INSTITUTO SUPERIOR TÉCNICO
Universidade Técnica de Lisboa

VFC4FPS - Vector-Field Consistency for a First Person Shooter Game

Bruno Filipe da Costa Pereira Loureiro

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática e de Computadores

Júri

Presidente:	Professor Alberto Manuel Ramos da Cunha
Orientador:	Professor Paulo Jorge Pires Ferreira
Co-Orientador:	Professor Luís Manuel Antunes Veiga
Vogal:	Professor João António Madeiras Pereira

Novembro 2010

Agradecimentos

Quero agradecer e manifestar o meu apreço a todos os que colaboraram, directa ou indirectamente, na minha tese.

Agradeço aos meus orientadores Paulo Ferreira e Luís Veiga pelo apoio, incentivo, conselhos, conhecimentos e disponibilidade demonstrada em todas as fases que levaram à concretização deste trabalho.

Aos meus colegas e companheiros de tese Daniel Janeiro e João Lemos, por todo o apoio, companhia e momentos bem passados.

E claro, a todos os voluntários que despenderam algum do seu tempo a jogar esta tese por motivos de avaliação.

Lisboa, November 27, 2010
Bruno Filipe da Costa Pereira Loureiro

Resumo

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. Uma forma de aumentar a escalabilidade do jogo é através da redução do tráfego de comunicação, e uma forma de reduzir o tráfego de comunicação consiste em explorar os limites sensoriais dos jogadores. Contudo, as soluções que existem fazem uma filtragem tudo ou nada, onde um jogador apenas recebe actualizações de objectos dentro da sua zona sensorial. Neste trabalho usamos o *Vector-Field Consistency*. O VFC oferece uma redução progressiva da consistência. Isso é feito usando múltiplas zonas, cada uma com um conjunto de requisitos de consistência, os quais vão sendo reduzidos à medida que a distância aumenta. Pretendemos obter uma redução de pelo menos metade do tráfego original de um jogo *online*. Para esse efeito recorremos ao *Cube 2: Sauerbraten*. Tratando-se de um *First Person Shooter*, com um campo de visão limitado do mundo virtual, adicionámos ao VFC o conceito de campo de visão para melhorar o seu desempenho. Os resultados mostram que é possível reduzir significativamente o tráfego de comunicação sem prejudicar a consistência e a jogabilidade do jogo.

Abstract

Multiplayer online games are increasingly more popular. Keeping the game state updated and consistent among all players in soft real-time is critical. Sending the complete game state to all players does not scale with the number of players. One way to increase the game scalability is by reducing it's network traffic, and one way to reduce network traffic is by exploiting the player's sensory limits. However, current solutions typically use an all or nothing filtering, where a player only receives updates of objects inside his sensory zone. In this work we use the Vector-Field Consistency. VFC offers a progressive consistency reduction. They do this by using multiple zones, each with a set of consistency requirements, which are reduced with the increasing distance. We intend to obtain a network traffic reduction of at least half of the original traffic of an online game. To that effect we use the Cube 2: Sauerbraten, a First Person Shooter game. In this kind of game, players have a limited view of the virtual world. With that in mind we added the concept of Field of View to VFC in order to improve performance. Results show that is possible to significantly reduce network traffic without harming consistency and playability.

Palavras Chave
Keywords

Palavras Chave

Jogos Online Multi-jogador

Consistência Optimista

Gestão de Interesse

Localidade Espacial

Keywords

Multiplayer Online Games

Optimistic Consistency

Interest Management

Spatial locality

Índice

Lista de Figuras

Lista de Tabelas

Lista de Abreviações

AoI	Area of Interest
API	Application programming interface
C/S	Cliente/Servidor
FoV	Field of View
FPS	First Person Shooter
IM	Interest Management
MMOG	Massively Multiplayer Online Game
P2P	Peer-to-Peer
RPC	Remote Procedure Call
RTS	Real-Time Strategy
VFC	Vector-Field Consistency
VFC4FPS	Vector-Field Consistency for First Person Shooters

Capítulo 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][15]. 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. A AoI pode ser baseada em regiões, em raio de visão ou em linha de visão. Desta forma o jogador apenas está interessado, respectivamente, nas actualizações dos objectos dentro da região onde se encontra, que se encontram dentro de uma área envolvente ou que consegue ver.

Outra técnica, que permite reduzir a frequência das mensagens é o *Dead Reckoning*[15]. 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][15][6]. Estas técnicas concentram-se na compressão, agregação e *multicast* de pacotes de forma a tornar a comunicação o mais eficiente possível.

1.1 Objectivos

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. Pretendemos obter uma redução de tráfego para, pelo menos, metade do tráfego original. Para esse efeito, vamos utilizar o *Vector-Field Consistency* (VFC)[14] como base da nossa solução. O VFC, contrariamente às técnicas de IM, não aplica uma filtragem tudo ou nada. Permite que a consistência seja reduzida de forma gradual à medida que a distância aumenta. Um objectivo secundário é o desenvolvimento da nossa solução na forma de uma biblioteca. Podendo assim separar a lógica de jogo dos detalhes de manutenção de consistência.

1.2 Desafios

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 *First Person Shooter* (FPS) pois existe muita oferta para este tipo de jogo com as condições referidas. O FPS escolhido foi o *Cube 2: Sauerbraten*[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. Isto traduz-se na inclusão do conceito de campo de visão no VFC. Permitindo assim definir requisitos de consistência diferentes para os objectos, conforme eles se encontrem dentro ou fora do campo de visão do jogador.

1.3 Estrutura do Documento

Este documento está organizado da seguinte forma. No capítulo 2 descreve-se o trabalho relacionado, focando as várias soluções existentes de redução da quantidade de comunicação. O capítulo 3 descreve a

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

arquitectura do nosso sistema e o capítulo 4 contêm os detalhes sobre a sua implementação. No capítulo 5 descrevemos a avaliação que foi feita e os resultados obtidos. Terminamos com o capítulo 6, onde fazemos o sumário do trabalho realizado e apresentamos algumas ideias para trabalho futuro.

Capítulo 2

Trabalho relacionado

Este capítulo 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 2.1). 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 associada à centralidade do servidor é que se ele falha o jogo acaba para todos os jogadores ligados a ele.

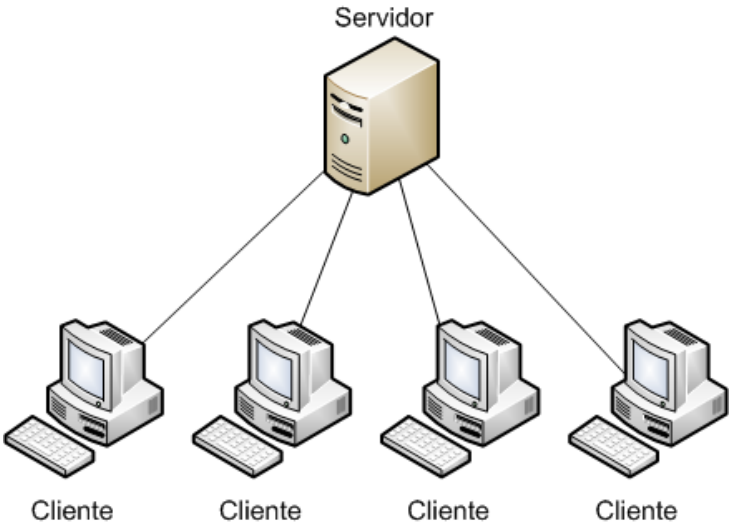


Figura 2.1: Exemplo de uma arquitectura Cliente/Servidor.

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.

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 2.2). 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

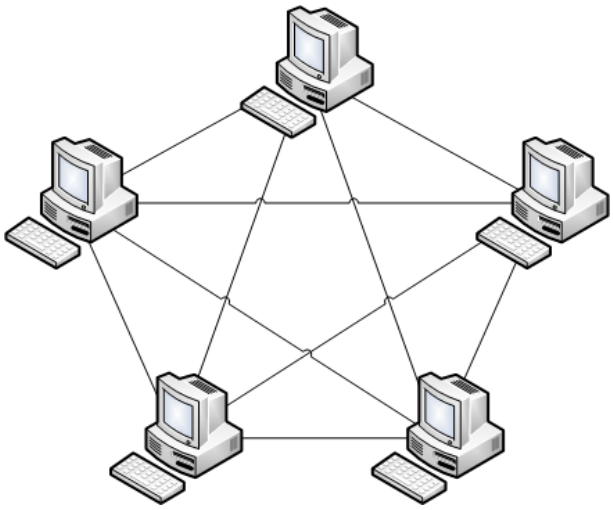


Figura 2.2: Exemplo de uma arquitectura P2P.

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 à informação local é mais rápido que aceder à mesma informação remotamente. No caso da comunicação com a réplica original se perder temporariamente ou por longos períodos de tempo, a réplica 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[17], 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.

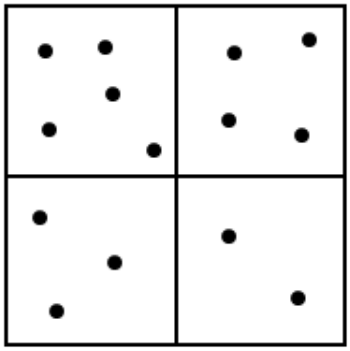


Figura 2.3: Exemplo da divisão do mundo virtual em 4 regiões.

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 2.3). 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[16][9].

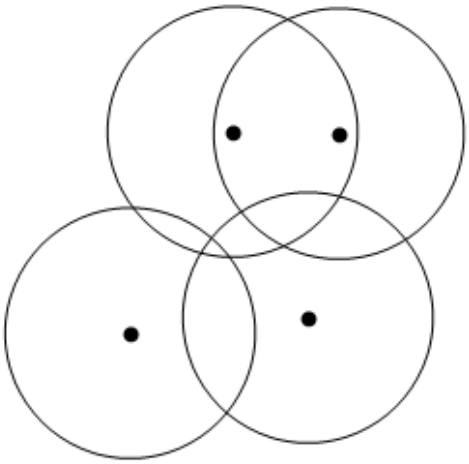


Figura 2.4: Exemplo com as auras associadas a cada *avatar*.

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 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 2.4 podemos ver um exemplo das auras associadas a cada *avatar*. Neste exemplo, apenas os dois *avatares* no topo da imagem trocam actualizações de estado.

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 2.5 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:

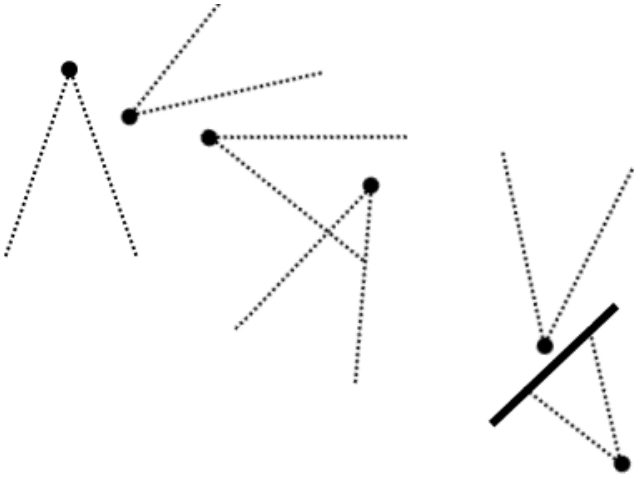


Figura 2.5: Exemplo com a linha de visão de cada *avatar*.

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][15] 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][15]:

Codificação usando o número de *bits* mínimo. Em vez de se representar as primitivas com a quan-

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

2.7.3 A^3

O A^3[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.

2.7.4 *Vector-Field Consistency*

O *Vector-Field Consistency*[14] 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 2.6). 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.

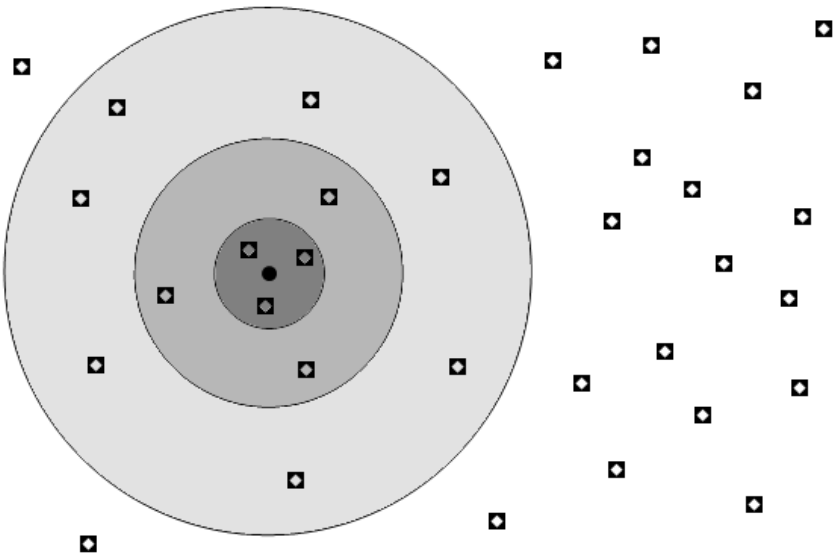


Figura 2.6: Exemplo das zonas de consistência do VFC relativas a um *avatar*.

Cada vector κ 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*[1] 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.

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

Capítulo 3

Arquitectura

Neste capítulo descrevemos em detalhe a arquitectura do nosso sistema. Começamos por apresentar o contexto que influenciou o desenho da nossa arquitectura. De seguida, descrevemos em detalhe o *Vector-Field Consistency*, seguido das nossas alterações. Concluímos este capítulo com a explicação do funcionamento e dos detalhes mais importantes da nossa arquitectura.

3.1 Introdução

Um jogo *online* multi-jogador na categoria dos *First Person Shooters* (FPS) requer que a visão que cada jogador tem das posições, movimentos e acções dos restantes jogadores corresponda oportunamente à realidade local de cada um desses jogadores. Isto requer comunicação muito frequente, na ordem das dezenas de pacotes por segundo, de forma a manter o estado local de cada jogador actualizado e consistente entre todos os jogadores.

Numa arquitectura Cliente/Servidor (C/S), o ponto de estrangulamento situa-se na largura de banda de saída do servidor. O tráfego de saída aumenta de forma quadrática com o aumento do número de clientes. Este facto é o principal limitador do número total de jogadores suportado. Contudo, como foi visto no capítulo 2, este tráfego de saída pode ser reduzido de várias formas. Uma delas consiste na gestão de interesse.

Como base do nosso sistema, decidimos usar o *Vector-Field Consistency*[14](VFC). Escolhemos o VFC pela sua flexibilidade e por permitir reduzir a consistência progressivamente com o aumento da distância. A flexibilidade do VFC manifesta-se na diversidade dos parâmetros que podem ser utilizados para controlar a consistência e pela liberdade com que podemos especificar as zonas de consistência. Com os parâmetros adequados, podemos usar o VFC para manter a consistência forte para todos os objectos ou então podemos usar o VFC para implementar um sistema com auras.

No entanto, apesar da flexibilidade do VFC, acreditamos que ele pode ser melhorado para satisfazer melhor o contexto dos FPS. Isto porque o VFC aplica os parâmetros de consistência de forma igual para os 360º à volta de cada *avatar*. Contudo, a visibilidade de um *avatar* está limitada ao seu campo de visão que constitui apenas uma parte dos 360º. Desta forma, complementámos o sistema original do VFC com o campo de visão (FoV - *Field of View*).

A introdução do campo de visão permite reforçar os requisitos de consistência no ângulo de visão do *avatar* (tipicamente 90º) ao mesmo tempo que permite relaxar a consistência para os objectos "atrás" do *avatar*. O problema causado pelo aumento do alcance de visão através do uso do *zoom* fica abrangido uma vez que quando o *zoom* aumenta, o campo de visão diminui.

Devido à adaptação do VFC ao contexto dos FPS, denominámos o nosso sistema de VFC4FPS (*Vector-Field Consistency for First Person Shooters*).

3.1.1 Escolha do Jogo

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

- Inteiramente implementado em *C++*. Isto é importante uma vez que o sistema foi implementado sob forma de uma biblioteca em *C#* e assim a interface entre o jogo e o sistema pode ser feita facilmente.
- Popular, tendo em conta o grande número de servidores disponíveis e a elevada percentagem de utilização dos mesmos.
- Inclui mapas de grande dimensão, complementados por mapas que podem ser obtidos em *sites* da comunidade.
- Proporciona um sistema de criação e edição de mapas dentro do jogo de forma fácil.

Entre os FPS *open source* considerados encontram-se o *Warsow*[1], *Nexuiz*[2], *Tremulous*[3], *Quake 3*[4], *Blood Frontier*[5] e *AssaultCube*[6].

