:

- Mensagem "incluir presença (X)" (ilustrada, no seu contexto, pela figura 3.6): mensagem replicada segmento-servidor, enviada quando é adicionado o primeiro objeto

no segmento cujo dono é o jogador X;

- Mensagem “excluir presença (X)”: mensagem replicada segmento-servidor, enviada quando é removido o último objeto do segmento cujo dono é o jogador X.

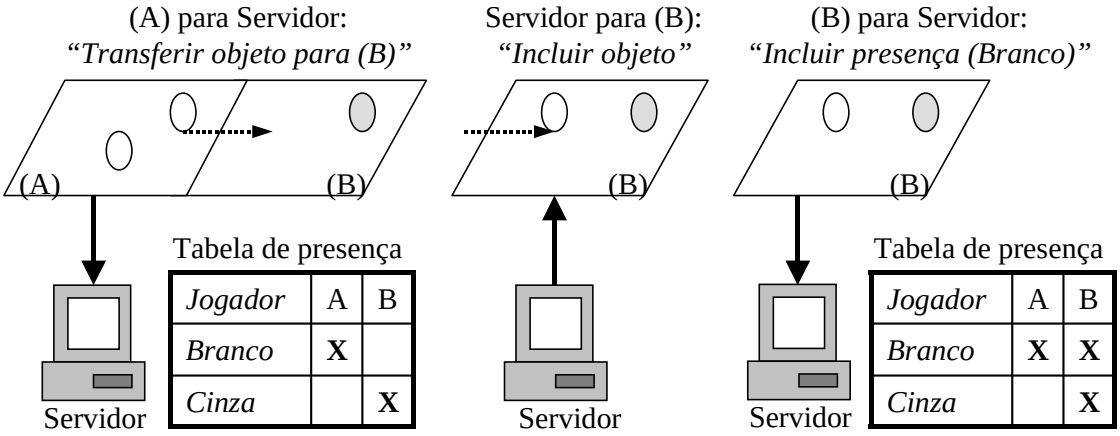


Figura 3.6: Mensagem “incluir presença” enviada após a movimentação de um objeto

A figura 3.6 ilustra o contexto em que se insere a mensagem do tipo “incluir presença”. Na figura, é importante observar que as setas entre os segmentos e o servidor são, na verdade, mensagens replicadas entre todos os assinantes do segmento e o servidor.

Estas mensagens de inclusão e remoção de presença permitem a implementação do critério “default” de determinação de áreas de interesse, bem como de alguns critérios mais complexos, como permitir que um cliente assine segmentos onde haja presença sua ou de qualquer um de seus aliados. Uma implementação do modelo FreeMMG deve, idealmente, permitir que novas mensagens de notificação sejam especificadas, de forma a permitir critérios mais complexos de determinação de áreas de interesse.

3.6 Segmentos inativos

Em uma situação ideal, uma rede FreeMMG deve possuir clientes suficientes para que todos os segmentos do mundo sejam simulados em tempo real. Isto seria ideal pois, em geral, um “mundo virtual” é projetado para que todas as suas “partes” sejam afetadas de forma igual com a passagem do tempo de simulação.

Porém, na prática, o “mundo virtual” deverá possuir segmentos que não serão do interesse de nenhum jogador, ao menos temporariamente. Um “segmento inativo” é um segmento do mundo virtual que não possui assinantes interessados e, portanto, fica armazenado no servidor (provavelmente em disco). O segmento inativo não é simulado, de forma que todos os objetos que estão localizados nele não serão modificados até que um grupo de assinantes se torne interessado no segmento.

Dependendo das regras que regem o mundo virtual, esta restrição pode não causar efeitos indesejáveis. Mas se, por exemplo, as regras estabelecem que certos objetos são modificados mesmo sem a intervenção de um jogador, é possível que a inatividade de segmentos seja um problema significativo. Por exemplo, se a aplicação modela um jogo

onde existem “recursos naturais” que se regeneram com o tempo (por exemplo, vegetação que fornece madeira para os jogadores), a inatividade de um segmento pode causar descontinuidades (figura 3.7) ou mesmo favorecer alguns jogadores.

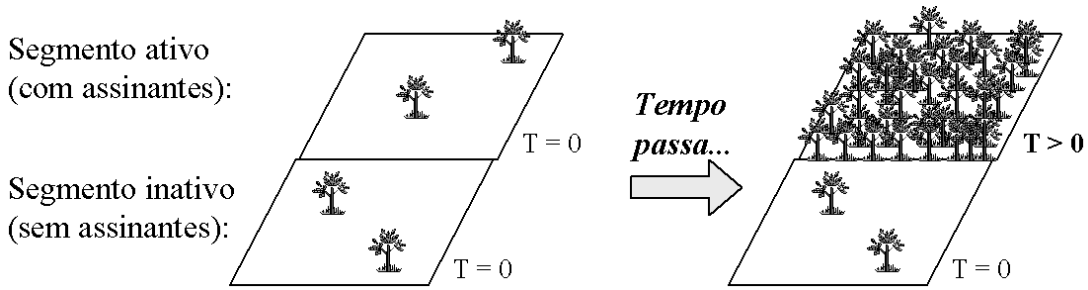


Figura 3.7: Exemplo de descontinuidade causada por segmentos inativos, onde “T” representa o tempo de simulação de cada segmento)

Para evitar este tipo de situação, é necessário que o servidor forneça assinantes “extras” para os segmentos que estão sem clientes interessados. Como será visto nas seções seguintes, um assinante “extra” pode ser qualquer cliente que, em princípio, não está interessado no segmento em questão, mas que possui os recursos necessários (poder de processamento e rede) para atuar como assinante. Estes assinantes podem ser alocados entre os próprios clientes que estão jogando ativamente, mas com recursos sobrando, quanto clientes dedicados, oriundos de uma iniciativa própria de computação distribuída similar a outras como SETI@Home (ANDERSON et al., 2002) e Freenet (CLARKE, 2002).

Outra alternativa é fazer com que o próprio servidor realize a simulação dos segmentos inativos, visto que há a vantagem adicional de que as atualizações ao segmento não precisariam ser enviadas a nenhum cliente remoto (pois não há interesse). Porém, esta abordagem exigiria um maior poder de processamento do servidor, tornando-se mais um obstáculo para a escalabilidade do modelo. A simulação de centenas ou milhares de segmentos inativos, onde cada segmento pode possuir centenas de objetos, pode ser uma tarefa muito custosa, e um dos objetivos principais do modelo FreeMMG é a adoção de um lado servidor que possua os menores custos possíveis, de comunicação e de processamento.

3.7 Algoritmo de assinatura

Em uma rede FreeMMG, os grupos de assinantes de cada segmento são formados dinamicamente. Quando o servidor detecta que um cliente se torna “interessado” em um novo segmento do qual não é assinante, é iniciado o protocolo de “assinatura” do cliente ao novo segmento. O algoritmo (ou protocolo) de assinatura do FreeMMG é responsável por inserir um novo cliente em um grupo de clientes que já são “assinantes” de um segmento, de forma similar a outros mecanismos de gerência de grupos em modelos de suporte a interação entre participantes distribuídos (BHARAMBE; RAO; SESHAN, 2002; FIEDLER; WALLNER; WEBER, 2002; CAI et al., 2002; GREENHALGH; PURBRICK; SNOWDON, 2000; CARLSSON; HAGSAND, 1993).

Um caso especial do algoritmo de assinatura ocorre quando o segmento a ser assinado está inativo. Quando uma rede FreeMMG é iniciada, ou seja, quando um “mundo virtual” é criado e disponibilizado pela primeira vez pelo servidor, todos os segmentos são

gerados e armazenados como “inativos” no servidor. Quando um segmento está inativo, ou seja, sem nenhum assinante, a tarefa de alocar o primeiro assinante consiste em apenas fazer o upload, do servidor para o cliente, do estado atual do segmento (ou *snapshot* do segmento). Quando o primeiro cliente recebe a cópia, a princípio ele já é um assinante, e pode começar a executar a simulação, se os parâmetros de “replicação mínima” do servidor permitirem isso.

Excetuando-se este caso especial, o algoritmo de assinatura é composto de três passos:

- Formação das conexões *peer-to-peer* entre os assinantes antigos do grupo com o novo assinante;
- Sincronização entre os assinantes antigos e posterior envio de um instantâneo (*snapshot*) inicial para o novo assinante;
- Sincronização entre todos os assinantes (antigos e novo) para inserção do novo assinante no algoritmo de simulação replicada.

3.7.1 Formação das conexões

O primeiro passo compreende a formação da rede de conexões entre os assinantes para que seja possível a posterior execução do algoritmo de “simulação *peer-to-peer* replicada” (figura 3.8). As implementações atuais do FreeMMG utilizam conexões ponto-a-ponto TCP/IP entre cada assinante de um grupo, mas outro tipo de tecnologia poderia ser empregado como, por exemplo, alguma forma de multicast confiável (LIN; PAUL, 1996; CRONIN et al., 2002) ou um protocolo ponto-a-ponto utilizando UDP/IP com extensões para entrega confiável de mensagens (CECIN et al., 2004).

No caso de conexões ponto-a-ponto entre os clientes, o primeiro passo do algoritmo procede da seguinte forma, sendo “S” o servidor, “N” o novo assinante e “A” o conjunto de assinantes antigos:

- S envia mensagem coletiva para A com os detalhes (endereço IP, identificador, etc) de N;
- Após confirmação da mensagem coletiva (feita pelo envio de uma mensagem individual de confirmação a S, por parte de cada A), S envia para N a lista contendo detalhes de todos os A;
- N dispara conexão para todos os A;
- Cada A aceita a conexão remota de N (pois já recebeu, de S, os detalhes para a autorização) e, a seguir, envia mensagem para S confirmando a conexão;
- Para cada conexão aceita, N envia mensagem de confirmação para S;
- S aguarda o recebimento de todas as mensagens de confirmação (*2*tamanho(A)* mensagens).

Após a execução do primeiro passo, o novo assinante já possui conexões de rede configuradas com todos os assinantes antigos. Todos os clientes envolvidos sabem que as novas conexões são aprovadas pelo servidor, o que reduz as chances de que um cliente seja assinante de um segmento sem ser autorizado pelo servidor.

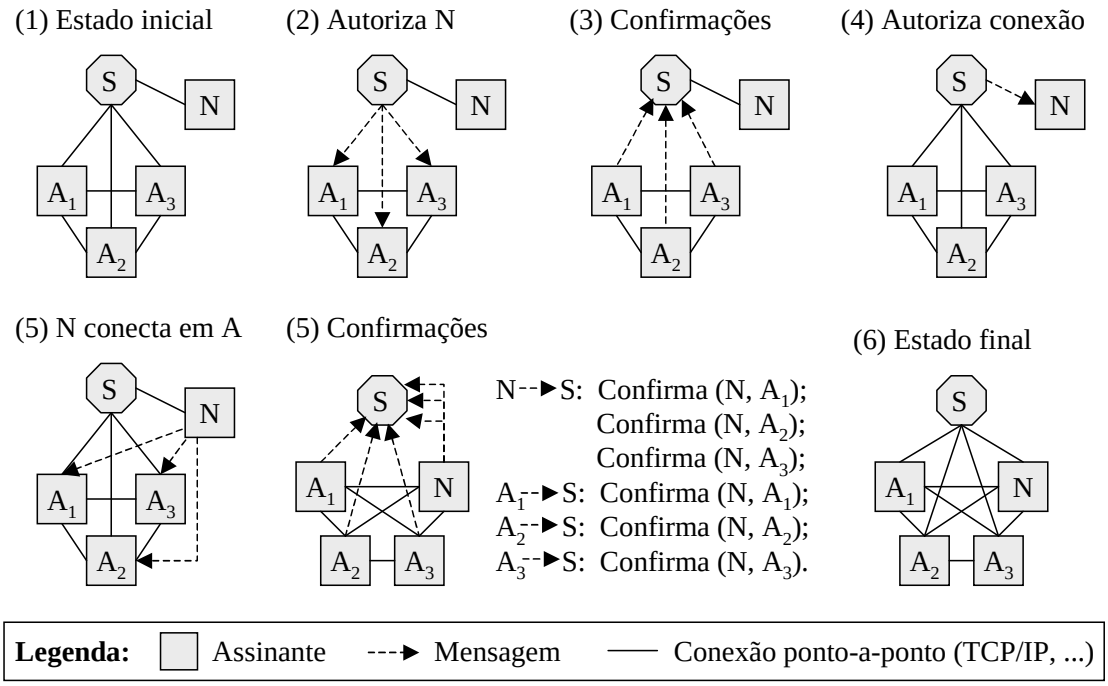


Figura 3.8: Exemplo de formação de conexões entre os membros de um grupo (A) e um novo membro (N) gerenciado pelo servidor (S)

3.7.2 Envio do *snapshot* ao novo assinante

O segundo passo envolve o envio de uma cópia “atual” estado do segmento para o novo assinante (figura 3.9). Este passo procede da seguinte forma (seguindo a notação do passo anterior):

- S envia para cada A mensagem solicitando o valor atual do relógio de simulação, juntamente com uma constante numérica C, configurada pela aplicação;
- Cada A recebe a mensagem e envia para S o valor R do relógio local (”turno” atual da simulação), bem como garante que não procederá com a simulação localmente se o relógio local atingir (R + C);
- S recebe todos os R, escolhe o maior (R’) e envia mensagem para cada A solicitando o envio do *snapshot* do segmento ao novo assinante no turno T, onde (T = R’ + C);
- Cada A recebe a mensagem, “cancela” o limite local (R + C) do simulador local e aguarda até a simulação chegar no turno T;
- Quando cada A atinge localmente T, envia para N uma mensagem contendo T, um “hash” da forma serializada do segmento, e a I-ésima parte da forma serializada do segmento onde I é a posição de cada A em uma lista que ordena totalmente os clientes da rede FreeMMG;
- Após o tempo de simulação T, cada A também passa a enviar para N todos os comandos gerados pelo jogador local, para que N seja capaz de atualizar o *snapshot*;

- N recebe todas as mensagens de A e restaura localmente uma cópia completa ou *snapshot* do segmento no tempo de simulação T (utilizando os valores de "hash" fornecidos para detectar erros) e então envia mensagem para S confirmando o recebimento da cópia.

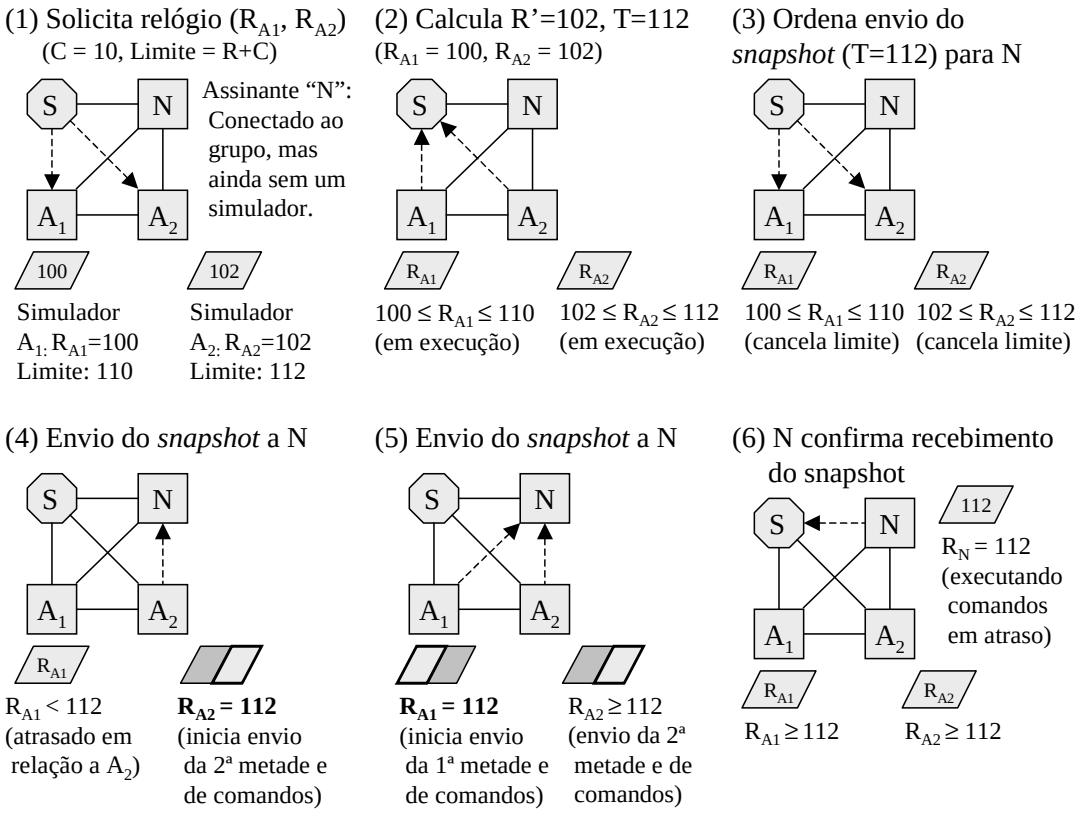


Figura 3.9: Exemplo de envio de *snapshot* para o novo assinante

Após a execução do segundo passo, o novo assinante tem uma cópia do estado do segmento para um tempo de simulação "T". Porém, enquanto o novo assinante estava fazendo o download do *snapshot*, os assinantes antigos estavam continuando com a simulação. Portanto, quando o novo assinante recebe o *snapshot*, este já está obsoleto. O novo assinante deve então processar os comandos "atrasados" localmente, de forma acelerada (ou seja, desconsiderando o intervalo de tempo real entre cada passo da simulação), até que não haja mais nenhum comando para ser processado.

3.7.3 Sincronização do novo assinante com os assinantes "antigos"

Quando o novo assinante consegue processar todos os comandos pendentes, o terceiro passo do algoritmo é executado para sincronizar o novo assinante com os assinantes antigos (figura 3.10). O terceiro passo procede da seguinte forma:

- N envia mensagem para S avisando que já conseguiu processar todas as mensagens acumuladas localmente;

- S envia para cada A mensagem solicitando o valor atual do relógio de simulação, juntamente com uma constante numérica C, configurada pela aplicação;

- Cada A recebe a mensagem e envia para S o valor R do relógio local, bem como garante que não procederá com a simulação localmente se o relógio local atingir (R + C);
- S recebe todos os R, escolhe o maior (R'), e envia mensagem para N e A notificando que o primeiro turno (tempo de simulação) em que N será considerado um assinante será (R' + C) e, a partir deste momento, S já considera N um assinante do segmento;
- Cada A recebe a mensagem de S, "cancela" o limite local (R + C) do simulador e configura o simulador para esperar comandos de N no turno (R' + C);
- N recebe a mensagem de S e configura o simulador local para enviar comandos para os outros assinantes a partir do turno (R' + C).

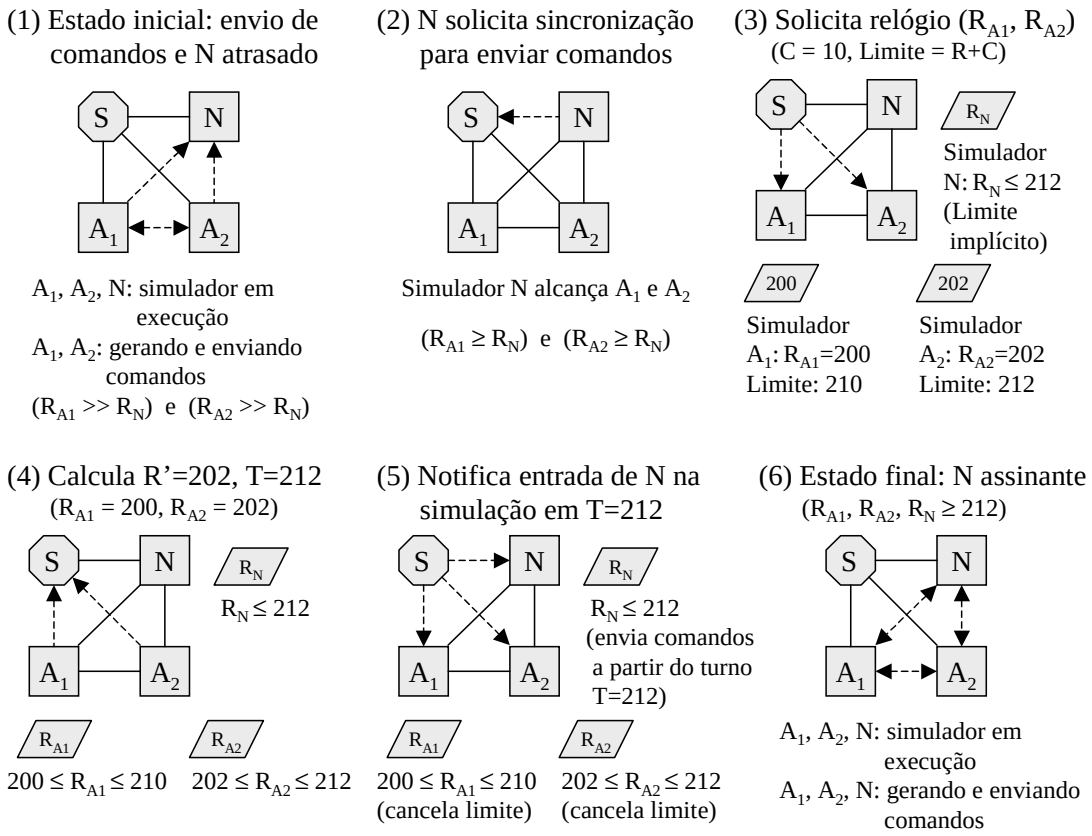


Figura 3.10: Exemplo de sincronização (habilitação da geração e envio de comandos) do simulador do novo assinante

3.7.4 Avaliando o desempenho do algoritmo de assinatura

Os critérios mais importantes para a avaliação do desempenho do algoritmo de assinatura são:

- Não-intrusão, ou seja, não afetar negativamente o funcionamento dos clientes que já são assinantes;
- Menor tempo total de execução, do ponto de vista do "novo assinante".

