Universidade de São Paulo
Instituto de Matemática e Estatística
Bacharelado em Ciência da Computação

Prevenção de Trapaças em Jogos Peer-to-Peer

Luciano Rodrigues Saraiva Leão

Monografia Final

mac 499 — Trabalho de Formatura Supervisionado

Supervisor: Dr. Wilson Kazuo Mizutani

São Paulo
2022

O conteúdo deste trabalho é publicado sob a licença CC BY 4.0
(Creative Commons Attribution 4.0 International License)

Resumo

Luciano Rodrigues Saraiva Leão. **Prevenção de Trapaças em Jogos Peer-to-Peer**. Monografia (Bacharelado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2022.

Protocolos de sincronização em jogos digitais em rede que utilizam modelo de rede *Peer-to-Peer* podem ser mais vulneráveis a trapaças do que os que usam o modelo Cliente/Servidor. Além disso, muitos desenvolvedores de jogos digitais optam por usar o modelo *Peer-to-Peer* em seus jogos, mas encontram muitos desafios em deixar seus jogos seguros contra trapaças. Esse trabalho exibe as diversas trapaças presentes em jogos digitais e discute protocolos que tentam preveni-las.

Palavras-chave: Trapaças. Peer-to-Peer. Jogos em rede.

Abstract

Luciano Rodrigues Saraiva Leão. **Cheating Prevention in Peer-to-Peer Games**. Capstone Project Report (Bachelor). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2022.

Synchronization protocols in networked digital games that use the Peer-to-Peer network model may be more vulnerable to cheating than Client/Server. In addition, many digital game developers choose to use the Peer-to-Peer model in their games but find many challenges in making their games safe from cheating. This paper shows the various cheats present in digital games and discusses protocols that try to prevent them.

Keywords: Cheating. Peer-to-Peer. Networked games.

Lista de figuras

2.1 Diagrama ilustrando o funcionamento de *Game Loop*, retirado de Nystrom (2014) 5
2.2 Envio de pacotes com *Jitter*, retirado de Steed e Oliveira (2010a). 6

3.1 Um diagrama ilustrando os níveis de trapaças 8

Capítulo 1

Introdução

1.1 Motivação

Jogos multijogadores produzem uma experiência de jogo diferente dos jogos de jogador único. O motivo disso é que esses jogos utilizam elementos com características sociais como cooperação e competitividade. Ademais, a invenção da internet tornou os jogos dessa natureza mais acessíveis, permitindo a criação de jogos digitais que suportam jogadores em grandes distâncias e quantidades. Para desenvolver esses jogos em rede, surgiram vários protocolos de sincronização, que utilizam, principalmente, um destes dois modelos de rede: Cliente/Servidor (C/S) e *Peer-to-Peer* (P2P).

A arquitetura Cliente/Servidor é a mais utilizada no contexto de jogos em rede. Isso ocorre porque a maior parte dos protocolos são mais fáceis de implementar e mais seguras contra a maioria das formas de trapaça (Webb e Soh, 2008). Por outro lado, a arquitetura *Peer-to-Peer* pode diminuir o tempo de sincronização e aumentar a escalabilidade do jogo.

1.2 Problema

Em jogos digitais que utilizam o modelo de rede *Peer-to-Peer*, em geral, não é possível garantir que há um nó confiável para receber e transmitir informação sem manipulá-la. Dessa forma, os protocolos que adotam esse modelo tendem a ser mais vulneráveis a trapaças que os protocolos que utilizam o modelo Cliente/Servidor.

1.3 Objetivos

Essa monografia tem como propósito entender as possíveis trapaças em jogos de arquitetura *Peer-to-Peer* e apresentar soluções para preveni-las. Para isso, os objetivos podem ser definidos como:

- Investigar os métodos de trapaça em jogos de arquitetura *Peer-to-Peer*.
- Estudar protocolos de sincronização que previnem trapaças.

- Implementar e comparar quantitativamente os protocolos estudados.

1.4 Organização do texto

O capítulo 2 desse trabalho expõe alguns conceitos sobre o funcionamento de jogos digitais em rede que serão úteis para o decorrer do texto. Esse capítulo define estado de jogo e mostra os modelos de redes e de consistência usados em jogos em rede. Além disso, o capítulo também explica alguns termos usados no desenvolvimento de jogos que serão utilizados nesse trabalho.

O capítulo 3 apresenta uma classificação para as possíveis trapaças encontradas em jogos digitais. Para isso, o capítulo define o que é trapacear em jogos digitais e divide as trapaças em quatro níveis, explicando e exemplificando as trapaças em cada nível.

O capítulo 4 descreve as abordagens que previnem trapaças em jogos *Peer-to-Peer*. Esse capítulo detalha os protocolos de sincronização criados para evitar algumas das trapaças apresentadas no terceiro capítulo. Finalmente, no capítulo 5 são apresentadas as conclusões e considerações finais da monografia.

Capítulo 2

Conceitos de jogos em rede

Esse capítulo apresenta algumas definições de jogos digitais multijogadores em rede que serão relevantes aos capítulos seguintes.

2.1 Estado de jogo

Jogos podem ser considerados máquinas de estados finitos, no qual cada estado é formado pelas descrições dos objetos do jogo, como posições, recursos, pontos de vida, entre outros. Em jogos em rede, há dois principais tipos de estado de jogo:

- Estado de jogo local, que guarda somente as informações relevantes para a simulação de certo jogador em dado momento;
- Estado de jogo global, que reúne informações que podem ser relevantes para algum estado local futuro.

Nessa monografia, o termo atualização será usado para se referir a mudanças do estado de jogo local ou global. A atualização de um estado poderá ser transmitida a outros estados que a considerarem relevante.

2.2 Modelos de rede

Em sistemas distribuídos, modelos de rede descrevem como as unidades computacionais, isso é, nós computacionais, estão conectados entre si através da rede. Em jogos em rede, esses nós estão divididos em:

- Nó jogador: o programa que simula estado de jogo local na máquina do jogador;
- Nó servidor: um programa que simula o estado de jogo global.

Para as seções a seguir, será considerado um jogo com necessariamente n jogadores e seus n nós jogadores correspondentes.

Cliente/Servidor

No modelo Cliente/Servidor (C/S), há somente um nó servidor que se comunica com todos os nós jogadores e cada nó jogador se comunica somente com esse nó servidor. Note que a largura de banda necessária para um nó jogador enviar uma atualização será constante em relação a n, já a largura necessária para o nó servidor será proporcional n. O intervalo de tempo para uma atualização sair de um nó jogador e chegar a outro nó jogador é dada pela soma do atraso entre o nó jogador de origem e o nó servidor com o atraso entre o nó servidor e o nó jogador de destino.

