Etelvina Nunes Pinho

Tolerância a Falhas em Jogos On-line

Universidade da Beira Interior

Departamento de Informática

Agosto 2009

Etelvina Nunes Pinho

Tolerância a Falhas em Jogos On-line

Tese submetida ao Departamento de Informática para o preenchimento dos requisitos para a concessão do grau de Mestre efectuada sob a supervisão da Doutora Paula Prata, Professora Auxiliar do Departamento de Informática da Universidade da Beira Interior, Covilhã, Portugal

Universidade da Beira Interior
Departamento de Informática
Agosto 2009

Agradecimentos

A finalização desta tese simboliza o terminar de uma etapa e o iniciar de outra. Como não poderia deixar de ser, agradeço a todas as pessoas que contribuíram para a elaboração desta tese, para o meu enriquecimento pessoal e intelectual.

À Professora Doutora Paula Prata por toda a ajuda e disponibilidade durante o desenvolvimento desta tese.

Aos meus pais e irmãs que sempre me apoiaram e incentivaram.

Como não poderia deixar de ser ao Dioguito que esteve sempre presente com as brincadeiras e birras.

Ao Miguel por todo o carinho, compreensão e ajuda. Pelo incentivo e por ter estado presente em todos os momentos.

A todos os meus colegas pela entreajuda e momentos de descontração.

A todas as pessoas anónimas que ao longo destes anos se cruzaram no meu caminho e de alguma forma contribuíram para a elaboração desta tese.

Resumo

A indústria dos jogos *on-line* em particular os MMOG movimenta milhões e milhões de dólares tendo cada vez mais adeptos. Os jogadores exigem jogos cada vez mais apelativos em termos gráficos e de conteúdo, com acessos cada vez mais rápidos e com uma taxa de disponibilidade cada vez mais elevada.

Apesar de um jogo não ser uma aplicação crítica no sentido tradicional do termo, a tolerância a falhas surge como um aspecto cada vez mais importante para assegurar uma alta taxa de disponibilidade e assegurar a continuidade de um jogo na presença de falhas. Algumas horas de inactividade de um jogo originam grandes prejuízos, tanto por perda directa de jogadores nesse intervalo de tempo como por perda da credibilidade do jogo.

Nesta tese construímos uma versão tolerante a falhas do servidor de base de dados e do servidor *web* do conhecido jogo Dungeons & Dragons (D&D). Foi implementada a replicação de dados segundo o modelo "*master-master* modo activo-passivo" e a replicação de estados seguindo o modelo "*all-to-all*" utilizando um servidor de *load balancing* e de DNS. Com estas duas formas de replicação consegue-se garantir a continuidade do jogo na presença de falhas de forma transparente para o utilizador.

O impacto dos mecanismos implementados no desempenho do jogo foi avaliado através da realização de testes de carga com a ferramenta JMeter, simulando múltiplos jogadores em simultâneo e obtendo os valores do *throughput* para as situações com e sem replicação, com e sem falhas. Os resultados para simulações até 300 jogadores mostraram que o custo da replicação de estados é reduzido não ultrapassando os 5% e o custo da replicação de dados feita ao nível do jogo pode atingir cerca de 30% impondo-se a procura de novas soluções. Os resultados mostram ainda que, com o aumento do número de utilizadores, a diferença entre o *throughput* sem replicação e o *throughput* com replicação se vai atenuando o que significa que o custo da replicação se vai repartindo pelos utilizadores.

Palavras chave: Jogos on-line, tolerância a falhas, replicação de dados, replicação de estados.

Abstract

The on-line video game industry, and MMOG in a special manner, bears million dollar transactions and a growing number of supporters. Players demand for more appealing games, both graphically and in contents, with even faster accesses and higher availability rates.

Despite games not being critical applications in a traditional way, fault tolerance presents itself as an aspect of growing importance in the path to deliver higher availability rates and assure game continuity in the presence of faults. A few hours or even days of game downtime can represent significant losses, either by subsequent player resignation and a strong credibility decrement.

In this thesis, a fault tolerant version of both web and database servers for the well-known game "Dungeons & Dragons" is built. Data replication is achieved through a "*master--master*" model in an active-passive mode, and state replication follows the "*all-to-all*" model using a *load balancing* and DNS server. With these two replication methods, we can assure game quality and continuity under the presence of faults, in a transparent way for the user.

The implemented procedures impact on game performance was accessed through load tests, carried on using the JMeter tool, simulating multiple simultaneous players and quantifying *throughput* values for situations with and without replication, with and without faults. Simulations with less than 300 players show a minimal state replication cost, lower than 5%, and data replication cost goes up to 30% urging the need for new solutions. Results also exhibit an attenuation in the difference between the *throughput* without replication and the *throughput* with replication, which becomes slender as the number of connected players raise, meaning that replication cost is apportioned through users.

Keywords: On-line games, fault tolerance, data replication, state replication.

Conteúdo

Lista de Figuras

Lista de Tabelas

Lista de Listings

Siglas e Acrónimos

IP	Internet Protocol
BD	Base de Dados
D&D	Dungeons & Dragons
P2P	Peer-to-Peer
MOG	Multiplayer Online Games
RTM	Real-Time Multiplayer
VSM	Voronoi State Management
MMG	Massive Multiplayer Games
RPG	Role-Playing Games
RTS	Real-Time Strategy
FPS	First Person Shooter
RTF	Real Time Framework
NPC	Non-Player Character
ISP	Internet Service Provider
DNS	Domain Name System
NTP	Network Time Protocol
TCP	Transmission Control Protocol

TSR Tactical Studies Rules

NAT Network Address Translator

AJP Apache Java Protocol

GCS Group Communication System

HTTP Hypertext Transfer Protocol

MMOG Massive Multiplayer On-line Games

RSI-PC Replicated Snapshot Isolation with Primary Copy

FT-D&D Fault Tolerant Dungeons & Dragons

ROWAA Read-One-Write-All-Available

MMORPG Massive Multiplayer On-line Role-Playing Game

Capítulo 1

Introdução

Os jogos *on-line* em particular os Massive Multiplayer On-line Games (MMOG) permitem que milhares de jogadores se liguem simultaneamente ao mesmo mundo virtual. Este tipo de jogos, nos últimos anos, tem tido um grande sucesso comercial resultante de factores tais como, a qualidade da interface gráfica apresentada, a possibilidade de interacção com outros jogadores e ao facto de permitirem que cada utilizador viva através da sua personagem um mundo de magia. Estes factores fazem com que cada vez mais utilizadores demonstrem interesse em conhecer e participar activamente no mundo dos jogos *on-line*. Com o aumento exponencial do número de utilizadores, surge a necessidade de cada jogo oferecer uma disponibilidade elevada e histórias cada vez mais empolgantes, cativando assim milhões de utilizadores. O jogo "World of Warcraft" foi um dos impulsionadores deste crescimento ao garantir, sozinho no ano de 2006, receitas de 471 milhões de dólares [48]. Em 2008, as subscrições deste jogo atingiram os 16 milhões [47], sendo ainda um dos jogos que possuiu mais subscrições activas em 2009.

A maioria dos jogos *on-line* segue uma arquitectura cliente-servidor, onde os servidores são responsáveis por guardar e gerir todos os dados relacionados com o estado do jogo. Os clientes, por sua vez, são responsáveis por mostrar graficamente esses dados. Os jogos *on-line* podem ser baseados no cliente ou no *browser*. Nos baseados no cliente, como é o caso do "World of Warcraft " é necessário instalar o *software* do jogo no computador do utilizador. Os jogos baseados no *browser* têm a vantagem de apenas ser necessário um computador com acesso à *internet* e um *browser* para aceder ao jogo. Têm ainda a vantagem de a versão mais recente do jogo estar sempre disponível. A qualidade gráfica é uma desvantagem do modelo baseado no *browser* quando comparado com o modelo baseado em cliente.

Apesar de os jogos não serem uma aplicação crítica no sentido clássico do termo, a tolerância a falhas é uma questão crucial, porque uma falha nos servidores dos jogos *on-line*, implica um efeito negativo na "experiência do jogo". Quem joga estes jogos espera que os servidores estejam permanentemente disponíveis e que os períodos em que isso não aconteça sejam reduzidos ou inexistentes. Estima-se que a falha que ocorreu no World of Warcraft em 2006 tenha custado à empresa Blizzard Entertainment cerca de $26198 por hora [3]. Hoje em dia, a escalabilidade nos MMOGs é conseguida através de um conjunto de servidores dedicados, ligados a uma rede de alta velocidade formando um *cluster* de servidores [16]. Esta arquitectura permite, aumentar a escalabilidade e a disponibilidade de um jogo através de técnicas de balanceamento de carga (*load balancing*). De notar que a distribuição dos componentes do jogo por vários servidores não corresponde necessariamente à replicação dos seus componentes. Assim, se ocorrer uma falha num dos servidores do jogo os utilizadores ligados a esse servidor serão afectados, caso não sejam redireccionados para uma máquina alternativa. Sobre os jogos existentes encontramos pouca informação disponível sobre a existência ou não de replicação.

Nesta tese exploramos técnicas de replicação que permitam tornar transparente a falha dos servidores dos jogos. Estudamos o impacto da replicação de dados e de estados em termos de *throughput* do jogo e da sua variação com o crescimento do número de utilizadores, simulando vários jogadores em simultâneo. Para este estudo, usamos uma versão simplificada do jogo D&D, desenvolvida num anterior projecto de fim de curso [2], a que adicionamos replicação de dados e de estados, a que chamaremos FT-D&D. O jogo *on-line* D&D originalmente conhecido como "The Fantasy Game" foi o primeiro role-playing game (RPG). Criado por Gary Gygax e Dave Arneson em 1972, foi lançado em 1974 pela Tactical Studies Rules [12] tendo levado à criação de muitos jogos do género. O jogo desenvolvido no projecto referido anteriormente foi baseado no jogo *on-line* "Dungeons & Dragons Basic Game 2006 Edition". É um jogo muito simples em termos de interface gráfica, que não tendo a atractividade dos jogos comerciais possui as características base de um jogo *on-line* multi-utilizador.

Através de testes de carga em que simulamos múltiplos jogos em simultâneo, observamos que a replicação da base de dados tem um impacto muito mais significativo do que a replicação de estados no desempenho do jogo. Com o aumento do número de jogadores concluímos que o impacto da replicação (dados e estados) é cada vez menor, o que significa que o principal factor para a diminuição do desempenho é o aumento do número de jogadores. Finalmente, simulamos a ocorrência de falhas quer no servidor *web*, quer no

servidor de bases de dados estudando o custo da recuperação do jogo, isto é, a diminuição do *throughput* quando ocorre uma falha. Observamos, em todas as simulações realizadas, que uma falha do servidor *web*, é transparente para o utilizador. Uma falha no servidor de base de dados implica uma pausa perceptível ao utilizador, sendo necessário procurar soluções mais eficientes.

1.1 Motivação e Objectivos

A motivação desta tese deve-se ao facto dos MMOG serem aplicações cada vez mais exigentes, quer do ponto de vista das tecnologias envolvidas, como das interfaces gráficas e bases de dados, quer do ponto de vista dos recursos necessários para garantir um elevado desempenho, escalabilidade, disponibilidade, segurança e tolerância a falhas [48].

Os MMOG representam um mercado muito lucrativo no sector dos jogos [42] o que leva ao constante aparecimento de novas empresas e como consequência novos jogos. Os seus utilizadores esperam que os jogos tenham 100% de disponibilidade o que leva à necessidade da implementação de técnicas de tolerância a falhas.

Os objectivos desta tese foram os seguintes:

1. Estudar as principais características dos jogos *on-line*, estudar os mecanismos de replicação de dados e de estados e quais os estudos existentes sobre a aplicação desses mecanismos a jogos *on-line*.

2. Completar a versão existente do jogo D&D e construir a versão com replicação de dados e de estados, isto é, o FT-D&D que permita a continuidade do jogo em caso de falha do servidor *web* ou do servidor de base de dados.

3. Simular o acesso de múltiplos jogadores em simultâneo de forma a medir o impacto da replicação no desempenho do jogo.

4. Detectar quais os pontos críticos da versão implementada e propor soluções.

5. Propor uma arquitectura generalizável a outros jogos.

1.2 Contribuições

Este trabalho para além do estudo das características dos jogos *on-line*, do estudo dos mecanismos existentes para replicação de dados e de estados e da sua aplicação aos jogos *on-line* tem três principais contribuições:

1. Foi implementada uma versão tolerante a falhas do servidor *web* e do servidor de base de dados do jogo D&D. Partindo de um jogo implementado com *software* de domínio público foram adicionados mecanismos de replicação mais uma vez com *software* não proprietário e avaliado o seu impacto no desempenho do jogo.

2. Foi feita a avaliação experimental do impacto da replicação de dados e estados num jogo *on-line* assim como o impacto da ocorrência de falhas e correspondente recuperação do jogo feita de forma transparente para o utilizador. Para isso foram simulados até algumas centenas de jogadores em simultâneo através de testes de carga feitos com a ferramenta JMeter. Do estudo das várias alternativas de replicação e do comportamento do jogo surgiu a proposta de uma nova arquitectura.

3. Elaboração de uma nova arquitectura para replicação de dados e estados. A arquitectura que propomos resultou da identificação dos pontos críticos na solução implementada. A solução proposta resolve os problemas identificados faltando fazer a sua avaliação em termos de desempenho e contrapor, o ganho que previsivelmente se obterá, com o custo dos recursos adicionais necessários para a implementar.

1.3 Organização da Tese

Esta tese tem a seguinte estrutura: no capítulo 2 após a introdução de alguns conceitos base da tolerância a falhas, são abordados os diferentes géneros de jogos *on-line* dentro dos MMOGs e as suas arquitecturas. É apresentado o jogo *on-line* D&D que usamos como ponto de partida para este trabalho. De seguida, fazemos uma revisão da literatura sobre os dois mecanismos de tolerância a falhas explorados neste trabalho, replicação de dados e de estados e a sua aplicação aos jogos *on-line*. Referimos ainda as actuais técnicas de *load balancing* utilizadas nos jogos *on-line* e necessárias para o redireccionamento dos acessos aos jogos. A descrição da versão tolerante a falhas do D&D (FT-D&D) e a implementação de replicação de dados e de estados está inserida no capítulo 3. No capítulo 4, descrevemos

o contexto experimental utilizado para testar os mecanismos implementados e medir o *throughput* do jogo para um crescente número de utilizadores através da simulação de múltiplos jogos. Apresentamos ainda neste capítulo os resultados obtidos. Na sequência deste trabalho propomos no capítulo 5 uma arquitectura tolerante a falhas que pode ser adaptada a qualquer sistema. Finalmente, no capítulo 6 apresentamos as conclusões desta tese.

Capítulo 2

Estado da Arte

Neste capítulo são introduzidos os conceitos base da tolerância a falhas e são apresentados os diferentes tipos de jogos *on-line*. De seguida faz-se uma revisão da literatura sobre os dois mecanismos de tolerância a falhas explorados neste trabalho, replicação de dados e de processos ou estados e a sua aplicação aos jogos *on-line*.

Na secção 2.1 são apresentados os conceitos gerais que um sistema tolerante a falhas deve possuir. A categorização dos diferentes jogos *on-line* é realizada na secção 2.2, sendo a secção 2.3 dedicada à arquitectura destes jogos. A história do jogo *on-line* D&D é introduzida na secção 2.4, por ser este o jogo base do FT-D&D. A versão tolerante a falhas do jogo D&D foi desenvolvida como plataforma de trabalho para estudarmos a replicação de dados e de processos em jogos *on-line*. Na secção 2.5 são apresentados alguns dos métodos genéricos de replicação de dados, enquanto na secção 2.6 são expostos métodos de replicação de dados aplicados os jogos *on-line*. A replicação dos estados dos jogos *on-line* é abordada na secção 2.7. Por fim, na secção 2.8 é introduzido o conceito de *load balancing* associado aos jogos *on-line* bem como alguns dos métodos existentes.

2.1 Tolerância a Falhas

Um sistema diz-se tolerante a falhas, se quando ocorre uma falha este é capaz de manter o correcto funcionamento do sistema. A confiabilidade de um sistema indica a qualidade do serviço que é prestado e a confiança que pode ser depositada nesse serviço. Um sistema diz-se confiável, se tiver elevada probabilidade de se comportar como esperado. A confiança no funcionamento de um sistema pode ser posta em causa quando algum componente do

sistema não funciona de acordo com a sua especificação. Consideram-se três aspectos que podem por em causa o correcto funcionamento de um sistema: falha, erro e avaria [25] definidos de seguida:

- Falha: entende-se por falha, a alteração do funcionamento de um componente do sistema, podendo ser ao nível do *hardware* ou do *software*, por exemplo, exixte um sector do disco onde não se consegue escrever.
- Erro: um erro corresponde à manifestação de uma falha, isto é, à corrupção de elementos de dados afectando o estado do sistema. Considerando o exemplo anterior ocorreria um erro quando a nossa aplicação tentasse gravar um valor no sector do disco que estava em falha.
- Avaria: entende-se por avaria, uma alteração do comportamento do sistema em relação ao esperado pelo utilizador do sistema. No nosso exemplo, a avaria ocorreria quando a aplicação tentasse usar o valor escrito de forma errada no sector em falha.

Neste trabalho o mau funcionamento de qualquer componente do sistema será designado como uma falha.

Os principais atributos que determinam a confiança de um sistema são:

- Fiabilidade: espera-se que o sistema funcione de acordo com o esperado.
- Disponibilidade: probabilidade de o sistema estar operacional num dado instante.
- Segurança contra falhas acidentais - ausência de consequências catastróficas para o utilizador ou para o sistema.
- Confidencialidade: inexistência de acessos não autorizados à informação.
- Integridade: inexistência de alterações incorrectas do estado do sistema.
- Facilidade de manutenção: capacidade de um sistema, com avarias, ser reparado continuando a funcionar.

Nos sistemas críticos, isto é, sistemas que colocam em causa vidas humanas como sistemas de controlo de voo ou de centrais nucleares é fulcral a implementação de técnicas

de tolerância a falhas sendo a fiabilidade o aspecto mais importante. Na indústria dos jogos *on-line* o aspecto mais importante em termos de tolerância a falhas, é que o jogo mantenha uma elevadíssima taxa de disponibilidade de forma a não frustrar as expectativas dos adeptos mais entusiásticos. Todas as técnicas de tolerância a falhas envolvem alguma forma de redundância, com implicações no sistema em termos de tempo de execução e de recursos usados (memória, código, processador(es)). A replicação pode ser de informação, temporal, de *hardware* ou de *software*. Neste trabalho, vamos estudar a replicação de dados, replicando a base de dados usada e a replicação do estado do servidor *web* do jogo.

A replicação de dados permite lidar com falhas ao nível dos nós que contenham o servidor de base de dados e com falhas ao nível da comunicação de dados que impeçam o acesso aos dados. Neste tipo de replicação existem várias cópias com a mesma informação, o que implica que cada operação deverá ter em conta a última versão dos dados e a actualização consistente das réplicas. Na replicação de estados é distribuído o estado de um sistema por várias cópias. Quando estamos perante um sistema replicado, torna-se possível tornar uma falha transparente para os utilizadores, através do redireccionamento das ligações para uma cópia disponível.

As técnicas foram implementadas no jogo *on-line* FT-D&D tendo como principal objectivo a medição do custo associado em termos do decréscimo do *throughput*.

2.2 Jogos *On-line* Multi-Utilizador

Hoje em dia os jogos *on-line* atraem cada vez mais utilizadores, o que leva a que este seja um mercado em franca evolução. Dentro da categoria dos jogos *on-line*, os MMOGs são os que mais nos fascinam ao permitirem que sejam estabelecidas milhões de ligações simultâneas para o mesmo mundo virtual sendo este o aspecto que os distingue dos outros jogos. A maior parte dos Massive Multiplayer Gamess (MMGs) que existem, inserem-se nas seguintes categorias:

- Role-Playing Games (RPG): é um jogo onde os jogadores assumem os papéis das personagens. Como acontece em jogos como “Dungeons & Dragons”, “World of Warcraft”, entre outros.

- First Person Shooter (FPS): é um jogo onde o mundo do jogo é criado de acordo com a perspectiva visual da personagem e é testada a destreza dos jogadores através da

pontaria com armas de fogo. O jogo "Quake" está inserido nesta categoria e suporta um grande número de ligações simultâneas. Um dos FPS que ganhou maior fama a nível de jogadores é o "Counter-Strike".

- Real-Time Strategy (RTS): neste tipo de jogos, o jogador posiciona e evolui as suas tropas de forma a assegurar ou conquistar uma determinada área do mapa. O jogo "Warcraft III" é um dos jogos que constituem esta categoria.

Tipicamente o mundo de um jogo *multiplayer* é composto por um tema, personagens que podem ou não ser controladas pelos jogadores e objectos dinâmicos, como por exemplo, alimentos, armas ou pontuação. As personagens não controladas pelos utilizadores, designadas por Non-Player Character (NPC), são controladas por algoritmos automatizados, podendo em alguns casos representar os inimigos. A regra base na maior parte dos jogos *on-line* é que um jogador assume o papel de uma personagem num mundo virtual, onde as personagens pertencem a diferentes raças como elfos, humanos ou a classes como magos e guerreiros, entre outras.