O algoritmo foi projetado especificamente para causar um impacto reduzido na "experiência de jogo" dos jogadores que já são assinantes do segmento (evitar travadas, lentidão, etc). Porém, este ganho possui como custo a imposição de (potencialmente) um maior tempo total de execução do algoritmo. Enquanto o algoritmo está executando, o "novo assinante" está, a princípio, sendo prejudicado no jogo, pois o servidor já lhe garantiu o direito de interagir com um segmento do mundo virtual mas, por causa do algoritmo utilizado na sincronização de novos assinantes, o jogador ainda não pode interagir com aquele segmento. Isto pode resultar em, por exemplo, objetos se tornando vulneráveis (pois não podem ser comandados pelo jogador que os possui) por um certo período de tempo.

Esta desvantagem pode ser contornada através do aumento das áreas de interesse dos jogadores. Ao invés de utilizar um critério de cálculo de área de interesse que exija que um cliente já possua objetos em um segmento para que seja assinante deste segmento, pode-se utilizar um critério que exija apenas um objeto em um segmento adjacente (figura 3.11). Desta forma, quando um cliente estiver movendo um de seus objetos para um segmento adjacente, é bastante provável que ele já seja assinante deste segmento. Porém, esta técnica possui duas possíveis desvantagens que precisam ser consideradas.

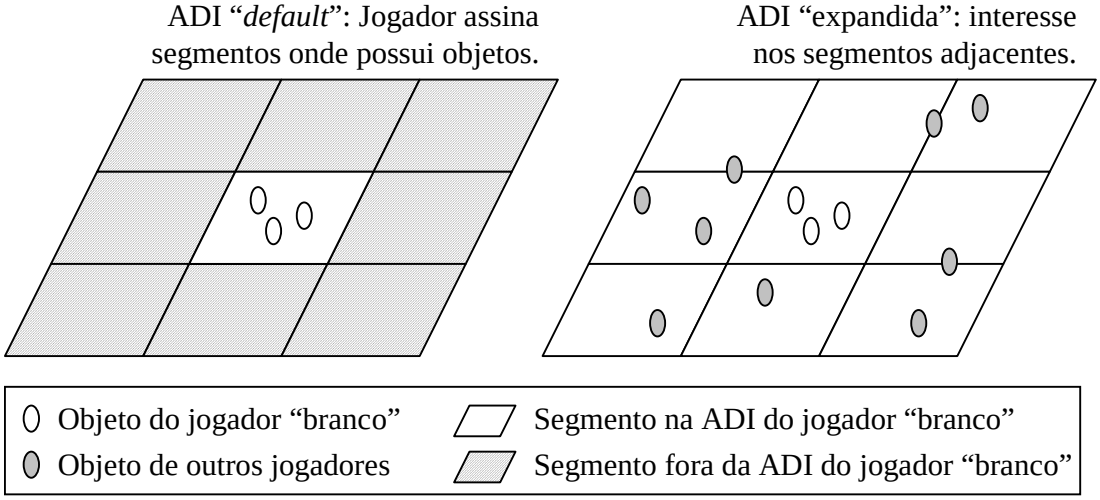


Figura 3.11: Exemplo de aplicação dos critérios "default" e "expandido" de cálculo de ADI (área de interesse) de um jogador

A primeira desvantagem é que isto pode reduzir o "fator surpresa" de um jogo pois, em princípio, se um cliente é assinante de um segmento, ele sabe de tudo o que está ocorrendo, em tempo real, naquele segmento. Se os segmentos, em uma determinada aplicação, já são configurados para serem grandes, isto significa que o "raio" de visão de um objeto pode ser muito grande. Jogos de estratégia como Age of Empires resolvem este problema fazendo com que o programa cliente do jogo "esconda" informação do jogador. Porém, se o cliente do jogo é adulterado, o jogador passa a ter a informação completa. Este problema não parece ter uma solução definitiva, visto que o protocolo de rede exige que a máquina do jogador (o "cliente" do jogo), em última análise, tenha a informação completa do jogo (ou, no caso do FreeMMG, do segmento).

A segunda desvantagem é que esta técnica pode aumentar significativamente o número de segmentos a serem assinados por um cliente. Visto que cada assinatura exige mais

poder de processamento e recursos de rede em um cliente, este critério “expandido” para determinação de áreas de interesse pode causar um esgotamento bem mais rápido dos recursos dos clientes, que acabariam, de qualquer maneira, tendo que optar (por iniciativa própria, ou por decisão do servidor) por assinar apenas os segmentos mais próximos dos seus objetos. Este problema pode ser mitigado através de uma otimização que será discutida na seção 3.17.

Uma maneira de mitigar tanto o problema da redução do “fator surpresa”, quanto o problema do excesso de assinaturas, é fazer com que um cliente possa assinar os segmentos “adjacentes” apenas se o jogador em questão possuir um objeto próximo da “fronteira” entre os dois segmentos (figura 3.12). Se os segmentos do mundo virtual forem suficientemente grandes, o ganho pode ser significativo. Para implementar este critério, algumas novas “mensagens replicadas” são necessárias, para que o grupo de assinantes de um segmento notifique o servidor sobre a “presença” de um jogador não só no segmento como um todo, mas também notifique a “presença” do jogador nas fronteiras do segmento.

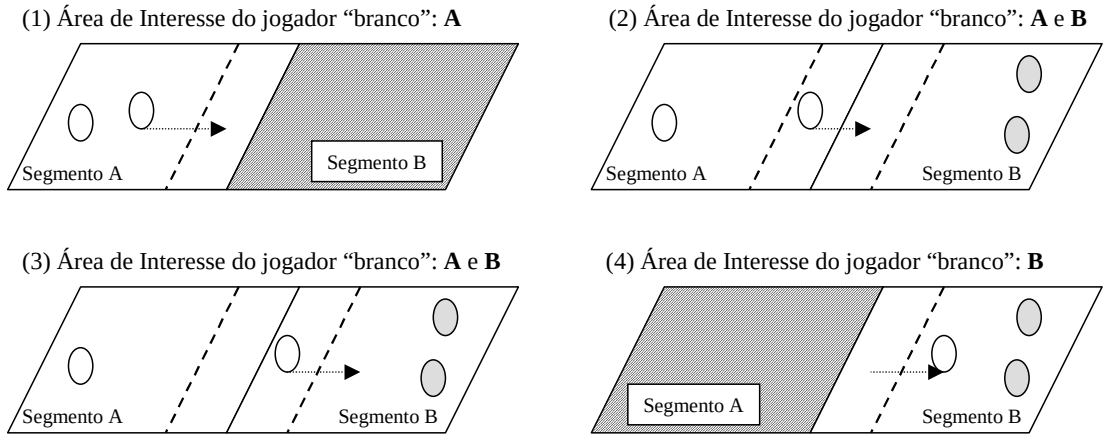


Figura 3.12: Expansão da ADI por objeto próximo à borda de um segmento

Por fim, é importante citar que o (possível) paralelismo do algoritmo de assinatura não foi explorado. A implementação realizada do algoritmo apenas funciona corretamente se, para cada segmento, somente uma nova assinatura for negociada de cada vez. Porém, pode-se dizer com segurança que ao menos o primeiro passo do algoritmo pode ser implementado de forma concorrente, ou seja, para vários “novos assinantes” de um mesmo segmento, ao mesmo tempo. Isto é possível pois a primeira parte do algoritmo de assinatura é apenas um procedimento de estabelecimento de conexões, sem necessidade de sincronização ou ordem na execução dos seus passos.

3.8 Algoritmo de desassinatura

Na rede FreeMMG, os clientes que não estão mais interessados em um segmento devem executar um protocolo específico de “desassinatura de segmento” (figura 3.13). O algoritmo é composto dos seguintes passos, onde S é o servidor, D é o assinante que está sendo retirado do segmento, e A o conjunto total de assinantes do segmento (ou seja, D pertence a A):

- D decide que quer cancelar sua assinatura e envia mensagem para S com a solicitação (passo opcional);

- S decide que o assinante D deve ser retirado do seu segmento (por solicitação de D ou outra razão, como mudança na área de interesse de D detectada por S) e envia mensagem para cada A solicitando o valor atual do relógio de simulação, juntamente com uma constante numérica C, configurada pela aplicação;
- Cada A recebe a mensagem e envia para S o valor R do relógio local, bem como garante que não procederá com a simulação localmente se o relógio local atingir (R + C);
- S recebe todos os R, escolhe o maior (R'), e envia mensagem para cada A informando que a assinatura de D será cancelada no turno T, onde (T = R' + C);
- Cada A recebe a mensagem de S, "cancela" o limite local (R + C) do simulador e configura o simulador local para esperar comandos de D apenas até o turno (R' + C);
- Quando cada assinante atinge o turno (R' + C) localmente, desfaz as conexões com D (se é o próprio D, desfaz as conexões com todos os outros assinantes), e envia mensagem para S notificando a conclusão da desassinatura;
- S espera o recebimento de todas as mensagens de conclusão de desassinatura para considerar D efetivamente retirado do segmento.

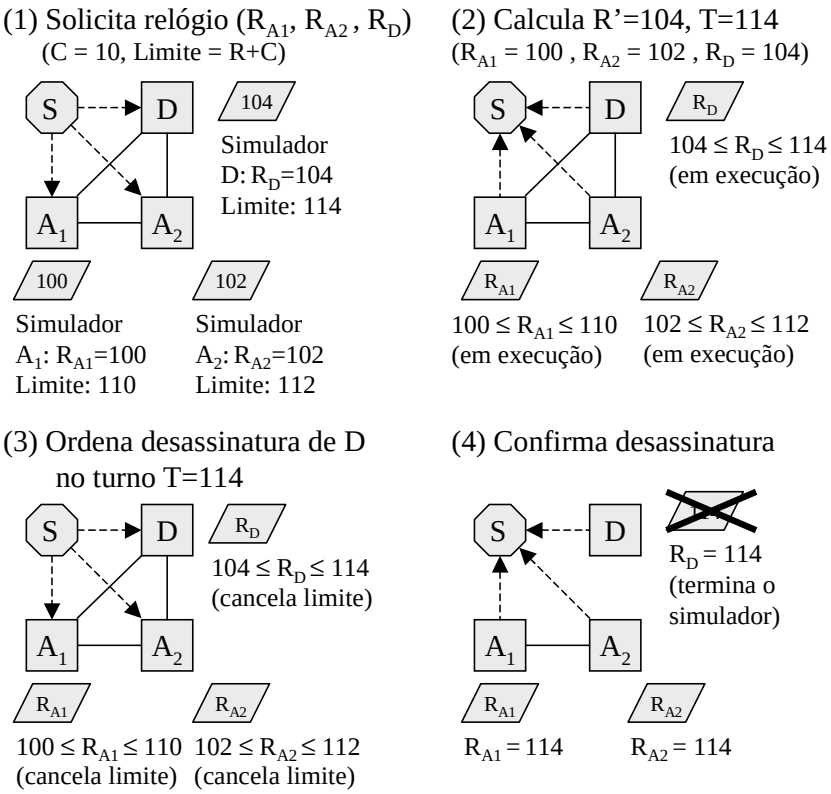


Figura 3.13: Exemplo de execução do algoritmo de desassinatura

Durante a desassinatura (assim como durante a assinatura), é necessário que seja feita uma sincronização entre os assinantes, o que pode causar uma suspensão da simulação,

causada pelos “limites locais”, durante a negociação da desassinatura. Para que o cancelamento de assinatura seja feito de forma a evitar suspensões e prejudicar a quem já é assinante do segmento, é necessário que haja um acordo entre todos os assinantes sobre qual será o último turno em que D será considerado assinante. Porém, enquanto este valor está sendo negociado, a simulação prossegue, em tempo real. Para que o acordo não se torne obsoleto antes de ser realizado, é utilizada a constante auxiliar “C”, discutida na seção anterior para o algoritmo de assinatura, que garante que a simulação não será executada locamente após um certo limiar, que é o valor mínimo a ser decidido pelo acordo.

Em princípio, vários protocolos de desassinatura podem executar em paralelo, para um mesmo segmento, pois a remoção, por desassinatura, de qualquer um dos assinantes durante a execução do protocolo não causa nenhuma perda de informação. Isto ocorre pois o assinante participa da negociação de desassinatura informando um valor para o “turno” a ser negociado que lhe interessa, ou seja, que evita que ele próprio saia de sincronia. Caso um assinante A saia do grupo durante a negociação de uma outra desassinatura, isto não prejudica os outros assinantes, pois o pior que pode acontecer é que a restrição imposta pelo assinante A seja utilizada, desnecessariamente, para o cálculo do turno que satisfaz a todos os assinantes. Porém o algoritmo de desassinatura, em princípio, não pode executar de forma concorrente ao algoritmo de assinatura, no mesmo segmento.

3.8.1 Desativação de segmentos

O algoritmo de desassinatura descrito acima trata apenas do caso em que existe interesse em continuar a execução da simulação após a remoção do assinante em questão. Isto ocorre quando o assinante removido não é o último assinante interessado, ou quando a implementação do modelo busca alocar assinantes “extras” para que os segmentos não se tornem inativos (como discutido na seção 3.6). Caso o servidor decida que a simulação do segmento não deve prosseguir por falta de assinantes interessados, o segmento é desativado. O protocolo de desativação pode ser executado no lugar do procedimento de desassinatura de um assinante, quando a remoção deste satisfaz o critério do servidor para desativação de segmentos.

A desativação de um segmento procede da seguinte forma:

- Servidor envia mensagem para todos os assinantes solicitando o valor atual do relógio de simulação, juntamente com uma constante numérica C, configurada pela aplicação;
- Cada assinante recebe a mensagem e envia para o servidor o valor R do relógio local, bem como garante que não procederá com a simulação localmente se o relógio local atingir (R + C);
- Servidor recebe todos os R, escolhe o maior (R’), e envia mensagem para cada assinante informando que a simulação deve parar no turno T, onde (T = R’ + C);
- Cada assinante recebe a mensagem, “cancela” o limite local (R + C) do simulador e, quando a simulação atinge localmente o turno T, envia para o servidor uma cópia do segmento no tempo de simulação T, e a seguir fecha todas as conexões com os outros assinantes do grupo;
- Servidor compara as versões recebidas e armazena a “melhor” (versão com mais assinantes concordantes, versão mais recente, ou uma combinação destes dois critérios)

localmente como cópia inativa.

Os últimos dois passos podem também ser implementados de forma que cada assinante envie apenas uma parte do segmento para o servidor, juntamente com um “hash” da cópia completa (ou um hash de cada uma das partes). Também é importante destacar que, caso o servidor decida que as cópias obtidas não sejam confiáveis (por exemplo, se cada assinante possui uma cópia diferente) então uma “falha fatal” ocorre e um dos procedimentos descritos na seção 3.14 é adotado para realizar a recuperação.

3.9 Número mínimo de assinantes e assinantes aleatórios

O “trapaceiro” é um jogador, em um jogo distribuído, que quer obter uma vantagem ilegal no jogo sem ser detectado. Existem várias maneiras de se trapacear em um jogo distribuído, bem como existem várias técnicas para prevenção, detecção e correção de trapaças, que dependem principalmente do modelo de distribuição adotado pelo jogo. Para o modelo FreeMMG, o pior tipo de trapaça, que é combatido através da replicação de cada segmento, é a alteração arbitrária do estado do jogo.

A replicação de estado, como discutido anteriormente, garante um certo nível de resistência contra jogadores trapaceiros pois, em princípio, um trapaceiro só poderá alterar a sua cópia local do estado do jogo, não podendo afetar as outras réplicas. Mesmo na presença de “grupos de trapaceiros” agindo de forma coordenada, a existência de replicação ao menos torna o desenvolvimento de software dos programas clientes “maliciosos” mais complexo.

Porém, o modelo apresentado até aqui permite que apenas um cliente seja assinante de um segmento em um determinado momento. Por exemplo, se um segmento possuir apenas objetos de posse de um único jogador, pelo critério “default” de cálculo de área de interesse, este jogador seria o único assinante do segmento. Este jogador poderia facilmente trapacear no jogo pois ele possui a única cópia ativa do segmento. As alterações ilegais deste cliente seriam simples alterações locais e sem correr o risco de uma outra réplica “honesta” avisar o servidor de que a simulação saiu de sincronia.

Para evitar esta situação insegura, uma implementação do modelo FreeMMG deve expor uma interface para a aplicação que permita a especificação de um número “mínimo” de assinantes (réplicas ativas) para que a atualização do segmento seja autorizada pelo servidor. Após especificado este valor, a implementação do FreeMMG deve inserir assinantes “extras” nos segmentos que possuírem menos assinantes do que o valor configurado. Os assinantes extras ou “assinantes aleatórios” são clientes que não estão interessados no segmento mas que participam da simulação do mesmo para ajudar a dificultar a ação de trapaceiros, bem como reduzir as chances de ocorrência de “falhas fatais”, que serão discutidas nas seções seguintes.

Assinantes “aleatórios” poderiam ser obtidos através de alocação dos clientes de jogadores ativos que estão com recursos sobrando, ou então através de clientes voluntários recrutados através de uma iniciativa de computação distribuída similar a SETI@Home ou Freenet. A participação como assinante extra em uma rede FreeMMG poderia ser recompensada através de algum bônus no jogo, incentivando assim que os jogadores executem o assinante extra quando não estão jogando ou que peçam para seus “contatos” na Internet que participem da iniciativa para ajudar o seu desempenho no jogo. Adicionalmente, assinantes extras podem contribuir para resolver o problema da “inatividade seletiva” do mundo virtual, que foi discutida na seção 3.6.

3.9.1 Problema da “maioria trapaceira”

É importante destacar que, mesmo com a adoção de assinantes aleatórios, não é possível garantir que não ocorrerão alterações ilegais. Isto ocorre porque mesmo que os assinantes “aleatórios” sejam escolhidos ao acaso, é possível que certos assinantes cadastrados como voluntários para assinaturas extras, junto ao servidor, sejam clientes maliciosos que estão procurando ativamente a oportunidade de serem alocados junto a outros trapaceiros. A única solução que resolve de fato o problema da trapaça por alteração de estado em uma rede FreeMMG é a exigência de que assinantes confiáveis, ou seja, alocados no “lado servidor”, participem da simulação de cada segmento ativo. Se cada segmento ativo possuir ao menos um assinante que executa no servidor, a execução de trapaças do tipo “alteração ilegal de estado” se torna impossível, pois a cópia do assinante confiável será sempre utilizada em caso de disputas. Por outro lado, a simulação irá se tornar, na prática, centralizada, o que torna o servidor novamente um gargalo de processamento e de comunicação.

3.10 Detecção e tipos de falhas de clientes (individuais)

Um cliente é considerado “falho” pelo servidor quando este se desvia do seu comportamento esperado (WEBER; JANSCH-PÔRTO; WEBER, 1996). Quando um cliente falha, todas as suas assinaturas falham, ou seja, todos os grupos (ou segmentos) de que o cliente estava participando são afetados negativamente, de uma forma ou de outra. Nesta seção são relacionadas as situações que fazem o servidor considerar um cliente como “falho”. As falhas de clientes (ou falhas de assinantes) causam as falhas de grupo (ou falhas de segmento) que serão discutidas na seção seguinte. Neste contexto, os principais tipos de falhas detectáveis de clientes são os seguintes:

3.10.1 Timeout, quebra de conexão ou violação de protocolo detectado pelo servidor

Neste caso, o servidor detecta diretamente a ocorrência da falha do cliente através da sua conexão de rede com o mesmo. Porém, é importante destacar que a falha não necessariamente indica que o cliente é malicioso. Pelo contrário, pode-se considerar um cenário onde todos os outros assinantes de um segmento unem-se para “sabotar” um dos assinantes, que acabará falhando.

3.10.2 Timeout, quebra de conexão ou violação de protocolo detectado por um cliente-assinante

Similar ao caso anterior, porém, quem detecta a falha de um cliente é outro cliente, que então avisa o servidor. Este caso é problemático pois o servidor não pode “acreditar” cegamente em algo relatado por apenas um cliente, mas também não pode descartar a possibilidade de que o cliente está sendo honesto. De qualquer forma, os grupos (segmentos) assinados pelos clientes envolvidos são considerados falhos, como será visto na seção seguinte.

