**UNIVERSIDADE FEDERAL DE SANTA MARIA**
**CENTRO DE TECNOLOGIA**
**CURSO DE CIÊNCIA DA COMPUTAÇÃO**

# *IMPLEMENTAÇÃO DE DEAD RECKONING PARA MITIGAR O EFEITO DA LATÊNCIA EM JOGOS MULTIPLAYER*

**TRABALHO DE GRADUAÇÃO**

**Diogo Busanello Cerda**

**Santa Maria, RS, Brasil**

**2018**

# *IMPLEMENTAÇÃO DE DEAD RECKONING PARA MITIGAR O EFEITO DA LATÊNCIA EM JOGOS MULTIPLAYER*

**Diogo Busanello Cerda**

Trabalho de Graduação apresentado ao Curso de Ciência da Computação da Universidade Federal de Santa Maria (USFM, RS), como requisito parcial para a obtenção de grau de
**Bacharel em Ciências da Computação**

**Orientador: Prof. Dr. Raul Ceretta Nunes**

**Trabalho de Graduação Nº 452**

**Santa Maria, RS, Brasil**

**2018**

**Universidade Federal de Santa Maria**
**Centro de Tecnologia**
**Curso de Bacharelado em Ciência da Computação**

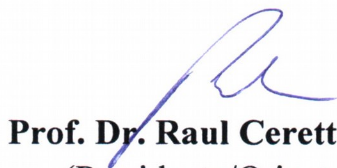
A Comissão Examinadora, abaixo assinada,
aprova o Trabalho de Graduação

**IMPLEMENTAÇÃO DE DEAD RECKONING PARA MITIGAR O EFEITO DA LATÊNCIA EM JOGOS MULTIPLAYER**

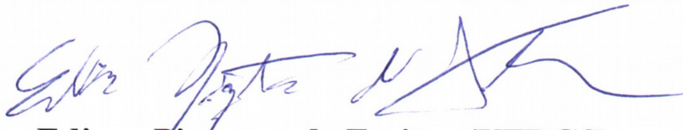
Elaborado por
**Diogo Busanello Cerda**

Como requisito parcial para obtenção do grau de
**Bacharel em Ciência da Computação**

**COMISSÃO EXAMINADORA:**



**Prof. Dr. Raul Ceretta Nunes**
(Presidente/Orientador)



**Prof. Dr. Edison Pignaton de Freitas (UFRGS)**

**Prof. Dr. Mateus Beck Rutzig (UFSM)**

Santa Maria, 13 de dezembro de 2018.

# RESUMO

Trabalho de Graduação
Curso de Ciência da Computação
Universidade Federal de Santa Maria

***IMPLEMENTAÇÃO DE DEAD RECKONING PARA MITIGAR O EFEITO DA LATÊNCIA EM JOGOS MULTIPLAYER***

Autor: Diogo Busanello Cerda
Orientador: Prof. Dr. Raul Ceretta Nunes
Local e data da defesa: Santa Maria, 13 de dezembro de 2018.

Torna-se cada vez mais comum a funcionalidade de *multiplayer* em jogos. Isso aumenta a complexidade do projeto de criação de jogos e apresenta novos desafios. Um dos desafios é caracterizado pelo fato da *Internet* não ser uma rede confiável. Um problema específico a ser solucionado em um projeto de jogo *multiplayer* diz respeito à latência. Sem tolerância à latência, um jogo *online* pode apresentar queda de qualidade da experiência dos jogadores. Existem soluções conhecidas para resolver problemas comuns de comunicação entre aplicações conectadas por rede. Este trabalho implementa uma das soluções para o problema da falibilidade da rede, o *dead reckoning*, cuja abordagem é a extrapolação da posição de entidades. O trabalho foi desenvolvido na linguagem C# e utiliza a *game engine Unity* para simular um jogo com funcionalidade *multiplayer* que permita a aplicação da solução de *dead reckoning* desenvolvida. Junto com a funcionalidade de *dead reckoning* foram desenvolvidos testes para analisar o desempenho da solução.

# ABSTRACT

Undergraduate Final Work
Graduation in Computer Science
Federal University of Santa Maria

**A DEAD RECKONING IMPLEMENTATION TO MITIGATE THE EFFECT OF LATENCY IN MULTIPLAYER GAMES**

Author: Diogo Busanello Cerda
Adviser: Prof. Dr. Raul Ceretta Nunes
Date and Location: December 13 2018, Santa Maria

Multiplayer functionality in games is becoming very common. In turn, it increases the complexity of game development projects and introduces new challenges. One of the challenges is characterized by the fact the internet is an unreliable network. A specific problem to be solved in a multiplayer game development project is lag. An online game with no lag tolerance can be detrimental to the players' experience quality. There are known solutions for common communication problems between networked applications. This paper implements dead reckoning, a solution to network fallibility by extrapolation of the game entities' position. The implementation was written in C# language and uses Unity game engine to simulate a game with multiplayer features so the developed solution can be applied to it. Along with the solution, tests were also implemented to help measure it's performance.

## Sumário

1 INTRODUÇÃO....8
1.1 Objetivo geral....9
1.2 Objetivos específicos....9
1.3 Estrutura do texto....10
2 A FUNDAMENTAÇÃO TEÓRICA....11
2.1 Sistemas distribuídos....11
2.1.1 Definições e características de sistemas distribuídos.... 11
2.1.2 Objetivos de um sistema distribuído.... 12
2.1.3 Modelos de Arquiteturas de Sistemas Distribuídos.... 14
2.1.4 Cliente-servidor.... 14
2.1.5 Sistemas Peer-to-Peer (P2P).... 15
2.1.3 TCP/IP e UDP.... 16
2.2 Conceitos básicos de programação de jogos....18
2.3 Programação de jogos *multiplayer*....19
2.4 Funcionamento de M*ultiplayer* no modelo P2P....21
2.5 Funcionamento de Multiplayer no modelo cliente-servidor....25
2.5.1 Predição na ponta do cliente (*Client-side prediction*).... 26
2.5.2 *Server time step*.... 29
2.5.3 Ações críticas.... 32
2.6 Estratégias de compensação de atraso....33
2.6.1 Filtro de percepção local (*Local Perception Filter*).... 33
2.6.2 *Time warp synchronization* e *trailing state synchronization*.... 33
2.6.3 *Rollback* aplicado a jogos.... 34
2.6.4 *Dead reckoning*.... 38
3 PROJETO CONCEITUAL....40
3.1 O projeto....40
3.1.1 Módulo controlador de ambiente de rede.... 41
3.1.2 Módulo de comunicação de rede.... 42
3.1.3 Módulo do modelo de *dead reckoning*.... 42
3.1.4 Módulo de lógica do jogo.... 43
4 IMPLEMENTAÇÃO....44

4.1 Ferramentas utilizadas....44
4.1.1 Unity 3D....44
4.1.2 *Microsoft Visual Studio*....45
4.1.3 C#....45
4.2 Desafios....45
4.3 Sobre a Unity....46
4.4 Sincronização de entidades....48
4.3 Funcionamento do dead reckoning....50
4.4 Gerenciamento de partidas e *matchmaking*....51
4.5 Definição de testes....52
4.6 Conceito e prática....53
5 RESULTADOS....55
6 CONCLUSÃO....58
6.1 Trabalhos futuros....58
REFERÊNCIAS....59

# 1 INTRODUÇÃO

Os jogos eletrônicos aumentaram em abrangência e complexidade ao longo dos anos, fato perceptível pelo crescente compromisso com detalhes, realismo e inovações na interatividade. No que tange à interatividade, chama-se de *multiplayer* (do inglês, multijogador) o modo de mais de uma pessoa interagir com o jogo ao mesmo tempo, sendo um componente importante da experiência em jogos eletrônicos. Seja pelo desafio de enfrentar alguém nas mesmas condições e ver quem se sobressai, seja por ser encorajador ter um companheiro para completar uma atividade. Assim, a interação entre pessoas por meio do jogo possui estímulo e apelo únicos.

Nesse contexto, com a consolidação da *Internet* como meio de comunicação, a limitação geográfica do *multiplayer* foi abolida com a tendência dos jogos se comunicarem enquanto instâncias em máquinas diferentes. Deste modo, a implementação da comunicação entre instâncias de um mesmo jogo se torna uma questão sensível, uma vez que existem características do jogo que necessitam ser consistentes de uma máquina para outra, tais como posições no mundo, atributos do personagem, entre outros. Já para os desenvolvedores de jogos, a opção de *multiplayer* pode ser um diferencial que agrega interesse de compra ao seu produto, existindo inclusive jogos que pressupõe a interação entre jogadores como uma de suas mecânicas base, os *Massive Multiplayer Online Games* (MMOG ou MMG).

Neste sentido, surge, então, a problemática de como desenvolver jogos com a funcionalidade *multiplayer online* (BARRON, 2001). Em sistemas distribuídos, um *middleware* é definido como uma camada abstrata à aplicação que realiza a comunicação entre as máquinas de acordo com protocolos predefinidos. Para se escolher a implementação desse *middleware*, também chamado de *netcode*, que melhor se encaixa às necessidades do jogo, leva-se em consideração a tecnologia disponível, bem como as limitações de projeto, dificuldade e tempo. Uma dessas limitações provém da arquitetura e funcionamento da *Internet*, em que pacotes que transportam informações entre máquinas podem sofrer atrasos ou se perder completamente, nunca chegando ao destinatário. Falhas na rede que causam atraso ou perda de mensagens podem gerar uma quebra na percepção do jogo como sendo em tempo real ou até mesmo perda de consistência, afetando a qualidade da experiência.

As necessidades de *multiplayer* variam conforme características do projeto, como, por exemplo, gênero de jogo ou plataforma. Existem jogos mais cadenciados como os de estratégia que conseguem tolerar centenas de milissegundos de atraso na rede, já jogos de luta ou esporte possuem

muita ênfase em reações rápidas e são considerados especialmente sensíveis a atrasos, necessitando de sincronização e atualização constantes das ações de todas as partes para manter consistência. A constante troca de mensagens necessária para isso torna o processo especialmente sensível a problemas na rede.

Sendo assim, muitos desafios existem para manter uma boa experiência *online*. O método de implementação de *multiplayer* mais comum é replicar a aplicação em cada máquina e mantê-las sincronizadas mudando seu estado em reação à operações iniciadas pelos usuários e pela passagem do tempo (XU e WAH, 2013). No entanto, essa abordagem possui desvantagens e não garante uma boa experiência ao jogador quando se fatora atrasos na rede.

Este trabalho tem como objetivo a implementação de uma técnica que lida com o problema da latência. Dentre abordagens conhecidas, o *dead reckoning* lida com a imprevisibilidade da rede minimizando o volume de mensagens de sincronização através de algoritmos de extrapolação do jogador remoto, baseado nas últimas informações recebidas. A fim de compreender seu funcionamento, define-se um protótipo de jogo que possua modo *multiplayer* e aplica-se nele a solução, ao final realizando testes para medir seu desempenho em poupar mensagens e contornar instabilidade.

## 1.1 Objetivo geral

O objetivo geral é o desenvolvimento de um protótipo de prova de conceito de funcionalidade *multiplayer* de um jogo de simulação de guerra com carros de combate. O foco é a implementação de um programa que simule uma partida via rede entre dois jogadores, permitindo a observação e análise do funcionamento da troca de mensagens entre as instâncias do jogo e o uso de técnicas de extrapolação nesse contexto.

## 1.2 Objetivos específicos

– Entender os principais cenários que exigem sincronização de ações no *multiplayer* de um jogo;

– Definir e implementar um protótipo de jogo e um *middleware* que agregue a ele funcionalidade *multiplayer*;

– Confirmar a eficácia da técnica de *dead reckoning*.

## 1.3 Estrutura do texto

Este texto está assim organizado: o capítulo dois delimita os objetivos do trabalho. O capítulo três descreve a fundamentação teórica necessária para o desenvolvimento do trabalho, abordando temas como sistemas distribuídos e programação de jogos. No capítulo quatro são descritas quais as bases do projeto teórico e ferramentas utilizadas. O capítulo cinco contém o desenvolvimento do projeto e dos testes, no capítulo 6 são reunidos os resultados, e no capítulo final encontra-se a conclusão e deliberações acerca dos resultados.

# 2 A FUNDAMENTAÇÃO TEÓRICA

## 2.1 Sistemas distribuídos

Um sistema distribuído é aquele que está compartilhado entre diversas máquinas, mas, tanto para a camada da aplicação que nele está envolvida quanto para o usuário, aparenta ser um sistema de um único computador. Pode-se considerar o sistema *multiplayer* a ser desenvolvido neste projeto como um sistema distribuído. Seus principais aspectos são abordados neste capítulo.

### 2.1.1 Definições e características de sistemas distribuídos

Em contraste aos sistemas centralizados que dominavam o mercado de sistemas surgiu junto à rede um novo meio de aplicar os mesmos: os sistemas distribuídos. Por definição: "*Um sistema distribuído é uma coleção de computadores independentes que parecem aos seus usuários como um sistema único e coerente.*" (TANENBAUM; STEEN, 2002, p. 2)

Uma importante característica de um sistema distribuído é a diferença entre os vários computadores que o compõe e a maneira como a comunicação entre eles é abstraída para o usuário. Outra importante característica é de que usuários e aplicações podem interagir com um sistema distribuído de maneira consistente e uniforme, independente de onde ou quando a interação acontece (TANENBAUM; STEEN, 2002).

Um sistema distribuído deve ser de fácil expansão e escalabilidade. Essa característica é consequência do sistema ser composto por independentes computadores, ou seja, o sistema deve ser planejado para que o número de computadores seja algo variável visto que podem acontecer adições e remoções de membros do grupo de máquinas e também falhas nas máquinas que compõe o sistema. A reposição e conserto de partes do sistema não devem interferir no seu funcionamento.

Para que sistemas distribuídos tenham suporte a um grupo de computadores e sistemas operacionais heterogêneos, esses se organizam em camadas de aplicação que possuem tarefas distintas. A camada mais baixa é aquela referente ao sistema operacional local. A camada que efetiva a comunicação e sincronização de maneira abstrata para a aplicação é a camada

conhecida como *middleware*. A camada mais próxima ao usuário é aquela em que a aplicação distribuída se encontra. A figura demonstra a organização dessas camadas de maneira gráfica.

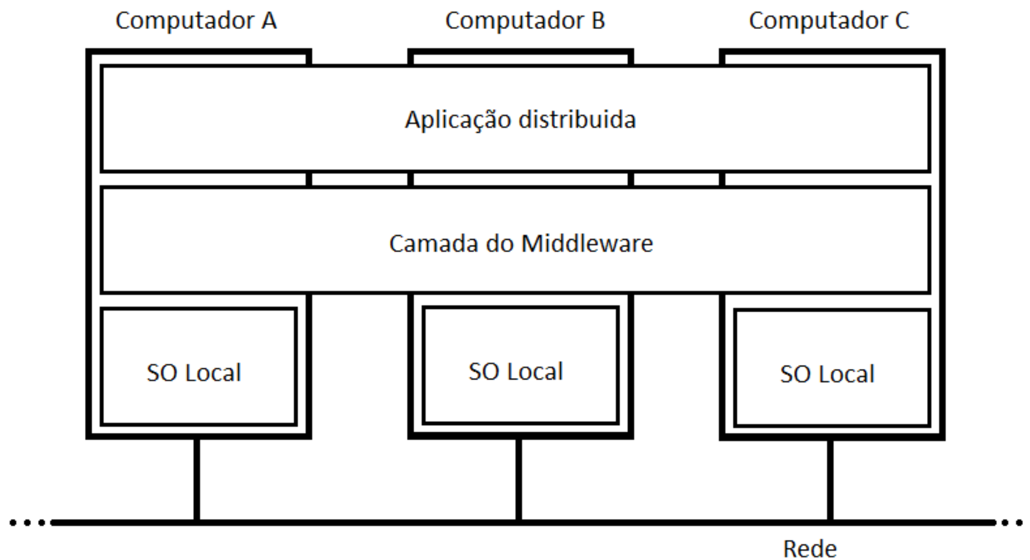



Figura 1: Organização de um sistema distribuído em camadas (COULOURIS, DOLLIMORE e KINDBERG, 2007).

### 2.1.2 Objetivos de um sistema distribuído

A capacidade de se construir um sistema distribuído não define sua utilidade. Logo, é preciso definir quais os objetivos de se construir um sistema distribuído para comprovar a necessidade do mesmo no ambiente em que ele será inserido.

O objetivo principal de um sistema distribuído é facilitar a conexão entre usuários e recursos. Esses recursos podem ser virtualmente qualquer coisa: os exemplos clássicos são impressoras, computadores, espaços para armazenamento, dados, arquivos e páginas Web. O motivo mais claro do porque a necessidade de conexão entre usuários e recursos é pelo motivo econômico, pois é mais barato manter menos cópias dos recursos sendo que essas são acessadas por vários usuários, como, por exemplo, uma impressora em um andar de um prédio comercial.