Os jogos podem representar três realidades distintas, sendo elas o passado, o presente ou o futuro. Nessa realidade o jogador vive a experiência do jogo, através da sua personagem. O estado dos jogadores inclui a sua posição no mundo virtual e o estado da sua personagem no jogo. No geral, são permitidas três acções aos jogadores sendo elas: a interacção com outros jogadores, a alteração da sua posição dentro de um mundo e a interacção com objectos do jogo. Como o mundo resultante é muito grande, normalmente é dividido em regiões que estão estaticamente ligadas entre si.

A generalidade das implementações baseia-se na partição estática do mapa do jogo, na qual, cada partição (região) está atribuída a um servidor [8]. Os jogadores trocam de servidor, à medida que se deslocam de uma região para outra.

Nos jogos FPS, onde os jogadores controlam directamente a sua personagem são toleradas latências até um máximo de 180 milissegundos. Por outro lado, em jogos RTS como "Warcraft" são já tolerados alguns segundos de latência [24].

2.3 Arquitectura dos Jogos *On-line*

A arquitectura cliente-servidor é o paradigma principal na implementação de MMOG. Neste modelo, os jogadores ligam-se a um servidor central usando o *software* que é disponi-

bilizado ao cliente. O servidor é normalmente responsável por manter e distribuir o estado do jogo pelos jogadores, bem como, a gestão das contas dos utilizadores e a respectiva autenticação [24].

Hoje em dia, a maior parte dos jogos *on-line* simula, tipicamente, um mundo virtual, separado numa parte estática e noutra dinâmica. A parte estática representa, por exemplo, propriedades ambientais como as paisagens, construções e outros objectos inalteráveis. A parte dinâmica abrange objectos como as personagens NPC, controladas pelo computador, *items* que podem ser coleccionados pelos jogadores ou no geral, objectos que possam alterar os seus estados. Esses objectos são chamados de entidades e a soma de todas as entidades corresponde à parte dinâmica do mundo de um jogo. Para a criação de um progresso continuo no jogo, o estado do jogo é constantemente actualizado em tempo real num ciclo infinito designado por *real-time loop*. A figura 2.1 mostra uma interacção do servidor com *real-time loop* para jogos *multiplayer* baseados na arquitectura cliente-servidor. Numa jogada o cliente efectua uma acção que é enviada ao servidor (passo 1). O servidor faz as actualizações correspondentes (2) e envia o novo estado ao cliente (3).

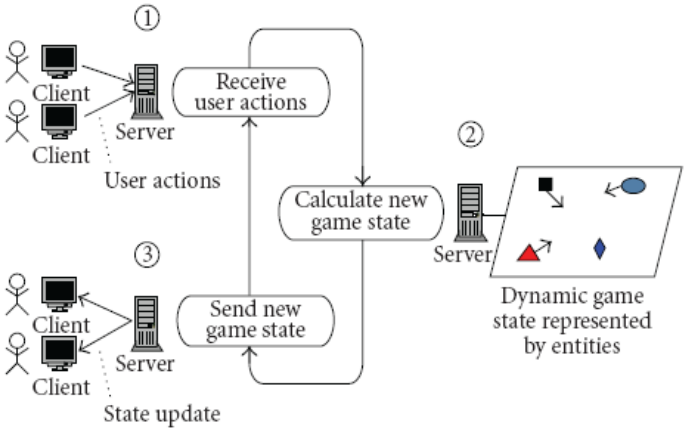


Figura 2.1: Interacção do servidor com *real-time loop* [14].

Dentro da categoria dos MMOG, os jogos *online* podem ser classificados segundo a forma como os clientes acedem ao jogo:

- Jogos baseados no *browser*, figura 2.2, requerem apenas o acesso à *internet* e a um *browser*. Possuem a vantagem de poderem ser acedidos de qualquer local com ligação à *internet*, não sendo necessário instalar qualquer tipo de *software* para aceder ao jogo.

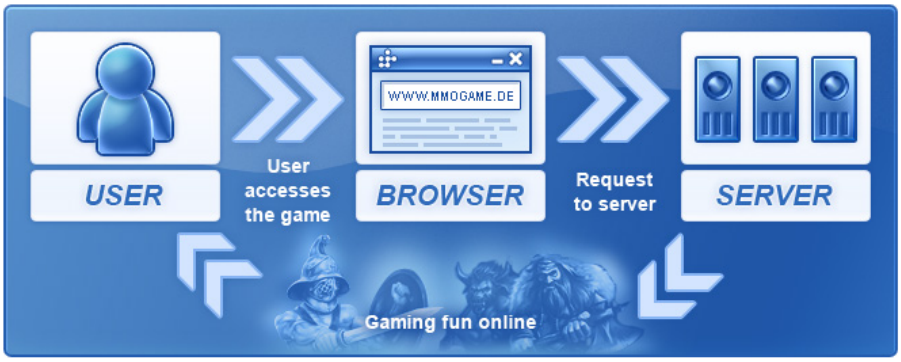


Figura 2.2: Baseado no *browser* [1].

- Jogos baseados no cliente, figura 2.3, assemelham-se mais aos jogos tradicionais, uma vez que é necessário instalar o *software* do jogo no computador em que o jogador pretende jogar. Até aos dias de hoje, os jogos baseados no cliente oferecem uma experiência gráfica superior à oferecida pelos jogos baseados no *browser*.

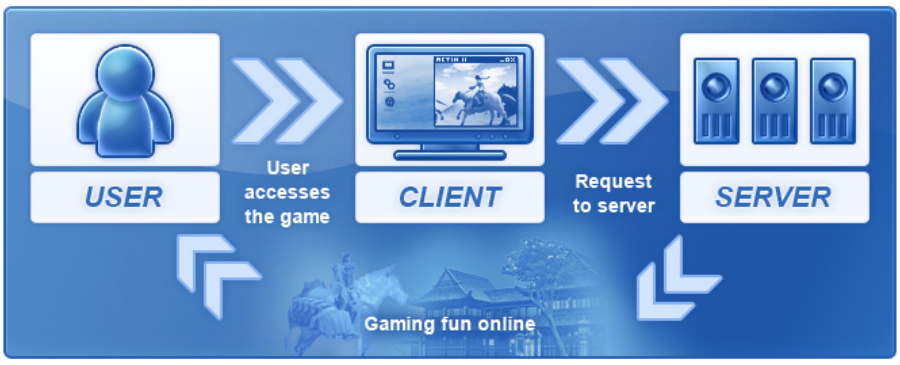


Figura 2.3: Baseado no cliente [1].

Ambientes virtuais em larga escala não podem ser eficientemente geridos num servidor único, mesmo com as actuais arquitecturas multi-core. Ambientes em larga escala podem incluir milhares de clientes, sendo o poder de processamento de uma máquina insuficiente. Para discutir os requisitos da escalabilidade, três abordagens afirmam-se actualmente:

- baseada em *cluster* de servidores [13, 32, 18, 5]

- baseada em Peer-to-Peer (P2P) [20, 11, 21]

- baseada em servidores distribuídos [11, 29, 38]

Destas três abordagens um *cluster* de servidores é a estratégia, normalmente adoptada, para suportar milhares de jogadores em simultâneo [24, 8], embora a mais dispendiosa. A escalabilidade é um problema importante nos servidores do jogo, pois os jogadores usufruem de complexas interacções, simulações físicas detalhadas, da possibilidade de interagir com um grande número de jogadores e objectos do jogo [8]. Esta abordagem oferece a garantia de melhores latências quando usada juntamente com protocolos específicos de encaminhamento, planeamento do tráfego e nós *gateway* [5].

A abordagem mais simples para dividir a carga entre os servidores de um *cluster* é a utilização de uma grelha onde cada servidor gere um conjunto de células. Os servidores migram as células para distribuir a carga [38, 18, 13]. O tamanho da célula pode ser configurado para ser igual ao campo de visão de um cliente, como sugerido em [13].

As abordagens P2P dependem dos clientes para calcular a lista de adjacência, eliminando assim a necessidade de um servidor para realizar tais cálculos. No entanto, estas abordagens não satisfazem as exigências da latência e podem obter um fraco desempenho se a densidade de clientes for elevada e os movimentos rápidos. Knutsson e os outros autores em [24] efectuaram a partição dos MMOG em regiões e analisaram este tipo de jogos num sistema P2P, onde o estado de todos os jogadores é mantido consistente através de *multicast* dentro de uma região. Então alguns *peers* tornam-se "donos" de uma região e assumem a responsabilidade de processar as tarefas do servidor para a tal região. Embora esta abordagem seja interessante sofre de vários problemas práticos. Os jogadores estão constantemente a entrar ou a abandonar o jogo sem aviso prévio havendo uma constante evolução na topologia do jogo. Outro problema é o "dono" de uma região bloquear ou ficar sem ligação à internet. Isto significa que são sempre necessárias máquinas de *backup* para substituírem os "donos" de uma região que se desligaram. Ao permitir que os computadores dos jogadores calculem parte da mecânica do jogo, torna-se muito complicado evitar o *cheating* [44].

Walker White e os outros autores em [48], descrevem as arquitecturas dos jogos classificando-as segundo a forma como os jogadores estão ligados uns aos outros através da *internet*. São considerados três tipos de jogos:

- Jogos sem ligação à *internet*: são aqueles que podem ser jogados localmente e não existe uma interacção com outras instâncias através da *internet*. A figura 2.4 mostra a arquitectura dos vários componentes de um jogo.

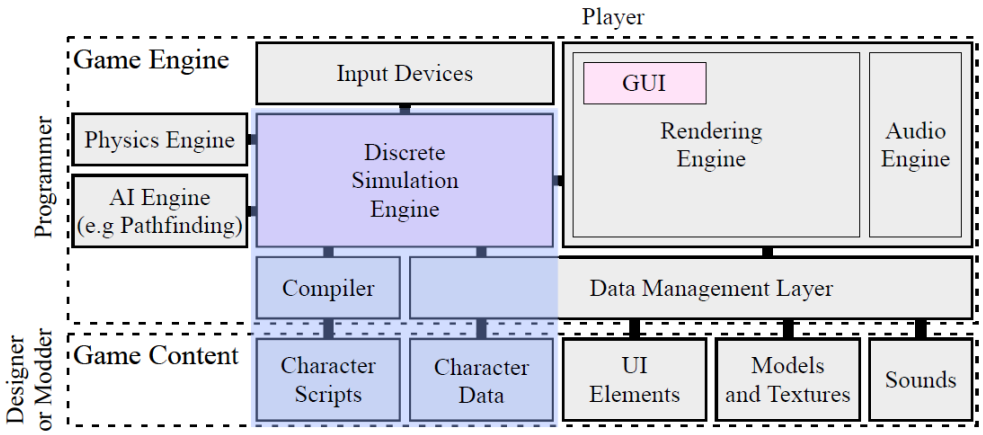


Figura 2.4: Arquitectura dos jogos sem ligação à *internet* [48].

- Jogos sem persistência de dados: são jogos com ligação à *internet* onde o estado do jogo é mantido apenas para uma sessão, ou seja, em jogos como *"Half-Life 2"* ou *"Halo 3"* se o jogador abandona o jogo durante a sessão, todos os seus estados são perdidos e o jogo inicializado quando o jogador resolve voltar. A arquitectura dos jogos sem persistência de dados é semelhante à arquitectura dos jogos sem ligação à *internet*, exceptuando o facto de possuírem uma camada de rede.

- Jogos com persistência de dados: são aqueles que possuem um armazenamento do estado do jogo em cada instante. Possuem, na maior parte das vezes, uma arquitectura cliente-servidor e tentam dar vida a mundos reais onde os jogadores evoluem as suas personagens em várias sessões. Daí surge a necessidade de guardar o actual estado do jogo. Para além, da base de dados onde estão armazenados todos os estados de um mundo, os jogos persistentes têm mais duas bases de dados, como representado na figura 2.5. Uma é utilizada para a monitorização do jogo e desempenha o mesmo papel nos jogos não persistentes. A outra corresponde a uma base de dados de autenticação, ou seja, possui informação sobre os jogadores que estão autorizados a aceder ao jogo.

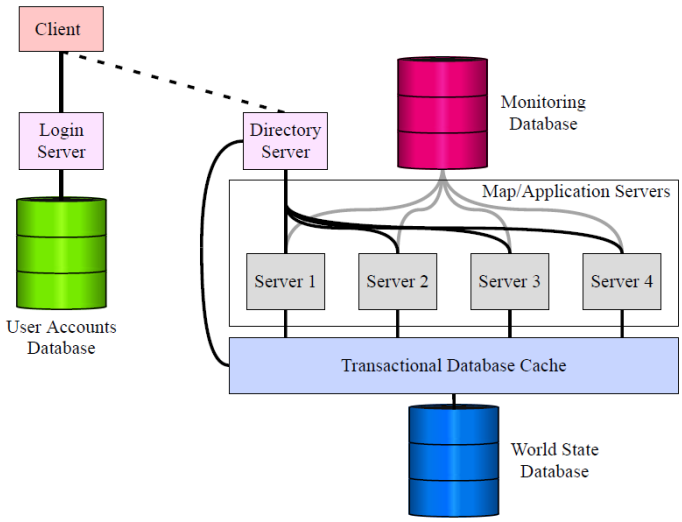


Figura 2.5: Arquitectura com persistência de dados [48].

2.4 O Jogo Dungeons & Dragons

O jogo D&D é um RPG de fantasia medieval, desenvolvido por Gary Gygax e Dave Arneson em 1972 e lançado em 1974 pela Tactical Studies Rules (TSR). É conhecido pelos seus curiosos dados de tabuleiro de até vinte lados, como pode ser observado na figura 2.6. O jogo D&D inspirou dezenas de jogos do género, como por exemplo, "Neverwinter Nights" e "Baldur Gate". Serviu de base para os jogos baseados no estilo RPG.

Figura 2.6: Dados do jogo de tabuleiro D&D.

Distinguiu-se dos jogos existentes ao permitir que um jogador crie e controle personagens que embarcam em aventuras imaginárias onde enfrentam monstros, reúnem tesouros, interagem entre si e ganham pontos de experiência para se tornarem cada vez mais fortes

à medida que o jogo evolui. Miniaturas ou marcadores num tabuleiro quadriculado são usados para representar as personagens no jogo, uma dessas representações está presente na figura 2.7. O D&D também introduziu o conceito de Mestre de jogo, que actua como juiz e narrador sendo responsável por manter o cenário de fantasia do jogo e por aplicar as regras a cada situação descrita. A última edição deste jogo foi lançada em 2008 pela "Wizards of the Coast".

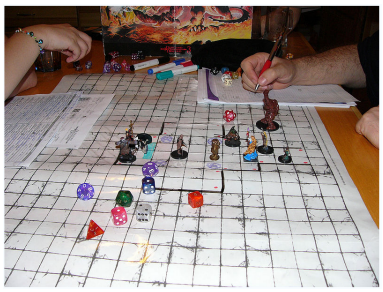


Figura 2.7: Tabuleiro do jogo D&D.

O D&D serviu de base a um jogo *on-line*, denominado por FT-D&D, no qual foi implementada a replicação de dados e de estados. A implementação inicial do D&D utilizada, foi desenvolvida no âmbito de um projecto de fim de curso [2].

Foi implementado no sistema operativo Windows usando Java, JSP, XHTML, JavaScript, JSP Scriptlet. Os dados do jogo e de login são armazenados num servidor de base de dados MySQL. O servidor web é o servidor "Web Apache Tomcat". Possui uma arquitectura multi-servidor com persistência de dados e é baseado no *browser*. No capítulo 3 será explicada a extensão FT-D&D e a implementação da replicação de dados e de estados.

2.5 Replicação de dados

Replicação de dados corresponde ao processo de manutenção de várias cópias de dados em diferentes locais, chamadas de réplicas. Assim, a disponibilidade pode ser aumentada uma vez que um sistema de base de dados pode tolerar melhor as falhas, a capacidade de resposta do sistema pode ser aumentada e os tempos reduzidos através da distribuição da carga das transacções por todas as réplicas.

Em [15], é sugerida a categorização dos protocolos de replicação segundo dois aspectos:

1. Onde as actualizações são executadas:

   - *Primary copy*: requer que todas as actualizações sejam realizadas na cópia primária e só depois nas outras cópias.
   - *Update everywhere*: a actualização dos dados pode ser realizada em qualquer réplica, por exemplo, dois clientes podem alterar o mesmo atributo, mas essa alteração é feita em servidores diferentes. Imaginando que o *cliente1* diz que o atributo a vai passar a ser $a = a+2$, enquanto *cliente2* diz que $a = a*2$, supondo que $a = 4$, para alguns servidores o atributo a terá o valor de 12 enquanto para outros será de 10, se não houver a adequada sincronização no acesso a variáveis partilhadas.

2. Quando são propagadas as actualizações:

   - *Eager*: os *updates* são propagados dentro dos limites de uma transacção, isto é, os utilizadores não recebem a notificação de *commit* até todas as cópias terem sido alteradas. Tipicamente esta replicação utiliza um esquema de *locking* para detectar e regular execuções concorrentes. Este é um protocolo síncrono e tipicamente usa o protocolo Read-One-Write-All-Available (ROWAA) [39].
   - *Lazy*: as actualizações são propagadas, após algum tempo da notificação de *commit* ter sido efectuada. É um protocolo assíncrono.

As quatro abordagens apresentadas podem ser combinadas conforme apresentado na tabela 2.1.

		Local de Actualização	
		Primary Copy	Update Everywhere
Propagação	Eager	Eager Primary Copy	Eager Update Everywhere
	Lazy	Lazy Primary Copy	Lazy Update Everywhere

Tabela 2.1: Combinação dos protocolos de replicação.

Cada uma destas abordagens possui os seus próprios problemas e vantagens. A abordagem *eager* atrasa a execução das transacções. Enquanto o protocolo *lazy* possui problemas com a consistência dos dados, se a cópia primária falha antes da propagação das alterações. A abordagem *eager* permite alcançar maior disponibilidade enquanto a *lazy* permite um acesso rápido e escalabilidade.

A abordagem *primary copy* requer que as actualizações sejam efectuadas na cópia primária. Por sua vez, o protocolo *update everywhere* requer um complexo controlo da concorrência e a resolução de conflitos.

Christian e Gustavo em [40], propuseram uma plataforma designada por *Ganymed*, para a replicação de dados ao nível da *middleware*, cuja intenção é fornecer escalabilidade sem que para isso a consistência dos dados seja afectada. A ideia principal inerente a esta plataforma é a utilização de um algoritmo de planeamento de transacções, o Replicated Snapshot Isolation with Primary Copy (RSI-PC). Este algoritmo é uma solução *primary copy* que separa as transacções de leitura das de actualização. As transacções de actualização são sempre direccionadas para a réplica principal, enquanto as transacções de leitura são tratadas por qualquer uma das restantes réplicas que actua como uma cópia de leitura. As transacções seguem o protocolo *lazy primary copy* onde a réplica primária representa um ponto crítico de falha. Para solucionar este problema proposeram a utilização de uma réplica de *backup* da réplica primária.

Matthias Wiesmann e André Schiper em [50] apresentaram uma comparação entre técnicas de replicação de dados baseadas em *total order broadcast*. Compararam essas técnicas entre si e com esquemas tradicionais de replicação de dados como o *locking* distribuído, replicação *lazy update everywhere* e *primary copy*.

As técnicas baseadas na comunicação em grupo dependem normalmente de uma primitiva designada por *total order broadcast*. Esta primitiva assegura a entrega das mensagens de forma fiável e pela mesma ordem em todas as réplicas.

São três as principais técnicas de replicação baseadas em *total order broadcast*: replicação activa, replicação baseada na certificação e replicação *weak vote*. Estas técnicas asseguram a serialização *one-copy* [6] e têm em comum as seguintes características:

- Não dependem do protocolo de *commit* atómico
- São técnicas de replicação *update-everywhere*
- Requerem interacções na rede de O(1)

Na técnica de replicação activa, também conhecida como replicação da máquina de estados, é colocada toda a transacção dentro de uma mensagem e enviada por *broadcast* (*total order broadcast*) para os servidores, como ilustrado na figura 2.8. Quando o servidor responsável S_d recebe uma transacção t de um cliente c, o servidor S_d envia t para todos os servidores usando *total order broadcast*. Todos os servidores entregam t e processam t. O processamento de t deve ser determinista. Pelo que se um servidor cancela a transacção t então todos os servidores a vão cancelar.

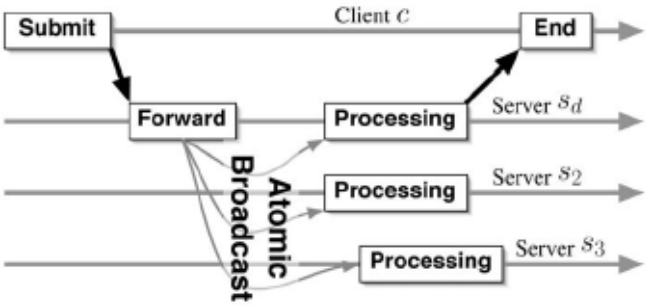


Figura 2.8: Esquema da replicação activa [50].

