VFC-game

Bruno Filipe Loureiro

Instituto Superior Técnico
bruno.loureiro@ist.utl.pt

Abstract. Os jogos *online* multi-jogador cada vez têm mais popularidade. Manter o estado consistente e actualizado entre todos os jogadores em tempo real de forma a garantir a jogabilidade, é crítico. Enviar o estado completo do jogo a todos os jogadores não escala com o número de jogadores. Assim, pretende-se adaptar uma técnica de gestão dos dados replicados e aplicá-la a um jogo real. Espera-se assim reduzir a largura de banda necessária para manter actualizado o estado de cada jogador de forma a aumentar a escalabilidade do jogo. Isto sem prejudicar a jogabilidade e possibilitando suportar um maior número de jogadores.

Key words: jogos multi-jogador, consistência optimista, gestão de interesse

1 Introdução

Nos últimos anos a popularidade dos jogos *online* multi-jogador tem vindo a crescer rapidamente. Entre as razões para o crescimento está a cada vez maior taxa de penetração da internet de banda larga. Um tipo de jogo que emergiu com estas novas possibilidades, foi o jogo *online* multi-jogador em massa (MMOG). Os MMOGs caracterizam-se pelo elevado número de jogadores suportados simultaneamente e pela partilha de um enorme mundo virtual persistente.

Uma categoria popular de jogos *online* que não se encaixa na dimensão dos MMOGs são os *First Person Shooters* (FPS). Os FPS têm uma interactividade rápida que dá ênfase à destreza e tempos de reacção dos jogadores. Em comparação com as centenas ou milhares de jogadores que participam simultaneamente num MMOG, o número de jogadores num FPS situa-se nas dezenas (tipicamente entre 16 e 32). Os FPS diferem dos MMOGs por permitirem que os servidores possam ser alojados por jogadores nos seus computadores pessoais. No entanto, o número de jogadores suportado por estes servidores é principalmente limitado pela largura de banda disponível. Os servidores do jogo também podem ser alojados em servidores mais poderosos e com maior largura de banda, normalmente proporcionados por comunidades de jogadores.

De forma a proporcionar um bom desempenho, cada jogador possui cópias locais (réplicas) do estado do jogo relativo aos outros jogadores (posições dos jogadores, projécteis, etc.). Manter as réplicas consistentes em tempo real de forma a garantir a jogabilidade sem utilizar demasiada largura de banda é a principal dificuldade na implementação de um jogo *online*. Associado à manutenção da

consistência está a escalabilidade do jogo, uma vez que quanto menor for a eficiência na comunicação menos jogadores são suportados.

Uma das soluções existentes para lidar com a escalabilidade é a arquitectura utilizada. Há dois tipos de arquitecturas principais: Cliente/Servidor (C/S) e entre pares (P2P)[5]. Na arquitectura C/S existe um servidor central que recebe as actualizações de estado dos clientes e que as propaga para os outros clientes. Na arquitectura P2P não existe um servidor central, as funções do servidor são divididas igualmente entre todos os clientes envolvidos, os quais comunicam directamente entre si. Ambas as arquitecturas possuem variações híbridas.

No caso da manutenção de consistência, existe uma técnica chamada gestão de interesse (IM)[3][14]. IM consiste na redução da quantidade de mensagens de actualização de estado necessárias. Esta redução é possível através da exploração dos limites sensoriais dos jogadores. Cada jogador tem uma área de interesse (AoI) associada e apenas está interessado nas actualizações de estado dos objectos dentro dessa área.

Outra técnica, que permite reduzir a frequência das mensagens é o *Dead Reckoning*[14]. Esta técnica permite reduzir a frequência das mensagens de actualização de posições dos objectos através da previsão de movimentos tendo em conta o movimento actual. Funciona bem para intervalos reduzidos entre mensagens[12].

Por fim, ao nível da camada da comunicação, temos um conjunto de técnicas utilizadas directamente pelos jogos ou por intermédio de bibliotecas[5][14][6]. Estas técnicas concentram-se na compressão, agregação e *multicast* de pacotes de forma a tornar a comunicação o mais eficiente possível.

Com este trabalho pretende-se adaptar uma técnica de IM existente e aplicá-la a um jogo real. O principal objectivo é aumentar a escalabilidade, através da redução da quantidade de comunicação, mantendo a jogabilidade e consistência. Para atingir este objectivo será desenvolvido um *middleware* que implementa a solução. Desta forma, é possível separar a lógica de jogo dos detalhes de manutenção de consistência.

Um aspecto fundamental para a aplicação da solução a um jogo real prende-se com a necessidade de o jogo ter o código fonte disponível, ou seja, o jogo tem de ser *open source* ou um jogo comercial para o qual foi disponibilizado o código fonte. Foi escolhido um FPS pois existe muita oferta para este tipo de jogo com as condições referidas. O FPS escolhido foi o *Cube 2: Sauerbraten*[1]. As razões para a sua escolha assim como outros tipos de jogos considerados encontram-se na secção 3.1.

Um desafio fundamental deste trabalho está ligado à natureza dos FPS. Os FPS oferecem principalmente mapas (mundos virtuais) de dimensão reduzida ao mesmo tempo que proporcionam um grande alcance de visão. De forma a aumentar o impacto da solução, vão ser tidas em conta as características dos FPS na sua adaptação. Para a avaliação da solução vai ser dada preferência a mapas de grande dimensão, o que faz sentido, uma vez que o objectivo é aumentar o número de jogadores suportado.

[1] Cube 2: Sauerbraten, `http://sauerbraten.org/`

Este relatório está organizado da seguinte forma: na secção 2 descreve-se o trabalho relacionado, focando as soluções existentes em termos arquitecturais e de consistência; a secção 3 contém a solução proposta; a metodologia para avaliação da solução encontra-se na secção 4; segue-se a conclusão na secção 5.

2 Trabalho relacionado

Esta secção começa com a explicação dos conceitos base relativos a jogos *online* multi-jogador, nomeadamente jogos na categoria FPS. Seguidamente, apresentam-se as arquitecturas principais, assim como as características de cada uma. De seguida, introduz-se os conceitos base relativos à consistência de dados replicados. Nas secções 2.4, 2.5 e 2.6, apresentam-se as soluções actuais para redução e optimização da comunicação nos jogos. Termina-se com a apresentação dos sistemas académicos e comerciais com maior relevância para este trabalho.

2.1 *First Person Shooters*

Num FPS vários jogadores competem num mundo virtual através da internet. Cada jogador controla um *avatar* (representação digital do jogador) no mundo virtual. Os *avatares* são controlados pelos jogadores através de dispositivos de entrada como, por exemplo, o teclado e rato. Os *avatares* podem cooperar em equipas ou competirem contra todos individualmente. Podem existir objectos que o *avatar* pode apanhar. Esses objectos podem ser pacotes de saúde, munições, armas, armaduras, etc.

