UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

RODRIGO HOLZTRATTNER REIS

Diminuição de Latência em Jogos Multijogador Utilizando Conexões Auxiliares P2P

Monografia apresentada como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação

Orientador: Prof. Dr. Claudio Fernando Resin Geyer

Porto Alegre
2018

“It is not true that people stop pursuing dreams because they grow old;
they grow old because they stop pursuing dreams.“

— GABRIEL GARCÍA MÁRQUEZ

AGRADECIMENTOS

Agradeço a todos que estiveram comigo durante todos esses anos, que participaram ativamente da minha vida permitindo que eu tivesse oportunidade de crescer e sempre buscar mais e o melhor que pude. Agradeço principalmente aos meus pais que foram responsáveis por toda a minha educação tanto acadêmica quanto pessoal, pilares essenciais em minha vida.

Agradeço também ao professor Claudio Fernando Resin Geyer, meu orientador, o qual me acompanhou durante todo o processo de construção deste trabalho e foi um grande apoio na realização do mesmo.

Meus sinceros agradecimentos também à Fabiana Cecin por toda ajuda e por cada conselho. O seu suporte foi essencial para que essa monografia fosse possível.

Gostaria de agradecer imensamente a todos os professores e funcionários do INF por me atenderem sempre em prontidão para esclarecimento de qualquer dúvida ou problema. Foi uma enorme satisfação e honra fazer parte desta instituição.

RESUMO

Uma das grandes dificuldades que se encontra na construção de um jogo multijogador é em como será feita a conexão entre tantos possíveis jogadores. Atualmente existem diversas formas de se realizar essa conexão mas cada uma delas possui, dentre suas características, grandes problemas com latência na troca de mensagem caso existam problemas na rede ou caso os pontos de comunicação estejam localizados muito longe um do outro. A proposta desenvolvida aqui é sugerir uma alternativa para o problema descrito, usando a conexão clássica cliente-servidor em conjunto com conexões auxiliares P2P, as quais possuem como objetivo reduzir o efeito da latência entre os jogadores presentes.

Palavras-chave: P2P. game. multiplayer.

Latency Decrease in Multiplayer Games Using Auxiliary P2P Connections

ABSTRACT

One of the major difficulties while designing a multiplayer game is how the connection between so many players will be established. There are several ways to make this connection but each one of them has, among its characteristics, great problems with latency in the message exchange if there are issues in the network or the communication points are located far from each other. The current proposal is to suggest an alternative to the described problem, using the classic client-server connection in conjunction with P2P helper connections, which aim to reduce the latency effect among present players.

Keywords: P2P, game, multiplayer.

LISTA DE FIGURAS

Figura 3.1 Separação das Camadas da abordagem Híbrida 18

Figura 4.1 Fluxo principal parte 1 25
Figura 4.2 Fluxo principal parte 2 26
Figura 4.3 Mensagem Invalidada 27
Figura 4.4 Perda Pacote sem Confirmação 28
Figura 4.5 Perda Pacote de Atualização 29
Figura 4.6 Perda Pacote de Requisição 30

Figura 5.1 Clientes e Servidor de mesma Latência (em milissegundos) 32
Figura 5.2 Clientes próximos e Servidor Distante (em milissegundos) 34
Figura 5.3 Clientes Distantes e Servidor no Meio (em milissegundos) 36
Figura 5.4 Múltiplos Clientes e um Servidor (em milissegundos) 37
Figura 5.5 Teste com Perda de Pacote 39

LISTA DE TABELAS

Tabela 5.1 Clientes e Servidor de mesma Latência - Primeira Medida 33
Tabela 5.2 Clientes e Servidor de mesma Latência - Segunda Medida 33
Tabela 5.3 Clientes e Servidor de mesma Latência - Comparação 33
Tabela 5.4 Clientes próximos e Servidor Distante - Primeira Medida.....34
Tabela 5.5 Clientes próximos e Servidor Distante - Segunda Medida.....35
Tabela 5.6 Clientes próximos e Servidor Distante - Comparação 35
Tabela 5.7 Clientes Distantes e Servidor no Meio - Primeira Medida.....35
Tabela 5.8 Clientes Distantes e Servidor no Meio - Segunda Medida.....36
Tabela 5.9 Clientes Distantes e Servidor no Meio - Comparação.....37
Tabela 5.10 Múltiplos Clientes e um Servidor - Primeira Medida 38
Tabela 5.11 Múltiplos Clientes e um Servidor - Segunda Medida.....38
Tabela 5.12 Múltiplos Clientes e um Servidor - Comparação 38
Tabela 5.13 Teste com Perda de Pacote - Primeira Etapa 40
Tabela 5.14 Teste com Perda de Pacote - Resultado Primeira Etapa 40
Tabela 5.15 Teste com Perda de Pacote - Segunda Etapa 41
Tabela 5.16 Teste com Perda de Pacote - Resultado Segunda Etapa 41

LISTA DE ABREVIATURAS E SIGLAS

MMO	Massively Multiplayer Online
RPG	Role Playing Game
TCP	Transmission Control Protocol
P2P	Peer-to-Peer
IETF	Internet Engineering Task Force
RFC	Request for Comment
IDE	Integrated Development Environment
API	Application Programming Interface
UFRGS	Universidade Federal do Rio Grande do Sul

SUMÁRIO

1 INTRODUÇÃO 11
1.1 Motivação 12
1.2 Objetivo 12
1.3 Organização do texto 13
2 CONCEITOS E DEFINIÇÕES RELACIONADAS 14
2.1 Cliente-Servidor 14
2.2 P2P 15
2.3 TCP 15
2.4 UDP 16
3 TRABALHOS RELACIONADOS 17
3.1 Distributing game instances in a hybrid client-server/P2P system to support MMORPG playability 17
3.2 FreeMMG: Uma arquitetura cliente-servidor e par-a-par de suporte a jogos maciçamente distribuídos 18
3.3 Latency Compensating Methods in Client/Server In-game Protocol Design and Optimization 18
4 PROPOSTA DO MODELO 20
4.1 Projeto Share Pack 20
4.2 Definições das Entidades e Objetos 21
4.2.1 Cliente 21
4.2.2 Servidor 21
4.2.3 Trapaça 22
4.2.4 Pack de Requisição 22
4.2.5 Pack sem Confirmação 23
4.2.6 Estado não Confirmado 23
4.2.7 Esfera de Influência 23
4.2.8 Mudança de Estado 24
4.3 Funcionamento Principal 24
4.4 Casos de Funcionamento Alternativos 26
4.4.1 Mudança de Estado Inválida 26
4.4.2 Perda de Pacotes 27
5 TESTES E VALIDAÇÃO 31
5.1 Detalhamento do Modelo de Teste 31
5.2 Resultados dos Experimentos 32
5.2.1 Clientes e Servidor com mesma Latência 32
5.2.2 Clientes Próximos e Servidor Distante 33
5.2.3 Clientes Distantes e Servidor no Meio 35
5.2.4 Múltiplos Clientes e um Servidor 37
5.2.5 Teste com Perda de Pacotes 39
6 CONCLUSÃO 43
6.1 Trabalhos Futuros 44
REFERÊNCIAS 45

1 INTRODUÇÃO

Atualmente a grande maioria dos jogos eletrônicos lançados no mercado possui suporte ao mundo multijogador, de fato a maior parte deles é totalmente fundada nessa modalidade (VALDES, 2016). Este fato tem uma razão: vivemos em uma era em que a comunicação se extende de forma global e cada vez mais os jogadores desejam compartilhar seus momentos e suas conquistas ao lado de amigos, conhecidos ou mesmo estranhos (GULLIFORD, 2016). Jogos que seguem essa tendência são muitas vezes os que melhor sobrevivem ao longo dos anos, geralmente estes recebem uma taxa muito maior de atualizações e modificações de forma a garantir a permanência de seus jogadores e migrar para áreas diferentes atraindo assim diversos nichos.

Um dos motivos principais para os desenvolvedores escolherem construir os seus jogos desta forma é a capacidade de lucrar não só com a venda inicial do jogo (que muitas vezes nem ocorre neste caso) mas sim com diversas outras opções como: mensalidades, venda de itens dentro do jogo, cosméticos, etc. (WALLACE, 2005). Muitas dessas opções só são possíveis nesse ambiente em que existe contato com outros jogadores, como por exemplo na compra de cosméticos, o jogador sente vontade de ser visto, de ser notado por outros jogadores, de ser diferente (NOORDZIJ, 2015).

Em conjunto com toda essas possibilidades, diversos problemas também acabam aparecendo e entre eles um dos principais definitivamente é o problema do *lag*, o atraso entre a realização de uma ação pelo jogador e a validação da mesma pelo servidor (MASTIN, 2016), esse fato muitas vezes acaba sendo decisivo na hora de convencer um novo jogador a continuar jogando ou mesmo garantir que uma grande quantidade de players permaneça *online*. Conforme *Mastin* (2015) constata, jogadores dessa categoria são duas vezes mais propensos a abandonar um jogo caso experimentem uma demora de mais de 500 milissegundos adicionais. Em geral o jogador espera que suas ações sejam interpretadas em tempo real e se sente desconfortável com a sensação de atraso.

Ao longo dos anos, várias formas de resolver esse problema de atrasos foram desenvolvidas, atualmente a grande maioria dos jogos utiliza um sistema *cliente-servidor* onde o jogador envia informações a um servidor autoritário que realiza uma validação, verificando se a ação tomada pelo jogador está de acordo com o seu estado, retornando assim uma resposta para o cliente. Este modelo implica que o tempo de resposta é no minimo maior do que o intervalo de comunicação entre o próprio cliente e o servidor, fator que muitas vezes é agravado quando a distância física entre os dois pontos da comu-

nicação é grande ou existe algum problema na rede que acaba gerando falha e/ou atrasos (FIEDLER, 2010). Existem alternativas que buscam aproveitar a localidade; tais fazem uso de comunicação entre clientes (P2P) ou o uso de diversos servidores localizados geograficamente de forma distribuida, mas de qualquer forma cada uma delas acaba tendo o seu *trade-off* que muitas vezes só vale a pena se o jogo for modificado para aceitar determinadas condições (fato que nem sempre é fácil de se fazer), o que não é algo agradável aos olhos de um desenvolvedor por muitas vezes limitar o escopo do projeto.

1.1 Motivação

Conforme já relatado, um dos maiores problemas enfrentado por desenvolvedores de jogos multiplayer atualmente é o *lag*. Este é um dos fatores importantes que jogadores utilizam para decidir se continuam jogando ou não pois o efeito desse atraso é muitas vezes não prazeroso (STRINGER, 2015). Por ser extremamente importante, muito se tem feito para resolver ou mascarar tal problema, mas infelizmente a solução geralmente acaba possuindo diversos *trade-offs*.

Levando em conta todos os fatos apresentados, existe um grande espaço para a busca de novas soluções alternativas que aproximem o jogador de uma experiência plena sem *lag*. Com o tempo diversas soluções hibridas começaram a aparecer e a se tornar populares (HOOK, 2006) mas ainda sim estas necessitam de grandes adaptações.