A figura 2.9 ilustra a técnica de replicação baseada na certificação [49]. Quando o servidor responsável S_d recebe uma transacção t de um cliente c, o servidor S_d executa a transacção t, mas atrasa as operações de escrita. Quando chega a altura de *commit* a transacção é enviada a todos os servidores usando *total order broadcast*. Após entregar a mensagem que contém t, cada servidor executa a fase de certificação determinista. A certificação decide se a transacção t pode *commit* ou deve ser cancelada.

Esta técnica partilha os requisitos de comunicação com a técnica de replicação activa. Por cada transacção é preciso apenas um *total order broadcast* e pode lidar com transacções interactivas. Por outro lado, esta é uma técnica optimista, o que significa que transacções em conflito apenas serão canceladas na fase de certificação. Esta técnica é adequada a situações em que os conflitos sejam raros.

A figura 2.10 ilustra o esquema de replicação *weak vote*[1]. A principal diferença entre esta técnica e a baseada na certificação é que o mecanismo de certificação determinista é substituído pela fase de *weak vote*, isto é, o servidor responsável S_d toma a decisão de

[1]Esta técnica é descrita em [22, 23] com o nome de protocolo *serializability*.

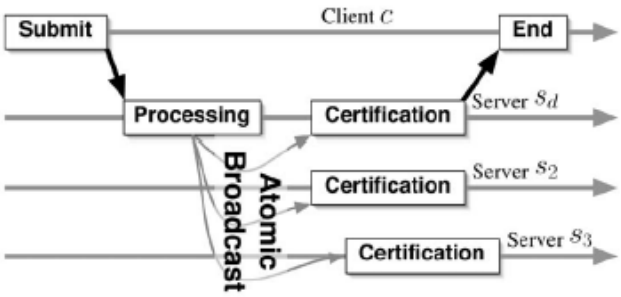


Figura 2.9: Esquema de replicação baseado na certificação [50].

commit ou *abort*. Designa-se por *weak vote* porque apenas o servidor responsável decide o resultado de uma transacção. Os outros servidores não influenciam a decisão e devem cumprir as decisões do servidor responsável.

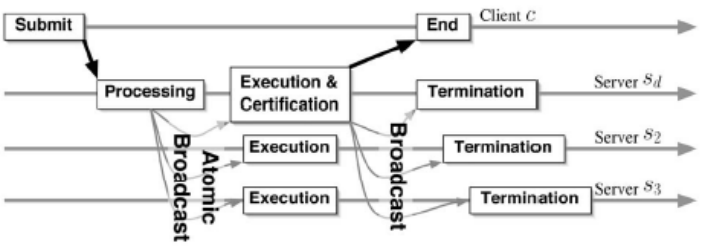


Figura 2.10: Esquema de replicação *weak vote* [50].

A avaliação do desempenho efectuada mostrou que as técnicas baseadas em *total order broadcast* superam significativamente os protocolos tradicionais de replicação de dados como o *locking* distribuído. Em condições óptimas, o desempenho é semelhante ao do protocolo *lazy*.

2.6 Replicação de Dados em Jogos *On-line*

A replicação de dados associada aos MMOG tem como objectivo a tolerância a falhas dos servidores de base de dados e simultaneamente, melhorar o tempo de resposta dos jogos se os clientes forem distribuídos pelas várias réplicas. Como nos Multiplayer Online Games

(MOG) existem muitas actualizações que correspondem às acções dos jogadores, estas têm de ser executadas em vários servidores e sincronizadas com as operações de leitura. A replicação de dados fornece mecanismos para distribuir e aplicar todas as actualizações, de forma eficiente a todas as réplicas. Por outro lado, fornece um controlo de concorrência para a sincronização entre operações de leitura e escrita [27].

Lin e os outros autores em [27], criaram um pequeno jogo do género *multiplayer* no qual testaram alguns protocolos de replicação de dados. Os testes foram realizados com a ajuda da *framework* MiddleSIR [26], que permite a implementação de vários protocolos de replicação de dados. A sua arquitectura é a apresentada na figura 2.11, onde todos os dados relacionados com o estado do jogo são armazenados numa base de dados relacional. As diferentes acções dos jogadores são modeladas como apenas de leitura ou como transacções de escrita sobre o estado do jogo. Para cada zona de um jogo MOG existe uma réplica MiddleSIR e uma réplica da base de dados. Se uma réplica de *middleware* falha, os clientes ligados a essa réplica são automaticamente redireccionados para outra réplica *middleware* fornecendo ao sistema tolerância a falhas de modo transparente.

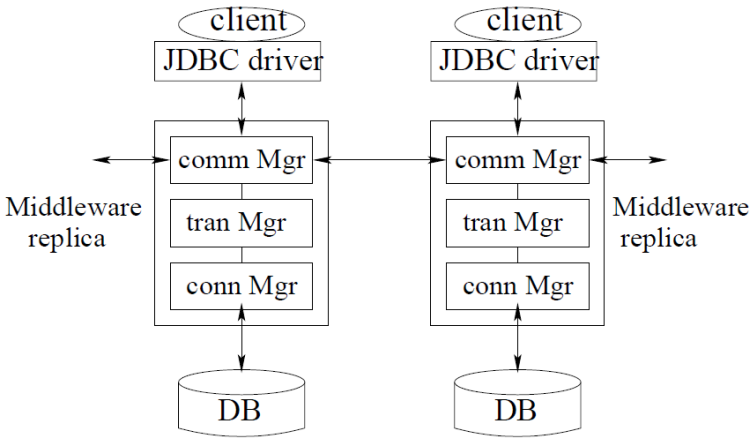


Figura 2.11: Arquitectura MiddleSIR [27].

Os protocolos testados foram o *lazy primary copy*, sendo este semelhante à abordagem Ganymed apresentada anteriormente, o simétrico, SRCA-REP[2] [27] e SEQ[3]. O protocolo *lazy primary copy* requer que as transacções sejam marcadas como de leitura ou de escrita. A abordagem simétrica requer que todas as operações sejam conhecidas no início de

[2]O protocolo SRCA-REP é uma abordagem *eager update everywhere* e utiliza Group Communication System (GCS).

[3]O protocolo SEQ é semelhante ao SRCA-REP, mas não utiliza GCS.

uma transacção. Estas duas abordagens colocaram as restrições apresentadas enquanto o SRCA-REP e o SEQ não colocaram qualquer restrição.

2.7 Replicação de Estados

Para além da performance e escalabilidade, o processamento do estado do jogo é vital para fornecer um serviço sem falhas, como é esperado pelos utilizadores. Os jogos Real-Time Multiplayer (RTM) são jogos em que o estado do jogo é actualizado frequentemente. A actual frequência varia entre 5 e as 35 actualizações por segundo, dependendo do tipo do jogo [37]. Os requisitos computacionais, para a actualização de um estado do jogo, crescem com o incremento do número das entidades do jogo que são processadas na actualização.

Carsten Gridwodz em [17], propõe um caminho para aumentar a escalabilidade nos MMOG através da separação do tráfego numa arquitectura *proxy* onde é definido um nível de urgência e relevância para cada elemento do jogo. Um nível elevado de urgência indica que é necessária uma latência baixa, enquanto um nível de relevância alto constitui um requerimento para uma segurança elevada.

Na literatura encontramos três propostas para replicação de estados em jogos *on-line*:

Gestão de estados baseado em diagramas de Voronoi

Hu, Chang e Jiang em [19], propuseram um esquema de gestão de estados para jogos, baseado em P2P, chamado de Voronoi State Management (VSM). O VSM particiona o ambiente virtual num número de regiões através dos diagramas de *Voronoi* [4] e nomeia clientes capazes como decisores para cuidar da gestão dos estados numa dada região. Para balancear a carga dos decisores VSM ajusta dinamicamente as fronteiras de uma região e insere novos decisores quando necessário.

Ao replicar os estados do jogo pelas regiões próximas, a tolerância a falhas e um eficiente controlo da consistência, podem ser alcançados sendo importantes dada a dinâmica das redes P2P e os requisitos de tempo real dos MMOG. O sistema de partição de um ambiente virtual pode facilmente ser integrado numa arquitectura de servidores em *cluster*, proporcionando assim uma ponte de transição da arquitectura cliente-servidor para P2P. Neste trabalho não é apresentada qualquer validação prática do esquema proposto.

Replicação de estados usando uma Real Time Framework (RTF)

Alexandre Ploss e os outros autores em [41], estudaram a portabilidade do jogo *"Quake 3 Arena Single-Server"* para uma arquitectura multi-servidor usando uma RTF [14] e a sua replicação de estados. A RTF é uma *middleware* que fornece suporte para o desenvolvimento de jogos *on-line* multi-servidor.

Para permitir a interacção entre todas as entidades do mundo de um jogo, cada servidor conhece todo o estado do jogo, isto é, possui uma cópia de todas as entidades. A figura 2.12 corresponde a um exemplo de replicação de estados. As entidades são distribuídas entre três servidores, de modo a que cada um possua uma lista denominada por "entidades activas", e uma lista com as entidades *"shadow"* que são replicadas de outros servidores.

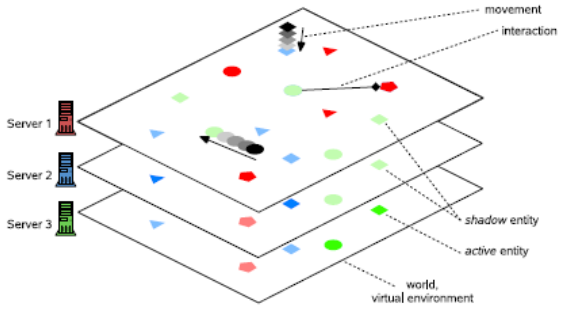


Figura 2.12: Replicação do mundo de um jogo [41].

A figura 2.13 mostra a arquitectura multi-servidor sugerida em [41] para a distribuição do processamento do estado do jogo e o fluxo de informação entre os processos distribuídos. Cada cliente liga-se a um servidor dedicado e envia os comandos do utilizador para o servidor a que está ligado recebendo a actualização do estado de todas as entidades visíveis. Deste modo, o processamento da distribuição dos estados é transparente para os clientes. A utilização de múltiplos servidores requer a comunicação entre servidores, as actualizações do estado são enviadas para as cópias das entidades *"shadow"* e a interacção entre as cópias das entidades activas e as *"shadow"* precisa de ser coordenada.

Replicação de estados usando uma arquitectura *proxy-server*

Müller, Gössling e Gorlatch em [37], propuseram a replicação de estados onde cada servidor que participa no processo de replicação possuia uma cópia completa do estado do

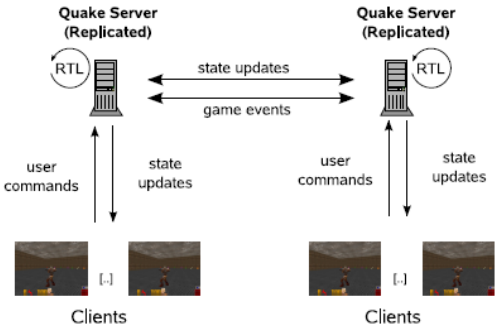


Figura 2.13: A arquitectura multi-servidor para o jogo "Quake" [41].

jogo. A replicação de estados foi implementada para os jogos "QFusion/Quake2". Para as entidades dinâmicas do jogo, como uma personagem NPC ou um item do jogo, existe um servidor dedicado que possui acesso de escrita e as suas próprias entidades activas. Todos os outros servidores possuem uma cópia *"shadow"* das entidades, a actualização dos seus estados é efectuada de acordo com a sincronização das mensagens enviadas pelo servidor principal, como ilustrado na figura 2.14.

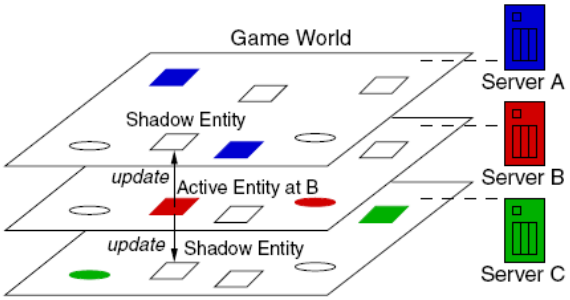


Figura 2.14: Replicação de estados [37].

Na arquitectura *proxy-server* os servidores devem ser colocados em diferentes Internet Service Provider (ISP) para que cada cliente se possa ligar ao servidor mais perto em termos da latência na comunicação, como representado na figura 2.15. Os servidores *proxy* estão ligados de forma P2P, se um *proxy* actualiza uma entidade activa local, é enviada uma mensagem de actualização para todos os servidores participantes que consequentemente actualizarão as suas cópias *shadow* locais.

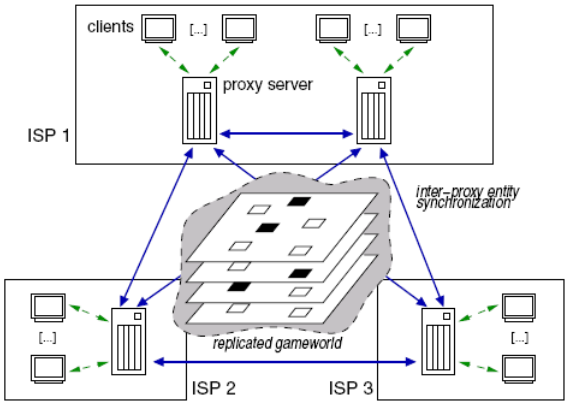


Figura 2.15: Arquitectura *proxy-server* [37].

A replicação de estados proposta em [37] e em [41] é muito semelhante. Em ambas as implementações, os servidores que participam na replicação possuem uma cópia total do estado do jogo e existem diferentes entidades activas para cada servidor. Para cada entidade do jogo, apenas um servidor tem privilégios de escrita. Sendo que em [37] é utilizada uma arquitectura *proxy-server*, (como apresentado na figura 2.15) e em [41] é utilizada uma arquitectura multi-servidor (como a apresentada na figura 2.13).

Replicação vs Distribuição de estados e escalabilidade

Para aumentar a escalabilidade de um mundo virtual, existem três tipos de *clusters* de servidores baseados na forma como é distribuído o estado do jogo podendo essa distribuição ser realizada com ou sem replicação de estados:

- Baseado em replicação, onde os servidores possuem uma topologia *point-to-point* e o estado do jogo é replicado entre os servidores, por exemplo, isto acontece nos servidores *proxy* [35, 36] e *mirror-servers* [10].

- Baseado em objectos, onde os objectos são uniformemente distribuídos entre os servidores [28, 30, 34].

- Baseado em regiões, onde os objectos do jogo são atribuídos através do particionamento espacial [9].

Os sistemas baseados em replicação possuem a vantagem de as acções dos jogadores poderem ser processadas em qualquer servidor, portanto, os jogadores podem ligar-se ao

servidor que possuir menor latência. No entanto a comunicação ponto a ponto, faz com que este sistema possua reduzida escalabilidade.

A abordagem baseada em objectos, geralmente, tenta separar os objectos de forma o mais regular possível pelos servidores. Esta abordagem processa a maior parte das acções localmente, a não ser que as acções ocorram perto dos limites de uma região. A comunicação entre os servidores pode assim ser constante para um determinado número de jogadores, alcançando uma melhor escalabilidade.

2.8 Balanceamento de Carga em Jogos *On-line*

Num ambiente cliente-servidor é comum ter mais que um servidor a processar os pedidos dos clientes. Em tais casos, deve existir um mecanismo para distribuir os pedidos dos clientes pelos servidores. Dependendo de como esse mecanismo funciona, os servidores podem ter diferentes cargas de trabalho. A carga de trabalho de um servidor é determinada pela quantidade de tempo de processamento necessária para executar todos os pedidos dos clientes atribuídos a esse servidor. Para alcançar melhores resultados no desempenho, um método de balanceamento de carga precisa de ser aplicado para minimizar as diferenças de trabalho entre os servidores. *Load balancing* é o termo dado a qualquer mecanismo que tenta atingir esse objectivo [46].

Através da distribuição dos pedidos é possível tornar uma falha, num servidor, transparente para os clientes que lá estavam ligados. Os mecanismos de *load balancing* são então uma parte fundamental num sistema tolerante a falhas.

Fengyun, Simon e Graham em [28], consideram que num *cluster* de servidores para Massive Multiplayer On-line Role-Playing Game (MMORPG) existem dois caminhos para implementar o *load balancing* sendo eles:

- Com base no jogador - os jogadores são distribuídos por diferentes servidores quando se juntam a um jogo.
- Com base na interacção - os servidores gerem a distribuição dos recursos de processamento com base nos padrões de interacção dos jogadores.

A abordagem de *load balancing* baseada no jogador é semelhante às técnicas *standard* de *load balancing* utilizadas em várias aplicações, baseadas no servidor. Estas abordagens

invocam a Network Address Translator (NAT), ou *software* equivalente, para distribuir eficientemente os clientes pelos servidores usando técnicas de *load balancing*, como por exemplo, *round robin* [46]. A tabela NAT "sabe" qual é o servidor a que um cliente específico está ligado, redireccionando todos os pedidos para esse servidor durante a sessão do cliente.

Usando apenas a tabela NAT para *load balancing*, é possível melhorar o desempenho do sistema direccionando todos os pedidos de uma sessão para o mesmo servidor. Esta abordagem de *load balancing* permite que sejam removidos servidores do *cluster* para manutenção ou adicionados, quando necessário, sem prejudicar os jogadores de outros servidores. Nos MMORPG a distribuição dos jogadores por mundos duplicados, pequenos *cluster's*, possui uma relação próxima com esta forma de *load balancing*, com a diferença de serem os jogadores, a escolher qual o mundo duplicado que pretendem visitar e não a NAT.

Uma vez que os jogadores estejam localizados num mundo replicado, continua a ser necessário efectuar o *load balancing*, através do *cluster* de servidores que suporta o mundo escolhido. Como os jogadores estão distribuídos pelos servidores, surge a necessidade dos servidores comunicarem entre si, para que os jogadores ligados a diferentes servidores possam interagir.

Simon e os outros autores em [44], propuseram uma estrutura P2P para a redução das falhas nos MMOG. Através da utilização de uma abordagem baseada em *cluster* dividem o mundo de um jogo em grupos de jogadores e não em regiões. Os *clusters* estão distribuídos em diferentes nós da rede P2P. Ao proporem um *cluster* de *peers* permitem que as máquinas se possam ligar ou desligar dinamicamente a um ou de um *cluster* de *peers*, utilizando o *load balancing* de acordo com as acções dos jogadores. Quando a carga de um servidor excede um limite, por exemplo, quando estão demasiados jogadores num *cluster* a carga pode ser reduzida através de três mecanismos:

- Mover todo o *cluster* de um nó para outro.
- Mover um ou alguns jogadores de um *cluster* para outro.
- Dividir um *cluster* em duas partes e mover uma das partes para outro servidor.

Esta movimentação dos jogadores permite reduzir a carga dos servidores e diminuir o número de falhas no jogo.

Capítulo 3

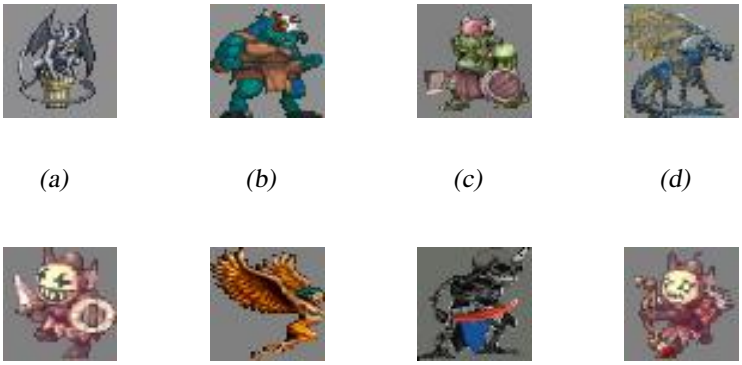
Fault Tolerant Dungeons & Dragons (FT-D&D)

O jogo D&D serviu de base para um novo jogo, denominado por FT-D&D onde estudamos a replicação de dados e de estados. FT-D&D é um jogo de fantasia medieval, que pode ser encarado como uma forma de entretenimento em grupo, uma vez que para a realização de um jogo são necessários cinco jogadores. Cada um dos jogadores tem obrigatoriamente de escolher uma das personagens disponíveis para o representar. Mestre, Clérigo, Elfo, Humano e Feiticeiro são as personagens do jogo, cujas imagens com excepção do Mestre são apresentadas na figura 3.1. Cada personagem possui características distintas ao nível da defesa, ataque ou vida, o que lhe confere uma diferente evolução. O jogador que representa a personagem Mestre é quem define a história do jogo, controlando os monstros, e escolhendo os cenários (mapas). Na figura 3.2 estão presentes os monstros que o mestre controla e adiciona ao jogo. O FT-D&D é um jogo com vários níveis no qual os jogos inacabados podem ser retomados, em qualquer altura, do ponto em que os deixaram, uma vez que possui persistência de dados. Para o jogo não se tornar monótono e para promover uma maior interacção entre jogadores, foram criados dois *chat's* que permitem a comunicação de forma rápida e intuitiva. O primeiro *chat* corresponde a um *chat* global onde os jogadores podem combinar qual o jogo em que vão participar ou até mesmo arranjar jogadores para iniciarem um jogo. O segundo *chat* tem como objectivo a promoção da interacção entre jogadores que participam no mesmo jogo, permitindo que o mestre conte a história do jogo e os jogadores debatam qual a estratégia a adoptar.