Cada *avatar* tem um estado associado. O estado é caracterizado por atributos como posição, nível de saúde, armas e respectivas munições, armadura, pontuação, etc. O estado é alterado através da interacção com outros *avatares* ou objectos e através da própria movimentação.

O modelo comum subjacente a estes jogos consiste num sistema de múltiplos servidores independentes. Cada servidor tem associado um mundo virtual com um determinado modo de jogo e com um número máximo de jogadores suportado. A escolha do servidor é feita pelo jogador tendo em conta o modo de jogo pretendido, o número de jogadores e a latência. Cada competição é limitada por tempo ou por pontuação; quando algum destes limites é ultrapassado o servidor inicia uma nova competição, a qual pode ser num mundo virtual diferente.

Uma vez que o estado local de cada jogador é constituído por numerosos objectos remotos, uma solução ingénua para os manter actualizados seria pedir a todos os jogadores envolvidos, durante o processamento de cada *frame*, o estado actualizado para os objectos remotos. Esta solução é impraticável no contexto da internet uma vez que as latências na comunicação excedem o tempo de processamento (tipicamente 16ms para uma frequência de actualização do monitor de 60 *frames* por segundo). Outro problema seria a quantidade de largura de banda necessária para receber o estado completo em cada *frame*.

O que se faz na prática é manter em cada cliente réplicas dos objectos remotos. Para efeitos de processamento da lógica de jogo e renderização recorre-se

ao estado das réplicas. As réplicas podem não reflectir o estado imediato e são actualizadas periodicamente de acordo com um modelo de consistência.

As principais limitações que levam à necessidade de modelos de consistência eficientes estão associados às características da internet, nomeadamente a largura de banda e a latência da comunicação.

Largura de banda. A largura de banda representa a quantidade de informação que é possível enviar e receber por unidade de tempo. É o principal limite no número de jogadores suportado. A escalabilidade de um sistema aumenta com a diminuição da quantidade de comunicação necessária por jogador.

Latência. A latência é o tempo que uma mensagem demora a ir e voltar a um servidor ou a outro cliente. Em termos práticos é o tempo que uma acção local envolvendo outro jogador demora até que os outros jogadores vejam o mesmo resultado. Assim sendo, valores de latência elevados prejudicam a interactividade dos jogos. Isto é principalmente importante nos FPS que, devido à sua natureza altamente interactiva, toleram latências até 100ms no máximo[11].

2.2 Arquitecturas

Por arquitectura entende-se a forma como os nós comunicam entre si. Os nós dividem-se em clientes e servidores. Os clientes, no caso particular dos jogos, são os terminais onde se processam as entradas dos jogadores. Os servidores são os responsáveis pelo processamento global do estado do jogo. Os servidores podem ser alojados em máquinas dedicadas de grande capacidade, tanto em processamento como de comunicação.

Há duas arquitecturas principais. A mais comum e mais simples é a arquitectura Cliente/Servidor (C/S). Na arquitectura C/S os clientes comunicam apenas com os servidores. Noutra arquitectura, a P2P, os nós são chamados de *peers* e têm funções de cliente e servidor. O processamento é dividido entre todos os nós de forma igual e a comunicação é feita directamente entre nós. Como forma de atenuar as desvantagens de cada um dos tipos de arquitecturas, ambas têm variações híbridas que utilizam alguns conceitos da outra.

2.2.1 Cliente/Servidor. Numa arquitectura Cliente/Servidor (C/S) os clientes comunicam com os servidores (figura 1). As mensagens de actualização de estado por parte dos clientes são enviadas para o servidor adequado mesmo que sejam destinadas a outros clientes. O servidor possui um estado global do jogo mais actualizado que os clientes por ser o ponto central e é responsável por manter os clientes actualizados.

Esta arquitectura é a mais utilizada e mais fácil de implementar. Todavia, a escalabilidade está limitada ao nível do servidor pela capacidade de processamento e, mais importante, pela largura de banda de recepção e envio disponível. Outras vantagens são o controlo centralizado sobre aspectos de segurança como a detecção de batotas e autenticação de clientes. Os clientes necessitam de pouca largura da banda por apenas comunicarem com o servidor. Outra desvantagem

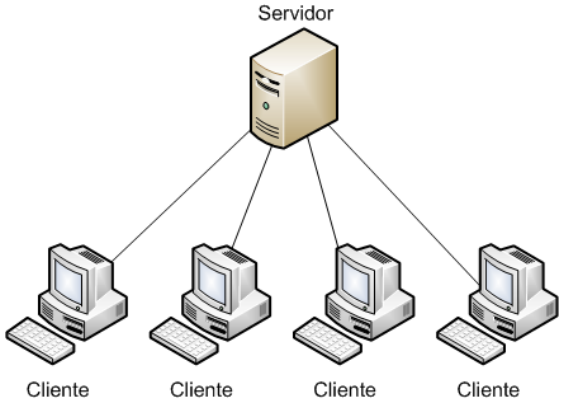

Fig. 1. Exemplo de uma arquitectura Cliente/Servidor

associada à centralidade do servidor é que se ele falha o jogo acaba para todos os jogadores ligados a ele.

De forma a aumentar a escalabilidade, mas mantendo a arquitectura C/S, existe uma variação que recorre à utilização de múltiplos servidores para distribuir a carga[4]. Nesta solução os vários servidores estão ligados através de uma rede dedicada de elevado desempenho.

Como a latência aumenta com o aumento da distância entre o cliente e o servidor, os servidores podem estar distribuídos geograficamente de forma a proporcionar uma experiência semelhante para todos os jogadores, independentemente da sua localidade. Para os clientes, apesar de existirem múltiplos servidores o funcionamento processa-se de forma igual à existência de um único servidor.

A divisão da carga em vários servidores pode ser feita de duas maneiras. Uma delas consiste em replicar o estado completo do jogo em todos os servidores. A alternativa é a divisão do mundo virtual em zonas e atribuir uma zona diferente a cada servidor.

Na replicação do estado completo, os clientes podem-se ligar manualmente ou automaticamente ao servidor com menor latência ou ao servidor com menos carga. Neste sistema é necessário manter as réplicas consistentes entre os vários servidores.

No sistema de divisão do mundo virtual, a atribuição dos clientes por servidor é feita através da sua posição no mundo virtual. Quando um cliente muda de região o seu estado é transferido para o servidor respectivo. Neste caso não existem réplicas; no entanto, quando a AoI de um cliente ultrapassa a fronteira da região é necessário subscrever os servidores abrangidos.

Através desta divisão em múltiplos servidores, deixa de haver um único ponto de falha, perdendo-se apenas os clientes associados ao servidor que falhou, os quais podem-se ligar a outro servidor no caso da replicação do estado completo. No caso da divisão por regiões, caso o sistema suporte, pode ser feita a redistribuição das regiões para ter em conta a região do servidor que falhou.

