INSTITUTO SUPERIOR TÉCNICO
Universidade Técnica de Lisboa

VFC-reckon
Consistência em jogos multi-jogador

Mário Jorge Ferreira dos Santos

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

Júri

Presidente:	Professor Pedro Manuel Moreira Vaz Antunes de Sousa
Orientador:	Professor Paulo Jorge Pires Ferreira
Co-Orientador:	Professor Luís Manuel Antunes Veiga
Vogais:	Professor João António Madeiras Pereira

Novembro de 2011

Resumo

Na sociedade de hoje, os telemóveis/tablets/netbooks são uma parte essencial da nossa vida e é difícil encontrar alguém que não tenha um dispositivo deste género. Estes dispositivos são usados para trabalhar, jogar, ouvir música, navegar na internet, etc... Os jogos multi-jogador para estes dispositivos, enquanto interessantes e divertidos de jogar, levantam grandes problemas de escalabilidade e jogabilidade ao exigirem uma grande comunicação entre eles, necessária para manter o estado do jogo consistente entre os jogadores. O objectivo deste trabalho é aumentar a escalabilidade dos jogos através da redução no número de mensagens trocadas entre os dispositivos o que vai reduzir o consumo de largura de banda e CPU. O nosso trabalho consiste na concepção e desenvolvimento de um modelo de consistência chamado VFC-reckon, baseado em técnicas de Gestão de Interesse e DeadReckoning. O VFC-reckon permite reduzir largamente o número de mensagens trocadas entre os dispositivos sem afectar de uma maneira notável a jogabilidade do jogo. O nosso sistema foi desenvolvido para a plataforma Android, em conjunto com um jogo distribuído que demonstra as vantagens do modelo de consistência VFC-reckon para ambientes móveis em redes ad-hoc.

Palavras Chave

Dispositivos Móveis, Redes Ad-Hoc, Consistência, Gestão de Interesse, DeadReckoning, Jogos Multi-Jogador

Abstract

Nowadays, mobile phones/tabs/netbooks are an essential part of our life and it is hard to find someone who does not have such a device. These devices are used to work, play games, listen to music, navigate on the internet, etc... Multiplayer games to such devices, while interesting and fun to play, raise serious scalability and playability difficulties as they require a massive communication flow between them, needed to maintain the game state consistent between players. The goal of this work is to raise the game scalability by reducing the number of messages exchanged between the devices which will reduce the bandwidth and CPU usage. Our work consists in the conception and development of a consistency model named VFC-reckon, based on Interest Management and Dead Reckoning techniques. VFC-reckon allows us to reduce the number of messages exchanged between the devices without affecting the game playability. We developed the system for Android portable devices along with a distributed game that shows the advantages of the VFC-reckon model for ad-hoc networks.

Keywords

Mobile Devices, Ad-Hoc Network, Consistency, Interest Management, DeadReckoning, Multiplayer Games

Conteúdo

1 Introdução **1**

2 Trabalho Relacionado **5**

2.1 Redução do impacto da Latência 6

2.1.1 Atraso da apresentação local 6

2.1.2 Dead Reckoning 7

2.2 Mecanismos ao nível da rede 8

2.2.1 Atribuição de Prioridades aos pacotes 8

2.2.2 Compressão e Agregação de pacotes 9

2.3 Modelos de Consistência 9

2.3.1 Replicação Pessimista 10

2.3.2 Replicação Optimista 10

2.3.3 Limitação da Divergência 10

2.3.4 Recuperação de Estado 11

2.4 Gestão de Interesse 11

2.4.1 Aura de Interesse 12

2.4.2 Campo de Visão 13

2.4.3 Divisão em Regiões 14

2.4.4 Aura de Interesse com Regiões 16

2.5 Sistemas Académicos - Gestão de Interesse 17

2.5.1 Donnybrook 17

2.5.2 Ring 18

2.5.3 A3 19

2.5.4 Vector Field Consistency 20

2.6 Dispositivos Móveis e Redes ad-hoc 20

2.7 Sumário 22

3 Arquitectura **23**

3.1 Vector Field Consistency (VFC) 24

3.1.1 Anéis de Consistência 25

3.1.2 Graus de Consistência . . . 25
3.1.3 Generalizações do VFC . . . 26
3.1.4 Especificação do Modelo de Consistência . . . 26
3.1.5 Vantagens do VFC . . . 27
3.2 DeadReckoning . . . 27
3.2.1 Tipos de estimações . . . 29
3.2.1.A Trajectórias lineares . . . 30
3.2.1.B Trajectórias circulares . . . 30
3.2.1.C Outras trajectórias . . . 31
3.3 VFC-reckon . . . 32
3.4 Arquitectura . . . 33
3.4.1 Leitura e Escrita de Objectos . . . 34
3.4.2 Serialização de mensagens . . . 35
3.4.3 Propagação de actualizações . . . 36
3.4.4 Gestor de Sessão . . . 37
3.4.5 Aplicação do VFC-reckon . . . 38
3.4.5.A Gestor de Consistência de Objectos . . . 39
3.4.5.B Módulo de DeadReckoning . . . 39
3.5 Sumário . . . 40

4 Implementação **43**
4.1 API do Sistema . . . 44
4.2 Módulo de Serialização . . . 45
4.3 Camada de Comunicações . . . 46
4.4 Representação dos objectos partilhados . . . 47
4.5 Especificação do Módulo de DeadReckoning . . . 47
4.6 Gestão de Consistência de Objectos . . . 48
4.7 Especificação do Modelo de Consistência VFC-reckon . . . 49
4.8 Plataforma Android . . . 50
4.9 Jogo Multi-jogador Asteroids . . . 51
4.9.1 Gestão do estado do jogo . . . 51
4.9.2 Objectos do Jogo . . . 53
4.9.3 Gestor de Objectos do Jogo . . . 54
4.9.4 Servidor do Jogo . . . 54
4.9.5 Componente gráfica do jogo . . . 55
4.9.6 Controles do jogo . . . 57
4.10 Sumário . . . 57

5 Avaliação **59**
5.1 Avaliação Qualitativa 61
5.2 Avaliação Quantitativa 63
5.2.1 Largura de banda 63
5.2.1.A Módulo de Serialização 64
5.2.1.B Avaliação Sintética 65
5.2.1.C Avaliação num jogo real 66
5.2.2 CPU 67
5.2.2.A Ronda do Servidor 67
5.2.2.B Módulo de DeadReckoning 68
5.2.2.C Módulo de Serialização 70
5.2.2.D Impacto no jogo com o VFC-reckon 70
5.2.3 Memória 72
5.3 Sumário 74

6 Conclusões **75**
6.1 Trabalho Futuro 77

A Anexos **83**
A.1 API do Sistema 84
A.2 Diagramas de Classes do Sistema 85

Lista de Figuras

2.1 Aura de Interesse de um Jogador 12
2.2 Campo de visão com ângulo de 180º 13
2.3 Campo de visão com obstáculos 14
2.4 Representação de partições de regiões a) Quadrados b) Hexágonos c) Brickworks 16

3.1 Exemplo do VFC com 3 anéis de consistência e com um pivô 24
3.2 Exemplo de um cenário de erro na estimação de posições 28
3.3 Exemplo de estimações de posições para uma trajectória linear 30
3.4 Exemplo de estimações de posições para uma trajectória circular 31
3.5 Arquitectura do sistema 33
3.6 Máquinas de estados de Gestor de Sessão do Cliente e Servidor 37
3.7 Arquitectura interna do módulo de DeadReckoning 39

4.1 Exemplo da especificação do phi 49
4.2 *Print screen* do nosso Jogo Asteroids 55
4.3 Figuras usadas para representar os objectos do jogo no ecrã. 56

5.1 *Print screen* do nosso Jogo Asteroids 62
5.2 Tamanho de mensagens contendo 5 DataUnits 64
5.3 Largura de banda usada pelo sistema por 1 DataUnit 65
5.4 Largura de banda usada pelo sistema durante um jogo normal 66
5.5 Tempo de CPU usado pela função de ronda do Servidor 67
5.6 Tempo de CPU usado pela função de estimação de novas posições 69
5.7 Tempo médio para a escrita e leitura de 5 DataUnits 69
5.8 Comparação da frame-rate de VFC-reckon 160 vs Basic 40 71
5.9 Descrição detalhada de tempo de CPU usado 71
5.10 Memória alocada durante a execução pelo sistema e jogo 73

A.1 Diagrama de classes da Camada de API do Cliente 85
A.2 Diagrama de classes da Camada de API do Servidor 86
A.3 Diagrama de classes do Módulo de DeadReckoning 86

A.4 Diagrama de classes das estruturas mais relevantes da Camada de Sessão 86
A.5 Diagrama de classes da Camada de Sessão do Servidor 87
A.6 Diagrama de classes da Camada de Comunicações do Cliente 87
A.7 Diagrama de classes da Camada de Comunicações do Servidor 88

1

Introdução

Os telemóveis são dispositivos que têm vindo a ganhar uma grande importância na sociedade. Nos dias de hoje qualquer pessoa possui um ou mais dispositivos deste género usando-os para trabalho ou para lazer. Nos últimos anos o desempenho do hardware destes dispositivos tem vindo a crescer quase exponencialmente. Se antes um telemóvel era um dispositivo unicamente destinado para trocar chamadas ou sms, hoje é um dispositivo altamente avançado possibilitando aos utilizadores o uso de aplicações avançadas, como a navegação na Internet, integração de um Sistema de Posicionamento Global (GPS) ou mesmo jogos 2D e 3D.

Os jogos multi-jogador para estes dispositivos começaram a aparecer nos últimos anos com a introdução de novos protocolos de comunicação como o Bluetooth[1] ou o WiFi[2]. Este trabalho foca-se nestes jogos tendo como suporte uma rede ad-hoc. Supondo um cenário em que existem duas pessoas com dispositivos deste género, elas podem competir entre si num jogo multi-jogador em qualquer lugar, seja num autocarro, num centro comercial ou em casa desde que estejam relativamente próximos um do outro. O ambiente ad-hoc torna possível a execução destes jogos sem recorrer a quaisquer infra-estruturas auxiliares; apenas são necessários os telemóveis dos jogadores.

Para qualquer tipo de jogos que impliquem vários jogadores, tendo cada um o seu dispositivo, é necessário manter a consistência entre os vários dispositivos. Esta consistência é necessária para que os vários jogadores tenham a mesma percepção do estado do jogo, mantendo uma jogabilidade agradável. A consistência entre os dispositivos é mantida através da troca de mensagens entre eles. Estas mensagens são eventos do jogo que acontecem nos dispositivos e que precisam de ser difundidos para conseguir manter a lógica do jogo o mais actualizado possível. A consistência acaba por estar directamente relacionada com o número de mensagens pois, se queremos ter os vários dispositivos o mais sincronizados possível, tem que existir um grande número de mensagens a serem transmitidas.

Uma excessiva troca de mensagens entre os dispositivos pode levantar vários problemas. Os adaptadores de rede existentes nestes dispositivos representam um grande consumo de bateria neste tipo de aplicações. O envio e recepção de mensagens implica também o processamento das mesmas o que corresponde a um maior consumo de CPU e de bateria.

A latência é um problema associado a qualquer aplicação distribuída seja ela um jogo ou não. As técnicas existentes[3, 4] conseguem mascarar a latência existente nas redes, mas não a conseguem eliminar. Actualmente são usadas técnicas de atribuição de prioridades aos pacotes em redes congestionadas, atrasos aplicados no lado do Cliente e técnicas de Dead Reckoning. O Dead Reckoning é uma técnica que mascara a latência do lado do jogador, tentando prever qual a próxima localização dos objectos no mapa.

A replicação dos dados entre os vários dispositivos permite reduzir o tempo de acesso à informação devido à maior disponibilidade da informação. A criação de réplicas dos dados introduz o problema da manutenção da consistência dos dados pois agora quando existem novas actualizações é necessário fazê-las chegar às réplicas. Existem dois tipos de modelos que lidam com a replicação: Replicação Optimista [5] e Replicação Pessimista [5]

Com a Replicação Pessimista[5] conseguimos sempre uma elevada consistência. Quando ocorre uma actualização, a réplica é obrigada a propagar as alterações a todas as outras, e só pode prosseguir quando receber uma confirmação das outras réplicas.

A Replicação Optimista[5] é um modelo de consistência muito usado em jogos multi-jogador. Através da Replicação Optimista conseguimos reduzir o número de mensagens necessárias para obter consistência em dados replicados entre os jogadores. Ao contrário de modelos Pessimistas, a Replicação Optimista permite que os jogadores possam ler dados inconsistentes.

Os mecanismos de Gestão de Interesse[6–9] são fundamentais em jogos multi-jogador. A Gestão de Interesse funciona como um filtro. Um jogador apenas recebe eventos que realmente lhe interessem, deixando de parte eventos irrelevantes. Estes filtros de eventos só se aplicam quando não têm um grande impacto na jogabilidade. A Gestão de Interesse é sempre aplicada segundo um critério como, por exemplo, a proximidade entre os objectos e o jogador.

Os sistemas existentes[9–11], que aplicam técnicas Gestão de Interesse em jogos multi-jogador, não estão focados para as limitações que os telemóveis apresentam, como a autonomia limitada da bateria, fraca capacidade de processamento e de memória. Os sistemas mostram ter pouca flexibilidade e alguns estão orientados para um tipo de jogo específico, não permitindo a aplicação do sistema em jogos de outro tipo.

O objectivo deste trabalho é aumentar a escalabilidade dos jogos através da redução do número de mensagens o que corresponde a um menor consumo de CPU e largura de banda dos dispositivos móveis. O nosso trabalho consiste na concepção e desenvolvimento de um modelo de consistência chamado VFC-reckon. O VFC-reckon corresponde à junção do modelo de consistência VFC (Vector Field Consistency)[12], baseado em técnicas de Gestão de Interesse, com técnicas de DeadReckoning[4]. O nosso sistema foi desenvolvido para a plataforma Android, e desenvolvemos um jogo que demonstra as suas vantagens para ambientes móveis em redes ad-hoc. O sistema é um *middleware* simples para os programadores de jogos multi-jogador usarem e que os vai abstrair do processo de manutenção de consistência entre os estados dos vários jogadores. A extensibilidade do sistema permite ao programador implementar novos protocolos de comunicação, usar outros modelos de consistência para além do VFC e

estender o módulo de DeadReckoning com novos tipos trajectórias.

Este documento tem a seguinte organização: no Capítulo 2 iremos abordar o trabalho relacionado nesta área, focando-nos nas principais técnicas e nos sistemas mais usados; no Capítulo 3 é descrito o Modelo de Consistência e a Arquitectura da solução; no Capítulo 4 descrevemos a implementação do sistema, bem como a sua API; o Capítulo 5 contém a avaliação do trabalho, dividido nas componentes de Quantitativa e Qualitativa e por último no Capítulo 6 encontram-se as conclusões e o trabalho futuro.

2

Trabalho Relacionado

Conteúdo

2.1 Redução do impacto da Latência **6**
2.1.1 Atraso da apresentação local 6
2.1.2 Dead Reckoning 7
2.2 Mecanismos ao nível da rede **8**
2.2.1 Atribuição de Prioridades aos pacotes 8
2.2.2 Compressão e Agregação de pacotes 9
2.3 Modelos de Consistência **9**
2.3.1 Replicação Pessimista 10
2.3.2 Replicação Optimista 10
2.3.3 Limitação da Divergência 10
2.3.4 Recuperação de Estado 11
2.4 Gestão de Interesse **11**
2.4.1 Aura de Interesse 12
2.4.2 Campo de Visão 13
2.4.3 Divisão em Regiões 14
2.4.4 Aura de Interesse com Regiões 16
2.5 Sistemas Académicos - Gestão de Interesse **17**
2.5.1 Donnybrook 17
2.5.2 Ring 18
2.5.3 A3 19
2.5.4 Vector Field Consistency 20
2.6 Dispositivos Móveis e Redes ad-hoc **20**
2.7 Sumário **22**

Nesta secção vamos abordar as principais técnicas usadas para a redução do impacto de latência e redução do número de mensagens necessárias para a manutenção da consistência em jogos multi-jogador. Serão também apresentados vários sistemas existentes baseados em técnicas de Gestão de Interesse. Por último, será apresentado o trabalho relacionado sobre dispositivos móveis e redes ad-hoc.

2.1 Redução do impacto da Latência

A latência entre dois nós de uma rede é um problema existente nos jogos multi-jogador. A existência de latência significa que as mensagens levam um certo tempo, desde a sua emissão, até chegarem aos seus receptores. A latência pode originar situações paradoxais como, por exemplo, num jogo de corridas haver 2 jogadores em primeiro lugar ao mesmo tempo.

As alternativas existentes [3, 4] não conseguem eliminar o problema da latência. Actualmente, o que se faz é aplicar técnicas que reduzem o impacto causado pela latência.

Uma das técnicas [3] é o atraso na apresentação local ao jogador, o que implica que os eventos sejam atrasados numa fila de eventos.

Outra das técnicas existentes [4] é o uso de mecanismos para estimação de eventos futuros. A posição do jogador altera-se no decorrer de um jogo sendo possível prever a próxima posição do jogador, com algum intervalo de confiança, tendo como base a sua posição actual e o histórico de posições anteriores.

2.1.1 Atraso da apresentação local

O atraso na apresentação local do estado do jogo[3] ao utilizador consegue reduzir o impacto da latência num jogo multi-jogador. Nesta técnica, os eventos produzidos pelos jogadores não são aplicados de imediato no estado do jogo. Os eventos são atrasados numa fila de eventos que são aplicados ao estado do jogo ao fim de um pequeno intervalo de tempo. Este intervalo de tempo tem de ser o mínimo possível, pois um grande atraso na aplicação dos eventos pode notar-se na jogabilidade deixando o jogador insatisfeito. Durante este intervalo de tempo, são recebidos todos os eventos de todos os jogadores de maneira a garantir que todos vão aplicar os mesmos eventos, pela ordem correcta, e vão ter um estado do jogo consistente entre eles. Para garantir a recepção de todos os eventos o intervalo de tempo tem de ser igual à latência máxima da rede, caso contrário, podem faltar eventos na fila de eventos.

A eficácia desta solução depende muito do tipo de jogo a que é aplicada. No caso de jogos que precisem de uma jogabilidade de resposta muito rápida, como é o caso dos jogos de corridas, ou jogos de tiros, esta solução tem menos sucesso pois o intervalo de

tempo tem que ser muito pequeno. Outra desvantagem desta solução é a possibilidade de ocorrer variância nas latências. No caso das redes sem fios, as latências podem variar bastante como, por exemplo, quando aparece um obstáculo entre os dispositivos, sendo difícil definir um valor para o intervalo de tempo.

2.1.2 Dead Reckoning

O Dead Reckoning[4] é uma técnica usada para reduzir o impacto da latência da rede, através da estimação de futuras mensagens de actualização. Com esta técnica conseguimos reduzir o impacto do atraso das mensagens, bem como a perda das mesmas.

Em jogos multi-jogador, para um jogador ver a posição actual de outro jogador, é necessário uma taxa constante de envio de mensagens, com as posições actualizadas de cada objecto, entres todos jogadores. O aumento do intervalo de envio das mensagens de posições cria uma descontinuidade no movimento dos objectos. O DeadReckoning é muitas vezes usado para corrigir esta descontinuidade, estimando as posições intermédias.

Os métodos de estimação de eventos de localização tentam antecipar a posição futura de um objecto, baseando-se na sua posição actual e nas posições anteriores. Para este cálculo usam-se dados como direcção, velocidade, aceleração e polinómios. O uso de polinómios permite uma melhor modelação quando a trajectória descreve a forma de, por exemplo, uma curva, tornando a previsão mais correcta.

Tratando-se de um processo de estimação de mensagens, por vezes as mensagens estimadas têm um grau de erro associado. O erro entre as posições estimadas e as posições reais pode ser mitigado de várias maneiras. Uma solução básica para este problema é colocar de imediato o objecto na posição correcta o que pode originar saltos repentinos, levando a uma diminuição da jogabilidade. Existem soluções mais avançadas que tentam convergir a posição do jogador usando uma convergência linear. Uma convergência linear escolhe vários pontos entre as duas posições. Com este método, a convergência entre os dois pontos é feita de uma maneira mais suave, reduzindo assim o impacto negativo na jogabilidade.

A maior vantagem desta técnica é permitir o aumento do intervalo de envio de novas mensagens de actualização de posição. Durante a ausência das mensagens são aplicadas as técnicas de estimação tentando assim compensar a falta das mesmas. A desvantagem desta solução é que depende muito da qualidade da previsão que é feita. O sucesso das estimações depende muito do tipo de jogo e do modo como o jogador joga. As acções dum jogador que jogue mais bruscamente e que faça movimentos rápidos, são mais difíceis de estimar, que as acções de um jogador que jogue de uma

maneira mais suave e lenta.

Em jogos em que o tipo do movimento dos objectos não varia muito como, por exemplo, os jogos de corridas é mais fácil prever qual vai ser a próxima posição do jogador. Isto acontece porque um jogador normalmente segue a trajectória da pista e as posições tendem a repetir-se em todas as voltas à pista. Em jogos como First Person Shooters torna-se mais difícil antecipar a próxima acção do jogador devido à quantidade de acções que um jogador pode efectuar em qualquer momento.

2.2 Mecanismos ao nível da rede

Existem várias soluções aplicadas ao nível da rede que reduzem a latência e a largura de banda necessária para o fluxo de pacotes pelos nós da rede. Estas soluções são aplicáveis a qualquer tipo de aplicações distribuídas e, em particular, a jogos multi-jogador. Duas soluções interessantes nesta área são: 1) Atribuições de um nível de urgência e relevância aos pacotes e 2) Agregação e compressão dos pacotes.

2.2.1 Atribuição de Prioridades aos pacotes

A atribuição de uma urgência e relevância dos pacotes [13] permite mascarar a latência introduzida devido ao congestionamento dos pacotes na rede. Pacotes com uma urgência mais elevada são enviados primeiro que pacotes menos urgentes, garantindo que chegam ao seu destino no menor tempo possível, reduzindo assim a latência introduzida por congestionamentos na rede. Os pacotes menos urgentes vão sendo atrasados até já não haver pacotes urgentes para enviar. A relevância atribuída aos pacotes é uma medida de fiabilidade muito útil quando aplicada a protocolos que não garantam qualquer tipo de fiabilidade como, por exemplo, o protocolo UDP. Um pacote com uma relevância alta significa que será retransmitido várias vezes até se confirmar a recepção no seu destino. Isto significa que existe um contador de reenvio de mensagens que se vai decrementando sempre que a mensagem não chega ao seu destino. Quando um pacote tem uma relevância baixa, significa que a sua perca não vai ter um impacto grande no jogo.

Esta solução tem duas grandes desvantagens quando aplicada a jogos multi-jogador. Em primeiro lugar não tem em conta qualquer tipo de localidade das entidades do jogo, ou seja, não são considerados factores como, por exemplo, a distância entre as duas entidades. Outra desvantagem é o facto de a atribuição ter que ser feita na fase de desenho do jogo. A urgência e relevância dos eventos estão incorporadas nas classes dos eventos. Desta maneira não é possível alterar a urgência e relevância de um evento

durante a execução do jogo. Esta desvantagem tem um grande impacto num jogo multi-jogador, em que um evento pode ter urgências diferentes consoante o estado do jogo.

2.2.2 Compressão e Agregação de pacotes

Actualmente existem técnicas de compressão e agregação de pacotes [14] que permitem reduzir a largura de banda necessária para a transferência de pacotes.

A ideia base da compressão é conseguir transmitir a mesma mensagem usando o menor número de bytes possível. Nas técnicas de compressão a mensagem inicial é comprimida no emissor, enviada ao receptor e por fim descomprimida, obtendo assim a mensagem inicial enviada pelo emissor. As mensagens, como vão comprimidas, ocupam menos bytes e são transmitidas mais rapidamente. A desvantagem desta solução é o custo associado aos passos de compressão e descompressão adicionados no emissor e receptor, introduzindo mais computação e tempo.

A agregação de pacotes consiste no agrupamento de pacotes num só pacote. Desta maneira em vez de enviar vários pacotes pequenos, os pacotes são agregados num só. Esta solução reduz a largura de banda necessária reduzindo os cabeçalhos dos pacotes IP. Em vez de ter n cabeçalhos de n mensagens, como os pacotes são agregados num só, é utilizado apenas um cabeçalho. A agregação pode ser feita com base num temporizador ou num contador.

Em agregações com um temporizador os pacotes são agregados até expirar um limite temporal. Em agregações com contador, os pacotes são agregadas até acumular n pacotes.

A agregação com base num temporizador tem como maior desvantagem a introdução de mais latência na rede, pois os pacotes agora só são enviados após expirar um limite temporal. Agregação com base num contador tem como desvantagem o intervalo de tempo até atingir n mensagens. Como só são enviados pacotes até acumular n pacotes corremos o risco de esperar muito tempo até acumular os n pacotes.

2.3 Modelos de Consistência

A replicação de dados consiste em manter múltiplas cópias de dados, chamadas réplicas, em computadores distintos. Com a replicação conseguimos ter maior disponibilidade no acesso aos dados mesmo quando uma das réplicas não está disponível. O uso de réplicas também pode servir com mecanismo de redução de latência permitindo aos utilizadores acederem a réplicas geograficamente mais próximas. Com a replicação dos dados nasce a necessidade da manutenção e consistência das réplicas, actualizando-as regularmente de modo a que todos os dados replicados estejam consistentes entre

si. Para esta consistência podem ser usados modelos que podem ser Optimistas ou Pessimistas.

2.3.1 Replicação Pessimista

As técnicas de Replicação Pessimista [5] adoptam um modelo de consistência equivalente ao que se dispõe quando existe uma única cópia. Quando ocorre uma actualização, a réplica é obrigada a propagar as alterações a todas as outras, e só pode prosseguir quando receber uma confirmação das outras réplicas. Este tipo de modelos chamam-se pessimistas por não permitirem concorrência de operações entre as réplicas. Este modelo assume que qualquer tipo de concorrência pode originar conflitos, ou seja, incoerência dos dados, então, não permite que haja concorrência.