A figura 3.3 mostra um dos cenários do jogo onde podemos ver um tabuleiro com as personagens e alguns monstros (ao centro) as características do jogador (à esquerda) e a

(a) Clérigo *(b) Feiticeiro* *(c) Humano* *(d) Elfo*

Figura 3.1: Imagens das personagens do jogo.



(a) *(b)* *(c)* *(d)*

(e) *(f)* *(g)* *(h)*

Figura 3.2: Imagens dos monstros do jogo.

informação dos jogadores que participam no jogo e a respectiva personagem (à direita). Na parte de baixo da figura vemos a área de texto do *chat* que permite a comunicação entre os jogadores.

Neste capítulo serão abordados os procedimentos efectuados para tornar o jogo FT-D&D tolerante a falhas, ou seja, a implementação da replicação de dados e de estados e o redireccionamento dos pedidos para os servidores *web*, ou seja, é realizado o *load balancing*. Na secção 3.1 é descrita a aplicação do protocolo *lazy primary copy* para a replicação de dados e justifica-se a sua escolha. Na secção 3.2 é apresentada a implementação da replicação de estados no jogo FT-D&D e a utilização de *load balancing*. Por fim, na secção 3.3 é apresentada a arquitectura tolerante a falhas implementada para este jogo.

Figura 3.3: Um dos tabuleiros do jogo FT-D&D.

3.1 Replicação de Dados

A replicação de dados consiste em ter os dados replicados por vários servidores de base de dados que se podem comportar como servidores de *backup* [15, 27]. Dos vários protocolos de replicação de dados que existem, o *Lazy primary copy* foi o implementado no jogo FT-D&D.

Com este protocolo, o processo de replicação de dados é dividido em três etapas, como pode ser observado na figura 3.4.

1. Os clientes efectuam pedidos de leitura ou de escrita que são direccionados para o servidor de base de dados *master*.

2. Após a actualização dos dados ter sido realizada no servidor *master*, é enviada a notificação de *commit* aos clientes.

3. A actualização dos dados é transmitida para o servidor *slave*.

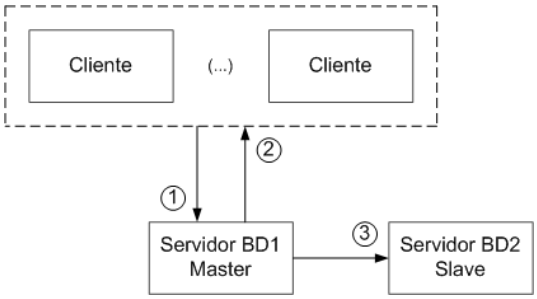


Figura 3.4: Esquema da replicação de dados no jogo FT-D&D.

A comunicação neste protocolo é assíncrona o que permite que se uma falha ocorrer num dos servidores os outros não sejam afectados. O ponto crítico dos protocolos assíncronos reside na possível inconsistência dos dados, no caso de o *master* falhar antes da propagação das alterações. Este problema aplicado ao jogo FT-D&D corresponde à falha do *master* imediatamente após a conclusão de uma jogada, mas antes de os dados serem enviados para o *slave*. A probabilidade de tal ocorrer é pequena e nunca ocorreu nos testes efectuados.

Nesta implementação, tanto os pedidos de escrita, como os pedidos de leitura são sempre efectuados no servidor configurado como *master*. Os pedidos poderiam ser distribuídos pelos servidores, ou seja, os pedidos de leitura direccionados para os *slaves* e os de escrita para o *master* para melhorar o desempenho do jogo. Isso não foi feito por limitações de tempo e por não ser o objectivo do trabalho.

A abordagem *lazy update everywhere* não foi a implementada porque obrigaria a que o jogo assegurasse a consistência dos dados. No protocolo *lazy update everywhere* a actualização dos dados pode ser efectuada em qualquer servidor e a propagação das alterações é realizada após os clientes serem notificados com o *commit*. Neste caso, poderia ocorrer que dois clientes alterassem o mesmo objecto, por exemplo, se num jogo existe uma poção mágica e o *cliente A* é o primeiro a bebe-la, quando o *cliente B* chega à mesma poção, o frasco deveria estar vazio ou conter apenas uma parte da poção. Se a consistência dos dados não fosse garantida poderia ser gerada a situação em que ambos os clientes são os primeiros a beber a mesma poção ficando assim com os mesmos poderes.

Na abordagem *eager* a actualização dos servidores, que participam na replicação, seria englobada numa única transacção, ou seja, as actualizações são propagadas para os servidores participantes, dentro dos limites de uma transacção, o que significa que os utilizadores só recebem a notificação de *commit* após todos os servidores terem sido actualizados com

sucesso. O facto da comunicação do protocolo *eager* ser síncrona, levanta o problema de os dados só se tornarem definitivos após todas as bases de dados terem sido alteradas, o que implica um atraso na execução das transacções. No caso da falha de uma base de dados, durante uma transacção de actualização, seria feito um *rollback* em todas as réplicas. O problema existe quer o protocolo seja associado com o *primary copy*, quer com o *update everywhere*.

O processo de replicação no MySQL é dividida em três etapas:

1. O *master* guarda as alterações dos dados num registo binário.

2. O *slave* copia os eventos do registo binário do *master* para o seu *relay log*.

3. O *slave* reproduz os eventos do *relay log* aplicando-os para alterar os seus próprios dados.

A primeira parte do processo é a criação de um registo binário no *master*. Imediatamente antes de cada transacção de actualização dos dados no *master*, o *master* guarda as alterações no seu registo binário. Após escrever os eventos no registo binário, o *master* diz aos mecanismos de armazenamento para fazer o *commit*. O próximo passo é para o *slave* copiar o registo binário do *master* para o seu próprio disco rígido, para dentro do *relay log*. Para começar inicia uma *thread* chamada I/O *slave thread*. Esta *thread* abre uma ligação comum do cliente ao *master* e inicia um processo especial de *bin log dump*. O processo do *bin log dump* lê os eventos do *log* binário do *master*. Se atingir o estado do *master* o *slave* espera que o *master* lhe sinalize a presença de novos eventos. A *thread* I/O escreve os eventos no *relay log* do *slave*. A *thread* SQL do *slave* efectua a última parte do processo. Esta *thread* lê e re-executa os eventos no *relay log* actualizando deste modo a informação do *slave* para que corresponda à do *master*. Enquanto esta *thread* acompanha a *thread* de I/O o *relay log* fica normalmente na *cash* do sistema operativo deste modo os *relay log's* têm muito pouco *overhead*. Os eventos executados pela *thread* SQL podem opcionalmente ir para o registo binário do *master*.

Esta arquitectura de replicação separa o processo de ir buscar e replicar os eventos no escravo permitindo que estes sejam assíncronos, isto é, a *thread* I/O pode funcionar independentemente da *thread* SQL. Também ela coloca algumas restrições no processo de replicação, sendo a mais importante o facto da replicação ser serializada no *slave*. Isto

significa que as actualizações que podem ter ocorrido em paralelo no *master*, não podem ser paralelizadas no *slave*.

A replicação de dados no FT-D&D foi realizada entre dois servidores de base de dados MySQL. Este servidor de base de dados permite que a replicação de dados siga dois modelos, o modelo *master-master* ou *master-slave* sendo que o modelo *master-master* divide-se em dois modos, o modo activo-activo e activo-passivo sendo este último o implementado.

A replicação de dados em servidores MySQL pode seguir um dos três modelos apresentados de seguida [45]:

- *Master-Master*: a replicação *master-master* também conhecida como *dual-master* ou replicação bidireccional, pode ser realizada de dois modos distintos:

  ◇ Modo activo-activo: neste modo existem dois servidores e ambos são configurados como *master* e como *slave* em simultâneo. Este modo é normalmente utilizado entre bases de dados separadas geograficamente e onde são requeridos privilégios de escrita para ambas as cópias. O maior problema deste modo reside na resolução de conflitos provocados pelas alterações. Quando é alterado o valor de um atributo, nas duas máquinas em simultâneo, leva a problemas sérios na consistência dos dados, uma vez que o mesmo atributo pode ter valores diferentes em cada máquina. Uma situação que exemplifica este problema é:

  ○ Na primeira máquina é executado:
  mysql > UPDATE tabela1 SET a = a + 1

  ○ Na segunda é executado:
  mysql > UPDATE tabela1 SET a = a * 2

  O resultado para o mesmo atributo passa a ter um valor diferente para cada máquina. Supondo que o valor original de a era 2, para a primeira passou a ser de 3 enquanto para a segunda máquina passou a ser 4. Os conflitos de dados neste modo são muito frequentes e a sua resolução muito difícil, por isso, este modo não foi o escolhido para a replicação de dados.

  ◇ Modo activo-passivo: esta configuração permite que a troca entre o servidor passivo e activo seja realizada facilmente porque as configurações dos servidores são simétricas. Permite ainda a manutenção, optimização das tabelas,

actualização do sistema operativo e a realização de outras tarefas sem que para isso exista um período em que o servidor não esteja disponível.

- *Master-Slave*: neste modelo são utilizados dois ou mais servidores de base de dados, um como *master* e o(s) outro(s) como *slave(s)*. É o modelo mais simples porque não existe interacção entre *slaves* e cada um comunica apenas com o *master*. É a topologia mais comum porque evita a maior parte das complexidades existentes nas outras configurações, por exemplo, se os *slaves* falham na replicação no mesmo ponto lógico, todos os *slaves* irão ler da mesma posição física nos *log's* do *master*.

A replicação *master-slave* adequa-se a situações em que é pretendido ter uma cópia dos dados (*backup*) sem necessidade de escrita. A replicação *master-master* com o modo activo-activo é implementada quando surge a necessidade de escrita em simultâneo nas duas bases de dados. Por fim, o modo de replicação activo-passivo, que corresponde ao modo implementado, é adequado para sistemas em que é pedida elevada disponibilidade, como é o caso dos jogos *on-line*.

Foram configurados dois servidores, um que desempenha o papel de *master* e outro de *slave*, para permitirem a replicação de dados entre si. O servidor *slave* pode-se comportar como *master* no caso de este falhar ou se surgir a necessidade de parar o servidor para uma manutenção, por exemplo, executar uma actualização.

O envio das mensagens, com os dados a replicar, é feito de forma assíncrona e através de Transmission Control Protocol (TCP)/Internet Protocol (IP). O protocolo utilizado para efectuar a ligação entre o jogo e as bases de dados é o protocolo cliente-servidor [45]. Este protocolo é *half-duplex*, ou seja, a uma determinada altura os servidores MySQL tanto podem enviar como receber mensagens, mas não em simultâneo. A ligação com os servidores de bases de dados é realizada através de um *driver* JDBC do MySQL.

3.1.1 *Load Balancing*

A partir do momento em que a replicação entre as bases de dados está configurada, é necessário tornar uma falha transparente para os clientes, sendo este, um ponto fulcral na implementação de um sistema tolerante a falhas. Na actual implementação do FT-D&D foi concebido um mecanismo de *load balancing*, para as bases de dados, estando embutido no próprio jogo. Este *load balancing* é conseguido ao fornecer os endereços IPs dos servidores de base de dados ao jogo. Como já foi referido, a ligação entre o jogo e os servidores de base

de dados é realizada com a ajuda de um *driver* JDBC do MySQL. Sempre que é detectada uma falha no servidor de base de dados onde o FT-D&D está ligado, é realizada a troca de servidores e o redireccionamento dos pedidos para o novo servidor.

Como a configuração das bases de dados segue um modelo *master-master* activo-passivo a alteração para *master* do servidor que desempenhava o papel de *slave* e a alteração contrária é relativamente simples.

A alteração de *slave* para *master* é conseguida ao introduzir o seguinte comando no *slave* a alterar, CHANGE MASTER TO MASTER_HOST = 'IP'.

Para a alteração contrária, a transformação de *master* em *slave* basta introduzir, RESET SLAVE, no *master* a alterar, quando este recupera da falha como ilustrado na figura 3.5. Estes comandos estão definidos no próprio jogo e quando é detectada uma falha, o jogo procede à alteração da configuração dos servidores (passo 1), é executada a mudança de servidor de base de dados (2 e 3).

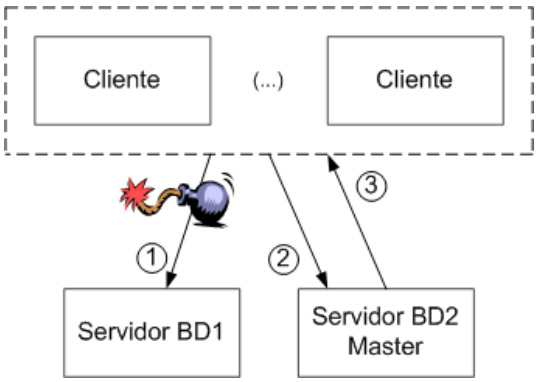


Figura 3.5: Representação da mudança de servidor de base de dados no jogo FT-D&D.

Quando o servidor que falhou recupera, é transformado em *slave* ficando a receber as actualizações do novo *master* e mantendo assim a consistência dos dados entre os servidores. A recuperação do servidor que falhou é ilustrada na figura 3.6. Onde (1) corresponde ao envio de pedidos de escrita ou de leitura para o actual servidor *master*; (2) é a resposta ao pedido e notificação de *commit*; (3) é o envio das alterações efectuadas, no servidor *master*, para o servidor *slave* assim que este recupera da falha e é transformado em *slave*.

A falha do servidor *slave* é imperceptível aos clientes, uma vez que estão ligados ao servidor *master*. Quando o servidor *slave* recupera, recebe as actualizações do *master* e assim mantém a consistência dos dados. Esta falha torna-se perceptível para os clientes,

se acontecer uma falha no servidor *slave* e de seguida uma no servidor *master* antes do primeiro recuperar ou vice-versa. Se esta situação se verificar o jogo fica bloqueado até que o último servidor configurado como *master* volte a estar disponível.

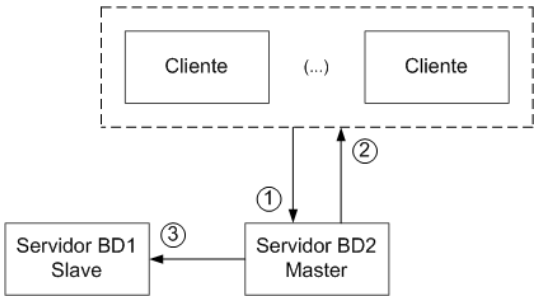


Figura 3.6: Recuperação do servidor de base de dados que falhou, no jogo FT-D&D.

3.2 Replicação de Estados

O estado de um jogo corresponde ao conjunto dos objectos estáticos e os objectos dinâmicos. No caso do FT-D&D os objectos estáticos correspondem, por exemplo, aos mapas em que decorre a acção de um jogo. Os objectos dinâmicos correspondem a objectos que estão constantemente a ser alterados, como é o caso da vida, da posição de uma personagem ou do número de pontos. A replicação do estado do jogo permite tolerar falhas no servidor *web*.

O servidor *web* do jogo pode ser configurado em *stand alone* ou em grupo, isto é, em *cluster*. A configuração em *cluster* permite a replicação dos vários estados do jogo entre os membros. Em caso de falha os utilizadores serão redireccionados para um servidor de *backup* através de técnicas de *load balancing* que serão abordadas na secção 3.2.2.

A replicação de estados no jogo FT-D&D foi implementada num *cluster* de servidores *web* Tomcat, composto por duas instâncias (Servidores *web* Tomcat). A definição dos membros do *cluster* pode ser realizada de duas formas [7]:

- Forma estática: Na forma estática é necessário definir em todas as instâncias o endereço IP dos membros que pertencem ao *cluster*. Esta é uma solução para redes em que a comunicação *multicast* não é possível e para *clusters* de dimensão reduzida e/ou IP

fixo. Mostra-se inviável em *clusters* em que é requerida uma elevada disponibilidade, pois para a remoção ou inserção de um membro no *cluster* é necessário alterar os ficheiros de configuração de todos os membros.

- Forma dinâmica: A forma dinâmica permite a adição/remoção de instâncias do *cluster* sem que para isso seja necessário alterar a configuração dos membros activos. Tal é possível, devido à auto-descoberta dos membros do *cluster* que é realizada através da comunicação *multicast*. Esta é assim uma solução para *clusters* de grande dimensão e/ou com IP dinâmico sendo esta a forma adoptada para a implementação do *cluster*.

O envio das mensagens com os dados do estado do jogo, entre os membros do *cluster*, pode ser realizado de forma síncrona ou assíncrona podendo ainda ter ou não um *acknowledged*.

- Modo assíncrono: neste modo os membros secundários do *cluster* só recebem as actualizações do estado do jogo após o novo estado ter sido enviado ao cliente, como representado na figura 3.7. Por ser assíncrono possui a vantagem de reduzir o tráfego na rede e de reduzir o tempo de resposta permitindo que o cliente visualize o novo estado de forma mais rápida. Estas vantagens foram os motivos que levaram à escolha da comunicação assíncrona para o *cluster* implementado.

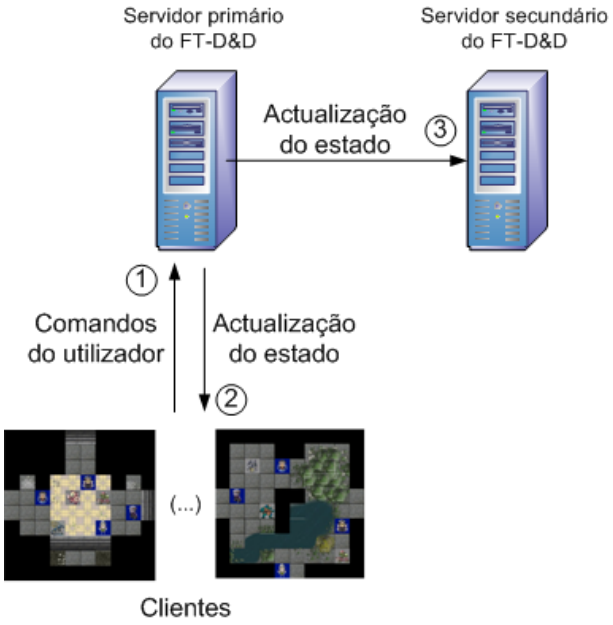


Figura 3.7: Esquema da replicação assíncrona no jogo FT-D&D

- Modo síncrono: neste modo os dados do estado do jogo são replicados para os membros do *cluster* antes do novo estado ser enviado ao cliente. A garantia da replicação dos dados representa uma vantagem deste modo. Contudo, o facto de o cliente só receber o novo estado após todos os membros do *cluster* serem actualizados, representa um atraso na resposta ao pedido do cliente. Este atraso não é justificável em jogos *on-line* uma vez que é exigida por parte dos utilizadores uma resposta quase que imediata aos seus pedidos.

O envio das mensagens com os estados do jogo, entre os membros do *cluster*, tem necessariamente de seguir a ordem pela qual as acções ocorreram. Para que a ordem seja garantida foram sincronizados os relógios dos membros do *cluster*. A sincronização foi efectuada com o Network Time Protocol (NTP) como explicado na secção 3.2.3.

A replicação de estados escolhida, segue o modelo *all-to-all* ou seja, cada membro do *cluster* envia mensagens com os dados do estado para os membros activos. Assim, ao ter sessões permanentemente replicadas a probabilidade de ocorrer perda de dados entre as instâncias Tomcat é reduzida. Podemos considerar como ponto crítico a situação em que o servidor *web* principal falha após a actualização do estado do jogo mas antes da replicação ser efectuada. A probabilidade de falha ocorrer, neste ponto, é miníma e nunca ocorreu nas simulações efectuadas.

3.2.1 Implementação

Para a implementação da replicação de estados no jogo FT-D&D foram configuradas as instâncias Tomcat que pertencem ao *cluster* de forma a replicarem todas as alterações sobre o estado de todas as sessões em execução nesse servidor. Cada utilizador possui uma sessão única que é criada assim que é efectuado o acesso ao jogo sendo enviado um *cookie* onde é armazenado o identificador da sessão que será explicado na secção 3.2.2.