Conectar usuários e recursos também facilita que usuários trabalhem colaborativamente e troquem informação. Por exemplo, sistemas que trabalham com versionamento de dados (SVN, GIT, etc.) são uteis no trabalho colaborativo na criação de software.

Outro objetivo muito comum na construção de um sistema distribuído é esconder o fato de que os recursos estão distribuídos fisicamente em múltiplos computadores. Chama-se transparência a característica de um sistema distribuído de se apresentar ao usuário e a aplicativos como um sistema de um único computador. Existem diferentes tipos de transparência para diferentes objetivos. Pode-se qualificá-las como a tabela abaixo mostra:

Tabela 1 – Diferentes formas de transparência em um sistema distribuído

| Transparência | Descrição |
|---|---|
| Acesso | Esconde as diferenças de como os dados estão representados e como eles são acessados. |
| Localização | Esconde onde os recursos estão localizados. |
| Migração | Esconde o fato de que um recurso talvez tenha sido movido para outra localização. |
| Realocação | Esconde o fato de que um recurso talvez precise ser movido para outra localização enquanto é usado. |
| Replicação | Esconde o fato de um recurso ser replicado. |
| Concorrência | Esconde o fato de que talvez um recurso seja compartilhado por alguns usuários competitivos entre si. |
| Falhas | Esconde a falha e a recuperação de um recurso. |
| Persistência | Esconde se um (software) recurso está em memória ou em disco. |

(COULOURIS, DOLLIMORE e KINDBERG, 2007)

Existe ainda o objetivo de abertura para um sistema distribuído. Um sistema distribuído aberto é aquele que fornece serviços de acordo com regras padronizadas que descrevem a sintaxe e a semântica desses serviços. Essas formalizações de como se dá o serviço também podem ser chamadas de protocolos. Em geral nesse tipo de sistema os serviços que ele provém são especificados em interfaces, as quais são descritas em Linguagem de Definição de Interface (IDL). As definições descritas em IDLs quase sempre apenas apresentam a sintaxe dos serviços. Em outras palavras, elas especificam os nomes de funções que podem ser acessadas por usuários e aplicativos para que o serviço do sistema seja cumprido. Junto aos nomes das funções segue os parâmetros dessas, os valores de retorno, suas possíveis exceções entre outras características. A parte semântica dos serviços normalmente não é descrita ou é simplificada e informal.

### 2.1.3 Modelos de Arquiteturas de Sistemas Distribuídos

A arquitetura de um sistema é a sua estrutura em termos de componentes especificados separadamente. O objetivo de uma arquitetura é de que ela atenda as demandas que um sistema tem e pode vir a ter. As demandas de um sistema normalmente são referentes à sua confiabilidade, gerenciabilidade, adaptabilidade e rentabilidade. (TANENBAUM; STEEN, 2002)

Existem vários padrões amplamente usados para a divisão de tarefas em um sistema distribuído que têm um impacto importante sobre o desempenho e a eficiência do sistema resultante, assim como existem em sistemas distribuídos várias arquiteturas para as diversas situações e necessidades do projeto para qual esse sistema está sendo construído. As seções a seguir apresentam as arquiteturas mais comuns de sistemas distribuídos.

### 2.1.4 Cliente-servidor

Essa é a arquitetura mais comum em sistemas distribuídos. Ele é baseado em clientes que hospedam interfaces do sistema, que por sua vez tem função de acessar o servidor contendo os serviços que devem resolver invocações dos usuários. Toda necessidade de um cliente em acessar um serviço do servidor gera uma invocação do servidor, que por sua vez gera outra mensagem em resposta para aquele devido cliente.

O servidor normalmente é o detentor da maioria dos recursos em um sistema ou, pelo menos, das indicações de onde os recursos estão localizados. Isso ocorre para que todos os clientes tenham um consenso de onde procurar os diversos recursos disponibilizados no sistema.

O servidor numa arquitetura cliente-servidor torna-se um problemático ponto fraco na tolerância de falhas por ser o único sanador de invocações e detentor da maioria dos recursos. Um servidor falho significa um sistema todo falho nessa arquitetura. Assim, apesar de ser um sistema fácil de ser implementado, não é inerentemente tolerante a falhas.

Esse tipo de sistema também pode ser hierarquizado, podendo haver requisições de um servidor para outros servidores, tornando assim o servidor de um sistema o cliente em outro. A figura a seguir demonstra a ideia de hierarquia entre servidor e invocações de clientes.

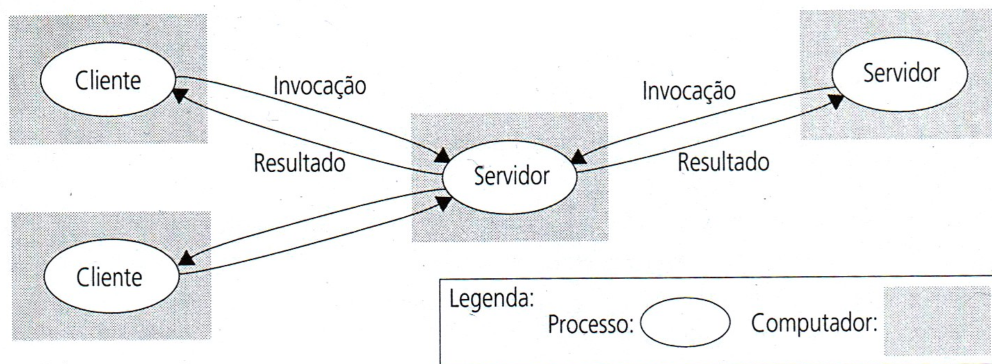



Figura 2: Organização de um sistema cliente-servidor (COULOURIS, DOLLIMORE e KINDBERG, 2007).

### 2.1.5 Sistemas Peer-to-Peer (P2P)

Diferente do sistema centralizado Cliente-servidor, nessa arquitetura todos os processos envolvidos em uma tarefa ou atividade desempenham funções semelhantes, interagindo cooperativamente como pares (*peers*), sem distinção entre processos clientes e servidores, nem entre computadores em que são executados.

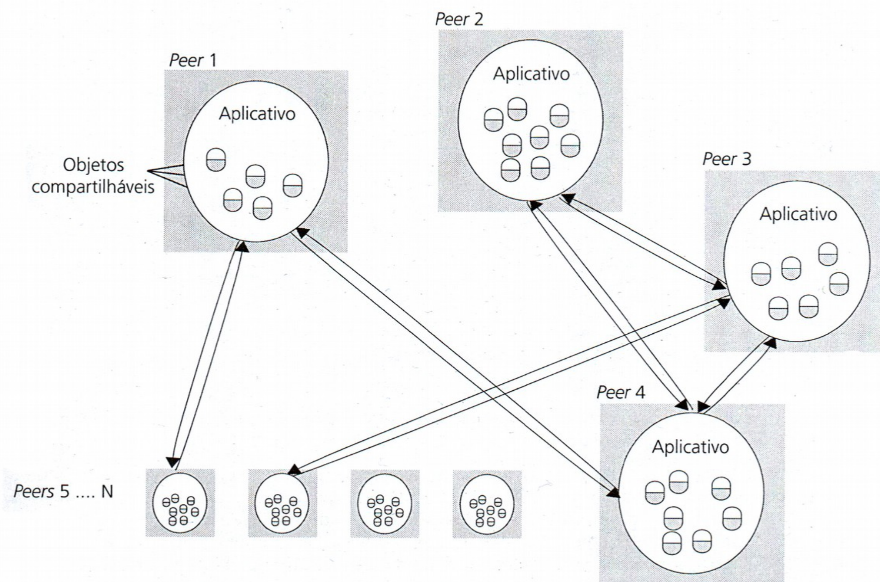



Figura 3: Organização de um sistema *peer-to-peer* (COULOURIS, DOLLIMORE e KINDBERG, 2007).

P2P não é uma ideia nova: a ARPANET, princípio da internet contemporânea, foi projetada para ser um sistema P2P que conectava certas universidades e instalações do governo dos EUA. Com expansão da ARPANET e o surgimento de vários serviços baseados em protocolos cliente-servidor (Telecine, e-mail entre outros), sistemas P2P não eram tão atrativos, visto o custo e a inviabilidade tecnológica de fazer com que os todos os computadores clientes tivessem hardware e banda suficiente para dividirem o pesar do processamento dos serviços prestados pelos servidores. O ressurgimento dos sistemas P2P foi uma consequência de acréscimo no número e na largura das conexões, assim como na capacidade do hardware dos seus diversos usuários.

Um exemplo de uso de P2P para comunicação foi o sistema *multiplayer* do jogo *Doom*, o qual independia de quaisquer servidores interagindo com o jogo para que os jogadores pudessem efetivar partidas na rede (SHIRKY, 2000). Bastava que os jogadores identificassem um *host* (um dos jogadores responsável por orientar o grupo no início da partida) que a partir de então todos os jogadores se tornavam membros igualitários para o grupo como clientes.

### 2.1.3 TCP/IP e UDP

Para efetivar a troca de dados entre um sistema distribuído deve ser definido certo protocolo de comunicação. Na atualidade, os protocolos usados hoje são formados por diversos outros protocolos com propósitos específicos na tarefa de comunicação entre computadores. Os diversos protocolos existentes para formar protocolos completos de comunicação podem ser classificados como estando em alguma das cinco camadas da pilha de protocolos TCP/IP. O nome dessa pilha de protocolos vem de *Transmission Control Protocol* (TCP) e de *Internet Protocol* (IP) os primeiros dois protocolos de comunicação a serem definidos (COULOURIS, DOLLIMORE e KINDBERG, 2007).

O conjunto de protocolos TCP/IP abrange as cinco diferentes camadas que devem trabalhar independentes uma das outras, sendo:

- Camada de aplicação: a camada mais próxima ao usuário. Ela deve implementar a interface com o programa que a implementa. Os processos que executam nessa camada são específicos da aplicação. Praticamente se recebem da aplicação os dados no formato interno ao código e se codificam eles (ou decodificam no retorno) em dados do formato definido no protocolo.

- Camada de transporte: essa é a camada responsável pelo dado conseguir alcançar sem erros seu destino, característica também conhecida como confiabilidade, e pela propriedade dos dados chegarem na ordem correta ao seu destino, propriedade também chamada de integridade. Nem todos os protocolos dessa camada se comprometem com total confiabilidade ou integridade. Visando uma menor quantidade de dados a ser repassado o protocolo que define as regras de transporte pode se comprometer apenas com uma fraca confiabilidade e com nenhuma garantia de integridade como no caso do protocolo UDP.
- Camada de rede: a camada que se deve direcionar através de um protocolo de endereçamento os pacotes de dados através da rede. O exemplo mais conhecido dessa camada é o protocolo de endereçamento IP, hoje difundido por toda a *Internet*.
- Camada de enlace: essa camada define protocolos para adição de *headers* aos pacotes que serão despachados pela rede física. Esses protocolos podem ser implementados tanto em nível de *software* (*device driver*) quanto em de *hardware* nas placas de rede e outros dispositivos que compõem uma rede. A responsabilidade dessa camada é de endereçamento, roteamento e controle de envio e recepção dos pacotes que vão circular pela rede.
- Camada física: essa camada trata dos protocolos de deslocamento físico de ondas eletromagnéticas ou físicas para a troca de informações.

O projeto de um sistema normalmente não é dependente dos protocolos de comunicação na qual essa deve funcionar, como é o caso do projeto desenvolvido nesse trabalho. Isso ocorre pelo fato dos protocolos estarem diferenciados por camadas que trabalham independentes da implementação das outras. Mas para a implementação do mesmo, foram definidos os protocolos da camada de aplicativo e camada de rede. As outras camadas são dependentes da plataforma e da rede em que o sistema estiver sendo executado.

O Protocolo UDP é um protocolo não confiável descrito pela RFC 768 (POSTEL, 1980). Esse protocolo permite que o aplicativo encapsule seus pacotes IPv4 ou IPv6 para que sejam enviados ao seu destino. Visto sua simplicidade e sua falta de confiabilidade quem o implementa tem a opção de aplicar diversos algoritmos de estruturas de controle como *timeouts*, retransmissões, *acknowlegments*, controle de fluxo, etc (TANENBAUM; STEEN, 2002).

## 2.2 Conceitos básicos de programação de jogos

Jogos eletrônicos podem ser assim classificados quando são jogos de computador, jogos para console ou dispositivos móveis. Um jogo de computador é um aplicativo como qualquer outro programa. Jogos para plataformas que não sejam o computador também são aplicativos, com a diferença de conter características específicas para a plataforma que executa a aplicação. O que diferencia um jogo de uma aplicação são algumas características comuns a todos os jogos.

Um jogo eletrônico deve ser executado em tempo real. Ou seja, sua execução deve passar noção de passagem de tempo e movimento. Essa característica não se compromete com a interatividade do usuário, possibilitando inclusive que a execução de um jogo seja assíncrona à entrada do usuário. A noção de movimento de um jogo está estritamente ligada à taxa de atualizações da tela. Essa taxa representa quantas vezes por segundo ela é redesenhada para dar a impressão de movimento dos objetos em jogos. Ela também é conhecida como FPS (*frames per second*).

As taxas de FPS mais comuns são as do formato NTSC (*National Television Standards Committee*) cerca de 30 FPS, do formato PAL (*Phase Alternation by Line*) com aproximadamente 24 FPS e a de muitos monitores de computadores modernos de 60 FPS a 120 FPS (HARRIS, 2000).

Um jogo também deve conter técnicas de computação gráfica (CG) em tempo real. Para aplicação das técnicas são necessárias APIs gráficas como o OpenGL (*Open Graphics Library, OpenGL Wikipedia Article*, 2014). Essas APIs proporcionam ferramentas para a criação de ambientes ideais para a renderização rápida das cenas dos jogos. Para se conseguir utilizar tais ferramentas é necessária uma placa de vídeo aceleradora.

Todo e qualquer jogo eletrônico também deve conter algum tipo de interface com o usuário. No mínimo uma interface de saída visual, comumente um monitor ou tela semelhante, e no mínimo uma interface de entrada, como teclado e mouse para a maioria dos jogos de computador, ou também controles, microfones, manches de simulação, luvas e capacetes de realidade virtual, entre outros dispositivos.

Integrando esses conceitos básicos de como se define um jogo, o cerne de sua produção se centraliza na programação computacional. A programação de jogos se dá em múltiplas linguagens de programação e diversas plataformas. Existem hoje diversas ferramentas, bibliotecas e *engines* (*Game Engine Wikipedia Article*, 2016) para o auxílio dos programadores de jogos, visto a complexidade na sua produção.

## 2.3 Programação de jogos *multiplayer*

O primeiro jogo de que se tem noticia a ser jogado online foi criado em 1969 por Rick Blomme, estudante do MIT em Massachusetts, EUA. O jogo era do estilo *Bulletin Board System* (BBS) onde um administrador recebia diariamente as jogadas de jogos de tabuleiros de diversos estudantes pelo sistema da universidade e respondia com o resultado das jogadas no tabuleiro, como se tivessem sido feitas simultaneamente. Apesar de terem sido criados antes, não foram os BBS que deram origem à série de jogos que hoje se espalham como os mais diversos jogos *online*. Os jogos *Multi-User Dungeon* (MUD), populares nas universidades da Inglaterra na década de 80, tinham um sistema *multiplayer* muito mais parecido com o que vemos nos jogos hoje em dia. Esses sistemas eram baseados em múltiplas entradas de texto em um servidor. Os textos descreviam pequenas ações que os jogadores pretendiam efetivar como os seus personagens do jogo. Dependendo dos textos inseridos pelos usuários os servidores respondiam propriamente com outro texto que descrevia os eventos desencadeados pela ação executada do jogador (BARRON, 2001).

Na atualidade, um grande número de jogos se comunica com outros pela *Internet* ou pela rede. Os jogos vão de partidas de dois jogadores a jogos feitos para suportar milhares de jogadores *online* simultaneamente como no caso dos jogos *Massive Multiplayer Online* (MMO).

A ideia mais intuitiva para definir uma comunicação entre instâncias de jogos é a de fazer com que os diferentes aplicativos troquem todas as informações referentes às execuções do atual *frame* na mesma taxa que o FPS de um jogo. Em um caso ideal de redes e de projetos sem limitações, essa sempre seria a melhor estratégia para implementação de um sistema *multiplayer*. Porém, essas limitações existem e são definidas tanto pela largura da banda dos jogadores, capacidade dos hardwares dos usuários e dos servidores, quanto pelas restrições do projeto.

Visto isso, torna-se praticamente impossível a criação de um jogo que mantenha uma taxa aceitável de quadros por segundos em uma partida *multiplayer* de muitos jogadores nesses moldes. O sistema *multiplayer* de um jogo deve ser um sistema de troca de mensagens em frequência assíncrona à de FPS e com mensagens de conteúdo de extrema importância, ou

seja, apenas o necessário para manter a percepção para os jogadores de que o jogo se mantém síncrono.