Nenhum destes jogos foi escolhido em vez do *Cube 2: Sauerbraten*, principalmente por a maioria ser implementada em *C*. Adicionalmente, razões como falta de popularidade, conterem mapas de pequena dimensão ou estarem em fase beta no desenvolvimento, contribuíram para a sua exclusão.

Foram também considerados jogos de *Real-time strategy* (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 é baixa. Outro problema prende-se com a própria natureza destes jogos, em que a lógica de jogo limita o número de jogadores. Limite esse tipicamente abaixo da dezena.

3.2 *Vector-Field Consistency*

O VFC foi originalmente desenhado para jogos multi-jogador em redes *ad-hoc*. Jogos estes para plataformas móveis em que os recursos computacionais, de memória e de rede são reduzidos. O objectivo era reduzir a largura de banda utilizada pelos clientes e servidor.

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

O VFC distingue-se de outros gestores de interesse através da introdução de um conceito de múltiplas zonas circulares concêntricas, centradas num *pivot*. Cada zona tem associado um nível de consistência. Objectos dentro da mesma zona estão sujeitos ao mesmo grau de consistência. Esse nível de consistência vai diminuindo à medida que as zonas se afastam do *pivot*. Os níveis de consistência baixam de forma monótona à medida que a distância ao *pivot* aumenta. Assim, é possível oferecer uma redução progressiva da consistência. Desta forma, objectos próximos do *pivot* estão mais consistentes do que objectos mais afastados.

No VFC os objectos estão dispostos num mundo virtual de N dimensões. Isto permite um mapeamento directo com o mundo virtual dos jogos. O mundo virtual é constituído por objectos, os quais podem ser *avatares*, armas, munições e outros. Cada cliente tem uma visão local deste mundo virtual, visão esta que pode conter objectos com algum grau de inconsistência em relação às respectivas réplicas primárias. A consistência dos objectos em cada vista dependem da distância ao *pivot*. No caso particular deste sistema, o *pivot* corresponde ao *avatar* do jogador e os objectos correspondem aos *avatares* dos outros jogadores.

Os graus de consistência de cada zona são definidos através de vectores tridimensionais, chamados de vectores de consistência (κ). Estes vectores κ limitam a divergência máxima dos objectos em cada zona. As três dimensões que limitam a divergência são as seguintes:

- Tempo (θ): Especifica o tempo máximo (em segundos) que uma réplica tolera sem ser actualizada. Este valor garante que a inconsistência da réplica está limitada a κ_θ segundos em relação à respectiva réplica primária.
- Sequência (σ): Especifica o número máximo de actualizações que pode ignorar até necessitar de uma nova actualização. Limita a inconsistência da réplica a κ_σ actualizações atrasadas em relação à sua réplica primária.
- Valor (ν): Especifica a diferença máxima entre conteúdos das réplicas. O conteúdo considerado depende da implementação e a diferença é medida em percentagem. Garante que uma réplica tem o seu valor com uma diferença máxima de $\kappa_\nu\%$ em relação ao valor da réplica primária.

É possível ignorar qualquer uma das dimensões, especificando um valor infinito. Por exemplo, considerando um vector $\kappa = [1, \infty, 50]$, os objectos afectados por ele estão no máximo 1 segundo desactualizado ou com o conteúdo com uma variação de 50% em relação à sua réplica primária. Neste exemplo, o número de actualizações não é um factor limitador da divergência.

Uma vista (ϕ) é definida por um conjunto de parâmetros:

- Um parâmetro **O** que consiste num vector que contém os objectos abrangidos pela vista.
- Um parâmetro **Z** que consiste num vector que especifica os raios de alcance que cada zona de consistência. Um valor infinito representa uma zona de raio infinito e está normalmente associado à zona mais afastada do *pivot*.
- Um parâmetro **C** que corresponde a um vector que associa cada vector κ à respectiva zona.
- Um parâmetro **V** que consiste num vector que identifica os *pivots* da vista.

3.2.1 Generalizações

O VFC oferece duas generalizações para uma utilização mais abrangente: *multi-pivot* e *multi-zones*. A generalização *multi-pivot* permite a definição de múltiplos *pivots* para a mesma vista. Neste esquema, a consistência dos objectos é determinada em relação ao *pivot* mais próximo.

A generalização *multi-zones* permite que diferentes conjuntos de objectos com requisitos de consistência diferentes possam ter associadas zonas diferentes. Um exemplo onde poderia ser útil num jogo FPS, seria na utilização de requisitos de consistência diferentes para colegas de equipa e para adversários. Uma vez que o objectivo é disparar sobre os adversários, a consistência dos *avatares* dos adversários é mais importante que a dos colegas de equipa.

3.2.2 Arquitectura do VFC

O VFC foi desenhado para ser utilizado como uma biblioteca (figura 3.1). Desta forma o programador do jogo fica liberto dos detalhes da comunicação. Através da API disponibilizada pela biblioteca, o programador pode parametrizar os requisitos de consistência de acordo com a semântica do jogo. O VFC utiliza uma arquitectura Cliente/Servidor. O servidor é o ponto mais consistente e é ele que faz a gestão da consistência dos clientes com base no VFC.

Os clientes possuem réplicas de todos os objectos partilhados (no Repositório Secundário) e o servidor possui as réplicas primárias (no Repositório Primário). Os clientes são livres de ler as réplicas locais, mas quando fazem escritas nas suas réplicas locais, essas actualizações são enviadas para o servidor. O servidor fica então responsável por actualizar as réplicas dos outros clientes consoante as vistas que definem os requisitos de consistência de cada cliente.

As actualizações dos clientes e servidor são periódicas, mas independentes entre eles (não existe sincronismo entre cliente e servidor). Quando um cliente faz uma escrita numa réplica, a actualização da respectiva réplica primária não é feita de imediato, mas sim quando o intervalo chega ao fim (ronda). O servidor faz a gestão da consistência dos clientes a cada ronda, determinando quais as réplicas a actualizar em cada cliente, isto para as réplicas que recebeu actualizações desde a última ronda. A actualização

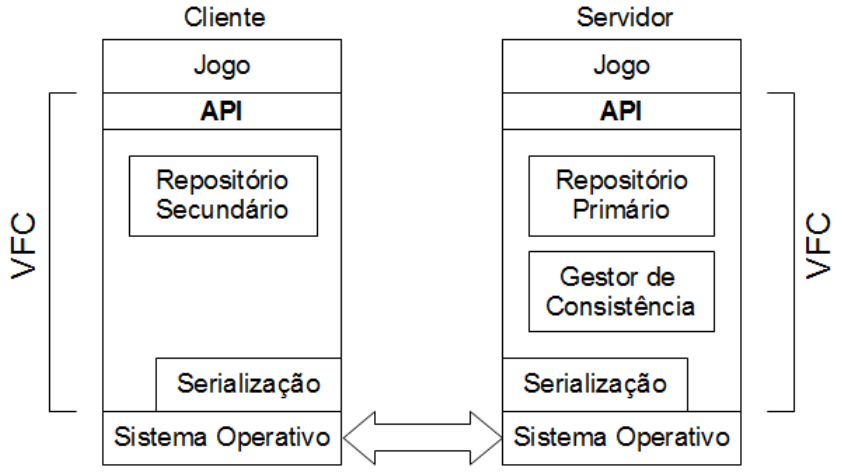


Figura 3.1: Arquitectura do VFC.

das réplicas é feita periodicamente uma vez que os objectos que elas representam são escritos muito frequentemente e isso implicaria demasiada comunicação.

O algoritmo do Gestor de Consistência, no servidor, é constituído por duas fases: fase de configuração e fase activa. A fase de configuração é a altura em que os clientes se ligam ao servidor, registam os objectos que querem partilhar e definem os parâmetros da sua vista. A fase activa é a fase em que o jogo decorre e é quando as actualizações dos objectos são enviadas em cada ronda periódica pelos clientes e pelo servidor.

A gestão da consistência é realizada, no servidor, através de duas funções: *update-received* e *round-triggered*. A primeira lida com a recepção de actualizações de réplicas, causadas por escritas nos clientes. A segunda, determina a cada ronda, de acordo com as vistas de cada cliente, quais as réplicas a serem enviadas para cada cliente. Os objectos são enviados entre cliente e servidor de forma serializada.

Como suporte à gestão da consistência, o servidor requer as seguintes estruturas de dados:

Tabela de vistas (TV) Consiste numa tabela que guarda, para cada cliente, a vista que define os seus requisitos de consistência.

Matriz de estado da consistência (MEC) Consiste numa matriz que guarda para cada cliente, para cada objecto, qual o seu estado actual de consistência. O estado de cada objecto contém o último instante temporal em que foi enviado ao respectivo cliente, o número de actualizações que foram recebidas e que não foram propagadas e o valor do objecto da última vez que foi enviado ao cliente.

update-received Função executada a cada recepção de actualização de objectos. Na recepção de uma actualização é incrementado o número de actualizações na **MEC** para o objecto a que a actualização diz respeito, para todos os clientes.

round-triggered A cada execução, é determinado para cada cliente que objectos é que atingiram os limites de consistência definidos pela sua vista e que consequentemente devem ser enviados para o cliente. Esta determinação é feita para cada objecto de cada cliente através dos seguintes passos:

1. O objecto é inspeccionado para determinar se foi actualizado desde o último envio, ou seja, se o número de actualizações na estrutura **MEC** é superior a zero.
2. Caso tenha sido actualizado, é calculada a distância do objecto ao *pivot* definido na **TV** do cliente a ser considerado; no caso de existir mais que um, é usado o *pivot* que lhe está mais próximo.
3. Usando a distância ao *pivot*, é determinada a zona de consistência a que pertence o objecto e obtido o respectivo vector κ.
4. É então comparado o **MEC** do objecto com o vector κ da zona em que se situa, nomeadamente se o tempo desde a última actualização, o número de actualizações ou o valor ultrapassou o limite.
5. Se algum destes parâmetros tiver sido ultrapassado, o objecto é considerado para ser enviado ao cliente e os valores do **MEC** são actualizados (o último tempo de envio é actualizado para o instante de tempo actual, o número de actualizações é posto a zero e o valor actual do objecto é guardado).

3.3 *VFC for First Person Shooters* (VFC4FPS)

Como referido anteriormente, no VFC cada zona circular concêntrica tem um vector κ que especifica a divergência máxima para essa zona. De forma a incorporar o campo de visão, o VFC4FPS adiciona um novo parâmetro à vista, denominado daqui para a frente por **F**. O parâmetro **F** consiste num vector similar ao vector **Z** e especifica os ângulos dos campos de visão. O último valor do vector **F** tem de corresponder a 360º de forma a completar os outros ângulos. Este ângulo está associado ao vector de orientação do *pivot*.

O parâmetro **C** passa a ser uma matriz bidimensional indexada por zona e campo de visão. Desta forma é possível ter mais que um vector κ por zona, ou seja, podemos ter um vector κ por ângulo de visão.

Devido à possibilidade dos clientes entrarem e saírem a qualquer altura num servidor de um FPS *online* multi-jogador, deixa de haver a fase de configuração do VFC. Isto porque seria difícil manter o parâmetro **O** da vista de cada cliente actualizado, pelo que o nosso sistema não o implementa e assume que apenas existe uma vista por jogador e que ela abrange todos os objectos em jogo. Elimina-se assim a generalização *multi-zones* do VFC. No entanto, é algo que não afecta negativamente este jogo em particular e possivelmente a generalidade dos FPS.

Os vectores κ dentro de uma mesma zona reduzem a consistência monotonamente à medida que o ângulo de visão aumenta, similarmente à redução entre zonas.

Como num FPS a granularidade do tempo é muito reduzida, a dimensão tempo dos vectores κ é alterada para especificar o tempo em milissegundos.

Na tabela 3.1 podemos ver o exemplo de uma vista com três campos de visão, três zonas de consistência e os respectivos vectores κ. A figura 3.2 demonstra visualmente como é que esta vista é aplicada a um *pivot* tendo em conta a sua orientação.

3.3.1 Arquitectura do VFC4FPS

O VFC4FPS mantém a arquitectura Cliente/Servidor utilizada pelo VFC, assim como a delegação de toda a funcionalidade na forma de biblioteca. Como se pode observar pela figura 3.3, a arquitectura VFC4FPS mantém a arquitectura do VFC, adicionando um módulo de compressão. Este módulo tem a funcionalidade de reduzir a dimensão dos objectos depois de serializados.

Em termos de funcionalidade do Gestor de Consistência, o algoritmo da função *round-triggered* foi modificado para ter em conta os campos de visão. Esta modificação é feita na altura em que se determina qual a zona de consistência em que um objecto se encontra. Adicionalmente a esta determinação é

FoV \ Zonas	150	250	∞
90º	[30, 1, 5%]	[60, 2, 10%]	[90, 3, 15%]
150º	[45, 2, 10%]	[75, 3, 15%]	[105, 4, 20%]
360º	[90, 3, 15%]	[120, 4, 20%]	[200, 5, 25%]

Tabela 3.1: Exemplo de uma vista.

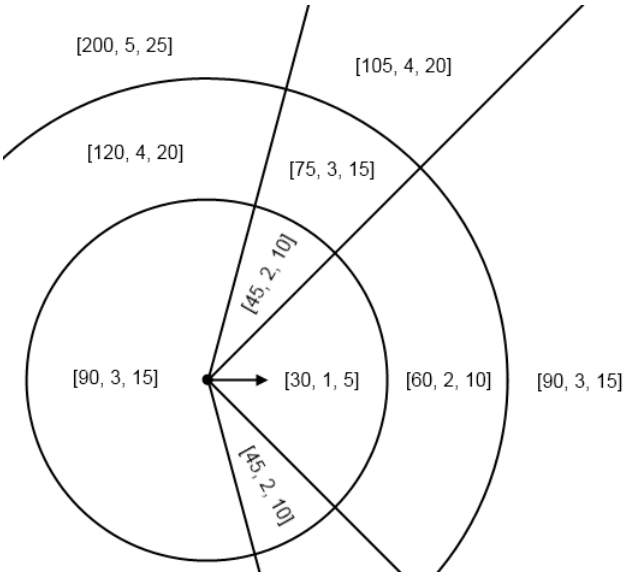


Figura 3.2: Representação visual de uma vista aplicada a um pivot.

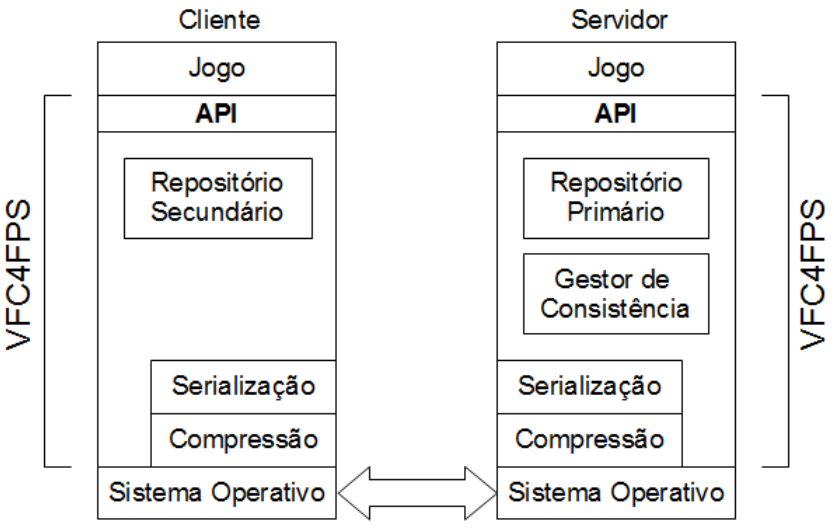


Figura 3.3: Arquitectura do VFC4FPS.

adicionado o cálculo do campo de visão em que o objecto se encontra, em relação à orientação do respectivo *pivot*.A obtenção do vector κ passa a ser baseada pela zona e pelo campo de visão.

Começando pelo lado do cliente, o cliente do jogo (designado daqui para a frente por Cube-Client) liga-se através da API ao cliente do VFC4FPS (designado daqui para a frente por VFC4FPS-Client). O servidor divide-se em duas partes, o servidor do jogo (designado daqui para a frente por Cube-Server) que consiste no servidor original do *Cube 2: Sauerbraten* e o servidor da biblioteca VFC4FPS (designado daqui para a frente por VFC4FPS-Server). A ligação interna entre o Cube-Server e o VFC4FPS-Server é feita através da API do VFC4FPS.

3.3.2 Modelo de Comunicação do *Cube 2: Sauerbraten*

A comunicação no *Cube 2: Sauerbraten* pode ser de dois tipos, actualização de objectos e eventos. A actualização de objectos consiste na actualização dos objectos que representam o estado dos *avatares*. Estas actualizações são periódicas. Os eventos correspondem a mensagens que alteram o estado global do jogo, são mensagens que são desencadeadas pela interacção com o mundo virtual. Estas mensagens são imediatamente enviadas.