1.2 Objetivo

Este trabalho apresenta a elaboração teórica de um modelo de comunicação para jogos multijogador em conjunto com o desenvolvimento de uma biblioteca. Esta implementa uma sessão reduzida do modelo teórico aqui apresentado e foi utilizada para a validação da ideia em um ambiente de teste onde se verificou a redução do atraso na comunicação.

1.3 Organização do texto

O trabalho está organizado nos seguintes capítulos que nortearam os assuntos relevantes ao processo de construção da solução:

Capítulo 1. Corresponde a esta introdução que apresentou a motivação que levou a construção desta solução, os objetivos a serem tratados durante o trabalho e as escolhas tecnológicas utilizadas.

Capítulo 2. Apresenta os conceitos e as tecnologias utilizadas para o desenvolvimento da ideia proposta.

Capítulo 3. Apresenta trabalhos relacionados com o tema proposto e diversas outras soluções existentes que também tentam solucionar o problema aqui abordado.

Capítulo 4. Apresenta a modelagem teórica do sistema proposto, detalhamento do funcionamento deste modelo em forma de um pseudo-algoritmo e uma avaliação das diversas qualidades esperadas do mesmo. Descreve o ambiente de teste utilizado, assim como uma breve descrição da *library* que foi construida e utilizada para a realização dos testes, aborda os resultados obtidos por estes e realiza comparações de forma a validar as qualidades esperadas.

Capitulo 5. Apresenta a conclusão do trabalho e algumas considerações finais, como pontos a melhorar e a serem explorados futuramente.

2 CONCEITOS E DEFINIÇÕES RELACIONADAS

Neste capítulo são apresentados os conceitos e as tecnologias utilizadas para o desenvolvimento da ideia proposta. A escolha dessas ferramentas se deu de acordo com a sua utilidade no desenvolvimento de soluções para os diversos problemas existentes no modelo multi-jogador, suas características relativas aos atributos da proposta e sua relevância de acordo com o resultado esperado.

Como a solução proposta caracteriza-se por sugerir uma nova abordagem de comunicação entre clientes e servidores, utilizaremos técnicas de troca de mensagens pela internet envolvendo múltiplos protocolos de comunicação, conceitos de arquiteturas como cliente-servidor e P2P também serão utilizados no desenvolver da solução. A questão de manter os estados (dos clientes e do servidor) sincronizados é de extrema importância para garantir o bom funcionamento do sistema, ideias que exploram tanto a inserção desse sincronismo quanto tratamentos nos casos de ausência do mesmo serão apresentados.

2.1 Cliente-Servidor

O termo *cliente-servidor* refere-se a um modelo bem conhecido de *networking*, geralmente aqui o cliente realiza requisições e/ou envia informações ao servidor que, ao recebê-las, realiza uma serie de validações e responde de forma adequada (ROUSE, 2008). Esse é um modelo de arquitetura que descreve os sistemas em termos de desempenho de tarefas computacionais e de comunicação. Quando referenciamos este modelo no mundo dos jogos digitais, diversas caracteristicas surgem, dentre elas o servidor passa a ser a base central para se manter o estado do jogo, provendo assim informações para outros cliente de forma a manter o estado destes o mais próximo do global.

O servidor aqui caracteriza-se por ser autoritário, ou seja, toda requisição vinda de um cliente é tratada como uma tentativa de mudança de estado que, a qualquer momento, pode ser negada por diversas razões sendo que neste caso é dever do cliente lidar com isso (FIEDLER, 2010). Esse modelo, se implementado corretamente, não permite que clientes inventem um estado ou realizem mudanças inválidas, impossibilitando diversas trapaças comuns em jogos multi-jogador, provendo assim uma camada de segurança inicial que faz com que esta seja a escolha mais comum principalmente em jogos do tipo *MMO*[1] (LUARD, 2015).

[1] *Massive Multiplayer Online*

2.2 P2P

Neste outro modelo de *networking*, todo os nodos que se encontram dentro de uma rede são tratados como *peers*, implicando que as conexões são realizadas de forma direta entre os nodos sem a necessidade de um servidor principal, ou mesmo categorizando cada um desses nodos como servidores autônomos (JOHNSON, 2017). Aqui como foi dito não existe a figura de um servidor principal e autoritário. Quando nos referenciamos a este modelo no mundo de *games* entende-se que cada nodo representa um cliente e que cada cliente precisa ter (preferencialmente) a mesma imagem do estado que os outros na mesma rede possuem, nada impede que um desses nodos faça suposições a respeito de seu estado de forma a desvencilhar-se do estado global.

Toda a comunicação, como já foi mencionado, ocorre entre os *peers* de forma que qualquer troca de mensagem não precise passar antes por um nodo central (servidor), a comunicação é direta e por isso tende a ser mais rápida (WATERS, 2002). Este modelo não fornece grandes atributos de segurança implícitos (quando utilizado sozinho, sem uma camada acima que adicione tais elementos) porque a qualquer momento um cliente pode querer fornecer informações falsas, fato que poderia gerar trapaça ou criar um estado alternativo ao estado global em um jogo multi-jogador (FROSTBYTE, 2017).

2.3 TCP

O *TCP*[2] é um protocolo de comunicação na internet orientado à conexão que realiza a troca de informações através do envio de pacotes de dados, este protocolo é definido pela *IETF*[3] no documento *RFC*[4] de número 793 (ROUSE, 2014).

A forma adotada por esse protocolo faz com que uma conexão seja criada quando se deseja realizar comunicação entre 2 nodos, tal conexão é mantida até que um dos envolvidos finalize-a ou ocorra algum problema que cause o fechamento da mesma. Entre os atributos principais implementados naturalmente por esse protocolo se destacam a garantia de entrega e o ordenamento, ou seja, todas as mensagens enviadas possuem garantia que em algum momento serão recebidas e além disso a ordem de recebimento será igual a de envio.

[2] *Transmission Control Protocol*
[3] *Internet Engineering Task Force*
[4] *Request for Comment*

O custo de uso do *TCP* se justifica quando utilizado em *jogos* multi-jogador que não realizam sua simulação em tempo real ou não necessitam de uma interação extremamente frequente, tolerando até certo ponto uma latência controlada que garante diversas qualidades a respeito da conexão (FIEDLER, 2014).

2.4 UDP

O *UDP*[5] é um protocolo de comunicação na internet usado primeiramente para estabelecer conexões de baixa latência e que toleram perda de dados, o envio de informações se dá através de datagramas, um agrupamento de dados enviado de uma só vez, como um bloco (ROUSE, 2015). Fora o encapsulamento do datagrama pelo cabeçalho do *UDP*, nenhum outro processamento é realizado pelo protocolo, fazendo com que o envio da mensagem seja mais rápido quando comparado com o *TCP*.

Ao mesmo tempo que o ganho se dá pela baixa latência de envio, a perda aparece na forma de que não existe garantia de entrega, muito menos de ordenamento, a aplicação é responsável por isso. Não são realizados controle de fluxo e diversas outras características que por exemplo o *TCP* faz são inexistentes.

O *UDP* é amplamente utilizado no mundo de jogos multi-jogador pois é da sua natureza interferir quase que de forma insignificável na latência de processamento (pré-processamento do pacote a ser enviado), quando determinado jogo necessita que certos atributos de controle e segurança estejam presentes, tais adições podem ser implementadas por uma camada acima, geralmente em *software*, fornecendo assim a caracteristica desejada (junto com o incremento do seu custo) sem a necessidade de arcar com todo o conjunto que o *TCP* possui implicitamente (FIEDLER, 2014).

[5] *User Datagram Protocol*

3 TRABALHOS RELACIONADOS

Neste capítulo são apresentados trabalhos que estão relacionados com a proposta sugerida nesta monografia. Boa parte da ideia proposta teve um baseamento nesse trabalhos na forma de inspiração e observação dos resultados.

3.1 Distributing game instances in a hybrid client-server/P2P system to support MMORPG playability

(BARRI; ROIG; GINé, 2013)

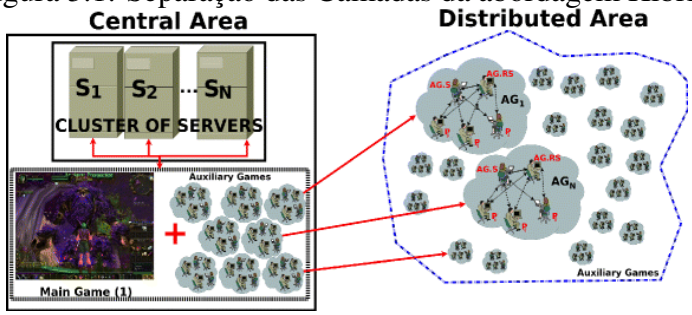
Esta solução utiliza uma solução híbrida para tentar reduzir diversos problemas comuns a jogos multi-jogador, entre eles o problema da latência. A ideia principal abordada é a divisão do jogo em diversas camadas (conforme imagem 3.1), cada uma atuando de forma bem definida em conjunto com as outras.

Como camada principal, existe um conjunto de servidores que são responsáveis por serem a base estrutural do jogo, cada um deles controla uma parte do jogo principal e alguns jogos auxiliares (definido mais adiante). Quando um jogador se conecta ao jogo, é com um desses servidores que ele começa a se comunicar (aqui o modelo de conexão utilizado é o cliente-servidor).

Existem jogos auxiliares que são instâncias executadas independentemente do jogo principal, necessitando do uso de servidores auxiliares. Para criar um servidor auxiliar, um servidor principal seleciona (de acordo com certas preferências) um cliente dentre os que irão participar desse jogo auxiliar, tornando-o uma espécie de arbitro. Cada jogo auxiliar utiliza conexões P2P nesse modo, permitindo que várias etapas de verificações, como por exemplo a checagem de trapaças, que normalmente existem no modelo cliente-servidor não sejam necessárias.

Essa solução acaba limitando de certa forma a estrutura do jogo, fazendo com que ele tenha que ser dividido em várias etapas, fator que não necessariamente funcionaria para todo tipo de jogo. Embora existam várias limitações, o uso de conexões P2P mostrou ganhos consideráveis, detalhe que pode ser visto melhor no próprio artigo.

Figura 3.1: Separação das Camadas da abordagem Híbrida



Fonte: (BARRI; ROIG; GINé, 2013)

3.2 FreeMMG: Uma arquitetura cliente-servidor e par-a-par de suporte a jogos maciçamente distribuídos

(CECIN, 2005)

Este artigo apresenta um modelo híbrido, cliente-servidor e par-a-par, de suporte para *MMOs* de estratégia em tempo real. A ideia básica aqui é a delegação de tarefas do servidor principal para outros clientes.