O código seguinte corresponde ao inserido em cada servidor *web* para a criação do *cluster*, detecção da falha de um servidor e ao envio de mensagens com os dados do estado do jogo:

Listing 3.1: Código para a replicação de estados.

```
<Engine name="Catalina" defaultHost="localhost" jvmRoute="worker1">

  <Cluster className="org.apache.catalina.ha.tcp.SimpleTcpCluster">

        <Manager className="org.apache.catalina.ha.session.DeltaManager"
                 expireSessionsOnShutdown="false"
                 notifyListenersOnReplication="true"/>


        <!-- Criação do cluster de forma dinâmica -->
        <Channel className="org.apache.catalina.tribes.group.
            GroupChannel">
             <Membership className="org.apache.catalina.tribes.
                membership.McastService"
                  address="228.0.0.4"
                  port="45564"
                  frequency="500"
                  dropTime="3000"/>


             <Receiver className="org.apache.catalina.tribes.
                transport.nio.NioReceiver"
                  address="auto"
                  port="4000"
                  autoBind="100"
                  selectorTimeout="5000"
                  maxThreads="6"/>

             <!-- Envio das mensagens com os dados do estado do jogo
                -->
             <Sender className="org.apache.catalina.tribes.transport.
                ReplicationTransmitter">
                  <Transport className="org.apache.catalina.tribes
                     .transport.nio.PooledParallelSender"/>
             </Sender>

             <!-- Detecção de uma falha no servidor -->
             <Interceptor className="org.apache.catalina.tribes.group
                .interceptors.TcpFailureDetector"/>
             <Interceptor className="org.apache.catalina.tribes.group
```

```
                    .interceptors.MessageDispatch15Interceptor"/>
34          </Channel>
35
36          <Valve className="org.apache.catalina.ha.tcp.ReplicationValve"
37                  filter=""/>
38
39          <Valve className="org.apache.catalina.ha.session.
                JvmRouteBinderValve"/>
40
41          <ClusterListener className="org.apache.catalina.ha.session.
                JvmRouteSessionIDBinderListener"/>
42          <ClusterListener className="org.apache.catalina.ha.session.
                ClusterSessionListener"/>
43          </Cluster>
```

Após a configuração da replicação entre os membros do *cluster*, é possível observar se o processo de auto-descoberta está a ser efectuado de forma correcta. Esta verificação é realizada através da análise da informação presente nos ficheiros **.log*. É apresentado a seguir o excerto do ficheiro onde é possível efectuar essa verificação:

Listing 3.2: Código para a replicação de estados.

```
1  INFO: Starting Servlet Engine: Apache Tomcat/6.0.18
2  21/Jul/2009 11:36:57 org.apache.catalina.ha.tcp.SimpleTcpCluster start
3  INFO: Cluster is about to start
4  21/Jul/2009 11:36:57 org.apache.catalina.tribes.transport.ReceiverBase
       bind
5  INFO: Receiver Server Socket bound to:/jogo.ubi.pt:4000
6  21/Jul/2009 11:36:57 org.apache.catalina.tribes.membership.
       McastServiceImpl setupSocket
7  INFO: Setting cluster mcast soTimeout to 500
8  21/Jul/2009 11:36:57 org.apache.catalina.tribes.membership.
       McastServiceImpl waitForMembers
9  INFO: Sleeping for 1000 milliseconds to establish cluster membership,
       start level:4
10 21/Jul/2009 11:36:58 org.apache.catalina.tribes.membership.
       McastServiceImpl waitForMembers
11 INFO: Done sleeping, membership established, start level:4
12 21/Jul/2009 11:36:58 org.apache.catalina.tribes.membership.
```

```
    McastServiceImpl waitForMembers
13 INFO: Sleeping for 1000 milliseconds to establish cluster membership,
    start level:8
14 21/Jul/2009 11:36:59 org.apache.catalina.tribes.membership.
    McastServiceImpl waitForMembers
15 INFO: Done sleeping, membership established, start level:8
16 21/Jul/2009 11:37:04 org.apache.catalina.ha.session.DeltaManager start
17 INFO: Register manager /jogo to cluster element Engine with name
    Catalina
18 21/Jul/2009 11:37:04 org.apache.catalina.ha.session.DeltaManager start
19 INFO: Starting clustering manager at /jogo
20 21/Jul/2009 11:37:04 org.apache.catalina.ha.session.DeltaManager
    getAllClusterSessions
21 INFO: Manager [jogo.ubi.pt\#/jogo]: skipping state transfer. No members
    active in cluster group.
22 21/Jul/2009 11:37:04 org.apache.catalina.ha.session.JvmRouteBinderValve
    start
23 INFO: JvmRouteBinderValve started
24 21/Jul/2009 11:37:04 org.apache.coyote.http11.Http11Protocol start
25 INFO: Starting Coyote HTTP/1.1 on http-8080
26 21/Jul/2009 11:37:04 org.apache.jk.common.ChannelSocket init
27 INFO: JK: ajp13 listening on /0.0.0.0:8009
28 21/Jul/2009 11:37:04 org.apache.jk.server.JkMain start
29 INFO: Jk running ID=0 time=0/94  config=null
30 21/Jul/2009 11:37:04 org.apache.catalina.startup.Catalina start
31 INFO: Server startup in 7493 ms
32 21/Jul/2009 11:37:05 org.apache.catalina.tribes.io.BufferPool
    getBufferPool
33 INFO: Created a buffer pool with max size:104857600 bytes of type:org.
    apache.catalina.tribes.io.BufferPool15Impl
34 21/Jul/2009 11:37:23 org.apache.catalina.ha.tcp.SimpleTcpCluster
    memberAdded
35 INFO: Replication member added:org.apache.catalina.tribes.membership.
    MemberImpl[tcp://worker2:jogo.ubi.pt:4000, worker2.jogo.ubi.pt, 4000,
     alive=18515,id={-84 -90 21 120 124 -101 75 -121 -89 19 -102 -104 69
    55 -105 58 }, payload={}, command={}, domain={}, ]
```

Este excerto mostra que a instância Tomcat *worker1* foi iniciada correctamente e descobriu automaticamente outra instância designada por *worker2* sendo adicionado como

membro do *cluster*. A adição do *worker2*[1] ao *cluster* corresponde ao apresentado na linha 34 e 35.

Configurada a replicação no *cluster* foi necessário tornar os objectos, a replicar, *serializable*. Neste ponto, as duas instâncias estão preparadas para replicar os dados relativos ao estado do jogo através de TCP. O exemplo que se segue representa a troca de mensagens entre os dois membros do *cluster*, onde se pode observar como é efectuada a criação de uma sessão para cada utilizador, bem como a adição de atributos e qual a instância que o cliente está ligado. Sendo que, neste caso o jogador "ana" tem o papel de "Mestre" no jogo 32 está ligado ao *worker1*.

Listing 3.3: Código para a replicação de estados.

```
1  21/Jul/2009 11:37:04 org.apache.catalina.core.ApplicationContext log
2  INFO: SessionListener: contextInitialized()
3  21/Jul/2009 11:37:19 org.apache.catalina.core.ApplicationContext log
4  INFO: SessionListener: sessionCreated('8340FEEEBEA70F404BF3E6479C204E74.
       worker1')
5  21/Jul/2009 11:37:20 org.apache.catalina.core.ApplicationContext log
6  INFO: SessionListener: attributeAdded('8340FEEEBEA70F404BF3E6479C204E74.
       worker1', 'nick', 'ana')
7  21/Jul/2009 11:37:20 org.apache.catalina.core.ApplicationContext log
8  INFO: SessionListener: attributeAdded('8340FEEEBEA70F404BF3E6479C204E74.
       worker1', 'jogo', '32')
9  21/Jul/2009 11:37:21 org.apache.catalina.core.ApplicationContext log
10 INFO: SessionListener: attributeAdded('8340FEEEBEA70F404BF3E6479C204E74.
       worker1', 'tipo', 'Mestre')
```

3.2.2 *Load Balancing*

Load balancing é o termo utilizado para descrever a tentativa de distribuição eficiente de uma aplicação, entre vários servidores, com base nos requisitos de processamento [28]. As técnicas de *load balancing* permitem aumentar a disponibilidade de um jogo, ao efectuar a distribuição dos pedidos entre os servidores do *cluster*. Esse aumento da disponibilidade pode ser conseguido porque permite repartir os utilizadores por diferentes servidores e

[1] O id correspondente ao *worker2* foi substituído pelo nome para facilitar a compreensão do ficheiro apresentado

também por tornar possível quer a adição quer a remoção de elementos do *cluster* de forma transparente. Para que a falha de um elemento do *cluster* seja transparente é necessário que ele esteja replicado.

Como já foi dito anteriormente na secção 3.2, para o jogo FT-D&D, foi implementado um *cluster* com dois servidores *web*. O redireccionamento dos clientes de um servidor para outro em caso de falha é feito usando um mecanismo de *load balancing*, tendo sido utilizado o *software* Apache Server 2.2 como o módulo designado de mod_jk. Este módulo utiliza o protocolo Apache Java Protocol (AJP)[2] para efectuar a ligação com os servidores *web* Tomcat. Foi definido um servidor como principal e outro como secundário, sendo os pedidos direccionados para o servidor principal exceptuando a situação em que este falha e passam a ser direccionados para o servidor secundário. O servidor secundário do *cluster* notícia que a instância primária falhou e assim deixa de enviar as mensagens com os dados do estado do jogo para o servidor em falha procedendo à sua remoção do *cluster*. Quando a instância primária recupera da falha é novamente adicionada ao *cluster*, começando a receber as mensagens com os dados do estado do jogo de todos os clientes e posteriormente todas as ligações voltam para ele. Desta forma é garantido que todos os jogadores, que constituem um jogo, estão ligados ao mesmo servidor. Para ser o servidor de *load balancer* a receber todos os pedidos dos clientes e efectuar o respectivo direccionamento para uma instância disponível, foi definido um Domain Name System (DNS) que retorna o IP do *load balancer*. Cada pedido dos clientes efectua o percurso apresentado na figura 3.8.

1. Pedido ao DNS local: o *browser* do utilizador tenta resolver o endereço IP do jogo a partir do seu nome, através de um pedido de pesquisa de DNS na rede, ao servidor de DNS local ao utilizador.

2. Pedido ao *authoritative* DNS: normalmente o servidor de DNS local ainda não tem o endereço IP do jogo em *cache* (só o terá a partir do primeiro pedido), por sua vez deve pedir ao servidor *authoritative* DNS do jogo o endereço IP do jogo que o utilizador deseja aceder.

3. O servidor *authoritative* DNS responde, com o IP do *load balancer*, ao servidor de DNS local

[2]O protocolo AJP é orientado a pacotes, baseado no protocolo TCP/IP. Fornece o canal de comunicação entre o Apache *web Server* e as instâncias Tomcat. AJP assegura uma boa performance ao reutilizar as ligações ao nível do TCP com o *container* Tomcat e assim reduz o *overhead* associado à abertura de um novo *socket* de ligação para cada pedido [33].

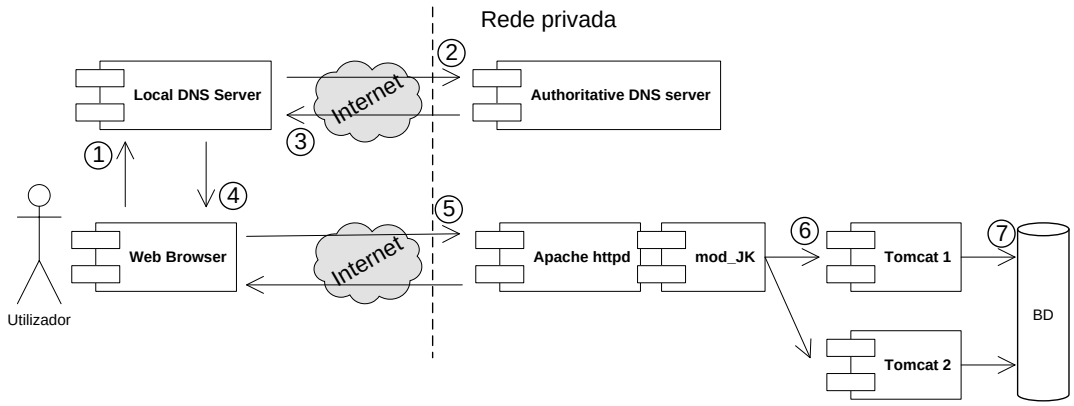


Figura 3.8: Etapas de um pedido através de um cluster Tomcat e Apache httpd.

4. Resposta ao DNS local: servidor de DNS local responde dando ao *browser* o endereço IP do FT-D&D.

5. Pedido Hypertext Transfer Protocol (HTTP): o *browser* faz um pedido HTTP para o endereço IP dado pelo DNS. Este pedido pode utilizar ligações HTTP *keep-alive* para eficiência da rede.

6. Os pedidos dos clientes são direccionados para os membros do *cluster* através do módulo "mod_jk".

7. O Tomcat envia um ou mais pedidos para os servidores de base de dados: o Tomcat pode depender de outros servidores de conteúdo dinâmico para criar a resposta que ele encaminha para o *browser*. Pode-se ligar a uma base de dados através de JDBC, ou JNDI para procurar outros objectos antes de ser capaz de construir o conteúdo dinâmico que corresponde à resposta.

Após a conclusão das etapas descritas acima, a direcção das setas inverte-se e a resposta de cada etapa é feita na ordem contrária à dos pedidos, a resposta é enviada através de uma ligação de rede já aberta.

Para a configuração do módulo mod_jk foi necessário criar um ficheiro chamado de *workers.properties* onde é realizada a descrição das características dos elementos do *cluster*, isto é, dos *workers*. Cada *worker* corresponde a um servidor *web* Tomcat, sendo a sua descrição composta por um nome, endereço IP do servidor Tomcat, porto e finalmente o

protocolo associado. No caso do jogo FT-D&D foi definido um *worker* como principal e outro como secundário. A definição das propriedades de cada membro é efectuada como representado de seguida:

worker.worker1.port = 8009

worker.worker1.host = IP do membro principal do cluster

worker.worker1.type = ajp13

worker.worker1.lbfactor = 1

Para reencaminhar todos os pedidos para o *worker* secundário, no caso do primário falhar (*worker1*) foi introduzida na configuração a seguinte linha:

worker.worker1.redirect = worker2

A definição das propriedades do *worker2* é igual às apresentadas para o *worker1* com excepção da última linha que é substituída pela seguinte:

worker.worker2.activation = disable

Esta linha corresponde à desactivação de todos os pedidos, excepto se ocorrer a falha do *worker1*, garantido assim a tolerância a falhas. Para que o *load balancing* funcione foi necessário definir em cada instância do *cluster* o nome atribuído no ficheiro *worker.properties*. Assim foi introduzida a seguinte linha na configuração do servidor *web* principal:

Listing 3.4: Código a introduzir nas instâncias Tomcat.

```
1 <Engine name="Catalina" defaultHost="localhost" jvmRoute="worker1">
```

No servidor *web* secundário foi introduzida uma linha semelhante à anterior, mas com "jvmRoute = worker2".

Sendo este um jogo muito simples não é efectuado nenhum tipo de particionamento, isto é, todos os jogadores ligam-se ao mesmo servidor.

Aquando da configuração do *load balancer* é possível definir a afinidade entre sessões, ou seja, é enviado um *cookie* que permite examinar a sessão HTTP e o jvmRoute (verificando

a que instância do *cluster* o cliente estava ligado). Se for utilizada a afinidade entre sessões todos os pedidos de um cliente são respondidos pela mesma instância Tomcat. No *load balancing* implementado para o jogo FT-D&D não foi utilizada a afinidade de sessão

3.2.3 Network Time Protocol

A sincronização dos relógios dos computadores é muito importante nas redes de computador modernas. A precisão e a sincronização do tempo é um aspecto crítico em muitas aplicações, particularmente em transacções em que a sequência das operações é importante. A sincronização dos relógios pode seguir um dos seguintes modelos [31]:

- Cliente-Servidor: corresponde a uma associação permanente e a forma mais comum de configuração. Um computador desempenha o papel de cliente, solicitando informações sobre o tempo a um servidor. O servidor reponde às solicitações do cliente com informações sobre o tempo sendo este o modelo implementado para os servidores *web* do jogo FT-D&D.

- Modo simétrico: Dois ou mais dispositivos NTP podem ser configurados como *peers*, para que tanto possam ir buscar o tempo como fornecê-lo, garantindo redundância mútua. Esta configuração faz sentido para dispositivos na mesma camada, configurados também como clientes de um ou mais servidores.

- *Broadcast* ou *Multicast*: O NTP pode utilizar pacotes do tipo *broadcast* ou *multicast* para enviar ou receber informações do tempo. Este tipo de configuração pode ser vantajosa no caso de redes locais com poucos servidores, fornecendo o tempo a uma grande quantidade de clientes.

A figura 3.9 representa o esquema implementado para a sincronização dos relógios no *cluster* de servidores *web*. Esta sincronização permitiu que o envio das mensagens relativas aos estados do jogo fosse realizado pela ordem correcta.

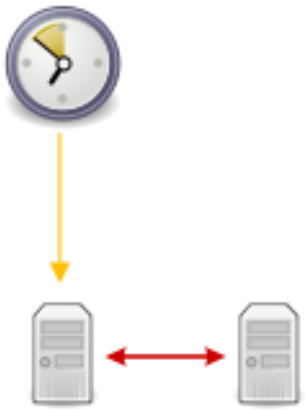


Figura 3.9: Sincronização dos relógios segundo o NTP.

3.3 Arquitectura do Jogo FT-D&D

O jogo FT-D&D é um jogo com arquitectura multi-servidor (*cluster* de servidores) e é baseado no *browser*, com persistência de dados.

Todos os dados sobre o estado do jogo são armazenados numa base de dados designada por "jogo". Por uma questão de segurança os dados sobre os utilizadores autorizados a aceder ao jogo são armazenados numa base de dados distinta, chamada de *"login"*. Em cada servidor de base de dados estão presentes as bases de dados "jogo" e *"login"*.

A arquitectura tolerante a falhas implementada para este jogo corresponde à apresentada na figura 3.10

Esta arquitectura contém dois pontos críticos, o servidor de DNS e o servidor de *load balancing* que, sendo únicos, em caso de falha de um deles todo o jogo é comprometido. No capítulo 5, propomos uma arquitectura para ultrapassar estes pontos críticos.

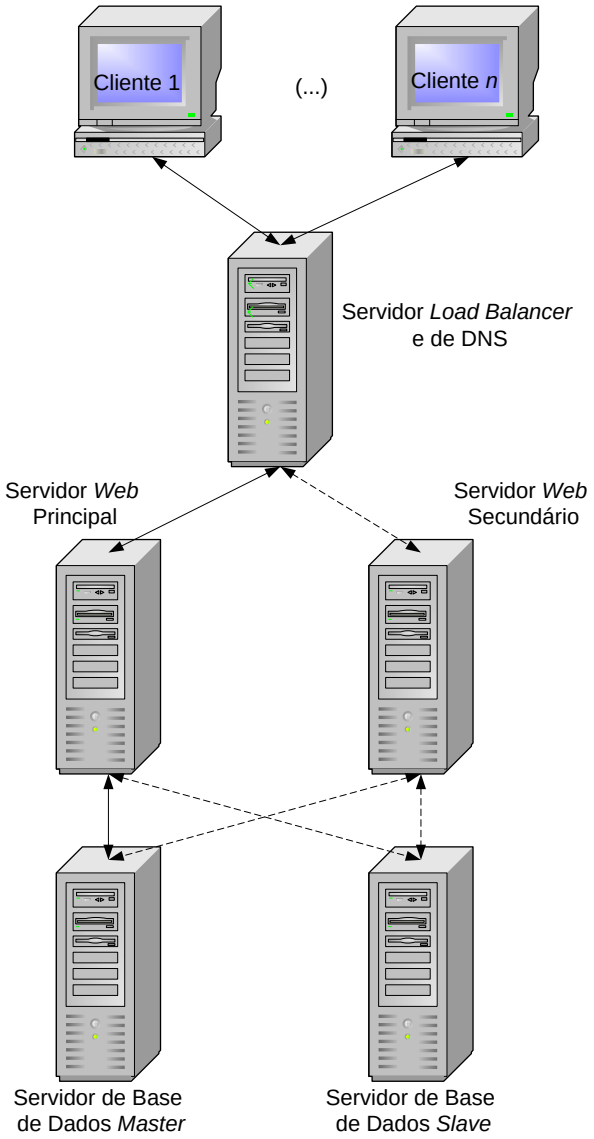


Figura 3.10: Arquitectura implementada para o jogo FT-D&D.

Capítulo 4

Avaliação Experimental

Para percebermos se as técnicas de tolerância a falhas possuem ou não um impacto significativo no desempenho do jogo, recorremos a testes de carga, sendo executados até sessenta jogos em simultâneo. Na secção 4.1 deste capítulo é apresentado o *software* utilizado para os testes de carga, o JMeter, os passos necessários para a simulação de 1, 15, 30 e 60 jogos em simultâneo o que corresponde a 5, 75, 150 e 300 utilizadores respectivamente e as características dos computadores utilizados. Na secção 4.2 é apresentada a arquitectura de cada simulação e apresentados os resultados obtidos em termos de *throughput* sendo realizada uma comparação entre os resultados das simulações efectuadas.

4.1 Ambiente Experimental

Para medir o custo das técnicas de tolerância a falhas implementadas no jogo *on-line* FT-D&D optou-se por simular vários jogos em simultâneo. Para esta simulação recorreu-se a um *software* para a realização de testes de carga, o JMeter. Esta ferramenta permite medir o desempenho de aplicações *web*. É uma ferramenta *multithreading* que permite uma amostragem simultânea de diferentes funções, separadas em *thread groups* [43]. O JMeter é um *software open-source*, mantido pelo grupo Jakarta Apache, que possui a capacidade de executar planos de teste, configurados através da sua ferramenta gráfica.