No *Cube 2: Sauerbraten* os únicos objectos que são partilhados são os *avatares*. Objectos como munições, armas, saúde e armadura são controlados através de eventos. O estado dos *avatares* pode ser mapeado directamente num objecto para ser utilizado pelo VFC4FPS. A natureza periódica utilizada no *Cube 2: Sauerbraten* para a transmissão destes objectos adequa-se perfeitamente ao funcionamento das rondas do VFC4FPS.

Por outro lado, os eventos não contêm estado, mas causam alterações ao estado global. A razão para este facto deve-se à grande dimensão do estado global do jogo. É mais eficiente enviar alterações pontuais ao estado do que enviar o estado completo com apenas uma pequena alteração. Assim sendo, não é possível reduzir a frequência de envio de eventos, podendo apenas ser filtrados. Contudo, se os eventos forem filtrados, a transição de estado perde-se nos clientes para quem o evento foi filtrado. Isto causa divergências entre o estado global de cada cliente.

O exemplo de um evento com este problema são os eventos de mudança de arma por parte de um *avatar* adversário. Quando o *avatar* está longe ou fora de visibilidade de um cliente, não interessa obter a mensagem do evento da sua mudança de arma. O problema surge se o mesmo *avatar* passar a estar perto e na visibilidade do cliente, sendo que neste caso o cliente vai ver o *avatar* a empunhar a arma errada.

3.3.3 Entrada de um Cliente

Quando um cliente é iniciado, juntamente com o Cube-Client é iniciada uma instância do VFC4FPS-Client. O estabelecimento de uma conexão Cliente-Servidor é feita como no jogo original entre o Cube-Client e o Cube-Server. Paralelamente é estabelecida uma conexão entre o VFC4FPS-Client e o VFC4FPS-Server.

Após uma conexão bem sucedida, o objecto que representa o *avatar* do Cube-Client, é registado no VFC4FPS-Client. Este registo é complementado pela especificação de uma vista que define os requisitos de consistência do cliente. De forma ao registo ficar completo, o *avatar* controlado pelo Cube-Client é

definido como *pivot*. Seguidamente, tanto a vista como o objecto são enviados para o VFC4FPS-Server. A partir deste momento, o VFC4FPS-Client começa a actualização periódica do seu objecto para o VFC4FPS-Server.

3.3.4 Actualizações num Cliente

O Cube-Client processa o seu estado (*frames*) várias vezes a cada segundo (tipicamente 60 *frames* por segundo). No processamento de cada *frame*, devido às interacções no mundo virtual, o estado do seu *avatar* é alterado. Estas alterações são aplicadas no respectivo objecto situado no Repositório Secundário do VFC4FPS-Client.

Periodicamente, e independentemente do Cube-Client, o VFC4FPS-Client inspecciona os seus objectos e determina se houve alterações desde a última ronda. Se houver alterações, os objectos são serializados, comprimidos, e subsequentemente enviados para o VFC4FPS-Server. Esta verificação serve para evitar enviar informação repetida para o servidor.

No processamento do estado a cada *frame*, além do processamento dos objectos locais, é necessário processar os objectos dos outros clientes. Para esse efeito, o Cube-Client lê o estado das réplicas dos objectos que estão no Repositório Secundário do VFC4FPS-Client. A consistência destes objectos é mantida pelo servidor, de acordo com a vista do cliente, e é controlada efectivamente pela frequência com que o servidor propaga as alterações.

3.3.5 Actualização no Servidor

A função do VFC4FPS-Server resume-se principalmente à execução da função *update-received* para cada actualização recebida e à execução periódica da função *round-triggered*. Como foi referido anteriormente, os eventos são geridos pelo jogo, o que significa que o Cube-Server tem como função gerir os eventos e o estado global do jogo.

No entanto, existem eventos que podem ser filtrados. Um deles é o evento de efeito de tiro, que se limita a um efeito gráfico (existe um outro evento para quando um tiro causa dano). Este evento pode ser filtrado para os clientes que não conseguem ver o caminho do tiro ou que estejam suficientemente longe. Esta filtragem é útil para as armas de disparo rápido que geram muitos eventos individuais. O outro consiste em eventos de efeitos sonoros, estes eventos têm uma localidade espacial associada e podem ser filtrados para os clientes que estão demasiado longe para os ouvir.

Esta filtragem é feita recorrendo ao estado contido no Repositório Primário do VFC4FPS-Server. O Cube-Server, para estes eventos, questiona o VFC4FPS-Server, de acordo com os parâmetros adequados (explicado em detalhe no capítulo 4), quais os clientes a que o evento é relevante. Desta forma os clientes para que o evento não é relevante não são considerados para receber o evento.

3.3.6 Alteração de uma Vista

No VFC4FPS é possível alterar qualquer parâmetro de uma vista. Isto é particularmente útil no caso da utilização do *zoom* por parte dos jogadores. Quando um jogador activa o *zoom*, os campos de visão na sua vista são alterados para reflectirem o novo campo de visão do *avatar*. O alcance das zonas de

consistência é aumentado de forma proporcional à redução do campo de visão. O inverso é executado quando o *zoom* é desactivado.

Algo que também pode ser feito é a alteração do *pivot* de uma vista. Os FPS oferecem uma funcionalidade a que se chama modo de espectador. Neste modo o cliente não controla nenhum *avatar*, pode, no entanto seguir o movimentos dos *avatares* em jogo. É dada a liberdade de trocar de *avatar*. Como suporte a esta característica, o *pivot* da vista pode ser alterado para reflectir o *avatar* que se está a seguir.

Estas alterações são feitas pelo Cube-Client e propagadas para o VFC4FPS-Server pelo VFC4FPS-Client. Para uma gestão de consistência óptima, estas alterações são enviadas de forma imediata.

Capítulo 4

Implementação

Neste capítulo descrevemos os detalhes mais importantes sobre como o nosso sistema foi implementado. Começamos por apresentar o ambiente usado para o desenvolvimento. De seguida, descrevemos a API disponibilizada pelo sistema, as estruturas de dados e os protocolos de comunicação utilizados. Concluímos este capítulo, com a apresentação das interfaces gráficas utilizadas para configuração e monitorização, e como os *bots* foram alterados para simular jogadores reais.

4.1 Ambiente de Desenvolvimento

O VFC4FPS foi desenvolvido sob forma de biblioteca. A biblioteca foi implementada em em C# na plataforma *.Net* da *Microsoft*[1]. O *Cube 2: Sauerbraten* está implementado em C++. A utilização da funcionalidade do VFC4FPS por parte do jogo é feita com recurso à API da biblioteca. De forma a simplificar a implementação e aproveitar a interoperabilidade oferecida pela plataforma *.Net*, o código C++ do jogo foi compilado como *managed* C++. Desta forma a interface entre jogo e biblioteca pode ser feita directamente.

4.2 API - *Application Programming Interface*

A API faz a fronteira entre o jogo e a funcionalidade do VFC4FPS. A API especifica a interface que é disponibilizada às aplicações (neste caso, jogo) de forma a poderem utilizar a funcionalidade oferecida pelo VFC4FPS.

A seguir, apresentam-se as interfaces que os objectos partilhados têm de respeitar para poderem ser lidos pelo VFC4FPS. No fim, apresentam-se as funções que constituem a API do VFC4FPS, tanto para o cliente do VFC4FPS como para o servidor do VFC4FPS.

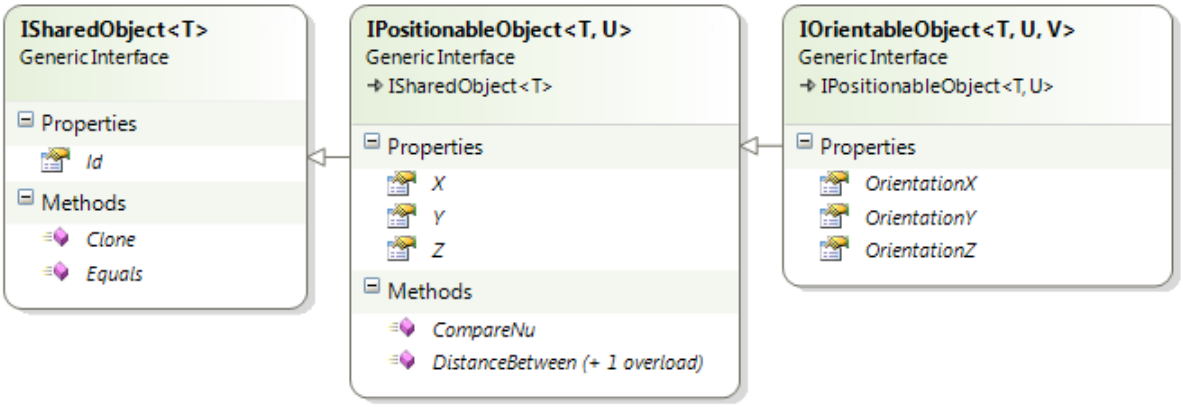


Figura 4.1: Hierarquia de interfaces para objectos partilhados.

4.2.1 Interfaces dos Objectos Partilhados

Nem todos os objectos de um jogo têm as mesmas características. Por essa razão o VFC4FPS oferece uma hierarquia de interfaces que os objectos podem implementar (figura 4.1). Em primeiro lugar temos a interface *ISharedObject*. Esta interface é útil para objectos sem localidade espacial, ou seja, objectos que representam o estado global do jogo. Assim, objectos que implementem esta interface requerem consistência forte e não usufruem da redução de consistência do VFC4FPS. A interface *ISharedObject* contém um campo que identifica o objecto (Id) e métodos que permitem clonar a instância do objecto (*Clone*) e determinar se duas instâncias são iguais (*Equals*).

A seguir na hierarquia temos a interface *IPositionableObject*. Esta interface aplica-se a objectos que tenham localidade espacial. Esta interface contém três campos para especificar a posição X, Y e Z no mundo virtual. Contém ainda um método para calcular a distância entre dois objectos (*DistanceBetween*) e outro para comparar a variação do parâmetro valor (*CompareNu*) especificado pelos vectores κ. A implementação do método *CompareNu* especifica como é que o valor é calculado.

Por fim, temos a interface *IOrientableObject*. Esta interface adiciona os campos que definem a orientação do objecto (*OrientationX*, *OrientationY*, *OrientationZ*). É através da orientação que é possível utilizar os campos de visão utilizados pelo VFC4FPS. A orientação tem de estar na forma de um vector normalizado. Esta interface é aplicável a objectos que tenham uma visão limitada do mundo virtual.

Apesar dos vários tipos de interfaces disponibilizadas, no contexto do *Cube 2: Sauerbraten* apenas é necessário utilizar a interface *IOrientableObject* para implementar os objectos com o estado dos *avatares*.

De forma a permitir utilizar os tipos primitivos mais adequados para representar os campos das interfaces, as interfaces foram especificadas com recurso aos *generics* de C#. Assim é possível especificar de forma independente, o tipo primitivo para o identificador do *ISharedObject*, o tipo primitivo para as coordenadas do *IPositionableObject* e o tipo primitivo para a orientação do *IOrientableObject*. Desta forma reduzimos a dimensão dos objectos e por sua vez, a quantidade de tráfego gerado.

No entanto, de forma a permitir o uso completamente genérico, todos os métodos que lidam com uma interface genérica precisariam por sua vez de estarem implementados de forma genérica. Na prática, por uma questão de simplicidade e por existir apenas um tipo de objectos partilhados no *Cube 2: Sauerbraten*,

[1] http://www.microsoft.com/net/

os métodos foram implementados para a única assinatura da interface *IOrientableObject*. Este facto reduz a abstracção do VFC4FPS como biblioteca. Contudo, sendo esse um objectivo secundário, como a comunicação de eventos continua a ser processada pelo jogo original, e o VFC4FPS estar limitado à plataforma *.Net*, é uma limitação aceite.

4.2.2 API do Cliente VFC4FPS

Nesta secção descrevem-se os métodos que fazem a interface entre o cliente do jogo e o cliente da biblioteca VFC4FPS.

AddObject Adiciona um novo objecto ao Repositório (Secundário) com o objectivo de ser partilhado entre todos os clientes (e servidor). O objecto é enviado imediatamente para o servidor onde passa a ser considerado no sistema de rondas. O objecto tem de ser uma instância de uma classe derivada de uma das interfaces especificadas na secção 4.2.1.

ConnectToServer Estabelece uma conexão entre o cliente VFC4FPS e o servidor VFC4FPS num endereço e porto especificado. No caso de uma conexão bem sucedida, devolve um identificador de cliente. A partir deste momento a rotina periódica que envia os objectos actualizados ao servidor é inicializada.

DelObject Remove um objecto que seja propriedade do cliente. É enviado um pedido imediato ao servidor para que o objecto seja removido do Repositório Principal.

DisconnectFromServer Desconecta o cliente VFC4FPS do servidor VFC4FPS e limpa o estado do cliente, voltando efectivamente ao estado inicial.

GetObjectRef Obtém a referência para um objecto que corresponda ao identificador introduzido. As alterações aos objectos são feitas directamente através da referência para o Repositório Secundário, não sendo necessário efectuar actualizações explícitas através de *commit*.

GetPhi Permite obter a vista do cliente para a sua manipulação, como por exemplo alterar o alcance das zonas de consistência, os ângulos para o campo de visão, o *pivot* ou qualquer outro parâmetro.

GetUpdatedIDs Obtém os identificadores dos objectos que foram actualizados a partir do servidor, desde a última invocação desta função. Serve para evitar processamento desnecessário no cliente do jogo, uma vez que indica quais os objectos que sofreram actualizações desde o último processamento do estado do jogo.

SetClientID Função opcional que permite alterar o identificador do cliente.

SetPhi Adiciona ou actualiza a vista do cliente e propaga imediatamente a vista para o servidor.

IsConnected Obtém o estado da conexão do cliente VFC4FPS com o servidor VFC4FPS.

4.2.3 API do Servidor VFC4FPS

Nesta secção descrevem-se os métodos que fazem a interface entre o servidor do jogo e o servidor da biblioteca VFC4FPS.

Init Inicializa o servidor do VFC4FPS num porto especificado.

ClientIDsAffectedByPositionableEvent É através desta função que o servidor do jogo procede à filtragem de eventos. Possui duas implementações, uma que recebe um vector com a posição pontual do evento e o raio máximo em que esse evento é válido. Outra que recebe dois vectores de posição, útil para representar segmentos de recta, e o índice máximo de zona de consistência em que esse evento é válido. Através da informação recebida, são determinados quais os clientes a que o evento tem relevância, ou seja, que estejam dentro dos limites máximos especificados. Devolve os identificadores dos clientes a que o evento é relevante.

No caso do *Cube 2: Sauerbraten*, foram filtrados dois eventos. Um dos eventos consiste em efeitos de som com localidade espacial (como por exemplo o som de um *avatar* a efectuar um salto). Isto porque um cliente apenas consegue ouvir sons até um raio máximo em torno do *avatar*.

O outro evento consiste nos efeitos de tiro, caracterizado por duas posições: origem e destino do tiro. Devido à complexidade computacional para determinar se um segmento de recta é visível num determinado campo de visão. E como os efeitos de tiro são muito frequentes. A filtragem é feita usando apenas os pontos de origem e destino do tiro. Isto significa que um evento de tiro só é relevante para clientes cuja origem ou destino do tiro estejam dentro da zona de consistência especificada na chamada da função.

GetObject Obtém uma cópia do objecto pretendido, qualquer alteração ao seu estado não é propagada para o Repositório Principal.

4.3 Estruturas de Dados

Como base à funcionalidade da biblioteca VFC4FPS são necessária estruturas de dados adequadas. As estruturas de dados principais estão encapsuladas em classes. De seguida apresenta-se a função dessas classes e como são constituídas.

4.3.1 *ObjectPool*

Esta classe corresponde à implementação dos Repositórios Principal e Secundário referidos na secção 3.2.2. É constituída por dois dicionários. Um dicionário guarda instâncias de *ISharedObject*. O outro guarda os identificadores do cliente que é proprietário de cada *ISharedObject*. Ambos os dicionários são indexados pelo identificador dos objectos. A escolha em usar dicionários prende-se ao seu bom desempenho na procura de informação uma vez que as consultas de objectos são muito frequentes.

4.3.2 *ClientStateData*

Esta classe é responsável por guardar o estado actual da consistência de cada *ISharedObject* de cada cliente. Para esse efeito é constituída por três dicionários indexados pelo identificador do cliente. Sendo que cada um dos três dicionários contém um dicionário por cliente que, indexados pelo identificador do objecto contém o instante temporal da última vez que o objecto foi enviado, o número de actualizações recebidas desde o último envio, e uma cópia da instância do objecto da última vez que foi enviado (para mais tarde fazer a comparação do parâmetro valor dos vectores κ).