Por conta da delegação de tarefas, boa parte do processamento pesado que normalmente deve ser realizado pelo, é dividido para certos clientes, resultando em uma economia considerável de uso de processamento pelo lado do servidor. Os clientes são escolhidos de acordo com o poder de processamento de suas maquinas, logo, nem todo cliente será utilizado nesta técnica.

Justamente por utilizar menos processamento, essa abordagem permite que seja utilizado maquinas não tão robustas como servidores. Além disso os resultados de escalabilidade obtidos foram muito promissores, pois mostram que o tráfego gerado entre servidor e clientes, é significavelmente menor se comparado com uma alternativa puramente cliente-servidor.

3.3 Latency Compensating Methods in Client/Server In-game Protocol Design and Optimization

(BERNIER, 2015)

Ambientando-se em uma arquitetura cliente-servidor, este artigo sugere métodos para a compensação de latência pelo lado do servidor.

Existem vários problemas relacionados a sincronização de ações em um jogo multi-jogador, tais problemas geralmente ocorrem por causa da demora para receber atualizações de outros jogadores, por exemplo, em um jogo hipotético, um determinado jogador A pode pensar que outro, o jogador B, está em certa posição e por isso realiza um "ataque em área"naquele lugar, mas por causa da latência o jogador A não percebeu a tempo que o jogador B havia se movimentado, fazendo com que o ataque deferido tivesse errado o alvo.

A principal técnica abordada é a de realizar as simulações e verificações do servidor "voltando no tempo", ou seja, guardar um breve histórico sobre as recentes atualizações e os seus horários de acontecimentos, dessa forma qualquer cálculo de ação deve levar em conta o estado os jogadores no momento que a mesma foi realizada.

4 PROPOSTA DO MODELO

Este capítulo realiza o desenvolvimento e apresentação da solução aqui proposta, denominada de *Share Pack*, faz também a introdução dos conceitos existentes e fundamentais sobre o funcionamento desta ideia, formando uma base que será utilizada no capítulo 5 para a realização dos testes.

4.1 Projeto Share Pack

O projeto *Share Pack* consiste no conjunto de funcionalidades básicas esperadas que uma implementação da solução apresentada aqui deve ter. Ele define o funcionamento padrão de todo o sistema de comunicação entre os pontos conectados e tem como objetivo fornecer uma alternativa ao problema principal que estamos tratando, o do *lag*.

O tratamento desta solução se baseia no mundo dos jogos digitais multi-jogador, ela tem como principal ideia o uso da comunicação *Cliente-Servidor* em conjunto com uma rede *P2P*[1] entre os clientes conectados. Nessa rede, qualquer atualização de estado enviada para o servidor por um cliente "A"também será refletida para outros clientes "B, C, D ..."que estejam em uma certa *esfera de influência* (a ser determinada mais adiante), de forma que dependendo de certas condições isso possa proporcionar uma atualização mais rápida de estado dos outros clientes "B, C, D ..."sem que estes precisem esperar tal atualização por uma mensagem do servidor autoritário.

Aqui serão definidos o contexto e as entidades participantes desse *framework* assim como o comportamento esperado entre os mesmos. Teremos no final uma implementação básica desse modelo em um ambiente de teste controlado e discretizado de forma a validar certos pontos da solução proposta.

O contexto usado para a descrição é um mundo multi-jogador, onde cada jogador comanda um avatar que pode se movimentar livremente em um mapa 3D e realizar interações/ações. Esse contexto foi escolhido para simplificar a explicação mas a mesma ideia se aplica a outros modelos de jogos de uma forma generalizada em conformidade com as caracteristicas de cada um deles.

Se entende como mensagem qualquer envio de dados seja este por uma conexão *TCP* ou *UDP*. A solução proposta não impõe restrições à forma escolhida para realizar a comunicação entre os pares que participem do modelo.

[1] *Peer-To-Peer*

4.2 Definições das Entidades e Objetos

Por entidade têm-se os pares (nodos) da comunicação, quem realmente tem uma participação ativa com trocas de mensagens e processamento das mesmas. Por objeto defini-se os contextos das mensagens trocadas e os estados das entidades.

4.2.1 Cliente

O Cliente é a principal entidade desse sistema pois ele, além de manter uma comunicação ativa com o servidor informando-o de qualquer mudança de estado, é responsável por se comunicar com outros clientes que estejam em uma certa esfera de influência e interpretar as mensagens recebidas destes.

Um Cliente (no contexto atual) representa um jogador, que através de comandos realiza ações com o seu personagem. Sempre que uma destas ações acaba sendo considerada como *mudança de estado*, uma *mensagem de requisição* é enviada ao servidor de forma a ser validada, garantindo que o servidor atualize a visão global que ele tem do jogo, propagando a atualização a outros jogadores e garantindo no caso da não-validação que o cliente não possa realizar certos tipos de trapaça.

4.2.2 Servidor

O Servidor é a entidade responsável por gerenciar a visão global do mundo multijogador, ele sempre está conectado com todos os jogadores ativos e tem como uma das suas principais funções processar qualquer *requisição* recebida. Uma requisição, por se tratar de uma mudança de estado, deve ser validada perante o estado global que o servidor se encontra no momento, ou seja, o servidor verifica se ela pode ser realizada pelo cliente pois esse possui todos os atributos necessários para tal mudança de estado. Após o recebimento e processamento de uma requisição, o servidor responde ao respectivo cliente autor com uma mensagem de resposta de acordo com o resultado da validação.

Sempre que uma validação tem seu resultado positivo, o Servidor também verifica quais outros Clientes fazem parte da esfera de influência do Cliente requisitante, enviando então uma mensagem de atualização para estes a fim de propagar as mudanças realizadas.

O Servidor também possui a função de verificar qualquer desconectividade, quando

um cliente resolve sair do jogo ou perde a conexão por alguma causa externa, realizar *pings* periódicos (enviando mensagens de controle ao cliente para medir o tempo médio de envio de mensagens) e fornecer a cada cliente uma lista de outros clientes que se encontrem em sua esfera de influência.

4.2.3 Trapaça

Trapaça aqui se caracteriza por qualquer vantagem que um jogador possa obter de forma inadequada em relação aos outros jogadores, fator que muitas vezes, caso ocorra com frequência, faz com que o jogo perda *status* e acabe perdendo *players* fiéis.

Como o sistema da solução apresentada aqui tem como base a estrutura *Cliente-Servidor*, acabamos evitando diversas formas de trapaça de forma intrínseca, um cliente não pode mentir em relação ao seu estado pois o servidor sabe qual é o estado verdadeiro que cada cliente possui e não deixa que uma informação falsa se espalhe.

Outras formas de trapaça ficam a cargo da implementação do jogo e de outros aspectos, a única forma possível de um jogador, no modelo atual, causar uma certa forma de trapaça é ele mentir sobre o seu estado para outros jogadores (enviando informações falsas nas conexões *P2P*) fazendo com que por um breve momento outros jogadores fiquem dessincronizados, fato que logo acabará sendo corrigido por alguma mensagem do servidor. Esse tipo de problema pode ser facilmente evitado caso exista um controle do confiança entre os jogadores de forma que se um cliente "mente"muito, ele passa a não ser confiável e temporariamente as suas atualizações (pela rede *P2P*) sejam suspensas.

4.2.4 Pack de Requisição

Se entende como *Pack de Requisição* toda a informação de mudança de estado que um jogador realiza e envia para o servidor, ficando no aguardo de uma resposta. Essa mesma mensagem é transmitida para outros clientes se transformando em um *Pack sem Confirmação*.

4.2.5 Pack sem Confirmação

Todo *Pack sem Confirmação* tem origem de uma atualização recebida de outro jogador e é interpretada da forma que o cliente melhor decidir. Existem diversas formas de se beneficiar dessa informação extra, o jogador pode por exemplo, ao receber a informação de movimentação de outro *player*, já atualizar a nova posição, tendo assim um ganho parcial em cima da latência existente.

É muito importante que toda ação que for realizada em cima desse *Pack sem Confirmação* tenha como ser desfeita, pois o jogador, enquanto não receber uma atualização do servidor, permanecerá em um *Estado não Confirmado*.

4.2.6 Estado não Confirmado

Esse estado começa quando um cliente recebe um *Pack sem Confirmação* de outro jogador e encerra ou por uma mensagem de atualização do servidor, confirmando ou não a atualização recebida, ou por *timeout*, sendo que nos casos em que o *timeout* ou a não confirmação forem o resultado final, o cliente fica responsável por desfazer qualquer ação feita em cima do *Pack sem Confirmação* ou ao menos mascarar qualquer erro que possa ter sido cometido.

4.2.7 Esfera de Influência

A *Esfera de Influência* de um jogador engloba todos os outros jogadores que têm participação do estado do primeiro, de forma simples: qualquer jogador que esteja influenciando outro, seja por simplesmente ser visível ou por estar realizando alguma ação que influencie este primeiro jogador, pode ser considerado como pertencente a esta esfera. A ideia desse conceito depende muito estilo de jogo em questão, mas em geral é bem definida por: todos os jogadores que o servidor precisa repassar informações de atualização referente à outro jogador.

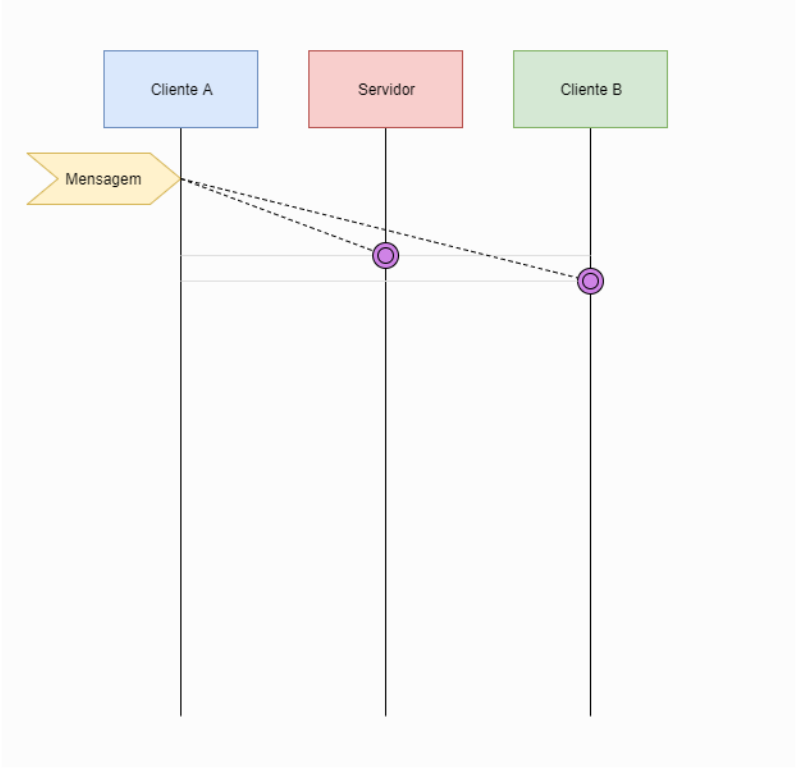
4.2.8 Mudança de Estado

Qualquer atualização realizada pelo cliente que tenha interferência com o espaço multi-jogador é considerada uma mudança de estado. Em um jogo podemos ter diversas representações destas sendo que na maioria das vezes tal acontecimento necessita ser repassado ao Servidor, garantindo assim a sincronização do estado global (visão que o servidor tem sobre o mundo multi-jogador) e o local (visão que cada cliente tem sobre o mundo multi-jogador). Exemplos de mudança de estado podem ser vistos quando o jogador realiza alguma ação como se movimentar, atacar, usar algum item. Claro que dependendo, tal ação pode não causar uma interferência externa, mas isso é totalmente depente do tipo de jogo e a forma que o mesmo foi implementado.