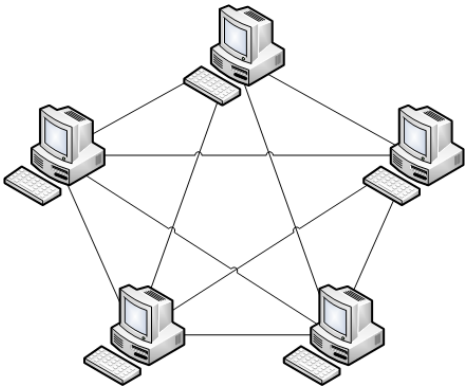

Fig. 2. Exemplo de uma arquitectura P2P

2.2.2 P2P. Na arquitectura P2P não existe um servidor central; todos os nós são tratados da mesma forma com funções de cliente e servidor (figura 2). Cada nó contribui com parte da sua capacidade de processamento e de largura de banda para a gestão global do estado do jogo. Uma vez que as responsabilidades de servidor estão distribuídas entre todos os nós, no caso da falha de um nó, o sistema continua a funcionar. Todavia, para isso é necessária redundância nas responsabilidades de forma a permitir a saída de nós sem prejudicar o estado do jogo.

A escalabilidade desta arquitectura é elevada, pois com cada entrada de um nó os recursos disponíveis aumentam. Contudo, é necessário organizar bem a estrutura da comunicação, uma vez que para um número elevado de nós torna-se impraticável comunicar com todos, principalmente tendo em conta a baixa largura de banda de envio disponível para a maioria dos utilizadores. Outra vantagem deste tipo de arquitectura é a redução na latência da comunicação. Isto verifica-se pois os clientes comunicam directamente entre si, não sendo necessário passar pelo ponto intermédio do servidor.

Outro problema prende-se com a complexidade de implementação associada à divisão de responsabilidades entre os vários clientes e à necessidade de utilização de mecanismos para manter o estado sincronizado entre todos os clientes. Uma dificuldade que surge com a característica de cada cliente ser responsável pelo processamento de parte do estado é como evitar batotas. Devido a todos estes problemas, a utilização desta arquitectura em jogos de grande dimensão, como MMOGs, continua a ser mais académica do que comercial.

Uma variação da arquitectura P2P é a distinção entre responsabilidades de cada nó. Neste caso existem nós com maior responsabilidade que outros. Esses nós com maior responsabilidade podem ser servidores dedicados. Os servidores podem servir como um simples ponto de entrada com funções de validação, podem ser utilizados de forma a detectar inconsistências ou para aliviar a carga dos nós.

2.3 Consistência

A replicação da informação consiste em manter cópias (réplicas) da informação em diversos computadores. As motivações por trás da replicação são o aumento do desempenho e da disponibilidade relativamente às soluções sem replicação. O desempenho é obtido uma vez que o acesso a informação local é mais rápido que aceder à mesma informação remotamente. No caso da comunicação com o dono da informação necessária se perder temporariamente ou por longos períodos de tempo, a informação continua a poder ser acedida localmente. Contudo, de forma a manter as réplicas consistentes são necessários mecanismos de replicação. As principais dificuldades que estes mecanismos enfrentam são como lidar com os conflitos que surgem quando clientes diferentes alteram a mesma réplica ao mesmo tempo; e a rapidez com que a consistência é mantida de modo a que seja garantida a jogabilidade.

A gestão da replicação de forma a lidar com a concorrência de alterações das réplicas pode ser feita através de duas abordagens: replicação pessimista e replicação optimista[13]. A ideia por trás da replicação pessimista consiste em evitar preventivamente o aparecimento de conflitos. Quando um cliente quer alterar o estado de uma réplica, o acesso à réplica é bloqueado a todos os outros clientes até que a alteração esteja concluída em todas as réplicas. O processo de actualização das réplicas é feito de forma síncrona através de um algoritmo de sincronização que envolve todos os clientes detentores da réplica. A sincronização de vários clientes remotos introduz uma latência considerável para cada actualização. Tem a vantagem que nenhum cliente acede a informação inconsistente.

Por outro lado temos a replicação optimista. Esta abordagem assume a existência de poucos conflitos e permite que o estado das réplicas divirja nos vários clientes. Aqui, contrariamente à abordagem pessimista, um cliente é livre de actualizar a réplica local independentemente. Após a actualização local, a actualização das outras réplicas é feita de forma assíncrona, mas não necessariamente de forma imediata. Os conflitos são resolvidos quando surgem. A resolução de conflitos é dependente do sistema onde aparecem; podem ser resolvidos automaticamente ou de forma manual. Esta forma de replicação é designada de *eventual consistency*: se não existirem actualizações durante um longo período de tempo, o estado das réplicas irá convergir, mas dentro de um tempo não especificado. No entanto, alguns sistemas necessitam de garantias de consistência mais estritas, como o caso dos jogos *online*, de forma a garantir a jogabilidade.

A limitação da divergência das réplicas pode ser feita de várias formas. As mais simples são garantias temporais, que permitem que a réplica divirja até um limite máximo de tempo[1]. O sistema TACT[16], além de limitar a divergência através de limites temporais, permite especificar um valor numérico para limitar o número de actualizações locais toleradas até ser necessário propagar as actualizações. Outra forma de limitar a divergência é chamada de gestão de interesse.

2.4 Gestão de interesse

Nos jogos *online*, os clientes necessitam de ter as réplicas consistentes de forma a usufruírem de boa jogabilidade. Manter o estado consistente entre a totalidade

dos jogadores requer maior largura de banda à medida que o número de jogadores aumenta. Uma primeira solução é actualizar as réplicas periodicamente e não de cada vez que há uma alteração. No entanto, tal não é suficiente; a gestão de interesse explora os limites sensoriais dos jogadores de forma a proporcionar uma maior redução. Um jogador não percepciona o estado global da mesma forma, há objectos que estão fora da sua visão e há outros que estão tão afastados que não são relevantes para o jogador.

Através da gestão de interesse, objectos próximos são actualizados frequentemente, enquanto que objectos distantes são actualizados menos frequentemente ou filtrados por completo. A gestão de interesse pode ser aplicada de várias formas: através da divisão do mundo virtual em regiões, através da noção de aura ou através da linha de visão.

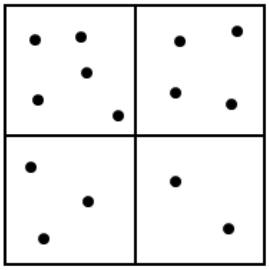

Fig. 3. Exemplo da divisão do mundo virtual em 4 regiões

2.4.1 Gestão de interesse baseada em regiões. Uma das abordagens de gestão de interesse, principalmente usada em jogos com arquitecturas P2P, é a divisão do mundo virtual em regiões (figura 3). Esta divisão permite que os clientes só recebam actualizações de estado de regiões em que estejam interessados. Um modelo comum que faz uso desta divisão é o *Publish-Subscribe*[7], onde os clientes subscrevem as regiões em que estão interessados em receber actualizações.