Em geral, o nó servidor está numa máquina mantida pelos desenvolvedores do jogo ou na máquina de um dos jogadores. Nos dois casos, o nó servidor pode ser ponto único de falha, embora a probabilidade disso acontecer é bem maior na segunda situação. No primeiro caso, é possível considerar o nó servidor como confiável, isso é, não irá tentar manipular o jogo. Já no segundo caso, não é possível garantir que o nó servidor é confiável, podendo manipular as atualizações recebidas e enviadas.

Peer-to-Peer (nó-a-nó ou par-a-par)

No modelo *Peer-to-Peer*, não há nós servidores, os nós jogadores se comunicam diretamente entre si ou através de outros nós jogadores. Nesse caso, a largura de banda para cada atualização necessária para um nó jogador é proporcional ao número máximo de jogadores os quais a atualização possa ser relevante. Já os atrasos das atualizações é no mínimo o atraso entre os nós jogadores de origem e o de destino.

Nesse modelo, em geral, considera-se que nenhum nó é confiável. Portanto, não há garantia que os nós transmitirão corretamente as atualizações.

Híbrido

O modelo híbrido mistura os modelos Cliente/Servidor e *Peer-to-Peer* num só, com nós jogadores que se comunicando entre si e com o nó servidor. A largura de banda e os atrasos podem corresponder aos valores do caso Cliente/Servidor, *Peer-to-Peer* ou a valores intermediários entre os dois casos.

Esse modelo pode ser útil, por exemplo, para que um servidor monitore o estado global de uma partida de um jogo *Peer-to-Peer*, permitindo que observadores possam assistir a partida ao vivo sem influenciar nas redes dos jogadores.

2.3 Consistência

Em jogos em rede, os nós trocam mensagens entre si para sincronizar os estados do jogo entre suas simulações, mantendo a consistência do jogo. A consistência das simulações do jogo podem ser divididas em duas classes (Steed e Oliveira, 2010b):

Conservativa

Na consistência conservativa, todos os nós computacionais guardam o estado global do jogo e toda atualização do estado de jogo é transmitida (direta ou indiretamente) para

todos os outros nós que aplicam as mesmas operações com essa atualização para se obter o mesmo estado global. Nesse caso, toda informação do jogo estará na máquina de todos os jogadores, mas que os jogadores não necessariamente conseguem acessa-la por completo através da interface do jogo.

Otimista

Na consistência otimista, nem todos os nós guardam o estado global e as atualizações não precisam ser transmitidas a todos os outros nós. Nesse caso, é possível que estados locais estejam leve ou temporariamente fora de sincronia com o estado global, de forma que essa incongruência não interfira muito para a experiência do jogo ou que ocasionalmente seja corrigida.

Note que, em geral, há pelo menos um nó que guarda o estado global. Porém, é possível uma situação em que nenhum nó guarde o estado global, mas que seja possível construí-lo com as informações de todos os estados locais.

2.4 Definições

Esta subseção apresenta e explica alguns dos termos usados em jogos em rede.

- **Área de interesse.** É a região do mundo do jogo que o jogador pode observar ou com a qual pode interagir.
- ***Game Loop.*** Nystrom (2014) É um padrão de projeto representado pelo laço principal do código do jogo. Em cada iteração do *Game Loop*, o jogo recebe as entradas do jogador, avança a simulação e gera as saídas (veja a Figura 2.1).
- **Ciclo** Representa uma iteração no *Game Loop*.
- ***Dead-reckoning.*** É a técnica de extrapolar parâmetros do estado do jogo para manter a consistência. Essa técnica é usada em protocolos com modelo de consistência otimista.
- ***Jitter.*** É a variação na latência da rede (Steed e Oliveira, 2010a) (veja a Figura 2.2). Em jogos, pode tornar movimentações menos suaves, por exemplo.

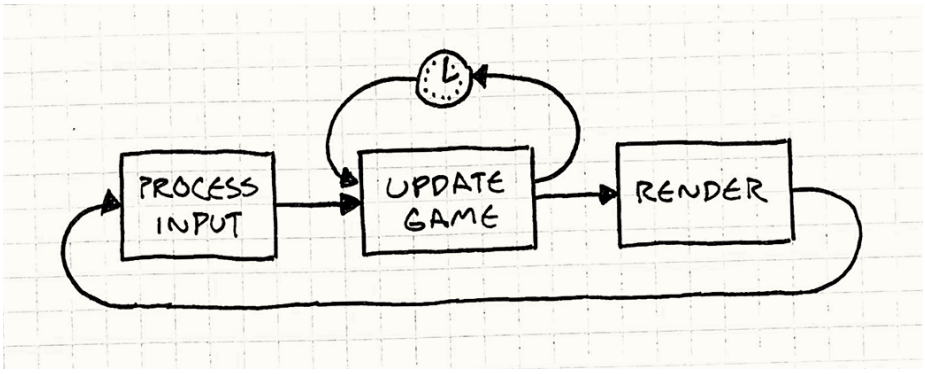

Figura 2.1: *Diagrama ilustrando o funcionamento de Game Loop, retirado de Nystrom (2014)*

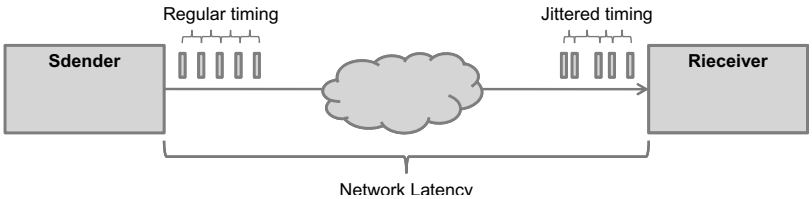

Figura 2.2: *Envio de pacotes com Jitter, retirado de Steed e Oliveira (2010a).*

Capítulo 3

Trapaças em jogos digitais

Nesse capítulo, serão analisadas as possíveis formas de trapacear em jogos digitais em rede.

3.1 Definição

Para elaborar uma classificação das possíveis trapaças, primeiramente é necessário definir o que é trapacear em jogos (digitais ou não). J. J. Yan e Choi (2002) define trapaça como qualquer *"Atitude que um jogador possa tomar para adquirir uma vantagem injusta, ou alcançar um estado que não deveria ser alcançável"* (tradução livre). Já J. Yan e Randell (2009) reformula a definição anterior:

> *Qualquer atitude que o jogador use para adquirir vantagem sobre os demais jogadores ou alcançar um estado é trapaça se, de acordo com as regras do jogo ou a critério do operador do jogo (isso é, o provedor do serviço do jogo que não necessariamente é o desenvolvedor do jogo), a vantagem ou o estado é tal que o jogador não deveria ter alcançado* (tradução livre).

Note que não necessariamente todas as restrições impostas pelas regras do jogo estão implementadas diretamente no *software*, em certas ocasiões nem é possível implementá-las. Por exemplo, não há como um *software* impedir que um jogador registre num papel um conteúdo no qual as regras do jogo estabeleciam que o jogador memorizasse.