Para a simulação dos jogos no JMeter, é necessário guardar todas as sequências de um jogo. Para isso, foi utilizado um *proxy* disponibilizado pelo próprio JMeter, que capturou todos os pedidos realizados ao servidor do jogo *on-line* FT-D&D. Para a simulação de um jogo foram usados cinco *browser's* devidamente configurados para a utilização de um

proxy, em que cada um representa um jogador. Cada jogo foi guardado no JMeter, em grupos distintos, o que tornou possível a simulação de vários jogos em simultâneo. Foram medidos os tempos de execução para 1, 15, 30 e 60 jogos em simultâneo, isto é, 5, 75, 150 e 300 utilizadores. Na figura 4.1 está presente o plano de teste utilizado na simulação de quinze jogos em simultâneo.

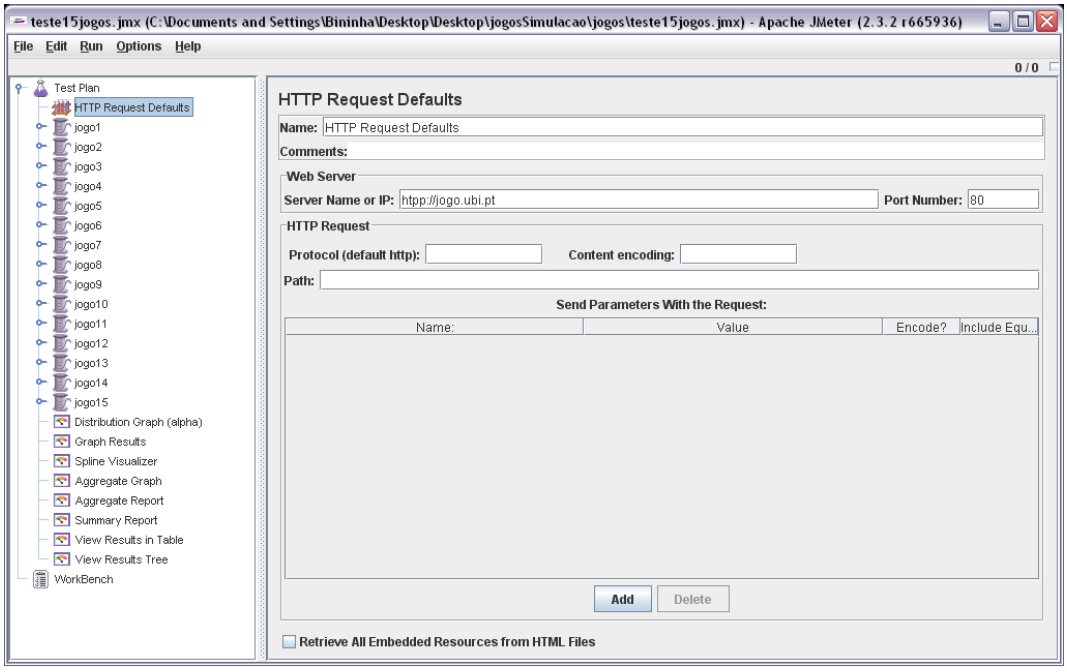


Figura 4.1: Representação do ambiente gráfico do JMeter com 15 jogos no *Test Plan*.

O JMeter também permite que os testes de carga sejam efectuados de forma distribuída, ou seja, permite que seja incluído o conceito de *master* e *slave* na sua arquitectura. Neste caso o *master* corresponde ao sistema que está a executar o "JMeter GUI", e que controla os testes. O *slave* por sua vez corresponde ao sistema que está a executar o "jmeter-server", que recebe os comandos do "GUI" e envia os pedidos para o sistema a testar. O modo distribuído permitiria melhorar os resultados uma vez que não é o *master* o responsável por executar todas as tarefas mas, implicaria a utilização de mais máquinas.

Para os vários cenários simulados, foram adicionadas máquinas, com o intuito de simular o maior número possível de utilizadores em simultâneo. Para a simulação de sessenta jogos em simultâneo, onde existe replicação de dados e de estados, foram utilizados:

- 2 servidores *web* Tomcat;

- 2 servidores de bases de dados MySQL;

- 1 servidor com o *load balancer* e DNS;

- 4 servidores com o JMeter;

Todos os servidores possuem características iguais, ou seja, processador Intel Pentium D a 3GHZ e 1Gb de RAM, sistema operativo "Windows Server 2003", com uma rede *directed broadcast* a 100 Mbits.

Todas as simulações passam obrigatoriamente pelo servidor de DNS e de *load balancer* que reencaminha o pedido para o servidor *web*. Foram necessários quatro servidores com a ferramenta JMeter para simular 60 jogos em simultâneo, devido a limitações de memória na máquina virtual do Java. Foi necessário distribuir por cada JMeter, 15 jogos iguais na sequência de jogadas, mas com diferente identificador. Se tivéssemos utilizado a forma de simulação distribuída teriam sido necessárias oito máquinas. Esse conjunto de 15 jogos tentou ser uma amostra que incluísse jogos com diferentes níveis de dificuldade. Um terço corresponde a jogos de baixo nível de complexidade, um terço de nível médio e os restantes de nível elevado.

4.2 Resultados

Foram simulados sete cenários distintos no jogo FT-D&D, com o intuito de medir o custo associado à inclusão de tolerância a falhas, nomeadamente a replicação de dados e de estados, no jogo e a sua recuperação quando ocorre uma falha. Dos sete cenários simulados, foram medidos os tempos para 1, 15, 30 e 60 jogos em simultâneo o que corresponde a 5, 75, 150 e 300 utilizadores respectivamente. A descrição das simulações efectuadas e a apresentação da arquitectura física utilizada é efectuada de seguida:

1. Sem replicação - não existe replicação de dados nem de estados. A arquitectura utilizada para esta simulação corresponde à apresentada na figura 4.2;

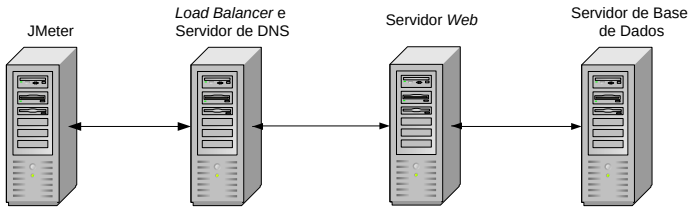


Figura 4.2: Arquitectura para o cenário em que não existe replicação.

2. Replicação de estados - existe apenas replicação de estados. A replicação é efectuada entre duas instâncias do *cluster* sendo também utilizada uma máquina com o DNS e o "load balancer". A representação desta arquitectura corresponde à apresentada na figura 4.3;

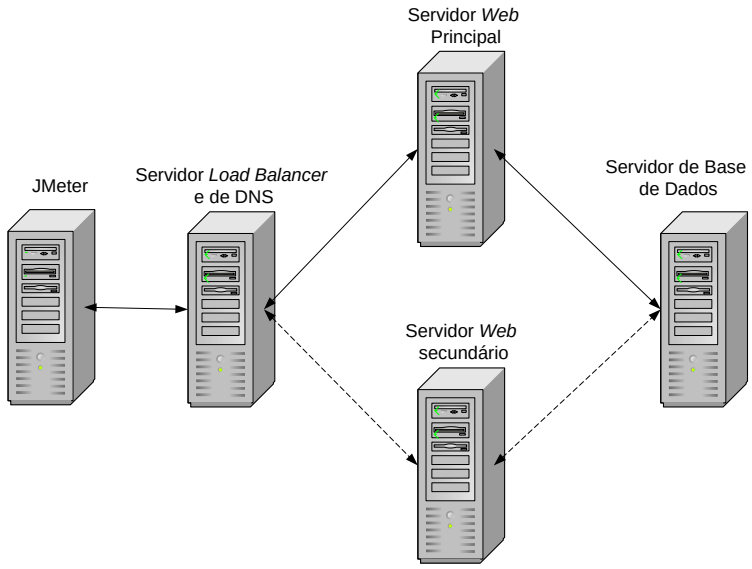


Figura 4.3: Arquitectura para o cenário em que existe apenas replicação de estados do jogo.

3. Replicação de dados - o servidor de base de dados principal (*master*) é configurado para efectuar a replicação, um segundo servidor desempenha o papel de secundário (*slave*). A arquitectura desta simulação corresponde à apresentada em 4.4;

4. Replicação de dados e de estados - corresponde à junção da situação 2 e 3 como apresentado na figura 4.5.

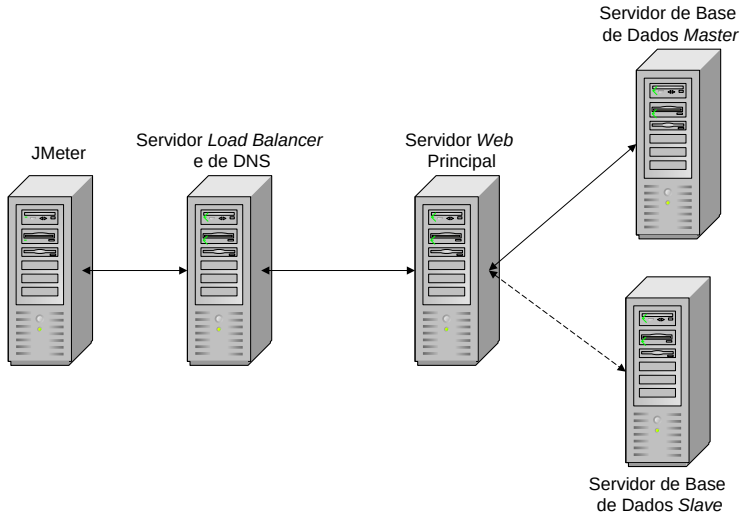


Figura 4.4: Arquitectura para o cenário em que existe apenas replicação de dados do jogo.

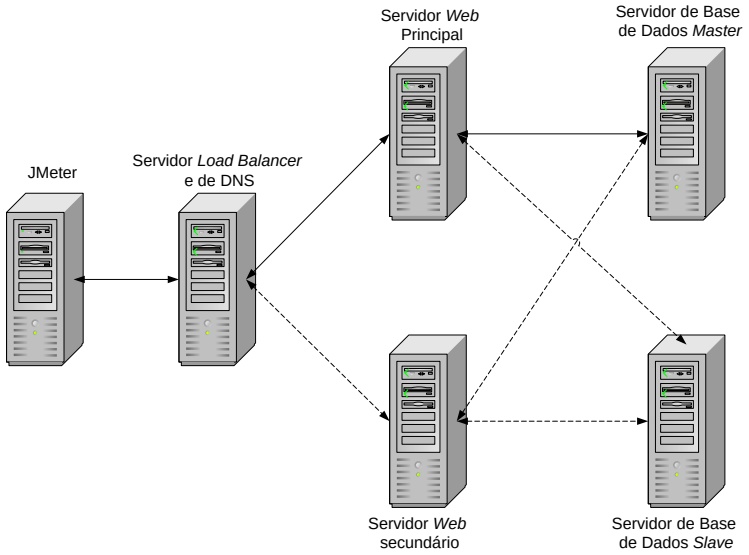


Figura 4.5: Arquitectura para o cenário com replicação de dados e de estados do jogo.

5. Falha do servidor *web* - é igual à situação 2, onde existe apenas replicação de estados, faz-se falhar o servidor *web* principal (“matando” o processo). Após a falha, os pedidos são redireccionados para o servidor secundário que possui informação sobre as sessões activas. Esta simulação corresponde à apresentada na figura 4.6.

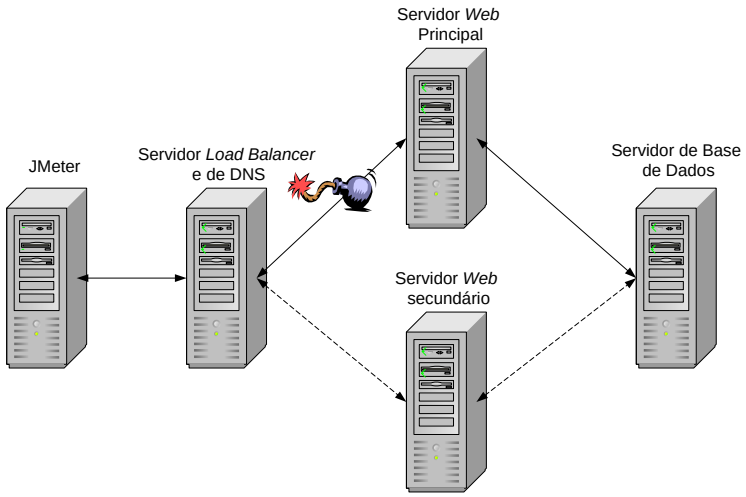


Figura 4.6: Arquitectura para o cenário que contempla a replicação de estados e a falha num dos servidores *web*.

6. Falha do servidor de base de dados - consiste na situação 3, onde existe apenas replicação de dados e é provocada uma falha no servidor de base de dados principal, como apresentado na figura 4.7.

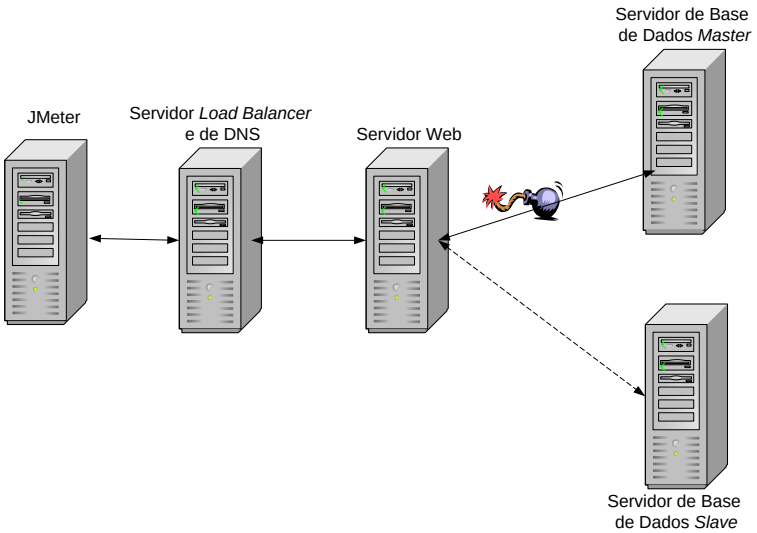


Figura 4.7: Arquitectura para o cenário que contempla a replicação de dados e a falha num dos servidores BD.

7. Falha do servidor de base de dados e do servidor *web* - Nesta situação existe replicação tanto de dados como de estados. É provocada uma falha no servidor *web* principal e posteriormente no servidor de base de dados principal (*master*). A figura 4.8 mostra a arquitectura utilizada nesta simulação.

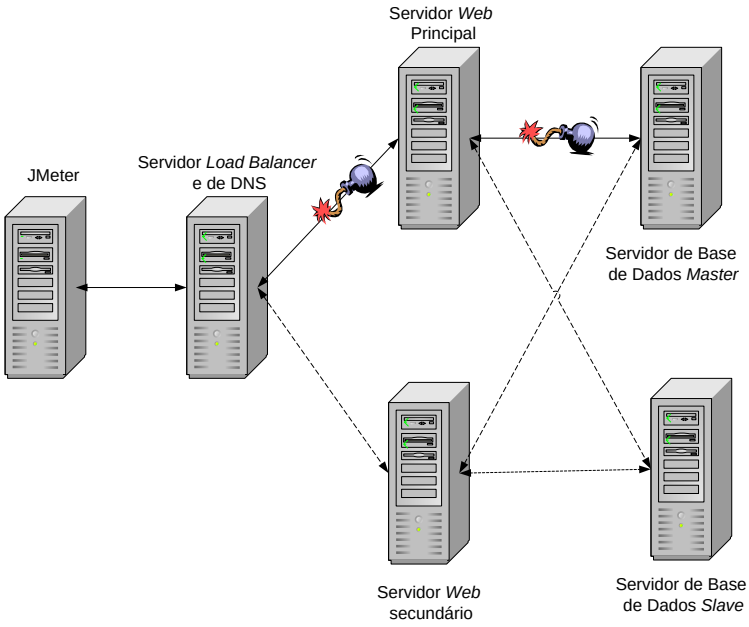


Figura 4.8: Arquitectura para o cenário que contempla a replicação quer de dados, quer de estados e a falha dos servidores *web* e BD.

4.2.1 Simulações Sem Falhas

Nesta secção apresentamos e discutimos os resultados obtidos, nas várias simulações efectuadas, não contemplando a ocorrência de falhas, o que corresponde às situações 1, 2, 3 e 4 apresentadas na secção 4.2.

Na figura 4.9 é apresentado um gráfico com o *throughput* (valor médio do número de pedidos processados por segundo) obtido para 1, 15, 30 e 60 jogos, construído a partir dos dados da tabela 4.1.

Como se pode observar no gráfico da figura 4.9 as linhas que representam as simulações "sem replicação" e "replicação de estados", que correspondem aos *throughput* mais elevados, têm um comportamento muito semelhante e tendem a aproximar-se cada vez mais. O que significa que o desempenho do jogo "sem replicação" e com "replicação de estados" irá

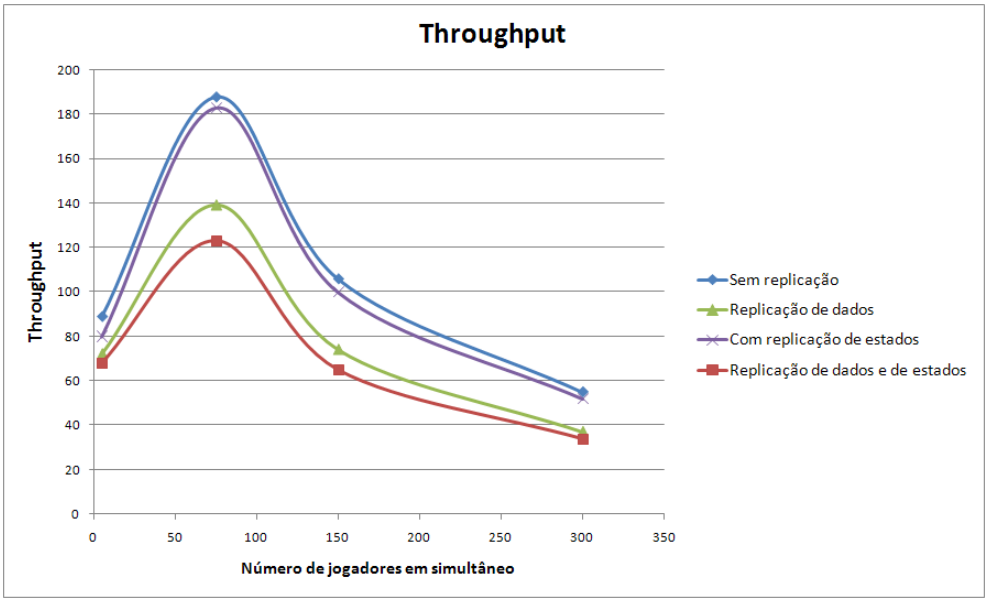


Figura 4.9: *Throughput* para 1, 15, 30 e 60 jogos em simultâneo.

	Throughput			
	1 Jogo	15 Jogos	30 Jogos	60 Jogos
Sem replicação	89	188	106	55
Replicação de estados	80	183	100	52
Replicação de dados	72	139	74	37
Replicação de dados e de estados	68	123	65	34

Tabela 4.1: Apresentação dos valores de *throughput* obtidos para as situações sem ocorrência de falhas.

ser cada vez mais semelhante. A diferença de pedidos que o servidor é capaz de responder por segundo vai diminuindo ao longo das simulações, como se pode ver nos resultados apresentados na tabela 4.1).

As linhas do gráfico da figura 4.9 que representam as simulações "Replicação de estados" e "Replicação de dados e de estados", que correspondem às linhas com menor *throughput*, possuem um comportamento semelhante ao descrito anteriormente. Para estes dois casos a diferença entre os pedidos que o servidor responde por segundo tende também a diminuir.

Como pode observar-se em todas as simulações verifica-se uma aproximação entre as

várias linhas do *throughput* com o aumento do número de utilizadores o que se deve ao facto das ligações estabelecidas entre os servidores estarem a ser reaproveitados pelo servidor de *load balancing*.

Apesar de serem apresentados resultados para a simulação de um jogo não foram estabelecidas comparações com esta simulação por considerarmos que os valores obtidos não são significativos.

De seguida é apresentada a tabela 4.2 e o respectivo gráfico na figura 4.10 onde é possível perceber qual o custo associado às técnicas de tolerância a falhas implementadas.

	Throughput em %			
	1 Jogo	15 Jogos	30 Jogos	60 Jogos
Sem replicação	100	100	100	100
Replicação de estados	90	97	94	95
Replicação de dados	81	74	70	67
Replicação de dados e de estados	76	65	61	62

Tabela 4.2: Valores de *throughput* obtidos em percentagem para as simulações sem falhas.