4.3.3 *CubeOriObj*

Esta classe corresponde à implementação da interface *IOrientableObject* para o objecto que representa o *avatar* do jogo. Adicionalmente aos campos requeridos pela interface, contém campos para representar a orientação do *avatar* sob a forma de ângulos de Euler (*yaw*, *pitch*), velocidade de queda nos três eixos (*fallingx*, *fallingy*, *fallingz*), velocidade do movimento nos três eixos (*velx*, *vely*, *velz*), estado físico (*physstate*), *flags* e arma seleccionada (*gunselect*).

A informação de orientação nos três eixos (necessária para a determinação dos campos de visão) é determinada através de funções de trigonometria aplicadas de forma adequada aos campos *yaw* e *pitch*. A implementação do método *CompareNu* consiste no uso da função *DistanceBetween* e na conversão do resultado em percentagem. Desta forma, o parâmetro valor limita a divergência com base na variação da distância desde a última actualização do objecto. A implementação desta classe está feita do lado da biblioteca e não no código do jogo devido à utilização de *generics* e por uma questão de simplicidade.

4.3.4 *KVec*

Esta classe corresponde à implementação dos vectores κ. É constituída por dois inteiros, para representar o tempo (*theta*) e a sequência (*sigma*), e por um *float*, para representar o valor (*nu*). Para representar um valor infinito para qualquer um dos parâmetros, utiliza-se o valor (-1).

4.3.5 *Phi*

Esta classe corresponde à implementação das vistas. É constituída por dois *arrays* de inteiros, um para a especificação das dimensões das zonas e outro para os ângulos de visão. Contém também um *array* bidimensional de instâncias de *KVec*. Este *array* é indexado por zona e campo de visão. Contém ainda uma lista de inteiros para representar os *pivots* da vista. Cada inteiro corresponde ao identificador de um objecto.

Possui um método (*ShouldBeSentToClient*) para determinar, consoante a vista, se um objecto excedeu o seu limite de consistência e consequentemente deve ser enviado. Para esse efeito usa como base as instâncias do objecto actual e de quando foi enviado a última vez, o último instante temporal a que foi enviado e o número de actualizações entretanto recebidas. Este método começa por comparar o posicionamento do objecto em relação ao *pivot* da vista, ou seja, em que zona e campo de visão se encontra. A zona é determinada com auxílio da função *DistanceBetween*. Para determinar o campo de visão, primeiro é necessário calcular o vector tridimensional entre o *pivot* e o objecto. De seguida, esse vector é normalizado e calculado o produto interno entre ele e a orientação (normalizada) do *pivot*. Através do resultado do produto interno, é então determinado qual o campo de visão em que o objecto se encontra em relação ao *pivot*. Depois de determinada a zona e o campo de visão, obtém-se o vector κ (*KVec*) e é verificado se algum dos três parâmetros foi excedido.

4.4 Comunicação

Como já foi referido, o *Cube 2: Sauerbraten* tem dois tipos de comunicação, eventos e actualizações de estado dos objectos. O *Cube 2: Sauerbraten* divide estes tipos de comunicação em dois canais: canal 0

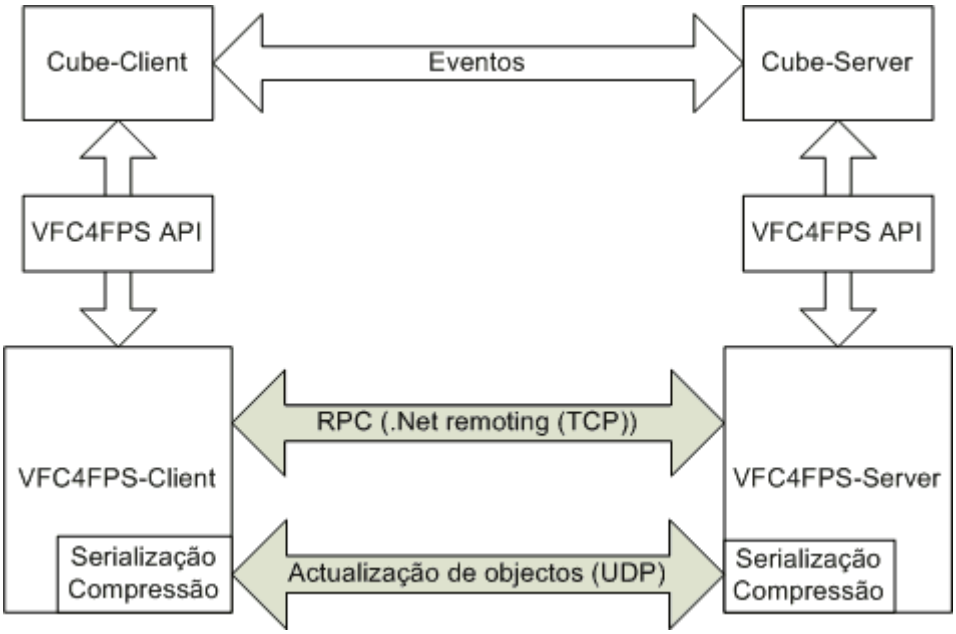


Figura 4.2: Arquitectura com a comunicação em detalhe.

(C0) para as actualizações de estado e canal 1 (C1) para os eventos.

No *Cube 2: Sauerbraten* original, um cliente envia os eventos ao servidor imediatamente após acontecerem. Já as actualizações de estado dos objectos por parte de um cliente são enviadas para o servidor periodicamente. O período utilizado pelo *Cube 2: Sauerbraten* é de 33 milissegundos. Adoptámos o período de 33 ms para as rondas no cliente e no servidor do VFC4FPS.

Como também já foi referido, não é possível integrar os eventos no VFC4FPS de forma independente e eficiente, pelo que eles continuam a ser geridos pelo sistema original do *Cube 2: Sauerbraten*. As actualizações do estado dos *avatares* passam a ser geridas e comunicadas via VFC4FPS. O estado dos *avatares* é mapeado para instâncias da classe *CubeOriObj*.

A comunicação entre o cliente e o servidor da biblioteca VFC4FPS é feita via *.Net remoting*. Segundo este método, a forma mais eficiente de troca de informação é através do protocolo TCP recorrendo à serialização binária. No entanto, a informação de controlo das chamadas remotas, a dimensão dos objectos serializados e o *overhead* do protocolo TCP geram demasiado tráfego de comunicação (em comparação com o jogo original). Que é agravado por invocações remotas muito frequentes.

Optou-se então por utilizar o *.Net remoting* apenas para os procedimentos menos frequentes, ou seja, para todos menos para enviar actualizações de objectos. Uma vez que o *Cube 2: Sauerbraten* utiliza o protocolo UDP para envio de actualizações de objectos, optou-se por fazer o mesmo no VFC4FPS. O protocolo UDP é simples e eficiente, mas sofre de possível perda de pacotes. Contudo, devido à grande frequência de envio de pacotes, se um pacote for perdido, não faz sentido reenvia-lo pois 33 ms a seguir é enviado outro pacote mais actualizado que o anterior.

O protocolo UDP foi implementado através de *sockets*. Os objectos são enviados serializados de forma binária. No entanto, uma instância (serializada) da classe *CubeOriObj* tem uma dimensão muito superior aos pacotes que continham a mesma informação no jogo original. Isto apesar do somatório da dimensão dos campos que constituem a classe terem uma dimensão aproximada (ligeiramente maior) dos pacotes

do jogo original. De forma a reduzir a dimensão dos objectos e ficarem comparáveis ao jogo original, depois de serializados, os objectos passam por um módulo de compressão e só então são enviados. Na secção 4.5 apresentamos como a compressão é feita e qual o formato dos pacotes. Na figura 4.2 podemos ver o diagrama do modelo de comunicação que foi apresentado.

4.4.1 Interface *.Net remoting*

Como foi referido, a interface entre os componentes cliente e servidor da biblioteca VFC4FPS comunicam entre si através de chamadas a procedimentos remotos (RPC - *Remote Procedure Call*). Estas chamadas são implementadas com recurso ao *.Net remoting*.

Um cliente VFC4FPS estabelece uma ligação com o servidor VFC4FPS através do procedimento *Connect*, para terminar a ligação utiliza o procedimento *Disconnect*. Para adicionar um objecto no servidor é utilizado o procedimento *AddObject*, para remover um objecto utiliza-se o procedimento *DelObject*. Para adicionar ou alterar uma vista utiliza-se o procedimento *SetPhi*. Caso haja necessidade de alterar o identificador de cliente atribuído pelo servidor, pode-se usar o procedimento *ChangeClientID*.

No sentido contrário, entre o servidor e o cliente, apenas existe um procedimento remoto (*Disconnect*) que serve para o servidor forçar a desconexão de um cliente.

4.5 Serialização e Compressão

Um objecto serializado contém uma grande quantidade de *metadata* em adição à dimensão dos campos. No caso da serialização de uma instância da classe *CubeOriObj*, a dimensão dos campos constitui apenas 11% da dimensão total. O sistema delta explicado na secção 4.5.1 permite evitar o envio da *metadata* de um objecto serializado, pois trata-se de informação imutável.

Apesar da filtragem da *metadata* de um objecto serializado, a dimensão total dos campos continuava ligeiramente superior à dimensão dos pacotes do jogo original. A causa deste facto deve-se ao sistema utilizado pelo jogo original para construir o pacote que continha o estado do objecto. Além de utilizar um sistema de redução do número de *bytes* de acordo com o valor presente em cada campo, alguns campos eram opcionais e podiam não ser enviados, isto é o caso dos campos que indicam a velocidade de queda do *avatar*, algo que só é necessário se o *avatar* estiver de facto em queda. De forma a conseguir igualar o desempenho do sistema utilizado pelo jogo original, depois de aplicado o sistema delta, é aplicada uma compressão baseada em mapa de *bits* (secção 4.5.2).

Um problema que surgiu no servidor, devido à grande quantidade de objectos e na frequência com que eram serializados, foi o tempo gasto a serializar objectos. Uma vez que o servidor não altera o estado dos objectos no Repositório Principal e, como os objectos são recebidos no servidor já serializados, para cada objecto é guardada a sua serialização recebida mais recentemente, evitando assim a serialização desnecessária quando se envia as actualizações aos clientes.

4.5.1 Sistema Delta

Na figura 4.3 podemos ver uma instância da classe *CubeOriObj* na sua forma serializada. Assinalado a cinzento está a região de *bytes* que corresponde à dimensão dos campos da classe. Através do texto *ASCII*

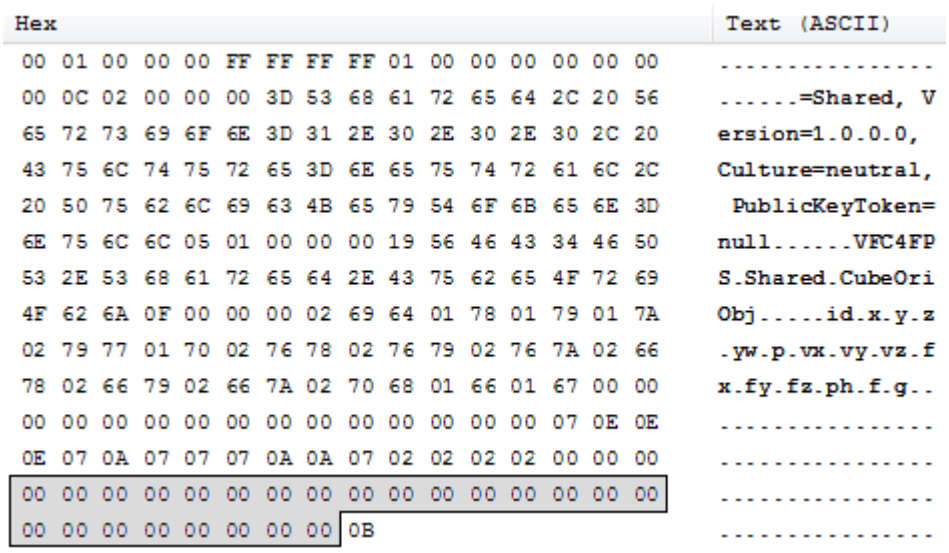


Figura 4.3: *CubeOriObj* serializado com os *bytes* dos campos destacados.

podemos observar a *metadata* da classe. Os únicos *bytes* que são modificados através da actualização da instância são os *bytes* na área cinzenta. Os restantes *bytes* não se modificam, mesmo entre diferentes instâncias da mesma classe.

Através da observação empírica dos *bytes* gerados por várias serializações binárias, foi deduzido que os campos são contíguos e se situam a um *offset* fixo do fim do *array* de *bytes*. Assim sendo, para determinar a dimensão total dos campos recorre-se à introspecção. Usando a dimensão obtida e excluindo o último *byte* do *array*, é calculado o *offset* onde começam os campos. Com estes dados, apenas é necessário enviar os *bytes* a partir do *offset* determinado até à sua dimensão total. A este pedaço de informação chamamos o delta do objecto serializado. Estes dados (*offset* de início, dimensão total dos campos e dimensão total do objecto serializado) são guardados em instâncias da classe *DeltaKey*. Esta classe contém ainda um *array* de *bytes* que contém a serialização da classe associada, necessário para reconstituir o objecto na recepção de um delta.

Para este sistema funcionar, é necessário que tanto o cliente como o servidor VFC4FPS possuam uma instância da classe *DeltaKey* para a classe cujo o delta se pretende transmitir. Uma vez que apenas o delta é transmitido, na sua recepção é necessário ter uma forma para identificar qual a instância *DeltaKey* a utilizar. Isto porque na recepção de um delta, é necessário obter uma serialização genérica da classe em questão, sobrepor o delta na posição correcta e proceder à sua deserialização.

A implementação completa deste sistema, de forma a identificar cada delta, utilizaria um *byte* extra para indexar a classe a que o delta pertence. Contudo, como no *Cube 2: Sauerbraten* só existe uma classe de objectos partilhada, todos os deltas dizem respeito à mesma classe. Isto permitiu uma implementação mais simples deste sistema por não ser necessária a utilização do *byte* de indexação.

Há três problemas com este sistema; ele assume a utilização de campos de dimensão fixa e por isso não funciona com campos do tipo *string* ou *array*; o posicionamento dos campos foi determinado empiricamente, o que pode causar problemas não antecipados; requer que os campos sejam declarados como públicos para serem considerados pela introspecção. Apesar destes problemas, devido à natureza dos dados partilhados neste jogo não causarem problemas e devido aos ganhos com este sistema, ele continuou a ser utilizado.

4.5.2 Compressão Mapa de *bits*

A compressão através de mapa de *bits* (*bitmap*) tem o objectivo de eliminar *bytes* inúteis, neste caso, *bytes* nulos. Este método de comprimir um *array* de *bytes* consiste em usar um *bitmap* para marcar se uma posição no *array* tem um valor nulo ou não. Uma vez que cada *bit* corresponde a uma posição no *array*, um *byte* pode indexar oito posições no *array*.

O funcionamento deste método é o seguinte (figura 4.4). Primeiro, divide-se a dimensão do array por oito (com arredondamento feito para cima) de forma a obter a dimensão do *bitmap*. De seguida, percorre-se cada posição do *array*. Se o valor nessa posição for nulo, o *bit* correspondente a essa posição é marcado com zero e salta-se para a próxima posição do *array*. Se por outro lado, o valor nessa posição não for nulo, o *bit* correspondente a essa posição é marcado com um e a posição do *array* é adicionada a um novo *array* que apenas contém *bytes* não nulos. Na prática, o array produzido tem todos os *bytes* não nulos do *array* original.

Após a construção do *bitmap*, o *array* final consiste na concatenação de um *byte* com a dimensão do *bitmap*, com os *bytes* que constituem o *bitmap* e com o *array* sem *bytes* nulos. A necessidade da indicação da dimensão do *bitmap* prende-se com o facto de não sabermos a dimensão original do array na fase de descompressão. Sabendo a dimensão do *bitmap* é possível calcular a dimensão original do *array* e reconstitui-lo na fase de descompressão.

Concluindo, no VFC4FPS, os objectos são serializados mas apenas é utilizado o delta. Caso existam deltas de vários objectos, eles são concatenados e depois são sujeitos à compressão mapa de bits, o resultado é por sua vez enviado através de *sockets* UDP.