3.10.3 Inconsistência detectada nas entradas enviadas pelos clientes

Um assinante pode fazer com que a simulação de um segmento saia de sincronia se, durante a execução do algoritmo de simulação, ele enviar comandos diferentes para cada outro assinante, em um mesmo turno. Isto faz com que os assinantes utilizem entradas

diferentes para a “simulação replicada”, fazendo com que o jogo saia de sincronia.

Para detectar este tipo de falha (que pode ser intencional ou não), uma implementação de um assinante para a rede FreeMMG deve fazer com que cada assinante envie não só os seus comandos para um turno T, mas também um “hash” de todas as entradas que recebeu durante a execução do turno anterior (T - 1). Desta forma, se um ou mais assinantes tentarem fornecer comandos diferentes para um mesmo turno, durante a negociação do próximo turno esta situação pode ser detectada, como é ilustrado pela figura 3.14.

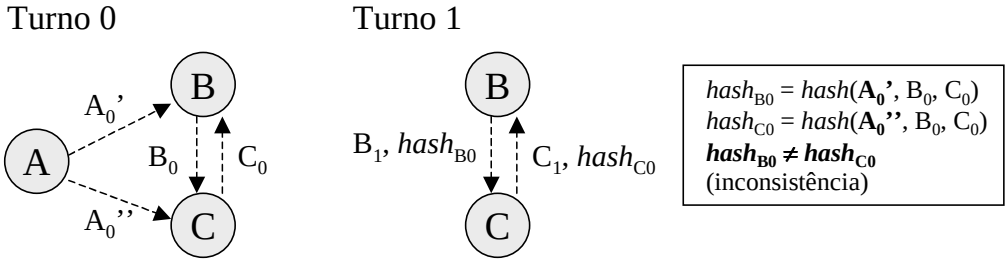


Figura 3.14: Exemplo de detecção de inconsistência nas entradas enviadas (simuladores sem “look-ahead”)

Neste caso, qualquer nodo “honesto” pode avisar o servidor sobre a falha. Porém, como discutido no item anterior, o servidor não tem como saber se a notificação é honesta ou não. Mesmo que quase todos os nodos façam a mesma notificação, o servidor não tem como saber se a maioria é que está mentindo de comum acordo. De qualquer forma, os grupos (segmentos) assinados pelos clientes envolvidos são considerados falhos, como será visto na seção seguinte.

3.11 Falhas de grupo: parciais e fatais

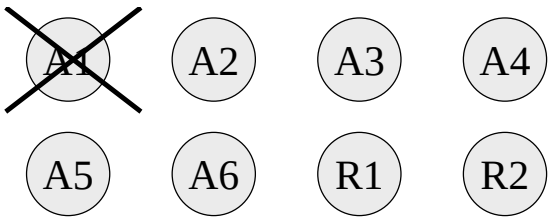
Quando o servidor detecta que um ou mais assinantes de um grupo de assinantes de um segmento falham, seja qual for a razão, o servidor marca o segmento como “falho” e dispara um procedimento de recuperação do segmento para um estado válido, através do tratamento da falha. Porém, antes de serem apresentados os mecanismos de recuperação de falhas de grupo (ou “falhas de segmento”), primeiro deve-se distinguir os dois tipos de falhas de grupo: falhas parciais e falhas fatais.

3.11.1 Falha parcial

Uma falha de grupo é considerada “parcial” pelo servidor se, após a falha do grupo, parte das cópias do estado do segmento, em posse dos assinantes que não falharam, ainda puderem ser utilizadas na recuperação da falha do grupo. Em geral, o critério para decidir se as cópias restantes são confiáveis é relativo ao maior número de assinantes restantes que ainda concordam entre si sobre a versão mais atual do estado do segmento. Se este número satisfaz um certo limiar, a falha é considerada “parcial”. Um bom exemplo de falha parcial envolve um cenário com um número elevado de assinantes (por exemplo, oito assinantes) onde apenas um assinante “falha” individualmente e todos os outros assinantes concordam com uma versão do estado do segmento (figura 3.15).

O limiar citado deve ser especificado tanto para assinantes “normais” quanto para assinantes “aleatórios”. Como discutido anteriormente, a aplicação pode determinar um número mínimo de assinantes “aleatórios” em cada segmento. Da mesma forma, a aplicação pode determinar também um número mínimo de assinantes “aleatórios” no conjunto de réplicas confiáveis após uma falha. Se este número não for satisfeito, a falha não é considerada parcial.

O objetivo deste número mínimo de nodos aleatórios após uma falha é o de evitar uma situação onde uma maioria de assinantes “normais”, que podem compor um grupo de clientes “trapaceiros”, resolvem eliminar os assinantes aleatórios (que provavelmente compõe a minoria dos assinantes) e forçar um cenário de recuperação de falha parcial onde seja possível que os trapaceiros forcem a adoção de uma versão incorreta do estado do segmento. Ao estabelecer que o grupo de réplicas restantes, após uma falha, componha tanto assinantes “normais” quanto assinantes “aleatórios”, se reduz a probabilidade de que a restauração da uma cópia ativa, nos assinantes, seja explorada para execução de uma trapaça.



Número de assinantes após a falha = **7** (A2, A3, A4, A5, A6, R1, R2)
(Todos os assinantes restantes concordam entre si sobre um estado atualizado do segmento)
Número de assinantes “aleatórios” após a falha = **2** (R1, R2)
Limiar: número mínimo de assinantes após uma falha = **5 (≤ 7)**
Limiar: número mínimo de assinantes aleatórios após uma falha = **1 (≤ 2)**

Resultado: **Falha parcial**

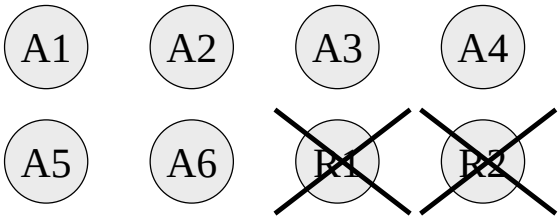
Figura 3.15: Exemplo de falha parcial em um grupo de assinantes

Por fim, se houver um assinante “confiável” no segmento, ou seja, se houver ao menos um assinante alocado no lado servidor, a falha do segmento é considerada sempre parcial. Porém, como foi discutido, este cenário exigiria que o servidor se comunicasse em tempo real com todos os clientes e realizasse a computação da simulação de todos os segmentos ativos, anulando a escalabilidade conferida pela simulação descentralizada do “mundo virtual”.

3.11.2 Falha fatal

A “falha fatal” de um grupo de assinantes ocorre quando o servidor decide que as cópias restantes (se alguma) do estado do segmento simulado pelo grupo não são confiáveis. Neste cenário, toda a informação de estado do jogo que está nos clientes deve ser descartada, sendo que o servidor fica responsável por utilizar a informação que tem armazenado localmente (por exemplo, uma cópia antiga do segmento) para restaurar o estado do jogo para uma versão confiável. Um bom exemplo de falhas fatal envolve um cenário de falha de grupo onde apenas um entre vários assinantes não falhou. Outro bom exemplo envolve um cenário onde restam todos os assinantes menos um (o que causou

a falha do grupo), mas cada um dos assinantes restantes possui uma cópia diferente do estado do segmento. A figura 3.16 ilustra um cenário de “falha fatal” que é causado pela insuficiência de “assinantes aleatórios” após a falha.



Número de assinantes após a falha = **6** (A1, A2, A3, A4, A5, A6)
(Todos os assinantes restantes concordam entre si sobre um estado atualizado do segmento)
Número de assinantes “aleatórios” após a falha = **0**
Limiar: número mínimo de assinantes após uma falha = **5 (≤ 6)**
Limiar: número mínimo de assinantes aleatórios após uma falha = **1 (> 0)**

Resultado: **Falha fatal**

Figura 3.16: Exemplo de falha fatal em um grupo de assinantes

3.11.3 Diferenças entre falha parcial e falha fatal

É importante distinguir entre “falha parcial” e “falha fatal” de um grupo de assinantes pois o impacto de uma falha parcial no sistema pode ser significativamente menor do que o de uma falha fatal. Mesmo que o servidor tivesse informação completa sobre o estado de cada segmento, ou seja, mesmo que o servidor fosse assinante de cada segmento ativo, uma falha parcial ainda seria menos custosa de ser reparada, pois ao menos os assinantes que não falharam não precisariam obter do servidor uma nova cópia “correta” do estado do segmento, economizando transferências de estado de segmento. E se o servidor não possuir uma informação atualizada, ou seja, se possuir apenas uma cópia antiga (por exemplo, a última cópia inativa do segmento), a falha parcial evita a restauração de uma versão “antiga” do segmento e, portanto, evita que o tempo de simulação do segmento volte atrás no tempo. Isto é significativo pois se um dos simuladores em uma simulação distribuída volta atrás no tempo de simulação, então é possível que inconsistências na simulação global sejam introduzidas. O problema da inconsistência global no FreeMMG será discutido em detalhe nas seções seguintes.

3.12 Algoritmo inicial de recuperação de falhas

Quando há uma falha de grupo, a primeira coisa que o servidor FreeMMG deve fazer é decidir qual o procedimento de recuperação que será adotado. Para isso, o servidor deve decidir se a falha é “parcial” ou “fatal”. Antes de saber qual o tipo de falha, o servidor executa um “protocolo inicial” de recuperação de falhas, composto dos seguintes passos:

- O servidor descarta o estado de todos os nodos que “falharam” (falha individual do assinante, falha por timeout, etc). Porém, como discutido na seção anterior, nem sempre o servidor terá certeza sobre qual assinante realmente falhou, como é o caso em que um ou mais assinantes avisam o servidor sobre a falha de outros assinantes.

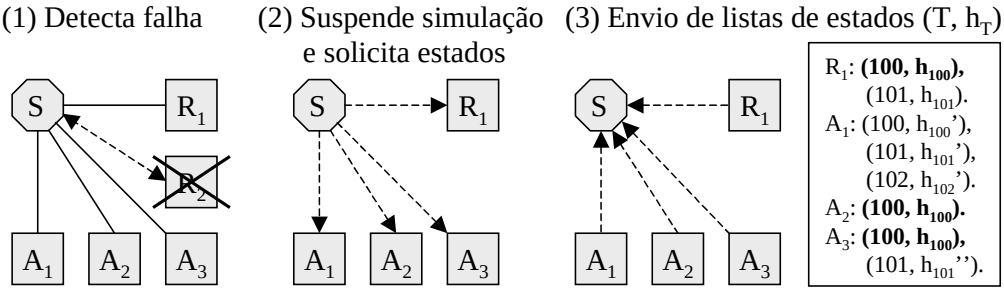
Em princípio, o servidor só descarta o estado, neste passo, de clientes que perderam a conexão com o servidor;

- O servidor verifica se a quantidade e tipos de clientes restantes satisfazem os critérios mínimos para falha parcial (supondo, neste ponto, que todos os clientes restantes concordam entre si em relação ao estado atual do segmento). Em caso afirmativo, o algoritmo prossegue para o passo seguinte. Caso contrário, o algoritmo determina a ocorrência de uma "falha fatal", e aplica-se um dos protocolos de recuperação de "falhas fatais" descritos na seção 3.14;

- O servidor envia mensagem para todos os assinantes (restantes) notificando a interrupção temporária da simulação do grupo e pedindo uma lista de estados recentes que o nodo possua no seu histórico (se algum);

- Cada assinante restante envia uma lista contendo tuplas (T, h) onde T é o "timestamp" de uma das cópias do segmento que está armazenada no histórico do assinante, juntamente com um "hash" ("h") calculado sobre o conteúdo binário da cópia;

- O servidor recebe as tuplas (T, h) de cada cliente e decide qual a tupla representa o estado mais confiável (que possui mais cópias dentre os assinantes, ou que melhor satisfaça os critérios para falha parcial). Se o conjunto de clientes que possui o estado escolhido como o "mais confiável" satisfizer os critérios mínimos para falha parcial, então o procedimento de recuperação de falha parcial é disparado (descrito na seção 3.13). Caso contrário, o algoritmo determina a ocorrência de uma "falha fatal", e aplica-se um dos protocolos de recuperação de "falhas fatais" descritos na seção 3.14;



(4) Servidor detecta "falha parcial" e escolhe a cópia "h_{100}" para a recuperação

Número de assinantes após a falha: **4** (A_1, A_2, A_3, R_1)
Número de assinantes aleatórios após a falha: **1** (R_1)
Estado escolhido (*timestamp*, *hash*): **(100, h_{100})**
Número de assinantes após a falha (que concordam com o estado escolhido): **3** (A_2, A_3, R_1)
Número de assinantes aleatórios após a falha (que concordam com o estado escolhido): **1** (R_1)
Limiar: número mínimo de assinantes (concordantes) após uma falha: **3 (≤ 3)**
Limiar: número mínimo de assinantes aleatórios (concordantes) após uma falha: **1 (≤ 1)**

Resultado: **falha parcial**.

Figura 3.17: Exemplo de execução do protocolo inicial de recuperação de falhas

A figura 3.17 mostra um exemplo da execução do protocolo inicial de recuperação de falhas. No exemplo da figura, o servidor (S) determina a ocorrência de uma falha parcial do grupo, pois os números mínimos de assinantes concordantes após a falha satisfazem, no exemplo, os limiares configurados pelo servidor. Após a execução deste protocolo inicial, um protocolo de recuperação de falhas específico para o tipo de falha do grupo (”parcial” ou “fatal”) é executado. Na seção 3.13 é detalhado o protocolo de recuperação de falhas parciais. Já na seção 3.14 são detalhados dois protocolos alternativos de recuperação de falhas “fatais”, sendo que cada um exige um tipo de suporte diferente para executar.

3.13 Recuperação de falhas parciais

Quando uma falha de grupo é “parcial”, isto significa que o servidor considera que pelo menos um conjunto dos assinantes que não falharam são confiáveis e estão de acordo entre si. Em uma situação ideal, uma rede FreeMMG sofreria apenas de falhas não-intencionais e esporádicas, como a perda de conexão de um único assinante, sendo que as falhas “fatais” raramente ocorreriam. Neste sentido, o objetivo do algoritmo de recuperação de falha “parcial” é o de prover um mecanismo de recuperação rápida, que tenha o menor impacto possível no andamento do jogo tanto para os assinantes do grupo afetado pela falha quanto para todos os outros assinantes da rede FreeMMG.

Após a detecção da falha parcial em um grupo, existirão assinantes em duas situações diferentes:

- Assinantes que possuem a cópia do “estado escolhido” pelo servidor como sendo o “mais confiável”;
- Assinantes que não possuem a cópia do “estado escolhido” e que precisam ser atualizados.

Portanto, o primeiro passo da recuperação consiste em passar uma cópia do “estado escolhido” pelo servidor para os assinantes que não possuem uma cópia. Isto pode ser feito de duas maneiras:

- Servidor obtém uma cópia, salva localmente, e depois envia para os assinantes que não a possuem;
- Servidor ordena que os assinantes que possuem a cópia enviem para os assinantes que não a possuem.

A primeira alternativa é a mais simples de ser implementada, porém, pode demorar mais tempo, especialmente se houverem muitos assinantes para receber a cópia. Para a implementação do protótipo do modelo (descrita no capítulo 4), foi escolhida a primeira opção. Porém, a segunda opção também pode ser utilizada por uma implementação do modelo. O algoritmo de recuperação completo, utilizando a primeira alternativa, consiste dos seguintes passos:

- Servidor obtém uma cópia do “estado mais correto” (para abreviar, “C”) do segmento junto aos assinantes que a possuem;
- Após receber C, o servidor recalcula as áreas de interesse dos jogadores baseado na informação contida em C (objetos presentes e donos de objetos);

- Servidor dispara as assinaturas necessárias para que sejam satisfeitos tanto os critérios de área de interesse, quanto os critérios de números mínimos de assinantes (descritos na seção 3.9). Os novos assinantes (se algum) recebem o estado C do segmento diretamente do servidor;

- Após a finalização das novas assinaturas, o servidor envia mensagem para todos os assinantes autorizando a continuação da simulação.

A figura 3.18 continua o cenário ilustrado pela figura 3.17, executando a recuperação da "falha parcial" que foi detectada anteriormente. É importante destacar que, segundo o terceiro passo do algoritmo acima, há o disparo de novas assinaturas para o grupo. Estas novas assinaturas são disparadas principalmente para que os critérios de "número mínimo" de assinantes ("normais" ou "aleatórios") sejam cumpridos. Por exemplo, se a aplicação especifica que cada grupo deve possuir no mínimo dois assinantes aleatórios durante a execução, e no mínimo um assinante aleatório após uma falha (para que a mesma seja considerada "parcial"), então, durante a execução do terceiro passo do algoritmo acima, será alocado um novo assinante "aleatório", para que o número mínimo de dois assinantes aleatórios seja cumprido. Esta situação é ilustrada no passo "3" da figura 3.18, onde há a inclusão de um novo assinante aleatório no grupo.

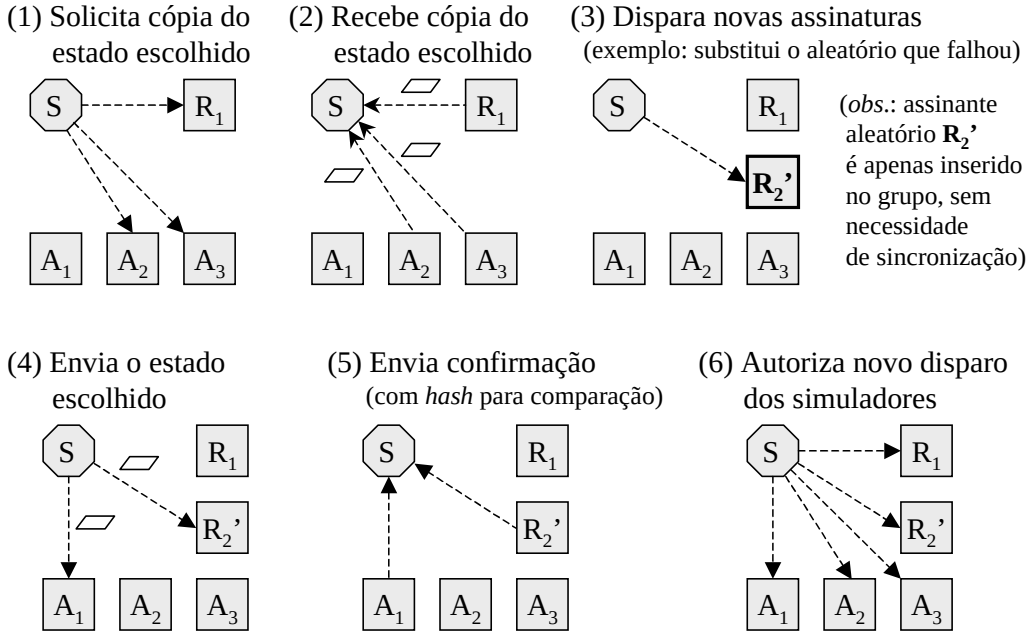


Figura 3.18: Exemplo de execução do protocolo inicial de recuperação de falhas

Também é importante destacar que o protocolo de assinatura utilizado durante a recuperação não é exatamente o mesmo utilizado para uma assinatura "normal". Isto ocorre pois, por um lado, as assinaturas ocorrem com a simulação parada e, portanto, a sincronização dos relógios dos simuladores não é necessária. Por outro lado, quem fornece o estado inicial da simulação para os novos assinantes é o servidor, e não os assinantes "antigos", que não falharam. A lógica por trás desta decisão de modelo é que, após uma falha, mesmo que o servidor decida "acreditar" nos assinantes restantes, o risco de que os assinantes restantes estejam trapaceando é maior do que o risco de ocorrência de trapaças

durante a execução normal, quando o número de réplicas é maior. Portanto, para minimizar problemas, o servidor é utilizado para entregar a cópia para os novos assinantes, reduzindo as chances de que a própria recuperação da falha resulte em uma situação de falha (o que pode acabar resultando em uma situação de falha "fatal").

3.14 Recuperação de falhas fatais

O mecanismo descrito na seção anterior, de restauração de falha parcial, é independente do mecanismo de restauração de falha fatal que pode ser empregado em uma implementação do FreeMMG. Nas próximas duas seções serão descritos dois protocolos alternativos de recuperação de "falhas fatais". Cada algoritmo necessita de diferentes mecanismos de suporte (*snapshots* periódicos, *logs* de comandos, etc) e causa maior ou menor aumento no custo total de processamento e de comunicação, tanto do servidor quanto dos clientes. Neste sentido, cada algoritmo também fornece diferentes garantias à aplicação como, por exemplo, manutenção ou não da consistência global em caso de falha, maior ou menor impacto no jogo durante a recuperação da falha, etc.

Durante a execução de ambos os algoritmos, o servidor já sabe que o grupo de assinantes não possui uma cópia confiável do estado da simulação. Portanto, o servidor sabe que pode utilizar apenas a informação que possui armazenado localmente para realizar a recuperação da falha. Dependendo do algoritmo de recuperação utilizado, outras medidas de suporte também precisarão de uma implementação, como obtenção de *snapshots* periódicos pelo servidor ou alocação de um assinante no servidor.