O desempenho deste método está directamente ligado à dimensão das regiões. Se a dimensão for muito pequena é necessário subscrever muitas regiões. Se, pelo contrário, for muito grande, a quantidade de actualizações irrelevantes ao jogador vai ser maior. A forma das regiões também influencia o desempenho no sentido em que as regiões hexagonais permitem que nos extremos fronteiriços se possam subscrever menos regiões em comparação com regiões rectangulares.Uma forma de resolver o problema da dimensão das regiões é através da subdivisão dinâmica de regiões à medida que a densidade populacional aumenta[15][9].

2.4.2 Gestão de interesse baseada em auras. A aura corresponde à área envolvente a um objecto e caracteriza as suas capacidades sensoriais. No contexto

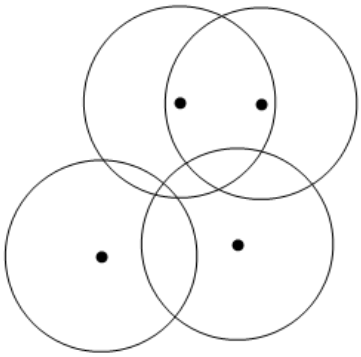

Fig. 4. Exemplo com as auras associadas a cada *avatar*

da gestão de interesse dois objectos só trocam mensagens quando as suas auras se intersectam. Uma utilização comum do conceito de aura é através da definição de uma área circular definida à volta dos *avatares*[3]. Este conceito é útil para limitar ou reduzir o estado de jogo transferido. Na figura 4 podemos ver um exemplo das auras associadas a cada *avatar*. Neste exemplo, apenas os dois *avatares* no topo da imagem trocam actualizações de estado.

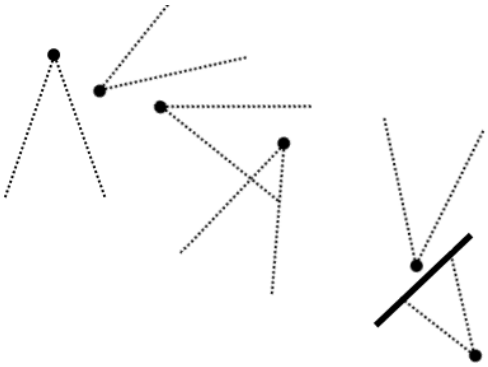

Fig. 5. Exemplo com a linha de visão de cada *avatar*

2.4.3 Gestão de interesse baseada em linha de visão. A abordagem anterior não distingue entre objectos visíveis e objectos ocultos pela geometria do mundo virtual. Esta abordagem combina o campo de visão dos *avatares* com a linha de visão. Além da redução da aura para ter em conta apenas o campo de visão, esta abordagem considera os obstáculos do mundo virtual para

filtrar objectos ocultos[3][10]. Isto permite uma melhor filtragem à custa de maior processamento. Em mundos virtuais com grande oclusão, tem a possibilidade de reduzir significativamente a comunicação. Na figura 5 podemos observar o impacto que a linha de visão tem na percepção de cada *avatar*.

2.5 *Dead Reckoning*

Outra abordagem para reduzir a largura de banda utilizada, consiste em enviar mensagens menos frequentemente. No entanto, é necessário que a redução da frequência não prejudique a jogabilidade. O *Dead Reckoning*[12] consiste em prever o movimento até que chegue um novo pacote, usando como base de cálculo os pacotes anteriormente recebidos.

As técnicas de previsão costumam utilizar a velocidade instantânea dos objectos de forma a melhorar os resultados. Se for usada também a aceleração consegue-se melhorar bastante a previsão. Como forma de reduzir o tamanho das mensagens, em vez destes valores adicionais serem enviados, podem ser obtidos aproximadamente através do histórico das últimas mensagens recebidas.

Quando é recebida uma nova mensagem, provavelmente a posição prevista não coincide com a recebida. A maneira mais simples de corrigir a posição é mover o objecto para a posição recebida, mas esta solução causa saltos desagradáveis nos movimentos. Uma solução melhor é aplicar um algoritmo de convergência linear, onde se calcula um ponto de convergência futuro para o qual o objecto se vai mover linearmente. Esta solução é melhor, mas continua a ter movimentos bruscos quando muda de direcção. Para melhorar ainda mais a convergência, pode-se aplicar um algoritmo que calcula pontos intermédios de adaptação de forma a produzir uma curva e proporcionando uma convergência suave. Esta solução, no entanto é mais exigente a nível computacional.

2.6 Técnicas ao nível da rede

A comunicação através da internet segue um modelo de melhor esforço onde a largura de banda e latências são heterogéneas. Como forma de reduzir o impacto destas limitações existem várias técnicas que podem ser utilizadas na camada de rede:

2.6.1 ***Multicast.*** Um servidor comunica com os clientes enviando pacotes individuais para cada um deles. No entanto, em muitos casos estes pacotes são iguais diferindo apenas no destino. Neste caso a técnica de *multicast*[5] pode reduzir o número de pacotes, igual ao número total de jogadores, a apenas um. Esta técnica tem o potencial de reduzir drasticamente a largura de banda utilizada pelo servidor. Contudo, para isto ser possível, é necessário que o *multicast* seja suportado ao nível do *hardware*, situação que não se verifica, pelo menos completamente, na internet. Existem alternativas que consistem na implementação de *multicast* por *software* em que os clientes se distribuem logicamente em forma de árvore e propagam as mensagens entre si desde a raiz da árvore até todas

as folhas. Isto tem a vantagem de reduzir os pacotes enviados, mas não tanto como a solução por *hardware*. Como cada pacote tem de percorrer vários clientes, incorre numa latência adicional possivelmente elevada.

2.6.2 Agregação. A comunicação gerada por um jogo costuma ser de pequena dimensão e bastante frequente. Esta característica traduz-se no envio de grande quantidade de pequenos pacotes, em que o cabeçalho pode ser maior que a informação enviada. A agregação[6][14] consiste em agrupar a informação em pacotes maiores de forma a utilizá-los mais eficientemente. A agregação de mensagens é realizada recorrendo a uma fila de mensagens. As mensagens vão sendo acumuladas até se verificar uma das seguintes condições: a dimensão máxima do pacote foi atingida, ou o tempo máximo de tolerância foi ultrapassado. Esta técnica tem a desvantagem de adicionar algum atraso ao envio dos pacotes, o qual está associado ao tempo de tolerância na acumulação de mensagens.

2.6.3 Codificação e compressão. A técnica que mais impacto tem na redução da largura de banda é a redução da dimensão das mensagens a enviar. De forma a atingir esse objectivo existem várias abordagens[6][14]:

Codificação usando o número de *bits* mínimo. Em vez de se representar as primitivas com a quantidade de *bits* normais, explora-se o contexto para que são usadas, representando os valores de forma eficiente. Normalmente necessita de intervenção do programador para atingir resultados óptimos.