4.6 Interfaces de Utilizador

Para fazer a visualização do desempenho do VFC4FPS em comparação ao sistema original do *Cube 2: Sauerbraten* foram criadas duas interfaces gráficas, uma para o cliente (figura 4.5(a)) e outra para o servidor (figura 4.5(b)). Através destas interfaces podemos ver o tráfego de entrada e saída para: eventos (C1), actualizações de objectos (C0) e o seu somatório. O tráfego é apresentado em *bytes* por segundo, de três formas: taxa actual, taxa média e maior taxa. Adicionalmente apresenta-se a totalidade de tráfego gerado.

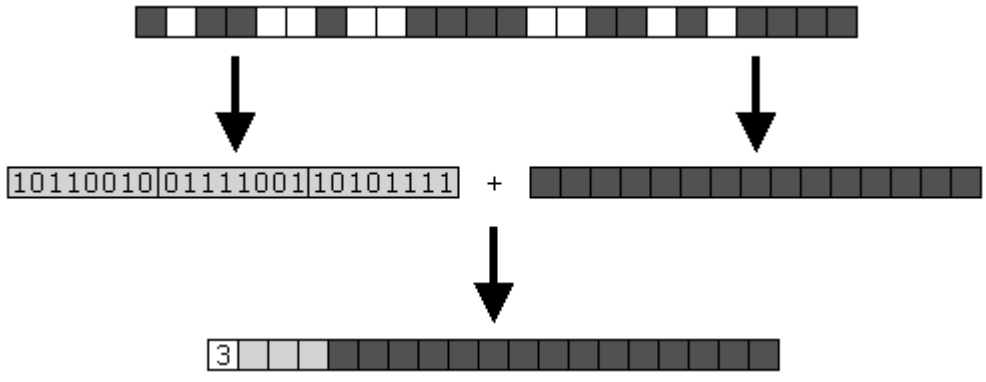
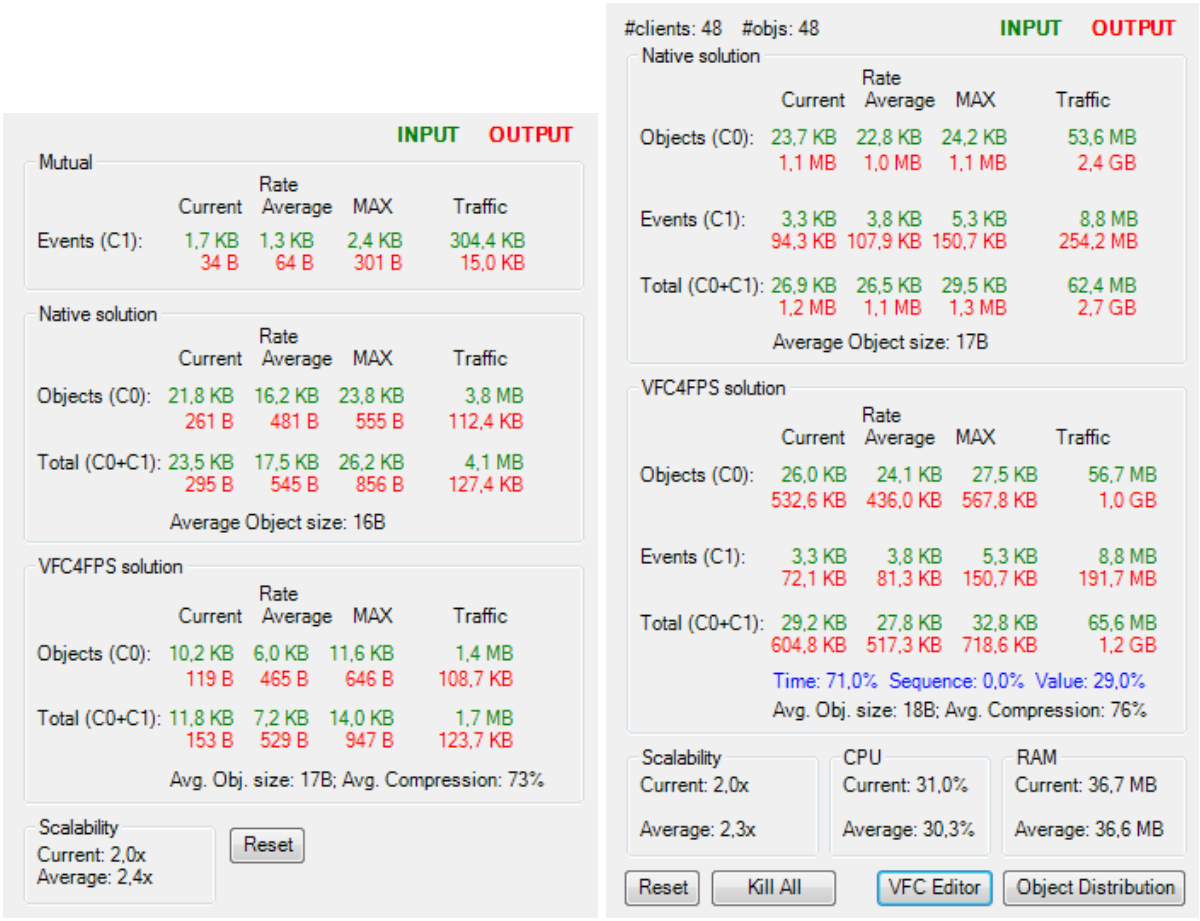
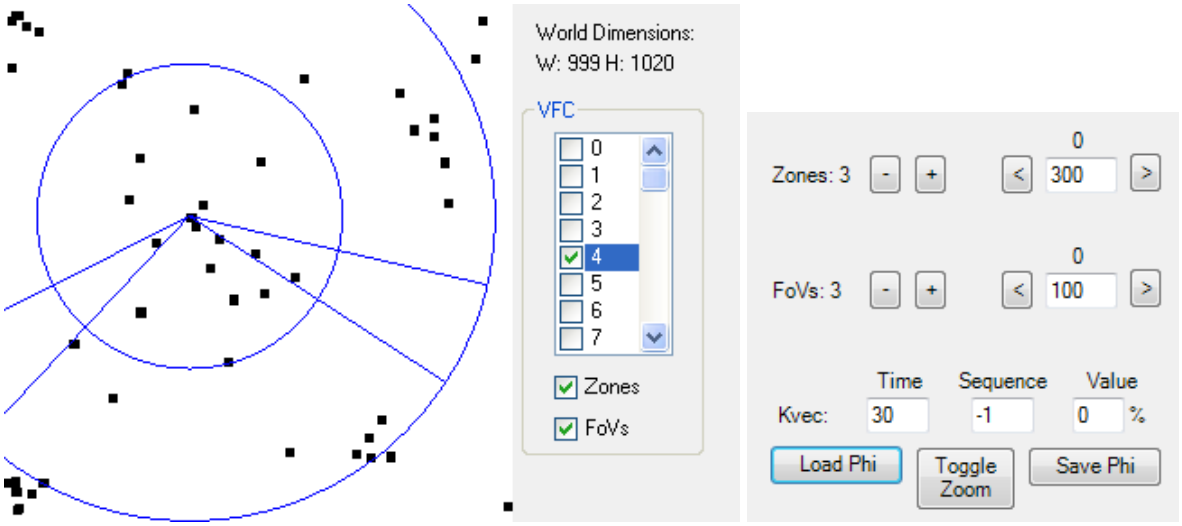


Figura 4.4: Esquema da compressão mapa de *bits*.



(a) Cliente (b) Servidor

Figura 4.5: Interfaces de monitorização.



(a) Distribuição de objectos (b) Edição vistas

Figura 4.6: Interfaces.

De forma a verificar a eficácia da compressão dos objectos serializados no VFC4FPS, e comparar com os objectos do jogo original, apresentam-se também as dimensões médias dos objectos e a taxa de compressão. Esta taxa de compressão reflecte apenas a redução ganha através da compressão mapa de *bits*. Apresenta-se ainda o rácio actual e médio entre o tráfego gerado pelo VFC4FPS e o tráfego gerado pelo jogo original. Adicionalmente, na interface do servidor são apresentados os consumos de CPU e de RAM. Assim como a distribuição da frequência com que cada um dos três parâmetros (tempo, sequência e valor) foi excedido.

Através da interface gráfica na figura 4.6(a) é possível ver a distribuição dos *avatares* no mundo virtual e a sua dimensão. Adicionalmente é possível activar para qualquer cliente a representação gráfica da sua vista (zonas e campos de visão).

De forma a facilitar a experimentação de vistas com parâmetros diferentes, utilizou-se a interface gráfica da figura 4.6(b). Através desta interface é possível aumentar ou diminuir o número de zonas de consistência ou de campos de visão, bem como os seus valores. Através dos índices de zona e campo de visão é possível aceder a todos os vectores κ e modificar os seus valores. É também possível simular a utilização de *zoom* por parte de um cliente real. As alterações feitas neste editor são aplicadas às vistas de todos os clientes.

4.7 *Bots*

Para fazer as avaliações de tráfego seriam necessários muitos jogadores e uma grande infraestrutura. Em vez disso, as nossas avaliações foram feitas através do uso de *avatares* controlados pela inteligência artificial do *Cube 2: Sauerbraten*. A este tipo de *avatares* normalmente dá-se o nome de *bot*.

O *Cube 2: Sauerbraten* oferece a possibilidade de adicionar *bots* a um mundo virtual. Contudo, o controlo dos *bots* é distribuído entre todos os clientes. O processamento de um *bot* é feito pelo cliente responsável por esse *bot*. Os problemas com esta abordagem surgem quando se pretende medir a quantidade de tráfego e de processamento de um cliente. Uma vez que um cliente pode processar vários *bots*, não é possível medir o processamento para um cliente normal que só controla o seu *avatar*. Por outro lado, como um cliente processa vários *bots*, a comunicação entre esses *bots* é feita dentro do processo do cliente. Isto causa uma redução na largura de banda utilizada para comunicar com o servidor, e que por sua vez não permite uma medição equivalente a clientes normais.

Por estas razões, o cliente do *Cube 2: Sauerbraten* foi alterado para que o *avatar* de cada cliente em vez de ser controlado por um jogador físico, passasse a ser controlado pela inteligência artificial que controla os *bots*. Conseguiu-se assim que cada cliente simulasse um jogador real.

No entanto, um cliente normal requer bastante computação gráfica. De forma a conseguir fazer as avaliações com um elevado número de *bots* sem necessitar de igual número de computadores, o *Cube 2: Sauerbraten* foi alterado para reduzir o *output* gráfico e o respectivo processamento. Para isso desactivou-se o carregamento de texturas, sons, mapas de luz e alguns tipos de modelos geométricos. Não foi possível eliminar completamente o *output* gráfico devido às dependências com a lógica de jogo. Desta forma obtivemos clientes controlados por inteligência artificial com baixos requisitos de computação gráfica. Isto permitiu correr dezenas de *bots* em cada servidor (máquina).

Capítulo 5

Avaliação

Neste capítulo vamos avaliar o desempenho do VFC4FPS numa série de parâmetros. Começamos por apresentar como é que as medições foram feitas, qual a infraestrutura e vistas utilizadas, e os mundos virtuais onde ocorreram as simulações. De seguida temos uma avaliação quantitativa dos resultados, seguida de uma avaliação qualitativa para apurar se a jogabilidade foi mantida.

5.1 Simulação do Tráfego Original

A comparação do tráfego gerado entre o sistema original e o VFC4FPS para ser justa, tem de ser feita na mesma simulação. Isto porque para fazer a simulação foram utilizados *bots*. O comportamento dos *bots* é controlado por inteligência artificial e não é determinístico entre duas simulações. Assim sendo, para fazer as simulações foram utilizados simultaneamente os dois métodos de comunicação (original e VFC4FPS) para o envio de actualização de objectos.

De forma a utilizar os dois métodos, o VFC4FPS foi utilizado juntamente com o sistema original. Isto significa que cada cliente envia as actualizações dos objectos para o servidor original e, em paralelo, o VFC4FPS cliente, envia as actualizações dos objectos para o servidor VFC4FPS. Isto permite comparar o tráfego de entrada no servidor. O problema surge na medição do tráfego de saída do servidor, pois o servidor envia cada actualização recebida para todos os clientes. Este tráfego, juntamente com o tráfego VFC4FPS produz um somatório elevado. Principalmente tendo em conta que as actualizações recebidas pelos clientes via sistema original iriam ser descartadas em favor das actualizações via VFC4FPS.

No intuito de contabilizar o tráfego de saída original, sem incorrer em tráfego adicional desnecessário, quando o servidor recebe uma actualização via sistema original, ele é descartado e é contabilizado o tráfego de saída que seria gerado através da multiplicação da dimensão do pacote pelo número de clientes para que iria ser difundida. A correcção da estimativa foi confirmada através de simulações com o sistema de difusão original completamente funcional.

Até aqui falou-se do tráfego gerado por actualizações de objectos. Para a contabilização do tráfego de eventos o sistema é diferente. A filtragem de eventos ocorre no processamento dos eventos recebidos, mas a contabilização do tráfego de saída só pode ser feito depois de filtrado. Isto significa que o tráfego contabilizado já se encontra filtrado. No entanto, pretendemos medir qual seria o tráfego gerado se não tivesse sido utilizado o filtro.

Para esse efeito, na fase de filtragem de eventos, é contabilizada a quantidade de tráfego filtrado. Esta estimação é feita através da multiplicação da dimensão do pacote do evento pelo número de clientes total menos o número de clientes a que o evento é relevante (este valor já inclui quem enviou o evento, uma vez que espacialmente a mensagem é sempre relevante a ele, apesar de não lhe ser enviada). Este tráfego filtrado é somado ao tráfego normal dos eventos para obter o tráfego que o sistema original causaria. Podendo assim ser medido o desempenho da filtragem de eventos.

5.2 Infraestrutura Utilizada

Para executar o cliente gráfico para os testes com utilizadores foi utilizado um computador de secretária com processador Intel Pentium 4 com 3.20 GHz e 1 GB de RAM. A placa gráfica utilizada foi uma ATI Radeon 9250 com 128 MB RAM. O sistema operativo utilizado foi o Microsoft Windows XP Professional com o Service Pack 3.

Ambas as versões do servidor do *Cube 2: Sauerbraten* foram executados num computador portátil com processador Intel Core Duo T2400 com 1.83 GHz e 2 GB de RAM. O sistema operativo utilizado foi o Microsoft Windows 7 Professional.

Os bots para as simulações foram executados em dois servidores (máquinas) idênticos com processador Intel Core 2 Quad Q6600 com 2.40 GHz e 8 GB de RAM. O sistema operativo utilizado foi o Microsoft Windows 7 Professional versão 64 bits.

As avaliações foram efectuadas num ambiente de rede local com velocidade de 100 *Mbps*.

5.3 Vistas Utilizadas

O desempenho do VFC4FPS está associado principalmente às vistas especificadas. As medições foram feitas usando três vistas diferentes: vista normal, vista *zoom* e vista VFC (apenas um campo de visão). A vista normal (tabela 5.1) contém os parâmetros adequados à visão normal do *avatar*. No caso do *Cube 2: Sauerbraten* um *avatar* tem 100º de campo de visão. Alem dos 100º, nesta vista podemos observar um campo de visão de 140º, isto é para evitar inconsistências momentâneas quando se faz uma rotação rápida. Estes 140º representam 20º para cada lado do campo de visão normal.

A vista *zoom* (tabela 5.2), contém os parâmetros utilizados no caso em que o *avatar* tem o *zoom* activado. Quando o *avatar* tem o *zoom* activado o campo de visão é reduzido de 100º para 35º. Isto traduz-se numa redução para mais do dobro. No entanto, optou-se por apenas duplicar o raio das zonas em relação à vista normal. Mais uma vez o campo de visão intermédio tem mais 20º para cada lado, o que se traduz nos 75º. Os vectores κ mantêm-se iguais aos da vista normal.

Zonas / FoV	300	600	∞
100º	[30, ∞, 0]	[60, ∞, 3%]	[90, ∞, 5%]
140º	[60, ∞, 6%]	[90, ∞, 9%]	[130, ∞, 13%]
360º	[90, ∞, 9%]	[200, ∞, 20%]	[300, ∞, 30%]

Tabela 5.1: Vista normal.

FoV \ Zonas	600	1200	∞
35º	[30, ∞, 0]	[60, ∞, 3%]	[90, ∞, 5%]
75º	[60, ∞, 6%]	[90, ∞, 9%]	[130, ∞, 13%]
360º	[90, ∞, 9%]	[200, ∞, 20%]	[300, ∞, 30%]

Tabela 5.2: Vista *zoom*.

FoV \ Zonas	300	600	∞
360º	[30, ∞, 0]	[60, ∞, 3%]	[90, ∞, 5%]

Tabela 5.3: Vista com apenas um campo de visão (vista VFC).

A vista VFC, é igual à vista normal, mas contém apenas um campo de visão que abrange a totalidade dos 360º em volta do *avatar*. Esta vista foi utilizada para oferecer uma comparação entre o VFC4FPS e o VFC, e provar que a inclusão do campo de visão permite um maior desempenho.