3.15 Restauração de falha fatal por *snapshots* (algoritmo 1)

O protocolo mais simples possível para recuperação de falha pelo servidor consiste na restauração da última "cópia inativa" ou *snapshot* salvo pelo servidor. A principal desvantagem desta técnica é que, após uma falha, o dano ao andamento do jogo pode ser grande, pois todas as "jogadas" realizadas pelos jogadores entre o tempo do *snapshot* e o tempo da falha são perdidas. Em passos, o algoritmo de recuperação por *snapshot* consiste do seguinte:

- Servidor detecta a falha do grupo e restaura o último *snapshot* disponível localmente (S);
- Servidor recalcula as áreas de interesse dos jogadores baseado na informação contida em S (objetos presentes e donos de objetos);
- Servidor dispara as assinaturas necessárias para que sejam satisfeitos tanto os critérios de área de interesse, quanto os critérios de números mínimos de assinantes (descritos na seção 3.9). Os novos assinantes (se algum) recebem o estado S do segmento diretamente do servidor;
- Após finalização das novas assinaturas, o servidor envia mensagem para todos os assinantes autorizando a continuação da simulação.

Para minimizar a perda de estado do jogo após a recuperação da falha, o servidor deve ser configurado para obter, periodicamente, *snapshots* ou cópias completas do estado da simulação de cada grupo ativo. Dependendo do tamanho da representação (em bytes) e

da quantidade de “objetos” que são criados pela aplicação, o custo de comunicação para a obtenção de um *snapshot* pode ser proibitivo. Como referência, um *snapshot* de Age of Empires II, medido pelo tamanho do arquivo de salvamento de jogo, pode ter entre 200 e 1500 Kbytes. Por outro lado, um *snapshot* do jogo-protótipo implementado (descrito no capítulo 4) pode ter menos de 10 Kbytes, mas o tamanho dos segmentos é configurado pela aplicação para ser bastante pequeno e, portanto, o número total de segmentos ativos no sistema pode ser muito grande.

Quanto mais freqüente for a obtenção de *snapshots*, menor será a perda causada ao jogo. De qualquer forma, é possível imaginar que muitos jogadores ficariam bastante insatisfeitos mesmo com uma perda de 5 ou 10 minutos de jogo. Além disso, a obtenção muito freqüente de *snapshots* pelo servidor pode negar um dos principais benefícios do modelo FreeMMG, que é a reduzida utilização dos recursos de rede do servidor.

Neste sentido, caso o servidor não possua recursos de rede suficientes, a única saída viável para a redução do impacto ao jogo é tentar evitar ao máximo que falhas do tipo “fatal” ocorram. Isto pode ser feito tanto através do aumento dos limiares mínimos de assinantes por segmento, bem como através da redução dos limiares mínimos de assinantes para configuração de “falha parcial”. Caso a replicação seja suficientemente alta, é possível que a obtenção de *snapshots* possa ser feita em intervalos bastante grandes (por exemplo, uma vez por dia) ou mesmo completamente descartada (recuperação de segmentos falhos através da restauração de uma cópia “em branco” do segmento).

3.15.1 Problema da inconsistência global

Um problema inerente a esta abordagem de recuperação por *snapshots* é a introdução de inconsistências na simulação “global”, ou seja, considerando-se o conjunto de todos os simuladores (grupos de assinantes de segmentos). Apesar de cada segmento possuir o seu próprio relógio de simulação, os estados dos vários segmentos são sincronizados entre si através da transferência constante de objetos entre os segmentos.

Dois exemplos claros de “inconsistência global” consistem na duplicação e na eliminação de objetos. A duplicação de um objeto pode ocorrer no seguinte cenário:

- Inicialmente, o segmento A possui objeto O, e segmento B está vazio;
- Servidor obtém *snapshot* do segmento A;
- Objeto O é transferido de A para B;
- Uma falha fatal ocorre no segmento A, e o *snapshot* obtido anteriormente é restaurado. Como resultado, o objeto O possui duas instâncias (uma em A, e outra em B).

De forma similar, um objeto pode ser eliminado da seguinte forma:

- Inicialmente, o segmento A possui objeto O, e segmento B está vazio;
- Servidor obtém *snapshot* do segmento B;
- Objeto O é transferido de A para B;
- Uma falha fatal ocorre no segmento B, e o *snapshot* obtido anteriormente é restaurado. Como resultado, o objeto O não possui mais nenhuma instância (pois B não possui mais o objeto).

Este tipo de problema é bastante estudado em simulação distribuída, no contexto de implementação de algoritmos otimistas ou especulativos (FUJIMOTO, 2000). Como a simulação é otimista e possui um relógio lógico global, um dos simuladores distribuídos pode detectar localmente, através da comparação de "timestamps" de mensagens, que aplicou mensagens fora de ordem. A restauração pode ser feita de várias formas, como através do cancelamento das mensagens ou eventos distribuídos (o que pode envolver envio de mensagens de cancelamento para outros simuladores) e posterior re-aplicação dos eventos na ordem correta (FUJIMOTO, 2000). Existem outras soluções como o algoritmo de Trailing State Synchronization (CRONIN et al., 2002; CRONIN; FILSTRUP; KURC, 2001), que realiza a manutenção de simuladores paralelos, cada um com um nível diferente de "otimismo". No TSS, a recuperação da sincronia é feita através da restauração do simulador mais "otimista" que ainda é correto, sendo que os outros são descartados.

No FreeMMG poderia ser implementado um algoritmo de recuperação similar. Como as transferências de objetos entre segmentos passam pelo servidor, o servidor pode, em princípio, manter um registro de todas as transferências de objetos relacionadas com um segmento deste o último *snapshot* obtido daquele segmento. Então, para restaurar a consistência global, em princípio bastaria restaurar o último *snapshot* e desfazer as transferências de objetos de e para aquele segmento desde o tempo do relógio associado ao *snapshot*. Porém, esta solução causa outro problema: ao "desfazer" uma transferência de objeto de ou para outro segmento, é causada a "falha" deste outro segmento, e também se faz necessária a restauração do último *snapshot* deste segmento, o que também pode causar a necessidade de falhar e restaurar outros segmentos, e assim por diante.

Mesmo que o impacto deste tipo de recuperação seja aceitável pelos jogadores, o que é difícil, a qualidade do estado restaurado ainda pode ser "inconsistente", caso a aplicação defina critérios mais complexos de consistência do jogo que envolvam, por exemplo, o posicionamento específico dos objetos. As transferências de objetos ocorrem em tempos diferentes entre si, mas o servidor, ao desfazer (ou refazer) as transferências de objetos, faz isso de uma só vez. E, como o servidor não sabe onde exatamente colocar os objetos (pois eles foram transferidos em tempos diferentes), isto pode resultar em objetos em locais absurdos ou não exatamente apropriados, como tanques de guerra posicionados dentro de bases inimigas, etc. Mesmo que nenhuma regra formal de consistência seja quebrada, o resultado final provavelmente decepcionaria os jogadores. Além disso, deve-se também considerar as inconsistências que se originam das transformações dos atributos dos objetos ao longo do tempo. Por exemplo, se os objetos "envelhecem" ou se tornam mais fortes ao longo do tempo, isto é perdido ao se desfazer uma transferência de objeto, pois o servidor só possui a cópia do objeto de quando este foi transferido.

Neste contexto, fica claro que, se o servidor obtém apenas *snapshots* periódicos, além de saber sobre a transferência de objetos entre segmentos, o desenvolvimento de um algoritmo de "rollback distribuído" não contribui significativamente para a manutenção da consistência global da simulação. Isto ocorre pois muita informação é perdida, e a informação que é recuperada não compensa o estrago perceptível pelos jogadores que é causado por um rollback.

3.15.2 Aplicabilidade do algoritmo

O algoritmo de recuperação de falha fatal por restauração de *snapshot* pode ser útil especialmente se o jogo ou aplicação for projetado para tolerar perdas eventuais no estado do jogo. Se a rede FreeMMG é configurada para garantir uma elevada replicação nos grupos de assinantes, a ocorrência de falhas fatais pode ser reduzida ao ponto em

que a perda de estado torna-se aceitável. Outra vantagem da abordagem é que ela não exige que as transferências de objetos sejam realizadas pelo servidor. Por exemplo, os objetos poderiam ser transferidos diretamente entre os assinantes, desde que respeitado um critério mínimo de replicação de mensagens para que não seja possível a fabricação gratuita de objetos através de transferências ilegais.

3.16 Restauração de falha fatal por *snapshots* e *logs* de comandos (algoritmo 2)

Como já discutido anteriormente, o problema da recuperação de "falhas fatais" é resolvido se cada segmento ativo possuir um assinante "confiável", alocado em uma máquina no "lado servidor" da rede FreeMMG. Mas, neste caso, o "lado servidor" torna-se novamente um gargalo de rede e de processamento à medida em que o número de clientes aumenta. Com milhares de clientes e segmentos ativos, isto anularia o benefício da distribuição da simulação entre os clientes, que é razão de ser do modelo FreeMMG. Porém, pode-se obter um resultado similar através da alocação de um "pseudo-assinante" no servidor, para cada segmento ativo. Este pseudo-assinante se diferencia de um assinante "completo" em três aspectos: não-envio de comandos, recebimento de "resumos" de comandos e processamento de comandos apenas para recuperação de falhas "fatais".

3.16.1 Não-envio de comandos

Em primeiro lugar, o pseudo-assinante não envia comandos. Desta forma, o custo adicional de tráfego no lado servidor já é reduzido pela metade. Porém, isto não é suficiente para manter a escalabilidade do servidor, pois milhares de assinantes enviando comandos para o servidor já caracterizam um gargalo na comunicação. Além disso, esta é uma medida que também poderia ser adotada por assinantes "normais" alocados no lado servidor.

3.16.2 Recebimento de resumos de comandos

Em segundo lugar, o pseudo-assinante recebe "resumos de comandos" enviados pelos outros assinantes em intervalos grandes, ao invés de receber comandos normalmente, em pequenos intervalos. Supondo que o passo dos simuladores seja configurado para 100ms, isto significa que um assinante "normal" recebe, em média, 10 mensagens por segundo, cada uma contendo um conjunto de comandos gerados pelo jogador remoto desde a última mensagem. Neste contexto, um "pseudo-assinante" pode ser configurado para receber uma única mensagem contendo os últimos 100 comandos gerados pelos outros assinantes, o que resultaria no envio, para o "pseudo-assinante" de uma única mensagem a cada 10 segundos. Isto tende a reduzir de forma significativa o custo de comunicação dos clientes com o servidor por duas razões.

Primeiro, como vários comandos são concatenados em uma única mensagem, isto significa que os ganhos através da compressão de pacotes podem ser bastante significativos. Considerando-se que o comando "padrão" enviado por um assinante é nulo (pacote vazio, ou 1 byte notificando que o mesmo não gerou nenhum comando naquele turno), um assinante que fique 10 segundos sem enviar nenhum comando estará gerando ou a concatenação de 100 mensagens vazias (nada) ou uma "string" de 100 bytes que possuem o mesmo valor, o que é bastante compressível. Mesmo que um jogador gere, em média, um comando por segundo, isto ainda resulta em seqüências grandes onde não há nenhum

comando gerado, o que ainda resultaria em ganhos significativos durante a compressão do resumo de comandos.

Segundo, há uma grande economia de tráfego efetivo através da eliminação dos cabeçalhos de pacotes. Uma implementação do FreeMMG pode utilizar protocolos-padrão da Internet como TCP/IP ou UDP/IP que, na prática, chegam a utilizar em torno de 50 bytes para o cabeçalho de um pacote (somando-se o overhead dos protocolos TCP, UDP, IP, etc). Já o tamanho médio do conteúdo de uma mensagem do tipo "comando" tipicamente não ocupará, em média, mais do que 10 bytes, sendo que o valor médio típico deve ser muito menor por causa dos períodos de inatividade dos jogadores (detalhes de implementação e resultados obtidos são discutidos no capítulo 4). Se considerarmos os tamanhos médios de cabeçalho (50 bytes) e comando (10 bytes) considerados, e o envio de "resumos de comandos" apenas a cada 100 turnos, o tráfego no servidor é reduzido para 17,5

3.16.3 Não-execução de comandos

Apesar de receber os comandos de todos os assinantes de cada segmento ativo, os pseudo-assinantes alocados no lado servidor não executam estes comandos em tempo real. Em uma situação normal, os comandos são apenas armazenados, sendo só utilizados quando é necessário que o servidor execute o algoritmo de recuperação de falha fatal. Assim, juntamente com as duas características descritas anteriormente (não-envio de comandos e recebimento de resumos), o servidor pode manter localmente um *log* atualizado de todos os comandos enviados por todos os assinantes de cada segmento ativo, sem que isto resulte em um aumento proibitivo no custo de comunicação e de processamento do "lado servidor".

3.16.4 Recuperação de falhas com *snapshot* e *logs* de comandos

Visto que o pseudo-assinante possui um custo de comunicação e de processamento que são bastante reduzidos, se comparados com o custo da alocação de um assinante "normal", resta saber como os dados coletados pelo pseudo-assinante (os "resumos" de comandos) são utilizados para o tratamento de "falhas fatais". Este algoritmo (Algoritmo 2) também é baseado no algoritmo anterior (Algoritmo 1) de recuperação de falhas fatais por restauração de *snapshot* armazenado no servidor. Neste sentido, o Algoritmo 2 consiste, basicamente, na aplicação dos *logs* de comandos sobre o último *snapshot* salvo. Como a "simulação replicada" do FreeMMG é determinística, o estado resultante deve ser idêntico ao estado calculado pelos clientes (pelo menos até o ponto onde foram recebidos os últimos resumos de comandos) e, portanto, reduzindo significativamente as perdas no estado da simulação. Em passos, a recuperação por *snapshot* e *log* consiste do seguinte:

- Servidor detecta a falha do grupo e busca localmente o último *snapshot* do segmento (S) e o *log* de comandos dos assinantes, e constrói uma cópia atualizada do segmento (S') através da execução dos "turnos" em que o servidor possui a lista completa de comandos (de todos os assinantes);
- Servidor recalcula as áreas de interesse dos jogadores baseado na informação contida em S' (objetos presentes e donos de objetos);
- Servidor dispara as assinaturas necessárias para que sejam satisfeitos tanto os critérios de área de interesse, quanto os critérios de números mínimos de assinantes (descritos na seção 3.9). Os novos assinantes (se algum) recebem o estado S' do segmento diretamente do servidor;

- Após finalização das novas assinaturas, o servidor envia mensagem para todos os assinantes autorizando a continuação da simulação.

Com esta abordagem, a freqüência de atualização do *snapshot*, no lado servidor, não precisa ser alta. Em princípio, a atualização do *snapshot* no servidor deve ser feita apenas se os *logs* de comandos de um certo segmento ou grupo tornarem-se muito grandes. Quando um *snapshot* é obtido pelo servidor, este pode descartar todos os comandos de clientes no *log* que possuem um timestamp menor do que o timestamp do novo *snapshot*.

Apesar de a perda causada pela ocorrência de uma falha ser bastante reduzida, a consistência global é garantida somente se duas condições forem satisfeitas. Primeiro, cada segmento deve possuir no mínimo um assinante "honesto" pois se todos os assinantes resolvem fabricar objetos através de transferências ilegais, o servidor não terá como detectar esta situação (pois o servidor não está executando os comandos dos clientes). Além disso, a consistência global só é garantida se as transferências de objetos forem adiadas, pelo servidor, até que os resumos de comandos do segmento de origem do objeto transferido sejam atualizados até o timestamp em que o objeto é retirado. Isto é necessário para garantir que a transferência de objeto, em caso de falha "fatal", não precisará ser desfeita pelo servidor.

3.16.5 Aplicabilidade do algoritmo e melhorias possíveis

Se o servidor possuir recursos de rede suficientes para o recebimento constante de resumos de comandos, este algoritmo pode resolver o problema da perda de estado devido à ocorrência de uma falha "fatal" em um grupo de simuladores. Porém, a "consistência global" da simulação só pode ser garantida se forem alocados assinantes "honestos" em todos os grupos. Além disso, a transferência de objetos entre segmentos deve ser "adiada" pelo servidor até que os resumos de comandos do segmento de origem sejam atualizados até o ponto onde o objeto deixa o segmento.

Em princípio, a garantia de consistência global, bem como a garantia de que transferências ilegais de objetos não possam ocorrer, poderiam ser obtidas se o "lado servidor" executasse os resumos de comandos que recebe de todos os assinantes. Porém, como já discutido, isto representa um gargalo de processamento para o servidor, pois uma rede FreeMMG pode possuir milhares de segmentos ativos simultaneamente. Porém, se a obtenção de poder de processamento (CPUs) para o servidor não for um problema, então é possível que o servidor forneça garantias sobre a manutenção da consistência global, bem como previna a ocorrência de "trapaças" de alterações arbitrárias no estado dos segmentos e de transferências ilegais de objetos entre segmentos.

3.17 Otimização de rede para assinantes inativos

Dependendo do tipo de jogo que for implementado sobre uma rede FreeMMG, é possível que um cliente venha a ser assinante de muitos segmentos. Isso pode resultar em um problema de falta de recursos suficientes, tanto de processamento quanto de rede, nos clientes que assinam os segmentos. Se um cliente executa uma parte muito grande da simulação do "mundo virtual", além de sobrecarregar a sua CPU, terá que se comunicar com uma grande quantidade de assinantes remotos. Isto é especialmente grave, se considerarmos que o FreeMMG precisa, em geral, manter um número elevado de assinantes em cada segmento para que a ocorrência de trapaças e o estrago provocado por falhas de assinantes seja minimizado.

Para minimizar pelo menos o problema do excesso de comunicação do cliente, é possível explorar a "localidade" das interações do usuário. Mesmo que um cliente controlado por um jogador humano venha a assinar um grande número de segmentos, e que estes segmentos sejam suficientemente "grandes", é muito provável que a atenção deste jogador esteja voltada sempre para uma parte reduzida do "mundo virtual", visto que jogadores humanos geralmente tem recursos limitados de tempo para interagir com a área de jogo, especialmente em jogos de estratégia. Por exemplo, se um assinante estiver assinando vinte segmentos, onde cada segmento corresponde a, aproximadamente, uma "tela cheia" de objetos, é provável que, em média, o jogador concentre sua atenção em apenas um ou dois segmentos de uma só vez. Este fato pode ser explorado para que se implemente uma otimização que reduza significativamente as mensagens enviadas por cada assinante. E como os outros assinantes também estarão executando a otimização, a otimização também reduz, na prática, o número de mensagens recebidas por cada assinante.

Em princípio, a freqüência com que os assinantes trocam mensagens entre si deve ser suficientemente alta para que o tempo entre a realização de um comando, por um jogador (por exemplo, mover um objeto) e a execução deste comando seja o menor possível. Mas se um assinante fica vários segundos ou minutos sem enviar nenhum comando para um segmento, em princípio as mensagens com "comandos" vazios deste jogador não precisariam ser enviadas. Porém, um assinante "inativo" não pode simplesmente parar de enviar comandos pois, como a simulação não é "otimista", os outros assinantes precisam esperar o assinante inativo avisar que não fará nada, em cada turno, para que a simulação possa prosseguir. Para resolver este problema, a otimização para assinantes inativos faz com que o assinante inativo "negocie", com os outros assinantes, uma nova freqüência de envio de comandos. Desta forma, quando um assinante é detectado como "inativo", este pode passar a enviar mensagens em uma freqüência bastante reduzida.

A detecção da inatividade é realizada individualmente por cada assinante, sem a necessidade de troca de mensagens adicional. Quando um assinante detecta, localmente, que não recebeu nenhum comando efetivo de um certo assinante "A" (que pode ser ele mesmo), apenas comandos "vazios", por uma certa quantidade "T" de turnos, então a otimização é ativada localmente para o assinante "A". Enquanto "A" estiver marcado como "inativo", de cada "S" turnos executados pelo simulador local, em apenas um turno será esperado um comando do assinante "A", sendo que, em todos os outros turnos do intervalo "S", os simuladores do grupo irão ignorar comandos de "A". Como "A" também detecta a sua própria inatividade, também não irá enviar comandos durante os turnos inativos. Neste contexto, "S" e "T" são constantes configuradas pela aplicação.

O único efeito colateral da otimização é que quando um assinante está inativo e o jogador associado gera um ou mais comandos naquele segmento (por exemplo, ordena a movimentação de um objeto), estes "primeiros comandos" só serão executados quando for o momento de enviar o próximo comando. Por exemplo, se a constante S é configurada para 50 turnos (S = 50) e os simuladores são configurados para executar 10 turnos por segundo, isto significa que os primeiros comandos gerados durante a inatividade de um assinante poderão demorar até 5 segundos para serem enviados e executados. Porém, quando todos os assinantes detectarem o envio de comandos "efetivos" por um assinante inativo, este passa novamente a ficar "ativo" e a enviar comandos em todos os turnos.

Variações desta otimização são possíveis. Por exemplo, pode-se configurar vários "graus" de inatividade de um assinante. Se um assinante fica inativo por 10 segundos, pode-se reconfigurar o envio de comandos para funcionar na metade da velocidade normal, ou seja, se normalmente são enviados 10 comandos por segundo, com esta "inativi-

dade leve" o assinante passa a enviar apenas 5 comandos por segundo (S = 2). E, caso o assinante permaneça inativo por um tempo elevado, por exemplo, 1 minuto, o "grau" de inatividade pode ser elevado, por exemplo, para que o assinante envie apenas 1 comando por segundo (S = 10). Esta configuração foi testada no protótipo implementado (jogo FreeMMG Wizards, apresentado no capítulo 4) e foi constatado informalmente que o impacto do "primeiro" grau de inatividade (S = 2) é difícil de ser notado pelo jogador, enquanto que o impacto do "segundo" grau (S = 10) pode ser notado mas, na prática, pode ser tolerado pelos jogadores.

