PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO

Arquitetura para Servidores de Jogos Online Massivamente Multiplayer

Ricardo Gomes Leal Costa

PROJETO FINAL DE GRADUAÇÃO

CENTRO TÉCNICO CIENTÍFICO - CTC

DEPARTAMENTO DE INFORMÁTICA

Curso de Graduação em Engenharia da Computação

Rio de Janeiro
Julho de 2009

Pontifícia Universidade Católica
do Rio de Janeiro

Ricardo Gomes Leal Costa

Arquitetura para Servidores de Jogos Online Massivamente Multiplayer

Projeto Final de Graduação

Relatório Final da disciplina Projeto Final II do Curso de Engenharia da Computação da PUC–Rio como requisito parcial para obtenção do título de Engenheiro de Computação.

Orientadora: Prof. Noemi Rodriguez

Rio de Janeiro
Julho de 2009

Agradecimentos

À minha família, pelo apoio que me deram durante o curso.
À Noemi, pela paciência e amizade.
Aos meus amigos, pela companhia e tudo que fizeram por mim.
Ao Clayton, por manter meu bom humor em todos os momentos.
Ao Roberto, por acreditar em nossos objetivos.
Ao José, por fazer pressão psicológica em momentos inoportunos.

Resumo

Costa, Ricardo; Rodriguez, Noemi. **Arquitetura para Servidores de Jogos Online Massivamente Multiplayer**. Rio de Janeiro, 2009. 44p. Relatório de Projeto Final II — Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Um jogo online massivamente multiplayer conta com milhares de jogadores conectados remotamente e interagindo entre si simultaneamente. Um sistema distribuído para a execução deste tipo de jogo deve atender a rígidos requisitos de desempenho, escalabilidade e tolerância a falhas. Neste trabalho, estudamos diversas arquiteturas existentes e propomos uma nova arquitetura que tira proveito dos recursos oferecidos pela linguagem de programação Lua. Desenvolvemos também uma ferramenta capaz de medir e analisar vários parâmetros de desempenho de um sistema distribuído. Os resultados foram muito satisfatórios, indicando que a arquitetura proposta poderia ser utilizada em jogos reais disponíveis atualmente no mercado.

Palavras–chave

Sistemas Distribuídos. Cluster de Servidores. Jogos Massivamente Multiplayer. Paralelismo. Desempenho de Servidores. Tolerância a Falhas.

Abstract

Costa, Ricardo; Rodriguez, Noemi. **Server Architecture for Massively Multiplayer Online Games**. Rio de Janeiro, 2009. 44p. Relatório de Projeto Final II — Department of Informática, Pontifícia Universidade Católica do Rio de Janeiro.

A massively multiplayer online game has thousands of players connected remotely and interacting simultaneously. A distributed system for this kind of game must conform to several requisites of performance, scalability and fault tolerance. In this work, we studied many existing architectures and then proposed a novel architecture that takes advantage of the features offered by the Lua programming language. We also developed a new tool to measure and analyze performance parameters of a distributed system. The results were very satisfactory, meaning that our proposed architecture could be used in real games currently available on the market.

Keywords

Distributed Systems. Server Cluster. Massively Multiplayer Games. Parallelism. Server Performance. Fault Tolerance.

Sumário

Lista de figuras

Lista de tabelas

1
Introdução

Com o crescimento do número de usuários com acesso à Internet e da qualidade desse acesso, surgiu um novo tipo de produto no mercado de entretenimento digital, denominado *Massively Multiplayer Online Game* (MMOG ou simplesmente MMO). Este tipo de jogo aproveita a explosão de conectividade para unir centenas ou mesmo milhares de jogadores num universo virtual persistente, onde todos interagem entre si em tempo real.

Os primeiros jogos MMO foram os *Role Playing Games* (RPG). Num RPG comum, o jogador encarna um personagem que desempenha um determinado papel no mundo fictício, devendo cumprir alguma missão sugerida pelo enredo. Ele pode interagir com outros personagens, sejam eles de outros jogadores ou *Non-Player Characters* (NPC) — personagens controlados diretamente pela inteligência artificial do jogo. A idéia de um MMORPG surge naturalmente na tentativa de estender a possibilidade de interação para jogadores geograficamente distantes.

O primeiro MMORPG gráfico, chamado "Neverwinter Nights" [Wik], foi lançado em 1991 para usuários da AOL, famoso provedor de Internet americano. Uma de suas novidades era o modo de jogo *Player versus Player* (PvP), possibilitando aos jogadores que se atacassem. Desde então, diversos MMORPG foram lançados, e outros gêneros de jogos MMO foram criados, como os de *Real-Time Strategy* (RTS) ou *First-Person Shooter* (FPS).

Nos tempos atuais, o MMOG de maior sucesso — "World of Warcraft" (WoW) [Bliz] — conta com mais de 9 milhões de usuários cadastrados, tendo mais de 600 mil online simultaneamente nos horários de pico. A *Blizzard Entertainment*, responsável pelo WoW, assim como a maioria das empresas desenvolvedoras de MMOG, cobra algum tipo de assinatura dos jogadores, movimentando anualmente enormes quantidades de capital.

Se analisarmos a evolução da quantidade de jogadores de MMOG ao longo do tempo (Fig. 1.1), é fácil perceber que este mercado tem grande potencial de expansão. De 1997 até o início de 2008, mais de 16 milhões de jogadores entraram nesse mercado. A tendência para o futuro próximo é manter o ritmo de crescimento acelerado com a popularização do acesso à Internet.

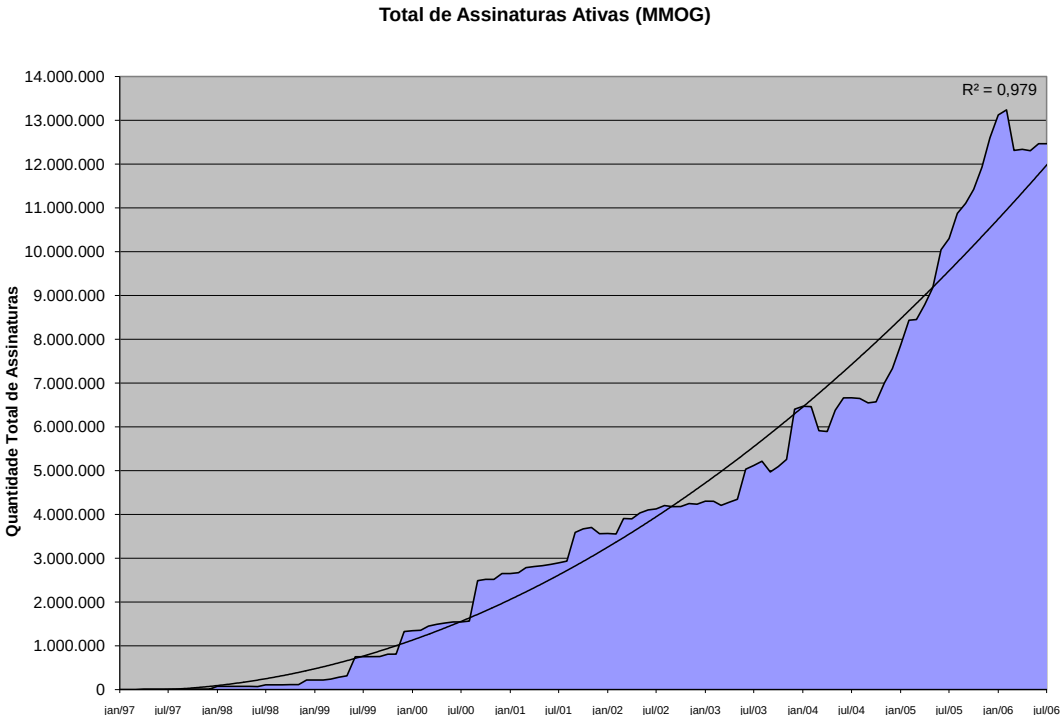

Figura 1.1: Número de Assinantes x Tempo (fonte: mmogchart.com).

Devemos notar então que essa área merece a atenção da indústria de software. Como se trata de um grupo muito grande de jogadores interagindo num mundo persistente, os requisitos de escalabilidade, desempenho, tolerância a falhas e segurança são muito elevados. A arquitetura trivial de cliente-servidor com uma única estação servidora não é suficiente nesse contexto, tornando-se essencial o desenvolvimento de paradigmas alternativos.

Este projeto teve por objetivo estudar e desenvolver uma arquitetura propícia para um jogo MMORPG, tendo-se em mente os requisitos de escalabilidade para suportar o maior número possível de jogadores conectados e de tolerância a falhas. Também estudamos tópicos de segurança, incluindo proteção contra trapaças, que são necessários para um sistema de jogo comercial.

Analisamos diversas propostas de arquiteturas que tentam atender aos requisitos supracitados. Os dois principais tipos de arquitetura são o cliente-servidor e o peer-to-peer. Cada um apresenta vantagens e desvantagens, como veremos mais adiante. De qualquer forma, identificamos um problema comum a ambos, que é a complexidade de implementação ao utilizarmos linguagens tradicionais como C++.

Um de nossos pontos de pesquisa então foi averiguar se o uso da linguagem de programação de alto nível Lua facilita a implementação dessas arquiteturas, sem impacto significativo no desempenho. Por ser uma linguagem interpretada, é possível fazer "hot swap", ou seja, carregar e alterar o código em tempo de execução. Além disso, é uma linguagem flexível, fácil de aprender e que permite um rápido desenvolvimento de aplicações. Em compensação, pode apresentar desempenho inferior às linguagens pré-compiladas, o que poderia tornar inviável seu uso num sistema de MMO.

Um grande problema que tivemos foi encontrar uma maneira de testar nossa arquitetura, o que resultou na necessidade de uma ferramenta capaz de medir o desempenho e a escalabilidade de clusters de servidores. Existem ferramentas complexas para medição de desempenho de servidores Web, mas precisávamos de algo mais genérico e fácil de usar. Assim, desenvolvemos nossa própria ferramenta de testes, chamada de *Massive Online Test* (MOT). Ela mostrou-se tão útil que pretendemos disponibilizá-la para uso público.