4.3 Funcionamento Principal

Considerando o mundo multi-jogador descrito em conjunto com uma série de clientes conectados a um servidor central, sendo consequentemente nossa principal forma comunicação. O caminho mais otimista possível ocorre quando um dos clientes (o jogador A para facilitar o exemplo) realiza uma mudança de estado, realizando o envio dessa atualização ao servidor na forma de um *pack de requisição* e, para os clientes que estiverem em sua *esfera de influência* (que será representado pelo cliente B), envia um *pack sem confirmação*, de acordo com a figura 4.1.

Figura 4.1: Fluxo principal parte 1



Fonte: Autor

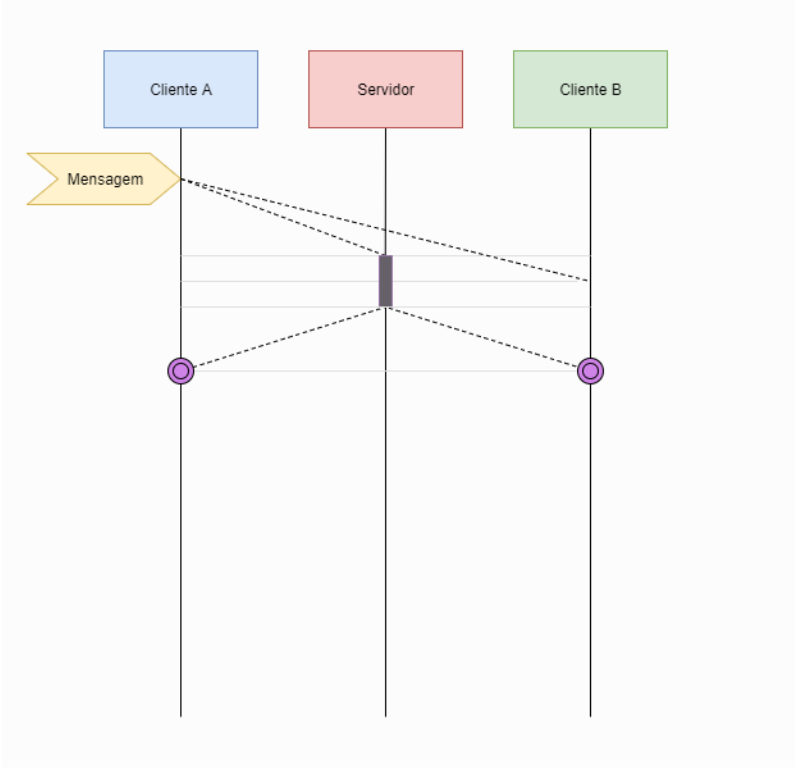
Depois de receber o *pack de requisição* do jogador A, o servidor realiza a validação da mensagem de acordo com a visão global que ele tem sobre o estado do jogo e se prepara para enviar sua resposta, enquanto isso ocorre o cliente B também processa os dados recebidos do *pack sem confirmação*, atuando de acordo com a mensagem recebida e passando para um *estado não confirmado*, podendo antecipar alguma ação do jogador A, para garantir uma maior fluidez do jogo.

Após o tempo de processamento necessário, o servidor envia uma resposta para o cliente A e logo após envia o resultado da mudança de estado para os outros clientes na *esfera de influência* do cliente A (isso pode ser visto na figura 4.2).

Ao receber a resposta do servidor, considerando que a mesma foi positiva, o jogador A confirma a sua *mudança de estado* e continua seu processamento normal. O cliente B ao receber a informação de atualização do servidor passa a validar os dados recebidos anteriormente do cliente A, realizando alterações conforme necessário.

Com o fim do ciclo principal, caso o jogador B tenha recebido os primeiros dados do cliente A antes que o servidor realizasse o seu processamento e enviasse a sua resposta, este mesmo jogador B possivelmente vai ter conseguido antecipar as ações do cliente A de forma a reduzir a sensação do atraso causado pelo *lag*.

Figura 4.2: Fluxo principal parte 2



Fonte: Autor

4.4 Casos de Funcionamento Alternativos

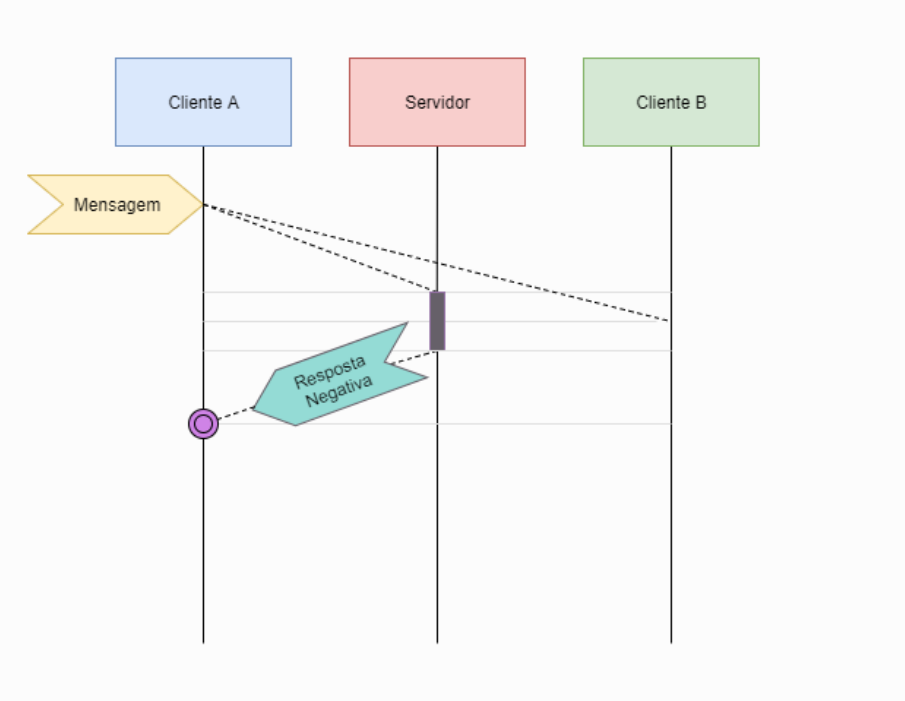
Além do funcionamento principal otimista descrito, existem diversas outras possibilidades de ocorrências que devem ser levadas em consideração. É esperado que tais ocorrências se manifestem em uma proporção muito menor que o caminho mais otimista acontece, mas caso contrário, a solução abordada aqui também prevê medidas para lidar com essas situações.

4.4.1 Mudança de Estado Inválida

Provavelmente o pior caso que pode ocorrer é a situação onde o servidor invalida a ação tomada pelo cliente, enviando uma resposta negativa ao jogador A e não comunicando os outros jogadores de tal acontecimento, fazendo que que estes permaneçam em um estado não confirmado até que outro acontecimento realize a recuperação do mesmo como mostrado na figura 4.3.

Quando isso ocorre, o jogador B fica sem saber se em algum momento futuro ele receberá uma confirmação do servidor sobre a mensagem antes recebida do cliente

Figura 4.3: Mensagem Invalidada



Fonte: Autor

A ou se essa mesma mensagem é inválida. Aqui muito provavelmente acontecerá uma dessincronização temporária onde o jogador B terá que se adaptar após detectar o alerta de *timeout* ou receber uma outra atualização do jogador A, sendo esta válida dessa vez.

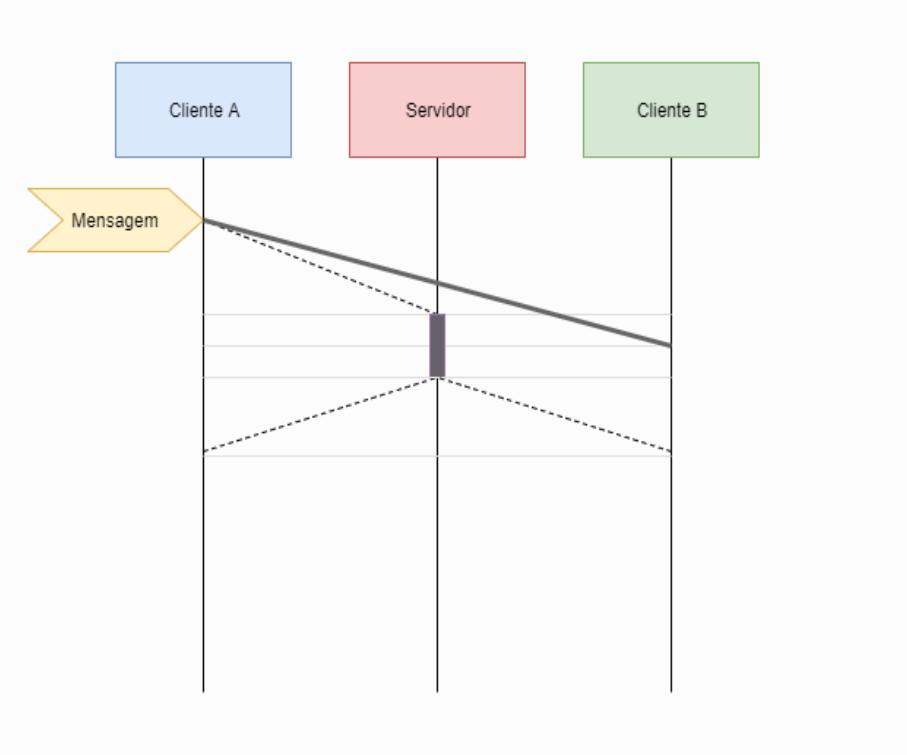
Esse dessincronismo acontece muito comumente em jogos multi-jogador e existem diversas formas de tratar tais ocorrências, cada uma peculiar ao tipo de jogo em questão e a outras características que não fazem parte do escopo atual dessa monografia.

4.4.2 Perda de Pacotes

A solução aqui não acrescenta muitos pontos novos fora dos já existentes em conexões cliente-servidor que possam oferecer riscos por perda de pacotes. Os caminhos novos e sem muito riscos que a ideia acrescenta são os envios entre os clientes e a resposta de confirmação do servidor para um certo grupo de jogadores em uma determinada *esfera de influência*. Existe um terceiro ponto que, caso ocorra uma perda, pode ocasionar um problema mais sério mas ainda sim de acordo com o esperado de um funcionamento padrão de comunicação entre clientes e servidor.