Caso sejam utilizados valores muito elevados para o parâmetro "S", é possível modificar a otimização para que o atraso do envio do "primeiro comando" após a inatividade se torne aceitável. Para isto, basta que o assinante inativo, ao detectar um comando gerado pelo jogador local, inicie uma troca de mensagens independente para a sua reativação. Isto poderia ser feito através do servidor, ou diretamente entre os assinantes. Esta variação não foi implementada mas, em princípio, poderia até fazer com que assinantes com um "grau" de inatividade elevado parassem completamente de enviar comandos.

3.18 Revisão e considerações finais

Este capítulo apresentou uma descrição detalhada do FreeMMG, um modelo híbrido, cliente-servidor e *peer-to-peer*, para simulação distribuída e interativa, em tempo real, de jogos maciçamente multijogador (MMGs ou *massively multiplayer games*). Neste contexto, foram apresentados os principais componentes do modelo, a saber:

- Servidor: utilização de um servidor para tarefas de gerência da rede, como autorização e autenticação, detecção e recuperação de falhas e gerenciamento dos grupos de clientes;

- Segmentos: partição do "mundo virtual" do jogo ou simulação em "segmentos" (ou células);

- Simulação *peer-to-peer* replicada: formação de "grupos de assinantes" em cada segmento, formando redes *peer-to-peer* de tamanho reduzido em cujo contexto é executado um algoritmo não-otimista de "simulação replicada";

- Assinatura e desassinatura: Suporte a inclusão (”assinatura”) e remoção (”desassinatura”) de clientes em cada grupo de assinantes, de forma dinâmica;

- Áreas de interesse: suporte a diferentes critérios para cálculo de "áreas de interesse" dos clientes, o que permite ao servidor determinar quais clientes devem assinar quais segmentos;

- Ativação e desativação de segmentos: mecanismos específicos para o tratamento de grupos que recebem o seu primeiro assinante ou retiram o seu último assinante interessado (de forma a não permitir trapaças ou exploração de vulnerabilidade nestes casos);

- Mensagens replicadas: eventos gerados de forma determinística por todos os clientes "assinantes" de um segmento, quando enviados para o servidor, devem ser enviados em mensagens por todos os assinantes;

- Transferências de objetos: interação entre diferentes segmentos é suportada através da transferência de objetos entre segmentos;
- Prevenção de falhas: manutenção de um número mínimo de réplicas de dois tipos (assinante "normal" e assinante "aleatório") em cada segmento, através da alocação de assinantes complementares ou "extras";
- Recuperação de falhas: um mecanismos para recuperação de falhas "parciais" (poucas falhas individuais de assinantes) e dois mecanismos alternativos para recuperação de falhas "fatais" (vários assinantes de um mesmo segmento falhando simultaneamente);
- Otimização para tratamento de clientes que possuem "assinantes" inativos em segmentos, visando a redução dos recursos de rede necessários para que um cliente participe como simulador assinante em vários segmentos simultaneamente.

É importante destacar os seguintes aspectos do modelo, que se fazem presentes na especificação e na avaliação de aplicabilidade de cada componente do modelo, individualmente, bem como do modelo como um todo:

- Reduzido consumo de recursos no "lado servidor": todos os componentes foram projetados com a preocupação constante de que o servidor não se caracterizasse como um "gargalo" para a simulação (tanto de processamento quanto de rede), o que anularia os benefícios da maior complexidade do modelo, que utiliza uma rede *peer-to-peer* de clientes para a execução da simulação;
- Proteção contra jogadores "trapaceiros": todos os componentes do modelo foram projetados para resistirem a tentativas de "trapaça", em especial às trapaças mais fáceis de serem executadas e que causam os maiores prejuízos, como é o caso de trapaças do tipo "alteração arbitrária de estado";
- Extensibilidade: apesar de ser uma tarefa difícil, o modelo foi desenvolvido de forma a impor o menor número possível de restrições em futuras extensões. Por exemplo, é possível a extensão do modelo para a utilização de um servidor distribuído em várias máquinas, para que uma maior escalabilidade seja alcançada;
- Aplicabilidade em jogos de tempo real: o modelo foi projetado para que fosse possível suportar jogos já existentes ou, no mínimo, jogos comparáveis aos jogos de tempo real disponíveis comercialmente.

O primeiro aspecto será avaliado em detalhes no capítulo 4, que apresentará os resultados experimentais obtidos com a execução de testes sobre o protótipo do modelo. Em relação ao segundo aspecto, o FreeMMG se posiciona como uma solução "resistente" a certos tipos de trapaças. Apesar de o modelo ser centrado na prevenção de um tipo específico de trapaça (alteração arbitrária do estado distribuído do jogo), certamente é possível a integração de outras técnicas de detecção e prevenção de trapaças ao modelo, o que vai de encontro ao aspecto de extensibilidade do modelo. Por fim, sobre a aplicabilidade do modelo em problemas reais, como o suporte a *massively multiplayer games*, pode-se afirmar que o modelo FreeMMG apresentado neste capítulo é adequado para suportar, no mínimo, jogos de estratégia em tempo real (jogos "RTS" ou *real-time strategy*) em um contexto maciçamente multijogador. Porém, devem ser levadas em consideração as

limitações atuais do modelo, como a não-transparência do mecanismo de distribuição do estado em “segmentos” pois, como será discutido no capítulo seguinte, as “fronteiras” entre os segmentos podem ser percebidas com uma certa facilidade pelos jogadores.

4 IMPLEMENTAÇÃO

Neste capítulo são apresentados os detalhes de implementação do protótipo que foi desenvolvido para validar o modelo, bem como os resultados que foram obtidos através de experimentos práticos. O protótipo implementado pode ser dividido em quatro partes principais:

- Servidor: implementa a funcionalidade do "lado servidor" do FreeMMG (provedor de autenticação, gerência de clientes, etc.);
- Cliente: implementa a funcionalidade do "lado cliente" do FreeMMG (algoritmo de simulação peer-to-peer replicada, gerência de múltiplas assinaturas, etc.);
- Jogo e interface: implementação da "aplicação", incluindo as regras de um jogo simples (batizado de "FreeMMG Wizards") e a interface gráfica para os jogadores humanos;
- Simulador de clientes em lote: ferramenta que conecta em um servidor de "FreeMMG Wizards" e simula o comportamento de múltiplos jogadores, utilizada nos experimentos para geração de resultados.

Na seção 4.1 é apresentado o protótipo desenvolvido, o jogo FreeMMG Wizards, sendo que esta apresentação é centrada nas características ("features") do protótipo e como elas atenderiam as expectativas de um usuário. Na seção 4.2 são apresentados os detalhes de implementação dos componentes principais (servidor, cliente, interface e jogo). Na seção 4.3 o simulador de clientes é apresentado, e na seção 4.4 é descrita da metodologia empregada para a realização dos experimentos com o simulador de clientes, bem como os resultados obtidos. A seção 4.5 discute os resultados, tecendo uma comparação com o desempenho tipicamente apresentado por MMGs cliente-servidor. A seção 4.6 compara o FreeMMG com algumas das arquiteturas apresentadas no capítulo 2. Por fim, a seção 4.7 apresenta algumas considerações finais relacionadas ao protótipo e conclusões.

4.1 O protótipo: jogo FreeMMG Wizards

Para a validação do modelo, foi desenvolvido um jogo simples, de estratégia em tempo real, que prevê a utilização de um "mundo virtual" de estado persistente e a participação de centenas ou milhares de jogadores, simultaneamente. O jogo é simples, mas é suficiente para exigir a maior parte das funcionalidades de uma implementação do modelo. O jogo foi batizado de "FreeMMG Wizards", e a versão utilizada para os testes (versão

0.0.5.1) pode ser obtida na página de desenvolvimento do projeto [1]. Todos os componentes do protótipo foram implementados sobre a plataforma Java J2SE versão 1.4, sendo necessária uma máquina virtual Java compatível para o disparo do cliente e do servidor.

4.1.1 Disparo do servidor

Para que os jogadores possam participar de uma "partida" de Wizards, primeiro é preciso iniciar o servidor. Para iniciar um servidor em um terminal texto, basta invocar a classe mmg.proto.ConServer, que dispara o servidor em "modo console". Utilizando o pacote pré-compilado, disponibilizado no site do projeto, o disparo pode ser feito a partir do diretório principal do pacote, como mostrado pela figura 4.1. Durante o disparo, o servidor restaura o último estado do jogo, o que inclui a última cópia ou versão do mundo virtual e a relação de jogadores cadastrados.

Para parar o servidor na máquina local, a classe mmg.proto.ConServerStop pode ser invocada. O encerramento do servidor consiste na desativação de todos os segmentos ativos, seguido do encerramento das threads do servidor. Alternativamente, o usuário pode simplesmente matar o processo do servidor (ou da máquina virtual Java que executa o servidor), pois o mesmo é tolerante à interrupção abrupta da execução. A única desvantagem do encerramento abrupto do servidor é que parte do estado do jogo será perdido, pois as cópias mais atuais dos segmentos ativos estarão com os clientes, visto que o servidor utiliza o algoritmo de recuperação de falhas por snapshots periódicos (descrito na seção 3.15). Este efeito indesejável pode ser evitado em uma futura implementação do modelo que, por exemplo, empregue o algoritmo de envio de "resumos de comandos" dos clientes para o servidor (apresentado na seção 3.16).

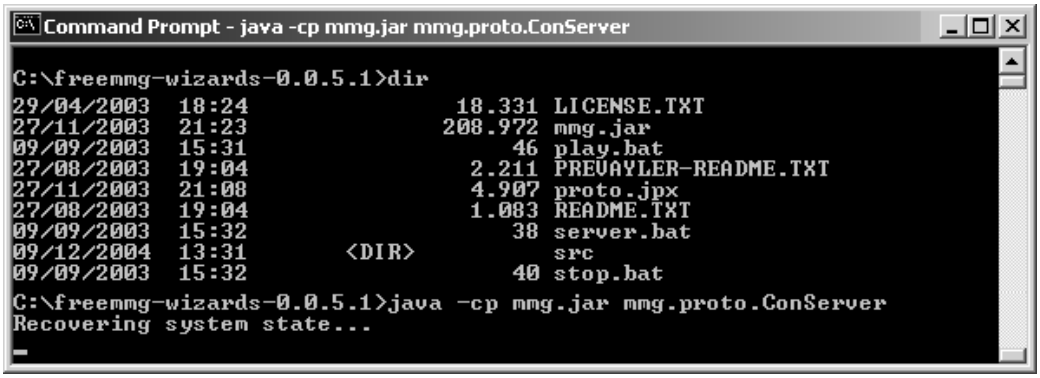


Figura 4.1: Exemplo de disparo do servidor em modo console

4.1.2 Disparo da interface gráfica

A interface gráfica, além de implementar o cliente do FreeMMG e a interface do jogo, também pode ser utilizada para disparar um servidor. Para disparar a interface, basta invocar a classe mmg.proto.Application1. Utilizando o pacote pré-compilado, o disparo pode ser feito como mostrado pela figura 4.2. Após disparada a interface gráfica, o servidor pode ser disparado através do menu "Server", opção "Start", e pode ser parado pela opção "Stop".

[1] http://sourceforge.net/projects/freemmg/

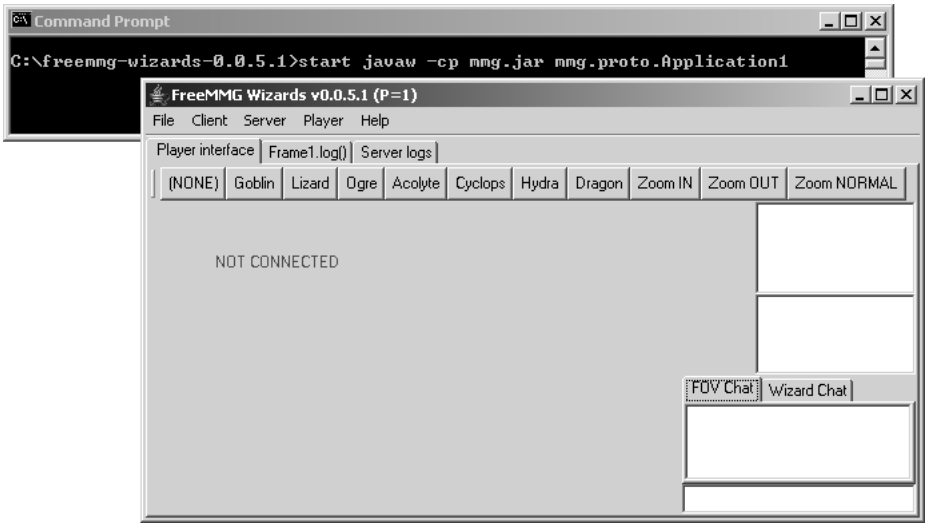


Figura 4.2: Exemplo de disparo da interface gráfica

4.1.3 Criando uma conta de jogador e conectando o cliente no servidor

Quando um novo jogador quer participar do jogo, a primeira coisa que ele deve fazer é solicitar uma conta de jogador ao servidor. Isto é feito através do menu “Client”, opção “New Account”. Após selecionar esta opção, interface solicita um nome simbólico para o jogador, a senha desejada pelo jogador, e o endereço do servidor. O cliente então conecta ao servidor e solicita a criação de uma nova conta, recebendo como resposta um número identificador único (o “Player ID”) para aquela conta de jogador. O pedido de criação de conta, se bem-sucedido, automaticamente dispara a primeira sessão de jogo do usuário. Quando o jogador decide encerrar a sua sessão de jogo, pode utilizar a opção “Disconnect” do menu “Client”. Para retornar ao jogo mais tarde, pode utilizar a opção “Connect” do menu “Client”, fornecendo o seu número identificador, a sua senha, e o endereço do servidor.

4.1.4 Visão geral do jogo e da interface do jogador

A interface do jogo é uma versão simplificada da interface típica de um jogo de estratégia em tempo real (”jogo RTS”). A área principal da interface exibe uma parte de um “tabuleiro” bidimensional plano de cor cinza, sem nenhum tipo de ornamentação como diferentes tipos de terreno, obstáculos, etc. O tabuleiro é particionado em retângulos, sendo que cada retângulo caracteriza um “segmento” do FreeMMG. Assim, um cliente pode “assinar” de forma individual cada “retângulo” ou segmento da área de jogo de forma independente dos outros segmentos. As “fronteiras” entre os retângulos são, por default, exibidas para os jogadores (figura 4.3). Isto é feito pois, em princípio, o jogador precisa saber onde ficam as fronteiras entre os segmentos para que ele possa adaptar a sua estratégia de jogo, já que o FreeMMG ainda não suporta a interação entre objetos localizados em segmentos diferentes.

Seguindo a terminologia da seção 3.1 (que descreve o modelo de aplicação do FreeMMG), o jogo possui apenas dois tipos de “objetos” que são visíveis para os jogadores: “magos” e “monstros”. Os magos são representados por elipses, e os monstros são representados por retângulos. Cada jogador possui, no máximo, um único mago. O jogador pode mover o seu mago livremente pelo tabuleiro, e também pode utilizar o mago para criar “monstros”

Figura 4.3: Exemplo de estado da interface durante uma sessão de jogo

através dos botões disponibilizados na interface (”Goblin”, “Lizard”, etc). Um jogador pode possuir e controlar qualquer número de monstros, simultaneamente. Assim como o mago, o jogador pode mover os monstros livremente pelo tabuleiro. A interface suporta a seleção individual de objetos, bem como a seleção de um grupo de objetos através de um retângulo de seleção (figura 4.3), de forma que o jogador pode mover cada objeto individualmente ou mover um grupo de objetos ao mesmo tempo.

A área do jogo pode ser composta de milhares de segmentos, mas cada jogador só estará interessado em assinar uma pequena fração destes segmentos. A atualização da área de interesse de cada jogador é feita pelo servidor, na medida em que os jogadores movem seus objetos entre os vários segmentos (figura 4.4). O servidor é notificado pelos assinantes de cada segmento quando um novo jogador se torna “presente” em um segmento (primeiro objeto deste jogador entra no segmento) e quando um jogador se torna “ausente” em um segmento (último objeto deste jogador sai do segmento ou é destruído). Quando mudanças nas áreas de interesse são detectadas, o servidor dispara as assinaturas e desassinaturas necessárias para que as mudanças sejam satisfeitas. O servidor determina que cliente é assinante de um segmento se, e somente se, o jogador em questão possuir no mínimo um objeto (”mago” ou “monstro”) neste segmento. Na interface, os segmentos que um jogador não está assinando são representados como retângulos pretos, e os segmentos assinados são representados por um fundo cinza-claro, mais os objetos, que são desenhados em primeiro plano (figura 4.3).

Tanto magos quanto monstros possuem dois atributos relevantes: pontos de vida e força de ataque. Quando dois objetos (magos ou monstros) colidem e são de posse de jogadores diferentes, ocorre um combate. O combate entre dois objetos A e B consiste em subtrair a força de ataque de A dos pontos de vida de B, e vice-versa. Após um combate, é possível que um dos objetos fique com uma quantidade de pontos de vida igual ou menor que zero. Neste caso, o objeto é removido do jogo, e o mago que controla o objeto “vito-

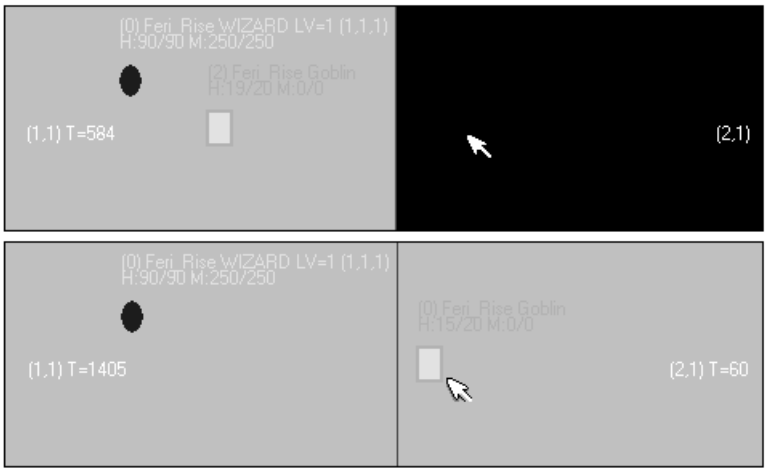


Figura 4.4: Alteração na área de interesse do jogador após transferência de objeto

rioso" recebe um bônus que o torna mais forte (capacidade de criar monstros com maior freqüência ou capacidade de criar monstros mais fortes). Como em boa parte dos jogos de estado persistente, como EverQuest ou Ultima Online, quanto mais o jogador participa do jogo, mais "forte" se torna o seu personagem. Caso o mago de um jogador seja destruído, o jogador pode reviver o mago através do menu "Player", opção "Respawn", que recoloca o mago em jogo em um segmento escolhido aleatoriamente pelo servidor. Esta opção também deve ser utilizada pelo jogador quando conectar no jogo pela primeira vez, pois os novos jogadores ainda não possuem nenhum objeto no tabuleiro.

4.1.5 Persistência, escalabilidade e adaptação do tabuleiro à quantidade de jogadores

O "mundo virtual" de Wizards, ou seja, o conjunto de objetos distribuídos pelos vários segmentos, é persistente. Se um jogador desconectar-se do jogo e, mais tarde reconectar, os objetos que o jogador possui ainda estarão no jogo, desde que não tenham sido destruídos por outros jogadores. Cada segmento persistido impõe um custo adicional no servidor mas, em princípio, o "mundo virtual" de Wizards não possui um limite "máximo" de tamanho, e também não há um limite específico para o número de jogadores cadastrados ou o número de participantes simultâneos no jogo. Portanto, os recursos computacionais disponíveis para a execução do servidor caracterizam o principal fator limitante.

O "mundo virtual" é uma grade bidimensional de segmentos retangulares, e cada segmento é identificado unicamente pela sua coordenada (X, Y) na grade. A grade é armazenada, no servidor, em uma tabela hash (mais especificamente, um java.util.HashMap), onde a chave é a coordenada (X, Y) do segmento, e o valor é uma estrutura de dados que descreve toda a informação gerenciada pelo servidor relativa àquele segmento (incluindo snapshots, logs de comandos, relação de assinantes, etc). Um novo segmento é alocado pelo servidor quando um objeto é transferido para uma coordenada (X, Y) cuja chave ainda não existe na tabela da grade. Desta forma, o tamanho do "mundo virtual" se adapta à quantidade de jogadores.