Este tipo de técnicas são pouco escaláveis e não se adequam aos jogos multi-jogador. Uma réplica para actualizar o seu estado precisa da confirmação das outras o que implica uma grande troca de mensagens. Por outro lado, se uma das réplicas não estiver disponível então todo o sistema deixa de funcionar.

2.3.2 Replicação Optimista

O que distingue as técnicas de Replicação Optimista[5] das pessimistas é o controlo de concorrência. As técnicas optimistas permitem que os dados existentes em réplicas sejam acedidos sem que estas se sincronizem com outras réplicas. Neste tipo de soluções assume-se que raramente vão ocorrer problemas que possam originar conflitos. Os conflitos ocorrem quando duas réplicas do mesmo objecto são alteradas mas ainda não propagaram as alterações. As alterações das réplicas são feitas localmente e mais tarde são propagadas para as outras réplicas. Os conflitos são resolvidos quando são detectados e podem ser resolvidos de forma automática ou manual.

Esta solução possibilita uma maior disponibilidade e diminui o número de mensagens na rede. A Replicação Optimista é muito usada em jogos multi-jogador. Neste tipo de jogos é necessário sincronizar o estado das réplicas existentes nos vários clientes mantendo o estado consistente. O ideal é encontrar um meio-termo entre manter o jogo altamente consistente, e reduzir o número de mensagens necessárias. Um jogador normal tolera alguma inconsistência no jogo desde que isto não afecte a sua jogabilidade [3]. É a partir desta tolerância que se pode aplicar a redução de mensagens.

2.3.3 Limitação da Divergência

As técnicas de Limitação da Divergência [15–18] são técnicas de replicação optimista que toleram que o estado das réplicas divirjam entre si, desde que não ultrapassem um

certo limite estabelecido. Estas técnicas são aplicáveis a jogos multi-jogador e têm como objectivo reduzir o número de mensagens trocadas.

Os limites, normalmente, são temporais e de ordem. Limites temporais servem para limitar o intervalo de tempo que uma réplica pode estar sem se sincronizar. Por exemplo, se atribuirmos um limite de 3 segundos, então isto significa que, no máximo, esta réplica só está inconsistente durante 3 segundos, nunca ultrapassando este limite. Os limites de ordem referem-se ao número de actualizações que ainda não foram aplicadas à réplica. Por exemplo, uma réplica com um limite N é considerada válida até se detectar que tem N actualizações pendentes. Após isto a réplica é actualizada.

O TACT[16] é um sistema que aplica técnicas de Limitação de Divergência como limites temporais e de ordem. Trata-se de um *middleware* aplicado às bases de dados e que possui flexibilidade na especificação dos limites permitindo combinações dos mesmos.

Sendo o TACT um sistema focado em bases de dados, a sua aplicação em jogos multi-jogador é pouco eficiente pois não tem em conta a lógica associada ao jogo como, por exemplo, a noção de mapa de jogo, jogador, ou entidades do jogo.

2.3.4 Recuperação de Estado

O Trailing State Synchronization (TSS)[19] é um algoritmo que permite a recuperação do estado do jogo quando é detectada uma incoerência na actualização do estado. Este algoritmo mantém réplicas do estado actual do jogo (estado principal), mas com um atraso temporal entre elas. Quando é detectada uma incoerência, o conteúdo de uma das réplicas é copiada para o lugar do estado principal. Como no estado anterior não existe ainda incoerência devido ao atraso temporal, as actualizações são novamente executadas na ordem correcta. Esta solução tem como desvantagem o armazenamento de várias réplicas do estado actual do jogo, necessitando de memória extra para o seu funcionamento.

2.4 Gestão de Interesse

Nos jogos multi-jogador em que o jogador é representado por um avatar[1] no mundo virtual, existe uma noção de percepção de objectos. O mundo virtual do jogo pode ser representado pelo mapa e todas as entidades do jogo. As entidades do jogo podem ser armas, paredes, avatar do jogador, etc... Cada jogador mantém uma cópia local do mundo virtual, que necessita de manter consistente. Acções efectuadas por jogadores, ou outras entidades do jogo, têm que ser propagadas para os jogadores afectados. Uma

[1] O avatar é a entidade que representa o jogador dentro do mundo virtual do jogo como, por exemplo, um soldado, uma figura ou uma nave.

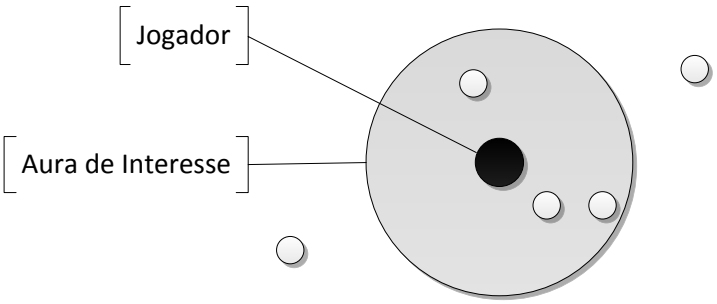

Figura 2.1: Aura de Interesse de um Jogador

solução simples para este problema seria todas as entidades do jogo enviarem a todos os jogadores as suas acções. Esta solução tem um grande problema, um jogador vai receber actualizações de entidades que podem não lhe interessar, o que pode causar um excesso de mensagens na rede, que num ambiente de dispositivos móveis não é tolerável.

A Gestão de Interesse tira partido da noção de interesse do avatar dentro do jogo. Um avatar só precisa de receber eventos de um pequeno conjunto de entidades relevantes para ele. Esta técnica aplica-se na perfeição em jogos multi-jogador para dispositivos móveis em redes ad-hoc. Sendo um mecanismo que tem como objectivo reduzir o número de eventos que um jogador precisa de receber, então conseguimos reduzir o número de mensagens trocadas entre os dispositivos, reduzindo a largura de banda usada pelo jogo bem como a quantidade de mensagens processadas.

Existem três técnicas principais na área de Gestão de Interesse: Aura de Interesse, Campo de Visão e Divisão em Regiões.

2.4.1 Aura de Interesse

A aura de interesse[6–8] tem como base a noção de interesse de um avatar. Um avatar dentro de um jogo apenas se consegue aperceber de outras entidades do jogo se estiverem dentro de uma certa distância de raio R. Normalmente, entidades que estejam relativamente perto do jogador têm um alto grau de detalhe e são muito perceptíveis, por outro lado, entidades distantes são muito pouco perceptíveis para o jogador. Uma aura pode ser vista como um círculo/esfera que rodeia o avatar como exemplificado na Fig. 2.1.

Neste modelo de Gestão de Interesse a aplicação cliente do jogador subscreve eventos que ocorram dentro da aura do avatar. Existem variações deste modelo que usam a distância do objecto ao avatar para calcular a frequência de envio de eventos, ou seja,

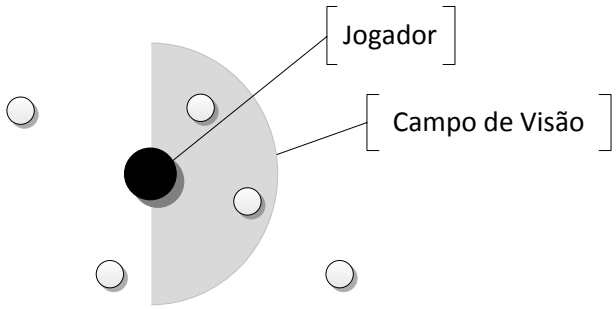

Figura 2.2: Campo de visão com ângulo de 180º

quanto mais perto estiver o objecto do avatar, mais frequentemente são enviados os eventos.

2.4.2 Campo de Visão

A aplicação de técnicas como o campo de visão[6, 9] é muito útil para Gestão de Interesse. Existem duas ideias principais que tornam esta ideia relevante: a amplitude de visão e a existência de obstáculos.

Um jogador normalmente só tem uma percepção de 180º a partir da posição em que se encontra[11], ou seja, não consegue ver objectos que se encontram atrás dele. Com isto podemos concluir que objectos que estejam atrás do jogador, não necessitam de enviar eventos ao jogador, ou pelo menos não com tanta frequência. Na Fig. 2.2 encontra-se um exemplo de um campo visão com um ângulo de percepção de 180º. Porém, é necessário ter atenção, a objectos que estejam atrás do jogador, mas relativamente próximos. Se não houver qualquer tipo de envio de eventos de objectos que estejam atrás do jogador, se o jogador se virar de repente pode não ver de imediato os objectos que ali se encontram, o que pode afectar negativamente a jogabilidade. O que normalmente se aplica é um pequeno raio em redor do avatar. O avatar passa a receber sempre eventos de objectos que estejam muito próximos mesmo que estes se encontrem atrás do si.

Outra componente importante do campo de visão é a existência de obstáculos como representado na Fig. 2.3. Um jogador dentro de um jogo não consegue visualizar objectos que estejam atrás de obstáculos, como por exemplo, paredes, muros ou árvores. Como o jogador não consegue ver os objectos, então tem menor interesse neles, reduzindo assim o número de eventos que precisa de receber e processar.

Existem análises de vários algoritmos de Gestão de Interesse[6] que usam o conceito de Campo de Visão. Nesta análise podemos ver que o uso de técnicas de Campo

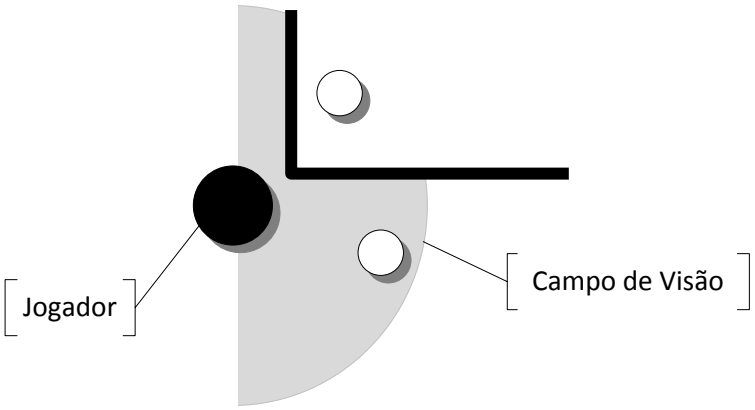

Figura 2.3: Campo de visão com obstáculos

de Visão, na existência de obstáculos, permitem reduzir o número de eventos que um jogador recebe até um factor de 6 vezes. O uso de mecanismos de Ray Visibility seria ideal para detectar que objectos estão visíveis ou não. O problema destes mecanismos é o consumo de recursos, mais propriamente, de CPU, para efectuar os cálculos necessários das intersecções. Esta desvantagem tem um peso enorme na escalabilidade de um jogo em termos dos seus recursos de CPU, o que se agrava se estivermos no contexto de dispositivos de capacidades limitadas de CPU e bateria como os dispositivos móveis. Uma solução eficiente passa pelo algoritmo de TilePathDistance que usa triangulação do espaço do jogo através de polígonos. Os contornos dos polígonos são formados a partir dos limites do mundo e dos obstáculos existentes. Este algoritmo implica um pré-processamento para a triangulação do mapa.

2.4.3 Divisão em Regiões

Na divisão do mundo virtual em regiões, o mundo virtual é repartido em várias regiões que não se intersectam entre si. Neste tipo de soluções de Gestão de Interesse, a aplicação cliente do jogador subscreve todos os eventos que possam ocorrer dentro da região em que se encontra, ou seja, qualquer evento que ocorra numa entidade dentro da região do jogador é imediatamente propagada para ele.

A divisão em zonas é muito usada para efeitos de balanceamento de carga[20, 21] nos jogos do tipo Massively Multiplayer Online Role-Playing Game (MMORPG) como,

por exemplo, o World-Of-Worldcraft[2] ou EVE Online[3]. Como este tipo de jogos têm um grande número de jogadores são necessários múltiplos Servidores. Para distribuir melhor a carga dos Servidores, são atribuídas regiões a cada um deles. Um jogador apenas pode estar numa destas regiões, sendo que o seu estado é gerido pelo Servidor da região.

As Regiões são então divisões que podem ser visíveis ou invisíveis para o utilizador. Divisões visíveis são aquelas de que o jogador tem a percepção de onde acaba a região. Nestas divisões, a transição de uma região para outra é normalmente feita com o uso de objectos especiais dentro do jogo como, por exemplo, um portal, porta da sala ou um aeroporto. Jogos como, por exemplo, EVE Online, StarTrek Online[4] e GuildWars[5] aplicam esta técnica.

As divisões invisíveis são aquelas que são transparentes para o utilizador. Nestas divisões o utilizador não tem qualquer noção do fim da região e a transição do jogador entre regiões é feita de uma maneira automática e transparente. Jogos como, por exemplo, Warhammer Online[6] e FreeRealms[7] aplicam esta técnica.

A divisão e atribuição de zonas aos Servidores pode ser feita de uma maneira estática ou dinâmica. Se a atribuição for estática o mapa do jogo é dividido antes da execução e cada Servidor tem um conjunto fixo de regiões. Após feita a atribuição já não é possível alterá-la. A atribuição dinâmica significa que os Servidores podem alterar as dimensões das regiões que gerem e redistribuí-las por outros Servidores. Por exemplo, um Servidor que contenha um grande número de jogadores dentro da sua região, pode dividir a sua região ao meio, e atribuir essa metade a outro Servidor que não esteja sobrecarregado.

Um exemplo de divisão dinâmica é a divisão do mapa com base numa estrutura chamada N-Tree[8]. Cada região pode ser recursivamente dividida até um dado limite. As sub-regiões que resultam desta divisão são divididas pelos Servidores.

Estando o mundo virtual partido em várias regiões, um jogador pode circular entre elas. Um Servidor que gere uma região, quando detecta que um jogador está prestes a sair da região, tem que informar a aplicação cliente do jogador e informar o Servidor para o qual o jogador se dirige. É necessário informar a aplicação cliente do jogador porque o seu Servidor está prestes a mudar. Após mudar de região o estado do jogador passa a ser gerido por um novo Servidor. O Servidor antigo tem que informar o Servidor novo do seu novo jogador, transferindo todo o estado que contém acerca do jogador.

O tamanho das partições é um factor de grande peso neste tipo de soluções. Uma

[2] http://eu.battle.net/wow/en/
[3] http://www.eveonline.com/
[4] http://www.startrekonline.com/
[5] http://www.guildwars.com/
[6] http://www.warhammeronline.com/
[7] http://www.freerealms.com/

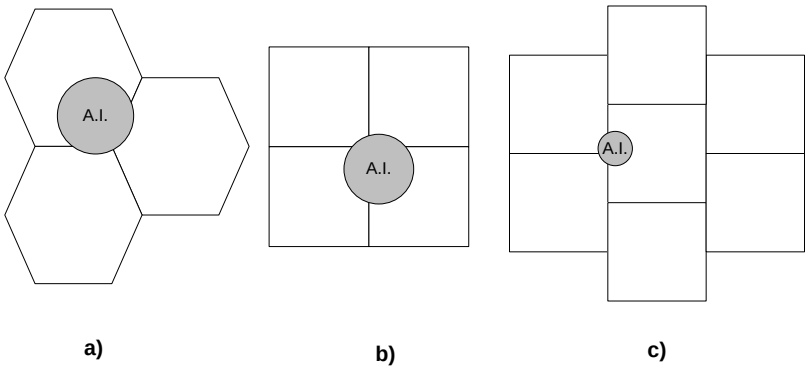

Figura 2.4: Representação de partições de regiões a) Quadrados b) Hexágonos c) Brickworks

região grande tem como desvantagem o elevado número de eventos que um jogador pode receber mas reduz a frequência de troca de Servidor associado ao jogador. Por outro lado, se uma região for pequena, um jogador recebe um número menor de eventos mas, em contrapartida, a transferência do jogador entre Servidores ocorre mais frequentemente.

2.4.4 Aura de Interesse com Regiões

Os mecanismos de Aura de Interesse e Divisão em Regiões podem ser aplicados em conjunto formando um mecanismo mais poderoso de Gestão de Interesse. Porém acrescenta um novo problema que não existia anteriormente, a proximidade de uma fronteira.

Como um jogador tem que receber actualizações de entidades que estejam dentro da sua aura, se um jogador se deslocar para perto de uma fronteira então vai ter de receber actualizações de entidades que estejam do outro lado da fronteira e que intersectem a sua aura. A troca de eventos de jogadores próximos das fronteiras mas em regiões diferentes, é um dos maiores problemas destas soluções e existem várias alternativas para resolver este problema.

Uma solução possível passa pelo *lock* de regiões e de objectos [22]. Um Servidor a executar um evento que afecta uma área do mapa, como por exemplo uma explosão, tem que pedir o *lock* para essa área. Todos os eventos são propagados usando mecanismos de Publish/Subscribe.

Outra das soluções é a criação de uma região de subscrição à distância D da fronteira[23]. Todos os objectos que estejam a uma distância menor que D da fronteira, vão receber actualizações do outro lado da fronteira.

A forma que as regiões têm é um factor de peso quando se aplica uma aura em con-

junto com divisão em regiões. Este problema é fácil de perceber analisando a Fig. 2.4. Podemos ver que se forem utilizadas regiões com a forma de quadrados, a aura do jogador intersecta até três regiões, o que significa que seria necessário comunicar com três Servidores para haver troca de eventos. A forma de um hexágono é mais apropriada porque apenas é possível, no máximo, intersectar duas regiões, porém, o cálculo de distâncias usando hexágonos é mais complexa quando comparado com quadrados.

O Brickworks [7] é um tipo de partição que usa a simplicidade do cálculo de quadrados, com a vantagem do mesmo número de intersecções que a forma de um hexágono tem. Através da divisão das regiões de uma maneira intervalada, é possível obter os mesmos resultados da forma dum hexágono. A única condição desta solução é que o diâmetro da aura seja menor que metade do lado quadrado.

2.5 Sistemas Académicos - Gestão de Interesse

Nesta secção vão ser apresentados os sistemas académicos que mais se enquadram em jogos multi-jogador. Estes sistemas aplicam técnicas de Gestão de Interesse como Aura de Interesse e Campo de Visão.

2.5.1 Donnybrook

O Donnybrook[10] é um sistema para jogos multi-jogador aplicado em arquitecturas Peer-to-Peer (P2P) e que usa várias técnicas de Gestão de Interesse. O Donnybrook baseia-se 3 princípios: 1) Jogadores têm uma atenção limitada 2) Interacções importantes têm que ser feitas num curto intervalo de tempo e têm que ser consistentes 3) Realismo não deve ser afectado pela precisão.

O primeiro princípio tem como base o número de entidades nas quais uma pessoa se consegue focar. Um avatar dentro de um jogo só se consegue focar num dado conjunto de entidades, o que significa que o interesse de um jogador é normalmente maior para certas entidades e menor para outras. As 3 regras que definem o interesse de um jogador numa entidade são a sua proximidade, mira do jogador e interacções recentes que tenham ocorrido.

O segundo princípio tem em conta interacções importantes e urgentes que ocorram entre dois jogadores. Supondo um cenário em que dois jogadores estão a tentar eliminar-se num jogo, é necessário que as interacções sejam o mais rápidas e consistentes possíveis, caso contrário a jogabilidade do jogo pode estar em risco. O Donnybrook usa um mecanismo de Entendimento Rápido entre Nós usando chamadas assíncronas entre apenas os jogadores envolvidos nessa interacção. Como usa um canal diferente dos canais usados para a transmissão de eventos, a sua transmissão é mais rápida e

fiável.

O último princípio foca-se na lógica do jogo. Entidades que não estejam dentro do foco do jogador enviam os seus eventos menos frequentemente. Eventos menos frequentes podem resultar em movimentações bruscas de entidades do jogo que podem violar a lógica do jogo. Um exemplo disto, são jogadores que estão numa posição e repentinamente aparecem noutro sítio. O Donnybrook aplica um mecanismo para guiar as entidades do jogo baseado em inteligência artificial.

Este sistema tem como principal desvantagem a dependência da lógica do jogo, estando orientado maioritariamente para jogos do tipo First Person Shooter (FPS). Uma das regras que define o interesse do jogador é o facto de a mira estar sobre o objecto. O conceito de mira deixa de fazer sentido quando aplicado em jogos que não sejam FPS.

2.5.2 Ring

O Ring[9] é um sistema para ambientes virtuais de vários utilizadores que pode ser aplicado a jogos multi-jogador. Este sistema aplica mecanismos de Gestão de Interesse pois tem como objectivo a redução de mensagens que um jogador tem que receber para manter o seu estado consistente. Esta redução das mensagens vem da oclusão existente nos mapas dos jogos multi-jogador. Um jogador não consegue ver entidades do jogo que estejam escondidos atrás de paredes ou outro tipo de objectos e, como tal, não precisa de saber da existência de entidades que não consegue visualizar, evitando assim mensagens que, do ponto de vista do jogador, não vão trazer qualquer benesse.

Os jogadores mantêm uma cópia do estado do jogo que pode conter réplicas de objectos. Estas são réplicas de objectos que estão no campo de visão de jogador. A comunicação entre os jogadores é feita exclusivamente através dos Servidores Ring, não havendo qualquer troca de mensagens entre os jogadores. Os Servidores Ring servem então como filtro de mensagens, sabendo quais dos jogadores são visíveis a partir de uma dada localização, reencaminhado os eventos só para os jogadores que se conseguem aperceber deles.

Para a determinação da visibilidade que um jogador tem, é feita uma pré-computação ao mapa do jogo dividindo-o em células. Para cada célula são calculadas as células que são visíveis a partir dela, tendo em conta a existência dos obstáculos e são gravados esses resultados. Durante um jogo, para saber se um jogador tem visibilidade para uma dada entidade, basta apenas determinar se essa entidade está numa das células visíveis calculadas anteriormente.

O Ring aplica um mecanismo de Gestão de Interesse com base no campo de visão do jogador que permite reduzir a largura de banda necessária dos clientes. O Ring apenas tem em conta se um jogador é visível ou não a partir de uma dada célula não tendo em

conta outros factores que podem influenciar o interesse do jogador como, por exemplo, a proximidade. Neste sistema o tráfego gerado não está dependente do número de jogadores total, mas sim do número de jogadores cujos campos visuais se intersectem. A eficiência desta solução depende do conjunto de obstáculos existente no mapa. A obrigatoriedade do tráfego passar pelos Servidores introduz uma maior latência, que pode aumentar consoante o número de Servidores.

2.5.3 A3

O A3 [11] é um algoritmo de gestão de Interesse, destinado a jogos do tipo MMORPG. Este algoritmo aplica os conceitos de proximidade, campo de visão e distância crítica. O interesse do jogador numa dada entidade é atribuído com base numa medida de relevância que varia entre 0 e 1. Uma relevância de 0 significa que o jogador não tem qualquer interesse na entidade não havendo qualquer envio de eventos. Por outro lado, uma relevância de 1 significa que a entidade é de extrema importância para o jogador originando uma grande frequência no envio de eventos.

A proximidade entre um jogador e uma dada entidade indica a relevância da entidade para o jogador. Quanto mais próximo da entidade, maior é a relevância da entidade o que implica maior frequência no envio de eventos.

Outro dos conceitos aplicados no A3 é o conceito de campo de visão. Um jogador no mundo virtual tem interesse apenas em entidades que pode ver. O jogador não precisa de receber eventos de entidades que estejam atrás dele, fazendo com que a sua relevância seja menor.

Se um jogador se virar de repente, usando apenas o mecanismo acima referido, a entidade pode levar algum tempo a ser perceptível para o jogador. Para evitar este problema o A3 introduz o conceito de distância crítica. A distância crítica trata-se de um pequeno círculo que rodeia o avatar e que indica que, para o jogador, qualquer entidade que esteja dentro do círculo tem relevância máxima. Assim o jogador já tem noção das entidades que possam estar atrás dele, fazendo com que entidades próximas apareçam de imediato quando ele se vira.

Uma desvantagem deste algoritmo é a visão global de aura. Todos os jogadores têm o mesmo tipo de aura e são aplicadas as mesmas métricas de distâncias. Nos jogos podem haver jogadores com atributos especiais que podem, por exemplo, ver outros jogadores que estejam a uma longa distância, o que ia implicar outro tipo de aura para estes jogadores. Este algoritmo não é flexível nesse sentido pois aplica a mesma aura a todos os jogadores apenas possibilitando uma aura global.

2.5.4 Vector Field Consistency

O Vector Field Concistency (VFC)[12, 24–26] é um modelo de consistência baseado em Gestão de Interesse. Este modelo utiliza o conceito de Aura de Interesse. Quando aplicado a jogos multi-jogador o VFC permite reduzir o número de mensagens necessárias para manter as réplicas dos jogadores o mais consistente possível. As réplicas são cópias locais de objectos do jogo que cada jogador mantém. O VFC tem como principais conceitos Anéis de Consistência e Graus de Consistência.

Os Anéis de Consistência são anéis que se formam em redor de uma entidade chamada pivô. Um pivô é um objecto especial que define o grau de consistência de todos os objectos que o rodeia. Um bom exemplo de pivô é o avatar do jogador. Cada anel formado em redor do pivô tem um grau de consistência diferente. Tal como no modelo de Aura de Interesse, objectos mais próximos do pivô têm um grau de consistência maior enquanto que objectos mais afastados têm um grau de consistência menor.

No VFC um Grau de Consistência é especificado num vector tridimensional. As 3 dimensões do vector são: Temporal, Sequencial e de Valor. A dimensão temporal define o tempo máximo que uma réplica pode permanecer válida sem receber novas actualizações. A dimensão Sequencial define o número de actualizações que uma réplica pode não aplicar e permanecer operacional. A dimensão de Valor indica a diferença máxima entre uma réplica e o objecto original.

Os principais pontos fortes do VFC são a sua facilidade de percepção do seu modelo de consistência baseado em pivôs e a flexibilidade na especificação de parâmetros do algoritmo como as 3 dimensões de um grau de consistência.