Tabelas de indexação. Nos casos em que as mensagens contêm *strings* frequentes, pode ser usada uma tabela para indexar essas ocorrências, permitindo que se envie um código reduzido que identifique a *string* pretendida. As tabelas são normalmente construídas dinamicamente à medida que as mensagens vão sendo comunicadas. É necessário que todos os intervenientes possuam as entradas nas suas tabelas. Esta técnica pode ser aplicada também para indexar as assinaturas dos métodos invocados frequentemente por *Remote procedure call* (RPC).

Compressão sem perdas. No caso das técnicas anteriores não se adequarem, ou como complemento, pode-se recorrer à compressão sem perdas[5]. Dentro desta técnica temos, por exemplo, a compressão *Huffman* para *strings* ou a compressão do conteúdo total do pacote através de algoritmos como *BZip2* ou *Delta*. O método *Delta* envia apenas as diferenças em relação ao último pacote confirmado como recebido, explorando a semelhança dos pacotes frequentes.

2.7 Sistemas académicos

Nesta secção apresentam-se os sistemas académicos mais relevantes para este trabalho. Os sistemas enquadram-se no tópico de gestão de interesse.

2.7.1 *Donnybrook.* O sistema *Doonybrook*[10] reduz agressivamente a quantidade de comunicação em ambientes com pouca largura de banda. Este sistema é aplicado a uma arquitectura P2P de forma a minimizar o impacto da reduzida capacidade de largura de banda de envio por parte dos clientes.

O *Doonybrook* explora três princípios. Primeiro, os jogadores, através dos *avatares*, têm uma percepção limitada do mundo virtual; assim, dá-se mais importância às actualizações relevantes ao *avatar*. A frequência das actualizações varia consoante um valor estimado de atenção (relevância para o *avatar*). Os vários clientes recebem periodicamente dos outros clientes, os valores estimados de atenção de forma a enviarem as actualizações aos clientes com maior atenção neles. A largura de banda para envio é dividida numa fracção fixa para os clientes com maior atenção, ficando a restante para enviar actualizações para os clientes menos actualizados recentemente. O valor de atenção é calculado através de uma soma ponderada de três métricas: proximidade, mira e frescura da interacção.

Segundo, a interacção deve ser rápida e consistente; devido a isto, eventos mais importantes têm prioridade no envio. Terceiro, o realismo não deve ser sacrificado pela precisão, isto acontece com os objectos mais afastados devido à sua actualização menos frequente. Para atenuar este problema é sugerida uma técnica para guiar os objectos baseada em inteligência artificial. Este sistema tem uma desvantagem importante, está muito dependente da lógica de jogo o que não permite a sua exteriorização.

2.7.2 RING. O sistema RING[8] suporta um grande número de jogadores em ambientes com muita oclusão. A ideia chave deste sistema é a limitação de envio de actualizações de estado apenas para entidades que tenham linha de visão para a mesma. Neste sistema, cada cliente processa adicionalmente um pequeno conjunto de réplicas de entidades remotas. Essas entidades são simuladas entre recepções de actualização de estado. A comunicação entre clientes é feita por intermédio de servidores. Podem haver vários servidores ou apenas um. A vantagem da arquitectura C/S é que o servidor fica responsável pelo processamento das mensagens antes de as propagar.

Os servidores RING fazem a filtragem das mensagens e enviam-nas apenas para os clientes que tenham réplicas possivelmente na sua área de visão. Esta filtragem é feita com auxílio a pré-computações de visibilidade que determinam para cada divisão do mapa, que outras divisões são visíveis a partir dela. Esta simplificação da visibilidade, tendo em conta as divisões e não a posição de cada cliente sobrestima a área de visão, mas reduz a computação necessária. É da responsabilidade dos servidores a notificação dos clientes de cada vez que uma réplica entra ou sai da sua potencial área de visão. A grande vantagem do RING é que a largura de banda necessária para os clientes, não está dependente do número total de clientes envolvidos. Isto permite uma boa escalabilidade do sistema apesar de ser baseado em C/S.

Entre as desvantagens, estão o processamento necessário nos servidores e a latência introduzida por esse processamento e pela comunicação entre os vários servidores caso sejam utilizados mais que um. De forma similar ao sistema ante-

rior, este está também dependente da lógica de jogo, principalmente à informação geométrica do mundo virtual.

2.7.3 A³. O A³[2] é um algoritmo que combina a aura circular com o campo de visão dos *avatares*. O algoritmo recorre a um campo de visão de 180º para filtrar os objectos que estão atrás do *avatar*. No entanto, no caso de uma rotação brusca, objectos que estejam mesmo atrás do *avatar*, podem demorar um pouco até aparecerem. Isto prejudica a jogabilidade do jogo. Para corrigir este problema, o campo de visão é complementado com uma pequena aura circular de raio pequeno. Este algoritmo explora ainda outro ponto que tem a ver com a distância dos objectos. Objectos mais distantes não precisam de uma actualização tão frequente como os que estão próximos. Por esta razão, esta solução propõe uma redução linear da frequência de actualização que aumenta com a distância em relação ao jogador.

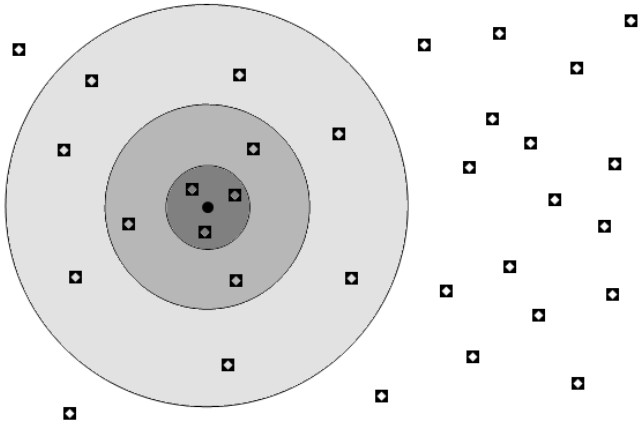

Fig. 6. Exemplo das zonas de consistência do VFC relativas a um *avatar*. A intensidade da cor representa o nível de consistência

2.7.4 *Vector-Field Consistency*. O VFC introduz um conceito de múltiplas zonas circulares concêntricas, centradas no *avatar*. Cada zona tem associado um nível de consistência. Esse nível de consistência é reduzido à medida que as zonas se situam mais longe do *avatar* (figura 6). Assim, é possível oferecer uma redução progressiva da quantidade de comunicação.

O VFC é um modelo de consistência optimista que possibilita que objectos replicados divirjam de forma limitada. Cada jogador tem uma vista local constituída por réplicas do mundo virtual completo. O VFC é responsável por gerir as divergências de cada vista. À medida que o estado do jogo vai progredindo,

o VFC dinamicamente aumenta ou diminui o nível de consistência das réplicas. Isto é feito com recurso à localidade espacial das réplicas em relação ao *avatar* do jogador, explorando o facto de que objectos mais afastados do *avatar* toleram um nível de consistência mais baixo do que objectos próximos.