4.2 Implementação do protótipo

A implementação da versão 0.0.5.1 do protótipo "FreeMMG Wizards" possui, no total, 11.939 linhas de código Java distribuídas entre 42 classes e interfaces (sem incluir classes internas, adaptadores, etc). Isto inclui a implementação do cliente, do servidor, da lógica do jogo, da interface gráfica e também do simulador de clientes em lote. A implementação do modelo foi feita especificamente para suportar o jogo "Wizards", de forma que a lógica do jogo e a implementação do cliente e do servidor estão fortemente acopladas. A implementação do modelo FreeMMG em um framework reusável, que isola a implementação do sistema de suporte e expõe uma API para o programador da aplicação, é um trabalho em andamento (atualmente com 6.735 linhas de código Java distribuídas entre 51 classes e interfaces, sendo que a maior parte deste código foi re-escrita).

Todo o código é baseado exclusivamente no perfil Java J2SE. Para a interface gráfica, foi utilizado o kit Swing (pacotes javax.swing, java.awt.event, etc), sendo que parte do código da interface (relativo aos menus, botões, etc) foi gerado pelo Borland JBuilder Foundation. Para a comunicação são utilizadas as sockets TCP/IP de Java (pacotes java.io e java.net), com a alocação de uma thread para cada socket, sendo que a leitura em cada socket é bloqueante. O protocolo TCP/IP foi escolhido pois os algoritmos do FreeMMG só utilizam entrega de mensagens de forma confiável e ordenada, e o suporte a TCP/IP é, pelo menos atualmente, mais ubíquo do que o suporte a protocolos do tipo "multicast". O suporte a serialização de objetos do Java é utilizado para o envio, pelas sockets, de "objetos de jogo", "snapshots", etc.

Para a persistência dos dados no servidor optou-se por utilizar a biblioteca Prevayler versão 1.02. O Prevayler garante a persistência de objetos Java de forma relativamente transparente, eliminando a necessidade de se mapear os objetos em outro paradigma (relacional) ou de se executar um "container" ou banco de dados juntamente com o servidor, o que simplifica bastante a distribuição e a instalação do mesmo. A principal desvantagem é que o Prevayler exige que todos os objetos a serem persistidos devem caber na memória da máquina virtual. Portanto, neste protótipo, o tamanho do "mundo virtual" persistido não pode ultrapassar o limite da memória física da máquina que executa o servidor.

4.2.1 Visão geral das principais classes

As principais classes que implementam o servidor são as seguintes:

- GameServer: classe principal do servidor, que gerencia instâncias das classes listadas abaixo;
- SegmentManager: gerencia um segmento (assinantes, snapshots, assinaturas, informação de presença, etc);
- Player: representa uma conta de jogador (identificador, nome do usuário, senha, etc) bem como trata a conexão com um jogador autenticado;
- SegPlayer: representa um "assinante" remoto de um segmento, ou seja, a associação de um Player com um SegmentManager. Trata as mensagens enviadas por um cliente no contexto de uma assinatura (ver classe Subscription).

As principais classes do cliente são as seguintes:

- GameClient: classe principal do cliente, que gerencia instâncias das classes listadas abaixo;

- Subscription: gerencia uma assinatura de segmento, tratando da comunicação com o SegmentManager associado (servidor), bem como da comunicação com os outros assinantes do segmento (ver classe Peer, abaixo);

- Peer: representa, no contexto de uma assinatura (Subscription), um assinante remoto (identificado por uma instância da classe PeerPlayer);

- Simulator: no contexto de uma assinatura (Subscription), gerencia a execução do algoritmo de simulação peer-to-peer replicada, tratando da geração e da leitura dos comandos dos jogadores, bem como da atualização da cópia local do segmento (a comunicação em si é tratada pela classe Subscription);

- SimPlayer: representa um assinante no contexto de um simulador local (Simulator), sendo responsável por armazenar o histórico de comandos recebidos de um assinante remoto;

As seguintes classes são utilizadas tanto pelo cliente, quanto pelo servidor:

- GameObject: representa um "objeto de jogo" (mago, monstro, e outros objetos auxiliares que não são exibidos para os jogadores);

- Segment: representa um "segmento" do mundo virtual, ou seja, uma coleção de GameObjects, juntamente com um "timestamp", que é o valor do relógio de simulação que está associado ao estado atual dos objetos representados.

- Frame1: implementa a interface gráfica do jogo.

A figura 4.5 ilustra as relações entre as classes através de um exemplo. A figura mostra o estado de uma rede FreeMMG com dois clientes conectados no servidor, sendo que ambos estão assinando o mesmo segmento. O servidor (GameServer) possui uma única conexão TCP com cada cliente, sendo que a thread que trata as mensagens do cliente fica na classe Player, e a thread que trata as mensagens do servidor fica na classe GameClient. Cada "assinatura" (Subscription) de um cliente também possui o seu próprio canal de rede com o servidor, que é tratado no servidor pela classe SegPlayer. Porém este canal de rede (entre um Subscription e um SegPlayer) é "virtual", sendo emulado sobre a conexão TCP entre as classes GameClient e Player. Isto foi feito para que fosse reduzido ao máximo o número de sockets TCP ativas que o servidor precisa gerenciar. A classe SegPlayer faz apenas um tratamento básico das mensagens recebidas pelo cliente, geralmente repassando as mensagens para que sejam tratadas pela classe SegmentManager. A classe SegmentManager é responsável por gerenciar as "mensagens coletivas" que são enviadas pelos assinantes, bem como gerenciar o andamento de assinaturas, desassinaturas, obtenção de *snapshots* periódicos, detecção e recuperação de falhas, etc.

A classe Subscription gerencia, no cliente, a rede de conexões "peer-to-peer" com os outros assinantes do segmento, bem como gerencia a conexão do assinante local com o servidor. Cada Subscription gerencia vários objetos Peer, onde cada um representa uma conexão com um dos outros assinantes remotos. Desta forma, os "grupos" de simuladores de um mesmo segmento se conectam de forma direta, em uma malha completa de conexões entre todos os membros do grupo. A conexão entre dois objetos Peer é uma nova conexão TCP/IP entre dois clientes. Ou seja, dois clientes podem compartilhar múltiplas conexões TCP/IP entre si, sendo cada uma no contexto de uma assinatura diferente.

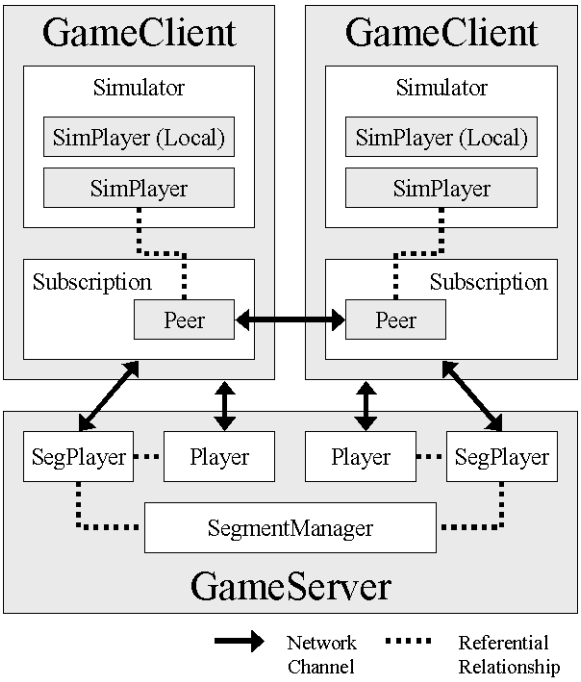


Figura 4.5: Estado de uma rede FreeMMG com dois clientes assinando um segmento

Cada instância de Subscription gerencia exatamente uma instância da classe Simulator. A classe Simulator e a classe SimPlayer implementam o algoritmo de "simulação peer-to-peer replicada" para um segmento, sem se preocupar com os detalhes da comunicação (se os outros jogadores são remotos ou locais, se usam sockets TCP/IP ou UDP multicast, etc). Um Simulator gerencia várias instâncias de SimPlayer, onde cada SimPlayer representa um jogador que está participando da simulação, ou seja, do qual são esperados comandos para que a simulação possa prosseguir. O Simulador aloca um SimPlayer para cada assinante remoto. Também é alocado um SimPlayer para representar um assinante local, caso o assinante local seja um assinante interessado em interagir com o segmento e não um assinante "aleatório", por exemplo. Cada SimPlayer mantém um registro dos comandos recebidos do cliente associado, de forma que o Simulator tem como determinar se possui todas as entradas necessárias para incrementar o relógio do simulador local e prosseguir com a simulação.

4.2.2 Algoritmo de simulação peer-to-peer replicada

O algoritmo de simulação replicada implementado é bastante similar ao algoritmo descrito por Bettner e Terrano e aplicado no jogo Age of Empires II (discutido em detalhes no capítulo 2). A principal diferença é que, no Wizards, o tamanho do "passo" do simulador é fixo em 100ms (em Age of Empires II, o passo do simulador se adapta aos recursos dos jogadores, variando tipicamente entre 50ms e 200ms, em passos de 50ms). O "lookahead" do simulador, para testes, foi configurado para dois turnos, o que permite o mascaramento da latência dos jogadores ate 200ms. Porém, esta configuração não foi testada em um cenário real de uso, e é provável que o valor do "lookahead" tenha que ser aumentado.

4.2.3 Gerenciador de segmentos (SegmentManager)

Como discutido no capítulo 3, os algoritmos relacionados à gerência de assinantes, que são coordenados pelo servidor, possuem restrições quanto à concorrência. Para evitar problemas, optou-se pelo processamento seqüencial destes algoritmos. Cada SegmentManager possui uma fila de requisições de quatro tipos: assinatura de cliente, desassinaturas de cliente, transferência de objetos entre segmentos, e recuperação de falha. Cada SegmentManager possui uma thread dedicada ao processamento destas requisições, sendo que a thread processa uma requisição de cada vez, de forma seqüencial.

O caso mais problemático que foi observado em relação à serialização das requisições é relativo à transferência de grandes quantidades de objetos entre dois segmentos, simultaneamente. Se um grupo grande de objetos cruza a fronteira entre dois segmentos de forma simultânea (ou quase), os objetos irão "sumir" simultaneamente do segmento de origem, mas aparecerão gradualmente no segmento destino. Além disso, as outras requisições, como assinaturas e desassinaturas, ficam impedidas de executar até que todos os objetos sejam transferidos.

Este problema pode ser minimizado de várias maneiras. Em primeiro lugar, a otimização mais óbvia é a transferência "em lote" de objetos, em uma mesma requisição. Ou seja, quando um SegmentManager for processar uma transferência de objeto, ele retira do início da fila todas as requisições de transferência de objetos e resolve todas de uma só vez. Outra possibilidade é a exploração da concorrência entre as requisições, que foi explorada superficialmente pelo modelo e constitui um trabalho futuro.

4.2.4 Alocação de assinantes "extras" e protocolo de recuperação de falhas

Neste protótipo, os seguintes componentes do modelo, relativos à prevenção e tratamento de falhas, não foram implementados:

- Alocação de assinantes "extras" (assinantes "aleatórios", etc) para que uma quantidade mínima de assinantes esteja presente em cada segmento ativo;
- Algoritmo inicial de diferenciação entre "falha parcial" e "falha fatal" (quantidade mínima dos vários tipos de assinantes após uma falha, etc);

Quando um ou mais assinantes falham, o servidor executa automaticamente o procedimento de recuperação de falha "parcial", sem considerar uma quantidade mínima de assinantes. Caso vários assinantes respondam com uma cópia válida, o servidor escolhe a mais atual. Caso nenhum assinante consiga fornecer uma cópia atualizada do segmento para a restauração, o servidor executa a restauração de falha "fatal" por snapshot restaura um "snapshot" (antigo) do segmento, armazenado localmente pelo SegmentManager correspondente. A implementação dos mecanismos de prevenção e tratamento de falhas é um trabalho em andamento no contexto do projeto FreeMMG, que atualmente trabalha na implementação de um framework reusável baseado no modelo FreeMMG.

4.3 Implementação do simulador de clientes

Para que fosse possível obter uma avaliação inicial sobre a escalabilidade potencial do FreeMMG, foi desenvolvido um simulador de clientes. O simulador de clientes é composto por um disparador de jogadores simulados (classe BatchClient) e um jogador simulado (classe ClientSimulator), que tenta simular o comportamento de um único jogador humano. O disparador de clientes executa um número configurável de jogadores

simulados, cada um na sua própria thread. Cada jogador simulado conecta em um servidor FreeMMG, cujo endereço é passado para o disparador.

O jogador simulado utiliza a mesma implementação do cliente FreeMMG que é utilizada pela interface gráfica. A diferença é que o jogador simulado sintetiza uma seqüência de comandos que pode ser gerada por um cliente controlado por um jogador humano. Os seguintes comportamentos sintéticos foram implementados:

- Estabelecimento da conexão com o servidor. Durante os experimentos, utilizou-se a criação de uma nova conta por cada jogador simulado, mas também há suporte para que o cliente simulado se autentique em uma conta já existente;
- Pedido de recolocação do mago em jogo (menu “Player”, opção “Respawn”, na interface gráfica);
- Criação de monstros (no mesmo segmento em que é colocado o mago);
- Movimentação aleatória do mago e dos monstros para segmentos adjacentes;

Por um lado, estes comportamentos são muito simples. A modelagem do comportamento fiel de jogadores humanos é um problema muito difícil, especialmente para tipos experimentais de jogos, que não possuem muitos casos conhecidos, como é o caso dos jogos de estratégia “maciçamente multijogador” suportados pelo FreeMMG. Por outro lado, o FreeMMG Wizards, na sua versão atual, é praticamente um jogo “sintético” pois não possui complexidade e objetivos suficientes para cativar um jogador humano, então a complexidade dos comportamentos, nestas circunstâncias, pode ser considerada adequada para a realização dos experimentos.

4.4 Experimentos realizados e resultados obtidos

Todos os testes utilizaram um mundo com 25 segmentos (grade 5x5). Foram realizados testes com todos os comportamentos de clientes simulados mas, para os resultados apresentados nesta seção, não foi utilizado o comportamento de movimentação de objetos entre segmentos, de forma que cada jogador simulado assina um único segmento de cada vez. Testes com movimentação de objetos entre segmentos foram executados, mas devido a problemas da implementação, estes testes não puderam ser executados de forma confiável para um número elevado de clientes simultâneos ou por um período significativo de tempo. Porém, durante estes testes limitados observou-se que o custo de rede do servidor não aumentaria, por exemplo, em uma ordem de magnitude ou mais.

Os experimentos utilizaram dois computadores Linux (kernel versão 2.6.5) com processadores Pentium IV e conectados por uma rede local. Um deles executou o servidor, e o outro executou o simulador de clientes. Com esta configuração foram executadas simulações com durações de 5, 10, 20, 30 e 60 minutos. Para cada duração foram feitas simulações com 50, 100, 150, 200 e 300 clientes simulados. Cada caso foi executado 20 vezes, sendo obtida a média. Foram consideradas execuções bem-sucedidas apenas as execuções em que o servidor não detectou nenhuma falha (de cliente ou do próprio servidor).

Os experimentos foram concentrados na medição da largura de banda utilizada, tanto pelos clientes quanto pelo servidor. Não foi gerado um gráfico de utilização de CPU, mas o uso de CPU foi verificado durante os experimentos para que não houvesse uma saturação que pudesse prejudicar os resultados obtidos. Foi verificado que a utilização

da CPU de ambas as máquinas manteve-se, em geral, sempre abaixo de 80%, tanto a máquina cliente, quanto a máquina do servidor, sendo 10% de utilização de CPU um valor mais comumente observado na máquina servidor, e 50% o valor mais aproximado do uso de CPU na máquina cliente. Esta tendência foi observada com a utilização do Linux kernel 2.6, que possui um bom gerenciamento de grandes quantidades de threads. Em experimentos iniciais, as máquinas executaram com o Linux kernel 2.4, que cria um processo para cada thread. Como o protótipo faz uso pesado de threads (mais de 5.000 threads simultâneas na máquina cliente foram observadas com freqüência), nos primeiros experimentos a utilização de CPU pela máquina simuladora de clientes era completamente saturada com apenas 10 clientes simulados. Mas, como constatamos mais tarde, isto era apenas devido ao grande número de threads. Em geral, as threads do protótipo ficam a maior parte do tempo bloqueadas em I/O (leitura de socket), sendo que não existe nenhuma tarefa realmente pesada, em termos de processamento, para ser executada pelo cliente ou pelo servidor.

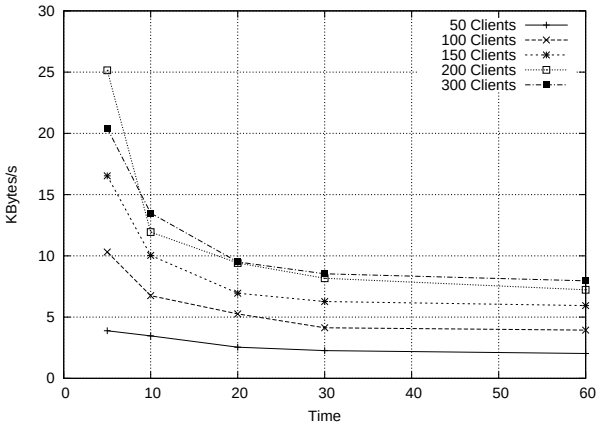


Figura 4.6: Tráfego no lado servidor, variando a quantidade de clientes e o tempo de execução

A figura 4.6 apresenta o tráfego no servidor durante a variação do tempo total do experimento e do número de clientes simulados. Pode-se observar que durante as simulações de menor duração a comunicação é maior e, nas simulações de longa duração, a comunicação é menor, sendo que o gráfico exibe uma tendência de estabilização. Isto ocorre porque no início da execução os clientes simulados realizam as assinaturas de segmentos e, após esta fase, o servidor é utilizado apenas para a criação de monstros em segmentos adjacentes ao do segmento em que o mago é criado. Isto também exercita o algoritmo de transferências de objetos e a comunicação com o servidor, pois a criação de um monstro é implementada como uma transferência de objeto entre o segmento onde está o mago e o segmento destino (que, no caso dos clientes simulados, é o mesmo segmento).

A figura 4.7 exibe o tráfego médio observado no servidor. Neste gráfico foram considerados apenas os resultados referentes aos experimentos com 30 e 60 minutos de duração, que compreendem um tempo suficiente para que as assinaturas iniciais já houvessem sido concluídas, apresentando uma comunicação mais estável ao fim dos períodos de medição.

A figura 4.8 exibe a medição do tráfego médio de um cliente durante os experimentos de duração de 5, 10, 20, 30 e 60 minutos. O valor individual, por cliente, foi obtido através da divisão do tráfego total gerado pelo processo simulador de clientes, dividido pelo número de clientes simulados.

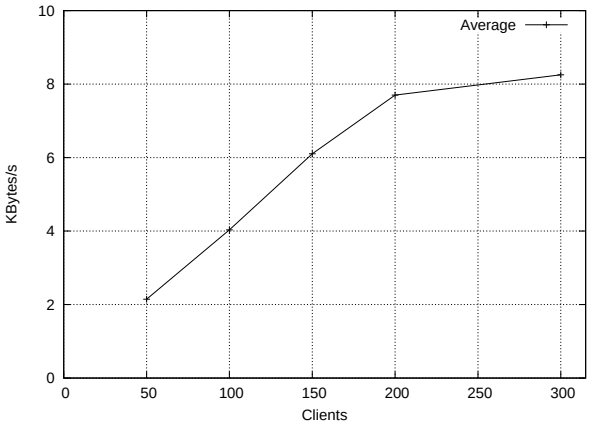


Figura 4.7: Tráfego no lado servidor, variando a quantidade de clientes

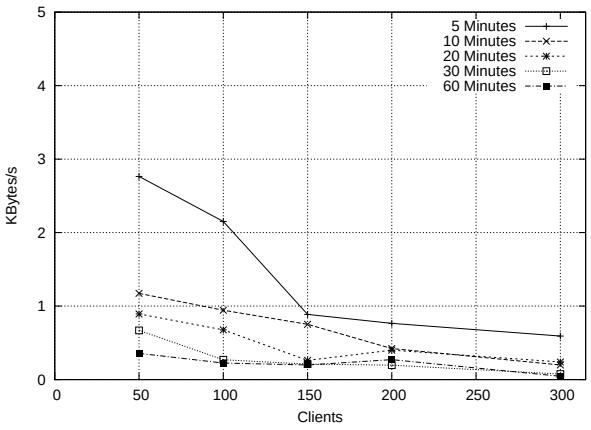


Figura 4.8: Tráfego médio do cliente, variando a quantidade total de clientes

4.5 Comparação dos resultados com o modelo centralizado

Para que os resultados obtidos possam ser colocados em perspectiva, esta seção compara os resultados experimentais obtidos com o desempenho provável de uma implementação hipotética de um jogo de estratégia em tempo real maciçamente multijogador em um paradigma de simulação centralizada. Em uma simulação centralizada, como é o caso da arquitetura cliente-servidor tradicional, ou mesmo da proposta Butterfly Grid (BUTTERFLY.NET, 2003), cada cliente envia constantemente pedidos (comandos) e constantemente recebe atualizações de objetos sobre a área de interesse do cliente. Tipicamente, jogos cliente-servidor baseados em “avatares” (ou controle de um único objeto dinâmico por jogador), como Quake e Ultima Online, consomem, de forma aproximada, entre 2 Kbytes/s e 5 Kbytes/s por jogador.