2.6 Dispositivos Móveis e Redes ad-hoc

Os dispositivos móveis são muito usados nos dias de hoje. Com a sua evolução em termos de Hardware ao longo dos últimos anos, hoje já podemos executar aplicações como, por exemplo, jogos 3D, o que era algo impensável à uns anos atrás. Apesar destes melhoramentos, ainda são muito grandes as limitações existentes nestes dispositivos. As principais limitações são a autonomia da bateria, a capacidade de processamento, capacidade de memória e a largura de banda.

A escolha no protocolo de comunicação entre os dispositivos móveis como, o WiFi[2] ou o Bluetooth[1] pode influenciar em muito o consumo de bateria de qualquer aplicação distribuída como, por exemplo, um jogo multi-jogador. Existem análises [27] que demonstram que o consumo de energia da tecnologia Bluetooth é muito menor que o consumo do WiFi. O consumo de energia de ambas as tecnologias varia consoante o intervalo de envio de mensagens e a largura de banda, ou seja, quanto maior o intervalo e menor

a largura de banda usada, menor é o consumo de energia associado à transmissão e recepção de mensagens.

As soluções actuais para jogos muti-jogador em ambientes ad-hoc para dispositivos móveis não estão focadas na lógica de jogo. Não existe qualquer aplicação de mecanismos de Gestão de Interesse, indispensáveis para a redução do número de mensagens em qualquer jogo multi-jogador.

Uma solução de partição do jogo[28] permite uma distribuição de carga entre o Servidor do jogo e os clientes. Quando o Servidor começa a ficar com poucos recursos disponíveis como, por exemplo, memória, CPU ou largura de banda, o jogo é repartido, distribuindo funções do Servidor pelos clientes existentes, aliviando assim a carga do Servidor. Esta divisão é feita tendo em conta a disponibilidade de memória, capacidade de processamento e largura de banda disponível nos clientes. A divisão tem que ser feita de uma maneira optimizada pois a partição da aplicação pode originar uma maior quantidade de tráfego na rede devido à distribuição das funções por vários clientes.

Uma rede ad-hoc é uma rede heterogénea que pode conter vários dispositivos de níveis de desempenho diferentes como telemóveis, tablets ou portáteis. A vantagem do uso de uma rede ad-hoc em jogos multi-jogador é que não são necessárias estruturas auxiliares para a comunicação entre os dispositivos como, por exemplo, um *router*. Os dispositivos comunicam através de canais de informação sem fios, cooperando entre eles para o reencaminhamento de pacotes. As duas arquitecturas mais usadas para jogos em redes ad-hoc são: Cliente-Servidor e Peer-to-Peer (P2P).

Na arquitectura Cliente-Servidor o dispositivo que corresponde ao Servidor tem a responsabilidade de gerir a lógica do jogo e actua como um coordenador da rede. Os clientes enviam os dados para o Servidor, que processa a informação, e envia respostas aos clientes afectados. Esta é uma arquitectura simples de implementar e tem como principal desvantagem a existência de apenas um Servidor. Como apenas existe um Servidor, se este se desligar, toda a rede fica parada enquanto o Servidor não se ligar ou for eleito um novo Servidor. A escalabilidade é outra das desvantagens que a arquitectura apresenta. O Servidor tem que lidar com todo o tráfego da rede, podendo tornar-se o *bottleneck* da rede.

Numa arquitectura P2P o dispositivo de cada jogador mantém uma cópia local do jogo e informa os dispositivos dos outros jogadores quando ocorrem actualizações. Esta arquitectura é mais difícil de implementar pois cada cliente tem que lidar com a lógica do jogo, não havendo uma entidade única para o efeito. Apesar desta desvantagem, uma arquitectura P2P tem uma maior tolerância a faltas que uma arquitectura Cliente-Servidor e uma maior escalabilidade. A paragem de um dispositivo não leva à paragem do sistema e não existe um ponto único na rede para onde todo o tráfego converge.

Existe uma solução, a nível arquitectural, de redes ad-hoc baseada em ZoneServers [29]. Esta solução tenta juntar as vantagens de uma arquitectura Cliente-Servidor com uma arquitectura P2P. Nesta arquitectura existem 2 tipos de nós na rede: Nós normais e ZoneServers.

Os ZoneServers são clientes que são eleitos com base nos recursos que têm disponíveis como, por exemplo, CPU, memória, largura de banda ou bateria. Cada ZoneServer gere um pequeno grupo de jogadores, recebendo as actualizações dos seus jogadores e propago-as para outros jogadores através dos outros ZoneServers existentes na rede. Se um destes ZoneServers deixar de responder, os jogadores associados a ele podem-se ligar a outros Servidores e continuar a jogar. A descoberta dos ZoneServers é feita com base no protocolo Service Location Protocol (SLP) em que os clientes fazem *multicast* de um URL do jogo e os Servidores respondem. A principal desvantagem desta arquitectura é funcionar apenas ao nível da rede não usando qualquer mecanismo de Gestão de Interesse.

2.7 Sumário

Nesta secção apresentámos as técnicas mais comuns usadas na redução da latência e largura de banda, desde o nível da camada de rede até ao nível da lógica da aplicação. Os sistemas académicos apresentados aplicam técnicas Gestão de Interesse de jogos multi-jogador mas não estão focados para as limitações que os telemóveis apresentam como a autonomia limitada da bateria, fraca capacidade de processamento e quantidade de memória. Os sistemas mostram ter pouca flexibilidade e alguns deles estão focados para um tipo de jogo específico, não permitindo a aplicação do sistema em jogos de outro tipo.

3

Arquitectura

Conteúdo

3.1 Vector Field Consistency (VFC) **24**
3.1.1 Anéis de Consistência 25
3.1.2 Graus de Consistência 25
3.1.3 Generalizações do VFC 26
3.1.4 Especificação do Modelo de Consistência 26
3.1.5 Vantagens do VFC 27
3.2 DeadReckoning **27**
3.2.1 Tipos de estimações 29
3.3 VFC-reckon **32**
3.4 Arquitectura **33**
3.4.1 Leitura e Escrita de Objectos 34
3.4.2 Serialização de mensagens 35
3.4.3 Propagação de actualizações 36
3.4.4 Gestor de Sessão 37
3.4.5 Aplicação do VFC-reckon 38
3.5 Sumário **40**

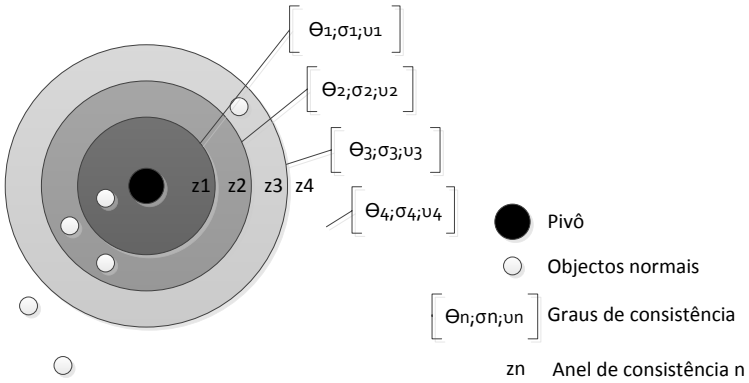

Figura 3.1: Exemplo do VFC com 3 anéis de consistência e com um pivô

A solução deste trabalho consiste no desenvolvimento do modelo de consistência VFC, em conjunto com técnicas de DeadReckoing. O sistema é um *middleware*, desenvolvido para a plataforma Android, que controla a consistência dos jogos multi-jogador. Neste sistema, o VFC tem um papel de filtro durante a execução, que selecciona as actualizações que devem ser ou não enviadas aos jogadores. As técnicas de DeadReckoning são aplicadas para corrigir a falta de mensagens de actualização, no lado dos clientes. Na secção 3.1 encontra-se a descrição do modelo de consistência VFC, na secção 3.2 está a descrição das técnicas de DeadReckoning , na secção 3.3 encontra-se descrito como é feita a junção entre o VFC e as técnicas de DeadReckoning e, por último, na secção 3.4 é apresentada a arquitectura Cliente-Servidor do nosso *middleware*.

3.1 Vector Field Consistency (VFC)

O VFC[12, 24–26] trata-se de um modelo de consistência de Replicação Optimista[5] e que aplica conceitos de Limitação de Divergência. Este algoritmo ajusta dinamicamente a consistência dos objectos replicados, com base no estado actual do jogo, gerindo o grau de consistência de cada objecto com base na sua distância a objectos especiais. Com o VFC consegue-se reduzir o número de mensagens que é preciso trocar entre jogadores em jogos multi-jogador. Esta redução de mensagens proporciona uma redução na largura de banda usada e uma menor carga processamento associada ao processamento de mensagens.

O VFC tem como principais conceitos: Anéis de Consistência e Graus de Consistência.

3.1.1 Anéis de Consistência

No VFC cada jogador tem uma vista local do mundo virtual correspondente ao jogo. Dentro de cada vista existem objectos especiais chamados pivôs. Os pivôs definem o grau de consistência de todos os objectos que os rodeiam. O grau de consistência de um dado objecto é dado pela função da distância do objecto ao pivô mais próximo. Isto significa que objectos próximos terão um grau de consistência mais alto, enquanto que, objectos mais distantes terão um grau de consistência mais baixo. Num jogo um objecto pivô pode ser, por exemplo, o avatar[1] que representa o jogador.

Os anéis de consistência são anéis formados em redor dos pivôs, que definem o grau de consistência dos objectos que se situam nesses anéis. Na Fig. 3.1 temos um exemplo de 3 anéis de consistência em redor de um objecto pivô. Nesta figura o círculo preto representa o objecto pivô e os círculos brancos representam os objectos existentes na vista do jogador. A intensidade de cor existente nos anéis representa o grau de consistência desse anel. Neste caso, o objecto situado no anel $z1$ terá um grau de consistência maior que os dois objectos situados no anel $z2$ e assim sucessivamente. O $z4$ representa a zona que está fora dos anéis de consistência. O grau de consistência de $z4$ pode ter um valor ou pode ser nulo. A nulidade do grau de consistência na zona $z4$ depende da lógica do jogo. Para este exemplo apenas se consideraram 3 anéis mas a flexibilidade do VFC permite-nos definir N anéis de consistência.

Apesar de na Fig. 3.1 os anéis serem de duas dimensões não existe nenhuma limitação de dimensões no VFC podendo usar D dimensões como, por exemplo, 3 dimensões onde neste caso os anéis passam a ser esferas.

3.1.2 Graus de Consistência

Cada anel de consistência tem um grau de consistência associado a ele. Um grau de consistência é um vector de 3 dimensões que especifica a divergência máxima permitida para objectos dentro desse anel. As 3 dimensões existentes neste vector são: Temporal(θ), Sequencial(σ) e Valor(υ).

A Dimensão Temporal θ especifica o intervalo máximo temporal de dessincronização de duas réplicas. Nesta dimensão especificamos o intervalo máximo de tempo (segundos) que uma réplica de um objecto pode ficar sem receber actualizações. Por exemplo se escolhermos um intervalo de $\theta = 2$ segundos significa que no máximo a réplica está desactualizada 2 segundos.

A Dimensão Sequencial σ especifica o número máximo de actualizações que uma réplica pode perder. Com esta dimensão garantimos que uma réplica de um objecto está

[1] O avatar é a entidade que representa o jogador dentro do mundo virtual do jogo como, por exemplo, um soldado, uma figura ou uma nave.

no máximo desactualizada em σ actualizações face ao objecto original. Considerando como exemplo que $\sigma = 2$ e o estado do objecto original e das suas réplicas estão sincronizados, então o objecto original pode aplicar até 2 actualizações e só à 3ª actualização é que propaga as actualizações para as suas réplicas.

A Dimensão de Valor υ especifica a diferença máxima de estado permitida entre a réplica e o objecto original. Esta dimensão necessita de uma função que calcule a percentagem de diferença entre 2 objectos o que a torna dependente da implementação do jogo. Se por exemplo usarmos um valor $\upsilon = 25\%$ significa que o estado entre o objecto original e a sua réplica tem uma diferença máxima de 25%.

Considerando agora como exemplo um grau de consistência representado pelo vector [θ ; σ ; υ] = [2 ; 3 ; 30] podemos concluir que os objectos existentes no anel com esse vector, estão desactualizados no máximo 2 segundos ou podem perder até 3 actualizações ou podem diferir de estado em relação ao objecto original em 30%. A partir do momento que o valor de qualquer uma das dimensões é atingido ou ultrapassado, dá-se inicio à actualização da réplica.

3.1.3 Generalizações do VFC

O VFC permite a aplicação de algumas generalizações: multi-pivô e multi-anéis. Multi-pivô significa que podem existir vários pivôs na mesma vista. Com a existência de vários pivôs um objecto pode ter vários graus de consistência diferentes, sendo cada grau atribuído por um pivô. Neste caso, o grau de consistência escolhido é o maior dos existentes.

A outra generalização, multi-anéis, permite-nos atribuir graus e anéis de consistência diferentes para objectos diferentes. Esta generalização permite que um objecto à mesma distância do pivô que outro objecto, possa ter um grau de consistência diferente do outro objecto. Por exemplo, num jogo existem objectos mais prioritários que outros. Se um jogador estiver próximo de outro então provavelmente a consistência entre eles terá que ser alta mas se considerarmos outro objecto à mesma distância como um pack de munições então a consistência será menor.

3.1.4 Especificação do Modelo de Consistência

O modelo de consistência do VFC é o conjunto de todos os objectos, pivôs, anéis de consistência e graus de consistência. Do ponto de vista do programador, para especificar todo o modelo de consistência associado a qualquer jogo multi-jogador, basta apenas especificar estes 4 conjuntos. A agregação destes 4 conjuntos corresponde ao phi do VFC.

3.1.5 Vantagens do VFC

Este modelo de consistência tem como principais vantagens a facilidade na especificação do sistema pelo programador, manter a jogabilidade do jogo aceitável e redução no número de mensagens trocadas entre os dispositivos.

A facilidade de especificação tem origem na fácil percepção por parte do programador. O modelo de consistência baseado em pivôs é intuitivo e fácil de perceber. A flexibilidade e simplicidade do VFC na definição dos graus de consistência tornam o algoritmo aplicável a vários tipos de jogos. Ao contrário de outros sistemas, o VFC é bastante flexível na definição de diferentes tipos de consistência para objectos diferentes.

A jogabilidade do jogador depende sobretudo dos parâmetros de consistência definidos em phi. O programador tem a responsabilidade de parametrizar correctamente o sistema consoante a lógica do jogo. Quando correctamente parametrizado a jogabilidade é muito pouco afectada com este algoritmo fazendo com que do ponto de vista do jogador e da lógica de jogo, todas as regras do jogo sejam cumpridas.

A redução no número de mensagens transmitidas na rede é a principal vantagem do uso do VFC, especialmente em dispositivos móveis. Tratando-se de um algoritmo de Gestão de Interesse apenas propaga actualizações para os jogadores interessados, reduzindo assim o número de mensagens necessárias para a execução do jogo. Esta selecção e diminuição de mensagens permite aumentar a escabilidade do jogo originando uma redução no uso da largura de banda e carga de CPU.

3.2 DeadReckoning

As técnicas de DeadReckoning[4] são técnicas muito usadas em jogos multi-jogador devido à sua simplicidade e ganhos obtidos. O DeadReckoning é normalmente usado como método de redução do impacto da latência através da antecipação de futuras posições com base na posição actual e o histórico de antigas posições. Neste trabalho o objectivo do uso do DeadReckoning não é reduzir o impacto da latência, mas sim mascarar o descarte propositado de mensagens de actualização dos objectos.

Normalmente os jogos têm uma *frame-rate* que define o número de vezes por segundo que os entidades do jogo são actualizadas e desenhadas no ecrã. Para o jogador não se aperceber destas actualizações estas têm de ser feitas a uma taxa superior a 25fps (*frames*-por-segundo); abaixo deste valor será notável uma intermitência no movimento dos objectos.

Para um jogo multi-jogador obter uma *frame-rate* de, no mínimo, 25fps e não haver intermitências no movimento dos objectos remotos, é necessário que as actualizações sejam enviadas para os outros jogadores num intervalo menor ou igual a 40ms. Esta

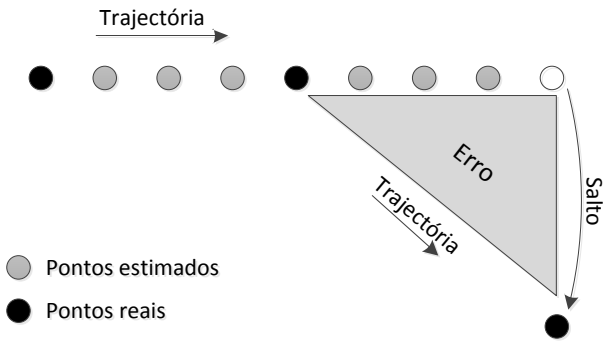

Figura 3.2: Exemplo de um cenário de erro na estimação de posições

taxa de envio de actualizações é uma taxa muito alta, especialmente para dispositivos móveis onde normalmente a capacidade de processamento, a capacidade da bateria e largura de banda são limitados.

Uma solução possível para reduzir a carga de processamento e o número de mensagens enviadas é o aumento do intervalo a que as mensagens são enviadas como, por exemplo, de 40ms para 160ms. Ao aumentarmos o intervalo de 40ms para 160ms começamos a notar uma descontinuidade no movimento dos objectos remotos, o equivalente a moverem-se a uma taxa de 6fps.

Através das técnicas de DeadReckoning é possível aumentar o intervalo de envio das actualizações, sem que isso afecte a jogabilidade do jogo, de uma maneira notável, do ponto vista do jogador. Com o aumento do intervalo são descartadas n posições intermédias em que $n = (TaxaActual/40ms) - 1$. Para uma taxa de 160ms entre cada intervalo de envio, existem 3 posições que deixam de ser enviadas face a uma taxa de 40ms. Usando o DeadReckoning podemos estimar estas 3 posições e obter um movimento nos objectos semelhante ao de uma taxa de 40ms.

Apesar de o DeadReckoning poder estimar qualquer número de pontos é preciso ter em conta que se tratam de estimativas. Em qualquer valor estimado existe sempre uma diferença entre o valor estimado e o valor real, o que neste caso, se reflecte no movimento dos objectos. Quando aumentamos o intervalo de envio de actualizações, estamos também a aumentar a quantidade de erro na estimação de posições. À medida que os intervalos de tempo entre cada posição real vão aumentando, a quantidade de erro também aumenta de uma maneira proporcional.

Na Fig. 3.2 podemos ver o resultado do aumento do intervalo de tempo entre cada posição real. O erro na estimação de posições é notável quando ocorre uma mudança de trajectória pois o DeadReckoning está a assumir um movimento linear. A detecção

da nova trajectória apenas será feita quando chegar uma nova posição real. Durante este intervalo de espera as posições estimadas são calculadas com base na trajectória anterior fazendo com que o objecto tenha um movimento incorrecto. Do ponto de vista do jogador, quanto maior for o intervalo entre cada posição real, maior será o salto em caso de erro.

Para qualquer posição nova gerada pelos clientes o funcionamento é sempre o mesmo. A posição é calculada no lado dos clientes, pela parte do jogo responsável por actualizar as componentes físicas dos objectos e é, posteriormente, enviada ao servidor. O servidor processa a mensagem recebida e envia a actualização aos clientes remotos. Ao receber a mensagem do servidor, o cliente remoto entrega a mensagem ao módulo de DeadReckoning para serem estimadas novas posições

Um aspecto importante do funcionamento do módulo de DeadReckoning é que a sua activação depende da mensagem que vem do servidor. O servidor informa o cliente através de três parâmetros: 1) Número de pontos, 2) Intervalo Temporal e 3) Tipo de Trajectória. O parâmetro número de pontos indica o número de pontos que devem ser estimados entre o intervalo temporal. O intervalo temporal indica o intervalo entre as posições reais enviadas pelo servidor, e serve para o motor de estimação calcular os intervalos a que as posições estimadas devem ser entregues à aplicação. O Tipo de Trajectória indica o tipo de trajectória (linear, circular, etc...) que o objecto segue actualmente.

Existe um problema associado ao cálculo de posições, feito pela parte do jogo responsável por actualizar as componentes físicas dos objectos, e que pode influenciar a estimação de posições. Uma maneira de calcular novas posições para os objectos nos jogos é através do intervalo de tempo entre cada *frame* onde a distância percorrida pelo objecto é calculada com base no intervalo de tempo entre a *frame* actual e a *frame* anterior. Estes cálculos são normalmente aproximações já que o intervalo de tempo entre cada *frame* varia ligeiramente. Uma pequena variância nestes intervalos faz com que os objectos não percorram sempre a mesma distância entre cada *frame*. Estas pequenas diferenças nas distâncias podem influenciar a qualidade das estimações gerando um aumento no erro na estimação de novas posições.

3.2.1 Tipos de estimações

Para este trabalho foram construídos 2 motores de estimação de posições, sendo um para trajectórias lineares e outro para trajectórias circulares.

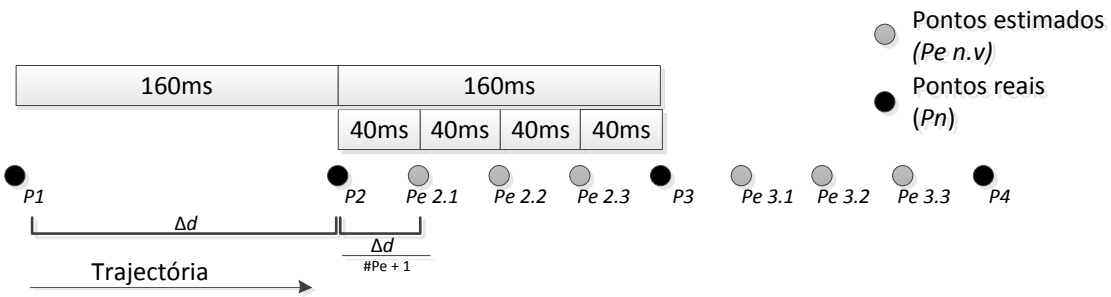

Figura 3.3: Exemplo de estimações de posições para uma trajectória linear

3.2.1.A Trajectórias lineares

A trajectória linear é normalmente a trajectória que mais se encontra nos jogos, especialmente quando existem entidades artificiais como, por exemplo, *bots*[2] onde a tendência em manter a mesma trajectória é maior. Em jogos multi-jogador, durante grande parte do tempo de jogo, o jogador move-se segundo uma trajectória linear alterando algumas vezes a direcção.

No nosso sistema, o cálculo segundo uma trajectória linear é muito simples. As distâncias a percorrer pelo objecto, bem como a sua direcção, são calculadas através do histórico de posições e da posição actual usando apenas as coordenadas X e Y das posições do objecto. Basicamente, é feita a diferença entre as componentes X e Y e dividimos consoante o número de posições intermédias que são necessárias calcular.

Na Fig. 3.3 temos o exemplo da estimação de 3 posições, segundo uma trajectória linear, em que o intervalo de envio de actualizações é de 160ms. Para o início do funcionamento do motor de estimação de posições são necessárias no mínimo duas posições reais, neste caso, $P1$ e $P2$. Os pontos estimados $Pe(n.v)$ na figura são calculados a partir dos pontos reais Pn e $P(n-1)$. Para calcular a posição de um Pe é necessário saber o número de pontos Pe entre os dois pontos Pn. Com o número de pontos Pe entre cada ponto Pn ($\#Pe$) podemos calcular o deslocamento entre cada Pe a partir da fórmula $\frac{\Delta d}{(\#Pe+1)}$ onde Δd é a distância percorrida entre os dois pontos reais $P1$ e $P2$. Por fim, para obter os pontos estimados $Pe(n.v)$ basta somar o deslocamento $\frac{\Delta d}{(\#Pe+1)}$ a cada posição.

3.2.1.B Trajectórias circulares

O segundo tipo de trajectórias mais utilizadas em jogos são as trajectórias circulares. Tal como no cálculo de uma trajectória linear, só é possível começar a estimar novas

[2]Entidades do jogo controladas por Inteligência Artificial

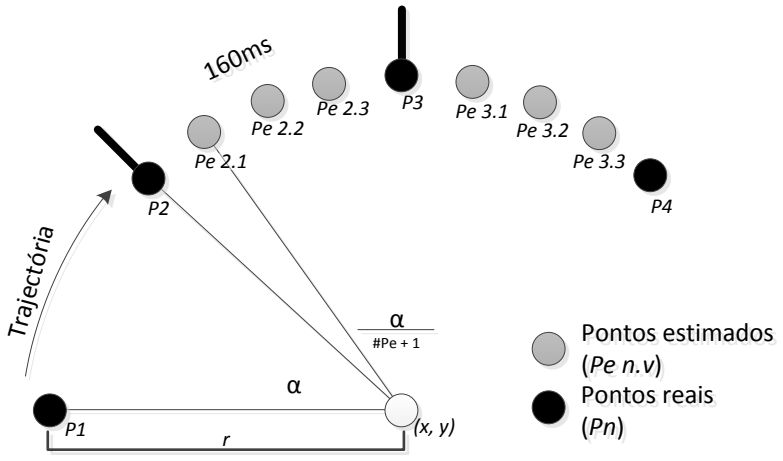

Figura 3.4: Exemplo de estimações de posições para uma trajectória circular

posições quando temos, no mínimo 2 posições reais. A estimação de novas posições com base nesta trajectória é baseada no ângulo, no ponto central da circunferência e no raio.

Na Fig. 3.4 temos o exemplo da estimação de 3 posições para um intervalo de 160ms. Para calcular a posição dos pontos estimados ($Pe(n.v)$) é necessário, em primeiro lugar, calcular o novo ângulo a partir do ângulo α entre $P1$ e $P2$. O novo ângulo é o incremento que cada Pe tem que ter em relação à posição anterior, sendo calculado a partir da fórmula ($\frac{\alpha}{\#Pe+1}$). Por fim, para obter os pontos estimados $Pe(n.v)$ basta somar o novo ângulo ao α anterior e calcular a posição com base no raio r, o novo ângulo α e o centro da circunferência, através de cálculos trigonométricos.