No primeiro caso se ocorrer a perda de um *pack sem confirmação* enviado (figura 4.4), o sistema atuará como funciona o modelo cliente-servidor, já que não teremos o fator que nos permite prever ações de outros usuários, sendo que o prejuízo será mínimo.

Figura 4.4: Perda Pacote sem Confirmação

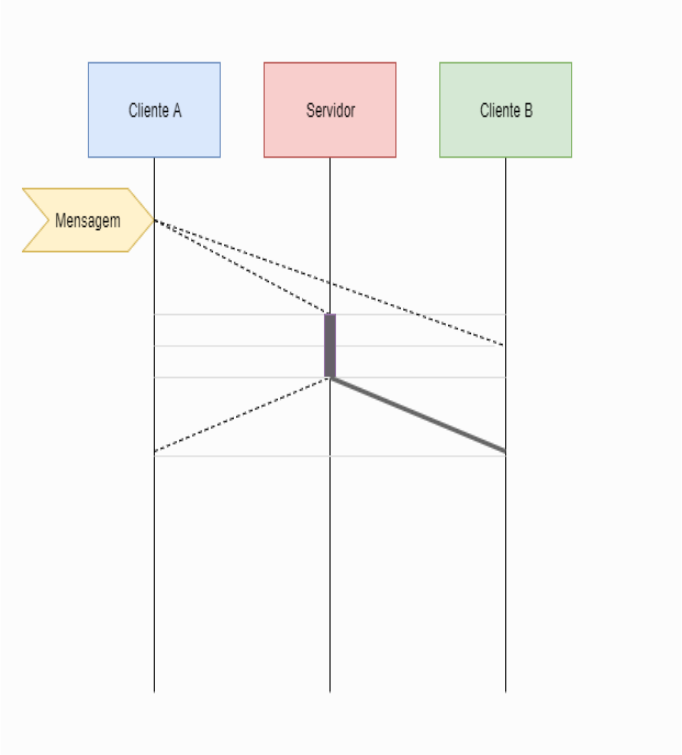


Fonte: Autor

A segunda opção ocorre quando uma propagação de *mudança de estado* é enviada do servidor para um determinado grupo de clientes, se tal mensagem for perdida esses clientes não terão como saber se um *pack sem confirmação* anteriormente recebido será validado, resultando num comportamento muito semelhante ao que acontece quando a validação obtém resultado negativo, fazendo com que ocorra uma dessincronização momentânea que será resolvida quando uma nova troca de mensagem ocorrer com sucesso (figura 4.5).

Existe um ponto crítico que deve ser levado em consideração que, neste caso, é quando a mensagem perdida acaba sendo um *Pack de Requisição*, nessa situação tanto o cliente A, quanto o cliente B estarão esperando por suas respectivas mensagens, no primeiro caso o resultado da validação e no segundo caso a confirmação da *mudança de estado* até o momento que o alerta de *timeout* ocorra.

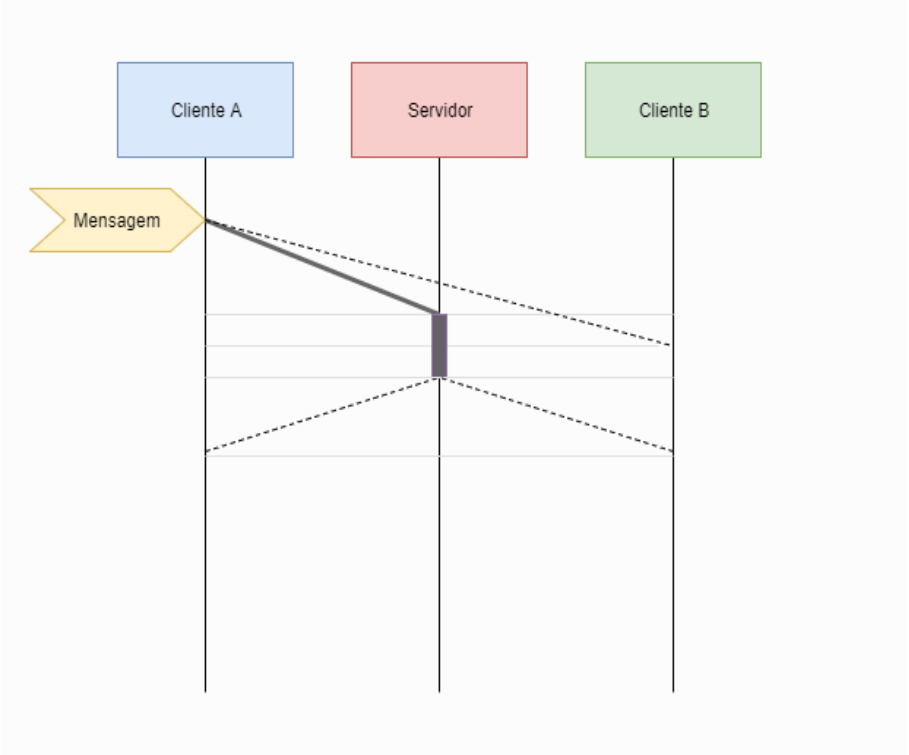
Figura 4.5: Perda Pacote de Atualização



Fonte: Autor

O acontecimento do *timeout* faz com que o cliente A reenvie a primeira mensagem na espera que dessa vez ela seja processada e respondida corretamente e, no caso do cliente B, ocorra a detecção de uma falha de sincronização que só será resolvida com a próxima mensagem de atualização válida (figura 4.6). Novamente existem diversas formas de se tratar esse e outros problemas, algumas soluções serão apresentadas em breve no capítulo 5.

Figura 4.6: Perda Pacote de Requisição



Fonte: Autor

5 TESTES E VALIDAÇÃO

Com a finalidade de demonstrar o funcionamento e validar as suposições aqui feitas, um total de 5 experimentos foram realizados, utilizando uma biblioteca (de autoria do proprio autor) que implementa a solução aqui proposta. Foi construído um programa capaz de realizar essas simulações e obter os resultados de acordo com os tipos de testes escolhidos.

Como a ideia aqui proposta representa uma solução a um problema existente (a latência) e a sua causa depende de diversos fatores, sendo que muitas vezes o seu motivo é a forma de implementação do jogo, nem todos os casos possíveis podem ser abordados. Os testes feitos englobam um espaço menor mas muito próximo ao que se tem comunalmente em jogos multi-jogador, tentando englobar variações principalmente do lado do cliente.

5.1 Detalhamento do Modelo de Teste

O programa foi desenvolvido com a ferramenta Visual Studio da Microsoft, que é uma *IDE* com diveras caracteristicas facilitadoras para aprogramação. Ela trabalha principalmente em ambiente Windows mas possui capacidade de compilar para outras plataformas.

A linguagem de programação escolhida foi o C++ em conjunto com a library SmallPack que pode ser encontrada em <https://github.com/RodrigoHolztrattner/SmallPack>. Essa library foi construída de forma a se aproximar o máximo possível da teoria envolvida na solução apresentada neste trabalho, podendo ser utilizada para simulações e com o objetivo de ser continuada até o ponto que seu uso em jogos seja considerável.

Para a simulação de latência, foi utilizado a ferramenta Clumsy que pode ser encontrada em <http://jagt.github.io/clumsy/index.html>. Este utilitário permite realizar diversas interferências diretamente nas portas do sistema, podendo simular vários efeitos na rede, entre eles a latência, que é justamente o que estamos tentando minimzar.

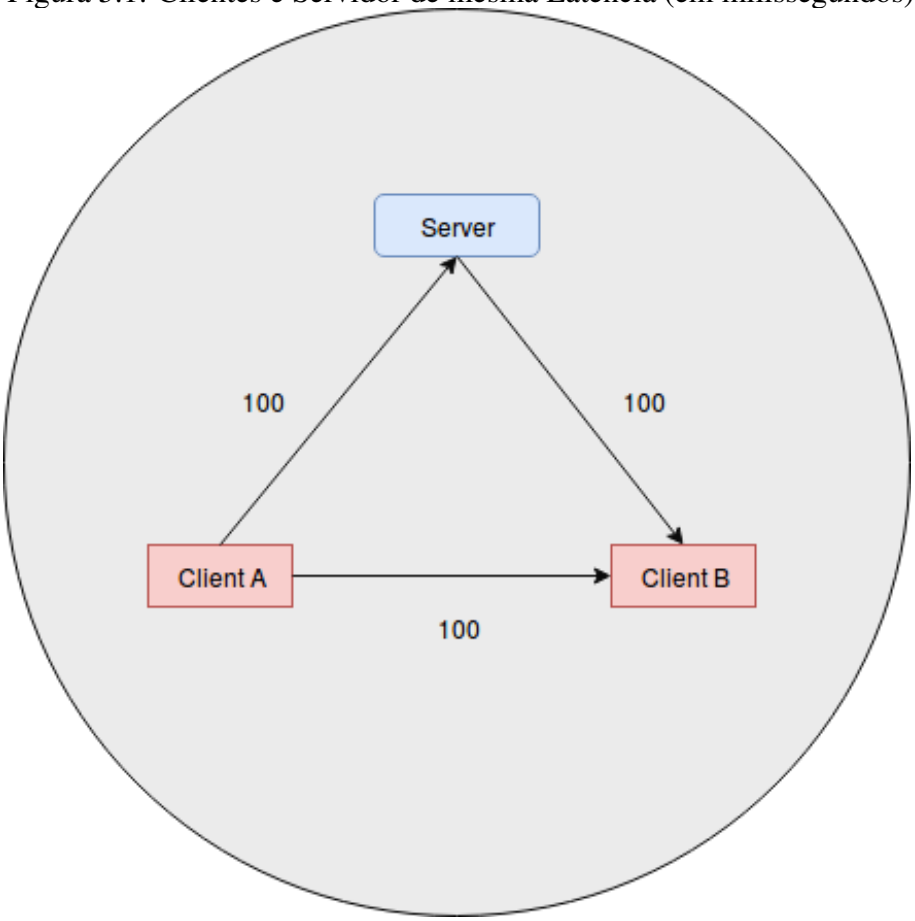
5.2 Resultados dos Experimentos

Todos os resultados obtidos com os testes estão representados de acordo com a média total aritmética dos mesmos, foram feitas **50 medições** automatizadas pelo programa construído e repetidos **3 vezes**, totalizando **150 amostras** para cada medição. As informações de tempo informadas (latência) **incluem** tanto o tempo de propagação da mensagem assim como o tempo de processamento.

5.2.1 Clientes e Servidor com mesma Latência

O primeiro teste idealiza um cenário em que existam, além do servidor, os clientes (nomeados) A e B, sendo que os 3 pontos de comunicação possuem a mesma latência de ocmunicação entre si. Aqui todos os caminhos possuem **100 milissegundos** de atraso. O teste consiste em o cliente A realizar uma mudança de estado e portanto, realizar as devidas trocas de mensagens (figura 5.1).