A partir dessa definição, também é possível inferir que o operador do jogo pode determinar se certo comportamento não é permitido mesmo que não seja explicitamente proibido pelas regras do jogo. Logo, ações que não foram levadas em consideração durante a elaboração das regras podem ser julgados como trapaças pelo operador durante ou posteriormente à execução do jogo.

Essa última definição será a adotada por essa monografia. A partir disso, já é possível categorizar as trapaças em jogos digitais.

3.2 Estrutura das classificações

A classificação das trapaças nesse texto é baseada na divisão elaborada por Webb, Soh e Lau (2007) que separa as trapaças em quatro níveis (ilustrados na Figura 3.1): jogo (Seção 3.3), aplicação (Seção 3.4), protocolo (Seção 3.5) e infraestrutura (Seção 3.6). As trapaças que envolvem falhas de segurança em geral não foram levadas em consideração (com exceção de *Spoofing*).

Além disso, as seções a seguir exibem exemplos de cada tipo de trapaça. Os exemplos seguem a seguinte notação:

M: representa o jogador que comete a trapaça;

A, B e C: representam jogadores honestos;

X e Y: representam avatares de jogadores.

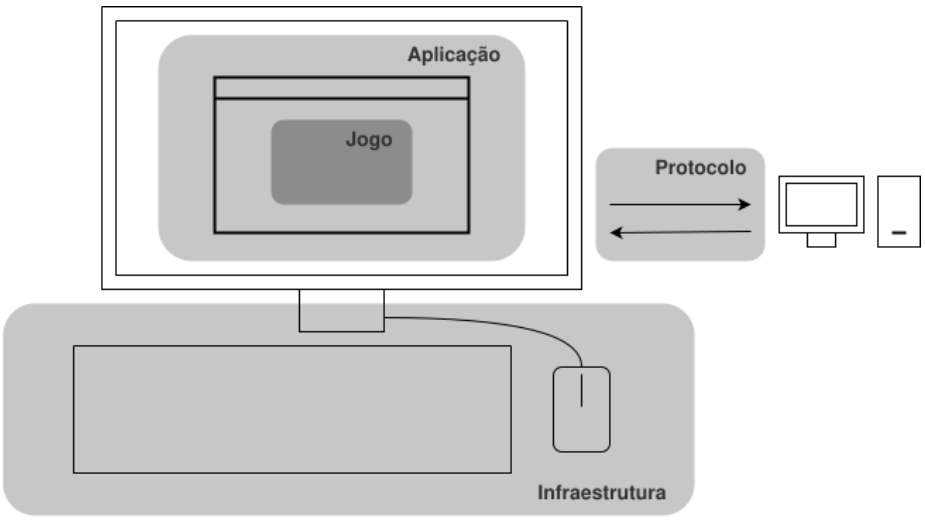

Figura 3.1: *Um diagrama ilustrando os níveis de trapaças*

3.3 Nível de jogo

Nesse primeiro nível, usuários mal-intencionados conseguem trapacear utilizando somente a interface do jogo, sem necessitar modificar qualquer *software* ou hardware. Portanto, é possível que jogadores sem conhecimento técnico ou sem más intenções consigam trapacear.

Há dois tipos de trapaça nesse grupo: por meio de falha de implementação e de falha de design de jogo.

3.3.1 Falha de implementação

Descuidos dos programadores no código fonte do jogo podem gerar vulnerabilidades, das quais os jogadores podem tirar proveito sem que necessitem modificar o jogo ou utilizar algum outro *software*.

Exemplo 1:

Contexto: Os programadores não trataram o caso de *underflow* de inteiros no cálculo dos pontos de vida após um ataque.

Trapaça: Sob um efeito que diminui sua força de ataque, *M* obtém uma força de ataque negativa e retira uma quantidade absurda de pontos de vida de *A* ao atacá-lo.

Exemplo 2:

Gênero: Simulação 3D.

Contexto: Colisões só são detectadas quando as caixas de colisão se sobrepõem.

Trapaça: *M* atravessa paredes ao se movimentar muito rápido, pois sua caixa de colisão não sobrepõe com a da parede em nenhum dos frames que ocorrem durante a movimentação.

3.3.2 Falha de design de jogo

Trapaças através de falha no design ocorrem em situações nas quais os jogadores, seguindo as regras do jogo, alcançam condições vantajosas não previstas pelo designers de jogo e que são consideradas injustas.

Exemplo 1:

Gênero: RPG [1].

Contexto: No mundo do jogo é possível obter um escudo que reduz o dano recebido pela metade e uma armadura que replica o efeito do escudo equipado

Trapaça: *M* equipa o escudo e a armadura, tornando-o invulnerável.

Exemplo 2:

Gênero: Jogo de cartas.

Contexto: Os desenvolvedores não limitaram a duração de tempo dos turnos.

Trapaça: *M* não finaliza seu turno para que A desista da partida e M vença por desistência.

3.4 Nível de aplicação

A nível de aplicação, os jogadores mal-intencionados trapaceiam através de modificações nos arquivos executáveis e de dados, ou utilizando *software* para acessar ou alterar a memória reservada ao jogo.

Observe que, em geral, é necessário conhecimento em programação, técnicas de engenharia reversa e formatos de arquivos. Porém, é possível que mesmo jogadores leigos

[1] *Role-playing game*, gênero de jogo no qual os jogadores controlam personagens e avançam numa narrativa. Os personagens tipicamente possuem características e equipamentos que os permitem um melhor desempenho.

nesses assuntos efetuem trapaças desse tipo, através de *software* de terceiros específicos para certo jogo ou não, que possuem uma interface de usuário amigável que permite que o usuário altere o jogo facilmente. Vale ressaltar que jogos de código aberto permitem alterações em seus *software* de forma relativamente simples, tornando-os mais suscetíveis a esse nível de trapaça.

Nesse nível há três tipos de trapaça: por meio de exposição de informação, de facilitadores e de comandos inválidos.

3.4.1 Exposição de informação

A trapaça por exposição de informação acontece no momento no qual o jogador consegue acessar conteúdo secreto do estado do jogo, isso é, que está armazenado em sua máquina mas não é transmitido ao usuário.

Exemplo 1:

Gênero: Jogo de Cartas.

Trapaça: *M* acessa a memória para visualizar as variáveis que contém os valores das cartas dos adversários.

Exemplo 2:

Gênero: FPS [2].

Trapaça: *M* altera o código fonte do jogo para enxergar no escuro e através de paredes.

3.4.2 Facilitadores

Facilitadores são modificações no jogo ou *software* externo que criam comandos de entradas no jogo, e possuem o objetivo de auxiliar o jogador, muitas vezes utilizados a fim de executar tarefas repetitivas ou que exijam reflexos apurados.

Exemplo 1:

Gênero: FPS.

Trapaça: *M* insere um algoritmo no código do jogo para mirar automaticamente no alvo.

Exemplo 2:

Gênero: RPG.

Contexto: Equipar um escudo previne ataques de inimigos.