A maneira com que jogos se comunicam online não está restrita a um tipo de protocolo, de modo que existem os mais diferentes sistemas distribuídos para a comunicação entre jogos. Estes podem se comunicar com auxílio de um servidor, ou independes de um com troca de mensagens entre os jogadores (P2P). Jogos *multiplayer* podem implementar sistemas tolerantes a falhas dos membros, ou bloqueantes. Podem se comunicar com protocolos mais seguros de troca de dados como TCP ou protocolos mais rápidos como UDP. Todo tipo de configuração do sistema distribuído de um jogo é definido segundo suas necessidades e peculiaridades.

Num sistema de comunicação implementado em um jogo *multiplayer* não é usado apenas a sua confiabilidade em manter a informação coerente como critério de eficiência. O desempenho em relação à velocidade de troca de mensagens também é outro critério muito comum a ser avaliado. Para isso, se utiliza o termo *lag* (latência) para se rotular o tempo de atraso no envio e recebimento de uma resposta para certo comando enviado. A unidade comum a ser usada para tal medição de tempo é o milissegundo. Uma forma comumente utilizada para medir o *lag* em sistemas *multiplayer* é o programa *ping* (PING artigo da Wikipedia).

A tolerância ao *lag* em um jogo depende do tipo de jogo que está em execução. Em caso de jogos baseados em turnos ou de baixa ação, um *lag* alto não é considerado um problema, pois na execução do jogo a demora de resposta dos outros jogadores ou das próprias ações tomadas não atrapalha as condições de jogo. Já em jogo de resposta rápida, como no caso dos jogos do tipo luta ou esporte, o *lag* tem reflexo direto nas condições de jogo dos diversos jogadores. Nesses casos o *lag* deve ser mantido o menor possível. Isso não depende só da estrutura do jogo, mas também de toda estrutura de redes e dos computadores que a integram, pois por mais bem planejado que seja um sistema ele ainda está sujeito a problemas como falhas nas máquinas ou falta de banda na rede para os dados. Logo, para cada tipo de jogo existem necessidades únicas para a comunicação em rede. São essas questões que devem ser trabalhadas em cada protocolo para que a troca de mensagens seja eficiente e supra as necessidades do projeto em comunicação.

Os protocolos devem fazer o máximo possível para manter as propriedades de legalidade e simultaneidade do jogo. Manter a legalidade requer que a latência de uma instância de dados entre dois processos remotos permaneça constante. Por exemplo, no caso de um jogo de corrida, a legalidade é respeitada se o tempo que um carro leva para variar entre duas posições consecutivas é o mesmo para ambos jogadores, ou seja, a velocidade do carro é a mesma nas duas simulações. Já a simultaneidade diz que o tempo físico entre as consequências de duas atualizações consecutivas deve ser o mesmo para todos os usuários. No mesmo exemplo do jogo de corrida, a simultaneidade

é mantida se, havendo uma colisão entre dois carros, eles ocupam o mesmo lugar em um determinado momento para todos os jogadores. Se ambas as propriedades são respeitadas, diz-se que o sincronismo do *multiplayer* possui consistência perceptiva (KHAN, CHABRIDON e BEUGNARD, 2007).

> *The difficulty with finding out-of-sync errors is that very subtle differences would multiply over time. A deer slightly out of alignment when the random map was created would forage slightly differently – and minutes later a villager would path a tiny bit off, or miss with his spear and take home no meat. So what showed up as a checksum difference as different food amounts had a cause that was sometimes puzzling to trace back to the original cause* (TERRANO; BETTNER, 2001, p. 1).

## 2.4 Funcionamento de M*ultiplayer* no modelo P2P

Supondo um jogador A local e um jogador B remoto em um jogo dividido em etapas formadas por frações de tempo arbitrárias, em uma etapa jogador A realiza uma ação e na seguinte B responde a essa ação. A latência e variações da rede adicionam intervalos de espera no tempo de decisão para determinarmos o resultado da ação, criando também realidades diferentes para ambos jogadores, de forma que uma ação de A começará mais tarde na realidade de B. Havendo uma possível resposta por parte de B, como bloquear um ataque, essa resposta também aparecerá mais tarde na realidade de A.

O modo mais comum de colocar um jogo online é executar simulações em cada máquina e mantê-las sincronizadas através da abordagem de atraso local, que consiste em computar localmente a ação de um jogador local A mais tarde do que ela é enviada ao jogador remoto B e percebida como uma entrada no jogo, de forma que as duas realidades possam se sobrepor para fins de tomada de decisão por parte dos jogadores. Alguns jogos utilizam um valor mínimo de atraso independente da possibilidade da latência da rede ser menor, para que todos os jogadores fiquem em pé de igualdade e a experiência seja mais robusta, como por exemplo o jogo de estratégia *Warcraft 3* cujo valor de atraso local mínimo é 250ms (XU e WAH, 2013).

Se o estado das simulações é definido apenas por uma entrada, as ações dos jogadores, garantir que a mesma entrada será repassada em tempo hábil para ambas partes garante sincronia e que o resultado será o mesmo em cada máquina. Uma das vantagens desse método é que a lógica do jogo e do *multiplayer* ficam separadas, os desenvolvedores do jogo não

precisam se preocupar com detalhes de funcionamento da rede, e qualquer jogo poderia ganhar uma versão *multiplayer* mesmo anos depois de lançado.

Infelizmente ele também carrega uma grande desvantagem: esses jogos costumam ser planejados para amostrar a entrada antes de cada atualização da simulação. Um *frame* de simulação no jogo não pode ser executado até que os comandos de jogadores remotos sejam recebidos.

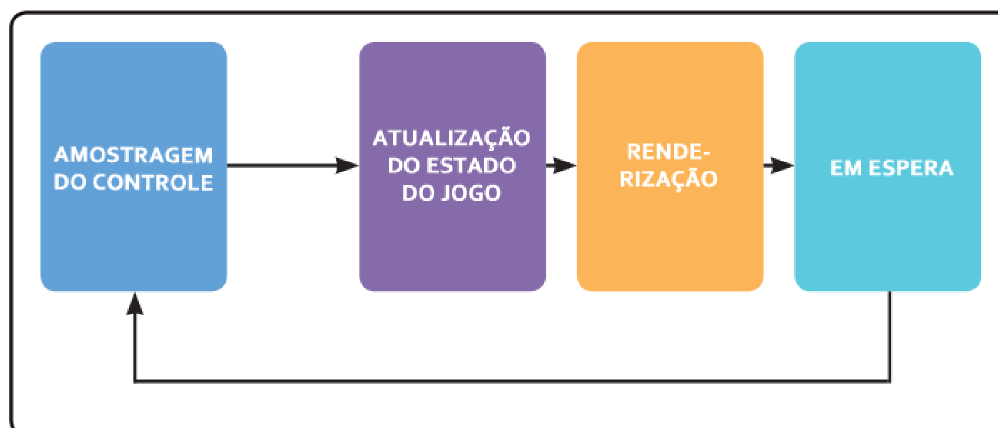



Figura 4: ciclo padrão de um jogo interativo (CANNON, 2012).

O ciclo apresentado na figura 3 começa com uma amostragem dos controles do jogador que é analisada pelo controlador de simulação para gerar o próximo quadro do jogo, renderizando o resultado. Muitos jogos atualizam o ciclo a uma taxa constante de 60 Hz, sendo necessário um estado de espera antes da próxima amostragem. Existem variações deste ciclo onde, por exemplo, um jogo pode amostrar e atualizar seu estado a 30 Hz enquanto realiza a renderização a 120 Hz com uma interpolação entre os últimos dois estados (CANNON, 2012).

Passados os protocolos para conexão inicial entre os jogadores, a única preocupação é trocar seus comandos pela rede. Os comandos são transmitidos em pacotes que possuem uma estrutura padrão, com espaços reservados para endereçamento, teste de integridade e dados. E é aqui que começam as dificuldades para manter a sincronização devido ao funcionamento da internet, pois:

– pacotes levam tempos diferentes para chegar ao destino e não há garantia de ordem;

– pacotes podem ser perdidos no caminho;

– pacotes podem ter seus dados corrompidos;

– as máquinas podem estar executando o jogo a velocidades diferentes;

– uma máquina pode ficar ocupada com outras tarefas e atrasar *frames* do jogo.

Existem formas de contornar alguns desses problemas. O modo mais simples de manter a sincronização de dois jogadores é atrasar o processamento da entrada de ambos por um número

determinado de *frames* suficientes para que a troca de pacotes seja realizada, como mostra a figura 4. Segundo Mauve (2012), uma das formas para calcular esse número é:

Atraso = ArredondarPraCimaCima(ViagemCompleta + Constante / (2 * DuraçãoDeFrame)) (1)

A constante serve como margem de erro de segurança, uma vez que a medição de tempo em computadores é relativamente imperfeita. Como só precisamos do tempo da viagem de ida do pacote dividimos o valor por dois, e para converter o valor resultante para número de *frames* dividimos pelo tempo de duração de um *frame*, que no caso de 60 FPS equivale a 16,67ms.

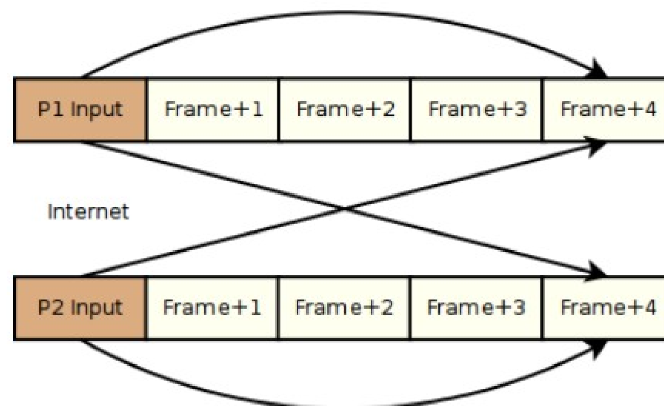



Figura 5: exemplo do funcionamento de atraso local de 4 *frames* (MAUVE, 2012).

Quando a sincronia está atrelada a um *frame* específico, no caso de perda de pacotes por algum engarrafamento na rede é preciso interromper a execução e esperar pela chegada do pacote, aproveitando-se para mandar um pedido de reenvio. No entanto, existem formas de se prevenir contra isso no modelo de atraso local. Os comandos de um jogador por *frame* geram uma quantidade pequena de dados, sendo possível armazenar o equivalente a vários *frames* de entrada em um único pacote. Isso garante que não seja necessário esperar um ciclo completo de troca de mensagens para receber o reenvio do pacote em caso de falha, amenizando os efeitos sentidos pelo jogador. Em conjunto com essa prática, pode-se escolher adicionar um *frame* extra de atraso para evitar interrupções na execução. Como é possível ver na figura 5, o pacote enviado no *frame*+1 que carregava o comando a ser processado na máquina remota no

*frame*+3 foi perdido, porém a informação duplicada contida no pacote do próximo frame garante que os comandos referentes ao *frame*+3 chegarão a tempo de serem processados, sem a necessidade de interrupção e pedido de reenvio do pacote perdido (MAUVE, 2012).

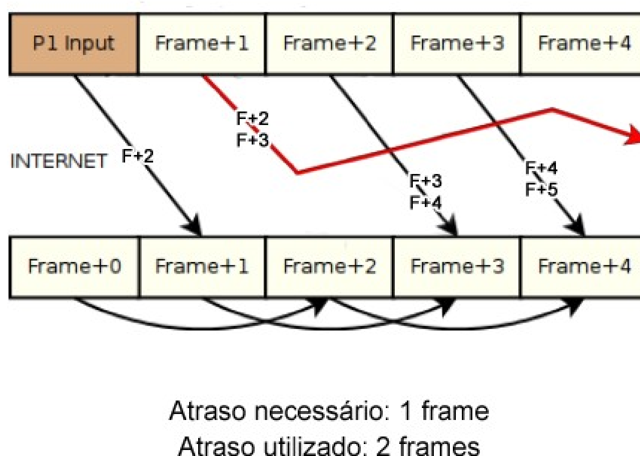



Figura 6: *frame* adicional de atraso e múltiplas entradas por pacote trabalhando em conjunto (MAUVE, 2012).

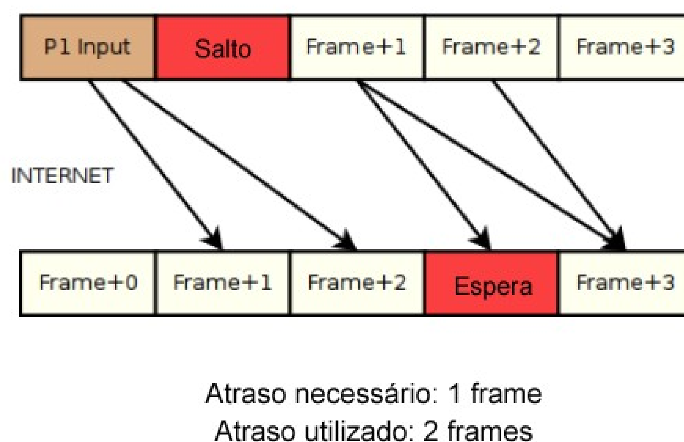



Figura 7: tratamento de queda de *frames* (MAUVE, 2012).

É importante ressaltar que a intenção do sistema é manter a sincronia a todo custo, a despeito de diferenças de performance entre computadores. Se uma instância está executando mais lenta que a outra e apresenta queda de *frames*, as quedas precisam ser refletidas no outro lado, como mostra a figura 6, ou haverá falha de sincronia e desvios do comportamento esperado. Infelizmente isso anula o benefício do *frame* adicional, mas pode ser tratado desde que se conheça o atraso

necessário e o valor do atraso vigente. Quando um comando é recebido se determina se ele corresponde ao atraso mínimo para o *frame* atual, e se ele corresponder a um atraso menor, pula-se o *frame* para manter a sincronia entre as plataformas.

Por meio desse modelo simples é possível compensar a perda de pacotes mantendo sincronia completa entre as partes. Reajustar o atraso mínimo necessário enviando requisições de *ping* de tempos em tempos também é uma boa ideia (MAUVE, 2012).

## 2.5 Funcionamento de Multiplayer no modelo cliente-servidor

A noção intuitiva de um jogo em arquitetura C/S é a de que o servidor seria uma figura autoritativa detentora de todos os recursos do jogo e o cliente apresentaria uma interface simples que enviaria apenas seus comandos para o servidor. Por sua vez, o servidor processaria o resultado das ações realizadas no lado do cliente e retornaria uma atualização correspondente ao seu novo estado, e também os estados de todos os outros usuários que o cercam. Desse modo, não haveria necessidade de garantir determinismo entre todas as máquinas pois o jogo existe somente no servidor e cada cliente agiria como um terminal burro representando uma aproximação do jogo que se desenrola no servidor (FIEDLER). Apesar de simples e prático, a limitação desse modelo é explicitada na figura 9: o tempo de comunicação entre o cliente e o servidor é o tempo que o jogador precisa esperar até sua ação ser refletida localmente.

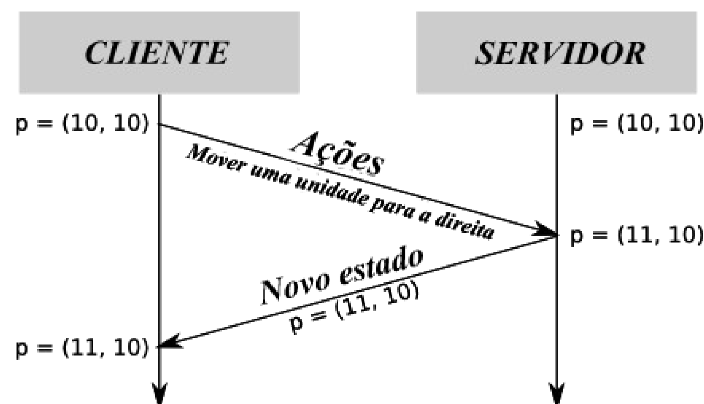



Figura 9: linha do tempo da troca de mensagens referente ao processamento da ação do usuário (GAMBETTA, 2012).

Em outras palavras, o atraso no jogo é o dobro da latência entre cliente e servidor. Em uma rede LAN cuja transmissão de pacotes pode ser considerada instantânea isso não seria um problema, porém em um ambiente de rede como a internet, em que a latência pode chegar a décimos de segundo, um jogo pode aparentar ser irresponsivo.

### 2.5.1 Predição na ponta do cliente (*Client-side prediction*)

Sendo o mundo do jogo determinístico o suficiente, é possível assumir, na maior parte do tempo, que a entrada recebida pelo servidor será válida e o estado do jogo será modificado de forma previsível. Isso significa que se um personagem de um jogo está na posição (10,10) e a seta para a direita é pressionada, ele acabará na posição (11, 10) dentro do mundo do jogo. Supondo que o atraso seja de 100 milissegundos e a animação de movimento do personagem de uma unidade de espaço para outra leva outros 100 milissegundos, na implementação ingênua a ação seria completada em 200 milissegundos. Sob a suposição do determinismo, o cliente tem a capacidade de prever corretamente qual será o estado do jogo após sua entrada ser processada. Assim, em vez de enviar sua entrada e esperar receber o novo estado do jogo para renderizá-lo, o cliente pode começar a renderizar o resultado da entrada como se ela tivesse sido válida enquanto espera o servidor retornar o estado de jogo "correto" – que, na maioria dos casos, concordará com o estado calculado localmente.