Este documento está organizado da seguinte forma: no Capítulo 2 descrevemos o modelo básico de um jogo MMO e seus principais requisitos; no Capítulo 3 estudamos as diferentes arquiteturas de rede existentes para MMOs; no Capítulo 4 especificamos o design, arquitetura e técnicas empregadas no nosso projeto; no Capítulo 5 explicamos como foi feita a implementação do projeto; no Capítulo 6 apresentamos os resultados dos testes de escalabilidade; e no Capítulo 7 fazemos as considerações finais.

2
Modelo Básico de um MMO

Um jogo MMO tipicamente é dividido em mundos independentes onde participam grupos distintos de jogadores. Um mundo possui partes imutáveis (terreno e algumas construções), partes mutáveis (objetos como veículos, armas, ferramentas, alimentos, etc), personagens controlados pelo jogador e NPCs controlados por inteligência artificial.

Dependendo do jogo, as partes imutáveis podem vir pré-carregadas no software do jogador (cliente), recebendo atualizações pela Internet quando necessário, ou então serem transferidas para o cliente na medida em que torna-se necessário utilizá-las (*on-demand*). As partes mutáveis são sempre enviadas on-demand, já que seu estado é constantemente atualizado.

O estado de um personagem é descrito pela sua posição no mundo e seus atributos (aparência, pontos de vida, habilidades, equipamentos, etc). As ações permitidas ao jogador em geral são mover-se, interagir com partes mutáveis e interagir com outros personagens.

Após um certo período de tempo, pode ser necessário restaurar o estado de alguns elementos do jogo, como NPCs inimigos que foram eliminados e tesouros que foram encontrados, para que novos jogadores possam participar das mesmas missões e obter os recursos que foram alterados em momentos anteriores. A maioria dos MMOs utiliza o sistema de "respawns" automáticos que recriam periodicamente os NPCs e itens removidos por outros jogadores. Pode também existir um "reset" do mundo do jogo que restaura seu estado inicial enquanto o servidor está *offline* para manutenção. O estado dos personagens é persistente e não é afetado por estes esquemas de restauração.

A parte de autenticação é importante e pode influenciar o funcionamento do jogo. Por exemplo, um usuário que pagou mais pode obter uma conta "premium" com benefícios para seu personagem que os outros jogadores não têm (acesso a setores especiais do mapa, uso de itens exclusivos, etc).

3
Arquiteturas de Rede para MMO

Analisaremos agora as vantagens e desvantagens das tradicionais arquiteturas de rede quando aplicadas a jogos MMO.

3.1 Arquitetura Cliente-Servidor Simples

Este é o modelo mais tradicional para aplicações online. Sua lógica é simples e fácil de ser implementada. Seu principal problema em relação aos jogos MMO é o excesso de carga depositada no servidor. Ele precisará atender a centenas ou milhares de conexões, autenticando-as e gerenciando o estado do jogo, o que exige um significativo esforço computacional. Embora os processadores modernos tenham grande capacidade de processamento, inclusive com múltiplos núcleos operando paralelamente, existem outras limitações e gargalos, como por exemplo o número máximo de conexões simultâneas permitidas pelo sistema operacional.

Além disso, devido à natureza centralizada, ele apresenta o indesejado *ponto único de falha* — qualquer falha no hardware ou na conectividade da estação servidora poderia derrubar instantaneamente todos os jogadores. Outra conseqüência ruim é a maior vulnerabilidade a ataques do tipo *Denial of Service* (DoS), que tornam o servidor inoperante. Por conta destes problemas, o uso de um único servidor é insuficiente para jogos MMO, embora seja freqüentemente utilizado em jogos multiplayer que se limitam a algumas dezenas de jogadores.

3.2 Arquitetura Cliente-Servidor com Cluster

Para contornar as limitações de hardware, podemos agrupar diversos servidores físicos — um *cluster* de servidores — atuando como um único servidor lógico. Isto permite o uso de técnicas como o *balanceamento de carga* e *failover*, que aumentam a escalabilidade e eliminam o ponto único de falha, respectivamente. Esta arquitetura é corriqueiramente adotada por servidores Web com grande demanda de usuários. Entretanto, quando tentamos adaptá-la ao mundo dos jogos MMO, descobrimos que algumas dificuldades aparecem.

O problema reside no fato de que é necessário compartilhar o estado do jogo entre os jogadores. No caso Web, o estado compartilhado geralmente encontra-se no banco de dados, que é ocasionalmente acessado pelos clientes quando executam alguma operação de leitura ou escrita de informação. O software de banco de dados possui mecanismos de exclusão mútua que garantem a ordem e consistência das operações. Em contrapartida, nos jogos MMO, o estado do jogo inteiro e de cada jogador está em constante atualização, o que poderia facilmente sobrecarregar o banco de dados.

Outra diferença é que a latência de resposta é um fator crítico em sistemas de tempo real como jogos. É aceitável esperar 1 ou 2 segundos para uma página web carregar, mas esta latência num jogo acabaria com a ilusão de tempo real. Portanto, é imprescindível que a sincronização ocorra no menor tempo possível. Assim, alguns modelos que levam em conta a estrutura do jogo são propostos para adaptar o uso de cluster de servidores aos jogos MMO, que descrevemos a seguir.

3.2.1 Cluster de Servidores com Replicação Total

A maneira mais direta é criar um grupo de servidores totalmente conectados entre si onde cada estação contém uma réplica completa do estado do jogo. As conexões dos jogadores são distribuídas com balanceamento de carga entre esses servidores, o que elimina a limitação do número máximo de conexões suportado pelo hardware.

Esta estratégia ainda assim requer um certo esforço computacional pelo fato de que o mundo inteiro está contido em cada servidor. Em particular, um único servidor deve ser escolhido como "mestre" para executar a lógica principal do jogo (algoritmos dos NPCs, cálculo dos danos causados em ataques, atualização das habilidades dos personagens, etc) com a finalidade de evitar inconsistências no mundo do jogo. Também seria de sua responsabilidade a autenticação e armazenamento do estado dos jogadores. Ele torna-se então o gargalo computacional do cluster, limitando sua escalabilidade.

Outra desvantagem é a grande quantidade de informações trocadas na rede interna do cluster, já que todos os servidores precisam manter-se atualizados quanto ao estado do jogo, e isso é feito através de broadcast de mensagens partindo do servidor onde o evento ocorreu. Um cuidado redobrado deve ser tomado para que as mensagens sejam entregues na mesma ordem a todos os servidores, caso contrário o estado do jogo poderia seguir rumos diferentes em cada servidor (imagine que um personagem dá um passo para a esquerda e depois outro passo para cima, seguindo um estreito caminho cercado por lava; se algum servidor do cluster receber a mensagem "passo para cima"

antes de "passo para a esquerda", ele poderia interpretar que o personagem deveria ter morrido ao pisar na lava).

Para evitar este problema, deve-se utilizar um algoritmo de ordenação total de mensagens no broadcast do cluster, ou então utilizar o servidor mestre como um intermediário para todas as mensagens trocadas no cluster, de maneira que ele possa reenviá-las para todos os demais servidores na ordem correta. Em ambos os casos, ocorrerá um "overhead" na rede interna pela quantidade adicional de mensagens trocadas.

3.2.2 Cluster de Servidores Baseados em Tarefa

Podemos dedicar cada servidor do cluster a uma tarefa específica. Por exemplo, um servidor contém apenas informações de autenticação, outro servidor contém a parte imutável do jogo, um terceiro servidor envia e recebe requisições de ações dos jogadores e um quarto servidor efetua cálculos de física ou inteligência artificial. Durante cada etapa do jogo, o cliente se conectará a um ou mais servidores que atendam às tarefas desejadas.

Este modelo resolve até certo ponto a questão do gargalo computacional, mas o número de conexões por servidor volta a ser um fator limitante, embora não tão grave quanto no caso de um servidor único, já que cada jogador não estará necessariamente conectado a todos os membros do cluster num dado momento.

Para resolver isso, podemos ter um grupo de servidores atuando como *front-end*, onde os clientes se conectam com balanceamento de carga. Este grupo está ligado pela rede interna ao *back-end*, composto por servidores baseados em tarefa que executam a lógica do jogo.

Assim como no cluster com replicação total, há a necessidade de troca de mensagens entre os servidores para sincronização dos eventos. Observe que a rede interna não precisa ser totalmente conectada, pois servidores de tarefas independentes não precisam comunicar-se.

3.2.3 Cluster de Servidores por Grupo de Interesse

Se levarmos em consideração o funcionamento de um jogo MMO, podemos notar que em geral existe uma certa localidade por parte de cada jogador. Cada personagem tem limitações em velocidade e alcance de visão, logo existe uma área de interesse para cada jogador determinada por esses limites. O uso desse tipo de contexto para evitar a troca desnecessária de informações é denominado *gerenciamento de interesses* [Mor96], o que equivale em termos práticos a fazer um multicast apenas para o grupo interessado em vez de um broadcast global.

Para aplicar o gerenciamento de interesses ao MMO, devemos dividir o mapa do mundo em vários setores ou regiões. Assim, cada servidor será responsável por apenas uma região, reduzindo efetivamente a carga por estação e aumentando a escalabilidade.

Grupo de Interesse com Divisão Geográfica (Zoned)

A maneira mais trivial de se dividir o mapa é considerando sua geografia. Como exemplo, imagine um mapa composto por duas ilhas e o continente principal, cortado por um rio. Cada ilha poderia ser atribuída a um servidor, e o continente poderia ser dividido nas duas partes separadas pelo rio, associadas a outros dois servidores.

Com esse tipo de divisão, os clientes podem conectar-se diretamente ao servidor correspondente a sua localização. As bordas entre as diferentes zonas do mapa são fisicamente inalcançáveis pelos personagens e, portanto, é sempre fácil saber em que servidor o cliente deverá conectar-se. Os eventos dentro do próprio jogo para troca de zona (a animação de um navio transportando os personagens, ou a travessia de um túnel) esconderiam a mudança de servidores por parte do cliente.

A técnica de divisão geográfica traz algumas desvantagens. Primeiro, não é possível balancear a carga dos servidores. Enquanto que uma ilha só é visitada por poucos jogadores, na outra pode estar ocorrendo um evento do tipo "Woodstock" que reune milhares de jogadores. Além disso, é altamente dependente da geografia do mapa, o que poderia trazer limitações ao design do mapa do jogo.

Grupo de Interesse com Divisão por Carga (Seamless)

O objetivo da divisão por carga é separar regiões do mapa de acordo com a média de usuários que permanecem ali, independentemente da geografia do mapa.