Trapaça: *M* insere um algoritmo no código fonte que detecta ataques inimigos e equipa o escudo automaticamente.

[2] *First-person shooter*, gênero de jogo em primeira pessoa focado no combate com armas de fogo.

3.4.3 Comando inválido

Em jogos digitais, comandos são eventos que alteram o estado do jogo. A trapaça por comando inválido ocorre quando o jogador modifica o jogo para inserir ou modificar comandos que o tragam vantagem. Em jogos em rede, os comandos inválidos podem ser transmitidos aos outros nós, mas a depender do protocolo, é possível que esses comandos não sejam aceitos ou reconhecidos pelos outros nós, tornando o estado local de jogo do usuário mal-intencionado inconsistente com o estado global.

Em modelos de rede Cliente/Servidor ou modelos híbridos, essa trapaça pode ser detectada e corrigida mais facilmente, pois o nó servidor pode validar os comandos e detectar quem enviou os comandos alterados. Já no modelo *Peer-to-Peer*, há jogos que os comandos só são enviados para os nós jogadores numa área de interesse, então o jogador pode usar comandos inválidos sem ser detectado quando não há ninguém por perto. Além disso, mesmo se um nó conseguir identificar que outro nó enviou comandos inválidos e alertar os outros nós sobre isso, os outros nós não podem garantir que o nó que alertou não está mentindo.

Exemplo 1:

> Gênero: Jogo de cartas.
>
> Trapaça: *M* modifica a memória para aumentar o valor da variável que corresponde ao limite de cartas na mão para que consiga usar mais vezes o comando de obter mais cartas.

Exemplo 2:

> Gênero: Simulação.
>
> Trapaça: *M* adiciona uma comando que, ao pressionar um botão, o permite teleportar ao local que desejar.

3.5 Nível de protocolo

Trapaças a nível de protocolo envolvem manipulação direta nos dados transmitidos entre os nós, ou seja, podem alterar, adicionar, remover ou duplicar pacotes (Webb e Soh (2008)). Portanto, para executar essa trapaça, é essencial conhecimento em protocolos de comunicação e também pode ser necessário entendimentos de criptografia. Contudo, um jogador leigo pode utilizar *software* de terceiros específicos para o protocolo usado no jogo para interferir nos pacotes.

Nesse nível há nove tipos de trapaça: por supressão de atualização, atraso fixo, remarcação de tempo, inconsistência, complô, *spoofing*, *replay*, oponentes cegos e revogação.

3.5.1 Supressão de atualização

Em jogos que utilizam a técnica de *dead-reckoning*, em geral, adotam uma estratégia de desconectar o jogador após um certo número n de atualizações perdidos em seguidas. Dessa maneira, a trapaça por supressão de atualização consiste no jogador omitir $n - 1$

atualizações aos outros nós e enviar somente a n-ésima, assim ocultando o estado real do seu avatar para os outros jogadores.

Exemplo:

Gênero: FPS.

Trapaça: M se movimenta no mundo sem informar sua posição frequentemente aos outros jogadores, deixando de enviar $n-1$ atualizações, mas transmitindo a n-ésima para não ser desconectado por inatividade. Na perspectiva dos outros jogadores, M estará se teleportando pelo mundo, tornando-o um alvo mais difícil de acertar.

3.5.2 Atraso Fixo

A Trapaça por Atraso Fixo acontece quando o jogador mal-intencionado atrasa de propósito o envio de pacotes a fim de fazer a jogada ótima ou de diminuir o tempo para adversários os reagirem.

Exemplo:

Gênero: Jogo de perguntas e respostas.

Trapaça: M espera A, B e C divulgarem suas respostas para escolher a resposta pela qual a maioria optou.

3.5.3 Remarcação de tempo

Em jogos que usam o modelo de consistência otimista é comum que atualizações transmitidas entre os nós tenham informações do instante de tempo no qual foram enviadas. Um jogador pode alterar essa marcação de tempo para alterar o passado do estado do jogo e assim obter vantagem.

Exemp:

Gênero: Jogo de Luta.

Contexto: Os jogadores que aplicam um golpe ficam vulneráveis durante os primeiros milissegundos do golpe.

Trapaça: M aplica um golpe em A, mas mente ao gerar o pacote da mensagem, informando que o golpe se iniciou milissegundos antes do tempo atual, pulando a parte vulnerável do golpe e não deixando oportunidade para A reagir.

3.5.4 Inconsistência

Trapaça por inconsistência ocorre quando um nó envia atualizações diferentes para um conjunto de nós, gerando estados globais de jogo discordantes entre os nós. Note que, essa trapaça, geralmente, não acontece em arquiteturas C/S, pois o servidor impõe seu estado de jogo aos clientes.

Exemplo:

Contexto: Os jogadores realizam uma votação.

Trapaça: M transmite para A que votou na opção 1 e para B que votou em 2, tornando os estados do jogo de A e B contraditórios, estragando a experiência de jogo.

3.5.5 Complô

Complô ocorre quando jogadores (ou somente um jogador controlando múltiplas instâncias do jogo) de equipes adversárias cooperam entre si, geralmente utilizando meios de comunicação externos.

Note que não necessariamente todos os jogadores envolvidos na trapaça ficarão com vantagem, pelo contrário, é possível que alguns se prejudiquem para favorecer outros.

Exemplo:

Gênero: jogo de futebol.

Contexto: M joga pelo time 1 mas também controla X do time 2.

Trapaça: X realiza gols contra com o intuito de ajudar a equipe 1 a vencer.

3.5.6 *Spoofing*

No contexto de segurança de redes, ataque de Spoofing acontece quando um transmissor legítimo para de se comunicar com um receptor legítimo e o atacante inicia envio de mensagens ao receptor legítimo, fingindo ser o transmissor legítimo (Yilmaz e Arslan, 2015). Portanto, de forma semelhante ao ataque, a trapaça de Spoofing ocorre quando um nó se passa por outro(s) nó(s) com intuito de obter informações secretas ou de modificar o estado do jogo ao seu favor.

Exemplo:

Gênero: RPG.

Contexto: M controla X e A controla Y.

Trapaça: M se identifica como Y aos demais nós com intuito de fornecer os itens de Y para X sem o consentimento de A.

3.5.7 *Replay*

A trapaça por Replay ocorre quando cada jogador transmite suas jogadas de forma criptografada (geralmente por meio de assinaturas digitais), e um nó malicioso é capaz de guardar o pacote criptografado da jogada de algum jogador para que possa retransmitir a jogada futuramente quantas vezes quiser (Corman *et al.*, 2006).

Exemplo:

Gênero: Simulação 2D.

Contexto: A efetua jogadas de movimentação nas quatro direções no plano, gerando pacotes assinados $P1$, $P2$, $P3$, $P4$.

Trapaça: M difunde cópias dos pacotes $P1$, $P2$, $P3$, $P4$ com propósito de mover A para uma posição específica.