3.2.1.C Outras trajectórias

Uma das preocupações do nosso sistema, relativamente ao tipo de trajectórias, foi a sua extensibilidade. No nosso sistema implementámos os motores de estimação de posições para trajectórias lineares e trajectórias circulares mas permitimos que o programador possa implementar um novo motor de estimação para outro tipo de trajectória. Estes motores são simples funções que recebem a posição actual e o histórico de posições reais e que devolvem um conjunto de posições estimadas.

O tipo de trajectória que um objecto segue é especificado no objecto do jogo através da invocação de um método chamado *setPredictionEngine()*. Saber o tipo trajectória que um objecto tem especificado facilita o papel do módulo de DeadReckoning que, só tem que consultar o tipo de trajectória, e passar de imediato o objecto ao motor de estimação correspondente. Uma das limitações deste sistema é que não altera de forma automática

a trajectória de um objecto, tendo esta que ser sempre alterada de forma manual através do método *setPredictionEngine()*. Esta limitação só podia ser contornada se o módulo de DeadReckoning detectasse o tipo de trajectória com base nas posições anteriores e na actual o que ia implicar uma maior carga de processamento.

3.3 VFC-reckon

O VFC-reckon[3] permite juntar as vantagens do VFC e das técnicas de DeadReckoning, o que contribui para o objectivo principal do sistema: O aumento da escalabilidade a partir da redução do número de mensagens trocadas entre os dispositivos. Apesar do VFC e das técnicas de DeadReckoning terem objectivos comuns, o papel que desempenham no sistema difere bastante. O VFC efectua uma selecção das mensagens que devem ou não ser enviadas com base na distância dos objectos aos pivôs, enquanto que, as técnicas de DeadReckoning mascaram a ausência de mensagens criada através devido ao aumento dos intervalos de troca de mensagens entre o servidor e os clientes.

A especificação dos graus de consistência do VFC-reckon corresponde a uma extensão no número de parâmetros usados nos graus de consistência do VFC, estendendo de 3 para 4 o número de parâmetros. Para além dos parâmetros Temporal(θ), Sequencial(σ) e Valor(υ) temos agora um parâmetro novo (*estimationPoints*) que indica o número de pontos que devem ser estimados, pelo cliente, quando uma entidade se encontra dentro daquele anel zn.

Adicionar um parâmetro extra aos graus de consistência é uma solução que permite ao programador ter um modelo conceptual mais fácil, do ponto de vista da lógica do jogo, pois toda a parametrização relativa à consistência do objecto pode ser encontrada na especificação do VFC, ou seja, no phi.

A activação e parametrização do DeadReckoning estão a cargo do servidor que executa o algoritmo VFC. Cada vez que o VFC selecciona uma mensagem para enviar ao cliente, baseando-se nos anéis de consistência, verifica se o anel onde a entidade se encontra tem o parâmetro *estimationPoints* maior que zero. Quando o valor de *estimationPoints* é positivo, o servidor especifica na mensagem o número de pontos que é necessário o motor de estimação do cliente estimar, bem como o intervalo temporal. Se o valor de *estimationPoints* for zero significa que não é necessário estimar posições o que pode ser muito útil no lado do cliente para poupar algum tempo de CPU.

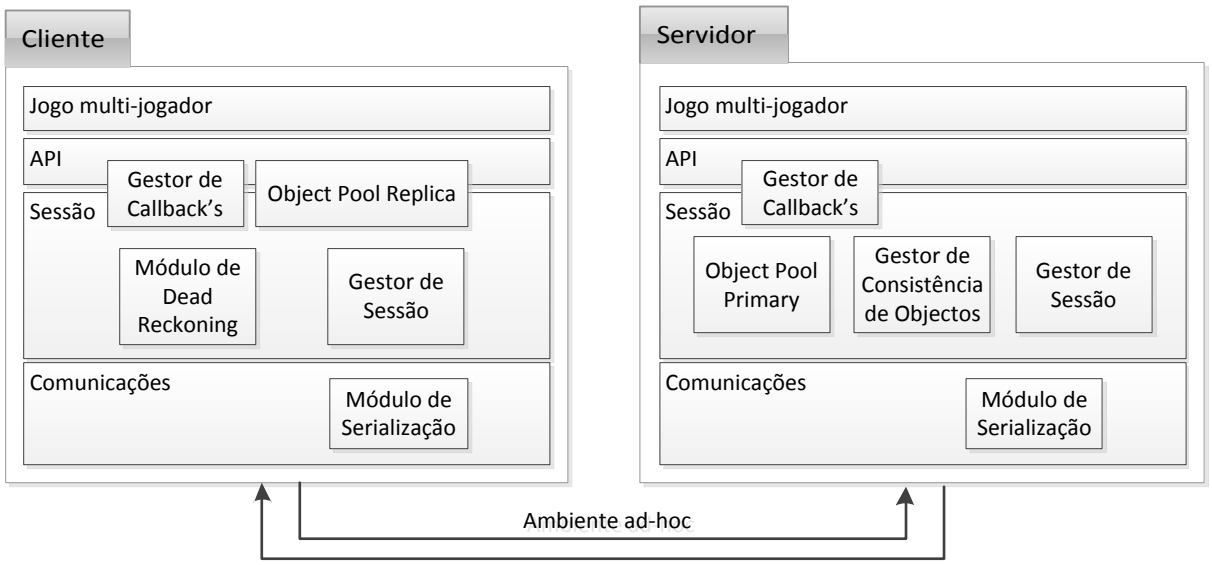

Figura 3.5: Arquitectura do sistema

3.4 Arquitectura

A arquitectura do sistema é baseada numa arquitectura Cliente-Servidor, apresentada na Fig. 3.5, e serve como um *middleware* para o jogo multi-jogador. O facto de apenas um nó na rede, neste caso o servidor, aplicar o VFC-reckon torna mais simples a sua implementação e execução pois toda a informação necessária para a aplicação encontra-se no nó do servidor. O protocolo Cliente-Servidor é implementado na componente de Gestor de Sessão. Do lado do servidor, esta componente tem as responsabilidades de processar os pedidos de actualização dos clientes e propagar as actualizações aos clientes seleccionados pelo VFC-reckon. A comunicação entre clientes e servidor está implementada na Camada de Comunicações seguindo uma topologia do tipo estrela.

Os objectos do sistema que armazenam a informação partilhada do estado do jogo têm o nome de objectos primários e réplicas. Os objectos primários são armazenados no servidor, enquanto que, as réplicas são locais a cada cliente. O servidor executa rondas periódicas onde filtra e envia as actualizações relevantes aos clientes de modo a que estes possam actualizar as suas réplicas. As escritas e leituras dos clientes são feitas nas réplicas locais sendo depois propagadas, no caso de serem escritas, em *background* para o servidor.

A aplicação do VFC-reckon depende de duas componentes principais: Gestor de

[3]Nome vem da junção dos nomes *VFC* e Dead*Reckon*ing

Consistência de Objectos e Módulo de DeadReckoning.

As rondas do servidor são efectuadas pelo Gestor de Consistência de Objectos. O VFC-reckon é aplicado neste módulo, filtrando os objectos que devem ser enviados para os clientes com base no modelo de consistência phi definido por eles na fase de configuração do jogo.

O Módulo de DeadReckoning, existente apenas nos clientes, tem o papel de estimar as novas posições com base na posição actual e histórico de posições armazenadas que foram recebidas do servidor.

O módulo de serialização é dos módulos com maior impacto no desempenho do sistema. A elevada taxa de transmissão, na ordem dos 40ms de intervalo entre cada mensagem, pode facilmente fazer com que a serialização de mensagens se torne o *bottleneck* do sistema. A serialização por omissão do Java demonstra não ser uma boa solução devido à elevada taxa de transmissão.

3.4.1 Leitura e Escrita de Objectos

O estado do jogo é representado em objectos de 2 tipos: Réplicas e Objectos Primários. As réplicas são armazenadas localmente no cliente, pelo Gestor de Réplicas de Objectos, sendo as réplicas cópias dos objectos primários. Os objectos primários são armazenados no servidor pelo Gestor de Objectos Primários.

A aplicação do lado do cliente efectua todas as leituras e escritas que necessita a partir das réplicas locais. O facto de o cliente ler apenas os dados armazenados localmente, sem ter que consultar o servidor, diminui significativamente o tempo de acesso à informação. Como o cliente pode ler informação inconsistente este comportamento segue o modelo de Replicação Optimista[5].

A remoção dos objectos é gerida pelo servidor para evitar problemas de consistência como, por exemplo, existirem objectos num cliente e que já não existem noutro cliente. Para resolver este problema de consistência, quando um cliente deseja remover um objecto, envia ao servidor uma mensagem a solicitar a remoção do objecto e o servidor indica o sucesso ou erro na remoção. O sucesso da remoção ocorre quando se é o primeiro cliente a pedir a remoção do objecto. Se um cliente tenta remover um objecto, que já foi removido por outro cliente, recebe do servidor uma mensagem de erro. Este erro na remoção é comum quando existem, por exemplo, 2 clientes a tentarem remover o mesmo objecto, ao mesmo tempo. Neste caso um cliente tem sucesso e o outro recebe uma mensagem de erro. Quando a remoção ocorre com sucesso o servidor informa o cliente que a remoção foi bem sucedida e informa todos os outros clientes que o objecto foi removido do jogo.

3.4.2 Serialização de mensagens

O Módulo de Serialização é um dos módulos cruciais para o desempenho do sistema. Aplicações como jogos multi-jogador exigem uma grande taxa de transferência de informação podendo a serialização de mensagens tornar-se facilmente o *bottleneck* do sistema.

A serialização Java é fácil e simples de usar mas apresenta um péssimo desempenho quando estamos a usar intervalos de envio de 40ms. Outro dos aspectos negativos da serialização Java é a dimensão das mensagens serializadas. Como são guardados campos como, por exemplo, o nome completo da classe, o tamanho das mensagens é relativamente grande até mesmo para transferir apenas alguns bytes úteis de informação como, por exemplo, as coordenadas X e Y de um objecto no mundo virtual, o que corresponde a 8bytes (2 inteiros).

Para a solução do problema de serialização foi desenvolvido um Módulo de Serialização simples que implementa a serialização dos tipos primitivos mais comuns como, por exemplo, interios, doubles e Strings.

Cada mensagem trocada entre os dispositivos estende o tipo Message que contém 2 métodos especiais: *saveState()* e *restoreState()*.

O método *saveState()* recebe uma mensagem e transforma-a num *array* de bytes, enviando-os para o módulo de comunicação para serem transmitidos.

O método *restoreState()* funciona de maneira inversa do *saveState()*, recebendo um *array* de bytes proveniente do módulo de comunicação, transformando-os numa mensagem para entregar à Camada de Sessão.

Cada entidade do jogo que estende a classe especial DataUnit[4] ou uma das subclasses, pode opcionalmente estender os métodos *restoreState()* e *saveState()* para usar o módulo de serialização e tirar partido da rapidez e dimensão das mensagens pequenas.

Os métodos *restoreState()* e *saveState()* são opcionalmente implementados pelo programador. Se o utilizador não implementar os métodos *restoreState()* e *saveState()* será usada a serialização Java por omissão.

O formato das mensagens é não estruturada, ou seja, não são usadas etiquetas para identificar o tipo de objecto, mas sim a sua posição no *array* de bytes. O tipo de mensagem é identificado através de um inteiro no início do *array*. A escolha de um inteiro em vez do nome da classe, como na serialização Java, permite reduzir em muito a dimensão das mensagens bem como o tempo de processamento da mensagem. Como a responsabilidade da serialização é de cada mensagem esta tem que implementar os

[4] Um DataUnit representa um objecto do jogo partilhado entre todos os clientes

métodos de escrita e leitura dos valores seguindo sempre a mesma ordem. Por exemplo, se a mensagem escrever um inteiro e dois doubles, tem que ler um inteiro e dois doubles exactamente pela mesma ordem.

3.4.3 Propagação de actualizações

Para perceber como é feita a propagação de actualizações no nosso sistema é necessário ter em conta que existem duas fases de propagação de actualizações:
Servidor –> Cliente e Cliente –> Servidor.

Propagação de actualizações - Servidor –> Cliente

Para a propagação das actualizações do servidor para os clientes, o Gestor de Consistência de Objectos aplica um conceito de ronda periódica. Basicamente, o Gestor de Consistência de Objectos executa periodicamente uma função, com o objectivo de informar os clientes que for necessário sobre as novas actualizações.

Durante cada ronda, o servidor calcula as actualizações que são necessárias enviar para cada cliente, com base na especificação dos parâmetros do VFC, ou seja, o phi. Após o cálculo e selecção, são enviadas, para os clientes, as mensagens de ronda com todas actualizações destinadas àquele cliente em concreto. Quando o cliente recebe as mensagens de ronda aplica as actualizações às suas réplicas usando o Gestor de Réplicas de Objectos.

A recepção periódica de actualizações por parte do cliente permite sincronizar a aplicação do cliente com o sistema, através de *callback's* implementados pelo Gestor de Callback's. O Gestor de Callback's permite notificar a aplicação que o estado do jogo se alterou, permitindo, por exemplo, à aplicação actualizar a pontuação do jogador no ecrã.

Propagação de actualizações - Cliente –> Servidor

Na lógica do jogo existe uma componente responsável por actualizar a componente física dos objectos como, por exemplo, a velocidade, posição, etc... Enviar as actualizações para o servidor cada vez que ocorre uma actualização seria uma má escolha devido à quantidade de actualizações que existem em cada ciclo de jogo. Isto ia fazer com que o cliente enviasse N mensagens sendo N igual ao número de actualizações que ocorreram naquele ciclo do jogo. Uma maneira mais eficiente passa por propagar as actualizações dos clientes para o servidor de forma semelhante à ronda do servidor. As actualizações são agrupadas numa só mensagem e periodicamente enviadas em *background* para não interferir com o ciclo de jogo.

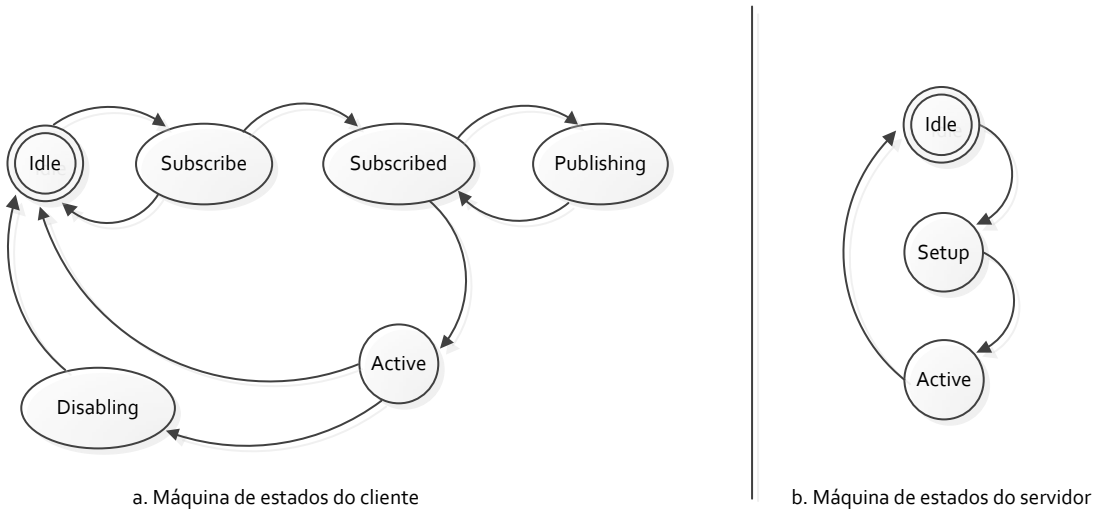

Figura 3.6: Máquinas de estados de Gestor de Sessão do Cliente e Servidor

O intervalo de ronda de envio de actualizações do cliente para o servidor deve ser igual ao intervalo de ronda do servidor. Se o intervalo de envio de actualizações do cliente for inferior ao do servidor isto significa que estamos a enviar actualizações que estão a ser descartadas pelo servidor pois só a última actualização recebida pelo servidor é que é usada.

Relativamente ao sincronismo do sistema, as actualizações dos clientes não são enviadas exactamente ao mesmo tempo. Para isto era necessário todos os clientes estarem sincronizados através de, por exemplo, um relógio global. Os clientes enviarem as actualizações todas ao mesmo tempo seria irrelevante pois, as actualizações recebidas no servidor só são utilizadas no momento da ronda, ou seja, receber todas as actualizações, de uma só vez, antes do momento da ronda ou ir recebendo as actualizações, temporalmente espaçadas, até ao momento da ronda acaba por ser igual.

3.4.4 Gestor de Sessão

O Gestor de Sessão do cliente e do servidor seguem o modelo de uma máquina de estados apresentado na Fig. 3.6. Durante o arranque, o servidor encontra-se no estado Idle à espera de uma ligação do cliente.

Quando um cliente pretende ligar-se ao servidor, transita para o estado Subscribe e tenta ligar-se ao servidor. Se o servidor aceitar a ligação o cliente transita para o estado Subscribed e o servidor para o estado Setup. No estado Subscribed o cliente pode enviar os parâmetros de configuração do VFC-reckon, ou seja, o phi. Para publicar novos

objectos transita para o estado Publishing e envia os objectos que pretende registar ao servidor. Os objectos publicados são os objectos que o cliente pretende partilhar com os outros clientes como, por exemplo, o seu avatar ou a sua pontuação. Após a resposta positiva do servidor, o cliente transita de volta para o estado Subscribed.

Durante o processo de publicação de objectos é atribuído um identificador global ao objecto. O identificador é um identificador numérico atribuído sequencialmente à medida que o servidor processa os pedidos de publicação de objectos. O facto de ser um identificador do tipo inteiro não é problemático pois não se espera que o número de objectos registados pelos clientes exceda o valor de máximo de um inteiro. O identificador facilita o acesso ao objecto se este estiver armazenado numa estrutura do tipo *array* onde o acesso é extremamente rápido.

Quando os clientes terminam o processo de subscrição e publicação de objectos qualquer um dos clientes pode tomar a iniciativa de transitar para o estado Active enviando uma mensagem de activação ao servidor. Se vários clientes enviarem uma mensagem de activação apenas a 1ª será processada. Assim que o servidor recebe a 1ª mensagem de activação envia de imediato a todos os clientes uma mensagem a informar para transitarem para o estado Active.

De um modo abstracto a máquina de estados pode decompor-se em duas fases: Fase de Configuração e Fase Activa.

A fase de configuração, onde o cliente se regista e envia a informação necessária do jogo, e a fase activa que representa o decorrer do jogo. A fase activa, corresponde aos estados Active do cliente e do servidor; é a fase principal onde se dá a maior troca de mensagens entre o cliente e o servidor.

Durante a fase activa o servidor utiliza as rondas periódicas, para enviar as actualizações necessárias, e processa as mensagens de actualização que vão chegando dos clientes. Para além de publicar os objectos durante a fase de configuração o cliente tem a opção de, durante a fase activa do jogo, publicar novos objectos ou remover objectos partilhados como, por exemplo, tiros, bombas, asteróides, etc...

A fase Activa termina quando, por exemplo, o jogo chegou ao fim. Neste caso o cliente envia uma mensagem de desactivação para o servidor e aguarda a resposta no estado Disabling. Recebendo a mensagem de desactivação, o servidor regressa ao estado Idle e informa os outros clientes que a fase activa chegou ao fim transitando os outros clientes do estado Active para o estado Idle.

3.4.5 Aplicação do VFC-reckon

Para a aplicação do VFC-reckon existem duas componentes principais: O Gestor de Consistência de Objectos existente na Camada de Sessão do Servidor e o Módulo de

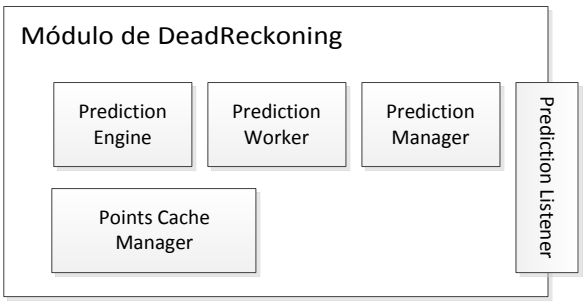

Figura 3.7: Arquitectura interna do módulo de DeadReckoning

DeadReckoning existente na Camada de Sessão do Cliente.

3.4.5.A Gestor de Consistência de Objectos

O Gestor de Consistência de Objectos é o único responsável pela aplicação do VFC. Esta componente é usada pelo Gestor de Sessão durante as rondas do servidor, pedidos de actualização por parte dos clientes e pedidos de registo de novos objectos dos clientes.

A especificação dos parâmetros de DeadReckoning é feita pelo Gestor de Consistência de Objectos. Durante cada ronda, para cada actualização, o servidor compara o objecto com o parâmetro do grau de consistência *estimationPoints*. O servidor especifica na mensagem o número de pontos que são necessário estimar, bem como o intervalo de tempo entre cada mensagem.

3.4.5.B Módulo de DeadReckoning

O Módulo de DeadReckoning situa-se na Camada de Sessão do Cliente. Ao receber uma mensagem o Módulo de DeadReckoning analisa a mensagem verificando se é necessário estimar pontos ou não.

Na Fig. 3.7, encontra-se a arquitectura do Módulo de DeadReckoning. Este Módulo tem 5 componentes fundamentais para o seu funcionamento: Prediction Listener, Points Cache Manager, Prediction Manager, Prediction Worker e Prediction Engine.

O Points Cache Manager é responsável pelo armazenamento e gestão do histórico de pontos de cada objecto. Existe um limite para o histórico de posições que, quando atingido, acciona a remoção de posições mais antigas.

A componente de Prediction Engine é a componente responsável por calcular a próxima posição com base na posição actual recebida e no histórico de posições, existente na componente Points Cache Manager.

Os Prediction Worker são responsáveis por apenas um DataUnit, ou seja, um objecto do jogo. Cada vez que recebe uma nova mensagem, relativo ao seu DataUnit, invoca a componente de PredictionEngine calculando o novo conjunto de posições futuras. O número de posições estimadas é o número indicado pelo servidor.

O Prediction Manager é responsável por fazer a gestão de todos os Prediction Workers. As mensagens que vão sendo recebidas, vão sendo entregues aos Prediction Workers correspondentes usando o DataUnit como mecanismo de indexação. Quando são criados novos objectos durante o jogo, são também criados e guardados novos PredictionWorkers. Os novos objectos são detectados através da existência de novos DataUnits.

Por último, a componente de Prediction Listener é a componente responsável por fazer a ponte entre o Gestor de Sessão e o Módulo de DeadReckoning. À medida que as mensagens vão sendo recebidas o Prediction Listener calcula se é necessário o uso do módulo de DeadReckoning através do tipo de DataUnit e do número de pontos que são necessários estimar. Quando o número de pontos é igual a 0, a mensagem é ignorada não sendo necessário usar o módulo de DeadReckoning. Tratando-se de mensagens onde é necessário estimar novas posições estas são entregues ao Prediction Manager.

As posições estimadas pelos motores de estimação são guardadas numa *cache* situada ao nível da API do Sistema. Na *cache* os pontos são armazenados com um contador numérico especial. Este contador numérico indica em que ciclo do jogo o ponto deve ser entregue ao jogo. Quando o contador tem o valor de 0 isto significa que o ponto é para ser entregue no ciclo actual. Quando o contador tem um valor n significa que o ponto deve ser entregue ao jogo dentro de n ciclos. Com esta *cache* é possível sincronizar a entrega de posições ao jogo com o ciclo do mesmo. Esta sincronização é útil para evitar que os pontos sejam entregues em qualquer altura do jogo o que podia originar movimentos repentinos nos objectos.

No ciclo do jogo existe uma fase onde as componentes físicas dos objectos são actualizadas. Durante esta fase o jogo solicita à *cache* os pontos que são para ser entregues naquele ciclo, ou seja, os pontos que têm o contador 0. No fim, os contadores dos restantes pontos são decrementados em uma unidade sendo mais tarde entregues.

3.5 Sumário

Nesta secção apresentámos o modelo de consistência VFC-reckon. O VFC-reckon corresponde à extensão do VFC com técnicas de DeadReckoning. A junção do VFC com as técnicas de DeadReckoning é feita de uma maneira simples utilizando os graus de consistência do VFC. O modelo de consistência VFC é usado como mecanismo de

selecção de actualizações, enquanto que, as técnicas de DeadReckoning são usadas para mascarar a ausência de mensagens de actualização, ausência esta, criada devido ao aumento do intervalo de tempo entre cada mensagem de actualização. Os motores de estimação suportam trajectórias lineares e circulares e são extensíveis a outras que o programador queira implementar. A arquitectura do sistema segue o modelo Cliente-Servidor, usando uma topologia do tipo estrela para a comunicação.

4

Implementação

Conteúdo

4.1 API do Sistema **44**
4.2 Módulo de Serialização **45**
4.3 Camada de Comunicações **46**
4.4 Representação dos objectos partilhados **47**
4.5 Especificação do Módulo de DeadReckoning **47**
4.6 Gestão de Consistência de Objectos **48**
4.7 Especificação do Modelo de Consistência VFC-reckon **49**
4.8 Plataforma Android **50**
4.9 Jogo Multi-jogador Asteroids **51**
4.9.1 Gestão do estado do jogo 51
4.9.2 Objectos do Jogo 53
4.9.3 Gestor de Objectos do Jogo 54
4.9.4 Servidor do Jogo 54
4.9.5 Componente gráfica do jogo 55
4.9.6 Controles do jogo 57
4.10 Sumário **57**

O sistema VFC-reckon, e a aplicação demonstrativa, foram desenvolvidos na linguagem Java para a plataforma Android 2.2 Froyo e são compatíveis com versões superiores. Nesta secção são apresentadas as principais componentes da implementação do sistema especificando a API, descrição detalhada dos principais componentes, e as estruturas mais relevantes para o funcionamento do sistema. Através da informação descrita nesta secção o programador pode aplicar o sistema a um jogo multi-jogador previamente desenvolvido.