Os níveis de consistência associados ao *avatar* são especificados pelo programador do jogo e consistem em vectores de três dimensões que definem os limites de divergência da réplica no tempo, sequência e valor. Cada vector está associado a uma zona. Um *pivot* gera zonas de consistência circulares concêntricas. Um *pivot* pode ser um *avatar* ou outro objecto. Objectos dentro da mesma zona estão sujeitos ao mesmo nível de consistência. Os níveis de consistência baixam de forma monótona à medida que a distância ao *pivot* aumenta. Como o nível de consistência de um objecto depende da sua distância aos *pivots*, cada objecto pode ter níveis de consistência diferentes para cada vista.

Os vectores tridimensionais associados a cada zona limitam a divergência máxima dos objectos numa determinada vista. As três dimensões que limitam a divergência são as seguintes:

- Tempo: Especifica o tempo máximo que uma réplica tolera sem ser actualizada. Este valor representa o tempo em segundos.
- Sequência: Especifica o número máximo de actualizações que pode ignorar até necessitar de uma nova actualização.
- Valor: Especifica a diferença máxima relativa entre conteúdos das réplicas. O conteúdo considerado depende da implementação e a diferença é medida em percentagem.

Quando alguma destas dimensões é excedida significa que a réplica necessita de nova actualização. É possível ignorar qualquer uma das dimensões, especificando um valor infinito.

O VFC oferece duas generalizações para uma utilização mais abrangente: *multi-pivot* e *multi-zones*. A primeira consiste na utilização de múltiplos *pivots* na mesma vista. A outra, permite que diferentes conjuntos de objectos com requisitos de consistência diferentes possam ter associadas zonas diferentes.

2.8 Sistemas comerciais

A informação disponível ao público acerca de jogos comerciais é muito reduzida, e da pouca informação disponível, ela é normalmente de alto nível e não contempla o tópico de gestão de interesse. Apesar de não ser um jogo comercial, de seguida apresenta-se a gestão de interesse aplicada num MMOG.

O *PlaneShift*[2] é um MMOG *open source* na categoria dos *Role Playing Games* (RPG) que está em activo desenvolvimento. Através da documentação disponível foi possível saber como é feita a comunicação entre os servidores e clientes. Segundo a documentação, o *PlaneShift* recorre à gestão de interesse baseada em auras. Cada cliente apenas recebe a informação relativa aos jogadores que estão dentro da sua aura. A aura neste caso consiste numa área circular centrada no *avatar*, a qual é nomeada de lista de proximidade.

[2] PlaneShift, `http://www.planeshift.it/`

3 Solução

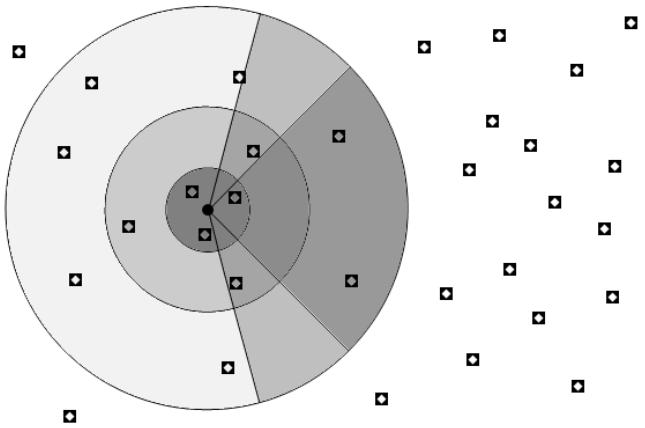

Fig. 7. Exemplo do VFC com a inclusão do campo de visão

As abordagens da secção 2.4, sofrem todas de um problema similar: usam um sistema de filtragem tudo ou nada. Isto pode causar o aparecimento ou desaparecimento repentino de objectos no campo de visão. A solução comum para minimizar esta situação é através do aumento da AoI. Isto prejudica o desempenho pois aumenta a comunicação necessária.

Foi escolhido o VFC como gestor de interesse para esta solução, pois oferece uma redução progressiva dos requisitos de consistência à medida que a distância aumenta. A flexibilidade do VFC foi outra razão para a sua escolha. A liberdade na especificação das zonas de consistência bem como os requisitos de cada uma, permite facilmente implementar vários modelos de consistência. Podemos especificar uma zona que abranja todo o mundo virtual com requisitos estritos de consistência e obtemos um modelo de consistência completo. Por outro lado, podemos especificar uma única zona de raio adequado e ficamos com um sistema similar às auras.

Este trabalho pretende expandir a funcionalidade do VFC e modificar um jogo de maneira a usá-lo como gestor de interesse. O objectivo é reduzir a largura de banda necessária para a comunicação e consequentemente aumentar a escalabilidade do jogo. Um objectivo secundário é implementar a solução como forma de *middleware* para permitir abstrair os detalhes de consistência da lógica de jogo, facilitando o desenvolvimento.

O VFC vai ser alterado tendo em conta o contexto dos FPS. Os FPS oferecem um grande alcance de visão e armas com *zoom*. Isto requer consistência forte até mais longe. Associado ao facto que a dimensão dos mapas dos FPS são relativamente pequenos, usar o VFC sem alterações causaria que os requisitos

de consistência fortes nos 360º que envolvem o *avatar* não iriam oferecer ganhos significativos em relação ao sistema actual utilizado pelo jogo. O sistema utilizado por este jogo, assim como em muitos outros deste tipo, é o envio do estado global para todos os jogadores.

De forma a adaptar o VFC a estes requisitos, pretende-se adicionar o conceito de campo de visão. Assim, permite reforçar os requisitos de consistência no ângulo de visão do *avatar* (tipicamente 90º) ao mesmo tempo que se permite relaxar a consistência para os objectos "atrás" dos *avatares*. O problema do *zoom* fica abrangido uma vez que quando o *zoom* aumenta, o campo de visão diminui. Como forma de evitar inconsistências repentinas nos objectos periféricos no caso de uma rotação rápida, é introduzido, de forma semelhante ao sistema de múltiplas zonas, uma zona intermédia de consistência ligeiramente mais reduzida. Isto pode ser observado na figura 7. Juntamente com a redução do campo de visão em situação de *zoom*, as zonas de consistência podem ser expandidas.

O *middleware* vai ser implementado em C#. A interacção entre o código C++ do jogo com o C# vai ser feita recorrendo à adaptação da interface *.Net* do *middleware* para objectos COM. O facto da linguagem escolhida ser o C# limita os jogos com que pode ser aplicado, nomeadamente jogos implementados em C. De forma a manter a independência em relação à lógica do jogo, vai ser disponibilizada uma interface que o jogo deve implementar para os objectos partilhados.