Existem dois modos de se fazer a divisão por carga. No *modo estático*, ela é feita enquanto o jogo está *offline* (durante o "reset", por exemplo), baseando-se em estatísticas coletadas automaticamente durante o jogo ou pelos próprios desenvolvedores.

Os clientes recebem durante a autenticação uma tabela que associa regiões aos endereços de servidor, e estabelecem conexão com um ou mais servidores, caso estejam localizados próximos às fronteiras de duas ou mais regiões. Os servidores trocam informações com seus vizinhos sobre eventos que afetam múltiplas regiões (uma explosão na fronteira que mata jogadores de ambas as regiões, por exemplo).

No *modo dinâmico*, a divisão das regiões é atualizada durante o andamento do jogo de acordo com seu estado atual. Naquele caso do Woodstock ocorrendo na ilha, vários servidores seriam selecionados para gerenciá-la, enquanto que o restante do mapa ficaria por conta de poucos ou apenas um servidor.

Em conseqüência de sua natureza dinâmica, os clientes não podem conectar-se diretamente aos servidores de região. Em vez disso, deve-se implementar um cluster front-end que atende aos clientes e troca informações apenas com os servidores do back-end responsáveis pelas regiões em que eles se encontram.

Com um número adequado de estações no front-end e no back-end, o único gargalo desta arquitetura seria a largura de banda da rede interna do cluster. Como atualmente a velocidade das redes locais pode atingir 10 Gbps com baixo custo, este problema pode ser facilmente resolvido.

Outra grande vantagem diz respeito à tolerância a falhas. No caso de um dos servidores de região falhar ou precisar de manutenção, os servidores restantes podem rapidamente cobrir a região pela qual ele era responsável, desde que pelo menos algum deles mantenha uma réplica do estado da região.

A principal desvantagem fica por conta da dificuldade de se implementar a lógica que gerencia a distribuição dinâmica de regiões do mapa entre os servidores. Ela deve contabilizar a população em cada parte do mapa, efetuar a divisão ótima e redistribuir o estado do jogo entre os servidores eleitos sem que os jogadores sejam afetados durante o processo.

Nos dois modos, o formato da região pode ser quadrado, retangular ou hexagonal, dependendo das características do jogo. O formato hexagonal apresenta o menor número médio de travessias de fronteira, dado que num ponto da fronteira podem existir no máximo 3 regiões visíveis (considerando-se que o tamanho do hexágono foi escolhido de maneira a ser maior que o alcance de visão do jogador).

3.3 Peer-to-Peer

A principal vantagem de uma rede peer-to-peer é a descentralização, inerentemente robusta em relação aos problemas que afetam a arquitetura cliente-servidor. Entretanto, uma série de novas questões surge com a descentralização.

Como será mantida a consistência do estado do jogo entre todos os jogadores? Jogos têm atualizações freqüentes que precisam ser propagadas sob certas restrições de tempo. Além disso, cada "peer" tem largura de banda limitada por serem terminais na rede Internet. Deve-se considerar ainda que

os jogadores podem desconectar-se a qualquer momento e, por conta disso, o estado de jogo que eles mantêm precisa ser replicado.

De que maneira será feita a troca de dados entre cada jogador? Em conseqüência do grande número de usuários, não é possível mantê-los totalmente conectados. Então, deve-se adotar políticas especiais para propagação das informações, com técnicas de roteamento e organização dinâmica da rede.

Será possível garantir a autenticidade e segurança dos dados de cada jogador? Como toda a lógica do jogo encontra-se no computador do jogador, ela é passível de adulterações com o intuito de trapacear ou obter informações privadas de outros jogadores.

Uma *arquitetura peer-to-peer estruturada* permite que técnicas de roteamento e organização sejam utilizadas com certa eficiência, ao mesmo tempo que a rede é escalável para um grande número de clientes — características atrativas para jogos MMO.

Existem diversos mecanismos de estruturação para redes de arquitetura peer-to-peer [Theo04]. O *Tapestry* [Zha01], por exemplo, oferece a funcionalidade de uma tabela hash distribuída, mapeando a chave de algum objeto a um único nó da rede. Ele suporta o roteamento de mensagens para clientes de maneira distribuída, automática e tolerante a falhas, oferecendo estabilidade ao driblar nós falhados e adaptar a topologia às circunstâncias com rapidez.

Veremos a seguir como utilizar esta técnica para o controle de estados do jogo, e quais métodos de tolerância a falhas podem ser implementados nesta arquitetura.

3.3.1 Controle de Estados

Uma maneira de implementarmos um MMO com o mecanismo Tapestry é sugerida em [Knut04]. Devemos primeiro delegar o estado persistente do usuário (informações da conta, lista de personagens, etc) a algum servidor centralizado, enquanto que todo o estado do jogo ficará distribuído na rede peer-to-peer.

O uso de gerenciamento de interesses também se aplica aqui, de maneira que cada área de interesse corresponde a um grupo de peers. Dessa forma, as mensagens relevantes são trocadas apenas entre os membros desse grupo.

O estado dos jogadores é acessado com uso do padrão "único-escritor múltiplos-leitores". Cada jogador atualiza sua própria posição quando se move, enquanto que interações entre jogadores só afetam os estados dos envolvidos. A notificação de mudança de posição é enviada por multicast em intervalos pré-definidos de tempo, para evitar excesso de mensagens. A fim de evitar

trapaças, a validação de cada uma dessas ações é feita por outro nó, escolhido como coordenador.

O estado das partes mutáveis do jogo também é controlada por coordenadores. Cada objeto tem um coordenador associado, para o qual são enviadas todas as atualizações relacionadas. Ele trata qualquer atualização conflitante e serve como um repositório para as informações do objeto. Após validadas, as atualizações são confirmadas por multicast de volta ao grupo.

As partes imutáveis vêm pré-instaladas em cada cliente, não sendo necessário haver troca de informações sobre elas entre os clientes. Atualizações podem ser obtidas de um servidor Web quando disponibilizadas pelo desenvolvedor.

Para a escolha dos coordenadores, cada região do mapa recebe um ID da rede peer-to-peer, computado por algum algoritmo de hash resistente a colisões (SHA-1, por exemplo). A partir daí, o cliente que tiver o ID mais próximo ao da região é eleito coordenador. Ele passa a servir como coordenador de todos os objetos da região e também como o seu distribuidor de multicast.

É interessante notar que é improvável que o coordenador de uma região faça parte dela no jogo, por conta da escolha aleatória de IDs. Isso reduz a possibilidade de trapaças já que ele não é responsável pela área de seu personagem. Além do mais, a independência de sua posição no mapa torna desnecessário efetuar a troca de coordenador até que o jogador saia do jogo. Outra vantagem é que o impacto de eventos numa determinada região — seja do jogo ou do mundo real — que causem a desconexão dos seus usuários tem menos chances de fazer o estado dela ser perdido.

3.3.2 Tolerância a Falhas

Numa rede peer-to-peer, sempre existe a possibilidade de algum nó se desconectar ou falhar. Portanto, é fundamental que exista replicação das informações presentes em cada nó. O número de réplicas pode ser ajustado de acordo com as necessidades, mas deve-se contar com pelo menos uma réplica para cada nó.

No caso de mecanismos como o Tapestry, podemos tirar proveito do seu método de roteamento. As mensagens referentes à região K são roteadas ao nó cujo ID é o numericamente mais próximo de K. Conforme mencionado acima, o coordenador de uma região será o nó cujo ID é o mais próximo do ID da região. Então, escolhemos como réplica o nó que tiver o ID mais próximo do ID do coordenador.

Por exemplo, se o ID da região é K, o nó numericamente mais próximo N será o coordenador, e a réplica ficará no nó M, que é o numericamente mais

próximo de N. Assim, se N falhar, as mensagens referentes a K, que seriam roteadas até M, passarão a ser roteadas a N.

Quando o nó N recebe uma mensagem que deveria ser entregue a M, significa que M falhou e portanto ele é o novo coordenador. Para criar a réplica de N, o nó envia, para o vizinho que tiver o ID mais próximo, uma requisição para que ele se torne sua réplica.

De maneira análoga, se um novo nó entrar e tiver um ID mais próximo ao da região que o coordenador atual, ele receberá a mensagem que era destinada ao coordenador e entenderá que se tornou o novo coordenador. Neste momento, ele pedirá ao coordenador atual a transferência de seu status acompanhado de todo o estado da região. Então, o coordenador antigo se torna a réplica do novo nó.

Esta técnica não resolve o caso de uma falha em todos os coordenadores e suas réplicas, já que isso implicaria na perda do estado daquela região. Contudo, considerando as probabilidades de um evento desses ocorrer de acordo com o número de jogadores e o tempo médio de uma sessão, pode-se ajustar o número de réplicas de maneira que a chance de uma falha catastrófica seja a menor possível.

4
Projeto e Especificação do Sistema

Este projeto é uma versão simplificada de um MMORPG, com foco na escalabilidade para um grande número de jogadores. Neste capítulo, definimos o funcionamento do jogo, assim como sua arquitetura geral e as técnicas utilizadas.

4.1 Design de Jogo

O mundo é representado por um mapa plano bidimensional onde os jogadores e NPCs podem caminhar. As posições do mapa são discretas, ou seja, o mapa pode ser considerado uma grade cujas posições válidas são dadas por um par de números inteiros (x, y). Cada posição pode conter apenas um jogador ou NPC num dado instante.

Os NPCs, que chamaremos de "monstros", caminham aleatoriamente pelo mapa até avistarem um jogador. Neste momento, o monstro andará em direção ao jogador e, quando chegar próximo o suficiente, fará um ataque. O jogador também pode efetuar ataques tanto nos monstros quanto em outros jogadores. Cada ataque infere um valor de dano ao alvo, que é subtraído de seu total de pontos de vida. Ele morrerá quando este total chegar a zero. Quando o jogador mata um monstro ou outro jogador, seu nível aumenta, o que torna seus ataques mais fortes. Por outro lado, quando o jogador morre, seu nível diminui.

4.2 Arquitetura do Sistema

A escolha pela arquitetura certa depende muito dos recursos disponíveis por parte do desenvolvedor, do número de usuários esperado e da complexidade aceitável do software. A arquitetura cliente-servidor simples é comumente adotada em jogos multiplayer limitados a algumas dezenas de jogadores por ser de fácil implementação mas, como já vimos, não é recomendada para suportar o cenário de um MMO com milhares de clientes. Já a arquitetura peer-to-peer tem pontos fortes em escalabilidade e desempenho, porém é pouco utilizada em MMOs por depender de algoritmos complexos e ainda apresentar muitos problemas nas áreas de tolerância a falhas e segurança.