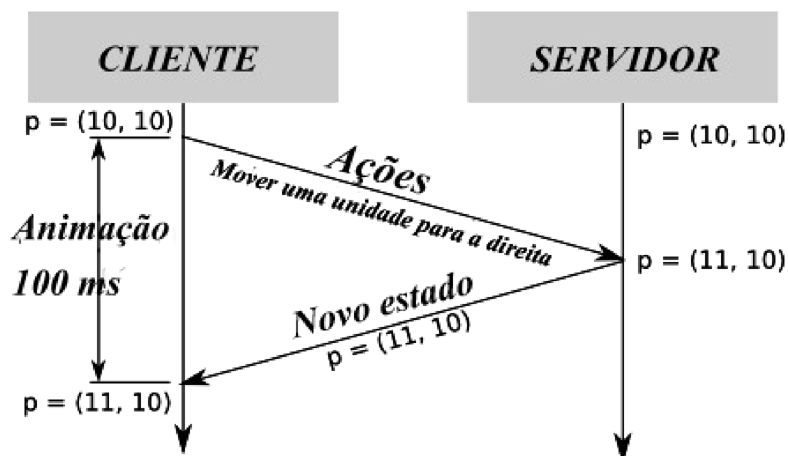



Figura 10: a animação acontece enquanto o servidor confirma a ação (GAMBETTA, 2012).

Com essa alteração se elimina o atraso entre as ações do jogador e os resultados na tela, enquanto o servidor permanece autoritativo e qualquer entrada inválida será descartada e não terá

efeito no servidor, que é o que outros jogadores percebem (GAMBETTA, 2012). Ao mesmo tempo que resolve o problema do atraso, essa escolha adiciona um outro problema, que pode ser demonstrado alterando o exemplo anterior de forma que o tempo de espera de resposta agora é 250 ms, e o jogador tenta se mover duas unidades para a direita em sequência.

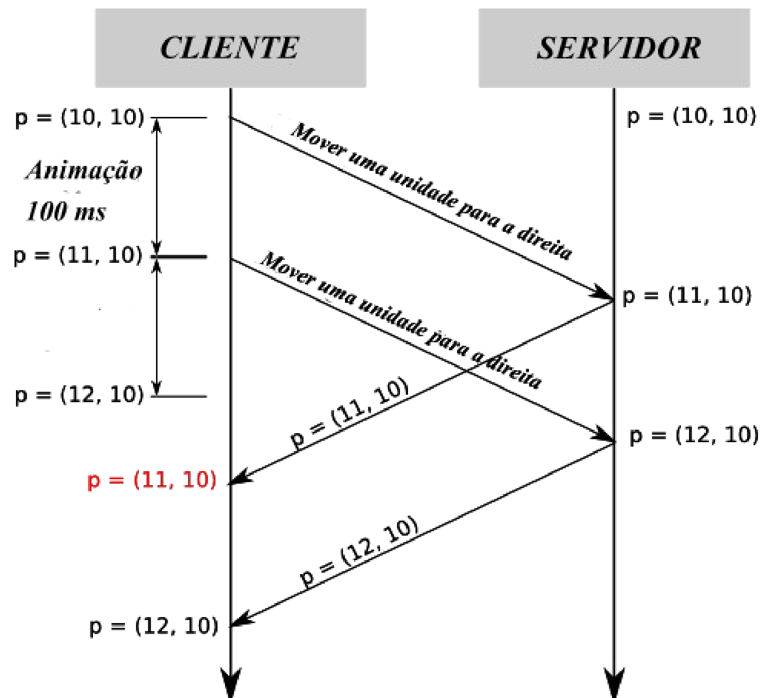



Figura 11: discrepância entre estado previsto e estado autoritativo (GAMBETTA, 2012).

O estado previsto pelo cliente é x = 12, mas a primeira resposta do servidor retorna dizendo que x = 11. Como o servidor é autoritativo, o cliente deve mover o personagem de volta para x = 11. Mas então, um novo estado chega em t + 350 ms que diz que x = 12, portanto o personagem sofre outro salto, dessa vez na outra direção. Do ponto de vista do jogador, ele apertou a seta para a direita duas vezes, o personagem se moveu duas unidades para a direita, ficou lá por 50 ms, saltou uma unidade para a esquerda, permaneceu lá por 100 ms e saltou uma unidade para a direita. Obviamente esse comportamento não é o desejado.

Apesar de o cliente ver o mundo do jogo em tempo real, as atualizações que ele recebe do servidor são resultados de ações que aconteceram no passado. Para resolver isso, o cliente adiciona um número sequencial para cada requisição guardando-as em uma lista, quando o servidor responde ele inclui o número da requisição da última entrada processada para que,

assim, o cliente possa conferir na lista, descartar requisições defasadas e recalcular sua posição considerando as que restaram.

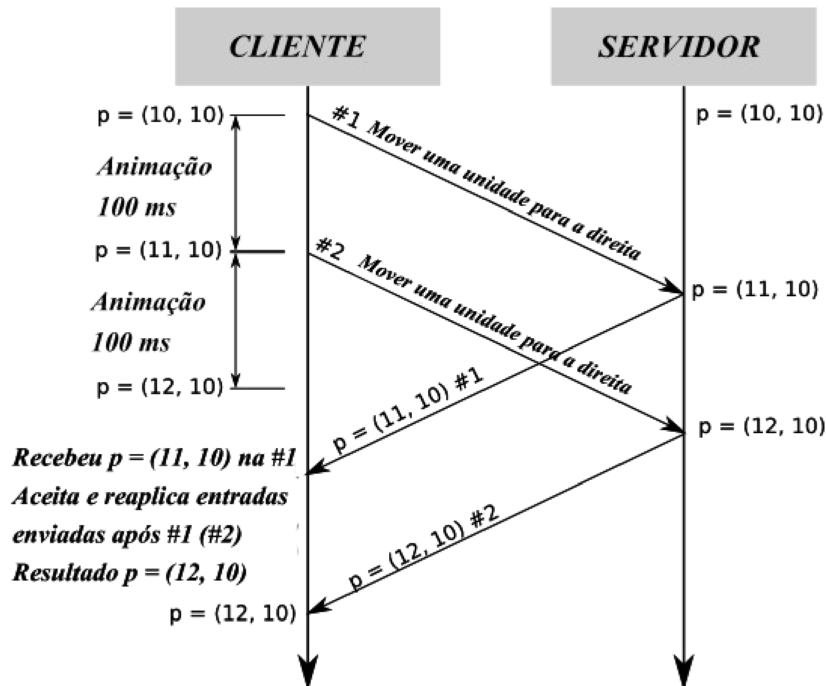



Figura 12: previsão na ponta do cliente e reconciliação com servidor (GAMBETTA, 2012).

O exemplo discutido aborda movimento, mas o mesmo princípio pode ser aplicado para quase todo o resto. Por exemplo, se em um jogo de turnos o jogador ataca outro personagem, pode-se mostrar sangue e um número representando o dano infligido mas não se deve subtrair o valor dos pontos de vida até o servidor ordenar. Da mesma forma, não se deve matar um personagem de outro jogador quando os pontos de vida estão abaixo de zero no estado de jogo do cliente, pois ainda não se sabe se o outro jogador tomou uma ação que pode ter prevenido sua morte. Isso mostra que mesmo trabalhando com um mundo de jogo determinístico é possível que o estado previsto pelo cliente divirja do estado que o servidor enviar, especialmente quando se leva em consideração ações de outros jogadores. O estado de jogo nem sempre é reversível, sendo preferível não tomar ações definitivas antes de receber a confirmação do servidor (GAMBETTA, 2012).

Em resumo, é preciso dar ao cliente a ilusão de responsividade enquanto espera um servidor autoritativo processar sua entrada. Para isso, ele simula o resultado das entradas e, quando recebe a atualização do servidor, o estado previsto é recomputado a partir do estado atualizado e dos comandos que o servidor ainda não reconheceu. Nessa abordagem, é preciso tratar efeitos sonoros do jogo com cuidado especial, pois a correção vinda do servidor pode obrigar o cliente a interromper, reiniciar ou manter um som ativo por mais tempo (BERNIER, 2001).

Uma vez que o cliente está executando parte do código do jogo na sua ponta, faria sentido permitir que fosse ele a atualizar o servidor com suas jogadas e posições, não mais dependendo do servidor autoritativo confirmá-las. No entanto, essa abordagem permitiria que jogadores mal intencionados, munidos de conhecimento sobre esse tipo de sistema, pudessem interferir no código do jogo e enviar jogadas, legais ou ilegais, que lhes dessem vantagens contra outros jogadores. Embora existam casos de servidores que confiam plenamente nos clientes, como é o caso de simulações de jogos de guerra militares, para jogos a escolha do servidor autoritativo faz sentido pois por mais que o mundo seja determinístico, jogadas perfeitamente legais podem acabar sendo descartadas e corrigidas quando a ação de entidades de jogadores remotos afeta o mundo de jogo compartilhado (BERNIER, 2001). Apesar de ser uma parte importante do planejamento do projeto, vulnerabilidades do sistema a trapaça e contramedidas não são o foco deste trabalho.

### 2.5.2 *Server time step*

Quando lida com múltiplos usuários, o funcionamento do servidor difere do descrito até agora. Como todos os usuários enviam seus comandos simultaneamente o tempo todo, seria inviável atualizar o estado do jogo a cada vez que uma requisição fosse recebida, por isso o servidor enfileira os comandos de cada jogador sem processá-los, se atualiza a uma frequência e envia os resultados para os usuários periodicamente a uma frequência menor – 10 vezes por segundo, por exemplo. O intervalo entre cada atualização é chamado *time step*, *tick rate* ou ainda *update rate*, que no exemplo utilizado é 100 ms (GAMBETTA, 2012).

Do ponto de vista local, essa abordagem funciona para o cliente pois a predição independe da taxa de atualização do servidor. Portanto, o comportamento permaneceria estável sob atualizações previsíveis, ainda que infrequentes. Porém, essas atualizações esparsas fornecem ao cliente informação limitada sobre outras entidades se movendo pelo mundo.

Uma primeira implementação dessa abordagem moveria as entidades do jogo quando recebesse uma atualização de estado, o que implica que a cada 100ms aconteceriam saltos na posição dos personagens ao invés de um movimento contínuo, como mostra a figura 12.

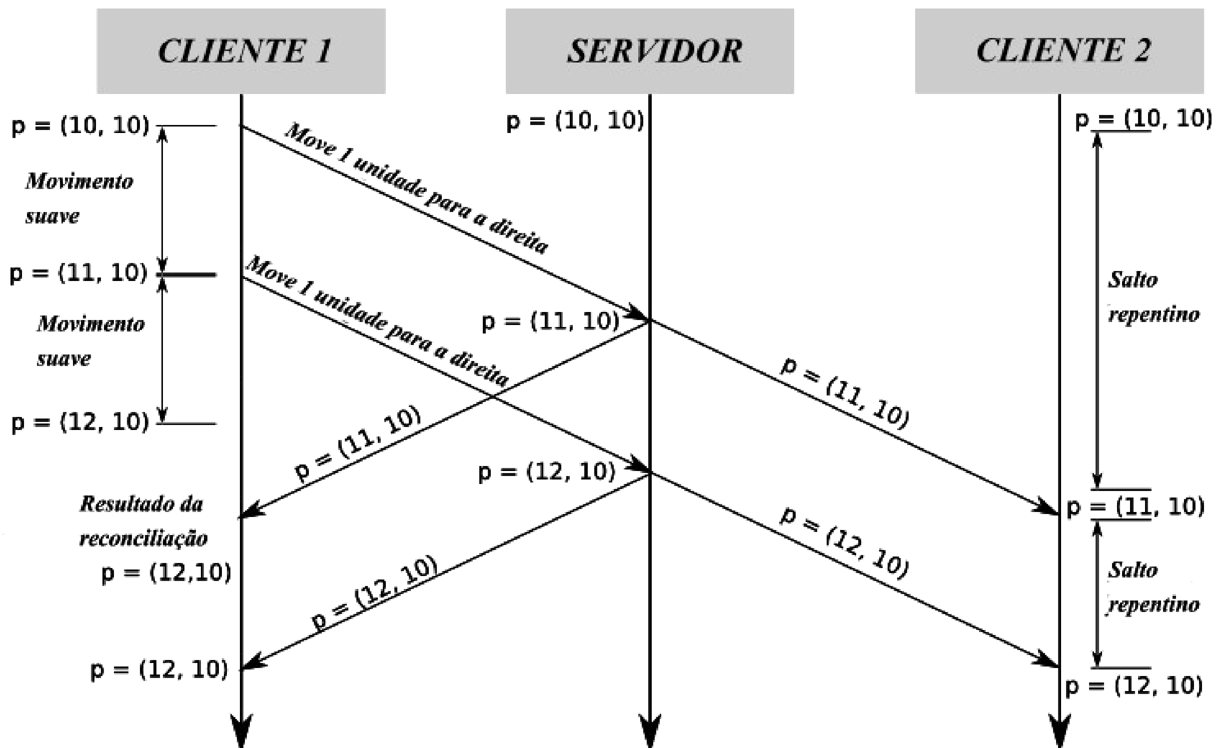



Figura 12: cliente 1 visto pela perspectiva do cliente 2 (GAMBETTA, 2012).

Existem formas de lidar com isso que dependem do tipo de jogo em questão, mas de modo geral quanto mais previsíveis forem as entidades do jogo mais fácil se torna contornar essa limitação. Essas formas podem ser divididas entre interpolação e extrapolação de posição. Para fins deste trabalho, a extrapolação pode ser entendida como o processo de inferir uma posição ainda não recebida com base em informações passadas, e interpolação como o processo de criar um número n de posições intermediárias entre duas posições conhecidas (BERNIER, 2001). Um exemplo de extrapolação é a técnica de *dead reckoning*.

O fator que define a técnica a ser usada é a previsibilidade de movimento. Em jogos onde o movimento é errático e a direção e velocidade podem variar abruptamente, técnicas de extrapolação não são úteis pois a tentativa de previsão falha na maior parte do tempo, gerando saltos indesejados na posição ou na animação. Um exemplo desse tipo seriam jogos de tiro em primeira pessoa (*First Person Shooter*), em que os jogadores costumam correr, parar e girar com pouco ou nenhum tempo de transição, tornando posição e velocidade difíceis de serem previstas com base em comportamento passado (BERNIER, 2001).

No exemplo, sabe-se que a posição das entidades será amostrada para o cliente a intervalos menos frequentes do que o necessário para que seu movimento seja renderizado fluidamente. Para resolver isso através da interpolação, toma-se a decisão de computar localmente a posição das

entidades mais tarde do que é recebida, em função do valor do *time step* do servidor, mais especificamente.

Assim, supondo que se receba do servidor a posição de um jogador referente a t = 900 ms, no cliente a entidade desse jogador só chegaria naquela posição em t = 1000 ms; tendo em mãos a posição referente a t = 1000 ms, a entidade terminaria seu movimento naquela posição em t = 1100 ms. A técnica de interpolação é aplicada entre posições, o atraso garante que a posição referente a 100 ms antes da atualização mais recente é sempre conhecida, portanto essas duas posições são usadas como limites para criar etapas de deslocamento e mover a entidade entre elas durante esse intervalo de renderização de 100 ms, gerando a impressão de movimento contínuo.

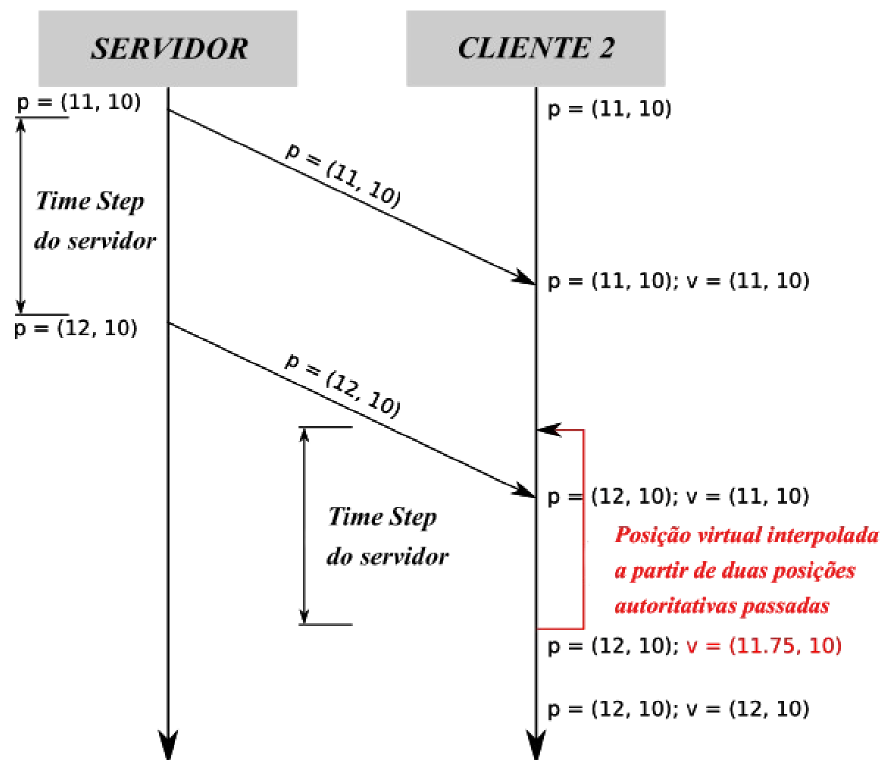


Figura 13: cliente 2 renderiza cliente 1 “no passado”, interpolando as últimas posições conhecidas (GAMBETTA, 2012).