Através da API vai ser possível adaptar os requisitos de consistência de forma dinâmica, principalmente o campo de visão. De cada vez que o jogador fizer zoom, o campo de visão pode ser reduzido para o valor adequado. Vai ser mantida a arquitectura C/S já existente no jogo, pois o próprio facto de usar uma arquitectura P2P iria ter grande influência na escalabilidade, e o objectivo é aumentar a escalabilidade através da utilização de gestão de interesse face às soluções existentes.

A generalização de *multi-zones* do VFC pode ser aplicada para separar os requisitos mais fortes dos *avatares* dos requisitos mais relaxados de outros objectos: armas, munições, projécteis, etc.

Uma optimização que se pretende fazer ao nível da lógica do jogo é introduzir uma diferenciação para mapas em espaço aberto e mapas com espaços fechados. Esta diferenciação é adicionada na configuração do mapa e é determinado manualmente. Desta forma consegue-se melhorar o desempenho consoante o tipo de mapa. Por mapas de espaço aberto entende-se mapas de grande dimensão com pouca oclusão. Pelo contrário, os mapas de espaços fechados são mapas, não necessariamente de grande dimensão, constituídos predominantemente por pequenas divisões e corredores.

3.1 Escolha do jogo

As razões para a escolha do *Cube 2: Sauerbraten* de entre todos os FPS disponíveis foram as seguintes:

- Inteiramente implementado em C++ e fácil de compilar.

- Popular, tendo em conta o número de servidores disponíveis e a percentagem de utilização dos mesmos.
- Inclui mapas de grande dimensão na instalação normal complementados por mapas grandes que podem ser obtidos em *sites* da comunidade.
- Proporciona um sistema de criação e edição de mapas dentro do próprio jogo de forma fácil.
- Contrariamente a muitos FPS que intensificam o contraste entre *avatares* e mundo virtual, este proporciona uma iluminação normal com áreas escurecidas, algo que permite explorar melhor os limites sensoriais e potenciar maior impacto com este trabalho.

Entre os FPS considerados encontram-se o *Warsow*[3], *Nexuiz*[4], *Tremulous*[5], *Quake 3*[6], *Blood Frontier*[7], *AssaultCube*[8].

Warsow. Apesar de ser um jogo com bastante popularidade, os mapas são sobretudo de dimensão reduzida e com muitos espaços abertos. Oferece muito contraste entre *avatares* e mundo virtual. Outra dificuldade prende-se com o facto de o jogo ser implementado em C.

Nexuiz. Este jogo foi um bom candidato pois apresenta mapas de grande dimensão, principalmente o facto de possuir um modo de jogo baseado no *Onslaught* do *Unreal Tournament 2004*, modo esse que recorre a mapas de grandes. No entanto é também implementado em C.

Tremulous. Um jogo que mistura conceitos de estratégia em tempo real (RTS) com FPS com mapas relativamente pequenos, mas predominantemente fechados; mais uma vez sofre por ser feito em C.

Quake 3. Um jogo muito popular tanto para jogar como para modificar a nível académico. Feito em C.

Blood Frontier. Um jogo baseado no motor *Cube 2*, mas ainda em fase *beta* e consequentemente com pouca popularidade.

AssaultCube. Outra adaptação baseada no *Cube 2*, mas que a maioria dos mapas são de pequena dimensão.

Foram também considerados jogos RTS. O grande problema é a pouca oferta de jogos deste tipo com código fonte, e mesmo dentro desses, de forma geral, a qualidade segundo padrões actuais e a popularidade dos mesmos é baixa. Outro problema prende-se com a própria natureza destes jogos, em que a lógica de jogo permite um número reduzido de jogadores, tipicamente abaixo da dezena.

Um tipo de jogo que podia ganhar bastante com este trabalho seria um MMOG, contudo, apenas existe um MMOG *open source* chamado *PlaneShift*. O *PlaneShift* tem no entanto um problema: ainda se encontra numa versão *beta*.

[3] Warsow, `http://www.warsow.net/`
[4] Nexuiz, `http://www.alientrap.org/nexuiz/`
[5] Tremulous, `http://tremulous.net/`
[6] Quake 3, `http://www.quake3arena.com/`
[7] Blood Frontier, `http://www.bloodfrontier.com/`
[8] AssaultCube, `http://assault.cubers.net/`

3.2 Arquitectura

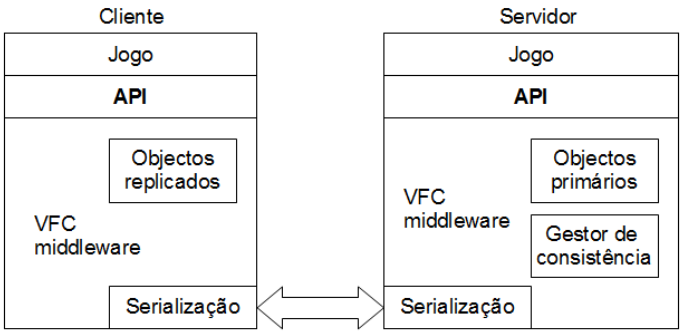

Fig. 8. Arquitectura da solução

O jogo vai assentar em cima do *middleware* e vai interagir com ele através de uma API. O *middleware* é dividido entre cliente e servidor. O cliente mantém um conjunto de objectos replicados. Os respectivos objectos primários encontram-se no servidor. A comunicação entre os objectos replicados e primários é feita via serialização. Esta comunicação é controlada pelo gestor de consistência, onde se situa a funcionalidade principal. A forma como os vários módulos se ligam entre si pode ser observada na figura 8.

A solução vai funcionar da seguinte forma. Cada cliente possui todas as réplicas do estado global. A consistência das réplicas em cada cliente é mantida conforme a sua relação espacial com o respectivo *avatar*. O cliente vai tratar as réplicas como objectos locais como faria se estivesse no modo *single-player*. Do lado do *middleware*, de cada vez que as réplicas forem alteradas o seu novo estado é enviado para o servidor. No servidor, as réplicas primárias são actualizadas e a propagação dessas alterações para os restantes clientes é feita tendo em conta os requisitos de consistência especificados.

Podemos dividir as actualizações das réplicas de duas formas. A primeira e mais simples lida com as alterações da réplica associada ao *avatar*, principalmente em relação à alteração da sua posição através dos movimentos efectuados no mundo virtual. Periodicamente, na ordem das dezenas de milissegundos, o estado da réplica do *avatar* é enviado ao servidor. Adicionalmente, a réplica contém informação acerca da orientação do *avatar* de forma a permitir que o servidor calcule o campo de visão necessário para a gestão da consistência.

A outra, está ligada com a interactividade por parte do *avatar* envolvendo outras réplicas. Neste caso, o cliente processa localmente a interacção e, no final, envia de imediato as alterações de todas as réplicas envolvidas ao servidor. O servidor tem a tarefa de actualizar os clientes cuja réplica foi alterada.