4.1 API do Sistema

A API do sistema indica o conjunto de funções que a aplicação usa para comunicar com o sistema, mais especificamente, com a camada de Sessão. A API da aplicação especifica os métodos que a aplicação tem que implementar para lidar com os eventos provenientes do sistema como, por exemplo, registar um novo DataUnit de outro cliente. Para a aplicação usar o sistema, tem que obedecer à interface definida por IVFCreckonApp. Esta interface possui, maioritariamente, métodos de notificação do sistema para a aplicação como, por exemplo, um novo objecto registado, início do Jogo, etc.

Para auxiliar a aplicação a realizar pedidos e lidar com os eventos provenientes do sistema foram criadas as classes VFCreckonClient, VFCreckonServer, VFCreckonClientListener e VFCreckonServerListener.

As classes terminadas em Listener (VFCreckonClientListener e VFCreckonServerListener) servem para indicar que estas classes são responsáveis por lidar com as notificações do sistema. As outras classes (VFCreckonClient e VFCreckonServer) servem para a aplicação enviar pedidos ao sistema.

A classe VFCreckonClient é responsável por lidar com os pedidos e ordens provenientes da aplicação. Para se efectuar a actualização de objectos, ou o registo de novos objectos, a aplicação deve invocar os métodos necessários nesta classe, que por sua vez comunica com a Camada de Sessão. No nosso sistema as actualizações de todos os objectos do cliente são propagadas periodicamente através de um temporizador que envia todas as actualizações para o servidor cada vez que expira.

A classe VFCreckonClientListener lida com as notificações da Camada de Sessão. Além de entregar as notificações necessárias ao jogo, esta classe tem também o objectivo de armazenar o estado dos objectos partilhados, ou seja, todas as DataUnits são armazenadas numa *hashTable* existente nesta classe.

Classes do servidor como VFCreckonServer e VFCreckonServerListener são mais simples que as dos clientes. O VFCreckonServer apenas possui um método para fazer a configuração do servidor, consoante o tipo de comunicações que esteja a utilizar. As

notificações do sistema para a aplicação são notificações simples que indicam quando, por exemplo, se estabeleceu uma ligação com um cliente.

A descrição detalhada da API do Sistema pode ser encontrada na Secção A.1 dos anexos. O diagrama de classes da Camada de API encontra-se nas figuras Fig. A.2 e Fig. A.1 da Secção A.2 dos anexos.

4.2 Módulo de Serialização

O Módulo de Serialização é o módulo responsável por processar as mensagens e convertê-las em *arrays* de bytes para que possam ser enviadas através da Camada de Comunicações usando um mecanismo simples de envio de bytes. A reconstrução das mensagens, pelo Módulo de Serializaçao, é feita usando um *array* de bytes entregue pela Camada de Comunicações.

A configuração do Módulo de Serialização encontra-se na classe SerializationConfigurations. Nesta classe são especificados identificadores numéricos que identificam a classe de cada tipo de DataUnit definido pelo programador como, por exemplo, ShipEntity, AsteroidEntity, ScoreEntity, etc... Alem disto o programador deve implementar o método *getClassId()* nos DataUnits de modo a este retornar o valor especificado em SerializationConfigurations.

A identificação dos DataUnits através de um inteiro, em vez de uma *string* com o nome da classe, permite reduzir o número de bytes transmitidos bem como o tempo necessário para reconstruir uma mensagem. Por exemplo, se o nome da classe for *com.VFCreckon.session.PositionableDU*, ao identificarmos a classe com um inteiro reduzimos parte do tamanho da mensagem de 36bytes para 4bytes.

As classes ByteWriter e ByteReader são as classes responsáveis por processar as mensagens e convertê-las em *arrays* de bytes e vice-versa. Estas classes contêm métodos necessários para lidar com os principais tipos de dados primitivos como, por exemplo, inteiros, *doubles* e *strings*.

A operação de serialização da mensagem para um *array* de bytes é feita fundamentalmente através de operações do tipo *shift left* e *shift right*. Um exemplo desta operação é a conversão de um inteiro para um *array* de bytes. Neste caso é feito um *shift* de modo a ficarmos apenas com os bits mais significativos armazenando os bits num byte. Esta operação repete-se, alterando o *offset* do *shift* até construir um *array* de bytes. O facto das operações de *shift* serem leves, do ponto de vista do processador, ajudam em muito no desempenho geral do módulo.

Para o Módulo de Serialização ser usado, mais especificamente, as classes ByteWriter e ByteReader, as mensagens e todas as estruturas existentes como, por exemplo,

DataUnits, devem implementar os métodos *saveState()* e *restoreState()*. Estes métodos recebem como argumentos os objectos ByteReader e ByteWriter que permitem ao objecto gravar ou reconstruir o seu estado. O uso do módulo de Serialização é opcional. Se os métodos *saveState()* e *restoreState()* não forem definidos nos DataUnits correspondentes será usado a serialização por omissão do Java.

Em resumo, para o uso do Módulo de Serialização é necessário 1) Definir as constantes dos DataUnits no ficheiro de configuração 2) Implementar os métodos *saveState() restoreState()* nos DataUnits e 3) Definir o retorno do método *getClassId()* como identificador do tipo da classe.

4.3 Camada de Comunicações

O sistema permite a comunicação através dos protocolos WiFi[2] com sockets TCP/IP e Bluetooth[1] com sockets Bluetooth. Apesar de funcionar com ambos os protocolos, o Bluetooth acaba por ser, normalmente, a melhor escolha devido ao seu baixo consumo de energia [27], quando comparado com o WiFi. Apenas estes 2 protocolos foram implementados mas a extensibilidade do sistema permite que outros tipos de protocolos sejam implementados.

Para usar um novo protocolo é necessário implementar as seguintes classes: ConnectionInfo que identifica o tipo de ligação, HostConfig solicita ao utilizador a informação necessária para criar um servidor localmente, HostRecord armazena a informação para criar um servidor localmente, NetworkConnection estabelece a ligação com o servidor ou aceita as ligações dos clientes conforme o caso, e por fim o ServerFinder que contém a informação para o cliente se ligar ao servidor como, por exemplo, endereço IP e a Porta no caso do TCP/IP.

Após a configuração da ligação, a ligação do lado cliente é gerida pela classe Connection, sendo que as ligações do servidor são geridas pelo NetworkManager onde a ligação em que cada cliente é representada por um ConnectionHandler.

O modelo de comunicação escolhido para comunicar entre os vários dispositivos é o modelo de Sockets. Tratando-se de um modelo simples que trata a informação como *arrays* de bytes foi necessário criar um módulo, o Módulo de Serialização que converte as mensagens em *arrays* de bytes e vice-versa.

Para o caso do servidor e o cliente se encontrarem no mesmo dispositivo foi criado um tipo de ligação directo chamado DirectConnection. Este método de comunicação permite que o cliente e o servidor comuniquem entre si através de uma *queue* onde as mensagens são depositadas pelo emissor e levantadas pelo receptor usando mecanismos de *lock* e *notify* do Java.

O DirectConnection foi criado devido à limitação do protocolo Bluetooth onde não existe a facilidade de enviar mensagens para a própria máquina como, por exemplo, o *localhost* do TCP/IP. Esta solução melhora o desempenho do sistema com a ausência da serialização das mensagens trocadas entre o cliente e servidor que se encontram no mesmo dispositivo.

Os diagramas de classes da Camada de Comunicações do Cliente e Servidor podem ser visualizados nas figuras Fig. A.7 e Fig. A.7 da Secção A.2 dos anexos.

4.4 Representação dos objectos partilhados

Qualquer objecto do jogo partilhado entre clientes tem de ser uma extensão do tipo DataUnit ou um sub-tipo da mesma. O tipo DataUnit armazena toda a informação necessária para aplicar as actualizações necessárias nos objectos partilhados, em especial, o identificador único do DataUnit, atribuído pelo servidor e que é global para todos os clientes. Dentro do tipo DataUnit existe o sub-tipo PositionableDU. O tipo PositionableDU transporta as coordenadas X e Y do objecto, que são usadas pelo servidor para a selecção das actualizações que devem ser propagadas usando o modelo de consistência VFC-reckon.

Na nossa aplicação demonstrativa existem alguns tipos de DataUnits como, por exemplo, ShipEntity_DataUnit que contém toda a informação que é necessário replicar pelos outros clientes. Para criar um novo tipo apenas é necessário estender o tipo DataUnit, ou sub-tipos, e implementar alguns métodos como, por exemplo, o método *merge()*, responsável por actualizar o estado do objecto com base nas actualizações recebidas.

A hierarquia e principais métodos da classe DataUnit podem ser consultados no diagrama de classes existente na Fig. A.4 da Secção A.2 dos anexos.

4.5 Especificação do Módulo de DeadReckoning

O Módulo de DeadReckoning é o módulo responsável pela estimação das próximas posições com base no histórico de posições e na posição actual. A estimação das posições é feita em classes especiais chamadas PredictionEngines (motor de estimação). Estas classes contêm um método que recebe o histórico de pontos do objecto e a posição actual e, com base nisso, devolve as próximas posições do objecto. No nosso sistema existem 2 tipos de motores de estimação, Linear e Circular.

O tipo de motor de estimação que deve ser usado na estimação de novas posições é especificado no PositionableDU do objecto através do método *setPredictionEngine()*. O

tipo de motor pode ser manualmente alterado no decorrer do jogo, invocando o mesmo método, mas com um motor diferente de estimação.

Para implementar um novo tipo de motor de estimação como, por exemplo, um motor com base numa trajectória oval, apenas é necessário estender o tipo PredictionEngine e no PositionableDU referente ao objecto da aplicação invocar o método *setPredictionEngine()* com o novo motor.

O diagrama de classes do Módulo de DeadReckoning encontra-se na Fig. A.3 da Secção A.2 dos anexos.

4.6 Gestão de Consistência de Objectos

A consistência de objectos e a selecção das actualizações que são necessárias propagar são definidas na Camada de Sessão do Servidor, mais propriamente na classe SessionServer. A selecção de actualizações para enviar aos clientes está definida dentro do ConsistencyManagementBlock (CMB) existente no servidor. Neste momento existem duas implementações do CMB: BasicCMB e VFCreckonCMB.

O BasicCMB é a representação da solução mais simples e que é normalmente usada nos jogos multi-jogador. Esta solução foi implementada no nosso sistema para efeitos comparação entre BasicCMB e VFCreckonCMB. No BasicCMB todas as actualizações são enviadas para todos os clientes funcionando como um *broadcast* na rede, não existindo qualquer mecanismo de selecção de actualizações. No caso do VFCreckonCMB a selecção de actualizações é feita através do modelo de consistência VFC-reckon.

O armazenamento dos objectos no servidor é feito numa estrutura chamada DataPool contida no CMB. O DataPool é responsável por armazenar e actualizar os objectos à medida que chegam actualizações dos vários clientes. As actualizações são aplicadas em série para evitar conflitos nas actualizações. Além das actualizações o CMB também permite que se registem novos objectos durante o decorrer de um jogo. Durante o registo de um novo objecto o DataPool está bloqueado para evitar conflitos com eventuais actualizações ou leituras ao DataPool.

O armazenamento de DataUnits no DataPool é feito usando um *array* como estrutura. Os DataUnits são indexados no *array* através do seu identificador. Como os *arrays* são estruturas que não permitem adicionar mais elementos além de um valor definido, para adicionar um novo objecto o DataPool converte o *array* num objecto do tipo *List*, adiciona o objecto e por fim converte-o de volta para um *array*. A estas operações foi dado o nome de *freeze* e *defrost*.

Os três métodos principais do CMB são: *updatesReceived()*, *computeUpdatesToDiffuse()* e *newPublishedObject()*. O método *updatesReceived()* é invocado quando são

```
@PhiAnnotation(
                zones = 3,
                zoneRange = { 350, 500, -1 },
                theta = { 1, 4, 8 },
                sigma = { 2, 10, 20 },
                niu = { 0.1, 0.2, 0.3},
                estimationPoints = {3, 0, 0}
)
```

Figura 4.1: Exemplo da especificação do phi

recebidas novas actualizações por parte dos clientes. O método *computeUpdatesTo-Diffuse()* é o método invocado durante cada ronda do servidor através um temporizador chamado RoundTrigger. Este método retorna as listas de actualizações que devem ser enviadas a cada cliente. Por último, o método *newPublishedObject()* é invocado quando um cliente pretende criar um objecto partilhado.

A extensibilidade do sistema permite que possam ser usados outros modelos de consistência, ou seja, outros CMB. Para definir outro modelo de consistência apenas é necessário estender a classe CMB e no Gestor de Sessão do Servidor seleccionar o novo CMB.

O diagrama de classes do CMB e das principais classes da Camada de Sessão do Servidor encontra-se na figura Fig. A.5 da Secção A.2 dos anexos.

4.7 Especificação do Modelo de Consistência VFC-reckon

A especificação do modelo de consistência é feita no objecto da aplicação que implementa a interface IVFCreckonApp apresentada na Secção 4.1. Recorrendo ao sistema de anotações do Java é possível definir o modelo de consistência phi de uma maneira simples e facilmente perceptível para o programador.

Na Fig. 4.1 encontra-se a representação de um modelo de consistência phi. O campo zones indica o número de zonas, ou seja, o número de anéis zn do VFC sendo que o último anel (-1) representa a área fora do anel mais exterior. Os restantes campos definem o grau de consistência de cada zona sendo que a 1ª posição de cada lista indica o grau de consistência do 1º anel, a 2ª posição do 2º anel e assim sucessivamente.

O campo zoneRange indica a dimensão do anel correspondente, o theta (Dimensão Temporal θ do VFC) indica o intervalo de tempo entre o envio de cada actualização usando como medida o número de rondas do servidor como, por exemplo, um valor de

4 significa que de 4 em 4 rondas o servidor envia as actualizações ao cliente. O valor de sigma (Dimensão Sequencial σ do VFC) indica número de actualizações que o cliente pode ignorar e, por último, o niu (Dimensão Valor v do VFC) indica a diferença máxima permitida entre os 2 DataUnits em percentagem. Por último, o valor estimationPoints indica o número de pontos que serão estimados entre cada actualização enviada do servidor para o cliente, ou seja, um valor de 3 indica que entre cada nova posição enviada do servidor para o cliente, o módulo de DeadReckoning do cliente vai estimar 3 posições intermédias. Um valor de 0 significa que não existem pontos para estimar.

O valor phi representado na anotação é enviado ao servidor durante a fase de estabelecimento de ligação entre o cliente e o servidor. Após a recepção do phi pelo servidor este é guardado e usado pelo Gestor de Consistência de Objectos VFCreckonCMB. Se durante a fase de estabelecimento de ligação não for definido um phi pelo cliente será usado um por omissão.

Relativamente aos pivôs, para especificar um objecto do jogo como um pivô do modelo de consistência VFC-reckon basta anotar o objecto com a anotação *@PivotAnnotation*. Neste caso, para o servidor distinguir os objectos que são pivôs, dos que não são, basta verificar se a classe do objecto do jogo contém a anotação.

4.8 Plataforma Android

O Android [1] é uma plataforma para dispositivos móveis, baseada na linguagem java, que possui uma máquina virtual própria chamada Dalvik Virtual Machine (DVM). Ao contrário da Java Virtual Machine, a DVM foi construída focando-se nas limitações dos dispositivos móveis como a memória, CPU e bateria.

Para o desenvolvimento deste sistema optámos pela versão 2.2 Froyo do Android por possuir grandes melhorias a nível de desempenho, principalmente em jogos, face à versão anterior, a 2.1 Eclair.

Existem 2 aspectos relevantes da arquitectura do Android relativos a este trabalho: Threads e Comunicação.

O Android possui suporte para threads bem como mecanismos de comunicação entre elas como Handlers e AsyncTasks. As threads são essenciais em jogos, para não influenciar a jogabilidade do jogo as operações mais pesadas devem ser feitas em background usando threads separadas. A comunicação entre threads torna-se mais importante dentro dos jogos onde, normalmente, existe uma thread responsável pela interface do utilizador e por desenhar as entidades no ecrã.

[1] http://developer.android.com

Relativamente à comunicação, o Android não permite o uso do Java RMI[2] . Para a comunicação entre dispositivos Android usámos comunicações com Sockets. Estes sockets usam protocolos como o WiFi, Bluetooth ou GPRS. O suporte de Serialização Java de objectos é um mecanismo importante para a transmissão de objectos, apesar de não ser muito usado neste sistema devido ao seu fraco desempenho quando, os intervalos de tempo entre cada mensagem são muito pequenos.

4.9 Jogo Multi-jogador Asteroids

Para a avaliação do sistema desenvolvemos um jogo multi-jogador, baseado no clássico asteroids. O nosso *middleware* foi usado para a comunicação e manutenção da consistência dos objectos do jogo entre os vários jogadores.

Tanto o jogo como o sistema foram desenvolvidos com o SDK da versão 2.2[3] da plataforma Android em conjunto com a ferramenta Eclipse 3.6.2[4]. A avaliação do jogo e sistema foram realizados em 2 emuladores[5] da plataforma Android e 2 tablets Samsung Galaxy com a versão Android 2.2. Apesar do jogo ter sido testado em tablets é compatível com dispositivos de menor dimensão como *smartphones*, adaptando a interface do jogo consoante as dimensões do ecrã do dispositivo.

O nosso jogo foi desenvolvido de raiz tratando-se de uma implementação de um jogo distribuído baseado no clássico Asteroids, mas neste caso, para plataformas móveis, nomeadamente a plataforma Android. Durante o jogo o objectivo do jogador é conseguir destruir mais asteróides que o jogador rival. Neste caso, se um dos jogadores colidir com um asteróide a sua nave perde uma percentagem da vida. Quando a vida de um jogador chega a 0 o jogador perde automaticamente o jogo.

As componentes mais relevantes da implementação do nosso jogo são: Gestão do estado do Jogo, Objectos do jogo, Gestor de objectos do jogo, Servidor do jogo, Componente gráfica do jogo e Controles do jogo.

4.9.1 Gestão do estado do jogo

O nosso jogo foi implementado seguindo o modelo de máquina de estados, sendo a máquina de estados gerida pela classe GameThread, existente na *package VFCreckon-Game.asteroids*, e que contém a *thread* principal do jogo. A *thread* principal do jogo tem como principais métodos: 1) *updatePhysics()* onde as componentes físicas dos objectos

[2]Java Remote Method Invocation - `http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136424.html`

[3]`http://developer.android.com/sdk/index.html`

[4]`http://www.eclipse.org/downloads/packages/release/helios/sr2`

[5]`http://developer.android.com/guide/developing/tools/emulator.html`

são actualizadas e 2) o método *doDraw()* responsável por desenhar as entidades do jogo e interface no ecrã do dispositivo.

Para facilitar a implementação do jogo, os estados do jogo nesta *thread* mapeiam-se quase directamente nos estados da máquina de estados cliente do nosso sistema, apresentados na Secção 3.4.4. A primeira etapa durante o arranque do jogo é o estabelecimento da ligação entre o cliente e o servidor o que corresponde ao estado Setup do jogo. O estado Setup corresponde ao estado Idle da máquina de estados cliente do sistema. Neste caso o jogador tem a possibilidade de escolher entre uma ligação Bluetooth e uma ligação WiFi. Após efectuada a escolha da ligação e inseridos os parâmetros da ligação como, por exemplo, o endereço IP e porta da ligação WiFi é efectuada a ligação com o servidor. Com o estabelecimento da ligação são enviados os parâmetros de configuração *phi* e são registados os objectos do jogo no servidor. O estabelecimento e configuração da ligação corresponde à transição do estado Idle para o estado Subscribed da máquina de estados cliente do sistema.

Com a configuração da ligação concluída é executada a *thread* do jogo que contém a máquina de estados do jogo. O estado inicial do jogo corresponde ao estado Ready o que corresponde ao estado Subscribed da máquina de estados cliente do sistema. Neste estado os jogadores têm no ecrã a mensagem *"Press to start..."*. O jogo tem início assim que um dos jogadores pressionar o ecrã.

Quando um jogador pressiona o ecrã, a máquina de estados do jogo transita do estado Ready para o estado Running o que corresponde à transição na máquina de estados cliente do sistema do estado Subscribed para o estado Active.

O estado Running, correspondente ao estado Active da máquina de estados cliente do sistema, é o estado principal do jogo. Neste estado as actualizações são enviadas periodicamente do Cliente para o Servidor e do Servidor para o Cliente.

Quando o jogo termina, a máquina de estados do jogo transita do estado Running para o estado Win ou para o estado Lose consoante a vitória ou derrota do jogador.

Em resumo, a mapeamento entre estados da máquina de estados cliente do sistema e estados do jogo encontra-se resumido na tabela Tab. 4.1.

Estados do Jogo	**Estados do Sistema**
Setup	Idle e Publishing
Ready	Subscribed
Running	Active
Win	Idle
Lose	Idle

Tabela 4.1: Mapeamento Estados do Jogo –> Estados do Sistema

4.9.2 Objectos do Jogo

Do ponto de vista do nosso sistema, a informação mais relevante existente no lado do jogo são os objectos do jogo partilhados entre os vários jogadores. Cada objecto do jogo, para o qual desejamos partilhar informação com outros jogadores tem um objecto do tipo DataUnit associado a ele. O DataUnit serve como uma cápsula que armazena a informação daquele objecto que é partilhada com os outros jogadores. É importante perceber que nem todo o objecto do jogo tem que ser partilhado com os outros jogadores. Apenas a informação crítica deve ser partilhada de modo a reduzir a quantidade de informação transmitida como, por exemplo, a sua posição no mapa do jogo.

No nosso jogo os objectos partilhados com os outros jogadores são o AsteroidEntity, IndAsteroidEntity, ShipEntity e ScoreEntity. Todos estes objectos contém um objecto do tipo DataUnit associado a ele e que armazenam a informação partilhada com os outros jogadores como, por exemplo, o objecto ShipEntity_DataUnit que corresponde ao DataUnit do objecto do jogo ShipEntity.

O ShipEntity é o objecto que representa a nave do jogador sendo o principal objecto do jogo. O ShipEntity é controlado pelo jogador através do ecrã do dispositivo onde o jogador indica a posição do ecrã para onde a nave se deve deslocar. O movimento do objecto é sempre feito segundo um deslocamento linear, alterando a direcção cada vez que o jogador pressiona o ecrã num ponto diferente. Tendo um deslocamento linear a maior parte do tempo, definiu-se como motor de estimação para este objecto o motor LinearEngine.

Os objectos do tipo AsteroidEntity são os asteróides que os jogadores têm que destruir para aumentar a sua pontuação. Quando este objecto é destruído são criados 2 novos asteróides mais pequenos. O movimento destes objectos é feito segundo uma trajectória linear pelo que o motor de estimação de novas posições associado a ele é o motor LinearEngine.

O objecto ScoreEntity é um objecto simples e que apenas contém a pontuação actual do jogador. A partilha deste objecto com os outros jogadores permite que seja apresentado no ecrã a pontuação actual de todos os jogadores. Neste caso não existe um motor de estimação associado pois a noção de posição no mapa do jogo não se aplica.

Objectos do tipo IndAsteroidEntity são asteróides indestrutíveis e cuja missão é sobretudo aumentar a dificuldade do jogo. Estes objectos seguem sempre uma trajectória circular e têm como motor de estimação de novas posições o motor CircularEngine.

Existe ainda outro objecto do jogo chamado ShootEntity que não é partilhado com os outros jogadores. Os tiros de cada jogador apenas existem localmente no seu dispositivo. Estes objectos são criados quando o jogador pressiona o botão *Fire* no ecrã do dispositivo. Se o objecto colidir com um asteróide, o asteróide é automaticamente

destruído e a pontuação do jogador é aumentada.

4.9.3 Gestor de Objectos do Jogo

O Gestor de Objectos do Jogo corresponde à classe EntityManager no nosso jogo. Este objecto tem a responsabilidade de gerir todos os objectos do jogo bem como gerir a comunicação entre o *middleware* e o jogo. Durante o ciclo normal do jogo a *thread* principal do jogo (GameThread) invoca no EntityManager os métodos *doDraw()* e *updatePhysics()* responsáveis por actualizar e desenhar todos os objectos do jogo, bem como gerir os objectos que são adicionados e removidos durante a configuração e execução do jogo.

Do ponto de vista do nosso *middleware* este é o objecto que faz a ponte entre o jogo e o sistema. O EntityManager implementa a interface IVFCreckonApp o que significa que implementa os métodos de *callback* do sistema (descrito na Secção A.1 dos anexos) como, por exemplo, novos objectos de outros jogadores ou outras notificações do sistema.

A especificação do modelo de consistência VFC-reckon (*phi*) utilizada neste jogo encontra-se especificada através da anotação *@PhiAnnoation* no topo da classe EntityManager.

O EntityManager contém uma instância do VFCreckonClient que lhe permite realizar os pedidos ao sistema. Os pedidos são maioritariamente pedidos para adicionar ou remover objectos tanto na fase de configuração do jogo como na execução do jogo. Os tipos de pedidos que o EntityManager solicita ao sistema encontram-se descritos na descrição da API do VFCreckonClient na Secção A.1 dos anexos.

4.9.4 Servidor do Jogo

A componente de servidor implementada no nosso jogo é muito simples pois assumimos que o dispositivo que desempenha o papel de servidor, desempenhava também papel o papel de cliente. A escolha foi meramente opcional pois não existe nenhuma limitação do sistema que obrigue o dispositivo servidor a ter também uma instância cliente a correr. Se o programador desejar pode executar apenas o servidor no dispositivo.

Na nosso jogo multi-jogador o servidor do jogo é gerido pela classe ServerHandler que implementa os métodos definidos em IVFCreckonServerListener (descrito na Secção A.1 dos anexos). A função da componente de servidor do lado do jogo é simplesmente notificar quando um jogador se liga ou desliga do servidor através de notificações simples no ecrã. O ServerHandler é o objecto do jogo que contém uma instância de VFCreckonServer, o que corresponde à componente de servidor do nosso sistema.