Pode ser observado que em todas as vistas, o parâmetro sequência do vector κ é sempre infinito. Este facto deve-se a uma escolha pessoal. A razão para esta escolha é a seguinte. Com um parâmetro tempo é possível especificar quantos milissegundos um objecto pode tolerar sem ser actualizado e ter uma ideia do seu impacto. O mesmo acontece com o parâmetro valor. No entanto, o número de actualizações recebidas não permite ter uma ideia de quanto é que o estado de um *avatar* se alterou. Pode-se ter uma percepção temporal, por estar ligado à frequência com que são recebidas, mas isso já é contemplado pelo parâmetro tempo, pelo que torna a sequência redundante neste contexto.

5.4 Mapas

Os mundos virtuais, referidos simplesmente por mapas daqui para a frente, utilizados para fazer medições são apresentados na tabela 5.4. Juntamente com o nome é apresentada a dimensão de cada mapa. A maioria das medições foram feitas nos mapas destacados (*wdcd*, *flagstone* e *urban_c*) por representarem três categorias de dimensões: normal, médio e grande. Adicionalmente algumas medições foram feitas para os restantes mapas, com o objectivo de consolidar os resultados.

5.5 Avaliação Quantitativa

Nesta secção apresentam-se as medições relativas à redução de tráfego, aos *overheads* por se utilizar o VFC4FPS, e o desempenho da compressão.

5.5.1 Redução de Tráfego

Para medir a redução do tráfego gerado recorreu-se à utilização de 48 *bots* em diferentes mapas. A escolha deste número de *bots* está ligada ao número normal da jogadores nos servidores. Esse número encontra-se à volta dos 24 jogadores. Como o objectivo é reduzir o tráfego para metade, optou-se por

Nome	Dimensão
aard3c	250x250
academy	250x250
aqueducts	875x750
arabic	875x750
akroseum	1000x1000
dust2	1000x1000
campo	1000x1000
venice	1000x1000
wdcd	1000x1000
redemption	1250x500
core_transfer	1250x750
face-capture	1500x250
damnation	1500x500
shipwreck	1500x1250
flagstone	1500x1500
hallo	1500x1500
ph-capture	1500x1500
river_c	1500x1500
mach2	1750x750
urban_c	2250x1750

Tabela 5.4: Mapas e respectivas dimensões.

duplicar o número de "jogadores"(*bots*) para determinar se os ganhos se aproximam do tráfego gerado por 24 jogadores.

Todas as medições foram feitas no modo de jogo *ffa* (todos contra todos, com vários tipos de armas) de forma a haver mais eventos de tiro e demonstrar melhor o impacto do filtro de eventos. As medições foram contabilizadas apenas a partir da altura em que todos os 48 *bots* estavam ligados ao servidor. A duração de cada medição foi de 10 minutos. Foi escolhido este valor para obter valores médios estáveis e por ser a duração normal de um mapa.

Servidor

Um servidor possui tráfego de entrada e de saída. O tráfego de entrada é reduzido e constante. O maior tráfego observa-se na saída, sendo esse o tráfego a que o VFC4FPS é aplicado. Por estas razões apenas se apresentam as medições do tráfego de saída do servidor.

No final desta secção apresentamos uma tabela com os valores médios.

Nas figuras 5.1, 5.2 e 5.3 estão apresentados os resultados relativos à aplicação da vista normal. Estes gráficos apresentam a taxa de tráfego de saída, em *bytes* por segundo, dos dois canais de comunicação para as duas soluções (Original e VFC4FPS). Como se pode observar, as taxas de tráfego mantêm-se estáveis ao longo do tempo. Apesar dos gráficos dizerem respeito a mapas com dimensões diferentes, podemos observar uma redução para menos de metade do tráfego de actualização de objectos. Algo que também podemos observar é o reduzido tráfego gerado pelos eventos. A pequena diferença entre os eventos originais e os eventos "VFC4FPS" deve-se à filtragem aplicada.

Nas figuras 5.4, 5.5 e 5.6 apresentamos resultados similares aos do parágrafo anterior, mas desta vez para com a vista *zoom*. Esta vista foi utilizada em todos os *bots* para representar o pior caso, ou seja,

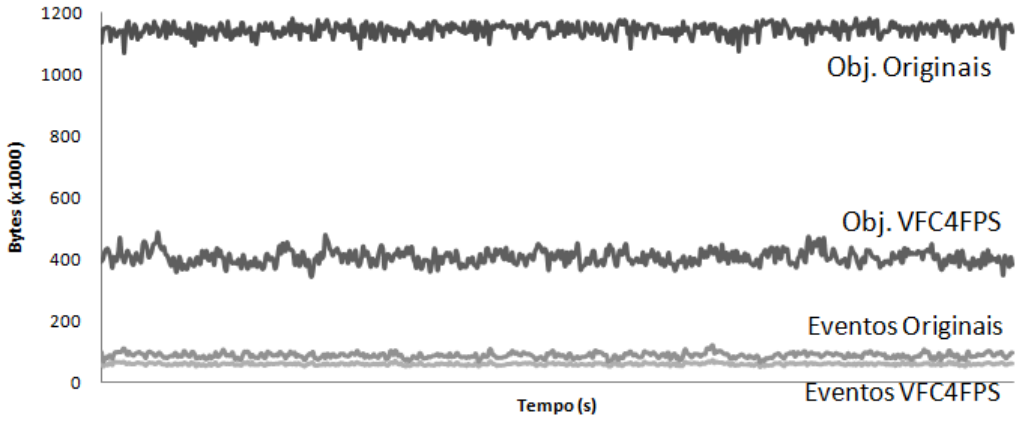


Figura 5.1: Tráfego de saída no servidor ao longo de 10 minutos no mapa *wdcd* usando a vista normal.

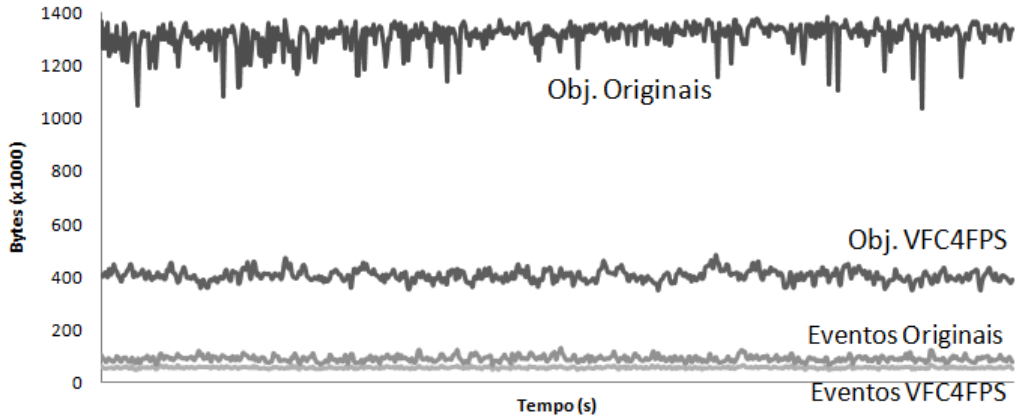


Figura 5.2: Tráfego de saída no servidor ao longo de 10 minutos no mapa *flagstone* usando a vista normal.

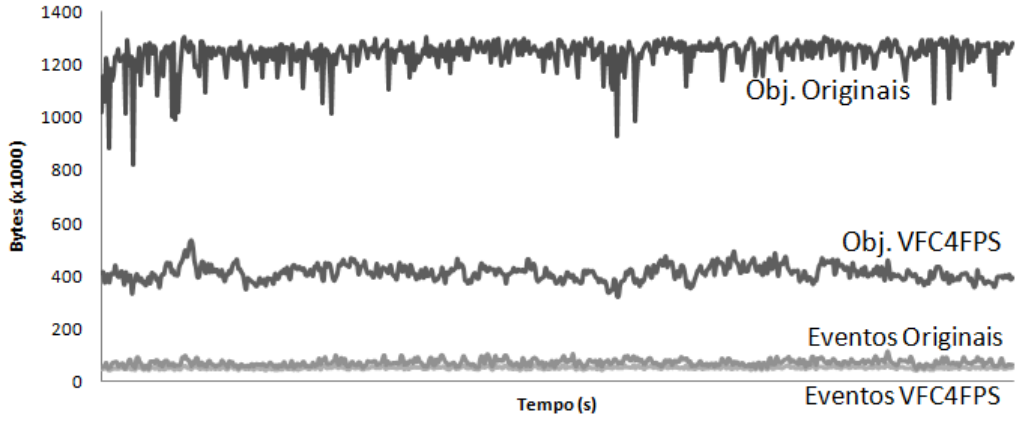


Figura 5.3: Tráfego de saída no servidor ao longo de 10 minutos no mapa *urban_c* usando a vista normal.

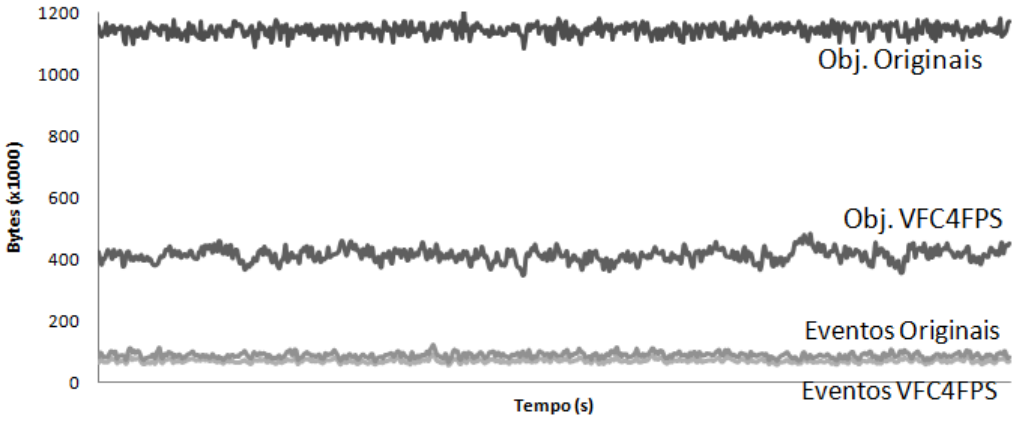


Figura 5.4: Tráfego de saída no servidor ao longo de 10 minutos no mapa *wdcd* usando a vista *zoom*.

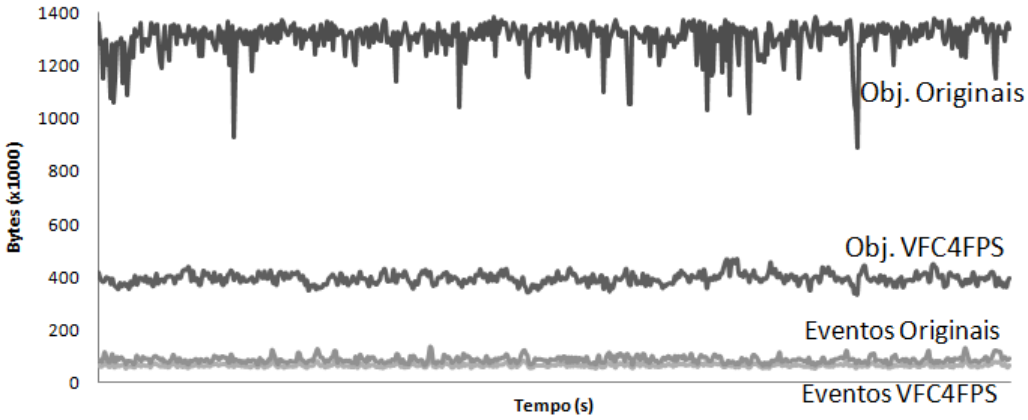


Figura 5.5: Tráfego de saída no servidor ao longo de 10 minutos no mapa *flagstone* usando a vista *zoom*.

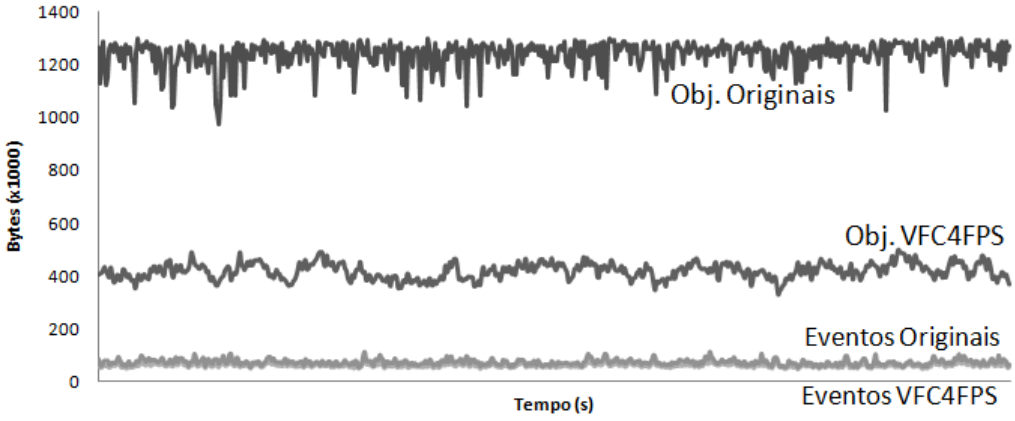


Figura 5.6: Tráfego de saída no servidor ao longo de 10 minutos no mapa *urban_c* usando a vista *zoom*.

em que todos os clientes têm o *zoom* activado. Outra razão prende-se com o facto da implementação dos *bots* não requerer a função de *zoom*, uma vez que se trata de uma característica visual. Por isso, de forma a confirmar que a utilização de *zoom* não prejudica a redução de tráfego, foi considerado o pior caso, em que todos os *bots* possuem o *zoom* activo. Como esperado, o aumento do raio das zonas juntamente com a redução dos campos de visão não prejudicou o desempenho do VFC4FPS em relação à vista normal. A principal diferença em relação aos resultados da vista normal é a menor diferença entre o tráfego de eventos. Isto deve-se ao facto de as zonas serem mais abrangentes o que reduz a filtragem.

Terminamos com as mesmas medições, mas desta vez para a vista VFC (figuras 5.7, 5.8 e 5.9). Como se pode observar, usando apenas um campo de visão, a redução de tráfego não chega à metade do tráfego original.

Na tabela 5.5 apresenta-se os valores médios para o rácio entre o tráfego original e o tráfego VFC4FPS para as diferentes vistas. Além dos resultados numéricos para os três mapas que foram apresentados anteriormente, apresentam-se os resultados para a vista normal noutros mapas. Observa-se uma tendência no rácio aumentar com o tamanho dos mapas. Isto é justificado pela possibilidade dos *avatares* se encontrarem mais dispersos comparativamente aos mapas pequenos. Numericamente, podemos concluir que a redução conseguida através da vista normal e da vista *zoom* é igual. Podemos ainda concluir que tendo apenas um campo de visão, a redução não chega à metade.

Voltando aos três mapas a que temos dado mais destaque, podemos observar que a redução de tráfego não aumenta entre o mapa *flagstone* e *urban_c*, apesar do segundo ser maior. De forma a tentar compreender a razão deste resultado, foram gerados mapas de calor através da amostragem periódica das posições dos *avatares* (figura 5.10). A ideia inicial era que a geometria do mapa era a principal influência no desempenho do VFC4FPS. Através dos mapas de calor podemos concluir que a concentração de jogadores é o maior influenciador do desempenho. A concentração de jogadores está associada aos pontos de interesse de cada mapa. O mapa de calor do *urban_c* concentra os *avatares* no centro do mapa, enquanto que no *flagstone* eles se encontram espalhados mais uniformemente.

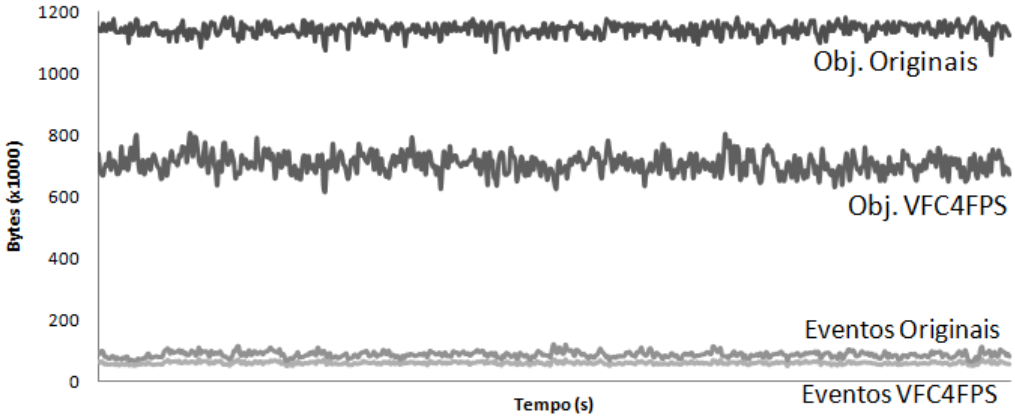


Figura 5.7: Tráfego de saída no servidor ao longo de 10 minutos no mapa *wdcd* usando a vista VFC.