Um exemplo de uma interacção é quando um *avatar* dispara sobre outro. O cliente responsável pelo evento reduz a vida do *avatar* atingido. De seguida,

o *middleware* envia o estado do *avatar* atingido ao servidor. O servidor, após actualizar a réplica primária, envia o novo estado ao cliente que possui o *avatar* que foi atingido.

Esta solução segue o modelo de *fat client/thin server*. Neste modelo a maior parte do processamento é feito no cliente, libertando o servidor para fazer a gestão de consistência das réplicas. A principal vantagem na utilização deste modelo é que a latência da comunicação tem um impacto muito reduzido na jogabilidade.

4 Metodologia de Avaliação

Para avaliar a solução proposta pretende-se comparar qual a largura de banda utilizada por cada abordagem. As abordagens que se pretendem comparar são:

- *Cube 2: Sauerbraten* com o sistema nativo.
- *Cube 2: Sauerbraten* com VFC normal, com parâmetros de consistência forte e completa (similar ao sistema nativo).
- *Cube 2: Sauerbraten* com VFC normal, com parâmetros de mapa de espaços abertos.
- *Cube 2: Sauerbraten* com VFC normal, com parâmetros de mapa de espaços fechados.
- *Cube 2: Sauerbraten* com VFC alterado, com parâmetros de mapa de espaços abertos.
- *Cube 2: Sauerbraten* com VFC alterado, com parâmetros de mapa de espaços fechados.

Para a medição quantitativa pretende-se recorrer à utilização de *BOTs* (*avatares* controlados por inteligência artificial) simulados remotamente. Para facilitar a simulação pretende-se adaptar os *BOTs* para correrem em modo consola. Desta forma, sem o processamento gráfico, podem ser executadas várias instâncias de *BOTs* em cada computador.

Para a medição qualitativa, com o objectivo de medir a jogabilidade pretende-se utilizar jogadores reais e obter através de um questionário quais as suas impressões sobre a qualidade de jogo em cada abordagem.

A simulação deve ser feita em ambiente de rede local. Para simular o funcionamento através da internet podem ser introduzidos atrasos artificiais na comunicação.

5 Conclusão

Com este trabalho apresentámos as várias maneiras de lidar com a escalabilidade dos jogos *online* multi-jogador. Foram abrangidos os tópicos de arquitectura, consistência, gestão de interesse, *Dead Reckoning* e técnicas ao nível da rede. Terminámos com a apresentação dos trabalhos académicos e comerciais mais relevantes. Propusemos uma solução com o objectivo de aumentar a escalabilidade do *Cube 2: Sauerbraten*. A solução é baseada na alteração do VFC para

ter em conta aspectos particulares dos FPS de forma a melhorar o seu impacto. Para terminar, foram apresentados os métodos através dos quais vai ser medido o desempenho da nossa solução.

References

1. Rafael Alonso, Daniel Barbara, and Hector Garcia-Molina. Data caching issues in an information retrieval system. *ACM Trans. Database Syst.*, 15(3):359–384, 1990.
2. Carlos Eduardo Bezerra, Fábio R. Cecin, and Cláudio F. R. Geyer. A3: A novel interest management algorithm for distributed simulations of mmogs. In *DS-RT '08: Proceedings of the 2008 12th IEEE/ACM International Symposium on Distributed Simulation and Real-Time Applications*, pages 35–42, Washington, DC, USA, 2008. IEEE Computer Society.
3. Jean-Sébastien Boulanger, Jorg Kienzle, and Clark Verbrugge. Comparing interest management algorithms for massively multiplayer games. In *NetGames '06: Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games*, page 6, New York, NY, USA, 2006. ACM.
4. Eric Cronin, Burton Filstrup, Anthony R. Kurc, and Sugih Jamin. An efficient synchronization mechanism for mirrored game architectures. In *NetGames '02: Proceedings of the 1st workshop on Network and system support for games*, pages 67–73, New York, NY, USA, 2002. ACM.
5. Jeff Dyck. A survey of application-layer networking techniques for real-time distributed groupware. Technical report.
6. Jeff Dyck, Carl Gutwin, T. C. Nicholas Graham, and David Pinelle. Beyond the lan: techniques from network games for improving groupware performance. In *GROUP '07: Proceedings of the 2007 international ACM conference on Supporting group work*, pages 291–300, New York, NY, USA, 2007. ACM.
7. Stefan Fiedler, Michael Wallner, and Michael Weber. A communication architecture for massive multiplayer games. In *NetGames '02: Proceedings of the 1st workshop on Network and system support for games*, pages 14–22, New York, NY, USA, 2002. ACM.
8. Thomas A. Funkhouser. Ring: a client-server system for multi-user virtual environments. In *SI3D '95: Proceedings of the 1995 symposium on Interactive 3D graphics*, pages 85–ff., New York, NY, USA, 1995. ACM.
9. Rajesh Krishna Balan, Maria Ebling, Paul Castro, and Archan Misra. Matrix: adaptive middleware for distributed multiplayer games. In *Middleware '05: Proceedings of the ACM/IFIP/USENIX 2005 International Conference on Middleware*, pages 390–400, New York, NY, USA, 2005. Springer-Verlag New York, Inc.
10. Jeffrey Pang. Scaling peer-to-peer games in low-bandwidth environments. In *In Proc. 6th Intl. Workshop on Peer-to-Peer Systems IPTPS*, 2007.
11. Lothar Pantel and Lars C. Wolf. On the impact of delay on real-time multiplayer games. In *NOSSDAV '02: Proceedings of the 12th international workshop on Network and operating systems support for digital audio and video*, pages 23–29, New York, NY, USA, 2002. ACM.
12. Lothar Pantel and Lars C. Wolf. On the suitability of dead reckoning schemes for games. In *NetGames '02: Proceedings of the 1st workshop on Network and system support for games*, pages 79–84, New York, NY, USA, 2002. ACM.
13. Yasushi Saito and Marc Shapiro. Optimistic replication. *ACM Comput. Surv.*, 37(1):42–81, 2005.

14. Jouni Smed, Timo Kaukoranta, and Harri Hakonen. A review on networking and multiplayer computer games. In *Multiplayer Computer Games, Proc. Int. Conf. on Application and Development of Computer Games in the 21st century*, pages 1–5, 2002.
15. Shi Xiang-bin, Wang Yue, Li Qiang, Du Ling, and Liu Fang. An interest management mechanism based on n-tree. In *Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, 2008. SNPD '08. Ninth ACIS International Conference on*, pages 917–922, Aug. 2008.
16. Haifeng Yu and Amin Vahdat. Design and evaluation of a conit-based continuous consistency model for replicated services. *ACM Trans. Comput. Syst.*, 20(3):239–282, 2002.

Anexo - Planeamento do trabalho

	Mês
Estudo do código fonte do jogo	Janeiro
Desenho da API do *middleware*	Fevereiro
Implementação	Março - Abril
Avaliação	Maio
Escrita da tese	Junho
Revisão e entrega da tese	Julho