3.5.8 Oponentes cegos

Essa trapaça pode ocorrer em arquiteturas híbridas, quando o jogador malicioso envia suas jogadas somente ao um nó servidor, deixando os outros nós jogadores com estados do jogo inconsistentes e portanto sendo necessário que o nó servidor mande atualizações para que os outros nós revertam seus estados.

Exemplo 1:

Gênero: Simulação.

Trapaça: M se movimenta no mundo sem informar A. Dessa forma, o nó servidor precisará conferir que o estado de jogo de A está inconsistente e enviar um pacote para corrigir esse estado e por isso A demora mais para receber a posição de M, diminuindo as chances para interagir com M.

Exemplo 2:

Gênero: Jogo de cartas.

Trapaça: M põe cartas na mesa sem informar A que espera indefinidamente a jogada de M para começar seu turno.

3.5.9 Revogação

A trapaça por revogação pode acontecer em protocolos que os jogadores anunciam uma assinatura das suas atualizações antes de transmiti-las. Essa trapaça ocorre quando um jogador anuncia sua jogada mas decide não envia-la após receber as jogadas dos adversários, desfazendo a sua jogada.

Exemplo:

Gênero: Jogo de perguntas e respostas.

Contexto: M escolha resposta 1; A, B e C escolhem a resposta 2.

Trapaça: M, A, B e C divulgam suas respectivas jogadas criptografadas: J_M, J_A, J_B e J_C. Então A, B e C revelam suas respectivas chaves C_A, C_B e C_C, e M decide não divulgar sua chave, pois percebeu que todos escolheram uma resposta diferente, anulando sua jogada para não perder pontos.

3.6 Nível de infraestrutura

As trapaças a nível de infraestrutura consistem em monitorar ou fazer alterações no *software* ou no hardware usados pelo jogo, atuando como um *proxy* entre a aplicação do jogador mal-intencionado e os outros nós, ou entre a aplicação e sua interface. Note que criptografar os pacotes não é uma solução definitiva, pois não é possível garantir que a chave armazenada na aplicação não é acessível.

3.6.1 Exposição de informação

Assim como na trapaça de mesmo nome apresentada no nível de aplicação, o jogador consegue ler conteúdo não exposto ao usuário. Porém, o processo para obter essas informações não envolve obter os dados diretamente da aplicação, mas através de monitoramento de recursos externos ao *software* do jogo.

Exemplo 1:

Contexto: *M* utiliza um computador como *proxy* para os pacotes enviados de outra máquina que roda o jogo.

Trapaça: *M* utiliza um *software* nesse computador que exibe um mapa do jogo com as posições dos outros jogadores.

Exemplo 2:

Gênero: FPS.

Contexto: *M* altera o *driver* de áudio e adiciona uma rotina que exibe numa tela a posição estimada da origem de um som.

Trapaça: *A* anda, produzindo barulhos de passos e consequentemente *M* descobre sua posição.

Exemplo 3:

Gênero: Jogo de ritmo.

Contexto: *M* altera o *driver* do *joystick* a fim de medir e exibir o intervalo de tempo entre os pressionamentos dos botões.

Trapaça: *M* consegue se adaptar melhor ao ritmo do jogo.

3.6.2 Manipulação

Do mesmo modo que a trapaça anterior, a trapaça por manipulação age sobre os intermediários utilizados pelo *software* do jogo, contudo, ao invés de acessar os conteúdos, o jogador modifica, remove ou insere os dados transmitidos entre o jogo e esses intermediários.

Exemplo 1:

Gênero: FPS.

Contexto: *M* utiliza um computador como *proxy* para os pacotes enviados de outra máquina que roda o jogo.

Trapaça: *M* altera os pacotes com o propósito de aumentar a precisão dos seus disparos ao alvo.

Exemplo 2:

Gênero: Jogo de ritmo.

Contexto: *M* insere uma rotina que simula o pressionamento de botões no *driver* do *joystick*.

Trapaça: *M* passa como entrada as notas musicais de determinada música de antemão para a rotina e como resultado toca a musica perfeitamente.

Capítulo 4

Protocolos anti-trapaça em jogos *Peer-to-Peer*

Esse capítulo apresenta diversos protocolos para prevenir e detectar algumas das trapaças apresentadas no Capítulo 3 em jogos digitais multijogadores com modelo de rede peer-to-peer.

Cada um dos protocolos descritos a seguir tenta solucionar um certo conjunto de trapaças e pressupõe algumas condições que o jogo deve satisfazer. Também é possível combinar alguns dos protocolos, escolhendo alguns deles para utilizar em paralelo para diferentes tipos de atualizações, ou permitindo que os usuários escolham o protocolo a ser usado com base nas trapaças que previne e como o protocolo pode interferir na experiência do jogo.

Note que nenhum dos protocolos define que ações tomar caso seja detectada alguma trapaça, cabendo essa decisão aos desenvolvedores do jogo.

4.1 Lockstep

Geralmente, jogos do gênero RTS [1] exigem que os jogadores sincronizem seus estados de jogo a cada ciclo da simulação. Baseado nisso, pode-se pensar num protocolo simples que consiste em cada nó enviar sua atualização durante um ciclo da simulação e esperar as atualizações dos outros nós para avançar o ciclo. Porém, esse protocolo é vulnerável a trapaças por atraso fixo, já que um jogador malicioso pode esperar as atualizações dos outros jogadores a fim de decidir uma jogada ótima para então transmiti-la.

A partir disso, (Baughman e Levine, 2001) propuseram o protocolo *Lockstep*. Esse protocolo é formado por ciclo divididos em duas etapas. Durante a primeira etapa, os jogadores calculam e transmitem uma assinatura da sua atualização, geralmente um *hash* e aguardam que todos os outros divulguem suas respectivas assinaturas, anunciando assim que receberem de todos.Depois que todos anunciarem as assinaturas das jogadas, é dado início à próxima fase. Nessa segunda etapa, cada jogador compartilha sua atualização,

[1] *Real-time strategy*, tipo de jogo de estratégia que progride em tempo real ao invés avançar por ciclos.

calcula as assinaturas a partir das atualizações recebidas e confere se coincide com os assinaturas recebidas na etapa anterior. Após todos jogadores terminarem essa etapa, o ciclo da simulação avança.

Vale ressaltar que é necessário calcular o *hash* da atualização combinada com um ruído aleatório e também revelar esse ruído junto com a atualização na segunda etapa do protocolo. Isso é importante para que a atualização não possa ser identificada a partir do *hash* e para evitar colisão entre os *hashs*. Além disso, considerando-se que o protocolo de transporte garante que os pacotes chegam inalterados e em ordem, é possível considerar que os jogadores que anunciaram assinaturas inválidas trapacearam. Caso contrário, o protocolo pode considerar que algum pacote foi corrompido e apenas descartar a jogada, abrindo espaço para a trapaça de revogação.