Apesar de termos optado por uma implementação simples da componente de servidor no nosso jogo, o programador, se assim desejar, pode desenhar uma interface para componente de servidor onde poderá, por exemplo, ver a listagem de jogadores ligados, ver a pontuação actual dos jogadores ligados ou até mesmo ser um espectador do jogo.

4.9.5 Componente gráfica do jogo

A componente gráfica do jogo é baseada numa *framework* da plataforma Android chamada Canvas2D[6] que disponibiliza uma API com funções básicas de desenho como, por exemplo, desenhar um círculo, um quadrado, escrever texto no ecrã, ou desenhar uma imagem no ecrã. Na nossa implementação, os gráficos do jogo são imagens PNG[7] que se deslocam no ecrã aplicando operações como rotações ou deslocamentos.

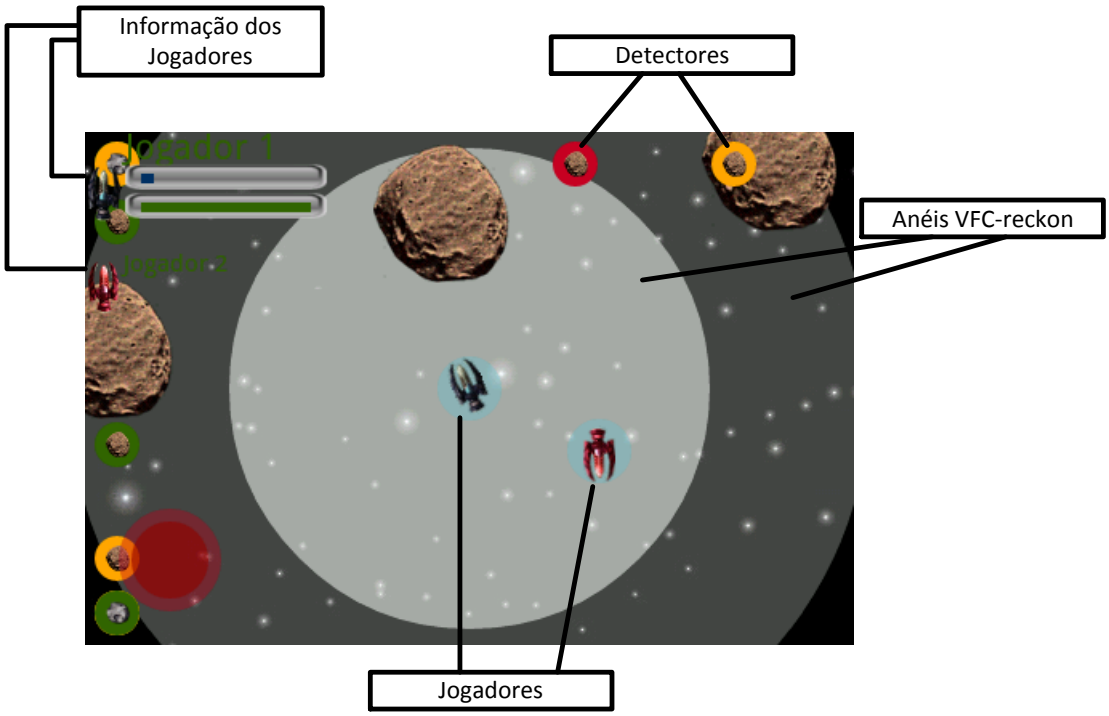

Figura 4.2: *Print screen* do nosso Jogo Asteroids

Na Fig. 4.2 encontra-se um *print-screen* do jogo onde podemos visualizar a interface e objectos do jogo desenhados no ecrã do dispositivo.

Os componente gráfica do jogo pode ser decomposta em: Objectos do Jogo, *Background*, limites do mapa e OSD.

[6] `http://developer.android.com/guide/topics/graphics/2d-graphics.html`

[7] O Portable Networks Graphics (PNG) foi escolhido por ser um formato de imagem *bitmap* sem perdas de informação.

Figura 4.3: Figuras usadas para representar os objectos do jogo no ecrã.

Os objectos do jogo são simples imagens no formato PNG que são desenhadas no ecrã e às quais são aplicadas transformações como rotações ou escalonamento da imagem. Na Fig. 4.3 estão representadas algumas das imagens dos objectos do jogo.

Um aspecto interessante do nosso jogo é que a nave não se desloca no ecrã. A nave do jogador situa-se sempre no centro do ecrã sendo que o que se desloca são as imagens de *background* e os outros objectos do jogo, dando uma ilusão de movimento da nave.

O mapa do jogo é um simples rectângulo com limites para os eixos X e Y. Quando um jogador, ou um asteróide, atinge o limite do mapa é automaticamente tele-transportado para o lado oposto do mapa. Os limites são apresentados no ecrã do dispositivo sendo representados por simples linhas azuis possibilitando que o jogador posso visualizar os limites de onde termina o mapa.

Para além de serem apresentados os objectos do jogo , *background*, e limites do mapa, o nosso jogo implementa também uma interface simples do tipo On-screen-display (OSD). No OSD são apresentados os detectores, pontuações dos jogadores, vida actual do jogador e velocidade da nave.

Os detectores são círculos simples, situados nas margens do ecrã, e que indicam a proximidade de objectos que não se encontram visíveis do ecrã como asteróides ou a nave de outro jogador. A proximidade é indicada pela cor do detector onde verde significa que o objecto se encontra a uma grande distância e vermelho significa que o objecto está muito próximo do ecrã.

A pontuação dos jogadores é descrita através do número de asteróides que o jogador já destruiu. Cada vez que o jogador abate um asteróide, é adicionada um asteróide de pequena de dimensão no canto superior esquerdo, debaixo do seu nome.

A vida actual do jogador e velocidade da sua nave são apresentados no canto superior esquerdo através de duas barras horizontais sendo que a barra de cima indica a velocidade da nave e a barra de baixo indica a vida actual do jogador.

Opcionalmente no ecrã pode também ser apresentada informação adicional como a *frame-rate* actual do jogo ou até mesmo os anéis do VFC-reckon.

A classe do jogo responsável por desenhar a parte gráfica do jogo é a classe ScreenManager. Esta classe disponibiliza os métodos usados pelo método *doDraw()* do ciclo principal do jogo (GameThread) como, *drawBackground()*, *drawMapBorders()* ou

drawOSDInterface(). Outra função do ScreenManager é fazer a conversão do espaço de coordenadas do ecrã para o espaço de coordenadas do jogo. Esta operação de conversão é necessária devido ao tipo de movimento da nave do jogador, onde não é a nave que se desloca, mas sim os outros objectos do jogo e *background*.

4.9.6 Controles do jogo

O nosso jogo muti-jogador é controlado através de uma interface *touch-screen*. No ecrã apenas é possível controlar o movimento da nave do jogador e accionar o botão de tiro que faz com que a nave dispare um tiro. O controlo da nave é bastante simples, o jogador apenas tem que pressionar no ecrã o ponto para onde quer que a nave se dirija. Os controlos do jogo foram implementados permitindo já a utilização de *multi-touch*. Isto significa que o jogador pode controlar a nave com um dedo no ecrã enquanto que pode pressionar o ecrã novamente para disparar um tiro.

Os controles do jogo são geridos pela classe ScreenControlsManager. Quando o jogador pressiona o ecrã, é gerado um evento, pelo sistema Android, chamado TouchEvent. Os TouchEvent's são enviados para aplicação e reencaminhados para a classe ScreenControlsManager que contém um método *handleTouchEvent()* responsável por processar os eventos do ecrã.

4.10 Sumário

Nesta secção descrevemos as principais características da implementação do sistema desde a API do sistema até às principais componentes e estruturas mais relevantes para o funcionamento do sistema. Durante a implementação do sistema houve a preocupação de implementar o sistema de maneira que a sua especificação e configuração fosse de fácil percepção para o programador. A extensibilidade do sistema permite ao programador usar novos protocolos de comunicação, novas trajectórias para a estimação de posições ou até mesmo novos modelos de consistência. Para a avaliação do sistema criámos um jogo multi-jogador baseado no clássico asteróides e usámos o nosso sistema para lidar com comunicação e consistência dos objectos do jogo entre os jogadores.

5

Avaliação

Conteúdo

5.1 Avaliação Qualitativa **61**
5.2 Avaliação Quantitativa **63**
5.2.1 Largura de banda 63
5.2.2 CPU 67
5.2.3 Memória 72
5.3 Sumário **74**

Para a avaliação deste trabalho desenvolvemos um jogo, compatível com tablets e smartphones Android, demonstrativo das vantagens do VFC-reckon. O jogo desenvolvido é um jogo 2D baseado no clássico Asteroids[1] em que o jogador compete com outros jogadores tentando conseguir destruir um maior número de asteróides que o adversário.

Os testes foram executados em 2 Tablets Samsung Galaxy com 512mb de RAM e um CPU a 1GHz, usando o protocolo Bluetooth como protocolo de comunicação.

Para avaliar o sistema usámos 3 tipos de modelos de consistência para selecção de actualizações por parte do servidor, que são eles: Basic, VFC e VFC-reckon.

Basic

No modelo Basic não existe qualquer selecção do que deve ser ou não enviado pelo servidor. Esta solução funciona como um *broadcast* em que todas as actualizações são enviadas para todos os clientes.

VFC

Com o VFC, o servidor efectua a selecção das actualizações que devem ou não ser enviadas através dos pivôs, anéis e graus de consistência, o que vai proporcionar uma redução no tráfego gerado na rede.

VFC-reckon

O VFC-reckon aplica o VFC em conjunto com técnicas de DeadReckoning que permitem mascarar a descontinuidade no movimento dos objectos.

Os 3 modelos têm um parâmetro de configuração em comum, o intervalo de tempo entre cada ronda do servidor. O tempo de ronda de cada modelo é especificado depois do nome como, por exemplo, o Basic 40 onde o número 40 indica que o intervalo de tempo de ronda é 40ms.

Como a ronda do servidor determina o intervalo mínimo a que o servidor envia novas actualizações aos clientes, o intervalo de tempo de envio das actualizações dos clientes para o servidor funciona com o mesmo valor. Como o servidor apenas usa a última actualização recebida do cliente, torna-se irrelevante para o cliente, enviar várias actualizações durante o tempo de uma ronda do servidor.

Para este trabalho adoptaram-se 2 tempos de ronda: 40ms e 160ms

Os 40ms estão relacionados com a *frame-rate*[2] do jogo. Um jogo que funciona a uma taxa de 25fps(*frames*-por-segundo), significa que por cada segundo existem, no máximo, 25 novas posições o que significa que a cada 40ms existe uma nova posição. O valor

[1] Exemplo do jogo em `http://www.thepcmanwebsite.com/media/flash_asteroids/`

[2] A frame-rate indica o número de *frames* por segundo do jogo

de 25fps é o mínimo que torna o movimento de um objecto fluido do ponto de vista do jogador.

O intervalo de 160ms foi escolhido por ser um múltiplo de 40ms. Isto significa que a ronda do servidor, e o envio de posições por parte do cliente, funcionam agora com um intervalo de 160ms. Na Secção 5.1 é avaliado o impacto na jogabilidade obtido com o aumento do intervalo de ronda.

Para intervalos acima de 200ms[3, 4] o erro na estimação de novas posições começa a ser demasiado notável na jogabilidade. A aplicação das técnicas de DeadReckoning permite minimizar a descontinuidade mas, o erro nas estimações começa a ser demasiado grande devido ao maior intervalo de tempo, entre cada posição real recebida.

Do ponto de vista da avaliação geral do sistema os modelos de consistência mais relevantes são o VFC-reckon 160 e o Basic 40. O VFC-reckon 160 corresponde à solução óptima proposta por nós onde reduzimos o número de mensagens trocadas sem afectar a jogabilidade. O Basic 40 é a solução simples de implementar e é normalmente a mais usada em jogos multi-jogador.

A avaliação do sistema está dividida segundo duas perspectivas: Avaliação Qualitativa e Avaliação Quantitativa.

A avaliação qualitativa tem como objectivo medir o impacto na jogabilidade criada pelo modelo VFC-reckon 160. A avaliação quantitativa pretende avaliar a escalabilidade do modelo de consistência VFC-reckon medindo os principais recursos usados pelo sistema: Largura de Banda, CPU e Memória.

5.1 Avaliação Qualitativa

Na solução VFC-reckon 160 o intervalo entre cada ronda é 4 vezes maior que 40ms. O valor de 40ms corresponde à taxa mínima a que as actualizações devem ser enviadas aos clientes, onde o movimento dos objectos é fluido. O facto de o intervalo ser 4 vezes maior gera uma descontinuidade no movimento das entidades do jogo, ou seja, em vez de terem um movimento fluido, o movimento passa a ser de um modo intermitente. O uso de um intervalo de tempo de 160ms só é aceitável, do ponto de vista do jogador, através da estimação dos pontos intermédios com o uso de técnicas de DeadReckoning.

A análise qualitativa tem como objectivo medir o impacto na jogabilidade criado pela aplicação do VFC-reckon. Com esta avaliação pretendemos comprovar dois pontos importantes:

1) O mecanismo de selecção de actualizações do VFC, com base na distância aos pivôs, não afecta a jogabilidade.

2) A descontinuidade criada pelo aumento do intervalo de ronda não é perceptível na

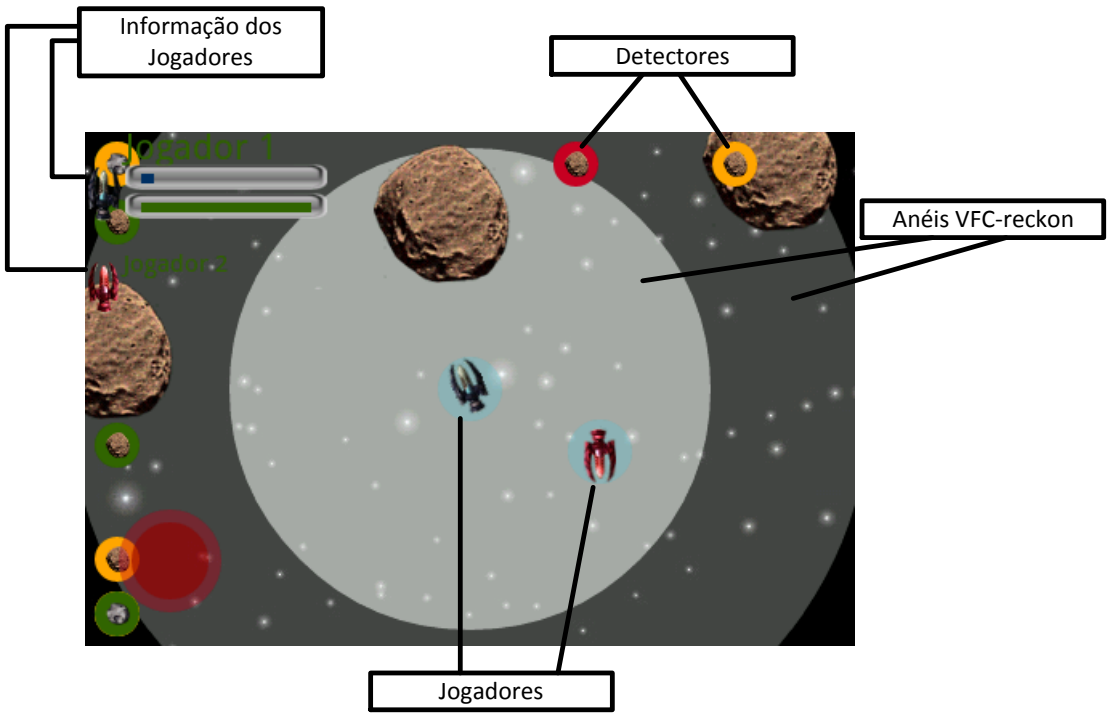

Figura 5.1: *Print screen* do nosso Jogo Asteroids

jobailidade com o uso de técnicas de DeadReckoning.

Esta análise foi realizada usando o nosso jogo demonstrativo em conjunto com o nosso *middleware*. O nosso jogo é baseado no clássico Asteroids, como apresentado na Fig. 5.1 onde cada jogador é representado por uma nave e o objectivo do jogo é destruir o maior número de asteróides que os seus rivais.

Para esta avaliação pedimos a várias pessoas para testarem o jogo e darem o seu *feedback* quanto ao impacto da jogabilidade, quando usamos VFC-reckon 160 vs a solução Basic 40. Os testes foram realizados com 10 pessoas, são maioritariamente jovens, de sexo masculino, com idades entre os 15 e 22 anos e utilizadores moderados em jogos multi-jogador. Todos os testes foram realizados segundo o mesmo esquema. Os primeiros jogos serviam sobretudo para o jogador se habituar aos controlos do jogo bem como à sua lógica. Após dominarem minimamente a lógica do jogo realizávamos um jogo usando o modelo Basic 40 e no jogo seguinte usámos o modelo VFC-reckon 160. A questão que se colocava aos jogadores no fim dos testes era se existia uma diferença notável na jogabilidade como, por exemplo, no movimento dos objectos, entre estas duas últimas versões que jogaram.

Os testes revelam que as diferenças entre as duas versões são praticamente imperceptíveis na jogabilidade pois nenhum dos jogadores detectou descontinuidade no

movimento dos objectos, ou outro tipo de diferenças. O foco do jogador centra-se sobretudo na zona do ecrã mais perto da nave do jogador. Como quase toda a zona do ecrã do dispositivo está abrangido pelos anéis de consistência mais consistentes, existe uma grande troca de mensagens para entidades situadas dentro dos anéis. O foco do jogador, para as entidades que se encontravam fora do ecrã do jogador, é muito reduzido. Tratando-se de um jogo que exige uma resposta muito rápida do jogador, ele tem que se focar nas entidades que se encontram dentro do ecrã e próximas da sua nave, ou seja, nas entidades que estão dentro dos anéis de maior consistência.

O uso do VFC-reckon 160 é notável para as entidades que estão a uma distância considerável da nave do jogador. No nosso jogo existem entidades chamadas detectores que indicam, nos limites do ecrã, a proximidade e localização dos asteróides. A proximidade é representada pela cor do detector. No VFC-reckon 160 estes detectores têm um movimento descontínuo pois a consistência destes objectos, e por sua vez a quantidade de mensagens, é muito menor. Esta descontinuidade passa quase despercebida aos jogadores pois o intervalo de tempo que o jogador se foca nestes detectores é muito pequeno. Os jogadores apenas olham durante cerca de 1 ou 2 segundos para descobrir onde estão os asteróides no mapa, tornando a focar-se, logo de seguida, na parte central do ecrã. O grau de interesse do jogador nos objectos do jogo pode ser definido, neste caso, como o intervalo de tempo que o jogador se foca numa entidade do jogo. Como se trata de um intervalo muito pequeno o jogador não tem tempo para se aperceber da descontinuidade no movimento dos detectores.

Com esta avaliação conseguimos provar que o impacto do VFC-reckon 160 na jogabilidade do jogo é quase imperceptível. A Gestão de Interesse, baseada na distância, é uma boa heurística como pudemos comprovar durante os testes. O uso de técnicas de DeadReckoning, especialmente nos anéis mais consistentes, torna o movimento dos objectos praticamente fluido sendo o movimento muito semelhante ao da versão Basic 40.

5.2 Avaliação Quantitativa

Para avaliar a escalabilidade do modelo de consistência VFC-reckon medimos os principais recursos usados pelo sistema: Largura de Banda, CPU e Memória.

5.2.1 Largura de banda

A largura de banda é dos recursos mais usados pelo sistema. A largura de banda usada pelo sistema foi medida segundo 3 tipos de avaliação: Módulo de Serialização, Avaliação sintética e Avaliação num jogo real. Com a avaliação do Módulo de Seria-

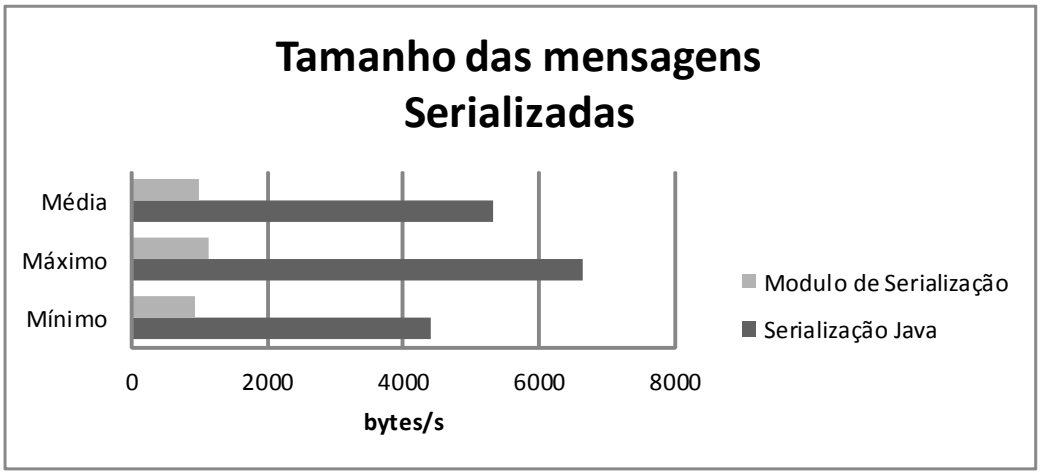

Figura 5.2: Tamanho de mensagens contendo 5 DataUnits

lização pretendemos medir o tamanho das mensagens geradas pelo módulo quando comparado com uma serialização por omissão do Java. A avaliação sintética pretende mostrar o comportamento do modelo VFC vs modelo Basic, para um caso simples com apenas uma entidade, de modo a conseguir ter exactamente os mesmos parâmetros para os testes e onde apenas se varia o modelo de consistência. A avaliação num jogo real pretende medir a largura de banda usada durante um jogo variando como parâmetros o modelo de consistência e o número de DataUnits.

O uso de técnicas de DeadReckoning é irrelevante nas medições da largura de banda. O DeadReckoning serve apenas para mascarar a ausência de mensagens do servidor, no lado do cliente, criada com o aumento do intervalo entre cada mensagem, não influenciando assim a largura de banda usada pelo sistema. Isto significa que nas medições relativas à largura de banda os modelos VFC-reckon 160 e VFC 160 apresentam resultados iguais.

5.2.1.A Módulo de Serialização

O Módulo de Serialização de mensagens tem um grande impacto na largura de banda. Devido à grande quantidade de mensagens trocadas entre o servidor e os clientes, a dimensão das mensagens deve também ser o mais pequena possível de modo a ocupar uma menor largura de banda.

A avaliação deste módulo foi feita através da comparação da dimensão das mensagens geradas pelo Módulo de Serialização e pela serialização por omissão do Java.

A Fig. 5.2 apresenta o tamanho das mensagens geradas pela serialização de 5 Data-

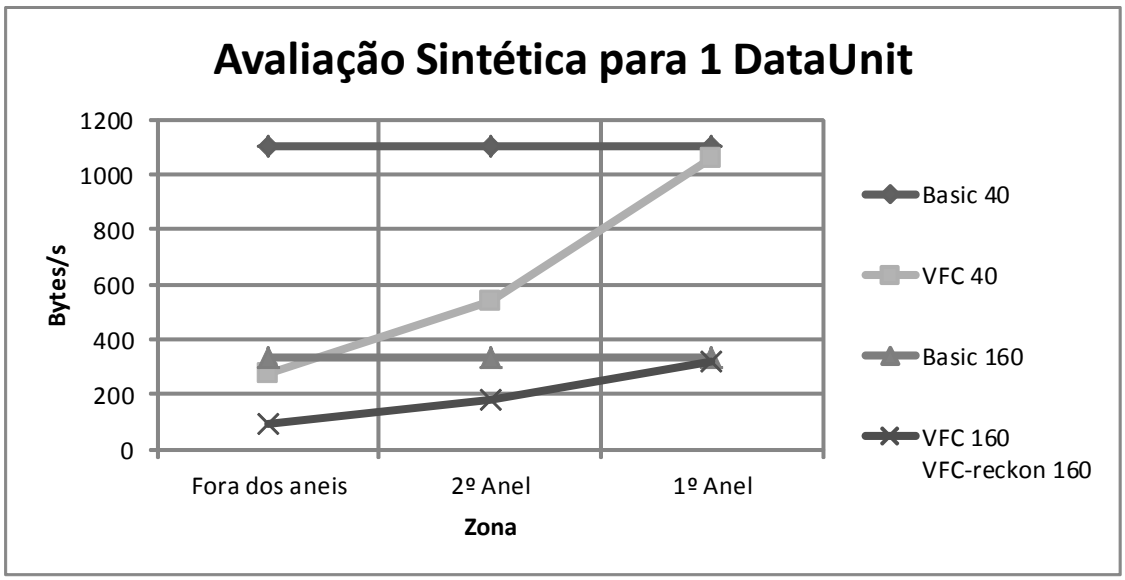

Figura 5.3: Largura de banda usada pelo sistema por 1 DataUnit

Units[3] durante 10 medições. Analisando o gráfico percebemos facilmente que o tamanho das mensagens usando a serialização do Java é significativamente maior do que usando o nosso Módulo de Serialização, resultando num maior *byterate*. Isto acontece porque o Java grava muita informação que, para o efeito, torna-se informação irrelevante como, por exemplo, a string do nome da classe quando podemos usar simplesmente um inteiro para identificar o tipo da classe do objecto.