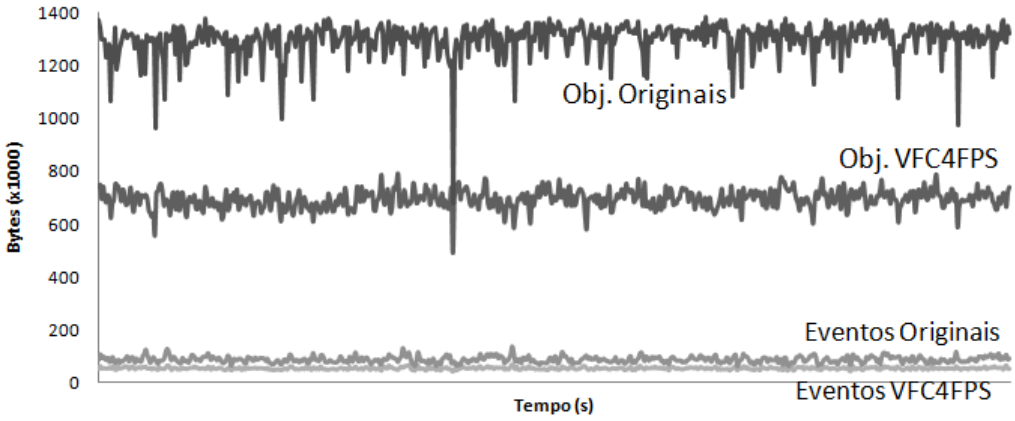


Figura 5.8: Tráfego de saída no servidor ao longo de 10 minutos no mapa *flagstone* usando a vista VFC.

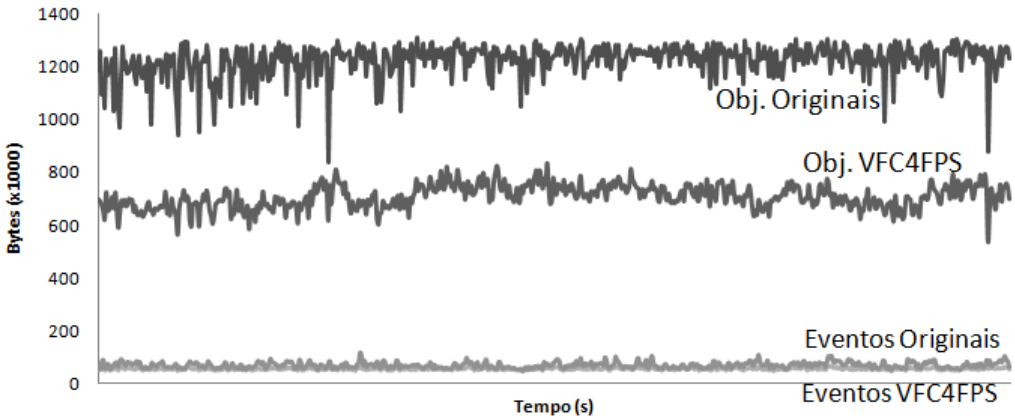
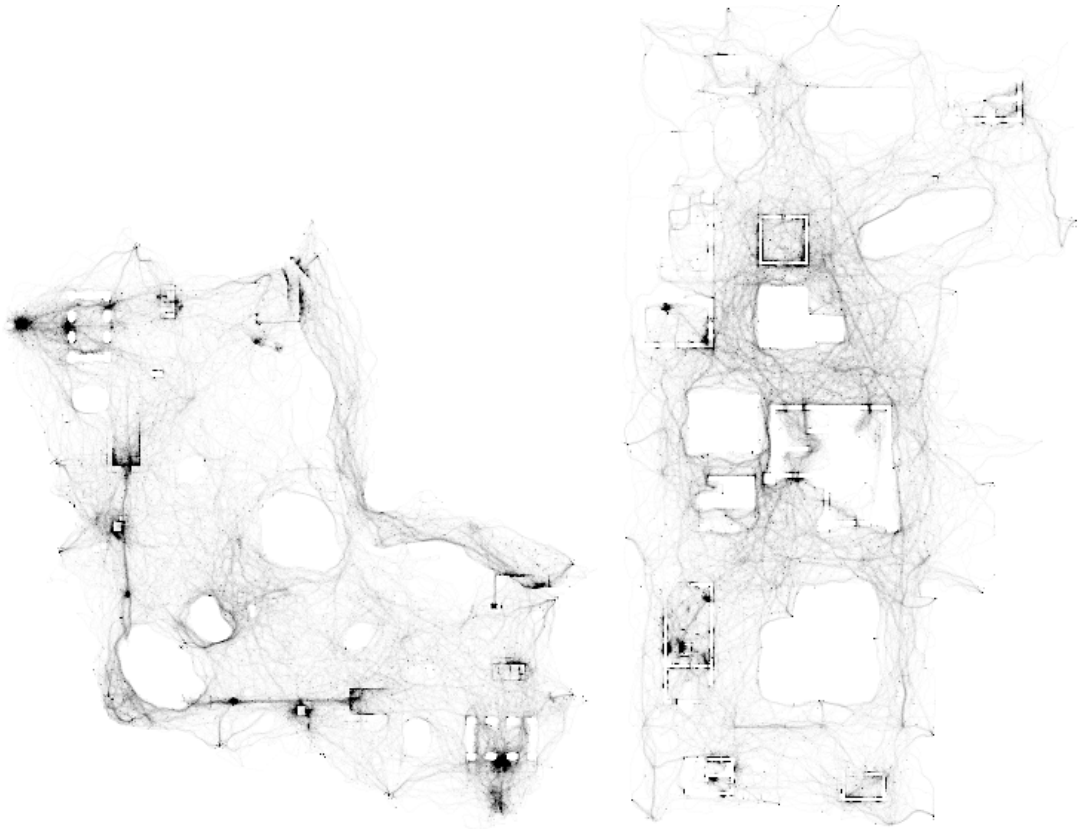


Figura 5.9: Tráfego de saída no servidor ao longo de 10 minutos no mapa *urban_c* usando a vista VFC.

Mapa		Vistas		
Nome	**Dimensão**	**Normal**	***Zoom***	**VFC**
aard3c	250x250	1,7		
academy	250x250	1,5		
aqueducts	875x750	1,7		
arabic	875x750	2		
akroseum	1000x1000	2,2		
dust2	1000x1000	2,5		
campo	1000x1000	1,7		
venice	1000x1000	2,5		
wdcd	1000x1000	2,6	2,6	1,6
redemption	1250x500	2,3		
core_transfer	1250x750	2,7		
face-capture	1500x250	2,5		
damnation	1500x500	2,1		
shipwreck	1500x1250	3		
flagstone	1500x1500	3	3	1,8
hallo	1500x1500	2,7		
ph-capture	1500x1500	2,7		
river_c	1500x1500	2,9		
mach2	1750x750	2,9		
urban_c	2250x1750	2,8	2,7	1,7

Tabela 5.5: Rácios médios de redução de tráfego obtidos ao fim de 10 minutos.



(a) *flagstone* (b) *urban_c*

Figura 5.10: Mapas de calor, à escala.

Cliente

Contrariamente ao servidor, num cliente, o maior tráfego é o de entrada. Por essa razão, nesta secção apresenta-se a redução obtida no cliente por via da redução feita no servidor. As condições de simulação foram as mesmas que para o servidor: 48 *bots*, 10 minutos, modo *ffa*, usando a vista normal. Para poder fazer a comparação com o tráfego original, foi activada a difusão de actualizações de objectos original. No entanto, o tráfego de actualização de objectos original é descartado na recepção pelo cliente.

Com recurso às figuras 5.11, 5.12, 5.13 e à tabela 5.6, observamos a redução obtida por um cliente. Em média, independentemente do mapa, um cliente tem uma redução de tráfego de mais de metade, quase chegando a uma redução para um terço do tráfego original. Observa-se alguma variação no tráfego VFC4FPS em relação ao tráfego original. Isto é porque a recepção de actualizações do servidor original tinha uma frequência constante, enquanto que no VFC4FPS o movimento e posição do *avatar* influência fortemente a frequência com que recebe as actualizações dos diferentes objectos.

5.5.2 *Overheads*

Nesta secção vamos medir os *overheads* causados pelo uso de uma linguagem de alto nível, pelo uso do *.Net remoting* e pelo próprio processamento requerido pelo VFC4FPS. Para a medição de *overheads* não ser influenciada pelo sistema de actualização de objectos original (necessário para comparações de tráfego) o envio destas actualizações foram desactivadas completamente.

Servidor

O VFC4FPS requer determinação das distâncias entre objectos e determinação de campos de visão. Em comparação, o sistema original do jogo limita-se a encaminhar as actualizações recebidas pelos restantes clientes. Se juntarmos o número de deserializações de objectos que são necessárias fazer no servidor e o facto do VFC4FPS estar implementado numa linguagem de alto nível, facilmente se conclui que os requisitos computacionais, bem como de memória vão ser superiores ao jogo original. Mediu-se para um número crescente de clientes, quais os requisitos de CPU e RAM utilizados pelo VFC4FPS comparado

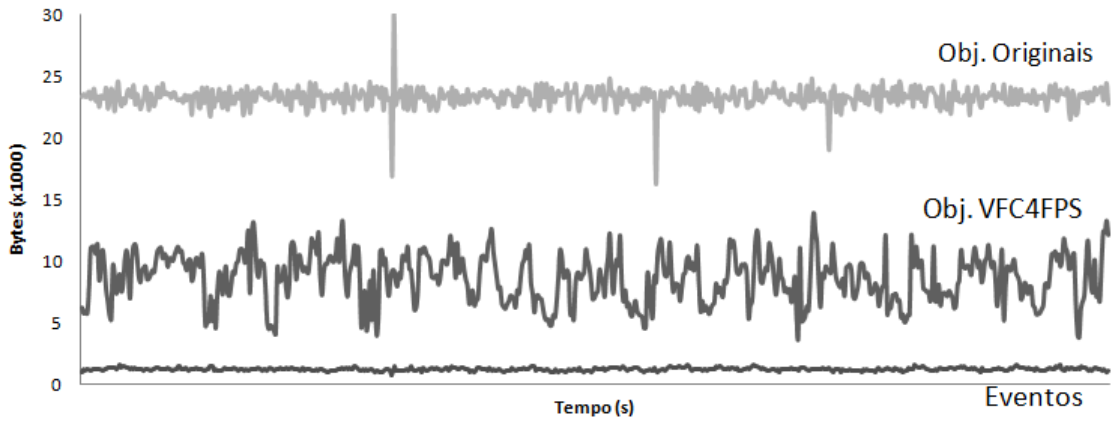


Figura 5.11: Tráfego de entrada num cliente ao longo de 10 minutos no mapa *wdcd*.

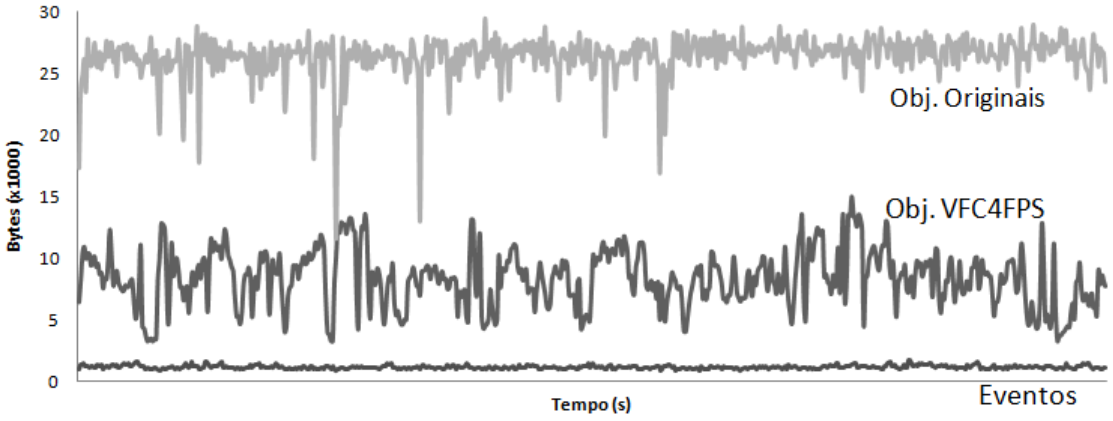


Figura 5.12: Tráfego de entrada num cliente ao longo de 10 minutos no mapa *flagstone*.

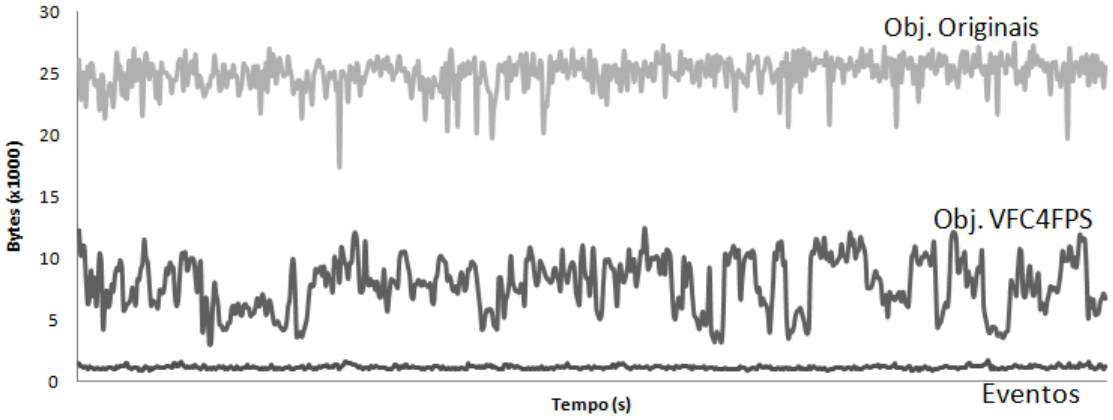


Figura 5.13: Tráfego de entrada num cliente ao longo de 10 minutos no mapa *urban_c*.

	wdcd	flagstone	urban_c
Rácio médio	2,5	2,9	2,9

Tabela 5.6: Rácios médios de redução de tráfego de entrada num cliente.

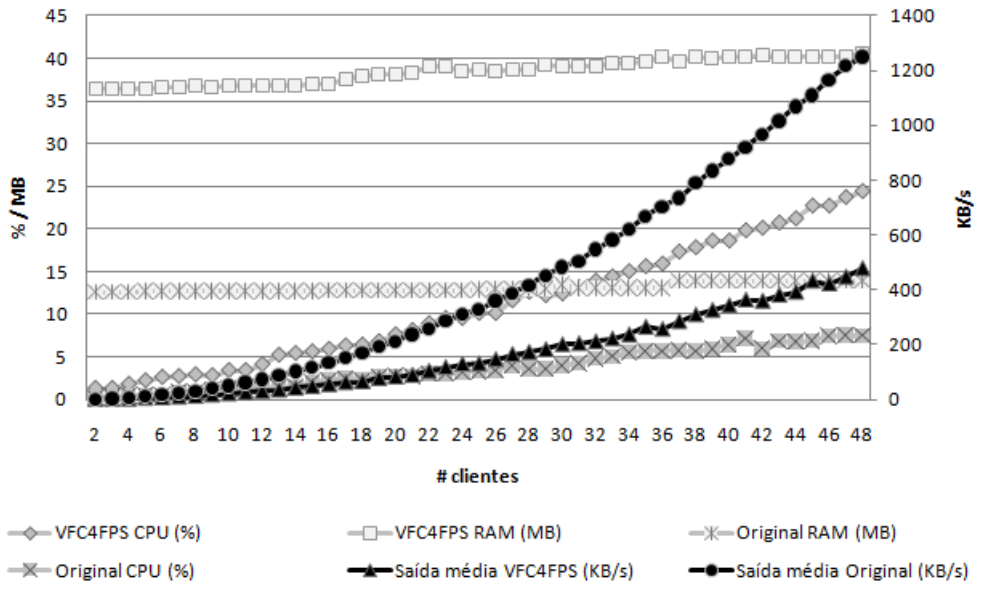


Figura 5.14: Comparação do desempenho do servidor entre o VFC4FPS e o jogo original.

com o jogo original. Estes valores são complementados pelo tráfego médio de saída no servidor para ambas as soluções. Os resultados encontram-se no gráfico 5.14.