Note que um ciclo da simulação no protocolo *Lockstep* demora $\Delta T_{frame} = max\{3 \cdot atraso_{max}, \Delta T_{min}\}$ com $atraso_{max}$ representando o maior atraso entre dois pares de nós durante esse ciclo e ΔT_{min} representando a duração mínima que os desenvolvedores do jogo escolheram para um ciclo Esse valor pode ser considerado lento para jogos em tempo real.

4.2 Sincronização Assíncrona

Com intuito de reduzir o número de pacotes trocados, Baughman, Liberatore *et al.* (2007) introduziram o protocolo de Sincronização Assíncrona. Diferentemente do *Lockstep*, esse protocolo não requer que todos os jogadores sincronizem seus estados de jogo entre si a todo momento, mas somente se os jogadores que estiverem próximos entre si, isto é, cujas **áreas de interesse** se intersectam. Dessa forma, os nós avançam na simulação assincronamente quando estão isolados e se reconciliam quando se aproximam dos outros nós. Apesar disso, a sincronização em si é idêntica à sincronização do *Lockstep*, disseminando as assinaturas das jogadas e posteriormente as jogadas.

O protocolo de Sincronização Assíncrona consiste em, inicialmente, esperar o jogador decidir sua jogada, transmitir a assinatura dela para todos os outros nós e receber as assinaturas de jogadas de ciclos anteriores ou não de outros jogadores. Após isso, o jogador revela sua jogada para os nós que estão na sua área de interesse e espera as jogadas deles, conferindo as assinaturas.

Há diversas maneiras para se determinar quais jogadores estão com suas áreas de interesse sobreposta. A mais básica delas consiste em escolher uma área de interesse circular, e em cada ciclo os jogadores compartilham as posições e os raios de suas áreas de interesse. Porém, um jogador malicioso pode-se aproveitar da informação sobre a posição dos oponentes, trapaceando por exposição de informação. A fim de evitar isso, pode-se tomar área de interesse de raios fixos, compartilhar as posições antes do primeiro ciclo e, a cada ciclo, estimar uma região contendo todas as possíveis áreas dos outros nós, somente quando a região de um desses nós cruzar com a área do jogador, eles informam suas posições entre si para verificarem se suas áreas de interesse realmente se sobrepuseram. Apesar disso, ainda pode ocorrer exposição de informação quando jogadores descobrem suas posições entre si, mas suas áreas de interesse não se intersectam.

Baughman, Liberatore *et al.* (2007) também propuseram um método que divide a área do jogo em células e através de técnicas de criptografia é possível saber se dois jogadores estão em células próximas ou não sem revelarem suas posições exatas.

Apesar do protocolo de Sincronização Assíncrona poder reduzir a largura de banda em relação ao *Lockstep*, o tempo entre os ciclos ainda é no mínimo 2 vezes o tempo entre os jogadores mais distantes e ainda pode ser considerado lento.

4.3 Pipeline Deslizante

Jamin *et al.* (2003) introduziram três protocolos que otimizam o protocolo *Lockstep* utilizando técnicas de *pipelining*.

O primeiro é chamado de *Pipelined Lockstep*, no qual os jogadores escolhem suas jogadas para p ciclos consecutivos com p um inteiro positivo constante e disponibilizam as assinaturas de todas elas. Assim, um jogador só divulga sua jogada de um ciclo na *pipeline* após receber todos as assinaturas desse ciclo, avança a *pipeline* até o ciclo da primeira jogada ainda não transmitida, escolhendo as jogadas dos ciclos que irão entrar na *pipeline* e enviando suas assinaturas.

A duração de cada ciclo para o *Pipelined Lockstep* é dada por $\Delta T_{frame} = max\{atraso_{max}/p, \Delta T_{min}\}$ com $atraso_{max}$ representando o maior atraso entre dois pares durante esse ciclo e ΔT_{min} representando a duração mínima que os desenvolvedores do jogo escolheram para um ciclo. Note que no caso em que $p = atraso_{max}/\Delta T_{min}$, ΔT_{frame} se torna mínimo, com valor igual a ΔT_{min}. Também pode-se perceber que esse protocolo se torna o *Lockstep* quando $p = 1$.

Apesar de ser possível diminuir bastante a duração dos ciclos escolhendo um valor alto para p, as jogadas terão p ciclos de atraso (inclusive a própria jogada do jogador, para manter a consistência). Além disso, um jogador malicioso pode trapacear ao enviar as assinaturas de todos os ciclos da *pipeline* e esperar para transmitir a jogada do seu primeiro ciclo da *pipeline* somente quando receber todas as jogadas de cada ciclo da *pipeline* com intenção de decidir sua próxima jogada (a que vai entrar na *pipeline*).

O segundo protocolo, *Pipeline* Adaptativa, foi criado para solucionar os dois principais problemas do anterior: a dificuldade de escolher um p que se gere ciclo com duração ótima ou quase ótima durante toda a partida, devido à variação da latência entre os jogadores com o decorrer do jogo; e a trapaça de atrasar o envio das assinaturas. Nesse protocolo, os jogadores transmitem as assinaturas e as jogadas junto com uma marcação do tempo de envio, a fim de monitorar os atrasos, e divulgam os seus respectivos atrasos máximos medidos para então decidir o atraso máximo entre quaisquer dois jogadores. A partir disso, pode-se variar o valor de p com o passar do tempo de forma que otimize o tempo dos ciclos. Também é possível detectar se um jogador mal-intencionado A esperou uma jogada de B no ciclo i ser revelada para decidir sua jogada no ciclo j com $j > i + p$, pois o jogador B pode medir o intervalo entre o tempo que enviou sua jogada revelada do ciclo i e o tempo no qual que recebeu a assinatura da jogada de A no ciclo j e verificar se esse valor é maior que o atraso estimado entre A e B acrescentado da duração do ciclo j. Vale ressaltar que esse protocolo consegue detectar todas os casos desse tipo de trapaça, porém

pode identificar erroneamente jogadores sofrendo com altas variações de latência como trapaceadores.

Finalmente, o terceiro e último protocolo, *Pipeline* Deslizante, surgiu para resolver o problema de *jitter* do protocolo anterior, que ocorre quando o tamanho da pipeline é ajustado. Para isso, esse protocolo estende o *Pipeline* Adaptativa, acrescentando um *buffer* para guardar as jogadas durante ajustes no tamanho da *pipeline*. É usada a técnica de *dead-reckoning* para prever as jogadas que ainda não chegaram, dessa forma, a simulação não é travada e os efeitos do *jitter* são reduzidos.

O protocolo *Pipeline* Deslizante traz duas garantias: a que nenhum usuário conseguirá ter acesso a jogadas de frames que ainda não divulgou a assinatura; e que nenhum jogador mal-intencionado conseguirá, continuamente, decidir uma jogada com informação mais recente que um jogador honesto (Jamin *et al.*, 2003).