A concessão que se faz ao usar interpolação dessa forma é a de que, apesar de perceberem suas próprias ações em tempo real, clientes enxergam ações de outras entidades pós-fato de forma mais acentuada, e cada jogador experiencia uma visão levemente diferente do mundo do jogo. Para lidar com perda de pacotes, seria possível aumentar o intervalo de interpolação para 200 ms. Considerando que o servidor mantém a taxa de 100 ms, seria possível não receber uma atualização e a entidade do jogador continuaria sendo interpolada

para uma posição válida. Gastar mais tempo interpolando é uma troca, pois aumenta a latência para ter fluidez visual maior (BERNIER, 2001). Os dados de posição utilizados para interpolar movimento dependem das necessidades do jogo e do quão precisa a interpolação deve ser, podendo-se também ajustar a frequência do *time step* de acordo. Uma outra abordagem é o servidor enviar com cada atualização uma sequência de posições do jogador remoto para que a interpolação do movimento seja a mais fiel possível.

Na prática, a evolução das tecnologias envolvidas faz com que o atraso necessário para interpolação seja pequeno e não facilmente perceptível para o jogador. Servidores de jogos de FPS mais recentes utilizam taxas de atualização bem mais altas do que o exemplificado, e chegam até a categorizar usuários de acordo com sua largura de banda enviando mais atualizações para os que possuem maior capacidade.

### 2.5.3 Ações críticas

Em geral, a escolha de interpolar o movimento de jogadores remotos gera uma experiência de jogo robusta e transparente, mas ações ou eventos de alta precisão espacial e temporal, como disparo de armas de efeito instantâneo, precisam de consideração adicional. Se o cliente enxerga entidades remotas apenas em posições defasadas, acertar um tiro preciso comparando a sua mira com a posição real da entidade seria inviável.

Para manter a consistência perceptiva da simulação para todos os jogadores e ser coerente com a visão de mundo deles, o servidor precisa analisar esses casos a partir de seus pontos de vista. Para isso, o cliente envia a informação completa do disparo, incluindo o momento preciso da ação. Tendo essa informação, o servidor retrocede a simulação para saber o que estava em sua mira no momento em que a ação aconteceu, e assim decide qual o seu efeito. Caso as atualizações do servidor sejam pouco frequentes o ideal seria tratar ações de efeito instantâneo assincronamente ao *time step*, pois atrasos na resolução dessas ações podem gerar inconsistências visuais e sonoras perceptíveis pelo usuário, e, mesmo assim, essa abordagem não estaria completamente livre de inconsistências. Um exemplo de uma inconsistência possível seria o jogador B ser atingido pelo tiro do jogador A logo após ter encerrado seu movimento se protegendo atrás de uma parede. Isso se justifica pois é possível que na visão de A, o jogador B ainda não estivesse sob cobertura no momento do tiro. Outro exemplo seria do ponto de vista do jogador B, que poderia ter percebido o jogador A mirando para outra coisa além de si mesmo no momento antes do disparo, o que é

explicado pelo fato de que a última atualização de estado referente ao jogador A poderia não haver chegado antes (BERNIER, 2001).

## 2.6 Estratégias de compensação de atraso

Nesta seção serão abordados métodos conhecidos de lidar com o problema de sincronização e consistência. Além de representarem um custo computacional adicional, algumas estratégias são específicas a determinadas tipos de jogos ou hierarquias de rede, tendo aplicação e vantagens restritas a esses casos.

### 2.6.1 Filtro de percepção local (*Local Perception Filter*)

É um método que prolonga o tempo de decisão para esconder o intervalo de espera , alterando a realidade do jogador. O jogador A verá sua ação acontecendo mais lentamente enquanto o jogador B a verá acontecendo mais rápido. Usado em jogos de tiro em primeira pessoa (*First Person Shooter*) para manter sincronização em situações onde jogadores estão distantes uns dos outros no espaço virtual e um projétil lento, como um míssil, é disparado. Na realidade de A, o míssil avança a uma velocidade normal até chegar próximo de B, quando perderá velocidade de uma forma que A não consiga facilmente perceber a diferença. Alternativamente, a velocidade em A poderia se manter normal e em B ser acelerada.

Essa abordagem não funciona bem para compensar o intervalo de espera em situações onde A e B estão suficientemente perto um do outro no espaço virtual e, portanto, qualquer aceleração ou desaceleração pode ser facilmente notada. Além disso, qualquer modificação do tempo de decisão é perceptível quando esse tempo já é muito curto (XU e WAH, 2013).

### 2.6.2 *Time warp synchronization* e *trailing state synchronization*

Uma estratégia de sincronização otimista que executa especulativamente comandos recebidos em ordem de chegada, sem garantia de ordem causal entre eles. A cada comando recebido é salva uma "imagem" do estado da simulação. Ao ser recebido um comando cuja

ordem de execução é anterior ao último executado, é realizado um *rollback,* revertendo a simulação, tendo como referência a imagem do último estado válido, e então são executados os comandos na ordem devida. Essa estratégia é onerosa em relação à memória devido ao custo envolvido em salvar imagens completas da simulação a cada mensagem recebida, assim como ao poder de processamento necessário para executar vários comandos numa mesma demanda sempre que houver um *rollback*.

Similar ao *Time Warp*, o chamado *Trailing state synchronization* também executa *rollbacks* quando detecta inconsistências, porém seu funcionamento pode implicar menor dependência de recursos computacionais. Em vez de salvar imagens da simulação no tempo de cada comando recebido, são mantidas apenas duas cópias da simulação separadas pelo tempo, uma estando no período vigente de execução local e outra um determinado intervalo de tempo atrás. A cópia mais atual é chamada de *leading state* e a defasada é chamada de *trailing state*. Havendo uma inconsistência no l*eading state* e a necessidade de um *rollback*, no lugar de copiar o estado de uma imagem, copia-se o *trailing state* e executa todos os comandos entre o ponto de inconsistência e o ponto de execução atual.

Essa estratégia apresenta vantagem em relação ao *Time Warp* apenas nos casos em que cópias da simulação são grandes e computacionalmente caras de manter, e o intervalo de tempo entre os estados possa ser pequeno, necessitando poucos passos para chegar ao estado atual novamente. De outra forma, o custo de um *rollback* nas duas abordagens é próximo (KHAN, CHABRIDON e BEUGNARD, 2007).

Apesar destas estratégias terem sido desenvolvidas para aplicações Cliente-servidor muitas vezes voltadas a lidar com modelos abstratos de dados, a noção de *rollback* aqui apresentada nos é interessante pois sua aplicação também existe em jogos.

### 2.6.3 *Rollback* aplicado a jogos

Na prática, o modelo de atraso local é percebido pelo jogador como um atraso na responsividade do jogo equivalente ao tempo que o pacote com os comandos leva para chegar ao destino. Isso difere da proposta de gêneros que demandam responsividade como os jogos de luta, e, em muitos casos, o atraso introduzido pela camada de rede pode mudar completamente a experiência do jogo *online*, especialmente se a compararmos à *offline*.

Em vez de atrasar o processamento das entradas esperando os comandos chegarem à simulação para então executar um quadro, um sistema com *rollbacks* usa execução especulativa,

processando os comandos do jogador local sem esperar os comandos do jogador remoto. Isso elimina a sensação de *lag* para o jogador local criada pelo modelo convencional de atraso. Quando a entrada do jogador remoto finalmente chega, o sistema se corrige (com um *rollback*) modificando o estado do jogo para refletir corretamente as consequências da ação do jogador remoto, mantendo a sincronia entre as simulações. Isso é feito transparentemente aos jogadores pois o tempo de viagem de um pacote é escondido entre o jogador remoto realizar uma ação e a simulação local perceber que uma ação foi feita por ele.

Suponhamos que duas pessoas estão jogando em uma rede onde o tempo de troca de pacotes é 60ms. Quando o jogador remoto executar uma ação a simulação local levará 60ms para perceber que uma ação foi feita. Assim, o sistema vai instruir o jogo a voltar no tempo em 60ms, corrigir a ação do jogador remoto, e então avançar a simulação de volta para o tempo atual antes da próxima renderização. Como resultado, os primeiros 60ms da animação da ação remota serão pulados na simulação local, ou seja, é como se a ação começasse 60ms à frente do normal. A figura 7 demonstra os diferentes momentos em que cada simulação recebe e aplica as ações dos jogadores local e remoto.

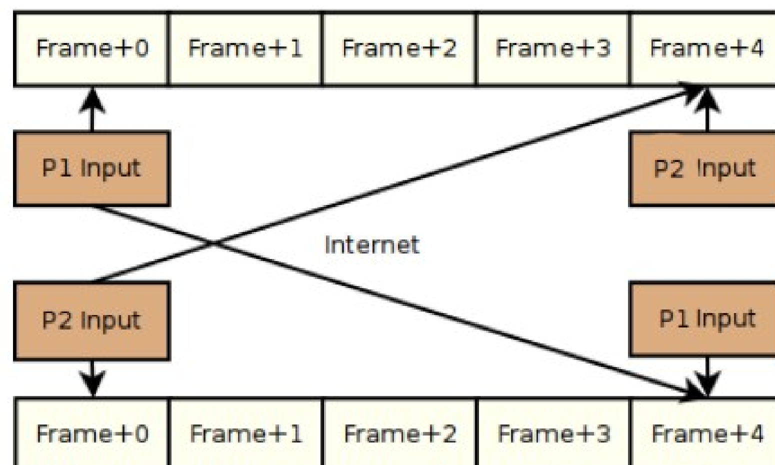



Figura 7: exemplo de um modelo de *rollback* com 4 *frames* de diferença (MAUVE, 2012).

Apesar de não ser ideal, pois significa que o jogador local nunca enxergará os primeiros *frames* da ação do jogador remoto, essa perda resulta em uma experiência preferível não só devido a uma melhora na resposta dos controles, mas porque os primeiros momentos

de uma ação são geralmente desconsideráveis uma vez que ações em um jogo de luta possuem tempo de preparação e/ou translação antes de afetar a simulação, são determinísticas e não são afetadas por comandos posteriores (CANNON, 2012).

A figura 8 mostra as fases de uma ação típica de um jogo de luta e suas durações. A preparação é a fase que representa o tempo que a ação leva para ficar pronta, a ativação é a fase em que a ação acontece, e a recuperação é a fase que ele deve esperar para poder realizar outra ação. Apenas a fase da atividade influencia diretamente o adversário e portanto é a que importa para fins de tomada de decisão.

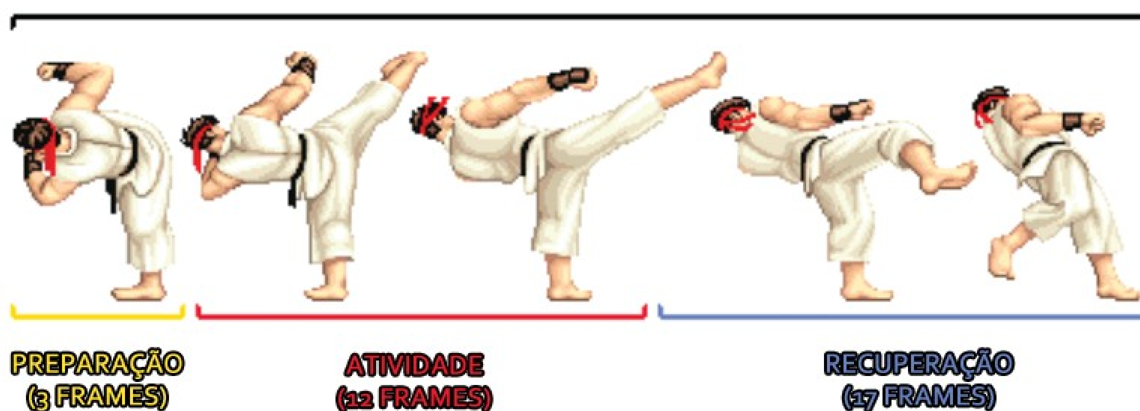



Figura 8: exemplo de ação (chute) em um jogo de luta numa divisão por fases (CANNON, 2012).

Devido ao seu funcionamento, atrasos adicionais não são mais necessários para garantir a continuidade do jogo em caso de falhas isoladas de comunicação, desde que se esteja disposto a reverter *frames* a mais do que o que foi definido a princípio. Basta manter a execução normal até receber dados e então retroceder com *rollback* o número de *frames* necessários, que no caso da figura 9 são dois. Obviamente, é prudente definir um limite razoável para a espera sem bloqueio (MAUVE, 2012).

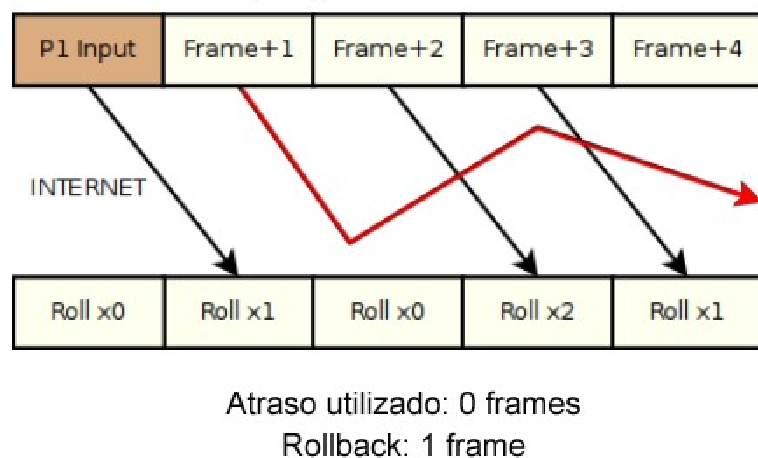



Figura 9: *rollbacks* mascarando pacotes perdidos (MAUVE, 2012).

A decisão de ignorar a entrada do jogador remoto por um número indeterminado de *frames* pode resultar em um comportamento indesejado quando a entrada chegar e for necessária uma correção. Por exemplo, em jogos onde a tela aproxima ou afasta a câmera em relação à proximidade entre os jogadores, uma correção na posição de um dos jogadores acarreta uma correção na profundidade da tela, podendo gerar um efeito visual de transição abrupta perceptível.

O gerenciamento de sons também pode complicar quando se lida com *rollbacks*. Supondo que o pedaço de um efeito sonoro foi reproduzido em um determinado *frame*, e esse *frame* foi corrigido por um *rollback,* é preciso verificar se o efeito ainda é válido. Caso não seja, deve-se verificar se esse mesmo efeito será reproduzido novamente em algum dos *frames* à frente, e se realmente for inválido, sua reprodução deve ser cancelada de forma suave para evitar o efeito de "estouro" nas caixas de som (CANNON, 2012).

Uma forma de evitar esses problemas é tentar prever as ações do jogador remoto para minimizar o número de *rollbacks* e, assim, de inconsistências visuais e sonoras que podem ocorrer durante uma correção do estado do jogo. A função de um algoritmo de predição é tentar prever os comandos que chegarão da rede no próximo ciclo, usando para isso o histórico dos comandos anteriores ou critério que melhor se adapte ao jogo em questão.

> A qualidade de um algoritmo de predição é calculada pela frequência e severidade dos erros de renderização causados por uma previsão errada. Por exemplo, em uma partida de *Pong*, uma previsão errada pode causar um salto na posição da raquete. Errar a posição da raquete é grave, porém um algoritmo que erra todas as vezes por

> uma margem de alguns *pixels* é preferível a um que erre apenas 10% das vezes mas que cada erro faça a raquete saltar um quarto da tela. Naturalmente, o melhor algoritmo de predição seria específico para cada jogo ou jogador. (CANNON, 2012, p. 13, tradução nossa).

### 2.6.4 *Dead reckoning*

Quando o comportamento das entidades do jogo segue um padrão à risca é possível extrapolar o seu movimento de forma eficaz. Baseado nas últimas informações recebidas como a posição, direção, velocidade e aceleração, o algoritmo de *dead reckoning* é utilizado para estimar futuras posições do jogador remoto. Dessa forma, apenas quando uma mudança na posição e orientação da entidade real diferir da estimada por uma margem de tolerância é que enviar uma nova atualização se faz necessária (IEEECS, 2012).