Como se pode observar pelo gráfico 5.14 para obter uma redução de tráfego superior a duas vezes, o processamento de CPU no VFC4FPS é cerca de três vezes maior que o original. No consumo de RAM verifica-se que o VFC4FPS também utiliza cerca de três vezes mais que o sistema original. Observa-se que à medida que o número de clientes aumenta os requisitos de CPU aumentam linearmente. Contudo, no total, o uso de memória é reduzido e a utilização de CPU para 48 *bots* não chega a 50%. Além disso, aumentar a quantidade de RAM e capacidade de CPU é mais fácil que o aumento da capacidade de largura de banda. Podemos então concluir, que apesar do VFC4FPS adicionar algum consumo em termos de RAM e CPU, ele não é significativo, e os ganhos na redução de tráfego compensam.

Tráfego *.Net remoting*

Algo que ainda não foi referido, foi o tráfego gerado pela interacção entre o cliente VFC4FPS e servidor VFC4FPS via *.Net remoting*. Isto por duas razões, primeiro as chamadas por *remoting* são feitas na entrada de um novo cliente. Segundo, porque não é possível medir este tráfego internamente. Contudo, uma chamada de *remoting* que pode ser feita mais frequentemente é a chamada para alterar os parâmetros de uma vista, neste caso, devido à activação ou desactivação de *zoom* por parte de um cliente. Podendo a utilização de *zoom* ser bastante frequente, isto origina algum consumo de tráfego adicional no servidor. Através de uma aplicação externa foi possível obter valores aproximados das dimensões de uma chamada deste tipo.

Para a comunicação no sentido cliente-servidor o tráfego para cada (des)activação do *zoom* é de 1109 *bytes*. A resposta do servidor (sentido servidor-cliente) corresponde a 39 *bytes*. Como se pode verificar, o tráfego de entrada no servidor é bem maior que o tráfego de saída. Isto porque a chamada inicial contém os novos parâmetros da vista. O tráfego mais crítico no servidor é o tráfego de saída mas, no caso deste tipo de chamadas, o tráfego adicional é reduzido. Outro facto que reduz o impacto deste tipo

de chamadas é a sua própria frequência, que (a uma escala reduzida de tempo) é baixa. Por estas razões, considerámos este tráfego adicional negligenciável.

Cliente

Da mesma forma que foi feito para o servidor, mediu-se para o cliente se havia impacto significativo, a nível de utilização de memória e processador, devido ao uso do VFC4FPS. Para esse efeitos, executaram-se 48 *bots* no mapa *flagstone* no modo *ffa* (todos contra todos) com a vista normal durante 10 minutos. Para fazer estas medições utilizou-se um dos *bots* com as reduções de processamento gráfico. Na tabela 5.7 apresentamos os resultados médios obtidos para um dos *bots* após esse intervalo de tempo. Tirando o ligeiro aumento da memória utilizada, os requisitos computacionais mantêm-se quase iguais. Isto é justificado pelo facto do processamento efectuado pelo VFC4FPS num cliente é muito menor que o do servidor. Um cliente VFC4FPS, só precisa de enviar actualizações periodicamente, e processar as actualizações de recebe do servidor.

5.5.3 Compressão

No capítulo 4 falou-se dos dois sistemas utilizados para reduzir a dimensão dos objectos transmitidos entre cliente e servidor VFC4FPS. O objectivo era aproximar a dimensão dos objectos VFC4FPS o máximo possível à dimensão dos objectos do sistema original. Isto para permitir a utilização de objectos de alto nível oferecidos pela linguagem de programação.

Para verificar se o objectivo foi atingido, as medições de redução de tráfego do servidor serviram adicionalmente para medir a dimensão média dos objectos originais, objectos do VFC4FPS e a percentagem média da dimensão dos objectos após aplicação da compressão mapa de *bits*.

Como se pode observar na tabela 5.8 os objectos VFC4FPS diferem em apenas um ou dois *bytes* dos objectos originais, e em alguns casos os objectos VFC4FPS são os mais pequenos. A diferença de valores entre mapas deve-se à geometria do mapa. Foi verificado que para mapas com uma superfície mais plana causava que alguns campos do estado dos objectos se encontrassem frequentemente nulos. Nesta situação a compressão mapa de *bits* do VFC4FPS e o sistema utilizado pelo jogo original conseguiam maiores reduções. Pode-se confirmar o sucesso da compressão mapa de *bits*, que consegue uma compressão média de 80% apesar de apenas eliminar os *bytes* nulos e ainda adicionar o mapa de *bits*.

5.6 Avaliação Qualitativa

Nesta secção pretendemos avaliar se a jogabilidade do *Cube 2: Sauerbraten* não foi prejudicada ao usar o VFC4FPS. Para esse efeito, foram conduzidos testes com utilizadores voluntários. Como a jogabilidade é impossível de avaliar objectivamente, esta avaliação foi feita comparando o *Cube 2: Sauerbraten* original

	VFC4FPS	Original
CPU	2,2%	2,1%
RAM	35 MB	26,7 MB

Tabela 5.7: Consumos médios de um cliente.

Nome	Dimensão	Objectos Originais	Objectos VFC4FPS	Compressão mapa de bits
aard3c	250x250	20	18	77%
academy	250x250	17	18	78%
aqueducts	875x750	17	19	81%
arabic	875x750	17	19	79%
akroseum	1000x1000	17	19	79%
dust2	1000x1000	19	18	79%
campo	1000x1000	17	19	79%
venice	1000x1000	17	18	78%
wdcd	1000x1000	17	18	79%
redemption	1250x500	18	19	80%
core_transfer	1250x750	19	19	81%
face-capture	1500x250	20	19	80%
damnation	1500x500	17	19	80%
shipwreck	1500x1250	20	19	82%
flagstone	1500x1500	20	19	82%
hallo	1500x1500	19	19	82%
ph-capture	1500x1500	18	18	78%
river_c	1500x1500	18	20	83%
mach2	1750x750	19	19	81%
urban_c	2250x1750	19	18	76%

Tabela 5.8: Comparação da dimensão dos objectos e taxa de compressão.

com o *Cube 2: Sauerbraten* com VFC4FPS. Cada utilizador jogou as duas versões do *Cube 2: Sauerbraten* sem saber qual era qual.

5.6.1 Metodologia

Os testes foram feitos um jogador de cada vez, contra 48 *bots*. Para o jogador se familiarizar com o jogo, foram dados cinco minutos de treino numa das versões, a qual foi alternada entre cada jogador. Após os 5 minutos, o jogador começou pela mesma versão em que tinha treinado, e jogou durante 10 minutos. Após este tempo, foi pedido ao jogador para trocar de versão e jogar mais 10 minutos. A escolha dos 10 minutos de duração de cada versão está ligada à duração de um jogo normal. Foram dados apenas 5 minutos de treino, porque mesmo assim já se totalizava quase 30 minutos que os voluntários tinham de despender. Os testes foram conduzidos no mapa *urban_c*, no modo *instagib* (onde um único tiro mata e não existe preocupação em procurar munições uma vez que só existem duas armas, uma com elevado número de munições e outra sem munições que só tem efeito à queima roupa).

No final dos 25 minutos de jogo, foi pedido ao jogador que respondesse a um curto questionário (figura 5.15), em que a pergunta principal era se tinha reparado em diferenças significativas na jogabilidade entre as duas versões, principalmente ao nível da fluidez e realismo dos movimentos dos oponentes. No caso de uma resposta positiva, pediu-se para especificarem que tipo de diferenças encontrou, qual a versão que preferiu e porquê. Adicionalmente, como forma de contexto, perguntou-se a idade, o sexo, qual a experiência com *First Person Shooters* e se usou *zoom* durante o jogo.

Idade:_____

Sexo: M ☐ F ☐

Experiência com First Person Shooters: Nenhuma ☐ Pouca ☐ Alguma ☐ Muita ☐

Utilizou zoom? Sim ☐ Não ☐

Notou diferenças entre as duas versões a nível da jogabilidade, como, por exemplo nos movimentos dos oponentes ou outra? Sim ☐ Não ☐

Se sim, quais?_____

Qual versão prefere?_____

Porque?_____

Figura 5.15: Inquérito feito aos voluntários.

5.6.2 Resultados

Os testes foram conduzidos com 16 voluntários. A média de idades dos voluntários situa-se nos 25 anos. Da totalidade dos voluntários, 2 são do sexo feminino, sendo os restantes do sexo masculino. Apenas 3 dos voluntários responderam que notaram diferenças entre as duas versões. Sendo que um deles favoreceu a versão do VFC4FPS. Isto traduz-se em 88% de respostas favoráveis ao VFC4FPS.

Entre os 12% de respostas negativas, foram apontadas diferenças a nível da velocidade dos oponentes e compensação dos movimentos. Esta última por parte de um jogador com muita experiência, que habituado a compensar o alvo do tiro em ambientes multi-jogador *online*, disse que observava na versão VFC4FPS uma compensação extra dos movimentos dos oponentes mais afastados, que lhe reduzia a pontaria.

No gráfico da figura 5.16 podemos observar as distribuições de experiência com FPS para as duas opiniões em relação às diferenças. A maioria dos jogadores, apesar de ter alguma ou muita experiência com FPS não conseguiu notar diferenças entre as duas versões.

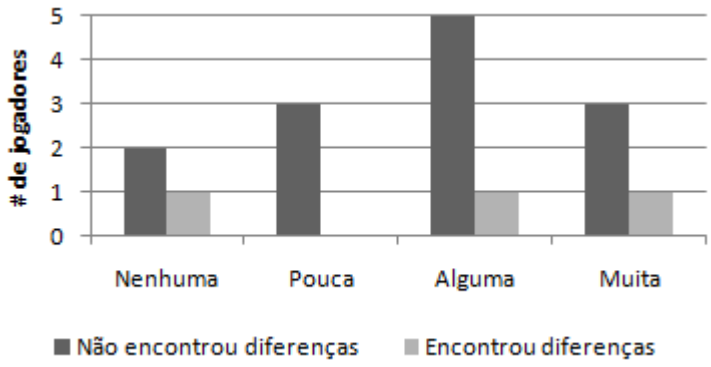


Figura 5.16: Distribuição de experiência com FPS.

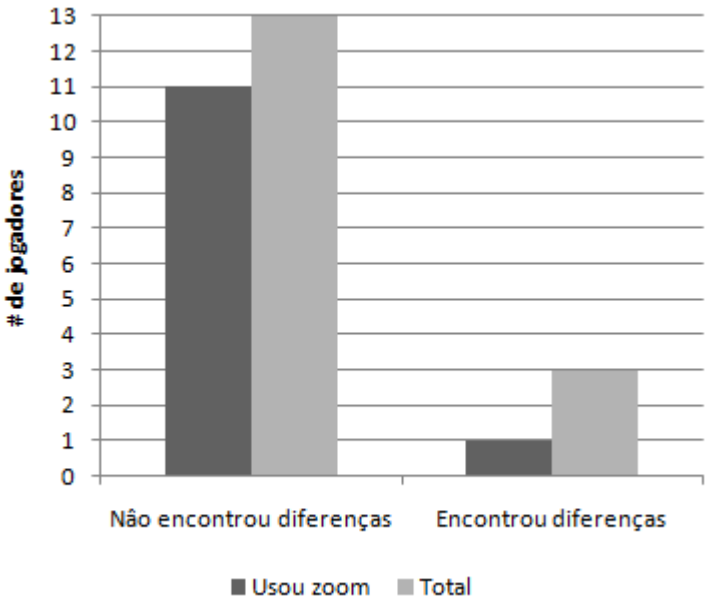


Figura 5.17: Utilização do *zoom* por parte dos jogadores.

A maior parte dos jogadores fez uso do *zoom*, o que confirma que a alteração da dimensão das zonas e dos ângulos de visão funciona bem. Podemos observar a utilização do *zoom* pelos jogadores, agrupada de acordo com a opinião relativa às diferenças, na figura 5.17.

Observou-se um pormenor peculiar no contexto relativo a quem disse que encontrou diferenças entre as duas versões. Todos estes jogadores declararam que, tendo encontrado diferenças, preferiam a última versão que tinham jogado. Uma vez que a primeira versão a ser jogada era alternada entre jogadores, pode-se deduzir que o facto de já ter mais prática quando se jogava a última versão pode ter influenciado a percepção, favorecendo a opinião em relação à última versão. Assim, podemos concluir que os resultados não estão influenciados, e que a utilização do VFC4FPS foi um sucesso.

Capítulo 6

Conclusão

Os *First Person Shooters* têm um ritmo acelerado em que a precisão dos movimentos dos oponentes é importante. Quando jogados *online* requerem comunicação muito frequente para manter os jogadores actualizados. Isto traduz-se num grande tráfego de comunicação, principalmente à saída do servidor em arquitecturas Cliente/Servidor. A grande utilização de largura de banda é o principal limitador do número de jogadores suportado.

Neste documento apresentámos as várias maneiras reduzir a utilização da largura de banda em jogos *online* multi-jogador. Foram abrangidos os tópicos de arquitectura, consistência, gestão de interesse, *Dead Reckoning* e técnicas ao nível da rede. Assim como trabalhos académicos e comerciais relevantes.

Propusemos o VFC4FPS, uma biblioteca para gerir a consistência de FPS. O VFC4FPS é baseado no VFC, o qual permite uma redução progressiva e controlada da consistência. O nosso sistema explora o contexto da visão limitada dos *avatares* num FPS e adiciona ao VFC o conceito de campos de visão. Permitindo assim reduzir a consistência à medida que os objectos se situam fora do campo de visão.

O VFC4FPS foi aplicado ao jogo *open source Cube 2: Sauerbraten* com o objectivo de reduzir, no mínimo para metade, a utilização da largura de banda no servidor (e consequentemente nos clientes). Podendo assim suportar um maior número de jogadores. Avaliámos o desempenho do *Cube 2: Sauerbraten* com o VFC4FPS em diversos mapas, e os resultados mostram que o objectivo de reduzir o tráfego para metade foi atingido e em muitos casos superado. Isto sem prejudicar a jogabilidade em comparação com o *Cube 2: Sauerbraten* original.

6.1 Trabalho Futuro

Os bons resultados deste trabalho abrem as portas a alguns melhoramentos:

- Actualmente, a implementação do VFC4FPS focou-se muito no funcionamento do *Cube 2: Sauerbraten*, sendo que algumas das funcionalidades não ficaram completas. Como trabalho futuro, uma implementação do VFC4FPS na forma de uma biblioteca completamente genérica, sem as limitações da plataforma *.Net* permitiria uma integração com um elevado número de FPS.
- Manteve-se a arquitectura Cliente/Servidor, utilizada por ambos *Cube 2: Sauerbraten* e VFC. Algo que pode ser experimentado é a modificação do VFC4FPS para uma arquitectura P2P.

- Investigar a possibilidade de incluir de forma eficiente e genérica, uma representação geométrica do mundo virtual para permitir maiores ganhos através da computação da visibilidade de cada cliente.

Bibliografia

[1] R. Alonso, D. Barbara, and H. Garcia-Molina. Data caching issues in an information retrieval system. *ACM Trans. Database Syst.*, 15(3):359–384, 1990.

[2] C. E. Bezerra, F. R. Cecin, and C. 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] J.-S. Boulanger, J. Kienzle, and C. 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] E. Cronin, B. Filstrup, A. R. Kurc, and S. 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] J. Dyck. A survey of application-layer networking techniques for real-time distributed groupware. Technical report.

[6] J. Dyck, C. Gutwin, T. C. N. Graham, and D. 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] S. Fiedler, M. Wallner, and M. 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] T. 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] R. Krishna Balan, M. Ebling, P. Castro, and A. 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] J. Pang. Scaling peer-to-peer games in low-bandwidth environments. In *In Proc. 6th Intl. Workshop on Peer-to-Peer Systems IPTPS*, 2007.

[11] L. Pantel and L. 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] L. Pantel and L. 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] Y. Saito and M. Shapiro. Optimistic replication. *ACM Comput. Surv.*, 37(1):42–81, 2005.

[14] N. Santos, L. Veiga, and P. Ferreira. Vector-field consistency for ad-hoc gaming. In *Middleware '07: Proceedings of the ACM/IFIP/USENIX 2007 International Conference on Middleware*, pages 80–100, New York, NY, USA, 2007. Springer-Verlag New York, Inc.

[15] J. Smed, T. Kaukoranta, and H. 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.

[16] S. Xiang-bin, W. Yue, L. Qiang, D. Ling, and L. 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.

[17] H. Yu and A. Vahdat. Design and evaluation of a conit-based continuous consistency model for replicated services. *ACM Trans. Comput. Syst.*, 20(3):239–282, 2002.