4.4 New-Event Ordering

O protocolo *New-Event Ordering* (NEO), proposto por GautheierDickey *et al.* (2004), utiliza a mesma ideia do Lockstep de comprometimento e revelação das jogadas, porém define um tempo limite para as chegadas das jogadas de um ciclo, ignorando as atualizações que, para a maioria dos jogadores, chegarem após esse prazo.

Nesse protocolo, em cada ciclo, os jogadores transmitem pacotes formados pela assinatura da jogada do ciclo mais recente, a jogada do ciclo passado e uma lista de bits com cada bit representando se recebeu o pacote de certo jogador dentro do limite de tempo no ciclo anterior. Então, a partir das listas de bits recebidas, os jogadores aplicam as jogadas que chegaram para a maioria dos jogadores e descartam as demais. Note que os jogadores que não receberam as jogadas que foram aplicadas pela maioria, deverão usar *dead-reckoning* para interpolar a jogada perdida.

Além disso, o protocolo NEO também pode ser utilizado com otimizações como: uso de pipelines adaptativas de forma semelhante à mencionada na seção anterior; ajuste da duração de cada ciclo no decorrer da partida, a partir dos valores dos tempos de atraso; e controle da taxa de envio para diminuir a perda de pacotes, já que, devido ao esquema de votação, os jogadores não precisam enviar suas atualizações a todos os outros jogadores, mas somente para a maioria.

Apesar disso, Corman *et al.* (2006) mostram que o NEO falha em prevenir algumas trapaças que os autores de protocolo se propuseram a resolver, como as trapaças de remarcação de tempo e de inconsistência. Esse protocolo também é vulnerável a outras trapaças, como as trapaças de *replay* e de spoofing. A partir disso, (Corman *et al.*, 2006) apresentaram o protocolo de *Secure Event Agreement* (SEA) que modifica o NEO para resolver essas vulnerabilidades, utilizando técnicas de criptografia.

4.5 Referee Anti-Cheat Scheme

O protocolo *Referee Anti-Cheat Scheme* (RACS), criado por Webb, Soh e Lau (2007) no contexto de jogos multijogador massivos, tenta reunir a segurança do modelo Cliente/Servidor com a escalabilidade do *Peer-to-Peer*, utilizando como base o modelo híbrido *Peer-to-Peer with Central Arbiter* (Pellegrino e Dovrolis, 2003) O protocolo RACS é formado por um nó confiável e autoritário denominado como juiz (*Referee*), um conjunto de jogadores e um servidor para autenticar e guardar os avatares dos jogadores.

Assim como o NEO, o protocolo RACS pode operar em dois modos: *PP* e *PRP* e possui ciclos com duração fixa que estão sincronizados entre os nós pode-se usar o protocolo NTP [2] para isso. Primeiramente, um jogador P_A é validado pelo servidor de autenticação que transmite ao jogador e a um juiz R o estado do avatar de P_A. Posteriormente, R envia a P_A a parte do estado do jogo relevante a P_A, e informa a presença de P_A aos jogadores dentro de uma certa área de interesse. Então, inicialmente, P_A entra no modo *PRP* com os outros jogadores: nesse modo dois jogadores trocam atualizações com os pacotes passando pelo juiz. Assim, gradualmente, R transmite mensagens a cada par de P_A e a P_A impondo que entrem no modo *PP* no qual dois jogadores trocam as atualizações diretamente entre si, mas também enviam ao juiz para que este último possa ter o estado global do jogo.

Em cada ciclo do modo *PP*, P_A envia sua atualização (formada pelos comandos de P_A e um número indicando o ciclo) a P_B e R. Se P_A receber a atualização de P_B, P_A envia um pacote a R, contendo o número de identificação de P_B, um *hash* da atualização recebida e o intervalo de tempo entre o envio (para que R possa determinar um valor ótimo para a duração do ciclo) e a chegada da atualização. Já no modo *PRP*, P_A transmite a atualização a R que encaminha a P_B. Ademais, tanto no modo *PP* quanto no *PRP*, P_A envia os comandos que contenham informação secreta somente a R que só revela aos demais jogadores quando as informações não forem mais sigilosas, desse modo evitando trapaças por exposição de informação.

O estado de jogo de R é autoritativo, então as atualizações que R receber farão parte do estado global, e as atualizações que não chegarem a R não estarão inclusas no estado global. Portanto, caso uma atualização de P_A não chegue a R, mesmo que P_B a tenha recebido, R irá enviar uma mensagem a P_A com os dados necessários para tornar o estado local de P_A condizente com o estado global do jogo. Da mesma forma, se P_A enviar a R um *hash* da atualização de P_B que difira com o *hash* calculado a partir da atualização de P_B recebida por R (também incluindo o caso que R não recebe nenhuma atualização de P_B), R também irá enviar uma mensagem para recuperar o estado de P_A. Note que essas condições foram impostas para prevenir trapaças por incosistência.

O protocolo RACS também define que P_A e P_B no modo *PP* devem voltar ao modo *PRP* caso: P_A e P_B saiam da área de interesse um do outro; P_A ou P_B receba menos que uma certa fração das últimas atualizações enviadas pelo outro jogador; P_A ou P_B fique mais de um valor definido de ciclos consecutivos sem receber atualições do outro. A primeira condição surgiu para diminuir a banda, já a segunda e a terceira foram criadas para jogadores mal-intencionados, jogadores com latência alta ou com alta taxa de perda de pacotes

[2] *Network Time Protocol*, protocolo de sincronização dos relógios de computadores em rede

não prejudicarem a experiência do jogo do outro jogador, assim evitando trapaças por supressão de atualização, atraso fixo, e oponentes cegos.

No caso em que nenhum jogador tenha problemas na rede e que ninguém tente trapacear, todos jogadores operam no modo *PP* e o juiz não precisa fazer reconciliações nos estados de jogo dos jogadores, então a largura de banda e o atraso ficam muito próximos com os valores do protocolo *Peer-to-Peer* padrão. Já no caso no qual todos os jogadores trapaceiam ou tenham problemas na rede, todos ficaram no modo *PRP*, prevenindo as trapaças exclusivas do modelo *Peer-to-Peer* e produzindo valores de atraso e largura de banda assim como o protocolo Cliente/Servidor padrão.

Capítulo 5

Conclusão

O modelo de rede *Peer-to-Peer* no contexto de jogos multijogadores em rede pode ser mais adequado que o modelo Cliente/Servidor em situações em que a latência entre jogadas devem ser mínima ou quando não há garantido a presença de um nó computacional confiável. Apesar disso, os protocolos de modelo *Peer-to-Peer* são mais complexos, o que torna suas vulnerabilidades menos óbvias.

Dessa forma, esse trabalho exibiu uma classificação das trapaças em jogos digitais e explicou o funcionamento de cada um dos tipos de trapaça, mostrando situação de exemplo em que cada trapaça ocorre. Esse estudo das trapaças possibilitou estudar algumas vulnerabilidades de protocolos de sincronização em jogos, e entender como certos protocolos efetivamente previnem e detectam trapaças em jogos de modelo *Peer-to-Peer*.