Observamos que a capacidade de resposta para as situações sem replicação e com replicação de estados não é muito diferente, isto é, o custo da replicação do estado do processo é pouco significativo. A capacidade de resposta do servidor, *throughput*, com replicação de processos passa para 95% quando consideramos 60 jogos (sendo de 97% para 15 jogos e 94% para 30 jogos). O custo da replicação de dados é mais significativo. Com 15 jogos obtém-se 74% de desempenho, para 30 jogos 70%, e finalmente para os 60 jogos o desempenho é 67% do obtido sem replicação. Finalmente na replicação de dados e estados, como seria de esperar, existe um acumular da perda de desempenho. Para 60 jogos a replicação de dados e estados tem um desempenho de 62% (isto é um custo de 38%). Verifica-se que com o aumento do número de jogos há uma perda de desempenho mas essa perda vai sendo cada vez mais atenuada, isto é, a distância entre o *throughput* para o jogo sem replicação e o *throughput* obtido com replicação vai sendo cada vez menor. O custo da replicação vai sendo repartido pelos vários jogadores como pode observar-se pela aproximação das linhas do gráfico da figura 4.9. Podemos concluir que o aumento do custo se deve essencialmente ao aumento do número de pedidos.

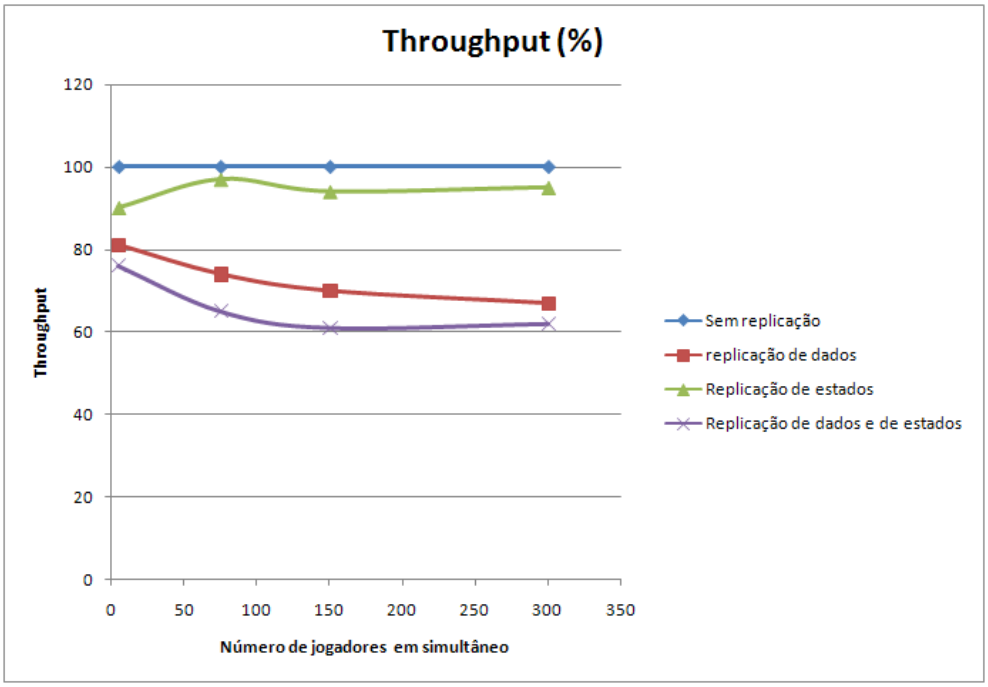


Figura 4.10: *Throughput* em percentagem para 1, 15, 30 e 60 jogos em simultâneo na ausência de falhas.

4.2.2 Simulações de Falhas

Nesta secção serão analisados os resultados obtidos para as simulações que contemplam situações de falha quer do servidor de base de dados quer do servidor *web*. Para as situações 5, 6 e 7 apresentadas anteriormente na secção 4.2, os tempos apurados incluem o tempo do redireccionamento para os servidores secundários o que assegura a continuidade do jogo. Não se incluem os tempos para recuperar os processos que falharam e a sua reintegração na arquitectura do jogo por estes não terem impacto significativo no desempenho do jogo.

Mais uma vez os resultados obtidos para um jogo serão desprezados por os considerarmos pouco significativos.

Ao observar o gráfico 4.11, que foi construído a partir da tabela4.3, podemos constatar de imediato, através das linhas que definem o desempenho do jogo, que a "Falha no servidor *web*" é a que apresenta melhores resultados, sendo que, as simulações "Falha no servidor Base de Dados (BD)" e "Falha no servidor *web* e BD" apresentam um comportamento coincidente excepto para 15 e 30 jogos em simultâneo.

	Throughput			
	1 Jogo	15 Jogos	30 Jogos	60 Jogos
Sem Replicação	89	188	106	55
Replicação de estados e falha do servidor *web*	54	87	63	43
Replicação de dados e falha do servidor BD	4	49	37	25
Replicação de dados e de estados e falha servidor *web* e BD	4	48	35	25

Tabela 4.3: Valores de *throughput* obtidos para as situações em que são contempladas falhas.

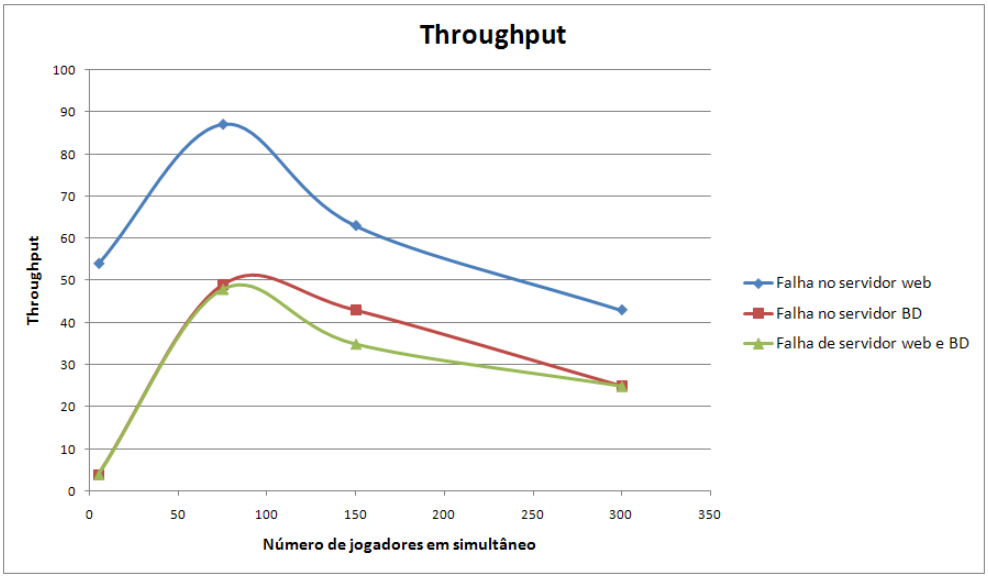


Figura 4.11: *Throughput* para 1, 15, 30 e 60 jogos em simultâneo para situações de falha.

Comparando o número de pedidos respondidos por segundo para a "Falha no servidor *web*" com a "Falha no servidor BD" podemos afirmar que o *throughput* tende a igualar-se à medida que são adicionados mais jogos à simulação.

De seguida é apresentada a tabela 4.4 e o respectivo gráfico na figura 4.12 com o intuito de analisar em termos de percentagem o custo do desempenho associado ao jogo.

	Throughput em %			
	1 Jogo	15 Jogos	30 Jogos	60 Jogos
Sem Replicação	100	100	100	100
Replicação de estados e falha do servidor *web*	61	46	59	78
Replicação de dados e falha do servidor BD	4	26	35	45
Replicação de dados e de estados e falha servidor *web* e BD	4	26	33	45

Tabela 4.4: Valores de *throughput* obtidos em percentagem para as situações com falha.

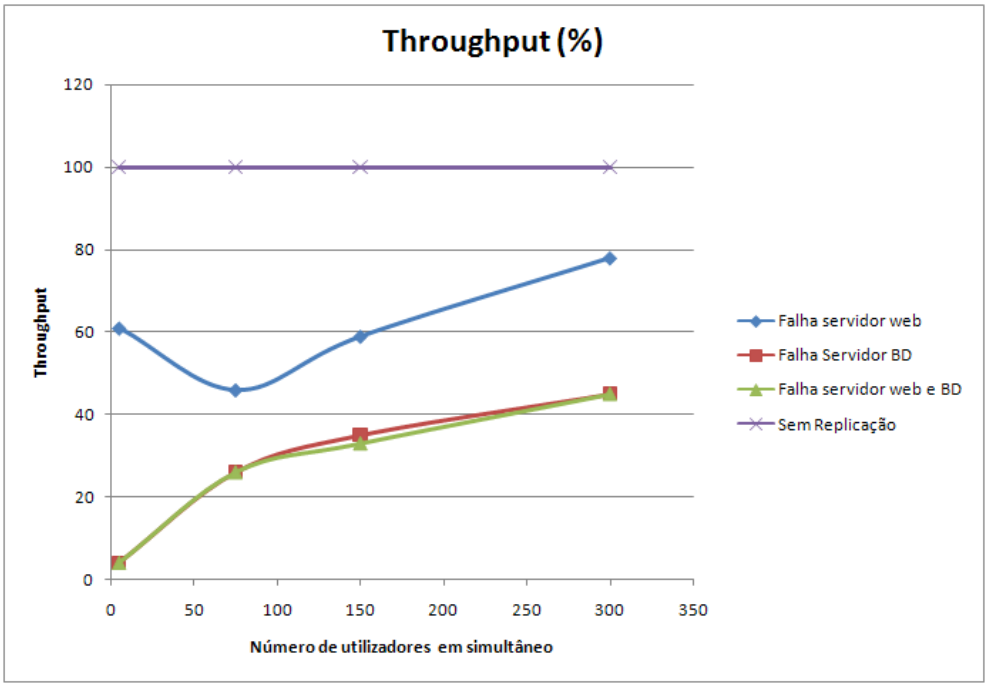


Figura 4.12: *Throughput* em percentagem para 1, 15, 30 e 60 jogos em simultâneo para situações de falha.

No caso de ocorrência de falha de dados e estados, o que corresponde à simulação "Falha de servidor *web* e BD", no período de tempo considerado para a simulação, o

desempenho passa para 45% do desempenho obtido quando não existe qualquer mecanismo de replicação, no caso dos 60 jogos. É de notar em primeiro lugar que a falha do servidor web tem um impacto praticamente nulo. O tempo gasto na mudança de servidor *web* parece ser compensado pelo tempo poupado ao não haver comunicação dos dados para a réplica. A falha da base de dados tem um impacto significativo, o que talvez se deva ao facto da detecção da falha e mudança de servidor de base de dados estar a ser feita ao nível do jogo. Para verificar esta hipótese será necessário implementar a replicação de dados através dos mecanismos de *clustering* dos servidores de bases de dados. As simulações efectuadas tiveram uma duração de alguns minutos, correspondendo ao completar dos jogos considerados. A probabilidade de ocorrer uma falha num intervalo de tempo reduzido é praticamente nula. Assim podemos considerar que o custo da tolerância a falhas implementada corresponde essencialmente ao custo da replicação.

Capítulo 5

Arquitectura Proposta

Neste capítulo vamos analisar alguns pontos críticos da arquitectura implementada e propor uma solução para os ultrapassar, estando organizado da seguinte forma: na secção 5.1 explicamos o porquê de ser necessário procurar alternativas mais eficientes na troca de servidores de base de dados e propomos uma solução que nos parece mais eficaz. De seguida, na secção 5.2 enunciamos os pontos críticos da actual implementação e apresentamos uma arquitectura sem os pontos críticos identificados. Finalmente, na secção 5.3 apresentamos a arquitectura completa, ou seja, uma arquitectura mais fiável capaz de oferecer replicação de dados e de estados e de redireccionar automaticamente todos os pedidos de um servidor em falha para um disponível.

5.1 Arquitectura Proposta para a Replicação de Dados

Os actuais jogos *on-line* possuem persistência de dados, daí surge a necessidade de serem adaptados e criados novos métodos de replicação de dados que sejam adequados às necessidades destes jogos.

A implementação de replicação de dados no jogo FT-D&D mostrou a necessidade de procura de melhores soluções para jogos *on-line* uma vez que possui um elevado custo. A troca de servidor de base de dados apresentou atrasos significativos o que leva à necessidade de serem implementados métodos de *load balancing* mais eficazes e que satisfaçam melhor os requisitos dos MMOG.

5.1.1 *Load Balancing* nas Base de Dados

A ideia básica por trás do *load balancing* consiste na partilha do trabalho o mais uniformemente possível por um conjunto de servidores (*cluster*). A maneira mais comum para o fazer é colocar o *load balancer* à frente dos servidores. Então o *load balancer* direcciona os pedidos para os servidores menos ocupados de entre os disponíveis. Poder-se-ia ainda proceder ao direccionamento dos pedidos apenas de leitura pelos *slaves* e os de escrita para o *master*.

A figura 5.1 mostra uma implementação típica de *load balancing*, com um *load balancer* para o tráfego HTTP e outro para o tráfego MySQL. Mostra ainda a distribuição dos pedidos de leitura pelos *slaves* e direccionamento dos pedidos de actualização para o *master*.

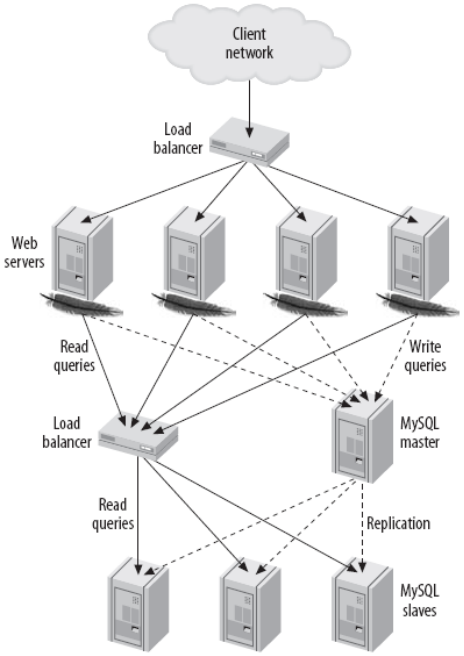


Figura 5.1: Arquitectura típica de *load balancing* com distribuição dos pedidos de leitura e escrita [45].

São várias as formas que existem para a implementação de *load balancer's*, estas formas podem ir desde a criação de um DNS, passando pela utilização de *software* próprio até à utilização do MySQL *proxy* entre outras opções.

Com o objectivo de implementar uma replicação de dados mais eficaz, na presença de falhas, propomos a implementação de um *cluster* MySQL e como *load balancer* a utilização do MySQl *proxy*, como representado na figura 5.2. O ponto essencial desta arquitectura

corresponde à inclusão de um servidor de *load balancing*. O *load balancer* deverá estar configurado para reencaminhar todos os pedidos, tal como implementado, para o servidor *master* e no caso de este falhar passam a ser direccionados para os servidores *slave* que será transformado em *master*, ou seja, deverá continuar a seguir a topologia *master-master* e modo activo-passivo como implementado para este jogo.

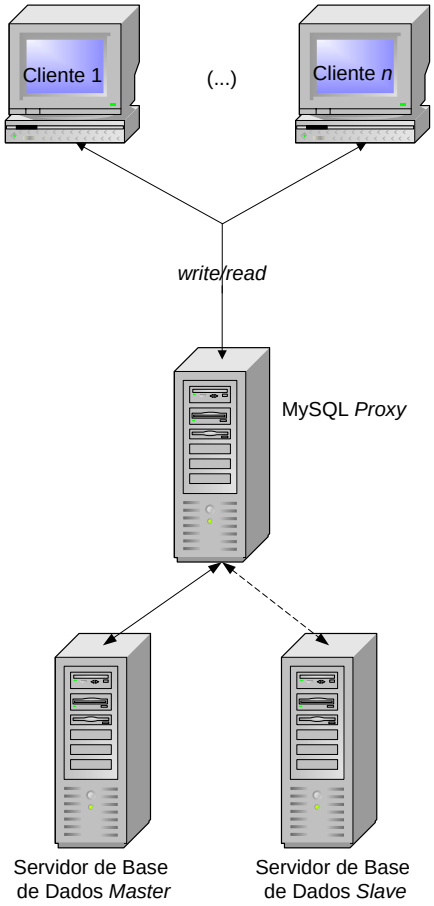


Figura 5.2: *Cluster* de base de dados.

A arquitectura apresentada anteriormente corresponde a uma arquitectura muito simples de um *cluster* de base de dados. No entanto, apresenta um ponto crítico, o servidor de *load balancing*, caso ocorra uma falha neste ponto deixa de ser possível aceder aos dados o que se traduz numa falha grave no jogo. Assim sendo, a arquitectura presente na figura 5.3 apresenta-se como uma solução mais eficaz que a apresentada na figura 5.2 uma vez que foi removido o ponto crítico existente através de um servidor de *load balancing* alternativo.

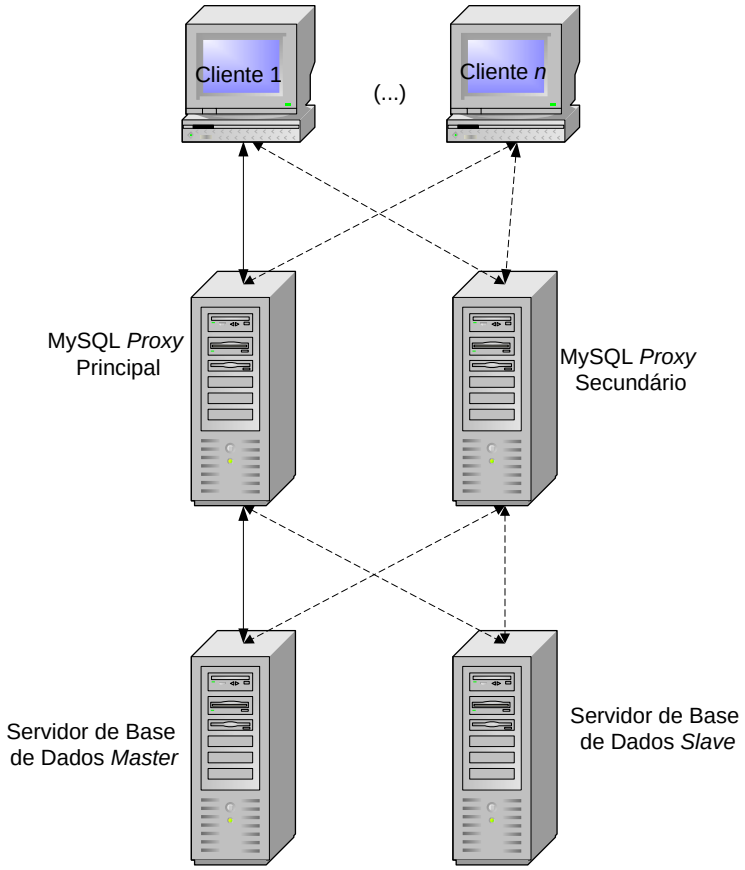


Figura 5.3: *Clusterização* da base dados com eliminação do ponto crítico.

5.2 Arquitectura Proposta para a Replicação de Estados

A replicação de estados de um jogo é fundamental para permitir que uma falha seja transparente para os utilizadores, e assim, responde a um dos principais requisitos dos MMOG, a disponibilidade do jogo.

A arquitectura implementada para a replicação de estados não satisfaz todos os requisitos de um MMOG na medida em que possui dois pontos críticos detectados, sendo eles o servidor de DNS e o servidor de *load balancing* que estão alojados no mesmo servidor físico como apresentado na figura 5.4.

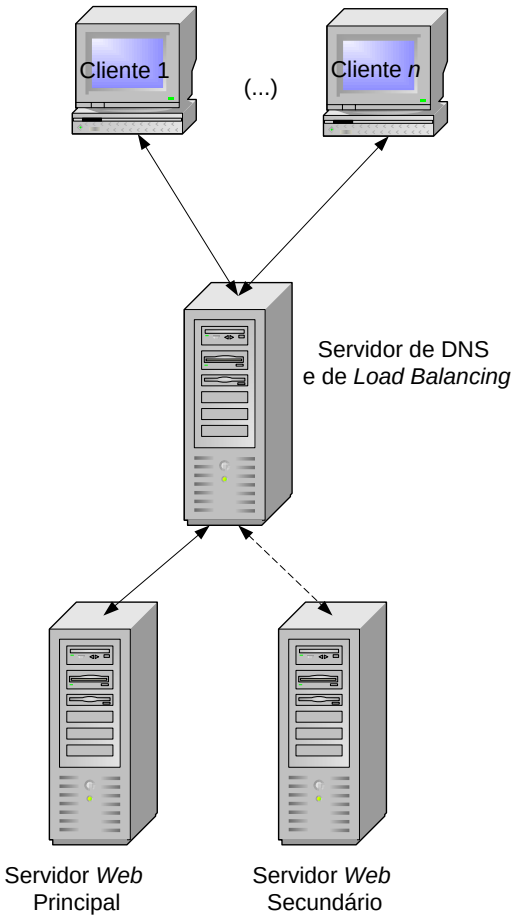


Figura 5.4: Arquitectura implementada para a replicação de estados do jogo FT-D&D.