A margem pode ser definida em função de variações de velocidade ou de ângulo da trajetória. Outra forma de entender a margem de tolerância é imaginá-la como uma gaiola virtual cujo centro é a última posição recebida da entidade, quando o movimento do modelo estimado atingir as paredes da gaiola, uma atualização se faz necessária. Em uma implementação completamente assíncrona, ou seja, independente de taxa de atualização fixa, a simulação local também mantém um modelo estimado da sua própria posição para saber quando é preciso enviar uma nova atualização de posição à outras simulações.

A vantagem dessa abordagem é a redução da banda necessária para manter sincronia, pois permite menor frequência de mensagens de atualização enquanto mantém entidades remotas em trajetória coerente. Essa técnica é mais utilizada em jogos de corrida, e é especialmente eficaz quando o trajeto é linear ou de fácil predição, como, por exemplo, uma pista de corrida oval.

O estado estimado pode ser diferente do estado real recebido na próxima mensagem de atualização, caso em que um método de convergência é usado para interpolar a diferença entre o estado estimado e o estado correto. Idealmente, o método deve convergir o estado previsto e o estado real de forma suave e cadenciada, para não afetar negativamente a experiência dos jogadores.

O *Distributed Interactive Simulation* (DIS) foi uma iniciativa conjunta entre governo e indústria que culminou no padrão 1278, construído pelo IEEE. Nele são definidas normas de infraestrutura que permitem que simulações interativas de projetos e propósitos distintos se comuniquem. Nesse padrão são definidos algoritmos de *dead reckoning*, assim como formato e semântica de mensagem visando interoperabilidade, o chamado *Protocol Data Unit* (PDU) (IEEECS, 2012). A fórmula de número 5 apresentada no documento 1278.1:2012 (IEEECS, 2012) para calcular a posição extrapolada de uma entidade em um determinado momento se dá por:

$$P = P_0 + V_0 \Delta t + \frac{1}{2} A_0 \Delta t^2 \qquad (2)$$

Existem diferentes abordagens para a correção do estado extrapolado, cujo intuito é convergir para uma posição mais próxima ao estado real da entidade. A figura 10 exemplifica os pontos finais de uma extrapolação convergindo para um ponto médio da extrapolação seguinte.

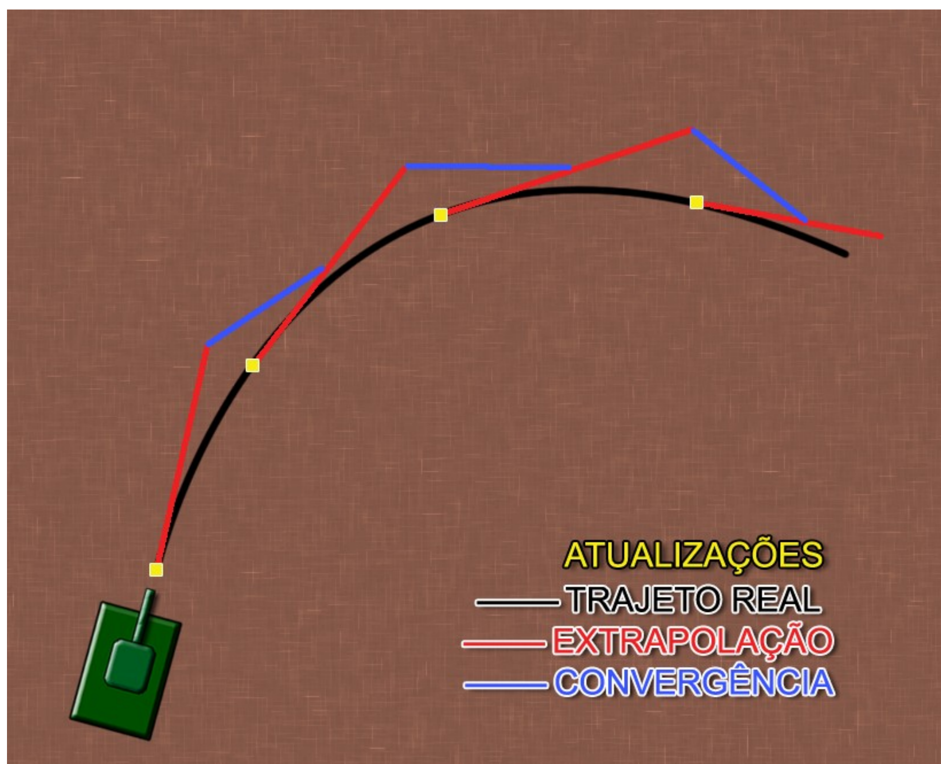



Figura 10: exemplo de extrapolação e convergência de um trajeto amostrado.

Como é possível notar no exemplo da figura 10, nem sempre o movimento resultante do *dead reckoning* serve para caracterizar o movimento original. A técnica de *dead reckoning* definitivamente tem seu custo, pois além de exigir que as entidades tenham um comportamento previsível, exige também que todos os computadores da rede executem um algoritmo para extrapolar cada entidade visível na sua simulação. Porém, quando se pode trocar processamento por menor uso de banda e menor latência aparente, é uma opção preferível a atrasar a computação local de atualizações de entidades remotas para poder interpolar seu movimento.

# 3 PROJETO CONCEITUAL

Jogos eletrônicos interativos quando planejados na forma *multiplayer online* precisam transpor o problema da latência para apresentar uma experiência suficientemente responsiva aos seus jogadores. Dependendo da plataforma e gênero, isso pode implicar a construção de uma lógica de jogo completamente diferente da contraparte *single player* do mesmo, pois ele passa a depender não apenas do jogador local, mas também de ações de jogadores remotos para avançar o estado da simulação. Supondo que a comunicação entre instâncias do jogo se desse sob circunstâncias próximas de ideais, como se configuraria se todas estivessem dentro de uma mesma rede local, a latência poderia ser tratada de forma trivial ou até desconsiderada, e não seria necessário ir além. Porém, com o avanço das tecnologias e da I*nternet*, temos jogadores, muitas vezes separados por cidades ou continentes, interagindo sob latências variáveis, e, nesse contexto, é preciso garantir a robustez do sistema oferecendo respostas à inerente não confiabilidade da rede.

## 3.1 O projeto

Este trabalho propõe a criação de um simulador simples de batalha de carros de combate, em outras palavras, um protótipo de jogo interativo. O protótipo possui elementos comuns ao gênero, dentre eles uma arena limitada servindo de localidade por onde os carros se movimentam. Ele também apresenta suporte a um modo *multiplayer* simples cujo funcionamento se dá à base de troca de mensagens de atualização de posição. Esse modo seria então expandido através de um *middleware* que implementa algoritmos de *dead reckoning*, se espelhando em normas do padrão 1278, conforme relevância. A fim de compreender a eficácia dessa técnica de extrapolação, criam-se diferentes perfis de movimentação e observa-se o comportamento da simulação em resposta a eles, comparando-se a performance desses algoritmos com uma implementação que não considera o problema da latência, em função da variação da taxa de atualizações. O *middleware* deve ser capaz de variar a frequência de mensagens enviadas, assim como simular falhas da rede.

Espera-se que através deste seja possível compreender melhor o funcionamento de um jogo *multiplayer* e como ele é afetado pela latência. A figura 11 evidencia o projeto conceitual do simulador proposto.

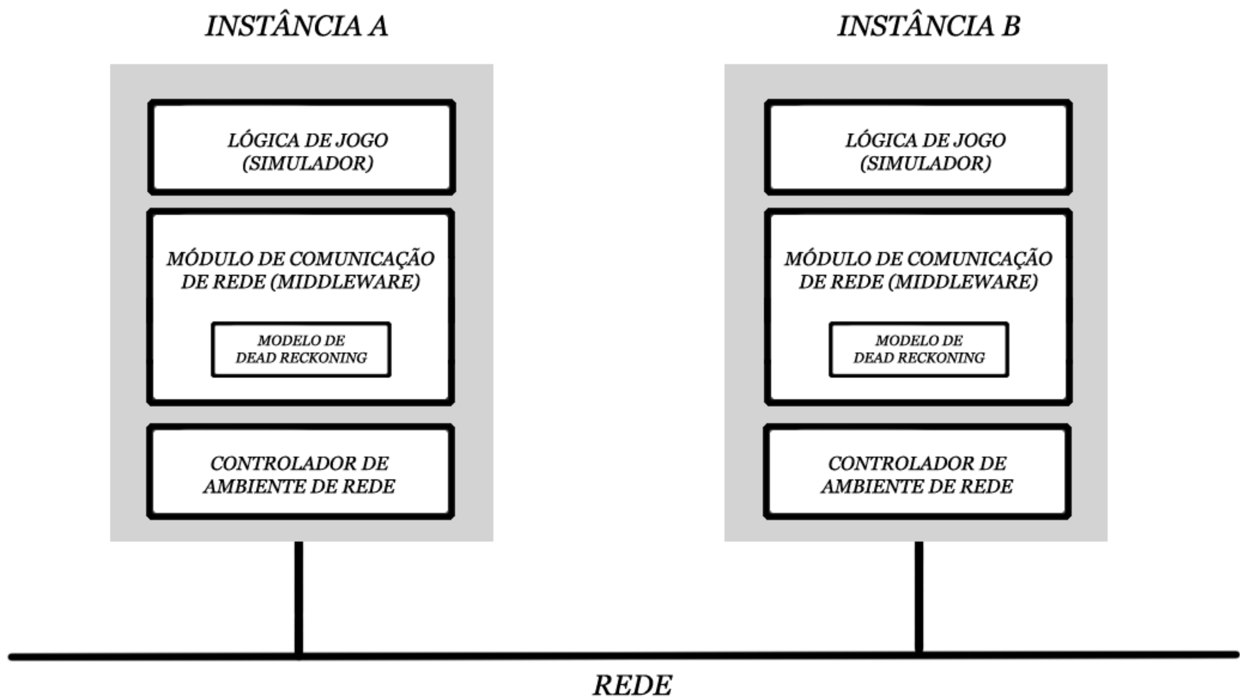



Figura 11: módulos que compreendem uma instância do simulador.

As seções seguintes definem os módulos introduzidos pela figura 11 e formalizam o escopo das suas funções.

### 3.1.1 Módulo controlador de ambiente de rede

A função deste módulo é controlar as características de rede a fim de emular o comportamento de uma rede não-confiável, de forma que se possa criar um ambiente de teste mais próximo do desejado independente das condições existentes entre as instâncias no ambiente em que estão inseridas. Todas as mensagens recebidas pelo simulador são recebidas por este módulo e, a seu critério, enviadas para a análise do módulo de comunicação. Através dele definem-se valores para propriedades da rede como a latência, *jitter*, porcentagem de pacotes perdidos, entre outros.

### 3.1.2 Módulo de comunicação de rede

Este módulo compreende o *middleware* propriamente dito. Sua principal função é gerenciar as entidades do jogo que compõe o âmbito *multiplayer*, e de forma mais direta reger as entidades extrapoladas de acordo com o método de extrapolação e convergência vigentes, além de definir parâmetros de comunicação como a frequência de mensagens de atualização por intervalo de tempo.

### 3.1.3 Módulo do modelo de *dead reckoning*

Planejado para ser uma sub-rotina do módulo de comunicação, contém a especificação dos modelos de *dead reckoning* e informações referentes ao modelo vigente, cuja ideia é ser mutável em tempo de execução. Os algoritmos previstos são baseados nas fórmulas de movimento estacionário, linear, e acelerado, conforme definidas pelo DIS (IEEECS, 2012).

| Field | Model | Formula | Examples |
|---|---|---|---|
| 1 | STATIC | N/A | Static entities |
| 2 | DRM (FPW) | $P = P_0 + V_0\Delta t$ | Constant velocity (or low acceleration) linear motion |
| 3 | DRM (RPW) | 1) $P = P_0 + V_0\Delta t$ 2) $[R]_{w\to b} = [DR][R_0]_{w\to b}$ | Similar to DRM 2 but where orientation is required (e.g., visual simulation) |
| 4 | DRM (RVW) | 1) $P = P_0 + V_0\Delta t + \frac{1}{2}A_0\Delta t^2$ 2) $[R]_{w\to b} = [DR][R_0]_{w\to b}$ | Similar to DRM 5 but where orientation is required (e.g., visual simulation) |
| 5 | DRM (FVW) | 1) $P = P_0 + V_0\Delta t + \frac{1}{2}A_0\Delta t^2$ | High speed (e.g., missile) or maneuvering at any speed |
| 6 | DRM (FPB) | 1) $P = P_0 + [R_0]^{-1}_{w\to b}([R1]V_b)$ | Similar to DRM 2 but when body-centered calculation is preferred |
| 7 | DRM (RPB) | 1) $P = P_0 + [R_0]^{-1}_{w\to b}([R1]V_b)$ 2) $[R]_{w\to b} = [DR][R_0]_{w\to b}$ | Similar to DRM 3 but when body-centered calculation is preferred |
| 8 | DRM (RVB) | 1) $P = P_0 + [R_0]^{-1}_{w\to b}([R1]V_b + [R2]A_b)$ 2) $[R]_{w\to b} = [DR][R_0]_{w\to b}$ | Similar to DRM 4 but when body-centered calculation is preferred |
| 9 | DRM (FVB) | 1) $P = P_0 + [R_0]^{-1}_{w\to b}([R1]V_b + [R2]A_b)$ | Similar to DRM 5 but when body-centered calculation is preferred |

Figura 12: tabela com fórmulas para cáculo da extrapolação de dead reckoning. Utiliza-se de matrizes de rotação em ambiente 3D (IEEECS, 2012).

### 3.1.4 Módulo de lógica do jogo

Mantém e manipula as informações referentes ao jogo, como posição, pontos de vida, munição, número da rodada, etc. Como sub-rotina, pretende-se a automatização do carro da simulação local, para que se mova de forma independente sem necessitar de interação com usuário, de forma a facilitar testes.

# 4 IMPLEMENTAÇÃO

## 4.1 Ferramentas utilizadas

### 4.1.1 Unity 3D

A *Unity* é uma *game engine* proprietária criada e mantida pela empresa *Unity Technologies*. Ela oferece a capacidade de criar jogos em 2D ou 3D, para isso dando suporte à várias APIs gráficas consolidadas, como  Direct3D, OpenGL e WebGL. A *engine* oferece uma API de *scripting* primariamente em C#, tanto para o editor quanto na forma de plugins e os jogos, propriamente. Sua interface funciona como um editor tudo-em-um extensível, contando, por exemplo, com a opção de criação e edição de terrenos ao modo de um modelador 3D, possuindo ferramentas amigáveis direcionadas para artistas e desenvolvedores. A imagem 12 exemplifica essa interface.

Figura 13: interface da *engine Unity*.

Outro ponto que torna a *Unity* vantajosa é sua capacidade de exportar um projeto para várias plataformas diferentes, desde dispositivos móveis a navegadores, *desktops* e consoles. Com várias funcionalidades e extensões extras, incluindo serviços voltados a desenvolvedores, a *Unity* possui

seus próprios componentes de rede para gerar funcionalidade *multiplayer*, o que torna a *engine* suficiente para implementar o trabalho proposto (UNITY, [2018]).

### 4.1.2 *Microsoft Visual Studio*

O *Visual Studio* é um ambiente de desenvolvimento integrado (IDE) da empresa *Microsoft* para desenvolvimento de *software*, em especial *software* dedicado ao .NET *Framework* e às linguagens *Visual Basic* (VB), C, C++, C# e F#. Também possibilita desenvolvimento na área *web*, usando a plataforma do ASP.NET para *websites*, aplicativos *web*, serviços *web* e aplicativos móveis. As linguagens trabalhadas com maior frequência nessa plataforma são: VB.NET (Visual Basic.Net) e o C♯ (VISUAL STUDIO, [2018]).

Das vantagens de utilizar essa IDE, além de funcionalidades avançadas que facilitam o trabalho de programadores particulares a esse editor, pode-se citar como fundamental na sua escolha a opção de integração à *Unity* como ferramenta de depuração de *scripts*.

### 4.1.3 C#

A linguagem C# faz parte do conjunto de ferramentas oferecidas na plataforma .NET, projetadas para funcionar na *Common Language Infrastructure*, e tem como objetivos ser uma linguagem simples, robusta, orientada a objetos, fortemente tipada e altamente escalável a fim de permitir que uma mesma aplicação possa ser executada em diversos dispositivos de *hardware*. A sua sintaxe orientada a objetos foi baseada no C++ mas inclui muitas influências de outras linguagens de programação, principalmente Java, além de oferecer suporte parcial a algumas operações do paradigma funcional (C# artigo da Wikipedia, 2015).

## 4.2 Desafios

Pode-se citar como uma dificuldade o tempo disponível, que se provou bastante curto para abordar suficientemente o tema e dominar as ferramentas utilizadas. Grande parte do

tempo dispendido em completar este trabalho foi gasto tentando entender os pormenores da *Unity* e como adaptar o pensamento e lógica de programação de acordo com o seu paradigma.