Figura 5.1: Clientes e Servidor de mesma Latência (em milissegundos)



Fonte: Autor

Foram realizadas duas medidas, a primeira seria o funcionamento normal cliente-servidor no modelo construído, o cliente A se comunica com o servidor, que processa a mensagem recebida, respondendo a requisição do cliente A e também atualizando o cliente B, os resultados podem visualizados na tabela 5.1.

Tabela 5.1: Clientes e Servidor de mesma Latência - Primeira Medida

Fluxo Normal (cliente-servidor)	
Cliente A até o Servidor	104ms
Servidor até Cliente A	105ms
Servidor até Cliente B	105ms
Total A -> B	**209ms**

Na segunda medição, exposta na tabela 5.2, foi realizada utilizado a solução abordada aqui, onde o cliente A, além de se comunicar com o servidor, também se comunica com o cliente B, enviando para este um pack sem confirmação, sendo que o restante continua exatamente igual ao fluxo do primeiro teste.

Tabela 5.2: Clientes e Servidor de mesma Latência - Segunda Medida

Usando o Share-Pack (solução proposta)	
Cliente A até o Servidor	104ms
Cliente A até o Cliente B	**104ms**
Servidor até Cliente A	105ms
Servidor até Cliente B	105ms

Comparando as duas medições, o tempo que o cliente B demorou para "notar"a atualização do cliente A foi superior na primeira medida, que utiliza o sistema convencional *cliente-servidor*. De acordo com a tabela 5.3 é possível notar que em um modelo como esse que não privilegia nenhum dos participantes, a solução proposta prevaleceu como a melhor.

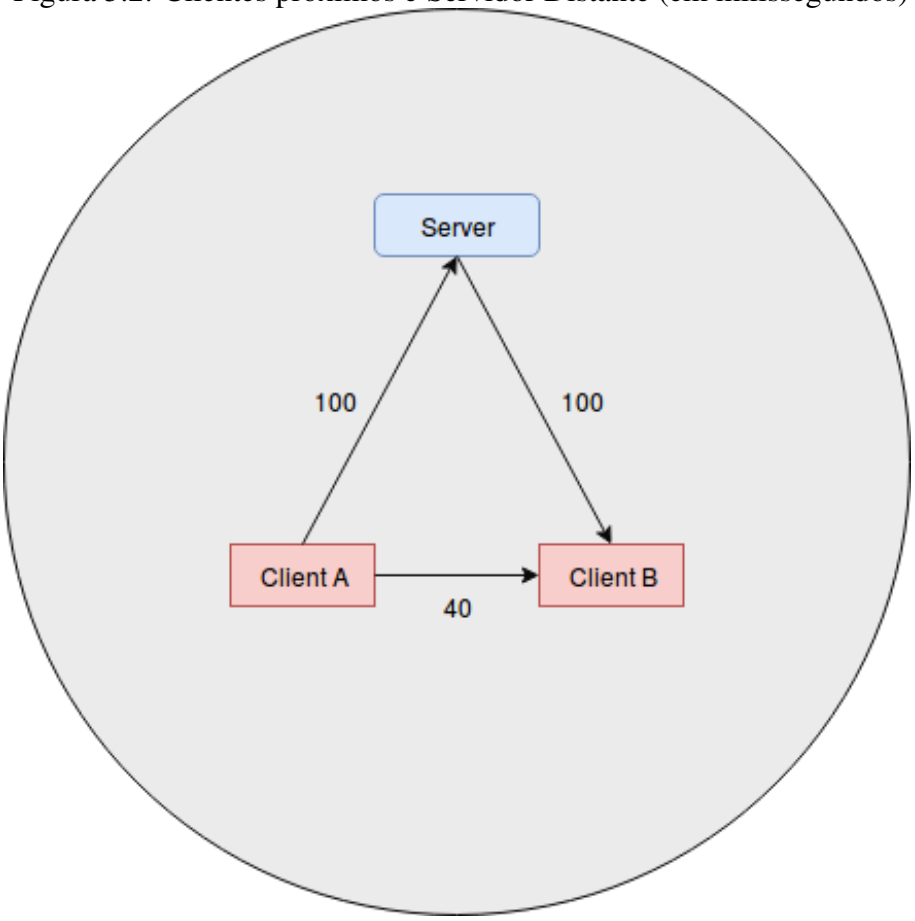
Tabela 5.3: Clientes e Servidor de mesma Latência - Comparação

Primeira Medida (A -> B)	209ms
Segunda Medida (A -> B)	104ms
Ganho Observado	**105ms**

5.2.2 Clientes Próximos e Servidor Distante

O segundo teste parte do princípio que os dois clientes existentes, A e B, possuem uma latência menor entre si (**40ms**) quando comparado ao servidor (**100ms**), essas definições tentam simular um cenário comum a jogos do tipo *MMO* onde os jogadores são da mesma região (geralmente do mesmo país), podendo ou não estarem jogando em grupos.

Figura 5.2: Clientes próximos e Servidor Distante (em milissegundos)



Fonte: Autor

O fluxo de comunicação se mantém o mesmo, na primeira medição o cliente A envia a mensagem de atualização para o servidor, que responde logo após o cliente A e atualiza o cliente B da situação do cliente A. Essas medições podem ser vistas na tabela 5.4.

Tabela 5.4: Clientes próximos e Servidor Distante - Primeira Medida

Fluxo Normal (cliente-servidor)	
Cliente A até o Servidor	104ms
Servidor até Cliente A	105ms
Servidor até Cliente B	105ms
Total A -> B	**209ms**

A segunda medida foi realizada de forma que o cliente A, ao invés de apenas enviar o seu *pack de requisição* ao servidor, também envia um *pack não confirmado* ao cliente B, resultados de acordo com a tabela 5.5.

Tabela 5.5: Clientes próximos e Servidor Distante - Segunda Medida

Usando o Share-Pack (solução proposta)	
Cliente A até o Servidor	104ms
Cliente A até o Cliente B	**44ms**
Servidor até Cliente A	105ms
Servidor até Cliente B	105ms

Embora o cliente B apenas terá a confirmação do *pack sem confirmação* um bom tempo depois (conforme a tabela 5.5), ele poderá muito antes (conforme a tabela 5.6) prever a atualização do cliente A, agindo conforme o jogo foi programado para tal acontecimento.

Tabela 5.6: Clientes próximos e Servidor Distante - Comparação

Primeira Medida (A -> B)	209ms
Segunda Medida (A -> B)	44ms
Ganho Observado	**165ms**

5.2.3 Clientes Distantes e Servidor no Meio

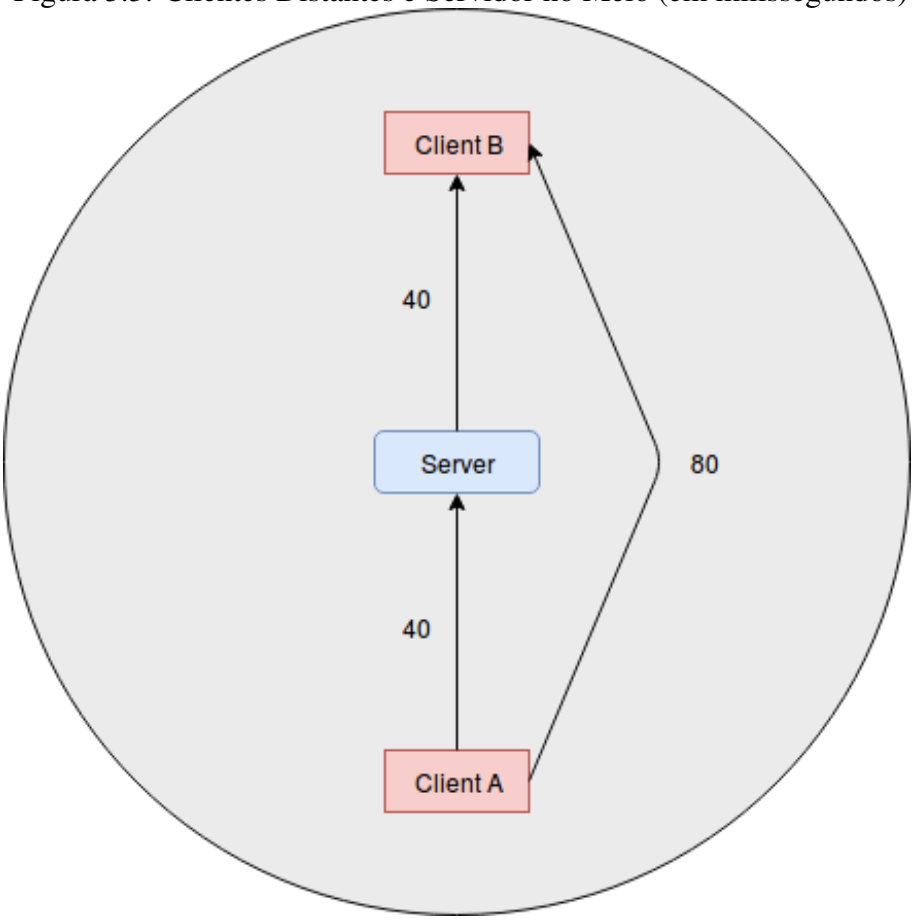
No terceiro experimento os clientes A e B estão posicionados nos extremos de forma que o servidor se encontre entre eles, aqui a latência entre os clientes é de **80 milissegundos** e qualquer conexão dos clientes com o servidor é de **40 milissegundos**. Este caso de teste se aproxima muito do pior caso possível, onde temos 2 jogadores possivelmente posicionados muito longe um do outro geograficamente, em conjunto com um servidor que atende ambas as regiões (figura 5.3).

O fluxo de comunicação segue o mesmo para ambas as medições, no primeiro caso a mensagem parte do cliente A até o servidor, que além de responder esta mensagem, informa ao cliente B a *mudança de estado* do cliente A. (Referência: tabela 5.7).

Tabela 5.7: Clientes Distantes e Servidor no Meio - Primeira Medida

Fluxo Normal (cliente-servidor)	
Cliente A até o Servidor	44ms
Servidor até Cliente A	45ms
Servidor até Cliente B	45ms
Total A -> B	**89ms**

Figura 5.3: Clientes Distantes e Servidor no Meio (em milissegundos)



Fonte: Autor

Na segunda medição, seguindo novamente o modelo apresentado aqui, foi realizado o teste que provavelmente é o pior possível para a solução apresentada, portanto temos o pior valor medido, quando comparado em proporção, de acordo com a tabela 5.8.

Tabela 5.8: Clientes Distantes e Servidor no Meio - Segunda Medida

Usando o Share-Pack (solução proposta)	
Cliente A até o Servidor	44ms
Cliente A até o Cliente B	**84ms**
Servidor até Cliente A	45ms
Servidor até Cliente B	45ms

Mesmo este sendo um caso desfavorável, o Share Pack se mostrou mais eficiente, embora em uma margem bem menor quando comparado com os dados obtidos do sistema *cliente-servidor* de acordo com a tabela 5.9, proporcionando um ganho pequeno, mas existente.