Por outro lado, a arquitetura de cluster de servidores com divisão por carga é bastante utilizada pelos MMOs atualmente disponíveis no mercado em vista de suas diversas vantagens, ainda que possa ter também um alto grau de complexidade de desenvolvimento. Portanto, baseamo-nos nas idéias desta arquitetura para este projeto, sempre buscando maneiras de simplificar sua implementação.

O sistema proposto neste projeto é composto pelas seguintes entidades lógicas:

Cliente de Jogo: Aplicativo cliente utilizado pelos jogadores para se conectarem ao jogo.

Servidores de Login: Cluster de servidores que recebe diretamente as conexões dos clientes, autenticando-os para se conectarem nos servidores de jogo.

Banco de Dados: Banco de dados que armazena informações de autenticação e dos personagens de cada jogador.

Servidores de Jogo: Sistema que representa um dos diversos mundos de jogo. É dividido em clusters front-end e back-end.

A seguir, descrevemos detalhadamente cada uma destas entidades.

4.2.1 Cliente de Jogo

O aplicativo cliente é instalado no computador de cada jogador e funciona como a interface de acesso ao jogo. Ao iniciar o aplicativo, o jogador entrará com suas credenciais, e então uma conexão será feita a algum dos servidores de login. O cliente mantém uma lista com os endereços de cada servidor de login. É feita uma tentativa de conexão inicial aleatoriamente para algum dos servidores e, em caso de falha, novas tentativas são feitas em *round-robin* até que a conexão seja estabelecida com sucesso. Caso o jogador seja autenticado com sucesso, o servidor de login informará a lista de personagens do jogador, que deverá escolher um deles para jogar, e então o servidor de login enviará o endereço do servidor de jogo ao qual o cliente deve conectar-se.

Uma versão simplificada do aplicativo cliente também foi desenvolvida com o intuito de realizar testes de desempenho nos servidores.

4.2.2 Servidores de Login

Ao receber uma conexão, o servidor de login tenta autenticar o cliente buscando suas credenciais no banco de dados (ver Fig. 4.1). Em seguida, retorna ao cliente a confirmação de autenticação, redirecionado-o a algum dos servidores de jogo. Uma questão importante é que, no nosso projeto, o cliente não se autentica novamente quando conecta-se ao servidor de jogo. Este modo de autenticação apresenta brechas de segurança e não deve ser usado num jogo comercial. Uma descrição mais detalhada desses problemas, assim como propostas para solucioná-los, encontram-se na seção 4.5.

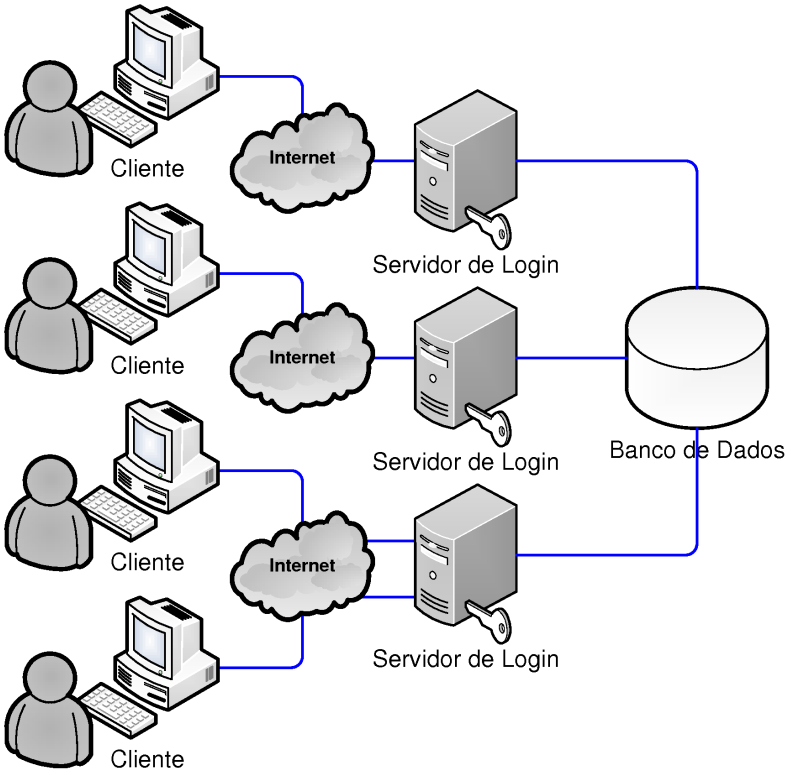

Figura 4.1: Topologia física dos servidores de login.

4.2.3 Banco de Dados

Para os fins deste projeto, o banco de dados é apenas uma lista de nomes e senhas. Nos sistemas reais, costuma-se utilizar um banco de dados SQL que armazena várias outras informações sobre o jogador e seus personagens, como por exemplo suas habilidades e itens adquiridos no jogo.

4.2.4 Servidores de Jogo

Cada grupo de servidores de jogo representa um único mundo de jogo. Ele é formado por dois clusters de servidores, o front-end (FE) e o back-end (BE), que são totalmente interconectados pela rede interna (ver Fig. 4.2). Os servidores BE são os que efetivamente gerenciam o estado do jogo e executam o balanceamento de carga. Os servidores FE recebem as conexões dos clientes e atuam como mensageiros, realizando a comunicação entre os clientes e os servidores BE responsáveis pela região em que seus personagens se encontram. É interessante notar que o protocolo cliente-servidor é conhecido apenas pelo FE, já que o BE não se comunica diretamente com os clientes.

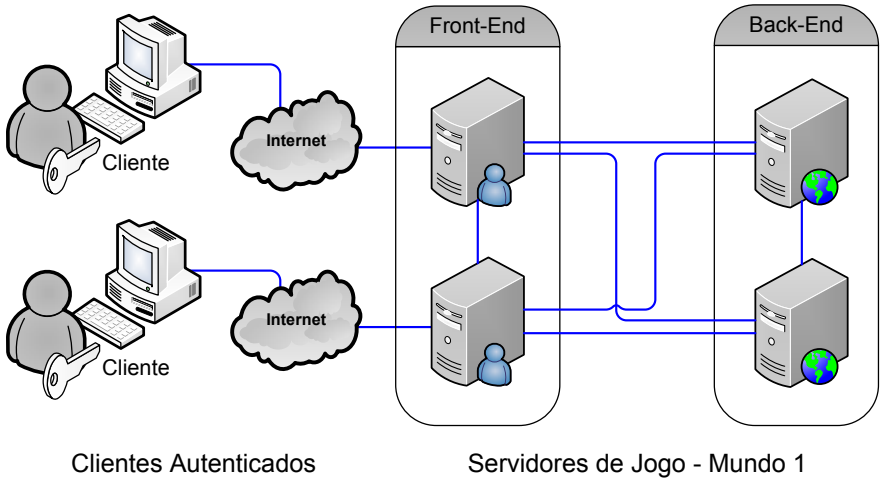

Figura 4.2: Topologia física dos servidores de jogo.

Observe que o grupo de servidores de jogo está associado a apenas um mundo de jogo específico. É possível utilizar o mesmo cluster para hospedar diversos mundos, porém cada mundo é isolado e independente um do outro.

4.3 Balanceamento de Carga

Os servidores BE dividem a responsabilidade pelas diferentes partes do mapa dinamicamente, de acordo com o esforço computacional necessário para gerenciá-las. A seguir descrevemos como este balanceamento é feito.

4.3.1 Divisão do Mapa

Para efetuar o balanceamento de carga, o mapa é dividido em uma grade regular composta de blocos de tamanhos iguais, que possuem informações sobre os jogadores e NPCs neles contidos. Cada bloco é associado a um único servidor

BE, responsável por gerenciar seu estado e executar tarefas computacionais, como as interações entre jogadores e os algoritmos de inteligência artificial dos NPCs daquele bloco. Cada servidor BE, por sua vez, pode ser responsável por um ou mais blocos do mapa.

Inicialmente, os blocos são distribuídos entre os servidores de acordo com uma configuração inicial, que pode ser automática (distribuição uniforme) ou manual (um arquivo diz quais blocos pertencem a cada servidor). Todos os servidores mantêm uma tabela de roteamento que indica qual servidor é responsável por cada bloco no mapa num dado instante.

Cada bloco tem um valor numérico associado que chamamos de "peso" (P). Este valor é proporcional ao número de elementos ativos no bloco, sejam jogadores ou NPCs. O peso total de um servidor com número de blocos igual a N_{blocos} é dado pelo somatório dos pesos de seus blocos:

$$P_{total} = \sum_{i=1}^{N_{blocos}} P_i$$

Por sua vez, cada servidor tem um "limite de peso" (P_{lim}), que pode ser configurado como um valor estático ou flutuar de acordo com certos parâmetros (uso de CPU, memória livre, etc). Os valores P_{total} e P_{lim} de cada servidor são compartilhados entre os demais servidores BE, que os armazenam em tabelas locais. Estes valores não são sincronizados em tempo real, mas sim periodicamente, com o intuito de evitar a troca de mensagens excessiva entre os servidores. Desta forma, os valores são apenas aproximações, embora isto não seja um problema para o uso a que são destinados, como veremos adiante.

O balanceamento de carga deve ocorrer apenas quando houver a possibilidade de ganho de desempenho nos servidores sobrecarregados. Portanto, precisamos primeiro determinar se um servidor está sobrecarregado. Definimos, então, o "fator peso" do servidor como sendo:

$$F_p = \frac{P_{total}}{P_{lim}}$$

Assim, um servidor com $F_p > 1$ está sobrecarregado. Neste caso, a sua rotina de balanceamento é chamada, que então decidirá o bloco a ser transferido e o servidor que irá recebê-lo. O algoritmo, que nós desenvolvemos, é descrito a seguir.