5.2.1.B Avaliação Sintética

Para compreender o fluxo de mensagens geradas pelo VFC, e por sua vez pelo VFC-reckon, realizou-se uma avaliação sintética em que apenas existe 1 DataUnit por cada jogador sendo cada DataUnit o pivô desse jogador. Neste teste existe uma entidade que está parada no mapa do jogo enquanto a outra entidade atravessa o mapa de uma ponta à outra. Através da Fig. 5.3 torna-se notável a vantagem do VFC face a uma solução simples Basic com o mesmo intervalo de ronda. Enquanto que numa solução Basic o número de mensagens recebidas pelo cliente é sempre o mesmo, independentemente da posição da entidade no mapa, com o uso do VFC o número de mensagens recebidas no cliente varia conforme a distância dos pivôs aos objectos. No anel mais consistente (1º Anel) é necessário uma grande troca de informação o que faz com que o *byteRate*,

[3]O valor de 5 DataUnits está relacionado com a lógica do jogo onde normalmente existe a nave do Jogador, a sua Pontuação e 3 asteróides

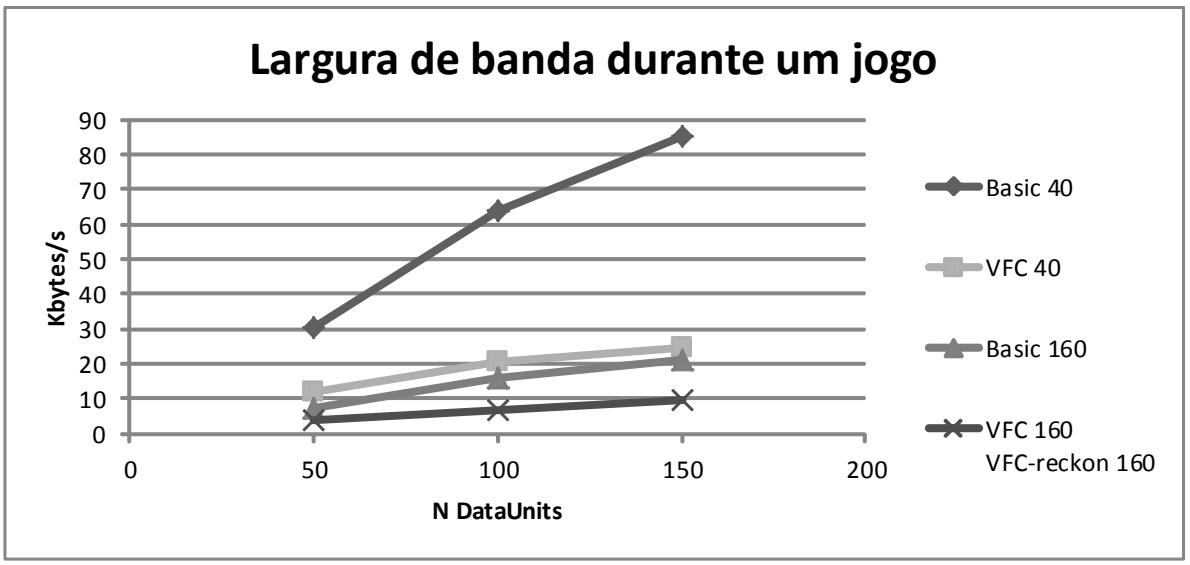

Figura 5.4: Largura de banda usada pelo sistema durante um jogo normal

dentro desse anel, seja semelhante ao da solução Basic. Com o VFC, à medida que começamos a afastar-nos do pivô o grau de consistência começa a diminuir o que corresponde a uma diminuição no *byteRate*.

5.2.1.C Avaliação num jogo real

Para medir os ganhos dos modelos VFC 160 e VFC-reckon 160 vs o modelo Basic 40 durante um jogo real, mediu-se o número de mensagens recebidas no lado do cliente, através do *byterate*. Efectuaram-se cerca de 12 testes, 4 para cada número de DataUnits, e em que apenas variávamos o número de DataUnits 50, 100 e 150 por cada cliente, existindo apenas 2 clientes. O limite máximo de 150 DataUnits está associado ao desempenho dos dispositivos. A partir de 150 DataUnits os dispositivos ficam lentos influenciando gravemente a jogabilidade e os resultados das medições. Os resultados apresentados na Fig. 5.4 correspondem às medias dos 12 valores registados.

A primeira conclusão que se destaca da Fig. 5.4 é a diminuição de tráfego para cerca 50% quando comparamos um modelo VFC com um modelo Basic com o mesmo tempo ronda. Esta redução demonstra que o critério de gestão de interesse do VFC, baseada na distância dos objectos, é extremamente útil pois muitos dos objectos no jogo estão a uma distância considerável dos pivôs e podem ser considerados como objectos menos interessantes para o jogador.

Analisando agora o comportamento dos principais modelos de consistência, VFC-

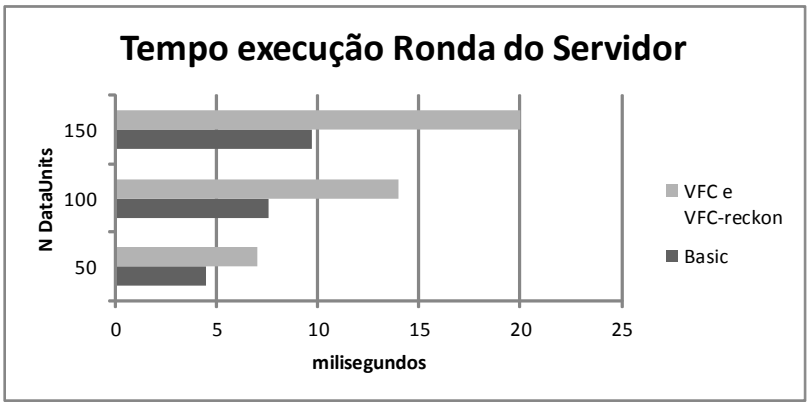

Figura 5.5: Tempo de CPU usado pela função de ronda do Servidor

reckon 160 e solução Basic 40, conseguimos, em média, uma redução de 88% na largura de banda usada. Esta redução ocorre devido ao maior intervalo entre o envio das mensagens, de 40ms para 160ms, e ao uso do VFC para a selecção de actualizações. Outro aspecto interessante é que o ganho é independente do número de DataUnits, ou seja, usar 50, 100 ou 150 DataUnits corresponde normalmente a um ganho de aproximadamente 88%.

É importante realçar que os ganhos do uso do VFC, e por sua vez do VFC-reckon, dependem fundamentalmente de 3 factores: Dimensão dos anéis, Graus de Consistência e Dimensão do mapa. Diminuir o tamanho dos anéis ou aumentar a dimensão do mapa vai diminuir o número de entidades que entram dentro dos anéis de consistência, que por sua vez, vai diminuir o número de mensagens que os clientes vão receber. A área dos anéis, neste teste, corresponde a cerca de 30% da área mapa o que, de certa maneira, é uma visão pessimista pois num jogo real espera-se que o rácio entre a área dos anéis e a área do mapa seja maior.

5.2.2 CPU

Esta avaliação pretende fazer os *benchmarks* e medir a carga de CPU usada nos principais módulos do sistema, bem como medir as reduções no uso de CPU que obtemos com o uso do modelo VFC-reckon 160 vs o modelo Basic 40.

5.2.2.A Ronda do Servidor

O VFC é um modelo de consistência que selecciona as actualizações que deve ou não enviar aos clientes com base na distância entre os objectos do jogo e os pivôs.

A selecção é feita pelo servidor onde para cada cliente é feita a selecção e envio das actualizações relevantes.

Em teoria, a ronda do servidor do VFC, e por sua vez do VFC-reckon, ocupa um maior tempo de CPU pois, ao contrário de uma solução Basic, é necessário processar e seleccionar as actualizações de cada cliente em particular. A ronda do servidor depende apenas do número de DataUnits e não do número de clientes. Do ponto de vista do cálculo e selecção de actualizações do VFC, processar 100 DataUnits de 2 clientes acaba por ser equivalente a processar 50 DataUnits de 4 clientes.

Para avaliar a quantidade de CPU usado nas rondas do servidor medimos o tempo necessário, em milisegundos, para a execução de cada ronda. Realizámos 6 medições onde variámos o modelo usado (VFC e Basic) e o número de DataUnits (50, 100 e 150) por cada cliente, tendo apenas 2 clientes.

Medindo os tempos de processamento de uma ronda do servidor os resultados apresentados na Fig. 5.5 demonstram o esperado. O VFC necessita de mais CPU devido à selecção das actualizações que devem, ou não, ser enviadas. Apesar do tempo de execução do servidor chegar ao dobro quando comparamos uma solução VFC com uma Basic para 100 e 150 DataUnits, é necessário analisar os resultados tendo em conta os intervalos de tempo entre cada ronda.

Se considerarmos as versões VFC 160 com a versão Basic 40 existe uma grande diferença relativamente ao número de rondas do servidor que existem por segundo. Na solução VFC 160 o intervalo entre cada ronda do servidor é 160ms, enquanto que, na solução Basic 40 o intervalo é de 40ms; isto significa que o número de rondas do servidor é 4 vezes maior para uma solução do tipo Basic 40. Como o tempo de execução da ronda do VFC é, no máximo, o dobro de uma ronda do tipo Basic, no global, uma solução do tipo VFC acaba por ocupar menos tempo de CPU.

5.2.2.B Módulo de DeadReckoning

Para avaliação deste módulo medimos o tempo que o módulo leva a estimar os pontos intermédios para vários DataUnits.

A Fig. 5.6 apresenta as médias das 10 medições retiradas dos dispositivos. O gráfico mostra o tempo que a função de estimar posições leva a executar após receber uma mensagem do servidor. Esta função existe apenas no lado cliente e depende da posição dos objectos em relação aos pivôs. Consoante o anel em que o objecto se encontre pode ser preciso estimar n pontos intermédios, em que n é um valor especificado em *phi* para aquele anel. No gráfico são apresentados os valores para 25, 50, 100 e 150 DataUnits por cada cliente, existindo apenas 2 clientes. Apesar da escala do gráfico estar em 25, 50, 100 e 150 DataUnits isto não significa que seja necessário estimar posições para

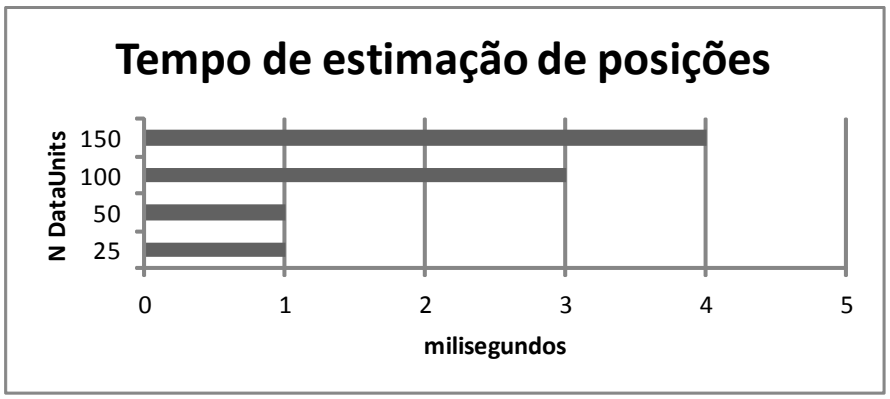

Figura 5.6: Tempo de CPU usado pela função de estimação de novas posições

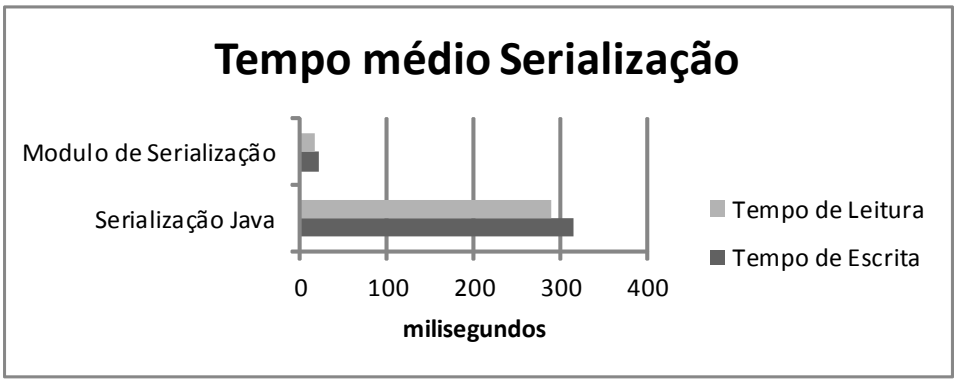

Figura 5.7: Tempo médio para a escrita e leitura de 5 DataUnits

todos os DataUnits. Apenas é necessário estimar posições para os objectos que sejam marcados pelo servidor, ou seja, os que estão dentro de anéis onde está definido um número de pontos para estimar.

Os tempos medidos e apresentados no gráfico mostram que a função de estimar as posições intermédias é uma função que ocupa pouco tempo de CPU quando comparada com outras como, por exemplo, a função da ronda do servidor. Com 150 DataUnits a função leva em média 4ms a ser executada sendo que se tratam de 4ms numa função que é invocada de 160 em 160ms se considerarmos o modelo VFC-reckon 160.

5.2.2.C Módulo de Serialização

A Fig. 5.7 apresenta os tempos médios, medidos em 2 emuladores, necessários para escrever e ler uma mensagem com 5 DataUnits[4], ou seja, serializar e reconstruir a mensagem. Os valores correspondem à média de 5 medições e foram feitas a partir de uma ferramenta de *profiling*[5] para os emuladores de Android. Este gráfico serve apenas para demonstrar a diferença relativa entre os 2 tipos de serialização, não sendo os tempos importantes mas sim o rácio de tempos; isto porque a ferramenta de *profiling* interfere com o tempo execução do sistema e o desempenho dos emuladores é muito mais baixo quando comparado com um dispositivo real.

Analisando os tempos podemos comprovar que o Módulo de Serialização leva, em média, 90% menos tempo a serializar as mensagens do que usando serialização Java. Esta redução no tempo está directamente ligada com o tamanho das mensagens já que é serializada muito menos informação e são usados maioritariamente tipos de dados simples como inteiros em vez de strings.

5.2.2.D Impacto no jogo com o VFC-reckon

Para medirmos o impacto do uso do sistema com o uso da solução VFC-reckon 160 face a uma solução Basic 40, fizemos uma análise através da *frame-rate* nos dois dispositivos. Nesta análise considerámos apenas as versões Basic 40 e VFC-reckon 160 por corresponderem à solução mais usada vs melhor solução proposta por nós. Apesar da *frame-rate* de um jogo ser variável, fizemos cerca de 10 medições e no fim realizámos as médias dos resultados obtidos.

A Fig. 5.8 demonstra os benefícios do uso do modelo de consistência VFC-reckon 160 quando comparado com o modelo Basic 40. A partir das 100 DataUnits por cada cliente, já não é possível manter a *frame-rate* ideal, ou seja, os 25fps. Os benefícios do modelo VFC-reckon 160 começam a notar-se a partir das 100 DataUnits por cada cliente. Isto acontece porque só a partir deste número, a carga de processamento das mensagens começa a ocupar uma grande fracção do tempo de processamento. Como o modelo VFC-reckon 160 reduz em grande escala o número de mensagens transmitidas pelo sistema, os seus ganhos só são visíveis quando a carga de processamento associada às mensagens ocupa um tempo considerável de CPU.

A partir das 100 DataUnits por cliente a carga das mensagens começa a ser notável, bem como os benefícios do uso do modelo VFC-reckon 160. Os ganhos face a uma

[4] O valor de 5 DataUnits está relacionado com a lógica do jogo onde normalmente existe a nave do Jogador, a sua Pontuação e 3 asteróides

[5] A ferramenta usada foi o Android DDMS. `http://developer.android.com/guide/developing/debugging/ddms.html`

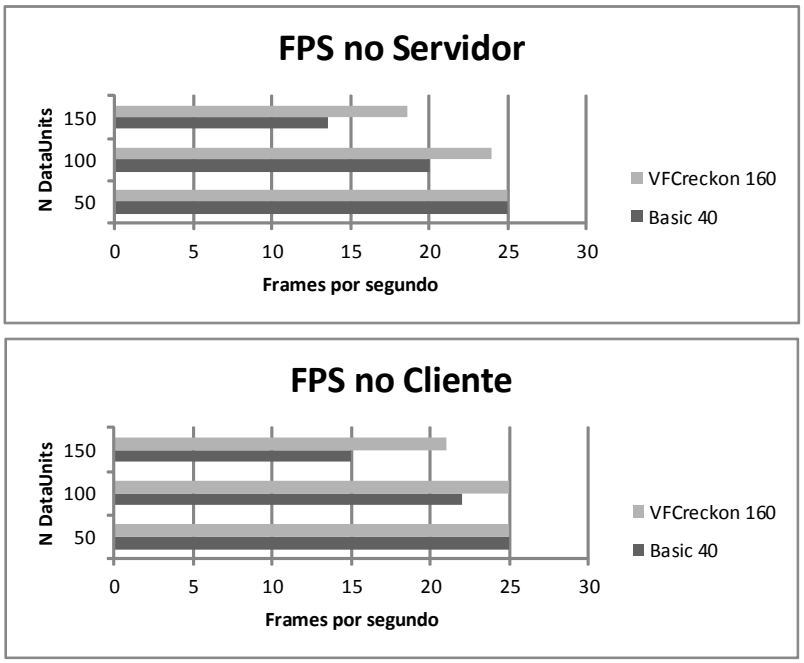

Figura 5.8: Comparação da frame-rate de VFC-reckon 160 vs Basic 40

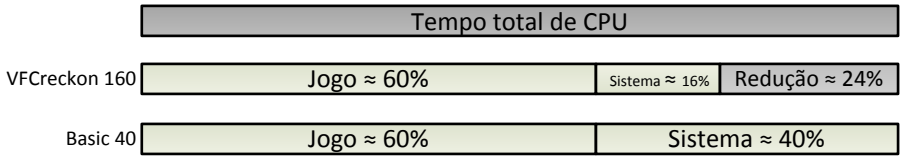

Figura 5.9: Descrição detalhada de tempo de CPU usado

solução Basic 40 são evidentes através dos gráficos. Apesar de com 100 DataUnits já existirem ganhos consideráveis, os ganhos de CPU são maiores, quanto maior for o número de DataUnits. Mais uma vez isto acontece porque quanto maior é a carga do processamento de mensagens, maior é o ganho com o uso do modelo VFC-reckon 160.

Os resultados demonstram ganhos entre 12% e, no máximo, 24%. Apesar de os resultados não parecerem satisfatórios é preciso ter em conta que não é apenas a carga de processamento de mensagens que aumenta com o aumento do número de DataUnits. O aumento do número de DataUnits aumenta também a carga de processamento relativa ao jogo como, por exemplo, a componente responsável por actualizar as componentes físicas dos objectos e a componente gráfica do jogo responsável por desenhar as entidades no ecrã.

Através de uma ferramenta de *profiling* conseguimos verificar que a carga do nosso sistema, com o modelo Basic 40, no dispositivo que desempenha o papel de servidor é de cerca 55% e no dispositivo que desempenha o papel de cliente cerca de 40%. Isto significa que quando aplicamos o modelo VFC-reckon 160, são estas as componentes que estamos a reduzir, através da diminuição do número de mensagens. Quando no cliente temos uma diminuição geral de 24% no tempo de CPU para 150 DataUnits, com o uso do VFC-reckon 160, isto significa que subtraímos 24% aos 40% pois a carga de CPU relativa ao jogo permaneceu inalterada, como apresentado na Fig. 5.9. Se considerarmos apenas a componente da carga do nosso sistema, a redução geral de 24% no uso do CPU, significa que o nosso sistema utiliza menos 60% de tempo CPU.

Outra das conclusões importantes que se pode retirar da Fig. 5.8 é que a carga de processamento de um dispositivo que tem o papel de servidor não é muito diferente da carga de processamento de um dispositivo que tenha o papel de cliente. Isto é visível através do *frame-rate* do jogo, em que a diferença é no máximo 3fps o que vem comprovar que a execução papel de servidor não tem um impacto muito grande no sistema.

5.2.3 Memória

Para avaliar a memória usada pelo sistema medimos a memória através de uma ferramenta de *profiling* em dois emuladores com o jogo e sistema em execução. Como o jogo e o sistema estão no mesmo processo, não é possível distinguir qual a percentagem de memória pertencente ao Sistema e qual a percentagem de memória pertencente ao Jogo.

As medições obtidas mostram a memória *heap* usada pelo processo. Nesta memória são guardadas as Classes, as instâncias das Classes (Objectos) e os tipos básicos como *short*, *int*, *byte*, *char*, etc... É de realçar que a memória *heap* não corresponde à memória total do processo. No nosso caso, os *bitmaps* relativos às imagens do jogo, são armazenados numa zona diferente da memória.

Nesta avaliação pretende-se comparar o modelo VFC-reckon 160 vs Basic 40 e ver o comportamento dos 2 modelos à medida que se varia o número de DataUnits entre 25 e 100 por cliente. Apenas a memória *heap* é realmente importante pois é onde os DataUnits e estruturas auxiliares para o funcionamento dos 2 modelos são armazenados. A memória usada pelos *bitmaps* acaba por se tornar irrelevante nestas medições já que não varia com o número de DataUnits.

As nossas medições foram feitas ao fim de algum tempo de execução do Jogo e Sistema. Apesar dos valores terem a tendência de aumentar com o decorrer do jogo, o *Garbage Collector* do Android vai removendo da memória os objectos que já não são necessários. Com a execução do *Garbage Collector* os valores mantêm-se, em média,

os mesmos ao longo do tempo.

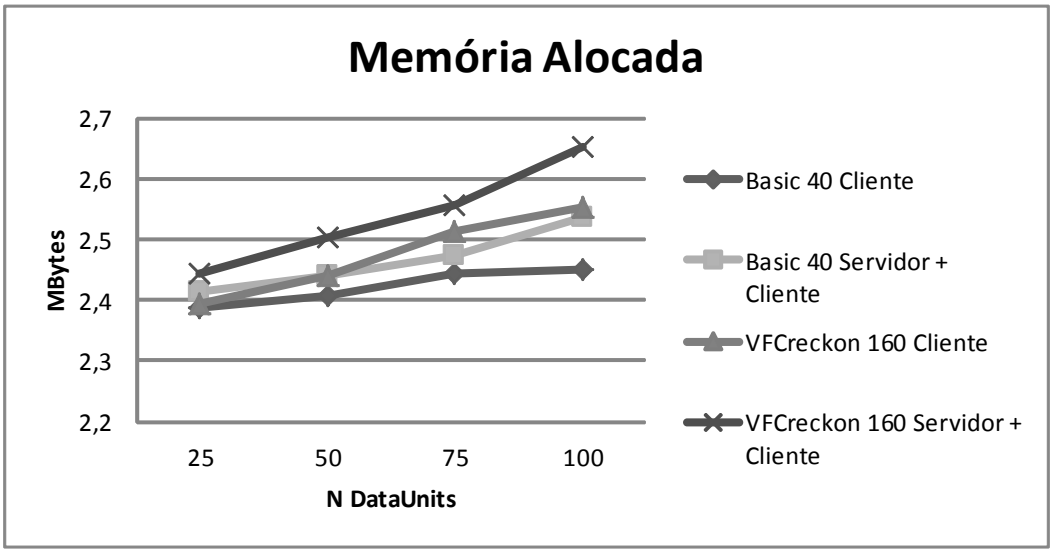

Figura 5.10: Memória alocada durante a execução pelo sistema e jogo

Na Fig. 5.10 temos os resultados das medições nos dispositivos com os papeis Cliente e Cliente+Servidor[6]. Durante as medições fomos alterando o número de DataUnits bem como os modelo usados, Basic 40 e VFC-reckon 160.

Através do gráfico podemos tirar várias conclusões. A primeira conclusão que se retira é que a quantidade de memória *heap* usada pelo jogo e sistema, independentemente do número de DataUnits, é relativamente pequena. O mínimo registado nas medições foi de 2,4Mbytes, enquanto que, o máximo foi de 2,65Mbytes. Isto significa que a diferença da pior solução para a melhor solução é, no máximo, 250Kbytes. No total, a quantidade de memória usada por todo o processo, incluindo *heap* e *bitmaps*, ronda os 11Mbytes. Mesmo para dispositivos móveis este número é relativamente pequeno pois, hoje em dia, qualquer destes dispositivos possui mais que 128Mbytes de memória.

Comparando agora os modelos de consistência Basic 40 e VFC-reckon 160 os resultados das medições demonstram o esperado. O VFC-reckon 160 necessita de mais memória devido à aplicação do VFC no lado do servidor e aplicação de DeadReckoning no lado do cliente.

O VFC, no lado do servidor, precisa de mais memória porque executa mais funções como, a selecção de actualizações. As operações de cálculo e selecção de actualizações precisam de alocar estruturas auxiliares para o seu funcionamento como, por exemplo, o conjunto de actualizações para um cliente em específico.

[6]Cliente+Servidor significa que o dispositivo tem o papel de Cliente e de Servidor ao mesmo tempo

No lado do cliente a maior quantidade de memória alocada deve-se, sobretudo, ao Módulo de DeadReckoning. Apesar de limitarmos o tamanho da estrutura que armazena o histórico de posições, a dimensão do histórico ainda é considerável o que faz com o cliente que usa o Módulo de DeadReckoning precise de mais memória.

Apesar do VFC-reckon 160 precisar mais memória que o Basic 40, a diferença entre os dois é relativamente pouca se tivermos em conta o tamanho total da *heap*. Comparando as várias medições, o *overhead* para o cliente é de cerca de 50Kbytes e, para o Cliente+Servidor, 100Kbytes o que corresponde a 2% e 4%.

5.3 Sumário

Nesta secção realizámos a avaliação do nosso sistema segundo uma perspectiva Qualitativa e Quantitativa. Para a avaliação foi criado um jogo, baseado no clássico Asteroids, com o objectivo de demonstrar as vantagens do VFC-reckon.

Na avaliação qualitativa avaliámos o impacto na jogabilidade criado pelo uso do modelo de consistência VFC-reckon vs uma solução Basic. Os testes realizados mostram que o impacto do VFC-reckon na jogabilidade é praticamente imperceptível. Através dos testes pudemos provar que a heurística da distância como critério de Gestão de Interesse é uma boa heurística pois a maior parte do foco do jogador situa-se na área mais próxima do seu avatar.