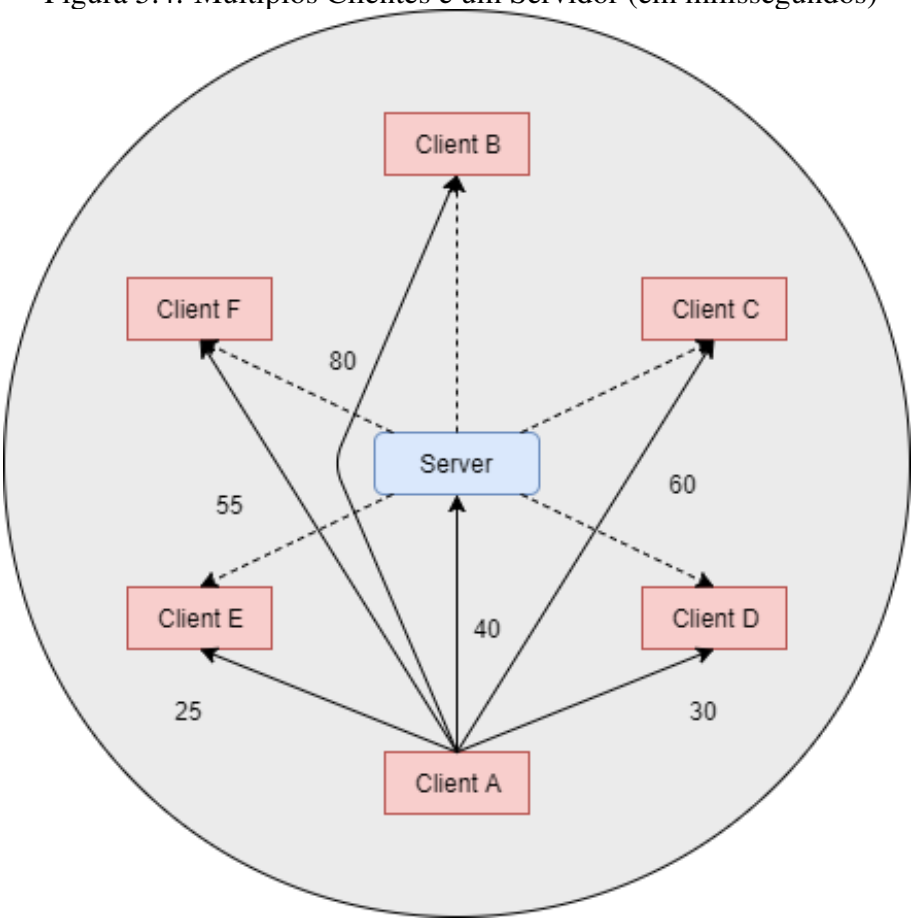
Tabela 5.9: Clientes Distantes e Servidor no Meio - Comparação

Primeira Medida (A -> B)	89ms
Segunda Medida (A -> B)	84ms
Ganho Observado	**5ms**

5.2.4 Múltiplos Clientes e um Servidor

O quarto teste tenta ilustrar uma situação possivelmente real de um jogo *MMO* onde temos vários jogadores com *pings* diferenciados entre si na mesma *esfera de influência* do jogador A. Para facilitar as comparações foi adotado que a demora de comunicação entre o servidor com cada cliente é de **40ms**. Os outros clientes possuem latência conforme indicado na figura 5.4.

Figura 5.4: Múltiplos Clientes e um Servidor (em milissegundos)



Fonte: Autor

A primeira medição tenta sempre criar uma base comparativa de como seria o funcionamento típico, agora o que muda é que além dos clientes A e B, existem também os clientes C, D, E e F. Todos devem receber a informação de atualização do cliente A.

Tabela 5.10: Múltiplos Clientes e um Servidor - Primeira Medida

Fluxo Normal (cliente-servidor)	
Cliente A até o Servidor	44ms
Servidor até Cliente A	45ms
Servidor até Cliente B	45ms
Servidor até Cliente C	45ms
Servidor até Cliente D	45ms
Servidor até Cliente E	45ms
Servidor até Cliente F	45ms
Total A -> B/C/D/E/F	**89ms**

Na segunda medição, temos o envio agora do *pack sem confirmação* também para os clientes C, D, E e F.

Tabela 5.11: Múltiplos Clientes e um Servidor - Segunda Medida

Usando o Share-Pack (solução proposta)	
Cliente A até o Servidor	45ms
Cliente A até o Cliente B	**84ms**
Cliente A até o Cliente C	65ms
Cliente A até o Cliente D	35ms
Cliente A até o Cliente E	**30ms**
Cliente A até o Cliente F	60ms
Servidor até Cliente A/B/C/D/E/F	46ms

Na tabela abaixo (5.12) podemos ver que mesmo para vários clientes a solução Share Pack se mostrou muito eficiente. O tempo médio se manteve praticamente constante; o fato de existirem vários jogadores em uma mesma *esfera de influência* adiciona uma necessidade de processamento a mais apenas nos clientes, liberando o servidor de qualquer esforço extra.

Tabela 5.12: Múltiplos Clientes e um Servidor - Comparação

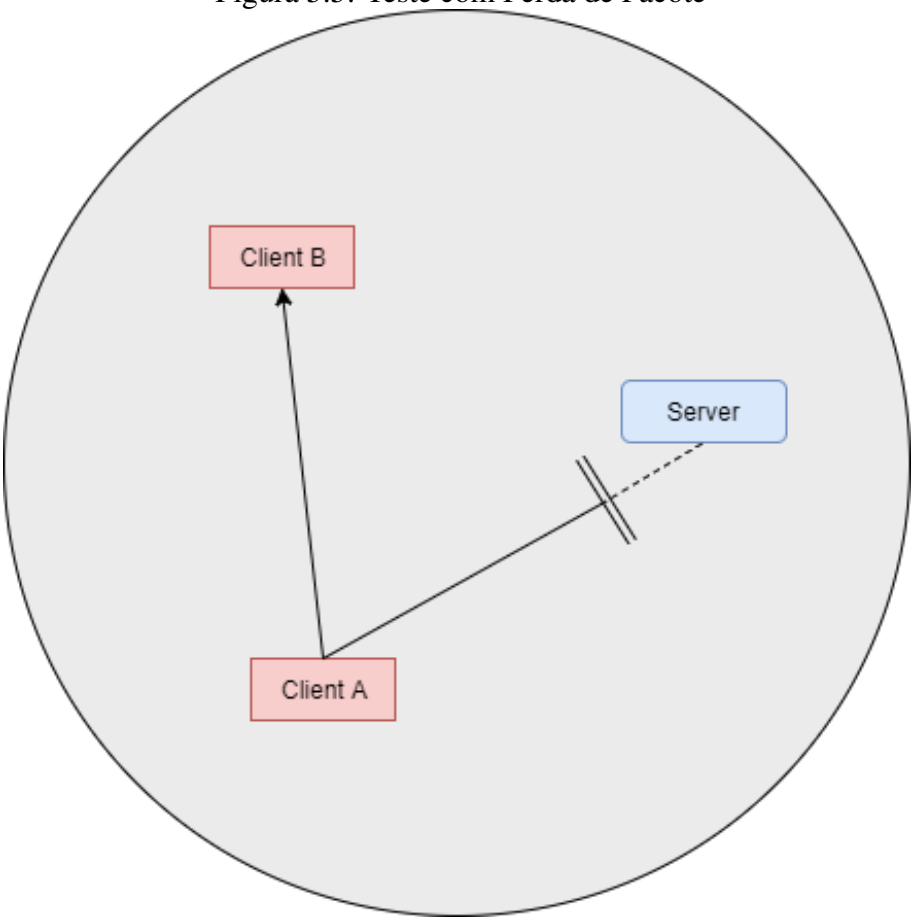
Primeira Medida (A -> B/C/D/E/F)	89ms
Segunda Medida (A -> B)	84ms
Segunda Medida (A -> C)	65ms
Segunda Medida (A -> D)	35ms
Segunda Medida (A -> E)	30ms
Segunda Medida (A -> F)	60ms
Ganho Observado Mínimo	**5ms (A -> B)**
Ganho Observado Máximo	**54ms (A -> E)**

5.2.5 Teste com Perda de Pacotes

Esse último teste tem como objetivo ilustrar como a latência se comporta caso o meio não seja seguro, situação que se manifesta quando alguns pacotes acabam demorando demais para chegar ao destino ou mesmo acabam sendo perdidos. Usaremos o contexto de que o cliente A está se comunicando com o servidor e também com o cliente B (este por estar dentro da *esfera de influência* de A).

A perda de pacote acontecerá no momento que o cliente A envia uma mensagem para o servidor, fazendo que a mensagem enviada seja perdida no meio do caminho, ocasionando que a mesma nunca chegue ao seu destino. Isso não interfere em nada com o envio da mensagem para o cliente B. A latência média simulada foi deixada em 40ms para todas as comunicações e o tempo de *timeout* para algum cliente perceber que o seu *estado não confirmado* é inválido (por não ter recebido uma confirmação do servidor) é de **2 segundos** (2000ms) (figura 5.5).

Figura 5.5: Teste com Perda de Pacote



Fonte: Autor

No primeiro momento a mensagem do cliente A chegou com sucesso até o cliente

B, por outro lado a mensagem com destino ao servidor, por algum motivo do meio, acabou se perdendo, nunca avisando o servidor da nova *atualização de estado*. Os valores medidos estão na tabela 5.13.

Tabela 5.13: Teste com Perda de Pacote - Primeira Etapa

Cliente A até o Servidor	∞ms
Cliente A até o Cliente B	45ms

Neste exemplo, o cliente B ao receber o *pack sem confirmação* do cliente A, faz uso dessa informação conforme o jogo foi programado. Passados 2 segundos (2000ms) o cliente B, por *timeout*, percebe que não recebeu uma mensagem de confirmação do servidor, fazendo que o seu estado atual seja invalidado, causando assim um *rollback* (tabela 5.14). Tais sequencias de acontecimentos fazem com que o mesmo tome ações para se recuperar deste *estado sem confirmação*, conforme foi programado.

Tabela 5.14: Teste com Perda de Pacote - Resultado Primeira Etapa

Tempo de timeout do Cliente B	2000ms
Tempo total até correção do *estado inválido* no Cliente B	**2045ms** (2000ms + 45ms)

A ideia é que qualquer ação que o cliente B tenha tomado em relação ao *pack sem confirmação* do cliente A seja passível de recuperação, ou seja, esta ação não deve ser algo que não possa ser desfeito e que cause uma reação muito perceptível ao usuário (claro que isso depende de jogo para jogo, depende do estilo e da forma que o mesmo foi programado, além disso existem diversas formas de mascarar os efeitos do *rollback*, tais formas não serão discutidas nesta monografia).