Inicialmente, o bloco de maior peso do servidor que está sobrecarregado é selecionado. Busca-se então o servidor menos carregado para o recebimento do bloco, com a condição de que, mesmo após receber o bloco, ainda terá um fator de peso menor que o do servidor atual. Observe que esta condição leva

Algoritmo 1 Balanceamento de Blocos do Mapa

$B_{lista} \Leftarrow$ lista de blocos ordenados por P decrescente
para $i = 1$ até N_{blocos} **faça**
  $P_{bloco} \Leftarrow$ peso de $B_{lista}[i]$
  para todo servidor back-end diferente do atual **faça**
    $F_{pp} \Leftarrow \frac{(P_{total} + P_{bloco})}{P_{lim}}$ // fator de peso potencial do servidor
  fim para
  $F_{pp_{min}} \Leftarrow$ menor F_{pp} encontrado
  se $F_{pp_{min}} + F_{epsilon} < F_p$ **então**
    transfira $B_{lista}[i]$ para o servidor de fator de peso potencial $F_{pp_{min}}$
    retorne verdadeiro
  fim se
fim para
retorne falso

em conta também o termo $F_{epsilon}$, que é uma constante positiva usada para compensar o custo do balanceamento, já que uma diferença muito pequena no fator de peso não chegaria a justificar a transferência do bloco. Caso nenhum servidor satisfaça a condição, é feita uma nova tentativa com o segundo bloco de maior peso, e assim por diante. Caso não seja possível transferir nenhum bloco, o balanceamento falha. Os detalhes da transferência são descritos na seção 4.3.2.

Independente de ter sido concluída com êxito ou não, esta rotina só poderá ser executada novamente após decorrido um intervalo mínimo de tempo definido na configuração, a fim de se evitar muitas tentativas de balanceamento num espaço curto de tempo.

É importante destacar que o algoritmo de balanceamento ocorre dentro da seção crítica de uma exclusão mútua distribuída, de maneira que nenhum outro servidor possa iniciar o balanceamento quando já houver um em andamento.

4.3.2 Transferência de Estado

A transferência de um bloco para outro servidor ocorre ainda dentro da seção crítica do balanceamento de carga, da seguinte forma:

1. Uma mensagem é enviada ao servidor de destino avisando que a transferência será realizada e que ele deverá, de agora em diante, armazenar numa fila todas as requisições relativas ao novo bloco. Em seguida, um pedido de atualização da tabela de roteamento é enviado por broadcast a todos os servidores BE, para que as requisições que dizem respeito a esse bloco sejam encaminhadas ao novo servidor.

2. O bloco é serializado e enviado para o novo servidor. Ao recebê-lo, as requisições pendentes que estavam armazenadas na fila são processadas. A partir desse ponto, todas as requisições seguintes são atendidas de imediato, não sendo mais necessário armazená-las.

3. Ao receber a confirmação do envio do bloco e da atualização da tabela de roteamento de todos os servidores, o servidor de origem apaga o bloco de sua memória.

4.4 Tolerância a Falhas

A tolerância a falhas tem por objetivo evitar a interrupção ou degradação significativa dos serviços por conta de problemas no servidor que os hospeda. Esta interrupção pode ocorrer não apenas quando acontece uma falha inesperada, mas também quando é prevista, como, por exemplo, no caso em que um servidor precisa ser desligado devido a uma manutenção preventiva. Em geral, o uso de clusters de servidores possibilita o uso de técnicas de replicação e *failover* para conseguir alta disponibilidade.

O cluster dos servidores de login é naturalmente tolerante a falhas programadas ou inesperadas, pois quando um deles falha, o cliente tentará automaticamente conectar-se a outro servidor que esteja disponível.

No caso de um servidor de jogo front-end, um grande problema é o fato de que existem clientes conectados a ele. Assim, esses clientes serão desconectados e deverão reconectar-se a algum outro servidor FE. Note que apenas os clientes que estavam naquele servidor FE específico precisarão reconectar, os demais não serão afetados. Assim, quanto maior o cluster, menor é o impacto gerado pela perda de um servidor.

Os servidores de jogo back-end, por outro lado, contêm informações sobre o estado do jogo, o que torna necessária a replicação das informações para ser capaz de contornar uma falha inesperada. Uma maneira comum de se fazer isso é ter um servidor espelho que recebe todas as mudanças de estado do servidor primário e entra em ação apenas quando este último para de funcionar. Nós decidimos não implementar esta funcionalidade por acreditarmos que não valeria o esforço necessário para este projeto. Porém, adicionamos a possibilidade de se fazer uma interrupção programada, já que esta é mais comum e pode ser tratada de forma mais simples. Para tal, o servidor a ser desligado executa um algoritmo similar ao de balanceamento de carga, descrito abaixo.

Algoritmo 2 Desligamento de Servidor Back-End

$B_{lista} \Leftarrow$ lista de blocos ordenados por P decrescente
para $i = 1$ até N_{blocos} **faça**
  $P_{bloco} \Leftarrow$ peso de $B_{lista}[i]$
  para todo servidor back-end diferente do atual **faça**
    $F_{pp} \Leftarrow \frac{(P_{total} + P_{bloco})}{P_{lim}}$ // fator de peso potencial do servidor
  fim para
  $F_{pp_{min}} \Leftarrow$ menor F_{pp} encontrado
  transfira $B_{lista}[i]$ para o servidor de fator de peso potencial $F_{pp_{min}}$
fim para

A diferença está no envio "obrigatório" de cada bloco para os servidores menos carregados, até que todos sejam enviados. Após isso, o servidor pode ser removido do cluster, sem que haja interrupção no serviço.

4.5 Considerações de Segurança

Durante o desenvolvimento dos servidores, é fundamental ter em mente que não há como garantir que o cliente não foi modificado ou que as mensagens recebidas respeitam o protocolo. Portanto, todas as decisões lógicas das regras do jogo devem ocorrer nos servidores, e cada comando enviado pelo jogador deve ser validado, evitando trapaças e violação de regras. O servidor deve estar preparado também para lidar com mensagens mal-formadas, que serão descartadas sem interromper seu funcionamento.

A autenticação de usuários deve ser feita não apenas pelo servidor de login, mas também pelo servidor de jogo, já que um cliente malicioso poderia tentar conectar-se a este diretamente, burlando o processo de autenticação do servidor de login. Uma solução é conectar todos os servidores de jogo front-end ao banco de dados e verificar novamente as credenciais para garantir que aquele personagem pertence mesmo ao jogador.

Caso isto não seja satisfatório, podemos utilizar um sistema de *tokens*. O cliente recebe do servidor de login uma mensagem criptografada por uma chave conhecida tanto pelos servidores de login quanto pelos servidores de jogo. Esta mensagem, que chamamos de token, contém informações sobre o jogador e o horário da requisição. O token é repassado pelo cliente ao servidor de jogo, que o valida e autoriza a conexão. Periodicamente, a chave deverá ser trocada para minimizar as chances de alguém descobrí-la.

Outro ponto importante é garantir a identidade e integridade das mensagens trocadas entre cliente e servidor. O processo de login requer o envio de credenciais pela rede, que poderiam ser interceptadas por um usuário mal intencionado. Este poderia, ainda, forjar mensagens enviadas ao servidor, passando-se

por outro jogador. Para que isto não ocorra, é importante criptografar as mensagens com uma chave exclusiva para cada jogador. Algumas sugestões são o uso de criptografia com chave pública/privada, ou a criação de uma única chave compartilhada entre cliente e servidor no início da sessão, que seria enviada por um canal também criptografado, como o SSL/TLS [IETF].

Por não ser o foco deste projeto, estas questões de segurança de autenticação e criptografia não são cobertas em nossa implementação. Porém, recomenda-se fortemente que elas sejam consideradas em sistemas de jogos comerciais.

5
Implementação do Sistema

5.1 DALua

Com o objetivo de facilitar a programação da comunicação entre servidores, adotamos a biblioteca DALua [Cos07], fruto de um projeto anterior nosso e aprimorada para este projeto, que fornece diversas facilidades para o desenvolvimento de aplicações distribuídas em Lua.

O DALua foi construído sobre a API do sistema ALua [Uru02], que permite troca de mensagens entre os membros de uma rede TCP de maneira assíncrona, com uso de callbacks. Isto significa que os métodos que fazem operações na rede retornam imediatamente, e uma callback é chamada posteriormente pelo ALua indicando se a operação foi bem sucedida. Assim, o processo pode executar outras tarefas enquanto as operações de rede são concluídas.

As mensagens do ALua são strings contendo código Lua, que é interpretado e executado ao ser recebido pelo processo destinatário. Desta forma, torna-se possível o *hot-swapping* de código via rede, ou seja, o envio de novos trechos de código (ou atualizações dos existentes) para processos que estão em execução em outras máquinas.

O DALua abstrai o modelo do ALua para tornar mais prático o desenvolvimento de aplicações distribuídas. Em vez de lidar diretamente com trechos de código Lua, o envio de mensagens é similar a chamadas de procedimentos remotos (RPC). Basta especificar o destinatário, o nome da função a ser chamada e os argumentos, que o DALua cuidará de serializar os argumentos (incluindo tabelas) e efetuar a chamada da função. É possível também especificar uma lista de processos como destinatários, e a mensagem será enviada por multicast. O exemplo abaixo envia uma chamada à função `print` do processo 2 da máquina 192.168.1.2, passando como argumento a string `"Hello World"`.

```
dalua.send("2@192.168.1.2", "print", "Hello World")
```

Observe que não há uma callback definida para a operação de envio, como seria o caso no ALua. Em vez disso, o DALua oferece um sistema de

eventos produtor-consumidor, onde os processos podem monitorar os eventos nos quais estão interessados e também disparar seus próprios eventos. Desta forma, vários processos podem ser notificados simultaneamente, o que o torna mais flexível do que uma simples callback.

No caso do método `dalua.send`, o processo se inscreveria *a priori* no evento `"dalua_send"` com a chamada `dalua.events.monitor("dalua_send", send_callback)`. Então, a função de nome `send_callback` seria chamada quando a operação fosse concluída, recebendo como argumento o resultado do `dalua.send`.