Além disso, alguns dos protocolos estudados podem impactar negativamente o ritmo do jogo e, portanto, também foram analisados protocolos que tentam prevenir trapaças de maneira eficiente, mas sem influenciar muito na experiência de jogo.

Ademais, se há algum tipo de confiança entre os jogadores, por exemplo, os jogadores se conhecem e sabem que pelo menos um deles não irá tentar trapacear, então um certo protocolo para prevenir nessa situação pode não ser totalmente necessário.

A escolha do modelo de rede *Peer-to-Peer* pode ajudar a reduzir o processamento nos servidores do jogo ou a descentralizar o controle do jogo. Esses fatores podem ser atraentes para as comunidades de jogos de código livre que, em geral, não podem custear máquinas sofisticadas acabam optando em deixar o nó servidor em máquinas mais simples ou numa das máquinas de um dos jogadores. Porém, jogos desses tipos são mais fáceis de modificar, logo mais fáceis para trapacear. Portanto, esse trabalho pode ser usado para encontrar vulnerabilidades e melhorar os protocolos em jogos de código livre.

5.1 Desafios do trabalho

Para esse trabalho, foi feito um código para tentar simular e comparar alguns dos protocolos estudados, porém não chegou a ser implementado a ponto de produzir resultados

relevantes para a monografia. O código dessa simulação está disponível sob a licença GPL 3.0 em https://gitlab.com/LeaoLuciano/p2p-simulations.

Nesse simulador, foi implementado dois protocolos simples do modelo Cliente/Servidor e *Peer-to-Peer*, e quatro dos protocolos estudados. É possível simular uma quantidade de até dez jogadores e medir o intervalo de tempo entre os *frames* e o tamanho total de *bytes* transmitidos na rede. Para finalizar o código, faltou implementar alguns dos protocolos estudados, protocolos modificados para trapacear e métodos para medir se as trapaças efetivamente foram prevenidas.

5.2 Próximos passos

A finalização do desenvolvimento do simulador pode ser usado para comprovar quantitativamente os resultados obtidos nessa monografia. Ademais, é possível aproveitar parte do código do simulador para criar uma biblioteca ou arcabouço que reúne as implementações dos protocolos para que sejam utilizados no desenvolvimento de jogos.

Além disso, esse trabalho abre espaço para estudos de casos que aplicam os protocolos estudados em jogos de variados gêneros para analisar a influência do protocolo na experiência de jogo.

Referências

[Baughman e Levine 2001] Nathaniel E Baughman e Brian Neil Levine. "Cheat-proof playout for centralized and distributed online games". Em: *Proceedings IEEE INFOCOM 2001. Conference on Computer Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society (Cat. No. 01CH37213)*. Vol. 1. IEEE. 2001, pgs. 104–113 (citado na pg. 17).

[Baughman, Liberatore *et al.* 2007] Nathaniel E Baughman, Marc Liberatore e Brian Neil Levine. "Cheat-proof playout for centralized and peer-to-peer gaming". Em: *IEEE/ACM Transactions On Networking* 15.1 (2007), pgs. 1–13 (citado nas pgs. 18, 19).

[Corman *et al.* 2006] Amy Beth Corman, Scott Douglas, Peter Schachte e Vanessa Teague. "A secure event agreement (sea) protocol for peer-to-peer games". Em: *First International Conference on Availability, Reliability and Security (ARES'06)*. IEEE. 2006, 8–pp (citado nas pgs. 13, 20).

[GautheierDickey *et al.* 2004] C GautheierDickey, Daniel Zappala, Virginia Lo e J Marr. "Low-latency and cheat-proof event ordering for distributed games". Em: *Proc. NOSSDAV*. Vol. 4. Citeseer. 2004, pgs. 134–139 (citado na pg. 20).

[Jamin *et al.* 2003] S Jamin, E Cronin e B Filstrup. "Cheat-proofing dead reckoned multiplayer games". Em: *Proc. of 2nd international conference on application and development of computer games, Hong Kong*. Vol. 67. Citeseer. 2003 (citado nas pgs. 19, 20).

[Nystrom 2014] Robert Nystrom. *Game programming patterns*. Genever Benning, 2014 (citado nas pgs. v, 5).

[Pellegrino e Dovrolis 2003] Joseph D Pellegrino e Constantinos Dovrolis. "Bandwidth requirement and state consistency in three multiplayer game architectures". Em: *Proceedings of the 2nd workshop on Network and system support for games*. 2003, pgs. 52–59 (citado na pg. 21).

[Steed e Oliveira 2010a] Anthony Steed e Manuel Fradinho Oliveira. "Chapter 10 - requirements". Em: *Networked Graphics*. Ed. por Anthony Steed e Manuel Fradinho Oliveira. Boston: Morgan Kaufmann, 2010, pgs. 313–353. ISBN: 978-0-12-374423-4. DOI: https://doi.org/10.1016/B978-0-12-374423-4.00010-0. URL: https://www.sciencedirect.com/science/article/pii/B9780123744234000100 (citado nas pgs. v, 5, 6).

[Steed e Oliveira 2010b] Anthony Steed e Manuel Fradinho Oliveira. "Chapter 11 - latency and consistency". Em: *Networked Graphics*. Ed. por Anthony Steed e Manuel Fradinho Oliveira. Boston: Morgan Kaufmann, 2010, pgs. 355–392. ISBN: 978-0-12-374423-4. DOI: https://doi.org/10.1016/B978-0-12-374423-4.00011-2. URL: https://www.sciencedirect.com/science/article/pii/B9780123744234000112 (citado na pg. 4).

[Webb e Soh 2008] Steven Webb e Sieteng Soh. "A survey on network game cheats and p2p solutions". Em: *Australian Journal of Intelligent Information Processing Systems* 9.4 (2008), pgs. 34–43 (citado nas pgs. 1, 11).

[Webb, Soh e Lau 2007] Steven Webb, Sieteng Soh e William Lau. "Racs: a referee anti-cheat scheme for p2p gaming". Em: *Proceedings of the 17th international workshop on Network and operating systems support for digital audio and video*. Association for Computing Machinery (ACM). 2007, pgs. 34–42 (citado nas pgs. 8, 21).

[J. Yan e Randell 2009] Jeff Yan e Brian Randell. "An investigation of cheating in online games". Em: *IEEE Security & Privacy* 7.3 (2009), pgs. 37–44 (citado na pg. 7).

[J. J. Yan e Choi 2002] Jianxin Jeff Yan e Hyun-Jin Choi. "Security issues in online games". Em: *The Electronic Library* (2002) (citado na pg. 7).

[Yilmaz e Arslan 2015] Mustafa Harun Yilmaz e Hüseyin Arslan. "A survey: spoofing attacks in physical layer security". Em: *2015 IEEE 40th Local Computer Networks Conference Workshops (LCN Workshops)*. IEEE. 2015, pgs. 812–817 (citado na pg. 13).