Na avaliação quantitativa avaliámos os principais recursos usados pelo sistema: Largura de banda usada, tempo de CPU e memória. Os resultados demonstram que a largura de banda diminui cerca de 88% com o uso do modelo de consistência VFC-reckon comparando com uma solução Basic. O tempo de CPU também diminui com o uso do VFC-reckon. Apesar do VFC-reckon exigir mais processamento para o servidor efectuar a selecção das actualizações, o sistema em si acaba por usar menos CPU devido à redução do número de mensagens. Com a diminuição do número de mensagens existe menos processamento de mensagens por parte do CPU como, por exemplo, serialização de mensagens. Os resultados demonstram que o tempo de CPU usado pelo nosso sistema diminui em cerca de 60% com o uso do VFC-reckon vs uma solução Basic. Relativamente à memória usada, o impacto do uso do VFC-reckon quando comparado com uma solução Basic é muito pequeno. Tanto para o dispositivo que tem o papel de cliente como para o dispositivo que tem o papel de servidor e cliente, o *overhead* por usar o VFC-reckon acaba por estar ordem dos 2% para o cliente e 4% para o cliente e servidor.

6

Conclusões

Conteúdo

6.1 Trabalho Futuro **77**

Neste trabalho foram apresentados vários mecanismos que permitem mascarar a latência numa rede e reduzir o tráfego necessário entre jogos multi-jogador. As técnicas mais usadas na redução de tráfego em jogos multi-jogador, são as técnicas de Gestão de Interesse. Estas técnicas seleccionam apenas os eventos que é necessário enviar para aquele jogador, descartando assim os menos relevantes. Relativamente aos dispositivos móveis e redes ad-hoc, é pouco o trabalho desenvolvido no âmbito da consistência em jogos multi-jogador. Os sistemas actuais não estão optimizados para os dispositivos móveis e não são flexíveis quanto ao tipo de jogo e especificação do modelo de consistência.

O objectivo deste trabalho é aumentar a escalabilidade dos jogos através da redução do número de mensagens o que corresponde a um menor consumo de CPU e largura de banda dos dispositivos móveis. O nosso trabalho consiste na concepção e desenvolvimento de um modelo de consistência chamado VFC-reckon. O VFC-reckon corresponde à junção do modelo de consistência VFC, baseado em técnicas de Gestão de Interesse, com técnicas de DeadReckoning.

O nosso sistema segue o modelo Cliente-Servidor, e corresponde a um *middleware* para a plataforma Android. Para a avaliação foi desenvolvido um jogo demonstrativo com o objectivo de mostrar as vantagens do VFC-reckon para ambientes móveis em redes ad-hoc.

A extensibilidade e simplicidade do sistema foram requisitos tidos em conta durante a construção do sistema. A simplicidade do sistema permite que o programador defina o modelo de consistência VFC-reckon de uma maneira simples através de anotações Java. O sistema suporta comunicação baseada nos protocolos de comunicação mais comuns como WiFi ou Bluetooth e é extensível a outros tipos de tecnologias de comunicação. Relativamente à estimação de posições baseado em técnicas de DeadReckoning, estão implementados motores de estimação para trajectórias lineares e trajectórias circulares. O nosso sistema permite a criação de novos motores de estimação para outros tipos de trajectórias que sejam necessário implementar. Actualmente encontram-se implementados os modelos de consistência VFC-reckon e Basic. A definição de outro modelo de consistência é possível através da extensão de uma classe interna.

Para a avaliação do sistema foi criado um jogo simples multi-jogador baseado no clássico Asteroids. A avaliação decompõe-se em duas componentes principais: Avaliação Qualitativa e Avaliação Quantitativa.

Para a avaliação qualitativa medimos o impacto na jogabilidade no nosso jogo, criado pelo uso do modelo de consistência VFC-reckon vs uma solução Basic. Os testes realizados demonstram que o impacto do VFC-reckon é praticamente imperceptível do ponto de vista dos jogadores. Através dos testes pudemos também provar que a heurística da

distância como método de Gestão de Interesse é uma boa heurística pois a maior parte do foco do jogador situa-se na área mais próxima do seu avatar.

Na avaliação quantitativa avaliámos os principais recursos usados pelo sistema: Largura de banda, tempo de CPU e memória. Os resultados demonstram que a largura de banda diminui cerca de 88% com o uso do modelo de consistência VFC-reckon 160 quando comparado com uma solução do tipo Basic 40. O tempo de CPU também diminui com o uso do VFC-reckon. Apesar do VFC-reckon exigir mais processamento para o servidor efectuar a selecção das actualizações o sistema em si acaba por usar menos CPU devido à grande redução no número de mensagens. Com a diminuição do número de mensagens existe menos processamento de mensagens por parte do CPU como, por exemplo, serialização de mensagens. Os resultados demonstram que o tempo de CPU usado pelo nosso sistema diminui em cerca de 60% com o uso do VFC-reckon vs uma solução do tipo Basic. A memória usada pelo sistema varia pouco quando comparamos as versões Basic 40 e VFC-reckon 160. Tanto para o dispositivo que tem o papel de Cliente, como para o dispositivo que tem o papel de Servidor e de Cliente, o impacto por usar o VFC-reckon acaba por estar entre os 2% e 4%.

6.1 Trabalho Futuro

Actualmente a especificação do tipo de trajectória usado para a estimação de novas posições é feita através da invocação de um método chamado *setPredictionEngine()*. Uma alternativa para este modelo de especificação de trajectória seria o próprio motor de estimação conseguir identificar padrões na trajectória do objecto e com isso deduzir se a trajectória que o objecto segue é, por exemplo, linear ou circular. Outra melhoria relativamente à estimação de posições é a alteração do comportamento do sistema de estimações quando ocorrem variações bruscas do tipo de movimento do objecto. Neste caso o sistema podia optar por aumentar a taxa de envio de actualizações e parar temporariamente a estimação de posições.

A grande heterogeneidade dos dispositivos móveis e a diferença de cargas de CPU entre os dispositivos que efectuam o papel de Servidor dos que efectuam o papel de Clientes põem muitas vezes o dispositivo que desempenha o papel de servidor em desvantagem devido ao maior consumo de recursos do sistema. Como solução para este problema poderia existir um protocolo de negociação, entre os vários dispositivos, com o objectivo de decidir qual o dispositivo que deve desempenhar o papel de servidor baseando-se em dados como, por exemplo, percentagem de bateria, capacidade de CPU ou largura de banda disponível.

O modelo de consistência VFC-reckon segue a lógica de um modelo estático que se

define no arranque da aplicação e mantem-se inalterado durante o decorrer da aplicação. Com base na informação do jogo ou do sistema operativo como, por exemplo, a *frame-rate* ou a autonomia da bateria, o servidor poderia ajustar a dimensão dos anéis do VFC e/ou ajustar os parâmetros de configuração da estimação.

Neste momento o nosso sistema não suporta tolerância a faltas. Uma proposta baseada neste contexto seria criar a possibilidade do utilizador poder abandonar o jogo, livremente ou acidentalmente, e regressar mais tarde através de mecanismos que permitam gravar e recuperar o estado actual do jogo.

Bibliografia

[1] J. Haartsen, “Bluetooth - the universal radio interface for ad-hoc, wireless connectivity,” Tech. Rep., 1998.

[2] “Ieee standard for information technology- telecommunications and information exchange between systems- local and metropolitan area networks- specific requirements part ii: Wireless lan medium access control (mac) and physical layer (phy) specifications,” IEEE Std 802.11g-2003 (Amendment to IEEE Std 802.11, 1999 Edn. (Reaff 2003) as amended by IEEE Stds 802.11a-1999, 802.11b-1999, 802.11b-1999/Cor 1-2001, and 802.11d-2001), pp. i –67, 2003.

[3] 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. New York, NY, USA: ACM, 2002, pp. 23–29. [Online]. Available: http://dx.doi.org/10.1145/507670.507674

[4] ——, “On the suitability of dead reckoning schemes for games,” in Proceedings of the 1st workshop on Network and system support for games, ser. NetGames ’02. New York, NY, USA: ACM, 2002, pp. 79–84. [Online]. Available: http://doi.acm.org/10.1145/566500.566512

[5] Y. Saito and M. Shapiro, “Optimistic replication,” ACM Comput. Surv., vol. 37, pp. 42–81, March 2005. [Online]. Available: http://doi.acm.org/10.1145/1057977.1057980

[6] J.-S. Boulanger, J. Kienzle, and C. Verbrugge, “Comparing interest management algorithms for massively multiplayer games,” in Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games, ser. NetGames ’06. New York, NY, USA: ACM, 2006. [Online]. Available: http://doi.acm.org/10.1145/1230040.1230069

[7] K. Prasetya and Z. D. Wu, “Performance analysis of game world partitioning methods for multiplayer mobile gaming,” in Proceedings of the 7th ACM SIGCOMM Workshop on Network and System Support for Games, ser.

NetGames '08. New York, NY, USA: ACM, 2008, pp. 72–77. [Online]. Available: http://doi.acm.org/10.1145/1517494.1517509

[8] S. Xiang-bin, W. Yue, L. Qiang, D. Ling, and L. Fang, "An interest management mechanism based on n-tree," in Proceedings of the 2008 Ninth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing. Washington, DC, USA: IEEE Computer Society, 2008, pp. 917–922. [Online]. Available: http://portal.acm.org/citation.cfm?id=1443223.1443755

[9] T. A. Funkhouser, "Ring: A client-server system for multi-user virtual environments," in Symposium on Interactive 3D Graphics, 1995, pp. 85–92.

[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] C. E. Bezerra, F. R. Cecin, and C. F. R. Geyer, "A3: A novel interest management algorithm for distributed simulations of mmogs," in Proceedings of the 2008 12th IEEE/ACM International Symposium on Distributed Simulation and Real-Time Applications, ser. DS-RT '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 35–42. [Online]. Available: http://dx.doi.org/10.1109/DS-RT.2008.11

[12] N. Santos, L. Veiga, and P. Ferreira, "Vector-field consistency for ad-hoc gaming," in Proceedings of the ACM/IFIP/USENIX 2007 International Conference on Middleware, ser. Middleware '07. New York, NY, USA: Springer-Verlag New York, Inc., 2007, pp. 80–100. [Online]. Available: http://portal.acm.org/citation.cfm?id=1516124.1516131

[13] C. Griwodz, "State replication for multiplayer games," in Proceedings of the 1st workshop on Network and system support for games, ser. NetGames '02. New York, NY, USA: ACM, 2002, pp. 29–35. [Online]. Available: http://doi.acm.org/10.1145/566500.566505

[14] J. Smed, T. Kaukoranta, H. Hakonen, and O. L. M. E. Ab, "A review on networking and multiplayer computer games," Tech. Rep., 2002.

[15] N. Krishnakumar and R. Jain, "Escrow techniques for mobile sales and inventory applications," Wirel. Netw., vol. 3, pp. 235–246, August 1997. [Online]. Available: http://dx.doi.org/10.1023/A:1019161318592

[16] H. Yu and A. Vahdat, "Design and evaluation of a conit-based continuous consistency model for replicated services," ACM Trans. Comput. Syst., vol. 20,

pp. 239–282, August 2002. [Online]. Available: http://doi.acm.org/10.1145/566340.566342

[17] N. Krishnakumar and A. J. Bernstein, “Bounded ignorance: a technique for increasing concurrency in a replicated system,” ACM Trans. Database Syst., vol. 19, pp. 586–625, December 1994. [Online]. Available: http://doi.acm.org/10.1145/195664.195670

[18] N. Preguiça, J. L. Martins, M. Cunha, and H. Domingos, “Reservations for conflict avoidance in a mobile database system,” in Proceedings of the 1st international conference on Mobile systems, applications and services, ser. MobiSys ’03. New York, NY, USA: ACM, 2003, pp. 43–56. [Online]. Available: http://doi.acm.org/10.1145/1066116.1189038

[19] E. Cronin, B. Filstrup, A. R. Kurc, and S. Jamin, “An efficient synchronization mechanism for mirrored game architectures,” in Proceedings of the 1st workshop on Network and system support for games, ser. NetGames ’02. New York, NY, USA: ACM, 2002, pp. 67–73. [Online]. Available: http://doi.acm.org/10.1145/566500.566510

[20] D. Min, D. Lee, B. Park, and E. Choi, “A load balancing algorithm for a distributed multimedia game server architecture,” in Proceedings of the IEEE International Conference on Multimedia Computing and Systems - Volume 2, ser. ICMCS ’99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 882–. [Online]. Available: http://portal.acm.org/citation.cfm?id=839287.841868

[21] J. Chen, B. Wu, M. Delap, B. Knutsson, H. Lu, and C. Amza, “Locality aware dynamic load management for massively multiplayer games,” in Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming, ser. PPoPP ’05. New York, NY, USA: ACM, 2005, pp. 289–300. [Online]. Available: http://doi.acm.org/10.1145/1065944.1065982

[22] M. Assiotis and V. Tzanov, “A distributed architecture for mmorpg,” in Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games, ser. NetGames ’06. New York, NY, USA: ACM, 2006. [Online]. Available: http://doi.acm.org/10.1145/1230040.1230067

[23] W. Cai, P. Xavier, S. J. Turner, and B.-S. Lee, “A scalable architecture for supporting interactive games on the internet,” in Proceedings of the sixteenth workshop on Parallel and distributed simulation, ser. PADS ’02. Washington,

DC, USA: IEEE Computer Society, 2002, pp. 60–67. [Online]. Available: http://portal.acm.org/citation.cfm?id=564062.564073

[24] A. P. Negrão, "Vfc large scale: Consistency of replicated data in large scale networks," Instituto Superior Técnico, november 2009.

[25] D. de Morais Tiago Lage, "Vfc para mobihoc.net - integração de modelo de consistência adaptado a jogos multi-utilizador na .net framework.: Vector-field consistency for .net multiplayer games," Instituto Superior Técnico, Av. Rovisco Pais, 1, november 2010.

[26] B. F. da Costa Pereira Loureiro, "Vfc4fps - vector-field consistency for a first person shooter game," Instituto Superior Técnico, Av. Rovisco Pais, 1, november 2010.

[27] R. Balani, "Energy consumption analysis for bluetooth, wifi and cellular networks," University of California at Los Angeles, Tech. Rep., 2007.

[28] M. Kumar M. M, A. Thawani, S. V, and Y. N. Srikant, "Analysis of application partitioning for massively multiplayer mobile gaming," in Proceedings of the 1st international conference on MOBILe Wireless MiddleWARE, Operating Systems, and Applications, ser. MOBILWARE '08. ICST, Brussels, Belgium, Belgium: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2007, pp. 28:1–28:6. [Online]. Available: http://portal.acm.org/citation.cfm?id=1361492.1361527

[29] S. M. Riera, O. Wellnitz, and L. Wolf, "A zone-based gaming architecture for ad-hoc networks," in Proceedings of the 2nd workshop on Network and system support for games, ser. NetGames '03. New York, NY, USA: ACM, 2003, pp. 72–76. [Online]. Available: http://doi.acm.org/10.1145/963900.963907

A

Anexos

Conteúdo

A.1 API do Sistema **84**
A.2 Diagramas de Classes do Sistema **85**

A.1 API do Sistema

Descrição da API de Cliente do Sistema:

VFCreckonClient	
Nome do Método	**Descrição**
subscribe()	Iniciar a fase de configuração do Jogo
publish()	Publicar novos DataUnits no Servidor
addDelayedPublish()	Guarda objectos para publicar numa *cache*
flushDelayedPublish()	Publica os objectos na cache no Servidor
delayedPublishEntity()	publish() na próximo ronda do Cliente –> Servidor
enable()	Iniciar a fase activa do Jogo
disable()	Terminar o jogo
sendPhi()	Enviar especificação do modelo de consistência *phi*
deactivate()	Remover um DataUnit do sistema
delayDeactivate()	deactivate() na próximo ronda do Cliente –> Servidor

Descrição da API de *callback's* de Cliente do sistema:

IVFCreckonApp	
Nome do Método	**Descrição**
callbackEnable()	Iniciar a fase de configuração do Jogo
callbackNewData()	Novos DataUnits registados por outros clientes
callbackNewInGameData()	Novos DataUnits registados por outros clientes durante a fase activa do Jogo
callbackError()	Indica que ocorreu um erro no sistema
callbackConnClosed()	Ligação com o servidor foi desactivada
callbackDeactivateAll()	DataUnit foi removido por um Cliente
callbackDeactivateOk()	DataUnit foi removido por este Cliente em concreto
callbackDisableAll()	Pedido de desactivação do Jogo feito por outro Cliente
callbackDisableOk()	Pedido de desactivação do Jogo feito com sucesso

Descrição da API de Servidor do Sistema:

VFCreckonServer	
Nome do Método	**Descrição**
open()	Iniciar o servidor local
close()	Desligar o servidor local
requestDirectConnection()	Criar uma ligação directa Cliente <–> Servidor

Descrição da API de *callback's* de Servidor do sistema:

VFCreckonServerListener	
Nome do Método	**Descrição**
callbackMessageIn()	Nova mensagem de Cliente
callbackClientPublished()	Cliente n ligou-se
callbackClientUnpublished()	Cliente n desligou-se

A.2 Diagramas de Classes do Sistema

Diagrama de Classes das principais classes, estruturas e camadas do sistema.

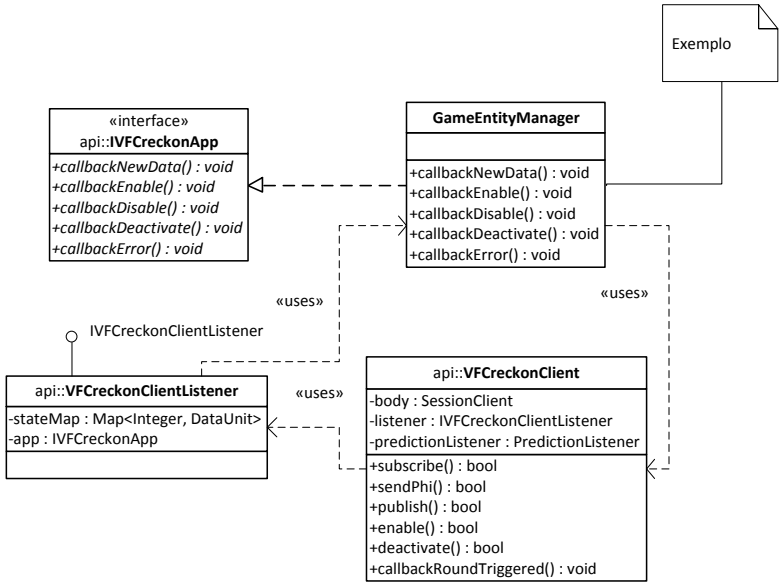

Figura A.1: Diagrama de classes da Camada de API do Cliente

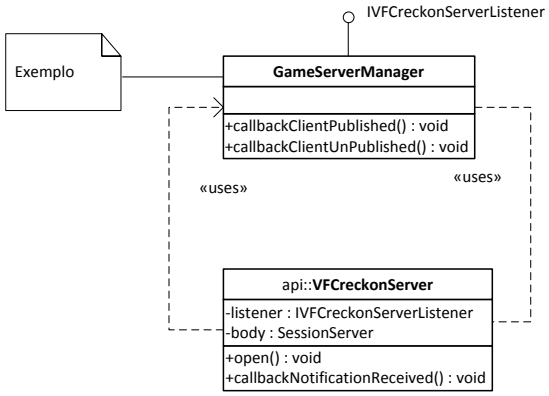

Figura A.2: Diagrama de classes da Camada de API do Servidor

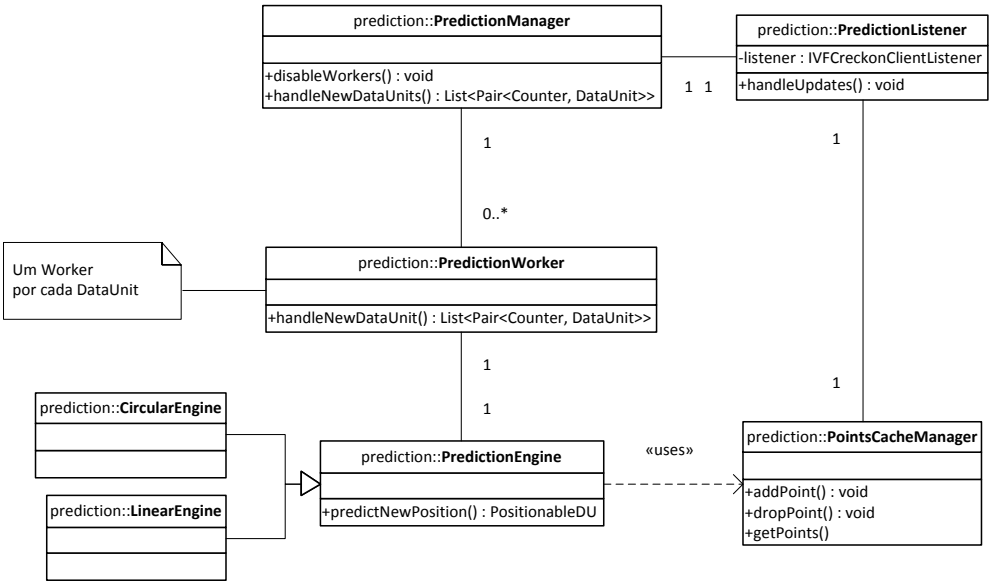

Figura A.3: Diagrama de classes do Módulo de DeadReckoning

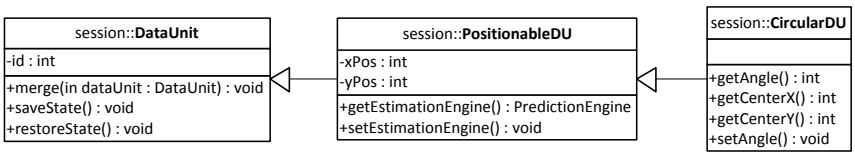

Figura A.4: Diagrama de classes das estruturas mais relevantes da Camada de Sessão

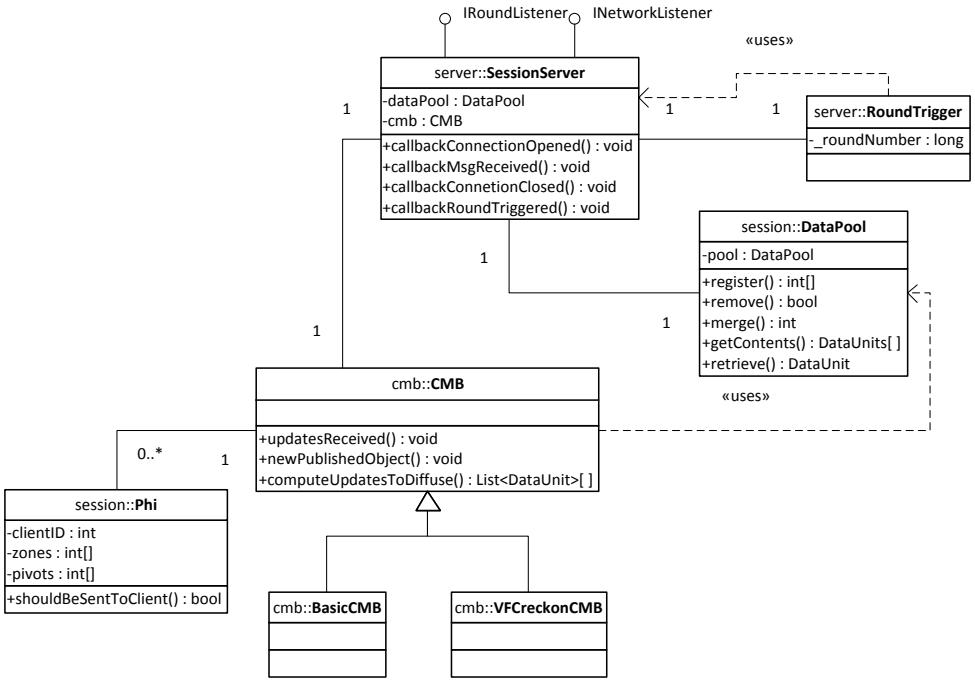

Figura A.5: Diagrama de classes da Camada de Sessão do Servidor

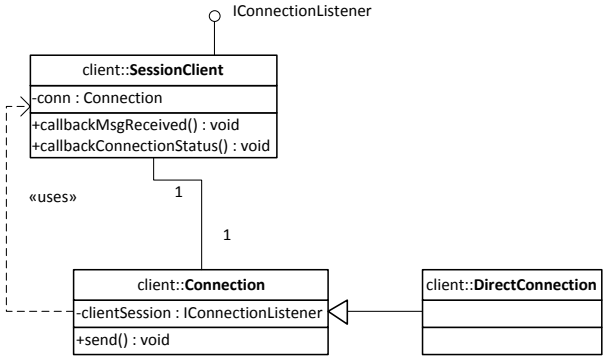

Figura A.6: Diagrama de classes da Camada de Comunicações do Cliente

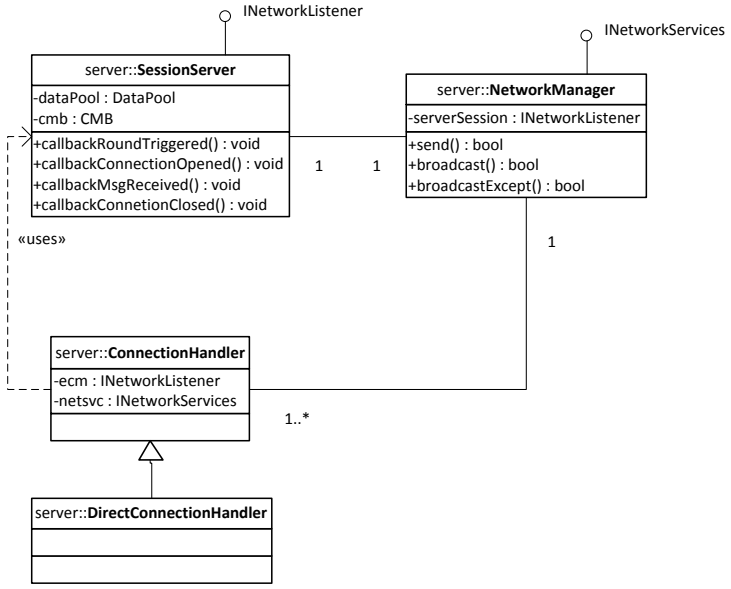

Figura A.7: Diagrama de classes da Camada de Comunicações do Servidor