Apesar de ser uma ferramenta extraordinária, o fato de ter passado por múltiplas revisões e recebido várias versões ao longo dos anos fez com que muitos dos exemplos encontrados na *Internet* estivessem defasados, e em alguns casos a própria documentação oficial deixa a desejar ou fosse insuficiente para passar conceitos básicos. Um exemplo é a documentação sobre como utilizar a camada de transporte de rede para iniciar *multiplayer*, que é bastante rasa e não serviu de muita ajuda em sua compreensão.

No desenvolvimento dos testes, encontrar um modo de automatizar o movimento da entidade dos jogadores foi um desafio, pois algumas operações realizadas pela HLAPI não permitem manipulação de seus parâmetros, ou então são caixas fechadas, não sendo possível vislumbrar sua implementação padrão para referência quando se intenciona customizar ou estender funcionalidades.

## 4.3 Sobre a Unity

Para entendimento do que foi feito, se faz necessário primeiro esboçar o funcionamento da comunicação de rede da própria *Unity,* chamada de UNET, e os elementos que dão suporte à criação de jogos com funcionalidade *multiplayer*, de modo a justificar as escolhas para implementação do protótipo. A figura 13 mostra como está organizado o sistema interno da Unity e a função de cada camada que o compõe (*UNITY user manual*, [2018]).

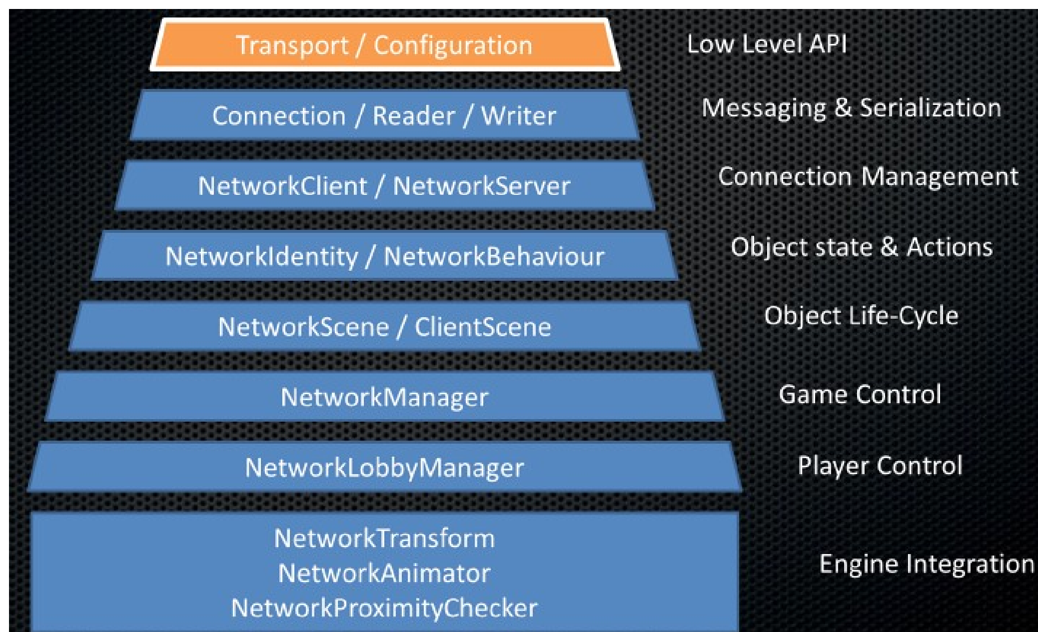



Figura 14: ordenado de cima para baixo, hierarquia de camadas que formam a rede da *Unity* e suas atribuições.

A camada base é a *Transport Layer*, também chamada de *Low Level API* (LLAPI), camada fina de comunicação em tempo real que figura o cerne do sistema e trabalha diretamente sobre a camada baseada em *sockets* do sistema operacional. Em contraste à camada base existe a *High Level API* (HLAPI), cuja classe central é o *NetworkManager*, que gerencia tarefas básicas de jogos *multiplayer*. Como a *API* de alto nível foi constituída através dos componentes da *API* de baixo nível, ambas podem ser usadas na construção de um modo *multiplayer*. Porém, enquanto a LLAPI requer alta configuração para oferecer controle granular sobre os canais de comunicação e tipos de mensagens trocadas, a HLAPI obriga uma topologia cliente-servidor autoritativa e entrega várias funcionalidades de imediato. Algumas desses funcionalidades são encapsuladas e devem ser acessadas da forma como existem enquanto outras necessitam de complementação simples através do editor da *Unity*, e outras podem ser estendidas através de código adicional (Unity user manual, *[2018])*.

Tendo em vista o tempo disponível e o fato de que este trabalho não intenciona criar um jogo completo, mas sim simular o funcionamento de conceitos vistos no capítulo 3, por ser uma alternativa mais rápida e direta foi escolhida a HLAPI para dar sequência à implementação. Em contrapartida, devido ao menor controle dos processos decorrente dessa escolha, ela impede uma implementação completamente modular, pois alguns dos módulos do

projeto conceitual compartilham código ou são suplantados pela HLAPI, a exemplo do *NetworkManager* que simula pacotes perdidos e latência.

### 4.4 Sincronização de entidades

O *NetworkManager* é responsável por instanciar, destruir e sincronizar objetos que interagem pela rede, criar a conexão entre cliente e servidor, entre outras coisas. Na *HLAPI* da UNET, *p*ara indicar que o estado de um objeto da *Unity* precisa ser sincronizado através da rede, adiciona-se a ele um componente do tipo *NetworkIdentity.*

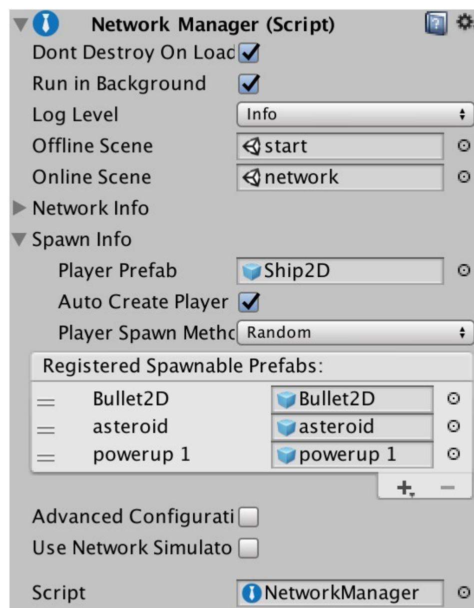



Figura 15: visualização do *NetworkManager* no inspetor da *Unity*. Popular as entradas com cenas e *prefabs* é o que permite uma configuração rápida do modo *multiplayer*.

Além desse, outros componentes específicos são necessários para definir exatamente que ações ou comportamentos do objeto precisam de sincronização, como por exemplo *NetworkAnimator* para sincronia de animação ou *NetworkTransform* para sincronia de movimento, bastando adicionar o componente certo ao objeto e torná-lo um modelo instanciável *(prefab*) na lista do *NetworkManager*, para que este o gerencie.

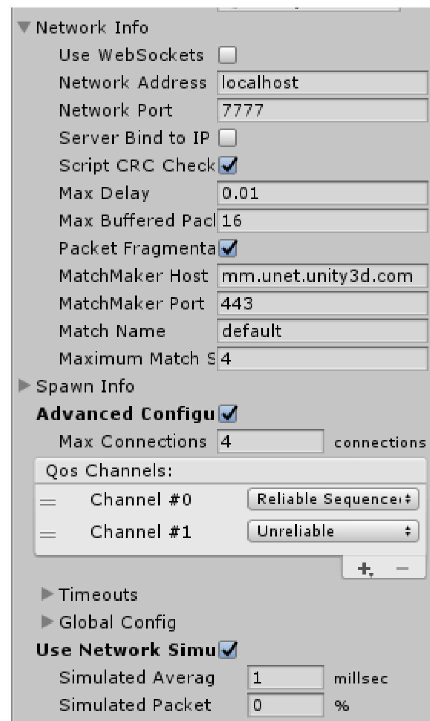



Figura 16: opções específicas à parte de rede do *NetworkManager* visualizadas dentro do inspetor da *Unity*. Os últimos campos correspondem a atraso em milissegundos e porcentagem de pacotes perdidos.

Mesmo utilizando-se de facilidades da HLAPI para a geração de modo *multiplayer,* ainda é possível ter maior controle de alguns objetos definindo pormenores da sincronização através de código, da mesma forma que seria necessário caso fosse escolhido usar a LLAPI. Para isso, abandona-se simplesmente atribuir componentes específicos nos objetos e atrela-se a eles *scripts* de código, o código fica responsável por especificar o comportamento do objeto em resposta a eventos padrão da *Unity*. Um exemplo seria inicializar variáveis dentro do método *Start()*, que é executado no momento em que o objeto é instanciado, e aplicar lógica de movimento no método *Update()*, que é executado uma vez antes de calcular cada *frame*.

No caso de funcionalidade *multiplayer*, cria-se um *script* que deriva de comportamento de rede (*NetworkBehaviour*) para especificar o modo com que a entidade de jogador atualiza e envia informações para o servidor, e utiliza-se marcadores para especificar quais propriedades do objeto devem ser sincronizadas, em que momento devem ser repassadas para o servidor ou repercutidas pelo servidor para outros clientes conectados.

Para simular uma abordagem simplista de *multiplayer*, foi desenvolvido um código em *script* que assume condições ideais de rede e simplesmente altera a posição da entidade ao passo que recebe uma mensagem de atualização.

## 4.3 Funcionamento do dead reckoning

O código que controla a execução do DR é definido por *script*, e por simplicidade foi escolhido integrar todas as variantes citadas no projeto conceitual dentro de uma só lógica. Como resultado disso, a sua implementação possui as seguintes características:

– diferencia entre objetos parados e em movimento;

– considera aceleração apenas quando a variação de velocidade excede certo valor;

– permite uma taxa de atualização variável, enviando mensagens apenas quando certos parâmetros de tempo ou orientação são extrapolados;

– considera apenas rotação em um eixo, pois o movimento do protótipo se dá sobre um espaço perfeitamente plano;

– aplica uma função de convergência quando o estado real e estimado diferem.

A convergência entre estado real e estado estimado tem início ao se receber uma nova atualização, quando então é feita uma nova estimativa de trajeto assumindo-se o mesmo intervalo de tempo até uma próxima atualização, e assim a posição começa a ser interpolada em direção ao que se espera ser o ponto final do movimento estimado, antes da próxima mensagem. A figura 17 apresenta o método usado para se calcular o ponto final do movimento estimado.

```
//calcula proxima posicao final estimada
else
{
    long deltaTimeSinceLastUpdate = (lastUpdateSent - ((System.DateTime.Now.Ticks / System.TimeSpan.TicksPerMillisecond) - startTimestamp));
    lastUpdateInterval = deltaTimeSinceLastUpdate - lastUpdateSent;
    float lastSpeedValue = currentSpeedValue;
    currentSpeedValue = MeasureSpeed(lastKnownPosition, positionUpdate, deltaTimeSinceLastUpdate);
    accelerationValue = lastSpeedValue - currentSpeedValue;
    lastKnownPosition = positionUpdate;

    estimatedNextPosition = transform.forward * currentSpeedValue + smoothedPosition * lastUpdateInterval;

}
```

Figura 17: variáveis utilizadas no cálculo de ponto final de movimento, ao se receber nova mensagem. Variáveis com da classe *Vector3* da *Unity* são utilizadas para definir uma posição no mundo.

### 4.4 Gerenciamento de partidas e *matchmaking*

Outra função importante do *NetworkManager* é gerenciar a criação e acesso de salas de jogo pelos jogadores, através de uma interface que comporte uma lista das salas disponíveis e botões que ativem as diferentes opções. Existe uma interface de usuário padrão simples que a Unity entrega pronta para fins de exemplo e testes, chamada de *NetworkManagerHUD*, cuja função é interagir com o *NetworkManager* para visualização de opções de criação ou acesso.

Ao ser adicionada ao objeto a que está atrelado o *script* do *NetworkManager*, o script da *HUD* automaticamente faz as conexões entre seus botões e os métodos que inicializam as funcionalidades de salas de jogo do *NetworkManager*. Para usar sua própria interface, basta criar botões e implementar *callbacks* para fazer a ligação os botões e os métodos especiais do *NetworkManager*, da mesma forma que a interface padrão. São métodos dessa classe que abrem conexões, carregam as cenas referenciadas como sendo a principal *online* ou *offline*, e instanciam objetos entidade do jogador. No caso da figura 16, a interface simples com um botão "*Host LAN*" ativa o método *NetworkManager.StartHost()* através de *callback*. O método em questão inicializa um servidor do jogo localmente e se conecta a ele como cliente, além de criar e posicionar os objetos em seus locais iniciais definidos.

Embora a HLAPI ofereça uma opção de *matchmaking* através dos serviços de *Internet* específicos da *Unity*, por esse serviço necessitar de cadastro do projeto, e também passar a ser um serviço pago após extrapolar certo limite de banda, optou-se por operar estritamente dentro de rede local.

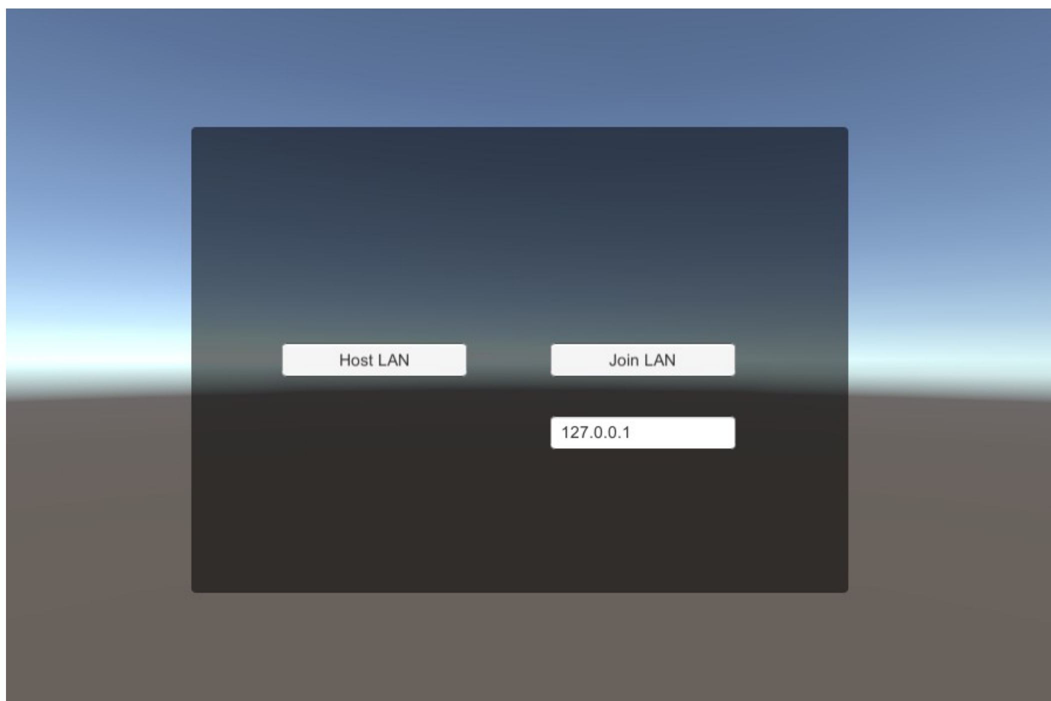


Figura 17: interface simples do protótipo que oferece opção de hospedar ou conectar em partida local.

### 4.5 Definição de testes

O caso ideal para a realização dos testes é o movimento das entidades ser manipulado obedecendo suas características, como aceleração, sem que isso interfira nas mensagens de atualização enviadas por rede. O método encontrado para automatizar o movimento da entidade do jogador foi definir um caminho através de pontos, representados por objetos vazios, adicionar esses pontos a uma lista e implementar um método que move localmente a entidade do jogador na direção de cada um dos pontos em ordem, em incrementos de posição, até que ele passe pelo último, quando pára. Durante a execução do método, o número de mensagens enviadas é contabilizado.

Foram construídos três caminhos diferentes a fim de testar a adaptabilidade do algoritmo de DR implementado: uma reta, um caminho com curvas leves e outro com curvas mais acentuadas. Para facilitar a manipulação dos elementos, escolheu-se simular ao mesmo tempo instâncias representantes das duas abordagens implementadas, mas fazer com que somente uma esteja visível em qualquer dado momento.

Para auxiliar no controle dos testes, foi criada um objeto *SimulationManager* que gerencia o processo, guarda referências aos objetos pertinentes à simulação, e permite reiniciar testes e visualizar informações através de uma interface disponível na tela do protótipo, conforme mostra a figura 18.

Figura 18: interface do protótipo com elementos para controle e acompanhamento da simulação.

## 4.6 Conceito e prática

A implementação realizada difere da solução modular apresentada no projeto conceitual. A figura 19 apresenta a relação entre os módulos planejados inicialmente e as entidades que, na prática, gerenciam suas funções.