Listagem 1: hello.lua: Hello World usando DALua

```
-----------------------------------
---- DALua Hello World Sample ----
-----------------------------------

-- This sample is the simplest DALua application.
-- The process sends a 'print' message to itself.

require("dalua")

function init_handler()
  dalua.send(dalua.self(), "print", "hello world!")
  dalua.send(dalua.self(), "dalua.exit")
end

function send_handler(event, status, errmsg, destid)
  if status == "error" then
    print(string.format("[%s] Error while sending
       message to %s: %s", event, destid, errmsg))
  end
end

dalua.debug = true
dalua.events.monitor("dalua_init", init_handler)
dalua.events.monitor("dalua_send", send_handler)
dalua.init("127.0.0.1", 4321)
dalua.loop()
```

O evento `"dalua_send"` indica se a mensagem foi enviada ou não, mas não garante que naquele momento todos os processos destinatários já receberam e executaram a mensagem. Freqüentemente é necessário ter essa garantia antes

de se continuar com a execução do programa. Para isso, o DALua oferece o método `dalua.acksend`, cujo evento de conclusão só é disparado após todos os processos destinatários enviarem uma confirmação de que a mensagem foi recebida com sucesso. Isto é gerenciado internamente pelo DALua, de forma que o uso deste método é idêntico ao do `dalua.send`.

Uma desvantagem em relação ao uso direto de callbacks é que qualquer chamada ao `dalua.send`, por exemplo, gera o mesmo tipo de evento `"dalua_send"`, não sendo possível distinguí-las. Para contornar este problema, a chamada retorna uma string que a identifica unicamente e que é passada como argumento no evento gerado por ela.

O DALua oferece também alguns algoritmos importantes como exclusão mútua distribuída, ordenação causal e total de mensagens, e criação de grupos lógicos de processos com notificação da entrada e saída de membros do grupo. Outro recurso interessante é o temporizador, que permite agendar o envio de mensagens com o número de repetições e período de tempo desejados. Neste projeto, utilizamos o temporizador para criar testes automatizados que simulam as ações dos jogadores. Detalhes sobre o uso desses recursos estão disponíveis na documentação online.

É possível mesclar uma rede de processos DALua com conexões tradicionais de clientes externos através do uso de canais TCP do ALua, que oferecem funcionalidade similar aos sockets TCP da biblioteca LuaSocket [Neh04]. Assim, os servidores deste projeto são capazes de receber conexões de clientes que não utilizam o DALua.

Algumas melhorias foram implementadas no DALua conforme surgiam necessidades específicas para este projeto, e serão disponibilizadas numa próxima versão pública da biblioteca.

5.2 Clientes

Desenvolvemos um cliente em C++ com as funcionalidades mínimas para testar o jogo, com uso do OpenGL para a visualização em 2D. Jogadores são representados por círculos laranjas e monstros por círculos vermelhos. O terreno é verde nas áreas em que é permitido andar e azul nas áreas consideradas obstáculos. É possível mover-se com as setas do teclado e atacar uma posição vizinha segurando-se a tecla “Control” juntamente com a seta de direção. Pode-se também enviar mensagens para os demais jogadores.

Outro tipo de cliente foi preparado em Lua, com o único objetivo de gerar sobrecarga no servidor para os testes de desempenho. Este cliente é disparado centenas de vezes e envia diversos comandos ao servidor através do uso de temporizadores do DALua.

5.3 Servidores de Login

Os servidores de login baseiam-se diretamente no LuaSocket, pois não há necessidade dos recursos de aplicação distribuída do DALua. Pode-se criar um número ilimitado de processos, onde cada um escutará num endereço IP e porta específicos. O cliente utiliza um servidor inicial aleatório para tentar conectar-se e, em caso de falha, tenta outros servidores por *round-robin*. Após a autenticação, o servidor de login indica ao cliente o servidor FE que irá atendê-lo. A distribuição de clientes nos servidores FE também é feita por round-robin, o que é satisfatório, mas também é possível utilizar técnicas mais avançadas de balanceamento de carga.

5.4 Servidores Front-End

Cada servidor FE tem um processo *Manager* que dispara um ou mais *Listeners* (Fig. 5.1). Cada Listener é um processo independente capaz de aceitar um certo número de conexões TCP, cujas mensagens são repassadas ao Manager. Esta separação é feita de maneira a permitir paralelismo entre os Listeners, sem a necessidade de um sistema de *multithreading* para Lua, e também pelo limite de conexões simultâneas aceitas pela função `select`, da qual o LuaSocket depende.

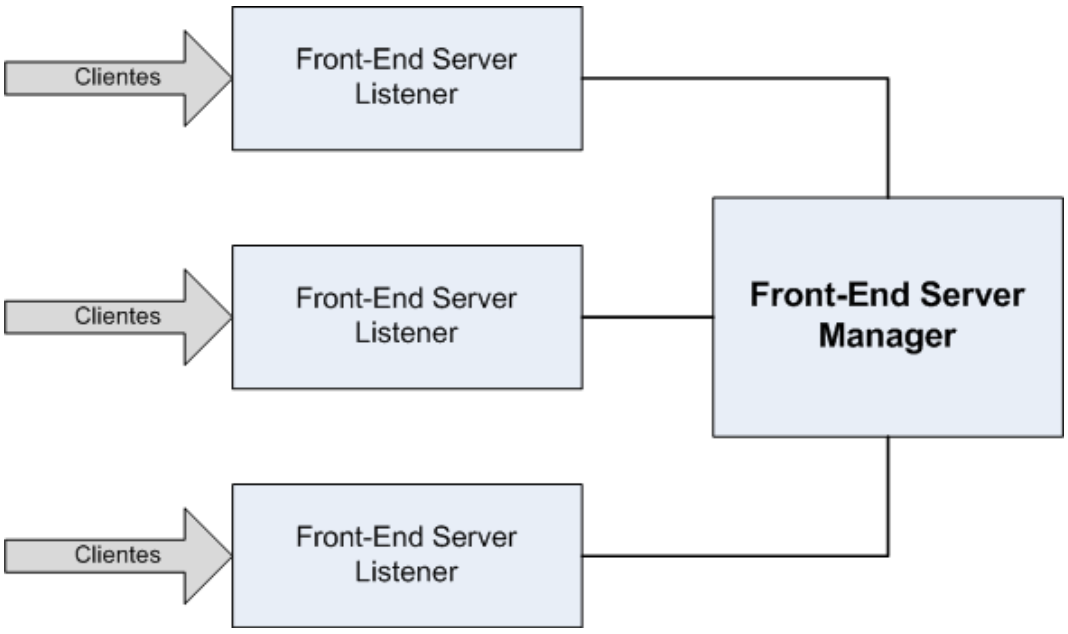

Figura 5.1: Arquitetura lógica de um servidor front-end.

Os Managers do cluster fazem parte de uma aplicação distribuída do DALua, podendo comunicar-se entre si. Eles sabem quais clientes estão sob

responsabilidade de cada Manager, o que permite que alguns recursos, como a troca de mensagens de bate-papo entre os jogadores ou a listagem dos personagens que estão online, ocorram diretamente entre os servidores FE envolvidos.

Os servidores FE também estão conectados aos servidores BE, com os quais é feita a troca de mensagens relativas às mudanças de estado do jogo. Cada ação efetuada por um jogador é notificada ao servidor BE responsável por aquele bloco do mapa. Analogamente, notificações de eventos ocorridos no BE são recebidos pelos Managers, que repassa a informação aos clientes que foram afetados, utilizando o protocolo cliente-servidor.

5.5 Servidores Back-End

Cada processo do BE é responsável pelas regiões do mapa que lhe foram atribuídas. Sua principal tarefa é processar eventos oriundos das ações dos jogadores ou da inteligência artificial dos NPCs, e então propagar as atualizações de estado para o FE quando necessário.

As ações dos jogadores precisam ser validadas antes de se tornarem eventos. Isto é necessário para evitar trapaças de jogadores que alteraram o programa cliente com o intuito de burlar as regras do jogo como, por exemplo, mover-se sobre obstáculos do mapa. Uma discussão mais detalhada sobre este tipo de problema encontra-se no capítulo 4.5.

Por outro lado, a inteligência artificial é completamente executada pelo BE e, portanto, não precisa desse tipo de validação. Neste projeto, o sistema de inteligência artificial resume-se a encontrar um inimigo dentro do campo de visão do NPC e então calcular o menor caminho até ele através do algoritmo A* [Rus03]. O resultado é uma seqüência de comandos similares aos de um jogador comum, que é transformada numa série de eventos a serem processados.

Um evento pode ocasionar uma mudança de estado que afeta direta ou indiretamente vários jogadores. Por exemplo, ao mover-se, a posição do jogador deve ser atualizada na tela de todos os demais jogadores que o contêm em seu campo de visão. Para saber quais jogadores foram afetados por um evento, é necessário buscá-los no bloco do mapa onde ele ocorreu e também nos vizinhos, pois jogadores na fronteira de blocos vizinhos podem estar próximos o suficiente do evento para que este encontre-se dentro de seu campo de visão.

Assim, dado que o campo de visão de um jogador é definido de modo que seja sempre menor que a área de um bloco do mapa, um evento pode afetar até quatro[1] blocos, que por sua vez podem ser de responsabilidade de até quatro

[1]Isto deve-se à opção pelo formato quadrado dos blocos. Um formato hexagonal reduziria o número máximo de vizinhos para três, porém sua implementação é mais complexa.

servidores distintos.

Sempre que um evento é capaz de alterar o estado do jogo, como a movimentação de personagens ou um ataque, é necessário que todos os servidores BE estejam de acordo sobre os efeitos deste evento, caso contrário teríamos um estado inconsistente. Isto poderia ocorrer, por exemplo, se dois jogadores na fronteira de blocos controlados por servidores diferentes tentassem mover-se para a mesma posição simultaneamente. Como apenas um jogador pode ocupar uma determinada posição num dado instante, é necessário decidir qual dos dois eventos terá prioridade e será aceito.

Problemas como esse são formas de *condições de corrida* entre servidores. Para resolvê-los, utilizamos a exclusão mútua distribuída, oferecida pelo DALua, entre os servidores BE envolvidos no evento. Desta forma, apenas um evento é processado por vez, de maneira que, no exemplo dado anteriormente, algum dos jogadores teria seu pedido de movimentação recusado.

Após o processamento dos eventos, informações que indicam alterações de estado do mapa e dos personagens são enviadas aos servidores FE, para que sejam repassadas aos clientes e representadas graficamente ao jogador.

5.6 Massive Online Test

Para testar o desempenho de nossa arquitetura, desenvolvemos uma ferramenta, que chamamos de *Massive Online Test* (MOT), para avaliar o desempenho de servidores em condições reais ou extremas de funcionamento. Ela foi implementada em Lua, e os componentes de interface gráfica foram feitos com IUPLua [Scu].