Isto é muito menos do que seria requerido por um jogo de estratégia em tempo real, com centenas de objetos dinâmicos a serem codificados em mensagens ou “deltas” de atualização de estado. Para fins de comparação, será utilizado o jogo Outgun, descrito no capítulo 2, que implementa um protocolo simples e otimizado para um jogo de ação cliente-servidor. No protocolo do Outgun, cada objeto dinâmico utiliza apenas 8 bytes para ser atualizado (posição, velocidade, aceleração atual e informações de animação do objeto). Atualizando cada cliente 10 vezes por segundo, o Outgun utiliza, em média, 0,5

Kbytes/s para atualizar cada cliente, quando apenas um objeto está visível para o jogador.

Se cada cliente possuir, em média, 200 objetos visíveis na sua área de interesse (uma quantidade normal para um jogo de estratégia multijogador convencional como Age of Empires II), o consumo de rede no servidor, por cliente, passaria para, no mínimo, 16 Kbytes/s, o que corresponde a 8 bytes por atualização multiplicados pela freqüência de 10 atualizações por segundo. Com a participação de 300 clientes simultaneamente, o uso de rede no servidor para atualização dos clientes seria então de 4,8 Mbytes/s, ou 38,4 Mbits/s (38.400 Kbits/s), o que corresponde a 16Kbytes/s multiplicados por 300 clientes.

A figura 4.6 mostra que o servidor FreeMMG com 300 clientes utilizou sempre menos do que 30 Kbytes/s ou 240 Kbits/s, ou seja, uma utilização de rede do servidor no mínimo 160 vezes menor (duas ordens de magnitude). A figura 4.8 também mostra que a utilização média da rede, pelos clientes, esteve sempre abaixo de 3 Kbytes/s, com uma média de 1 Kbyte/s após a estabilização, um valor bastante compatível com as tecnologias de acesso à Internet por "banda larga" já disponíveis para uso doméstico. Apesar de que cada cliente simulado só assinou um único segmento, muitas medidas ainda podem ser tomadas para que o uso de rede nos clientes seja reduzido, como a otimização relativa a clientes inativos descrita na seção 3.16.

Esta comunicação reduzida, tanto no cliente quanto no servidor, é devida ao protocolo de simulação replicada utilizado pelos FreeMMG que, além de isolar o servidor da maior parte do protocolo de simulação, utiliza os mesmos recursos de rede nos clientes para qualquer quantidade de objetos de jogo. A comparação acima pode ser estendida para 2.000 ou 20.000 objetos, apenas aumentando a vantagem do FreeMMG em mais ordens de magnitude. Porém, nestes casos, o custo do processamento de todos estes objetos, durante a execução do simulador nos clientes, pode passar a ser um problema, apesar de que o processamento dos objetos também seria problemático para o sistema cliente-servidor, que executa a simulação somente no servidor.

Por fim, é importante ressaltar que o protótipo implementado ainda não suporta a manutenção da consistência global em caso de falha. Como visto na seção 3.16 do capítulo anterior, uma possível solução envolveria o envio periódico para o servidor, por cada assinante, de "resumos de comandos". Este envio com certeza aumentaria significativamente o uso de rede do servidor FreeMMG e, por isso, os resultados obtidos, que apontam para uma boa folga em relação ao uso de rede do servidor, são bastante importantes para motivar a continuação do projeto.

4.6 Comparação entre o FreeMMG e trabalhos relacionados

Esta seção compara o FreeMMG com outros sistemas de suporte a MMGs que também buscam tratar dos principais problemas relacionados ao modelo cliente-servidor de suporte a MMGs. Para esta comparação foi escolhido o sistema Butterfly Grid (BUTTERFLY.NET, 2003), apresentado na seção 2.3, bem como os modelos Mercury (BHARAMBE; RAO; SESHAN, 2002) e NEO (GAUTHIERDICKEY; ZAPPALA; LO, 2004), e o modelo de Fiedler et al (FIEDLER; WALLNER; WEBER, 2002), apresentados na seção 2.6. A tabela 4.1 apresenta um resumo desta comparação, listando as características desejáveis em um modelo de suporte a jogos maciçamente multijogador e destacando os trabalhos que satisfazem estas características.

Como discutido na seção 2.3, o sistema Butterfly Grid funciona sobre uma infraestrutura complexa de hardware, composta de máquinas localizadas em domínios confiáveis e de interconexões de alta velocidade entre estas máquinas. Como mostra a

Tabela 4.1: Comparação entre o FreeMMG e trabalhos relacionados

Característica	**FreeMMG**	**Butterfly Grid**	**NEO**	**Mercury**	**Fiedler et al.**
Lado "servidor" (confiável) de baixo custo	•		•	•	•
Resistente a trapaças de alteração arbitrária do estado	•	•	parcial		
Trata gerência de grupos e interação entre grupos	•	•		•	•
Trata detecção e recuperação de falhas de clientes	•	•			
Trata autenticação de jogadores e persistência do mundo virtual	•	•	•		
Protocolo de simulação adaptado a jogos MMORTS	•		•		
Protocolo de simulação adaptado a MMGs de ação (MMORPGs e MMOFPSs)		•	•	•	•

tabela 4.1, todos os outros trabalhos analisados, inclusive o FreeMMG, não precisam de servidores confiáveis para simular, em tempo real, o "mundo virtual" do jogo. Os modelos FreeMMG e NEO dependem de "servidores confiáveis" apenas para tarefas de execução eventual, como é o caso da autenticação de usuários.

O sistema Mercury e a "arquitetura de comunicação para jogos maciçamente multijogador" de Fiedler et al tratam principalmente dos problemas relacionados à disseminação de informação de forma eficiente entre os participantes de um MMG. Neste sentido, estas arquiteturas oferecem suporte à gerência de grupos de clientes (adição e remoção de clientes em grupos), bem como oferecem suporte à interação entre clientes e objetos de jogo localizados em diferentes grupos. Contudo, ambas as arquiteturas não tratam dos problemas referentes a clientes trapaceiros e a falhas de clientes, além de não detalharem como seria feita a autenticação de clientes ou a persistência, de longo prazo, do estado do "mundo virtual" do MMG.

O modelo NEO é o que mais se aproxima do FreeMMG. Assim como o modelo FreeMMG, o modelo NEO emprega um protocolo de simulação, no contexto de um grupo de clientes, que é baseado no protocolo "lockstep" (GAUTHIERDICKEY; ZAPPALA; LO, 2004). O protocolo de simulação em grupos ou "segmentos" utilizado no FreeMMG é o "lockstep" com lookahead, apresentado na seção 2.5, que suporta jogos de estratégia em tempo real, mas não é adaptado para o suporte a jogos de ação (BETTNER; TERRANO, 2001).

Porém, o "protocolo NEO" elimina esta limitação ao realizar uma votação em cada turno. Basicamente, durante a troca de mensagens de um turno T, cada participante, além de enviar o seu comando para o turno T, informa de quais participantes ele já recebeu o

comando referente ao turno anterior (T - 1). Se a maioria dos participantes conseguiu receber o comando a tempo, ele é escalonado para execução no turno seguinte (T + 1), caso contrário, o comando é descartado. A votação dispensa a utilização de um protocolo de transporte confiável de mensagens, permitindo que o protocolo NEO seja implementado diretamente sobre UDP. Além disso, a votação elimina uma das principais desvantagens do protocolo "lockstep", que é a suspensão da simulação quando apenas um participante deixa de enviar um comando. Neste sentido, o protocolo NEO não necessita do "lookahead" para reduzir a ocorrência de "travadas" na simulação e, juntamente com o "dead reckoning" (TARR; JACOBS, 2002; GAUTHIERDICKEY; ZAPPALA; LO, 2004) que é executado em cada cliente, oferece suporte a jogos de ação.

O protocolo NEO é resistente a vários tipos de trapaças, inclusive a trapaça de alteração arbitrária do estado da simulação, que pode ser executada por jogadores malintencionados que participam de um mesmo grupo (ou "segmento"). Os autores de NEO referem-se a este tipo de trapaça como "collusion cheat" (GAUTHIERDICKEY; ZAPPALA; LO, 2004). Porém, o modelo NEO ainda não especifica como é feita a gerência de grupos e a interação entre clientes e objetos de jogo localizados em diferentes grupos, o que é uma característica essencial. Os autores de NEO ressaltam que vários problemas, inclusive relacionados a trapaças, encontram-se no caminho para a especificação destes serviços. Portanto, NEO é resistente a trapaças no contexto de um único grupo de clientes, mas esta característica ainda não está presente no modelo se considerarmos o suporte à interação entre os grupos, que ainda não foi modelado. Em contraste, o modelo FreeMMG fornece soluções resistentes a trapaças para a gerência de grupos (denominados "segmentos") e para interação entre os grupos, através da transferência de objetos entre segmentos. O modelo NEO também não especifica como é feito o tratamento das falhas dos clientes que, como discutido no capítulo 3, também caracteriza uma oportunidade para que jogadores trapaceiros realizem alterações arbitrárias no estado do jogo.

Por fim, destaca-se que, entre os modelos estudados, o FreeMMG é o único modelo que, além de suportar jogos maciçamente multijogador de estratégia em tempo real (jogos MMORTS), é resistente a trapaças e executa a partir de um "lado servidor" de baixo custo. O modelo NEO se aproxima destas características, mas não especifica vários mecanismos essenciais, como a gerência de grupos, o suporte à interação entre grupos e o tratamento de falhas de clientes, que já estão especificados no modelo FreeMMG e, até certo ponto, implementados no protótipo *FreeMMG Wizards* e validados através dos resultados obtidos.

4.7 Considerações finais

Este capítulo apresentou e avaliou a implementação do modelo, e utilizou os resultados experimentais obtidos para tecer argumentos que caracterizam o modelo como uma contribuição válida para o estudo científico do problema de suporte a jogos de tempo real, escaláveis e persistentes ("jogos maciçamente multijogador" ou MMGs). Foi apresentado um protótipo denominado "FreeMMG Wizards", que implementa um jogo simples de estratégia em tempo real maciçamente multijogador. A seguir, foram descritas as principais classes que implementam o protótipo, incluindo uma descrição do simulador de clientes, que foi utilizado na geração dos resultados. A seguir, os resultados dos experimentos foram apresentados, validando a utilização da rede FreeMMG para jogos com até 300 participantes simultâneos. Os resultados foram comparados com uma implementação hipotética de um jogo de estratégia maciçamente multijogador utilizando-se um

paradigma cliente-servidor, mostrando que, neste caso, a utilização da rede do servidor FreeMMG é ordens de magnitude menor. Por fim, o modelo é validado através da comparação de características (seção 4.6) com outras propostas de arquiteturas. Neste sentido, este capítulo mostrou que o modelo FreeMMG, além de possuir um desempenho satisfatório em relação ao problema que se propôs a tratar, possui originalidade em relação às características oferecidas, se comparado com práticas da indústria e com trabalhos científicos recentes da área.

5 CONCLUSÃO

O constante crescimento da disponibilidade do acesso à Internet em banda larga, bem como a popularização dos dispositivos móveis e tecnologias de acesso sem fim, contribuem para a popularização dos jogos distribuídos, ou "multijogador", e dos jogos distribuídos escaláveis, ou "maciçamente multijogador". Atualmente, percebe-se uma crescente demanda, na indústria de desenvolvimento de jogos, por novas tecnologias de suporte a jogos distribuídos. Neste sentido, também observa-se um crescente aumento em trabalhos de pesquisa, eventos e publicações científicas orientadas para o problema do suporte a jogos distribuídos.

Este trabalho insere-se no contexto de pesquisa de novas tecnologias de suporte a jogos maciçamente distribuídos na Internet. Foi mostrado que o problema do suporte a jogos competitivos é diferente do problema de suporte a, por exemplo, ambientes colaborativos, especialmente no que diz respeito à vulnerabilidade a alterações ilegais no estado da simulação distribuída do ambiente de jogo. Além disso, foram discutidas as desvantagens dos paradigmas de simulação atualmente empregados pelos desenvolvedores de MMGs, como o paradigma cliente-servidor e o paradigma Grid de execução, que centralizam todo o custo de simulação no "lado servidor" e aumentam os custos dos provedores de acesso aos jogos.

Neste contexto, foi proposto o FreeMMG, um modelo de suporte para jogos de estratégia em tempo real maciçamente multijogador. Foi demonstrado que o modelo é baseado, em parte, em uma arquitetura cliente-servidor, mas que o servidor empregado é escalável, através da distribuição da maior parte da simulação entre os próprios clientes (máquinas e conexões dos jogadores). Os resultados experimentais demonstraram que um servidor FreeMMG pode servir centenas de jogadores através de uma conexão de "banda larga" comum.

Por fim, em relação à originalidade do trabalho, pode-se dizer que o FreeMMG é, até onde foi pesquisado pelo autor, o único modelo que unifica as seguintes características:

- Arquitetura de suporte a jogos maciçamente multijogador (MMGs) de tempo real que depende apenas de um "lado servidor" de baixo custo (CPU e rede) como foi demonstrado pelos resultados experimentais obtidos;
- Suporte específico para jogos maciçamente multijogador de estratégia em tempo real, através da utilização de um protocolo conservador de simulação determinística e sincronização de entradas (comandos);
- Resistência a jogadores trapaceiros, por exemplo, através da replicação dos segmentos em vários assinantes e através da comunicação por "mensagens replicadas" durante a interação entre segmentos distintos.

O modelo FreeMMG continua em desenvolvimento, juntamente com uma nova implementação do mesmo, em formato *framework* reusável. Espera-se que a implementação do "FreeMMG Framework" seja útil como uma plataforma de suporte a jogos maciçamente multijogador de tempo real. Através da operação em servidores de baixo custo, espera-se que o FreeMMG beneficie, por exemplo, projetos de pesquisa relacionados a jogos escaláveis e projetos de desenvolvimento de jogos experimentais e estudo prático de "game design" aplicado a MMGs. Com o desenvolvimento de trabalhos futuros orientados à adição de mais mecanismos de segurança, o FreeMMG também pode se tornar uma alternativa para a aplicação em projetos de jogos comerciais, em especial de empresas iniciantes (ou independentes de publicadores) que, em geral, não possuem os recursos necessários para custear o "lado servidor" de um MMG cliente-servidor. Neste sentido, a implementação em software livre do "FreeMMG Framework" visa facilitar o envolvimento de empresas e outros pesquisadores interessados na evolução do modelo e da implementação do FreeMMG.

5.1 Contribuições

Identificam-se as seguintes contribuições no contexto deste trabalho:

- Desenvolvimento de um modelo de suporte para jogos maciçamente multijogador que unifica metas opostas como o suporte a jogos de estratégia em tempo real maciçamente multijogador, resistência a jogadores trapaceiros e utilização de "lado servidor" de baixo custo (relativo ao custo de processamento e custo de rede);

- Validação da escalabilidade do modelo proposto através da implementação dos principais componentes, para até 300 clientes conectados simultaneamente, sendo que o servidor não utilizou mais do que 30 Kbytes/s de recursos de rede em todos os testes, o que caracteriza um resultado significativo;

- Detalhamento de trabalhos futuros importantes, como a extensão do modelo para que haja garantias de consistência global e proteção contra jogadores trapaceiros através dos mecanismos de envio e, possivelmente, de processamento no servidor dos "resumos de comandos" (descrito na seção 3.16);

- Cinco publicações em eventos científicos (CECIN et al., 2004; CECIN; BARBOSA; GEYER, 2003a; CECIN et al., 2004; CECIN; GEYER, 2004; CECIN; BARBOSA; GEYER, 2003b), sendo a mais significativa (CECIN et al., 2004) um paper completo publicado na oitava edição do simpósio internacional DS-RT 2004 da IEEE;

- Participação na aprovação e na execução, em 2004, do projeto "FreeMMG: Um Framework Livre para Desenvolvimento de Massively Multiplayer Games", Edital MCT/CNPq/CT-INFO 01/2003, com a orientação de dois programadores para uma implementação reusável do FreeMMG ("FreeMMG Framework") e de um novo jogo demonstrativo (o jogo "Robonationz"), que caracterizam um trabalho em andamento;

- No contexto do projeto FreeMMG, um trabalho de conclusão de curso de graduação foi defendido (MARTINS, 2004) e três trabalhos de conclusão de curso de graduação (SANTOS MARTINS, 2004; FURSTENAU, 2004; IMMICH, 2004) e um trabalho de conclusão de curso de especialização (CARVALHO, 2004) encontram-se

em desenvolvimento. Além destes, dois trabalhos de disciplina relacionados a melhorias no modelo FreeMMG foram desenvolvidos (MENDES, 2004; COELHO, 2004);

5.2 Trabalhos futuros

O modelo FreeMMG pode ser melhorado em diversos aspectos. Alguns destes aspectos, como o tratamento de falhas por *snapshots* e *logs* de comandos (descrito na seção 3.16) já estão sendo implementados na primeira versão do *framework* reusável FreeMMG, cujo desenvolvimento é um trabalho em andamento. Outras possibilidades de melhoria, que ainda precisam ser modeladas, são relacionadas a seguir.

5.2.1 Novos mecanismos de segurança

O modelo FreeMMG oferece mecanismos de resistência e proteção contra alguns tipos específicos de trapaças, em especial as do tipo "alteração de estado" pois os clientes são responsáveis por executar a simulação e gerar as transferências de objetos entre os segmentos. Porém, existem várias maneiras de se implementar "trapaças" em um protocolo de jogo distribuído, sendo que modelos de simulação descentralizada ou *peer-to-peer* são especialmente vulneráveis. Neste sentido, um trabalho futuro consiste em estudar os vários tipos de "trapaças" conhecidos e publicados em textos científicos, e incorporar soluções no modelo FreeMMG.

5.2.2 Servidor distribuído

Os resultados obtidos nos experimentos apresentados no capítulo 4 apontam para uma escalabilidade do servidor FreeMMG, em relação aos recursos de processamento utilizados (CPU), para algumas centenas de jogadores. Porém, é provável que para quantidades de jogadores muito grandes, como centenas de milhares de jogadores, seja inviável ou muito custoso executar o servidor em uma única máquina.

Neste contexto, seria desejável a extensão do modelo para que múltiplas máquinas pudessem executar o "lado servidor" de uma rede FreeMMG (MENDES, 2004). Além disso, uma implementação do multi-servidor do FreeMMG permitiria a implementação de uma variante do algoritmo de recuperação de falhas por *snapshots* e *logs* de comandos (seção 3.16) que, ao executar os "resumos de comandos" de todos os jogadores em tempo real, garantiria a consistência global e a proteção contra trapaças oriundas de comandos e transferências ilegais de objetos, sem necessidade de alocação de assinantes "extras" ou "aleatórios" em cada segmento.

5.2.3 Particionamento transparente

No atual modelo FreeMMG, a partição do "mundo virtual" não é transparente. Isto ocorre devido aos atrasos na transferência de objetos e na impossibilidade de interação direta entre objetos localizados em "segmentos" diferentes. Idealmente, o particionamento realizado pelo mecanismo de distribuição de estado de um jogo multijogador deve ser transparente, ou seja, invisível aos jogadores.

A possibilidade da modificação do modelo para uma maior transparência do particionamento foi estudada de forma extensiva. Uma das alternativas estudadas aponta para a troca do algoritmo de "simulação *peer-to-peer* replicada" por uma variante otimista. Isto dispensaria a sincronização entre os comandos enviados por participantes de segmentos

adjacentes, permitindo que os simuladores de um segmento recebessem comandos de assinantes dos segmentos vizinhos. Porém, ainda não é claro como esta modificação poderia ser feita de forma a manter os outros benefícios, como a proteção contra trapaças e a escalabilidade do servidor.

5.2.4 Tratamento de *hot spots* no mundo virtual

Uma rede FreeMMG pode lidar com uma grande quantidade de jogadores simultâneos principalmente porque implementa um mundo virtual particionado. Se a distribuição dos jogadores e seus objetos de jogo no mundo virtual for uniforme, o desenvolvedor do jogo poderia simplesmente escolher um tamanho otimizado para os segmentos, de forma a manter o número médio de assinantes por segmento dentro de um intervalo adequado para o protocolo de simulação *peer-to-peer* replicada.

Porém, os mundos virtuais dos jogos maciçamente multijogador sofrem do problema dos *hot spots* (ou "pontos quentes"). Um *hot spot*, em um mundo virtual, é uma área onde há uma alta concentração de "avatares" ou objetos de jogo de muitos jogadores diferentes, causando a sobreposição de muitas áreas de interesse. Os MMGs cliente-servidor geralmente lidam com este problema através da utilização de arquiteturas multi-servidor e balanceamento dinâmico de carga (CUNNIN, 2000). No FreeMMG, um segmento é um *hot spot* se a quantidade de assinantes interessados se torna muito elevada (por exemplo, acima de 20 ou 30 assinantes), especialmente se todos os assinantes estão ativamente gerando comandos no contexto do segmento, sem a presença de assinantes inativos.