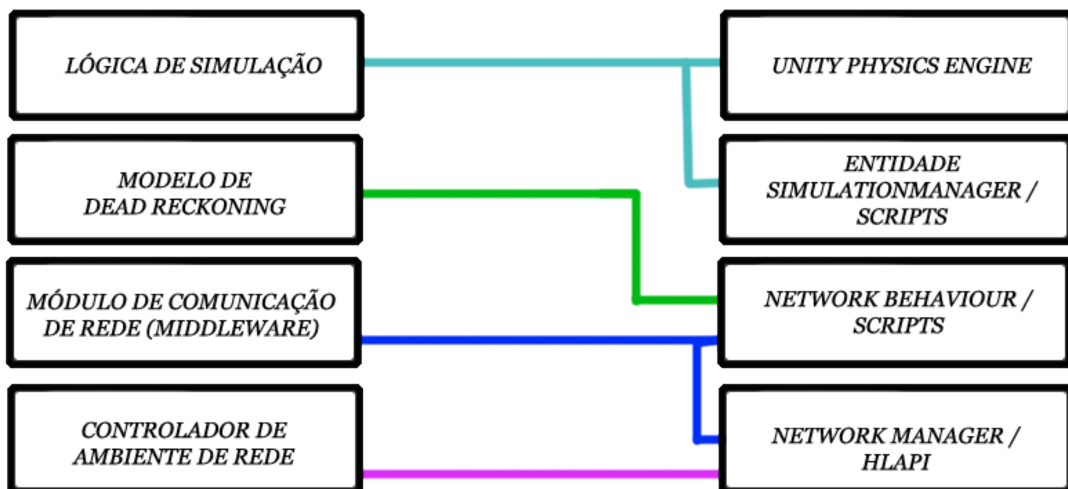



Figura 19: relação entre projeto conceitual e implementação.

O modelo de DR é definido completamente por *script* que deriva de *networkBehaviour*, contendo toda a lógica referente a entidades remotas, assim como código do tratamento da informação pelo servidor e repasse dessa informação aos clientes, que seria pertinente ao módulo de comunicação de rede. Pormenores do comportamento da rede e a comunicação inicial entre instâncias são gerenciados pela entidade *networkManager*, própria da *Unity*.

# 5 RESULTADOS

Os testes foram realizados executando duas instâncias do protótipo na mesma máquina, uma instância agindo como *host* e a outra como cliente. A escolha da taxa de atualização da abordagem simplista foi baseada no valor limite em que o movimento observado mantinha bom nível de fluidez, o que se deu no valor de vinte vezes por segundo. Já os parâmetros de controle do DR foram escolhidos após com alteração de valores em tentativa e erro, até chegar a uma resposta aceitável da aplicação. O gráfico apresentado na figura 17 mostra o número de mensagens enviadas por cada abordagem ao ser aplicado o teste usando o caminho em forma de reta.

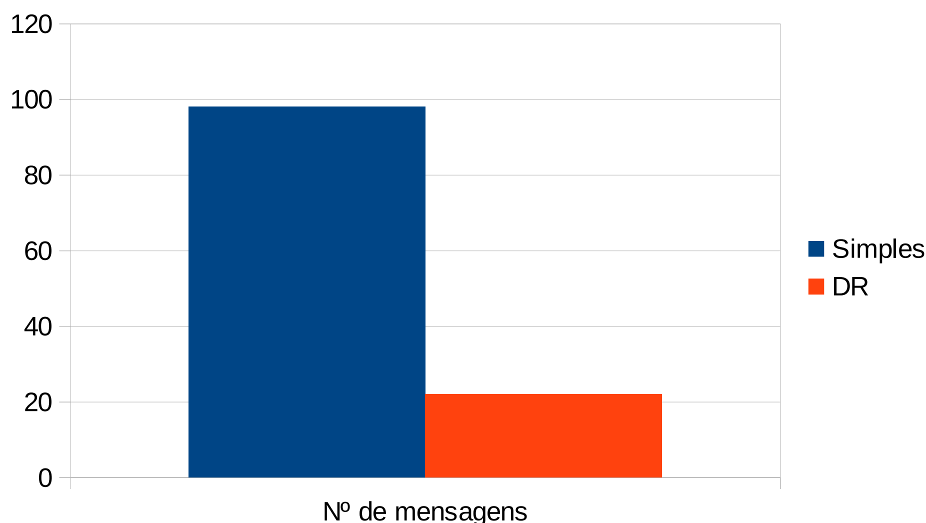



Figura 20: gráfico representando número de mensagens enviadas durante o teste que usa uma trajetória perfeitamente retilínea.

O gráfico faz sentido quando se considera que, enquanto a abordagem simplista manteve uma taxa de atualização constante por toda a duração do teste, a entidade executando DR permaneceu com direção e velocidade constantes uma vez que atingiu velocidade máxima, e portanto disparou menos atualizações.

Os gráficos a seguir representam os resultados referentes ao número de mensagens testando as abordagens, respectivamente, nos caminhos com curvas leves e com curvas acentuadas.

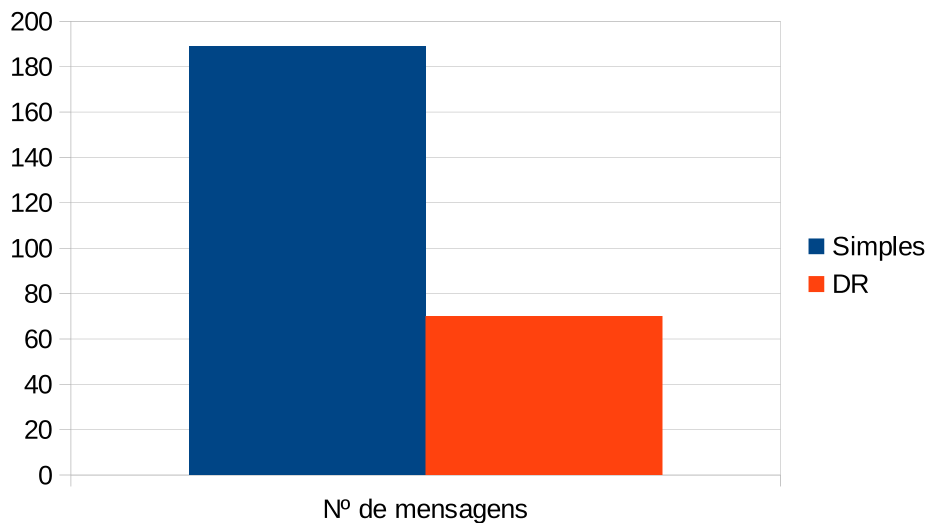



Figura 21: número de mensagens enviadas durante o teste que usa uma trajetória com curvas leves.

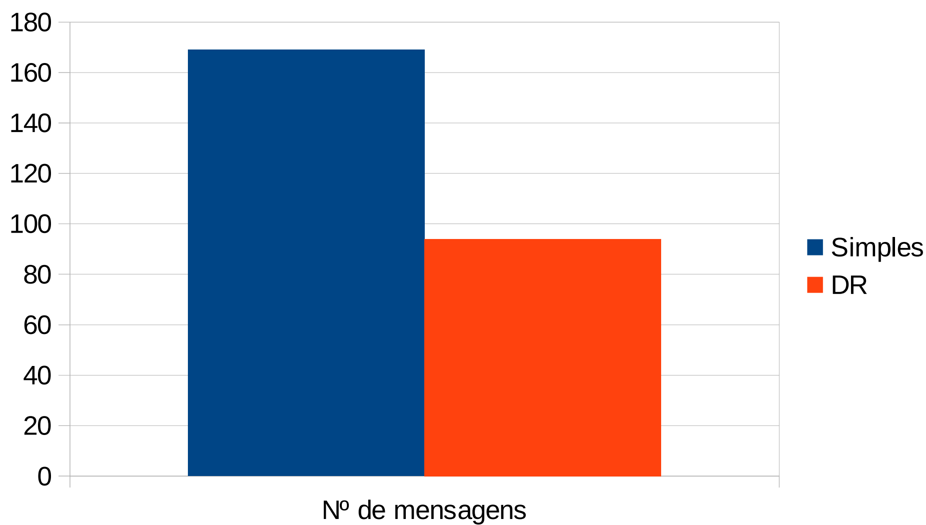



Figura 22: número de mensagens enviadas durante o teste que usa uma trajetória com curvas acentuadas.

Enquanto em uma trajetória sem curvas a implementação de DR proposta apresentou uma diferença de aproximadamente 77% em relação à abordagem simples, nos testes de trajetória curvilínea a diferença foi bem menor, chegando a 62% e em curvas acentuadas a diferença foi de apenas 45%. Apesar do grande salto entre os casos, os resultados demonstram que o DR pode economizar uma porcentagem considerável de mensagens, especialmente quando o movimento das entidades é previsível ou não sofre alterações bruscas regularmente.

Além disso, quando se simulou instabilidade na rede, como pacotes perdidos ou variações na latência, na abordagem simplista foi observada a ocorrência de saltos de posição, o que acarreta perda da consistência perceptiva. Diminuindo-se o número de mensagens por segundo na tentativa de igualar à abordagem com DR, a instabilidade da rede se torna ainda mais perceptível para o usuário. A alternativa para tornar o sistema mais robusto seria salvar as atualizações de posição em uma lista e interpolar entre elas aos poucos, com um certo atraso, como descrito no capítulo 2. Apesar de aumentar a robustez e possibilitar menor número de atualizações por segundo, essa abordagem mitiga a instabilidade da rede através da geração de atraso no sistema de atualização de posição de entidades remotas.

Em contrapartida, sob mesmas condições, a abordagem com DR apresentou menor ocorrência de saltos, mantendo consistência perceptiva mesmo com perda de pacotes e variações de latência, corrigindo sua posição com movimento coerente conforme estão definidos os parâmetros da função de convergência. Isso mostra que a consistência pode ser mantida mais fácil sem a necessidade de gerar atrasos, e mesmo que se aplicasse essa ideia à abordagem com DR visando maior robustez do sistema, seria possível definir valores de atrasos menores configurando a função de convergência para lidar com correções de posição maiores.

# 6 CONCLUSÃO

Através do estudo do tema e implementação deste trabalho foi possível compreender, comprovar e aprofundar conceitos de redes de computadores, projeto e desenvolvimento de software, em particular desenvolvimento de jogos, além de promover o aprendizado e familiarização com ferramentas e tecnologias novas.

Quando se considera uma simulação dispersa de grande porte com múltiplas entidades possivelmente se comunicando com várias simulações ao mesmo tempo, o impacto do *dead reckoning* na banda operante total pode ser consideravelmente alto. É possível acreditar que uma manipulação mais fina de parâmetros de controle, como a função de convergência ou faixas de tolerância, pode garantir desempenho em situações onde a operacionalidade da rede não é garantida. É razoável pensar que o ajuste ideal dos parâmetros dependeria muito das características da aplicação onde está inserido.

Concluiu-se também que a implementação de DR se mostrou robusta em relação a problemas na rede e que é uma abordagem válida para lidar com eles numa arquitetura cliente-servidor com servidor autoritativo.

## 6.1 Trabalhos futuros

O trabalho desenvolvido obteve resultados esperados, mas está longe de ser completo em sua proposta, pois melhorias e expansões poderiam ser adicionadas facilmente. Funcionalidades previstas no projeto conceitual do protótipo de jogo não foram incluídas priorizando os testes, e variações de testes não foram implementados por falta de tempo.

Uma possibilidade para o futuro é construir um protótipo semelhante através da LLAPI visando ter uma visão mais ampla de todo o processo, de preferência com mais tempo e um projeto mais firmemente estruturado. Outra possibilidade de projeto seria o desenvolvimento de um algoritmo de DR cujos parâmetros se autocorrigem para valores mais adequados à situação, se ajustando ao perfil da simulação.

# REFERÊNCIAS


BARRON, T. **Multiplayer game programming**. Prima Publishing, Rocklin, 2001.

BERNIER, Y. W. **Latency Compensating Methods in Client/Server In-game Protocol Design and Optimization**. 2001. Disponível em: <https://developer.valvesoftware.com/wiki/Latency_Compensating_Methods_in_Client/Server_In-game_Protocol_Design_and_Optimization>. Acessado em: 20 set. 2018.

C# Artigo da Wikipédia. **Wikipédia:** A Enciclopédia Livre, 2015. Disponivel em: <https://pt.wikipedia.org/wiki/C_Sharp>. Acesso em: 10 out. 2018.

CANNON, T. Fight the Lag! The trick behind GGPO's low-latency netcode. **Game Developer Magazine**, São Francisco, vol. 19, n. 9, Setembro 2012. Disponível em: <http://twvideo01.ubm-us.net/o1/vault/GD_Mag_Archives/GDM_September_2012.pdf>. Acesso em: 9 jul. 2014.

COULOURIS, G.; DOLLIMORE, J.; KINDBERG, T. **Sistemas Distribuídos:** Conceitos e Projeto. Tradução de João Tortello. 4ª. ed. Porto Alegre: Bookman, 2007.

FLIEDLER, G. **What every programmer needs to know about game networking**. Disponível em: <https://gafferongames.com/post/what_every_programmer_needs_to_know_about_game_networking/>. Acesso em: 20 set. 2018.

GAMBETTA, G. **Fast-Paced multiplayer**. Disponível em: <http://www.gabrielgambetta.com/client-server-game-architecture.html>. Acesso em: 20 set. 2018.

GAME Engine Wikipedia Article. **Wikipedia:** the Free Encyclopedia, 2014. Disponível em: <http://en.wikipedia.org/wiki/Game_engine>. Acesso em: 9 jul. 2014.

HAKALA, T.; MULHOLLAND, A. **Programming multiplayer games**. Wordware Publishing, Texas, 2004.

HARE, N. B. **Development and Deployment of multiplayer online games**. ITHare.com Website GmbH, Áustria, 2015-2017.

HARRIS, T. Como Funciona a Formatação de Videos. **HowStuffWorks**, 2000. Disponível em: <http://lazer.hsw.uol.com.br/formatacao-de-videos.htm>. Acesso em: 9 jul. 2014.

INSTITUTE OF ELECTRICAL AND ELETRONICS ENGINEERS COMPUTER SOCIETY. Std. 1278.1-2012. **Standard for Distributed Interactive Simulation** – Application Protocols. New York, 2012.

KHAN, A.M.; CHABRIDON, S.; BEUGNARD, A. **Synchronization Medium: A Consistency Maintenance Component for Mobile Multiplayer Games**. NetGames ‘07, set. 19-20, 2007.

MAUVE’S RAMBLINGS. **Understanding fighting game networking**. Jul. 2012. Disponível em: <http://mauve.mizuumi.net/2012/07/05/understanding-fighting-game-networking>. Acesso em: 9 jul. 2014.

MMORPG Artigo da Wikipédia. **Wikipédia:** A Enciclopédia Livre, 2014. Disponível em: <http://pt.wikipedia.org/wiki/MMORPG>. Acesso em: 20 jul. 2014.

OPENGL Wikipedia Article. **Wikipedia**: the Free Encyclopedia, 2014. Disponível em: <http://en.wikipedia.org/wiki/OpenGL>. Acesso em 20 set. 2014.

PING Artigo da Wikipédia. **Wikipédia:** a Enciclopédia Livre, 2014. Disponível em: <http://pt.wikipedia.org/wiki/Ping>. Acesso em: 9 jul. 2014.

POSTEL, J. RFC 768. **Internet Engineering Task Force**, 1980. Disponível em: <http://tools.ietf.org/html/rfc768>. Acesso em: 9 jul. 2014.

SHIRKY, C. OpenP2P. **What's P2P and what's not**, 2000. Disponível em: <http://scripting.com/davenet/2000/11/15/clayShirkyOnP2p.html>. Acesso em: 20 set. 2018.

TANENBAUM, A. S.; STEEN, M. V. **Distributed System:** Principles and Paradigms. Upper Saddle River: Prentice Hall, 2002.

TERRANO, M.; BETTNER, P. **1500 Archers on a 28.8:** Network Programming in Age of Empires and Beyond. Mar. 2001. Disponível em: <http://www.gamasutra.com/view/feature/3094/1500_archers_on_a_288_network_.php>. Acesso em: 20 set. 2018.

TCP/IP Artigo da Wikipédia. **Wikipédia:** A Enciclopédia Livre, 2014. Disponivel em: <http://pt.wikipedia.org/wiki/TCP/IP>. Acesso em: 9 jul. 2014.

UNITY. **Unity Homepage**. Disponível em: <https://unity3d.com/pt>. Acesso em: 25 Set. 2018.

USER Datagram Protocol Artigo da Wikipedia. **Wikipédia:** A Enciclopédia Livre, 2014. Disponivel em: <http://pt.wikipedia.org/wiki/Protocolo_UDP>. Acesso em: 9 jul. 2014.

VISUAL STUDIO. **Visual Studio IDE, Code Editor, VSTS & App center** – Visual Studio. Disponível em: <https://visualstudio.microsoft.com/>. Acesso em: 25 Set. 2018.

XU, J.; WAH, B.W. **Concealing network delays in delay-sensitive online interactive games based on just-noticible differences**. Multimedia and Expo (ICME), 2013 IEEE International Conference, 15-19 jul. 2013.