O MOT funciona como uma aplicação distribuída (ver Fig. 5.2). Ele controla um ou mais servidores de teste, que serão usados como plataforma para a execução dos testes nos servidores alvo. É possível disparar dois tipos de processo: clientes de teste, que simulam as atividades de clientes reais, e espiões, que servem apenas para coletar estatísticas.

A aplicação distribuída é criada pelo DALua. Inicialmente, o usuário dispara o processo mestre e configura o teste que será executado. O gerenciamento pode ser feito por diferentes aplicativos que se conectam ao mestre e se identificam como *controladores*. Por exemplo, pode-se utilizar interface gráfica, linha de comando, ou ainda um arquivo de configuração pré-determinado. É possível escolher o número de instâncias de clientes que serão disparados, o número de espiões e a distribuição destes processos entre os servidores de teste.

Os espiões podem ser remotos ou locais. Espiões remotos são executados a partir dos servidores de teste, em paralelo com os clientes. Eles são capazes de obter estatísticas de latência de conexão e resposta dos servidores alvo. Já

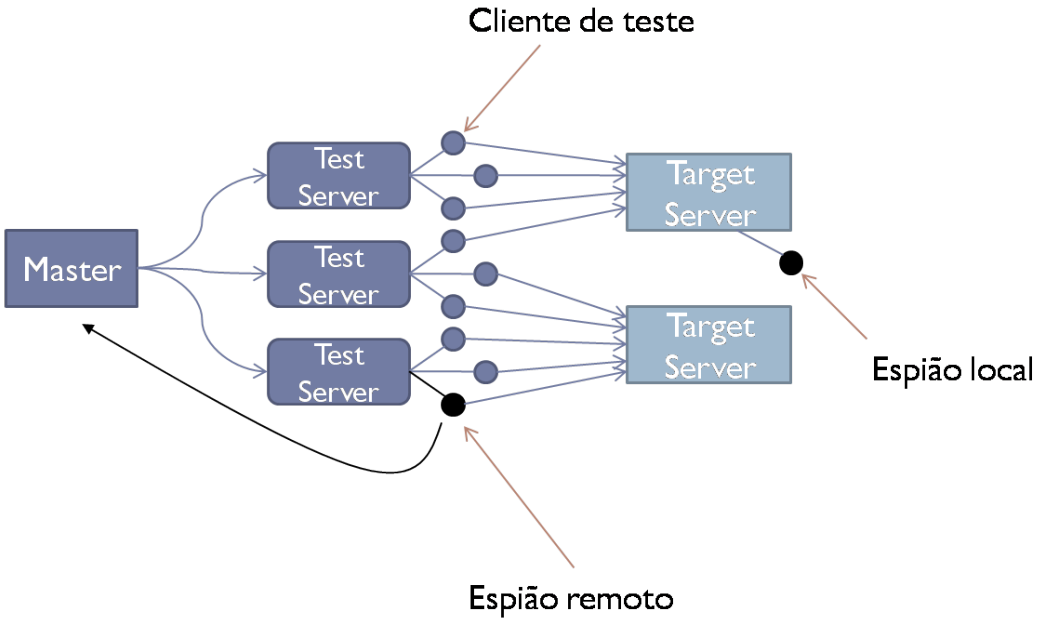

Figura 5.2: Funcionamento do Massive Online Test.

os espiões locais residem diretamente nos servidores alvo. Assim, é possível capturar também estatísticas de uso de memória, uso de CPU e tráfego de rede (bytes enviados/recebidos por segundo). Estes dados são obtidos com o auxílio de uma biblioteca escrita em C e importada pelo código Lua.

Todas as informações coletadas pelos espiões são enviadas de volta ao mestre, que adota um modelo produtor-consumidor para o tratamento dos dados. Diferentes aplicativos podem se inscrever para o recebimento destas estatísticas, e tratá-las de maneiras diferentes. Oferecemos dois aplicativos, um com interface gráfica que gera gráficos para as estatísticas (Fig. 5.3), e outro que salva os dados em arquivo, possibilitando comparações *offline* de testes executados em momentos distintos.

Assim como fizemos com o DALua, pretendemos disponibilizar esta ferramenta para o público. Ela mostrou-se muito útil nos testes de desempenho e funcionamento do nosso projeto, e acreditamos que poderá ser útil para outros desenvolvedores.

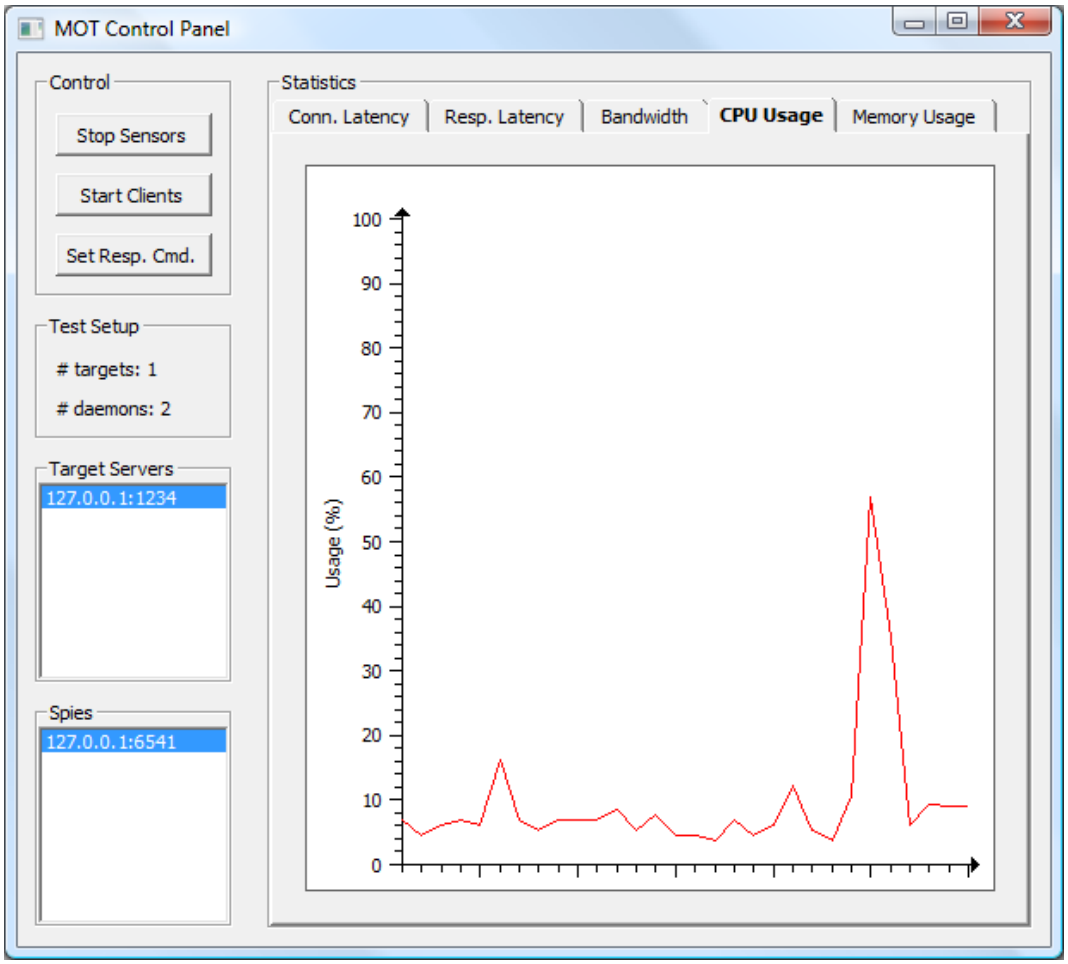

Figura 5.3: Interface de controle e estatísticas.

6
Testes e Resultados

Para os testes de nossa arquitetura, criamos um cluster com a seguinte configuração:

- 1 servidor de login,
- 3 servidores de jogo front-end,
- 4 servidores de jogo back-end.

O mapa do jogo, de dimensão 512x512, foi dividido em 16 blocos iguais de 128x128 cada. Inicialmente, cada servidor BE fica responsável por 4 blocos, que serão redistribuídos ao longo da execução do jogo pelo algoritmo de balanceamento de carga.

Para testamos o correto funcionamento do sistema, utilizamos um aplicativo cliente simples, descrito no capítulo 5.2. Com o objetivo de facilitar a visualização do balanceamento, criamos uma ferramenta gráfica auxiliar que exibe a divisão do mapa em blocos, o servidor responsável pelo bloco e o número de elementos (jogadores e NPCs) no bloco. A configuração inicial do jogo é representada por esta ferramenta conforme a Fig. 6.1.

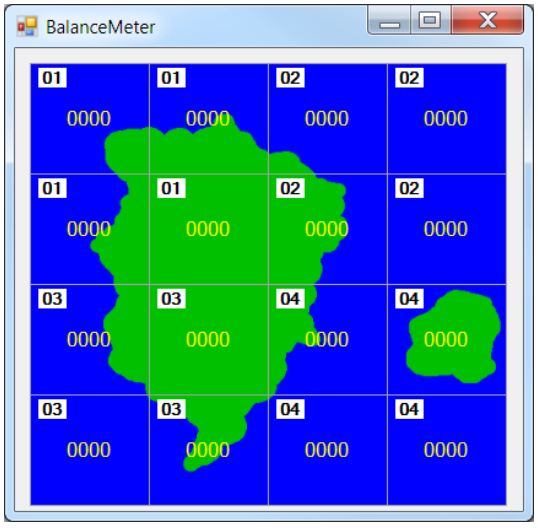

Figura 6.1: Configuração inicial. O servidor responsável pelo bloco é indicado no canto superior esquerdo, enquanto que o número de elementos no bloco é indicado no centro.

Para os testes de desempenho e escalabilidade, utilizamos 3 máquinas de teste extras para dispararem os clientes de teste, que são versões especiais do cliente com o único objetivo de trocar uma grande quantidade de mensagens com o servidor e mantê-lo ocupado. Estas mensagens resultam em eventos especiais de teste, que fazem o servidor back-end responsável por esse cliente executar uma rotina de alto custo computacional. Assim, podemos simular uma sobrecarga nos servidores, que seria equivalente ao custo de uma lógica de jogo mais complexa encontrada num cenário real.

Cada uma das máquinas de teste hospedou 1000 clientes, totalizando 3000. Como um processo que usa LuaSocket é, por padrão, limitado a 64 conexões simultâneas, foram necessários 48 processos Listeners no cluster FE, o que significa que cada um dos três Managers disparou 16 Listeners.