Foram estudadas três soluções para este problema. A primeira solução envolveria a subdivisão temporária de um segmento *hot spot* em segmentos menores. Esta solução não somente lidaria com a questão da superlotação de assinantes, como também permitiria que os segmentos fossem configurados, por *default*, para um tamanho grande, reduzindo o impacto da falta de transparência dos segmentos. Uma segunda opção seria a redução temporária da taxa de envio de comandos pelos jogadores, de forma análoga ao que é feito com os jogadores inativos, porém aplicando para todos os jogadores de forma simultânea. Por fim, a opção mais drástica envolveria um simples limite máximo de assinantes para cada segmento. Neste caso, se o número máximo é atingido, o servidor coloca os novos pedidos de assinatura em uma fila, para processamento quando novas vagas forem abertas. Estas soluções seriam combinadas para resolver o problema com o menor impacto possível aos jogadores.

5.2.5 Suporte a jogos de ação maciçamente multijogador

Inicialmente, o FreeMMG havia sido projetado para suportar MMGs em geral, sem considerar as diferenças entre um jogo de ação e um jogo de estratégia. O foco para "jogos de estratégia em tempo real" surgiu através do estudo do algoritmo de simulação *peer-to-peer* replicada, que garantia descentralização e resistência contra trapaças, mas não seria adequado para "jogos de ação", ou seja, baseados em avatares. A adaptação do FreeMMG para o suporte a jogos de ação, ou mesmo o desenvolvimento de um novo modelo que compartilhasse as características-chave do FreeMMG, como servidor com baixo custo (rede e CPU) e proteção contra trapaças, caracterizaria uma contribuição interessante, pois este tipo de modelo vai ao encontro das necessidades atuais da maioria dos MMGs atualmente no mercado. Neste sentido, a integração do protocolo NEO (GAUTHIERDICKEY; ZAPPALA; LO, 2004; GAUTHIERDICKEY et al., 2004) com os serviços oferecidos pelo modelo FreeMMG (gerência de grupos, interação entre grupos, etc) caracterizaria uma alternativa promissora.

REFERÊNCIAS

ALMEROTH, K. The evolution of multicast: from the MBone to inter-domain multicast to Internet2 deployment. **IEEE Network**, New York, v.14, p.10–20, Jan./Feb. 2000. Disponível em: <citeseer.ist.psu.edu/almeroth00evolution.html>. Acesso em: jan. 2005.

ALMEROTH, K.; AMMAR, M. **Multicast group behavior in the internet's multicast backbone (mbone)**. 1997. Disponível em: <citeseer.ist.psu.edu/almeroth97multicast.html>. Acesso em: jan. 2005.

ANDERSON, D. P.; COBB, J.; KORPELA, E.; LEBOFSKY, M.; WERTHIMER, D. Seti@home: an experiment in public-resource computing. **Communications of the ACM**, New York, v.45, n.11, p.56–61, Nov. 2002.

BASS, T. **Traffic congestion measurements in transit IP networks**. 1997. Disponível em: <citeseer.ist.psu.edu/bass97traffic.html>. Acesso em: jan. 2005.

BAUGHMAN, N. E.; LEVINE, B. N. Cheat-Proof Playout for Centralized and Distributed Online Games. In: ANNUAL JOINT CONFERENCE OF THE IEEE COMPUTER AND COMMUNICATIONS SOCIETIES, 20., 2002. **Proceedings...** [S.l.]: IEEE Press, 20., 2002.

BETTNER, P.; TERRANO, M. 1500 Archers on a 28.8: network programming in age of empires and beyond. In: GAME DEVELOPER'S CONFERENCE, 2001. **Proceedings...** [S.l.: s.n.], 2001. Disponível em: <http://www.gamasutra.com/features/20010322/terrano_01.htm>. Acesso em: jan. 2005.

BHARAMBE, A. R.; RAO, S.; SESHAN, S. Mercury: a scalable publish-subscribe system for internet games. In: NETWORK AND SYSTEM SUPPORT FOR GAMES, 2002. **Proceedings...** New York: ACM Press, 2002.

BIG HUGE GAMES. **Rise of Nations**. Disponível em: <http://www.microsoft.com/games/riseofnations/>. Acesso em: jan. 2005.

BLIZZARD. **Warcraft III**. Disponível em: <http://www.blizzard.com/war3/>. Acesso em: jan. 2005.

BLIZZARD. **Starcraft**. Disponível em: <http://www.blizzard.com/starcraft/>. Acesso em: jan. 2005.

BUTTERFLY.NET. **Powering Next-generation Gaming with Computing On-Demand**. White Paper. Disponível em: <http://www.butterfly.net/platform/technology/idc.pdf>. Acesso em: jan. 2005.

CAI, W.; XAVIER, P.; TURNER, S. J.; LEE, B. A Scalable Architecture for Supporting Interactive Games on the Internet. In: WORKSHOP ON PARALLEL AND DISTRIBUTED SIMULATION, PADS, 16., 2002. **Proceedings...** [S.l.]: IEEE Press, 2002.

CARLSSON, C.; HAGSAND, O. DIVE - a multi-user virtual reality system. In: IEEE VIRTUAL REALITY ANNUAL INTERNALTION SYMPOSIUM, 1993, Los Alamitos, CA, USA. **Proceedings...** [S.l.]: IEEE Computer Society Press, 1993. p.394–400.

CARVALHO, L. A. de. **Estudo de técnicas otimistas de sincronização para uso na plataforma para jogos distribuídos FreeMMG**. 2004. Trabalho de Conclusão de Mestrado (Especialização Redesis) — Instituto de Informática, UFRGS, Porto Alegre.

CASTRO, M.; LISKOV, B. Practical Byzantine Fault Tolerance. In: OSDI: SYMPOSIUM ON OPERATING SYSTEMS DESIGN AND IMPLEMENTATION, 1999. **Proceedings...** USENIX Association: Co-sponsored by IEEE TCOS and ACM SIGOPS, 1999. Disponível em: <citeseer.ist.psu.edu/article/castro99practical.html>. Acesso em: jan. 2005.

CECIN, F.; MIMICA, M.; ZAFFARI, P.; SLOMP, M.; JANNONE, R. **ZIG Game Engine**. 2004. Disponível em: <http://zige.sourceforge.net/>.

CECIN, F. R. **Massively Multiplayer Games**: alternativas para projeto e implementação. 2003. Trabalho Individual (Mestrado em Ciência da Computação) — Instituto de Informática, UFRGS, Porto Alegre. Disponível em: <http://www.inf.ufrgs.br/~fcecin/freemmg/>.

CECIN, F. R.; BARBOSA, J. L. V.; GEYER, C. F. R. FreeMMG: an hybrid peer-to-peer, client-server, and distributed massively multiplayer game simulation model. In: BRAZILIAN WORKSHOP ON GAMES AND DIGITAL ENTERTAINMENT, 2003, Salvador. **Proceedings...** Rio de Janeiro: SBC, 2003.

CECIN, F. R.; BARBOSA, J. L. V.; GEYER, C. F. R. FreeMMG: uma proposta baseada em peer-to-peer para simulação interativa competitiva em tempo real e distribuída em grande escala. **Cadernos de Informática**, Porto Alegre, v.3, 2003.

CECIN, F. R.; GEYER, C. F. R. FreeMMG: um modelo peer-to-peer distribuído de suporte para jogos massively multiplayer. In: ESCOLA REGIONAL DE ALTO DESEMPENHO, ERAD, 4., 2004, Pelotas. **Anais...** Pelotas: SBC, 2004.

CECIN, F. R.; MARTINS, M. G.; JANNONE, R. O.; BARBOSA, J. L. V.; GEYER, C. F. R. FreeMMG: a hybrid peer-to-peer and client-server model for massively multiplayer games. In: WORKSHOP ON NETWORK AND SYSTEM SUPPORT FOR GAMES, NETGAMES, 3., 2004. **Proceedings...** [S.l.: s.n.], 2004. Abstract.

CECIN, F. R.; REAL, R.; MARTINS, M. G.; JANNONE, R. O.; BARBOSA, J. L. V.; GEYER, C. F. R. FreeMMG: a scalable and cheat-resistant distribution model for internet games. In: IEEE INTERNATIONAL SYMPOSIUM ON DISTRIBUTED SIMULATION AND REAL TIME APPLICATIONS, DS-RT, 8., 2004. **Proceedings...** [S.l.: s.n.], 2004.

CECIN, F.; RITARI, N.; RIVINOJA, J.; PIMIÄ, V. **Outgun**. 2004. Disponível em: <http://outgun.sourceforge.net/>.

CLARKE, I. Protecting Free Expression Online with Freenet. **IEEE Internet Computing**, Los Alamitos, 2002.

COELHO, J. O. **Adição de funcionalidade ao FreeMMG Wizards**. 2004. Trabalho final da disciplina INF01008 — Instituto de Informática, UFRGS, Porto Alegre.

COHEN, B. **Incentives Build Robustness in BitTorrent**. 2003. Disponível em: <http://bittorrent.com/bittorrentecon.pdf>. Acesso em: jan. 2005.

COULOURIS, G. **Distributed Systems**: concepts and design. [S.l.]: Addison-Wesley, 2000.

CRONIN, E.; FILSTRUP, B.; ; JAMIN, S. Cheat-Proofing Dead Reckoned Multiplayer Games. In: INTERNATIONAL CONFERENCE ON APPLICATION AND DEVELOPMENT OF COMPUTER GAMES, ADCOG, 2003. **Proceedings...** [S.l.: s.n.], 2003.

CRONIN, E.; FILSTRUP, B.; KURC, A. R. **A Distributed Multiplayer Game Server System**. 2001. Disponível em: <http://warriors.eecs.umich.edu/games/papers/quakefinal.pdf>. Acesso em: jan. 2005.

CRONIN, E.; FILSTRUP, B.; KURC, A. R.; JAMIN, S. An Efficient Synchronization Mechanism for Mirrored Game Architectures. In: NETWORK AND SYSTEM SUPPORT FOR GAMES, 2002. **Proceedings...** New York: ACM Press, 2002.

CUNNIN, I. J. **Load Balancing Schemes for Distributed Real-Time Interactive Simulations**. 2000. Dissertação (Mestrado em Ciência da Computação) — University of Waterloo, Ontario, Canada.

EIXO do mal. 2004. Disponível em: <http://www.eixodomal.com.br/>. Acesso em: jan. 2005.

ENSEMBLE. **Age of Empires II**. 2003. Disponível em: <http://microsoft.com/games/age2>. Acesso em: jan. 2005.

FIEDLER, S.; WALLNER, M.; WEBER, M. A Communications Architecture for Massive Multiplayer Games. In: NETWORK AND SYSTEM SUPPORT FOR GAMES, 2002. **Proceedings...** New York: ACM Press, 2002.

FOSTER, I. What is the Grid? A Three Point Checklist. **GRID today**, [S.l.], v.1, July 22 2002. Disponível em: <http://www.gridtoday.com/02/0722/100136.html>. Acesso em: jan. 2005.

FUJIMOTO, R. M. **Parallel and distributed simulation systems**. New York: John Wiley, 2000.

FURSTENAU, R. J. **Um Cliente Móvel Para Monitoramento da Simulação Distribuída FreeMMG**. 2004. Trabalho de Conclusão (Bacharelado em Ciência da Computação) — Instituto de Informática, UFRGS, Porto Alegre.

GAUTHIERDICKEY, C.; ZAPPALA, D.; LO, V. A Fully Distributed Architecture for Massively Multiplayer Online Games. In: ACM SIGCOMM WORKSHOP ON NETWORK AND SYSTEM SUPPORT FOR GAMES, 2004. **Proceedings...** [S.l.: s.n.], 2004. Abstract.

GAUTHIERDICKEY, C.; ZAPPALA, D.; LO, V.; MARR, J. Low Latency and Cheat-Proof Event Ordering for Peer-to-Peer Games. In: ACM INTERNATIONAL WORKSHOP ON NETWORK AND OPERATING SYSTEM SUPPORT FOR DIGITAL AUDIO AND VIDEO, 14., 2004. **Proceedings...** [S.l.: s.n.], 2004.

GAUTIER, L.; DIOT, C. Design and Evaluation of MiMaze, a Multi-Player Game on the Internet. In: INTERNATIONAL CONFERENCE ON MULTIMEDIA COMPUTING AND SYSTEMS, 1998. **Proceedings...** [S.l.: s.n.], 1998. p.233–236.

GREENHALGH, C.; PURBRICK, J.; SNOWDON, D. Inside MASSIVE-3: flexible support for data consistency and world structuring. In: ACM INTERNATIONAL CONFERENCE ON COLLABORATIVE VIRTUAL ENVIRONMENTS, CVE, 3., 2000, San Francisco, USA. **Proceedings...** New York: ACM, 2000. p.119–127.

HANDLEY, M. **An examination of mbone performance**. 1997. Disponível em: <citeseer.ist.psu.edu/handley97examination.html>. Acesso em: jan. 2005.

HRISCHUK, C.; WOODSIDE, C. **Proper time**: causal and temporal relations of a distributed system. 1996. Disponível em: <citeseer.ist.psu.edu/article/hrischuk96proper.html>. Acesso em: jan. 2005.

ID SOFTWARE. **Quake**. 2004. Disponível em: <http://www.idsoftware.com/>. Acesso em: jan. 2005.

IGDA. **Online Games White Paper**. 2003. Disponível em: <http://www.igda.org/online/online_whitepaper.php>. Acesso em: jan. 2005.

IGDA. **Persistent World Games White Paper**. 2004. Disponível em: <http://www.igda.org/online/IGDA_PSW_Whitepaper_2004.pdf>. Acesso em: jan. 2005.

IMMICH, L. S. **Suporte a jogos multi-jogador, de ação, peer-to-peer, em dispositivos móveis, com um protótipo para J2ME**. 2004. Trabalho de Conclusão (Bacharelado em Ciência da Computação) — Instituto de Informática, UFRGS, Porto Alegre.

JOGALEKAR, P.; WOODSIDE, M. Evaluating the Scalability of Distributed Systems. **IEEE Transactions on Parallel and Distributed Systems**, New York, v.11, n.6, p.589, 2000. Disponível em: <citeseer.ist.psu.edu/jogalekar98evaluating.html>. Acesso em: jan. 2005.

KARLSSON, B. F. F.; FEIJÓ, B. An Overview on Security in Networked Computer Games. In: WORKSHOP BRASILEIRO DE JOGOS E ENTRETENIMENTO DIGITAL, WJOGOS/SBGAMES, 3., 2004. **Anais...** [S.l.: s.n.], 2004.

KELLER, J.; SIMON, G. Toward a Peer-to-peer Shared Virtual Reality. In: INTERNATIONAL CONFERENCE ON DISTRIBUTED COMPUTING SYSTEMS WORKSHOPS, ICDCSW, 22., 2002. **Proceedings...** [S.l.: s.n.], 2002.

KENT, S. L. **Making an MMOG for the Masses**. 2004. Disponível em: <http://www.gamespy.com/amdmmog/week3/>. Acesso em: jan. 2005.

KNUTSSON, B.; LU, H.; XU, W.; HOPKINS, B. **Peer-to-Peer Support for Massively Multiplayer Games**. 2004. Disponível em: <citeseer.ist.psu.edu/knutsson04peertopeer.html>. Acesso em: jan. 2005.

KUBIATOWICS, J. Extracting Guarantees from Chaos. **Communications of the ACM**, New York, v.46, p.33–38, 2003.

LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. **Communications of the ACM**, New York, v.21, p.558–565, July 1978.

LAMPORT, L. The weak Byzantine Generals problem. **Journal of the ACM**, [S.l.], v.30, n.3, p.668–676, July 1983.

LIN, J. C.; PAUL, S. RMTP: a reliable multicast transport protocol. In: INFOCOM, 1996, San Francisco, CA. **Proceedings...** [S.l.: s.n.], 1996. p.1414–1424. Disponível em: <citeseer.ist.psu.edu/lin96rmtp.html>. Acesso em: jan. 2005.

MACEDONIA, M. R.; ZYDA, M. J.; PRATT, D. R.; BARHAM, P. T.; ZESWITZ, S. NPSNET: A network software architecture for large-scale virtual environment. **Presence**, [S.l.], v.3, n.4, p.265–287, 1994.

MARTINS, M. G. **Empregando Estratégias Context-Aware em jogos na Pervasive Computing**: um estudo de caso sobre o freemmg. 2004. Trabalho de Conclusão (Bacharelado em Ciência da Computação) — UCPel, Pelotas.

MCGREGOR, D.; KAPOLKA, A.; ZYDA, M.; BRUTZMAN, D. Requirements for Large-Scale Networked Virtual Environments. In: INTERNATIONAL CONFERENCE ON TELECOMMUNICATIONS, 2003. **Proceedings...** [S.l.: s.n.], 2003.

MENDES, A. S. **FreeMMG Distributed Servers**. 2004. Trabalho final da disciplina INF01008 — Instituto de Informática, UFRGS, Porto Alegre.

MILLER, D.; THORPE, J. SIMNET: the advent of simulator networking. **Proceedings of IEEE**, New York, v.83, n.8, p.1114–1123, Aug. 1995.

MORSE, K.; BIC, L.; DILLENCOURT, M.; TSAI, K. **Multicast grouping for dynamic data distribution management**. 1999. Disponível em: <citeseer.ist.psu.edu/morse99multicast.html>. Acesso em: jan. 2005.

MORSE, K. L. **Interest Management in Large-Scale Distributed Simulations**. [S.l.: s.n.], 1996. (Technical Report ICS-TR-96-27).

NCSOFT. **Lineage**. 2003. Disponível em: <http://www.lineage.com/>. Acesso em: jan. 2005.

ORAM, A. **Peer-to-Peer**: harnessing the power of disruptive technologies. [S.l.]: O'Reilly and Associates, 2001.

ORIGIN. **Ultima Online**. 2004. Disponível em: <http://www.owo.com/>. Acesso em: jan. 2005.

PANTEL, L.; WOLF, L. On the Impact of Delay on Real-Time Multiplayer Games. In: INTERNATIONAL WORKSHOP ON NETWORK AND OPERATING SYSTEMS SUPPORT FOR DIGITAL AUDIO AND VIDEO, 12., 2002. **Proceedings...** New York: ACM Press, 2002.

PELLEGRINO, J. D.; DOVROLIS, C. Bandwidth requirement and state consistency in three multiplayer game architectures. In: NETWORK AND SYSTEM SUPPORT FOR GAMES, 2., 2003. **Proceedings...** New York: ACM Press, 2003. p.52–59.

PREISS, B. R.; LOUCKS, W. M. The Impact of Lookahead on the Performance of Conservative Distributed Simulation. In: EUROPEAN MULTICONFERENCE—SIMULATION METHODOLOGIES, LANGUAGES AND ARCHITECTURES, 1990, Nuremberg, FRG. **Proceedings...** [S.l.: s.n.], 1990. p.204–209.

REAL, R. **TiPS, uma proposta de escalonamento direcionada à computação pervasiva. 2004**. Dissertação (Mestrado em Ciência da Computação) — Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre.

ROBERTS, D. J. **A Predictive Real Time Architecture for Distributed Virtual Reality**. 1996. Tese (Doutorado em Ciência da Computação) — University of Reading, Reading.

ROBERTS, D.; WOLFF, R. Controlling Consistency within Collaborative Virtual Environments. In: IEEE INTERNATIONAL SYMPOSIUM ON DISTRIBUTED SIMULATION AND REAL-TIME APPLICATIONS, 2004, Budapeste, Hungria. **Proceedings...** [S.l.: s.n.], 2004. p.46–52.

SANTOS MARTINS, E. dos. **Identificação de um jogador malicioso para o FreeMMG**. 2004. Trabalho de Conclusão (Bacharelado em Ciência da Computação) — Instituto de Informática, UFRGS, Porto Alegre.

SCHODER, D.; FISCHBACH, K. Peer-to-Peer Prospects. **Communications of the ACM**, New York, v.46, p.27–29, Feb. 2003.

SCHULZRINNE, H.; CASNER, S.; FREDERICK, R.; JACOBSON, V. **RTP**: a transport protocol for real-time applications: RFC1889. [S.l.]: IETF, 1996.

SINGHAL, S.; ZYDA, M. **Networked Virtual Environments**: design and implementation. [S.l.]: Addison Wesley, 1999.

SONY. **EverQuest**. 2004. Disponível em: <http://www.everquest.com/>. Acesso em: jan. 2005.

SONY. **PlanetSide**. 2004. Disponível em: <http://www.planetside.com/>. Acesso em: jan. 2005.

TARR, R. W.; JACOBS, J. W. Distributed Interactive Simulation. In: SUMMER COMPUTER SIMULATION CONFERENCE. ORLANDO, FLORIDA, 2002. **Proceedings...** [S.l.: s.n.], 2002.

TOTAL WAR. **Rome**: total war. 2004. Disponível em: <http://www.totalwar.com/>. Acesso em: jan. 2005.

VALVE. **Half-life**. 2004. Disponível em: <http://half-life.sierra.com/>. Acesso em: jan. 2005.

VAN HOOK, D. J.; CALVIN, J. O. Approaches to Relevance Filtering. In: DIS WORKSHOP, 11., 1994. **Proceedings...** [S.l.: s.n.], 1994. p.367–369.

WEBER, T. S.; JANSCH-PÔRTO, I.; WEBER, R. F. **Tolerância a falhas**: conceitos e técnicas; aplicações; arquitetura de sistemas confiáveis. Notas de Aula da disciplina INF01209 do Instituto de Informática da UFRGS. 56p.