Seguindo com o teste, num jogo multi-jogador dificilmente existirá espaços de 2 segundos (para que possa ocorrer *timeout*), geralmente temos uma taxa de envio de dados bem alta, e isso foi levado em consideração na próxima etapa do teste. Agora o cliente A se comunica com o servidor e o cliente B a cada **60ms**, totalizando aproximadamente 16 atualizações de estado por segundo.

Além do *timeout*, é considerado que os clientes podem detectar a perda de pacotes recebendo uma mensagem de confirmação do servidor de um novo pacote, ou seja, na etapa anterior deste teste, caso o próximo *pack de requisição* do cliente A chegue ao servidor, sendo confirmado e repassado ao cliente B (por este estar na *esfera de influência* do cliente A), o cliente B é capaz de perceber que a mensagem anterior que ele recebeu do cliente A é inválida e provavelmente foi perdida, anulando assim muito mais rapidamente esse equívoco e logo depois atualizando-se conforme a nova mensagem.

Tabela 5.15: Teste com Perda de Pacote - Segunda Etapa

Segunda Mensagem	
Tempo de espera para a proxima mensagem do cliente A	60ms
Nova mensagem do Cliente A até o Servidor	44ms
Nova mensagem do Cliente A até o Cliente B	45ms
Propagação do Servidor até o Cliente B	45ms

Claro que neste caso além da nova mensagem enviada ao servidor (dessa vez com sucesso) pelo cliente A, a mesma também seria enviada ao cliente B, nesse ponto fica dependente da forma que o jogo foi implementado. Pode ser colocado uma regra que impede que os clientes utilizem mais de um *pack sem confirmação* por vez (impedindo o encadeamento de erros) ou mesmo usar um sistema robusto que permita que um cliente faça *rollback* em um *estado sem confirmação* mesmo com vários *packs sem confirmação*, mas novamente isso depende de diversos aspectos exteriores à solução aqui proposta. Os resultados medidos podem ser vistos na tabela 5.16.

Tabela 5.16: Teste com Perda de Pacote - Resultado Segunda Etapa

Segunda Mensagem	
Tempo para B receber a atualização do servidor	**89ms** (44ms + 45ms)
Tempo total gasto pelo Cliente B até detectar o problema	**149ms** (44ms + 45ms + 60ms)

Esse teste mostra que existem diversas formas de se tratar o problema de perda de pacotes, cada um com as suas vantagens e desvantagens. Na primeira etapa (tabela 5.14) tivemos um tempo de correção de mais de 2 segundos (2045ms), esse valor pode ser considerado como grande no primeiro momento. De fato 2 segundos é muito tempo num jogo multi-jogador, mas sempre é importante lembrar que qualquer ação tomada com a informação extra que a solução aqui propõe é especulativa, ou seja, ela não deve interferir com uma magnitude muito grande no fluxo do jogo.

Essa informação pode ser meramente utilizada, por exemplo, para carregar algum dado em cache ou mostrar algum efeito especial. Supondo hipoteticamente que em um certo jogo exista o conceito de habilidades, que são ataques especiais que jogador utiliza rapidamente durante uma suposta batalha, essas habilidades geralmente possuem efeitos especiais que devem aparecer junto com o seu uso, sendo que para mostrar esses efeitos especiais no jogo geralmente dependemos que diversos recursos de dados estejam em disco naquele momento, no caso do cliente A estar utilizando alguma habilidade, se o cliente B receber um *pack sem confirmação* disso, ele pode começar a carregar os dados necessários e colocá-los em cache, permitindo muito provavelmente já utilizar esses dados ao invés de ter que começar a recém a carregá-los (quando a confirmação do uso dessa

habilidade chegar pelo servidor).

6 CONCLUSÃO

O trabalho teve como principal objetivo a sugestão de uma nova técnica para a diminuição de latência em jogos multi-jogador. Fazendo uso de um simulador e idealizando uma pseudo comunicação entre jogadores, foi possível realizar diversos testes provando que, para esse ambiente simulado, a técnica sugerida em questão (colocada de lado com outras já existentes) possibilita um ganho expressivo.

Embora existe todo um processo de comunicação extra envolvido, os chamados *packs sem confirmação*, os mesmos não possibilitam diretamente inferir uma atualização válida, pois o recebimento deles não implica que sejam verdadeiros, ainda sim é possível realizar previsões e tomar certas ações que resultem para o cliente em uma melhor experiência de jogabilidade.

O uso híbrido usando conexões auxiliares *P2P* se mostrou vantajoso em relação ao modelo clássico *cliente-servidor*. De acordo com os testes realizados é possível perceber que, na grande maioria dos casos, esse sistema proporciona ganhos muito maiores por um custo pequeno. Os problemas causados por este modelo podem ser tratados de diversas formas (não abordadas diretamente nessa monografia), causando um impacto mínimo, caso existente. Uma súmula das caracteristicas dessa solução pode ser vista como:

- Conexões *P2P* não impactam o servidor principal do jogo.
- Pode ser implementado em conjunto com uma arquitetura cliente-servidor sem maiores problemas.
- Falhas e problemas de comunicação das mensagens *P2P* não causam problemas no fluxo do jogo.
- Ganhos consideráveis dependendo da latência entre os participantes dentro da *esfera de influência*.
- Seu uso não é restrito apenas à área de jogos, teoricamente qualquer arquitetura. cliente-servidor poderia fazer uso desta técnica (se for visto que seja vantajoso).

Existem várias oportunidades que podem ser exploradas com essa técnica, algumas já foram mencionadas aqui como por exemplo o carregamento prévio em cache de dados (final da seção 5.13) entre outras, o próprio modelo em si não cria restrições fazendo com que o desenvolvedor sinta-se livre para até realizar o descarte dessas informações quando conveniente.

6.1 Trabalhos Futuros

Todo o processo de desenvolvimento, principalmente na implementação do ambiente simulado para a realização do testes, não envolveu diretamente o uso dessa ideia em um jogo multi-jogador verdadeiro.

A continuação deste projeto tem como objetivo expandir a solução para um ou mais jogos multi-jogador existentes e aprimorar detalhes técnicos a respeito, possibilitando o uso de toda a informação extra que será fornecida com a utilização desta técnica nova, possibilitando algum tipo de ganho (dependente do tipo de jogo) em cima do atraso devido à latência.

REFERÊNCIAS

BARRI, I.; ROIG, C.; GINé, F. Distributing game instances in a hybrid client-server/p2p system to support mmorpg playability. **Springer**, p. 2005–2029, 2013. Available from Internet: <https://link.springer.com/article/10.1007/s11042-014-2389-0>.

BERNIER, Y. W. Latency compensating methods in client/server in-game protocol design and optimization. Valve, 2015. Available from Internet: <http://www.vis.uni-stuttgart.de/plain/seminare/computerspiele/latency.pdf>.

CECIN, F. Freemmg : uma arquitetura cliente-servidor e par-a-par de suporte a jogos maciçamente distribuídos. 2005. Available from Internet: <http://hdl.handle.net/10183/6223>.

FIEDLER, G. **What Every Programmer Needs To Know About Game Networking**. 2010. Disponível em: <https://gafferongames.com/post/what_every_programmer_needs_to_know_about_game_networking/>. Acessado em: 22/12/2017.

FIEDLER, G. **UDP vs. TCP**. 2014. Disponível em: <https://gafferongames.com/post/udp_vs_tcp/>. Acessado em: 22/12/2017.

FROSTBYTE, C. **The technical argument? Dedicated servers vs P2P local pool for FPS games**. 2017. Disponível em: <https://community.spiceworks.com/topic/1996629-the-technical-argument-dedicated-servers-vs-p2p-local-pool-for-fps-games>. Acessado em: 22/12/2017.

GULLIFORD, R. **Multiplayer is the Future of Gaming**. 2016. Disponível em: <http://plusmana.com/multiplayer-future-of-gaming/>. Acessado em: 22/12/2017.

HOOK, B. **Introduction to Multiplayer Game Programming**. 2006. Disponível em: <http://trac.bookofhook.com/bookofhook/trac.cgi/wiki/IntroductionToMultiplayerGameProgramming>. Acessado em: 22/12/2017.

JOHNSON, L. **The Advantages Of Using P2P Technology**. 2017. Disponível em: <https://www.ibm.com/developerworks/community/blogs/bf6e2b0a-577c-4d00-a45c-8248844c8985/entry/The_Advantages_Of_Using_P2P_Technology?lang=en>. Acessado em: 22/12/2017.

LUARD, M. **Autopsy of an Indie MMORPG - Part 1**. 2015. Disponível em: <http://cranktrain.com/blog/autopsy-of-an-indie-mmorpg-1/>. Acessado em: 22/12/2017.

MASTIN, P. **Digital distribution and the new Oz for video games**. 2015. Disponível em: <https://www.cedexis.com/blog/guest-blog-toto-were-not-in-gamestop-anymore-digital-distribution-and-the-new-oz-for-video-games/>. Acessado em: 22/12/2017.

MASTIN, P. **How Latency is Killing Online Gaming**. 2016. Disponível em: <https://venturebeat.com/2016/04/17/how-latency-is-killing-online-gaming/>. Acessado em: 22/12/2017.

NOORDZIJ, A. **The Rise of Video Game Vanity: Or, Why Skins Sell**. 2015. Disponível em: <https://www.linkedin.com/pulse/rise-video-game-vanity-why-skins-sell-adriaan-noordzij/>. Acessado em: 22/12/2017.

ROUSE, M. **client/server (client/server model, client/server architecture)**. 2008. Disponível em: <http://searchnetworking.techtarget.com/definition/client-server>. Acessado em: 22/12/2017.

ROUSE, M. **TCP (Transmission Control Protocol)**. 2014. Disponível em: <http://searchnetworking.techtarget.com/definition/TCP>. Acessado em: 22/12/2017.

ROUSE, M. **UDP (User Datagram Protocol)**. 2015. Disponível em: <http://searchnetworking.techtarget.com/definition/UDP-User-Datagram-Protocol>. Acessado em: 22/12/2017.

STRINGER, J. **The Causes Of Lag In PC Gaming And How Your Brain Interprets Lag**. 2015. Disponível em: <https://blog.parsec.tv/whats-actually-causing-my-brain-to-detect-lag-in-games-73dbf4430834>. Acessado em: 22/12/2017.

VALDES, G. **Ubiquitous multiplayer is gaming's future**. 2016. Disponível em: <https://www.cheatsheet.com/technology/what-is-the-future-of-multiplayer-video-games.html/?a=viewall>. Acessado em: 22/12/2017.

WALLACE, M. **The Game Is Virtual. The Profit Is Real**. 2005. Disponível em: <http://www.nytimes.com/2005/05/29/business/yourmoney/the-game-is-virtual-the-profit-is-real.html>. Acessado em: 22/12/2017.

WATERS, K. **Peer-to-Peer vs. Client-Server Networks**. 2002. Disponível em: <https://www.techwalla.com/articles/peer-to-peer-vs-client-server-networks>. Acessado em: 22/12/2017.