Os clientes foram distribuídos aleatoriamente pelo mapa no momento de sua inserção no jogo. Como eles são específicos para os testes de escalabilidade e são capazes de simular esforço computacional no servidor, achamos desnecessário utilizar NPCs nestes testes.

Efetuamos quatro testes ao todo: distribuição regular de clientes pelo mapa com e sem balanceamento de carga, e concentração de 50% dos clientes num único bloco, com e sem balanceamento.

Utilizamos, então, o MOT para medir a latência de resposta e o uso de CPU pelos servidores de jogo. A média aritmética dos valores registrados durante 60 segundos em intervalos de 1 segundo está nas tabelas abaixo.

Tabela 6.1: Distribuição regular sem balanceamento

Servidor	No. clientes	Latência (ms)	Uso CPU (%)
1	1200	1.5	76
2	350	0.6	19
3	1000	1.3	63
4	450	0.6	24

Tabela 6.2: Distribuição regular com balanceamento

Servidor	No. clientes	Latência (ms)	Uso CPU (%)
1	800	0.9	48
2	600	0.7	35
3	1000	1.3	66
4	600	0.7	32

Como podemos ver, o balanceamento aliviou a sobrecarga do servidor 1, transferindo-a para os servidores 2 e 4 (ver Fig. 6.2).

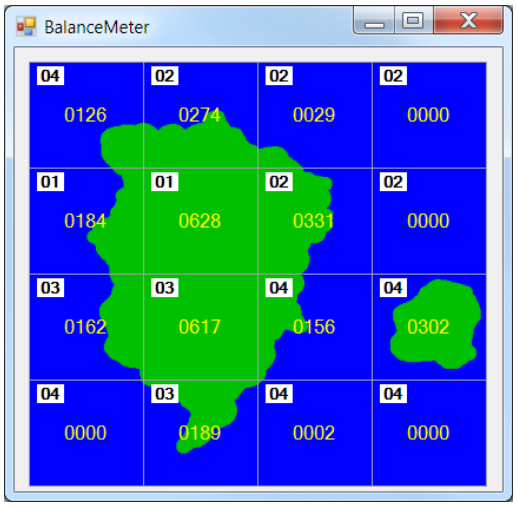

Figura 6.2: Configuração após balanceamento.

No teste a seguir, verificamos o que ocorre quando um grande número de clientes concentra-se num único bloco inicialmente alocado para o servidor 4.

Tabela 6.3: Distribuição concentrada sem balanceamento

Servidor	No. clientes	Latência (ms)	Uso CPU (%)
1	650	1.3	37
2	200	1.5	10
3	550	2.4	31
4	1600	5.6	98

Tabela 6.4: Distribuição concentrada com balanceamento

Servidor	No. clientes	Latência (ms)	Uso CPU (%)
1	650	1.1	36
2	300	1.4	17
3	550	2.2	31
4	1500	4.3	93

Aqui podemos ver que o servidor 4 passou todos os seus blocos para o servidor 2, com exceção daquele onde os clientes estão concentrados (ver Fig. 6.3). Para que o balanceamento fosse mais efetivo, seria necessário escolher blocos de tamanhos menores para que esta região pudesse ser quebrada e repartida entre os demais servidores.

Com base nestes testes, concluímos que o balanceamento de carga do back-end é uma ferramenta essencial para tirarmos um bom proveito do cluster. A nossa arquitetura pode ser dimensionada para um número virtualmente ilimitado de clientes, desde que o cluster seja escalado de acordo. É importante também definir o tamanho dos blocos do mapa de maneira que o

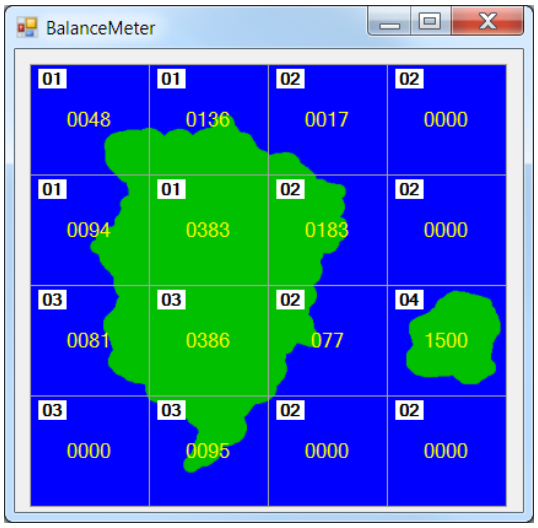

Figura 6.3: Configuração após balanceamento.

reparticionamento tenha granularidade suficiente para distribuir bem a carga entre os servidores, tendo sempre o cuidado de manter um tamanho maior que o campo de visão do jogador, como explicado anteriormente.

7
Considerações Finais

A arquitetura que propusemos para jogos MMORPG é viável e pode ser utilizada inclusive em sistemas de jogos comerciais, caso sejam implementadas as considerações de segurança descritas no capítulo 4.5. A escalabilidade, tópico que focamos neste projeto, é suficiente para atender a demanda de um jogo desse tipo com milhares de clientes, dependendo apenas da disponibilidade de hardware.

Uma desvantagem de nossa arquitetura é a relativa complexidade de implementação dos servidores de jogo e, principalmente, do cluster back-end. Caso o número de jogadores esperado não passe de algumas centenas, recomendamos o uso de uma arquitetura mais simples, como a de cliente-servidor sem uso de cluster, mesmo que isso exija um hardware mais potente.

A linguagem Lua mostrou-se de grande utilidade para a programação de sistemas distribuídos com os requisitos de jogos MMO. Sua flexibilidade e sua habilidade de "hot swapping" permitem a rápida implementação das funcionalidades necessárias à lógica do jogo, ao mesmo tempo em que seu desempenho é satisfatório e não se torna um gargalo para a escalabilidade.

A biblioteca DALua facilitou bastante a implementação dos algoritmos para os clusters de servidores, devido aos vários recursos que são oferecidos para esse tipo de aplicação. Este projeto, por sua vez, serviu também como um excelente teste para a robustez do DALua, o que acabou resultando em algumas melhorias para a biblioteca.

A ferramenta de testes de desempenho que desenvolvemos para este projeto, *Massive Online Test*, foi fundamental para medição dos resultados obtidos com nossa arquitetura. É uma ferramenta simples e funcional, capaz de capturar os indicadores mais importantes de desempenho dos servidores. Pretendemos disponibilizar esta ferramenta ao público, pois acreditamos que será útil para outros desenvolvedores.

Referências

[Bliz] ENTERTAINMENT, B.. **World of Warcraft**. `http://www.worldofwarcraft.com`. 1

[Bog03] BOGOJEVIC, S.; KAZEMZADEH, M.. **The architecture of massive multiplayer online games**. Technical report, Department of Computer Science, Lund Institute of Technology, Lund University, 2003.

[Calt02] CALTAGIRONE, S.; KEYS, M.; SCHLIEF, B. ; WILLSHIRE, M. J.. **Architecture for a massively multiplayer online role playing game engine**. Journal of Computing Sciences in Colleges, 18(2), December 2002.

[Cos07] COSTA, R.. **DALua**. `http://alua.inf.puc-rio.br/dalua`. 5.1

[IETF] IETF. **Transport Layer Security**. `http://www.ietf.org/html.charters/tls-charter.html`. 4.5

[Knut04] KNUTSSON, B.; LU, H.; XU, W. ; HOPKINS, B.. **Peer-to-peer support for massively multiplayer games**. Technical report, Department of Computer and Information Science, University of Pennsylvania, 2004. 3.3.1

[Mor96] L. MORSE, K.. **Interest management in large-scale distributed simulations**. Technical Report ICS-TR-96-27, University of California, 1996. 3.2.3

[Neh04] NEHAB, D.. **Luasocket: Networking support for lua**. Technical report, PUC-Rio, 2004. `http://www.tecgraf.puc-rio.br/~diego/professional/luasocket/`. 5.1

[Rus03] RUSSELL, S. J.; NORVIG, P.. **Artificial Intelligence: A Modern Approach.** Prentice Hall, 2003. 5.5

[Scu] SCURI, A. E.. **Portable User Interface**. `http://www.tecgraf.puc-rio.br/iup`. 5.6

[Theo04] ANDROUTSELLIS-THEOTOKIS; STEPHANOS; SPINELLIS ; DIOMIDIS. **A survey of peer-to-peer content distribution technologies**. ACM Computing Surveys, 36(4), December 2004. 3.3

[Uru02] URURAHY, C.; RODRIGUEZ, N. ; IERUSALIMSCHY, R.. **ALua: Flexibility for parallel programming**. Technical report, PUC-Rio, 2002. 5.1

[Wik] WIKIPEDIA. **Neverwinter Nights (AOL Game)**. `http://en.wikipedia.org/wiki/Neverwinter_Nights_%28AOL_game%29`. 1

[Zha01] ZHAO, B.; KUBIATOWICZ, J. ; JOSEPH, A.. **Tapestry: An infrastructure for fault-tolerant**. Technical report, Computer Science Division, University of California, Berkeley, 2001. 3.3

Apêndice I - Siglas

BE Back-End, cluster de servidores que atua na rede interna, sem contato direto com os clientes.

FE Front-End, cluster de servidores que é exposto à rede externa para receber conexões dos clientes.

FPS First-Person Shooter, tipo de jogo no qual a câmera posiciona-se nos olhos do personagem e com o objetivo de atirar nos inimigos.

MMOG Massively Multiplayer Online Game, tipo de jogo onde muitos jogadores interagem num mundo virtual através da Internet.

MMORPG Tipo de MMOG baseado no estilo de jogo RPG.

MOT Massive Online Test, programa que desenvolvemos para testar o desempenho de clusters de servidores.

NPC Non-Player Character, personagem no jogo que é controlado pelo computador através de inteligência artificial.

P2P Peer-to-Peer, arquitetura de rede em que não há servidor; os clientes trocam mensagens entre si diretamente.

RPG Role-Playing Game, estilo de jogo onde o jogador desempenha o papel de um personagem dentro do jogo.

RTS Real-Time Strategy, tipo de jogo com foco na estratégia que ocorre em tempo real, em contraste aos jogos de estratégia baseados em turnos (Turn-Based Strategy).

WoW World of Warcraft, MMORPG mais famoso e com maior número de usuários da atualidade.