Para a remoção do ponto crítico apresentado na figura 5.4 propomos que seja adicionado outro servidor de DNS e de *load balancing* de forma redundante. Na figura 5.5 apresentamos a arquitectura necessária para que a replicação de estados deixe de apresentar o ponto crítico identificado, isto é, é feita a introdução de uma réplica do servidor de DNS e de *load balancing*.

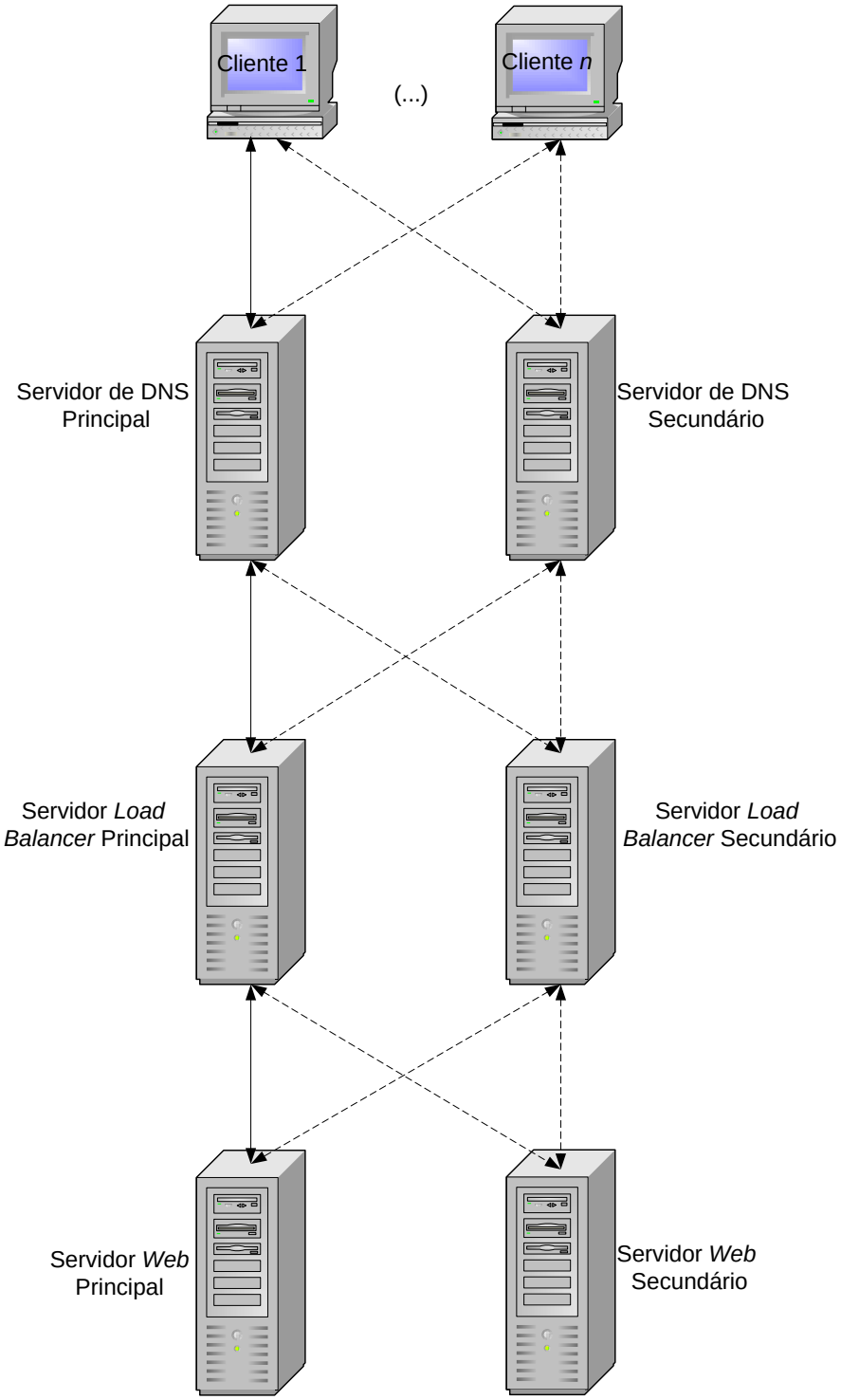


Figura 5.5: Arquitectura sugerida para a replicação de estados do jogo FT-D&D.

5.3 Arquitectura Final

A junção das arquitecturas sem pontos críticos, que corresponde às apresentadas nas figuras 5.3 e 5.5 resulta na arquitectura apresentada na figura 5.6 sendo esta a arquitectura proposta, cuja avaliação em termos de impacto no desempenho será avaliada como trabalho futuro. É de notar que ao eliminar pontos críticos da solução implementada passamos de uma arquitectura com cinco máquinas, ver figura 5.4, para uma arquitectura com dez máquinas.

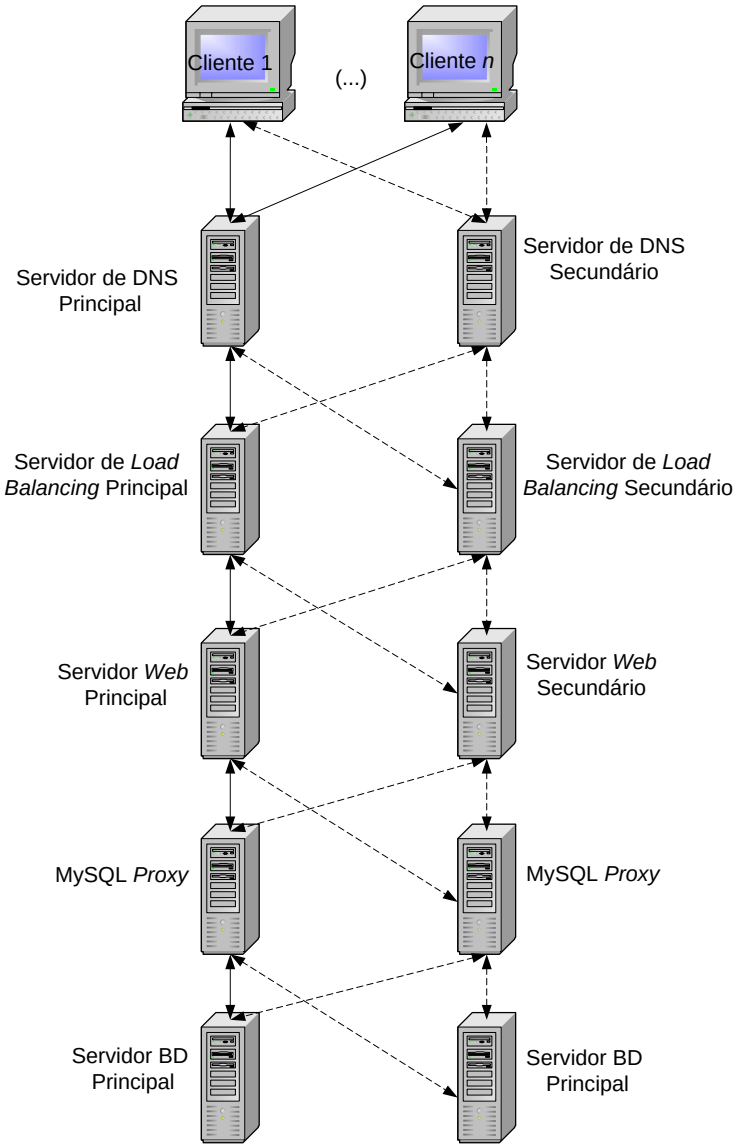


Figura 5.6: Arquitectura tolerante a falhas proposta para o jogo FT-D&D.

Capítulo 6

Conclusões

Nesta tese apresentamos um estudo do impacto da introdução de mecanismos de tolerância a falhas, como a replicação de dados e processos em jogos *on-line*. Foi desenvolvida e estudada uma versão tolerante a falhas do jogo D&D a que chamamos FT-D&D. Trata-se de um jogo baseado no *browser* com persistência de dados.

As técnicas de tolerância a falhas implementadas neste jogo foram a replicação de dados e de estados. Estas técnicas permitem assegurar a continuidade de um jogo mesmo na presença de falhas. Ambos os métodos implementados são capazes de tornar uma falha transparente para os utilizadores, como é esperado que aconteça em qualquer sistema tolerante a falhas. Poderá ocorrer uma situação de excepção quando a falha do servidor de base de dados acontece após uma actualização, mas imediatamente antes da propagação para o servidor de *backup*. Neste caso, a falha poderá não ser recuperável. A situação análoga poderá ocorrer para o caso do servidor *web*. A probabilidade destas situações ocorrerem, que supomos ser reduzida por corresponder a um passo de curta duração, não foi quantificada pois isso passaria por determinar a taxa de cobertura dos mecanismos de tolerância a falhas implementados através, por exemplo, de injecção de falhas.

Para a replicação de dados foram utilizados dois servidores de base de dados MySQL com a topologia *master-master* em modo activo-passivo seguindo o protocolo de replicação *lazy primary copy*. O *load balancing* entre os servidores de base de dados é realizado no próprio jogo, que detecta uma falha e procede à mudança de servidor. Essa decisão já tinha sido tomada no desenvolvimento da primeira versão do jogo e poderá ser uma das causas do elevado *overhead* introduzido pela replicação de dados.

A replicação de estados foi implementada num *cluster* de servidores *web* Tomcat e a

distribuição de pedidos realizada através de um servidor de *load balancing* próprio. Foi colocado o servidor de *load balancing* à frente dos servidores *web* de forma a distribuir os pedidos entre os membros do *cluster*. Foi definido que os pedidos seriam direccionados para o servidor de *load balancing* e só depois reencaminhados para os membros do *cluster* através da criação de um DNS que devolve o endereço IP do *load balancer*. O balanceamento efectuado entre os membros do *cluster* é semelhante ao realizado entre os servidores de base de dados, ou seja, todos os pedidos são enviados para o servidor *web* principal e só no caso de este falhar é que os pedidos são direccionados para o servidor *web* secundário. No entanto quando o servidor principal recupera da falha e recebe todos os dados relativos ao estado do jogo todos os pedidos voltam a ser direccionados para ele.

Através dos resultados obtidos com os testes de carga realizados para os vários cenários, com e sem a simulação de falhas, concluímos que o impacto da replicação de estados é diminuto causando um decréscimo no desempenho de cerca de 5%. A replicação da base de dados tem um impacto mais significativo passando a capacidade de resposta do servidor para 67%, isto é, um decréscimo de 33%. Com o aumento do número de jogadores o impacto da replicação é cada vez menor, o que significa que o principal factor para a diminuição do desempenho é o aumento do número de jogadores.

Tendo em conta que neste trabalho a detecção de uma falha e a mudança de servidor de base de dados está a ser realizada pelo próprio jogo, é necessário implementar outro método de *load balancing*, como o proposto no capítulo 5, para percebermos até que ponto a actual detecção de uma falha influência os resultados obtidos. Seria necessário executar as mesmas simulações no mesmo ambiente experimental para podermos efectuar uma comparação válida. A simulação necessária para comparação de resultados enquadra-se como trabalho futuro, bem como a implementação da arquitectura proposta no capítulo 5 onde são eliminados todos os pontos críticos detectados na arquitectura implementada. Ainda como trabalho futuro propomos a avaliação da taxa de cobertura dos mecanismos de tolerância a falhas implementados e tentar encontrar soluções para as situações de excepção descritas anteriormente.

Bibliografia

[1] Gameforge AG. Games. `http://www.gameforge.de`.

[2] Eduardo Aires. Multi-player online rpg com tolerância a falhas de dados. Technical Report 49, Departamento de Informática da Universidade da Beira Interior, 2008.

[3] Marios Assiotis and Velin Tzanov. A distributed architecture for mmorpg. In *NetGames '06: Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games*, page 4, New York, NY, USA, 2006. ACM.

[4] Franz Aurenhammer. Voronoi diagrams—a survey of a fundamental geometric data structure. *ACM Comput. Surv.*, 23(3):345–405, 1991.

[5] Daniel Bauer, Sean Rooney, and Paolo Scotton. Network infrastructure for massively distributed games. In *NetGames '02: Proceedings of the 1st workshop on Network and system support for games*, pages 36–43, New York, NY, USA, 2002. ACM.

[6] Philip A. Bernstein, Vassco Hadzilacos, and Nathan Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.

[7] Jason Brittain and Ian F. Darwin. *Tomcat: the definitive guide, 2nd edition*. O'Reilly, 2007.

[8] Jin Chen, Baohua Wu, Margaret Delap, Björn Knutsson, Honghui Lu, and Cristiana Amza. Locality aware dynamic load management for massively multiplayer games. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 289–300, New York, NY, USA, 2005. ACM.

[9] Roman Chertov and Sonia Fahmy. Optimistic load balancing in a distributed virtual environment. In *NOSSDAV '06: Proceedings of the 2006 international workshop on*

Network and operating systems support for digital audio and video, pages 1–6, New York, NY, USA, 2006. ACM.

[10] Eric Cronin, Anthony R. Kurc, Burton Filstrup, and Sugih Jamin. An efficient synchronization mechanism for mirrored game architectures. *Multimedia Tools Appl.*, 23(1):7–30, 2004.

[11] Christophe Diot and Laurent Gautier. A distributed architecture for multiplayer interactive applications on the internet. *IEEE Networks magazine*, 13:6–15, 1999.

[12] História do jogo on-line Gungeons & dragons. História do jogo on-line gungeons & dragons. `http://www.wizards.com/dnd/DnDArchives_History.asp`.

[13] Ta Nguyen Binh Duong and Suiping Zhou. A dynamic load sharing algorithm for massively multiplayer online games. In *Networks, 2003. ICON2003. The 11th IEEE International Conference on*, pages 131–136, Sept.-1 Oct. 2003.

[14] Frank Glinka, Alexander Ploss, Sergei Gorlatch, and Jens Müller-Iden. High-level development of multiserver online games. *Int. J. Comput. Games Technol.*, 2008:1–16, 2008.

[15] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. The dangers of replication and a solution. In *SIGMOD '96: Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, volume 25, pages 173–182, New York, NY, USA, June 1996. ACM Press.

[16] Robert Greene. Advanced data management for mmog. 2008.

[17] Carsten Griwodz. State replication for multiplayer games. In *NetGames '02: Proceedings of the 1st workshop on Network and system support for games*, pages 29–35, New York, NY, USA, 2002. ACM.

[18] M. Hori, T. Iseri, K. Fujikawa, S. Shimojo, and H. Miyahara. Scalability issues of dynamic space management for multiple-server networked virtual environments. In *Communications, Computers and signal Processing, 2001. PACRIM. 2001 IEEE Pacific Rim Conference on*, volume 1, pages 200–203 vol.1, 2001.

[19] Shun-Yun Hu, Shao-Chen Chang, and Jehn-Ruey Jiang. Voronoi state management for peer-to-peer massively multiplayer online games. In *Consum unications and Networking Conference, 2008. CCNC 2008. 5th IEEE*, pages 1134–1138, 2008.

[20] Shun-Yun Hu and Guan-Ming Liao. Scalable peer-to-peer networked virtual environment. In *NetGames '04: Proceedings of 3rd ACM SIGCOMM workshop on Network and system support for games*, pages 129–133, New York, NY, USA, 2004. ACM.

[21] Takuji Iimura, Hiroaki Hazeyama, and Youki Kadobayashi. Zoned federation of game servers: a peer-to-peer approach to scalable multi-player online games. In *NetGames '04: Proceedings of 3rd ACM SIGCOMM workshop on Network and system support for games*, pages 116–120, New York, NY, USA, 2004. ACM.

[22] Bettina Kemme and Gustavo Alonso. A suite of database replication protocols based on group communication primitives. In *ICDCS '98: Proceedings of the The 18th International Conference on Distributed Computing Systems*, page 156, Washington, DC, USA, 1998. IEEE Computer Society.

[23] Bettina Kemme and Gustavo Alonso. A new approach to developing and implementing eager database replication protocols. *ACM Trans. Database Syst.*, 25(3):333–379, 2000.

[24] Björn Knutsson, Honghui Lu, Wei Xu, and Bryan Hopkins. Peer-to-peer support for massively multiplayer games, 2004.

[25] Jean-Claude Laprie and Brian Randell. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Secur. Comput.*, 1(1):11–33, 2004. Fellow-Avizienis, Algirdas and Senior Member-Landwehr, Carl.

[26] Yi Lin, Bettina Kemme, Marta Pati no-Martínez, and Ricardo Jiménez-Peris. Middleware based data replication providing snapshot isolation. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 419–430, New York, NY, USA, 2005. ACM.

[27] Yi Lin, Bettina Kemme, Marta Patino-Martinez, and Ricardo Jimenez-Peris. Applying database replication to multi-player online games. In *NetGames '06: Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games*, page 15, New York, NY, USA, 2006. ACM.

[28] Fengyun Lu, Simon Parkin, and Graham Morgan. Load balancing for massively multiplayer online games. In *NetGames '06: Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games*, page 1, New York, NY, USA, 2006. ACM.

[29] John C. S. Lui and M. F. Chan. An efficient partitioning algorithm for distributed virtual environment systems. *IEEE Trans. Parallel Distrib. Syst.*, 13(3):193–211, 2002.

[30] John C. S. Lui and M. F. Chan. An efficient partitioning algorithm for distributed virtual environment systems. *IEEE Trans. Parallel Distrib. Syst.*, 13(3):193–211, 2002.

[31] D. L. Mills. Internet time synchronization: The network time protocol, 1989.

[32] Dugki Min, E. Choi, Donghoon Lee, and Byungseok Park. A load balancing algorithm for a distributed multimedia game server architecture. In *Multimedia Computing and Systems, 1999. IEEE International Conference on*, volume 2, pages 882–886 vol.2, Jul 1999.

[33] Matthew Moodie. *Pro Apache Tomcat 6 (Pro)*. Apress, Berkely, CA, USA, 2007.

[34] P. Morillo, J. M. Orduña, M. Fernández, and Jose Duato. An adaptive load balancing technique for distributed virtual environment systems. In *PROCEEDINGS OF THE 15TH IASTED INTERNATIONAL PDCS-03*, pages 256–261. Press, 2003.

[35] Jens Müller, Stefan Fischer, Sergei Gorlatch, and Martin Mauve. A proxy server-network for real-time computer games. pages 606–613. 2004.

[36] Jens Müller and SERGEI GORLATCH. Rokkatan: scaling an rts game design to the massively multiplayer realm. *Comput. Entertain.*, 4(3):11, 2006.

[37] Jens Müller, Andreas Gössling, and Sergei Gorlatch. On correctness of scalable multi-server state replication in online games. In *NetGames '06: Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games*, page 21, New York, NY, USA, 2006. ACM.

[38] B. Ng, R.W.H. Lau, A. Si, and F.W.B. Li. Multiserver support for large-scale distributed virtual environments. *Multimedia, IEEE Transactions on*, 7(6):1054–1065, Dec. 2005.

[39] M. Tamer Ozsu. *Principles of Distributed Database Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2007.

[40] Christian Plattner and Gustavo Alonso. Ganymed: Scalable replication for transactional web applications. In *In Proceedings of the 5th ACM/IFIP/Usenix International Middleware Conference*, pages 155–174, 2004.

[41] Alexander Ploss, Stefan Wichmann, Frank Glinka, and Sergei Gorlatch. From a single- to multi-server online game: a quake 3 case study using rtf. In *ACE '08: Proceedings of the 2008 International Conference on Advances in Computer Entertainment Technology*, pages 83–90, New York, NY, USA, 2008. ACM.

[42] Michael Gallagher president and ESA CEO. 2009 state of the industry address. `http://www.theesa.com/`.

[43] The Apache Jakarta Project. The apache jakarta project. `http://www.biometricnewsportal.com/palm_biometrics.asp`.

[44] Simon Rieche, Klaus Wehrle, Marc Fouquet, Heiko Niedermayer, Timo Teifel, and Georg Carle. Clustering players for load balancing in virtual worlds. *Int. J. Adv. Media Commun.*, 2(4):351–363, 2008.

[45] Baron Schwartz, Peter Zaitsev, Vadim Tkachenko, Jeremy Zawodny, Arjen Lentz, and Derek J. Balling. *High performance mysql, 2nd edition*. O'Reilly, 2008.

[46] Milan E. Soklic. Simulation of load balancing algorithms: a comparative study. *SIGCSE Bull.*, 34(4):138–141, 2002.

[47] Total MMOG Active Subscriptions. Total mmog active subscriptions. `http://www.mmogchart.com/Chart4.html`.

[48] Walker White, Christoph Koch, Nitin Gupta, Johannes Gehrke, and Alan Demers. Database research opportunities in computer games. *SIGMOD Rec.*, 36(3):7–13, 2007.

[49] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Understanding replication in databases and distributed systems. In *ICDCS '00: Proceedings of the The 20th International Conference on Distributed Computing Systems (ICDCS 2000)*, page 464, Washington, DC, USA, 2000. IEEE Computer Society.

[50] Matthias Wiesmann. Comparison of database replication techniques based on total order broadcast. *IEEE Trans. on Knowl. and Data Eng.*, 17(4):551–566, 2005. Member-Schiper, Andre.