**CENTRO UNIVERSITÁRIO CARIOCA**
**BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO**

**ANDRÉ FELIPE DOS SANTOS**

# DESENVOLVIMENTO DE JOGO DIGITAL USANDO BLOCKCHAIN

RIO DE JANEIRO
2019

ANDRÉ FELIPE DOS SANTOS

# DESENVOLVIMENTO DE JOGO DIGITAL USANDO BLOCKCHAIN

Trabalho de conclusão de curso apresentado ao Centro Universitário Carioca como requisito para a obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Prof. Anderson Fernandes Pereira dos Santos

RIO DE JANEIRO
2019

ANDRÉ FELIPE DOS SANTOS

# DESENVOLVIMENTO DE APLICAÇÃO GAMIFICADA COM BLOCKCHAIN

Monografia apresentada ao Curso de Ciência da Computação do Centro Universitário Carioca como requisito para a obtenção do Título de Bacharelado em Ciência da Computação.

Aprovado em _____ de _____ de 2019.

**BANCA EXAMINADORA**

_____
Prof. Anderson Fernandes Pereira dos Santos, D. Sc. – Orientador
**Centro Universitário Carioca**

_____
Prof. André Luiz Avelino Sobral, M.Sc.
**Centro Universitário Carioca**

_____
Prof. Alberto Tavares da Silva, D. Sc.
**Centro Universitário Carioca**

Agradeço a minha família, por todo o apoio e incentivo nos estudos e em seguir os meus sonhos, especialmente ao meu tio e padrinho, Angelo, por ter despertado em mim a paixão por jogos.

A todos os professores que participaram da minha caminhada, compartilhando seu conhecimento comigo.

E ao Centro Universitário Carioca, pela oportunidade de cursar a graduação de Ciência da Computação.

## AGRADECIMENTOS

Agradeço inicialmente a Deus, que me deu forças e perseverança para permanecer firme nos meus objetivos, mesmo nos piores momentos.

Em seguida, a minha mãe Adriana Pereira dos Santos e meu pai Edson Pereira dos Santos que sempre se esforçaram ao máximo para garantir uma educação de qualidade para mim e minhas irmãs.

A minha família que sempre me apoiou, especialmente meu tio e padrinho Angelo Pereira, responsável por me apresentar o mundo dos vídeo games e sempre acreditou no meu potencial.

Aos meus colegas de trabalho do TCE, em especial ao Matheus Fernal e ao Fábio Neves, por terem me orientado nesses últimos dois anos, compartilhando comigo todo conhecimento de TI que eles possuem, e ao Fulvio Longhi que me apresentou o tema Blockchain, permitindo assim que eu pudesse escolher esse tema para meu trabalho de conclusão.

A todos os professores que participaram da minha formação ao longo da vida, inclusive aos professores do Centro Universitário Carioca que, ao longo desses 4 anos e meio, compartilharam comigo todo seu conhecimento e me tornaram capaz de desenvolver este trabalho final.

Em especial, ao meu orientador Anderson Fernandes, por ter me orientado e compartilhado comigo seus conhecimentos, me mostrando o caminho para concluir este trabalho.

Por fim, ao Centro Universitário Carioca que disponibilizou minha bolsa de estudos e permitiu que, hoje, eu seja um cientista da computação.

## RESUMO

Esta pesquisa busca apresentar uma alternativa ao modelo tradicional Cliente-Servidor empregado em jogos online que, apesar de seus múltiplos benefícios, ainda possui pontos negativos, como a necessidade de investimento monetário por parte dos desenvolvedores para manter os servidores, e a possibilidade de os jogadores perderem todos os seus ativos adquiridos caso os servidores parem. Para tal, foi criado um jogo que opera em conjunto com uma blockchain, a fim de manter um registro das jogadas feitas. Verificou-se que é possível manter um jogo persistido em uma blockchain, gerando transações a cada jogada e tendo seus blocos minerados pelos jogadores, sem gerar custos a equipe que desenvolveu o jogo. Conclui-se que esta é uma prática promissora para se ter jogos online com menos custo e mais seguros, mesmo que mais pesquisas na área sejam necessárias antes que seja possível aplicar este conceito em produtos reais.



**Palavras-chave**: blockchain; jogos; jogos online; dominó; peer-to-peer;

## ABSTRACT

This research seeks to present an alternative to the traditional model Client-Server used in online games, that despite its benefits, still has negative points, such as a need for monetary investment by developers to maintain the servers, and players possibility to lost all their ownership in case that server stops. For that, a game that works with a blockchain has been created, to keep track of the moves made. It is concluded that this is a promising practice to issue for less costly and safer online games, even that more research in the area is required before it is possible to use this concept in real products.



**key-words**: blockchain; games; online games; Domino; peer-to-peer;

## LISTA DE ILUSTRAÇÕES

## LISTA DE QUADROS

## LISTA DE ABREVIATURAS E SIGLAS

| | |
|---|---|
| CEO | Chief Executive Officer |
| DPoS | Delegate Proof of Stake |
| GDD | Game Design Document |
| GUI | Graphical User Interface |
| HLAPI | High Level Application Programming Interface |
| HP | Health Points |
| HUD | Heads Up Display |
| LLAPI | Low Level Application Programming Interface |
| MMORPG | Massive Multiplayer Online Role Playing Game |
| MOBA | Multiplayer Online Battle Arena |
| MVVM | Model View ViewModel |
| OTEE | Over The Edge Entertainment |
| P2P | Peer-to-Peer |
| PK | Primary Key |
| PoB | Proof of Burn |
| PoS | Proof of Stake |
| PoW | Proof of Work |
| QoS | Quality of Service |
| TCP | Transmission Control Protocol |
| TLAPI | Transport Level Application Programming Interface |
| TSS | Time Stamp Service |

| | |
|---|---|
| UC | Unidade Certificadora |
| UDP | User Data Protocol |
| UI | User Interface |

**SUMÁRIO**

# 1. INTRODUÇÃO

## 1.1. Contextualização

A indústria de jogos é uma das que mais cresce no mundo, tendo superado a indústria cinematográfica e musical segundo (BBC News, 2019). Sendo assim, essa indústria tem se adaptado e evoluído de diversas formas, atingindo cada vez mais pessoas ao redor do mundo.

Segundo um estudo da Globosat, todo aquele que joga vídeo game é um *Gamer*, que hoje se caracterizam por um grupo completamente heterogêneo, com pessoas de diferentes idades, sexo e estilo de vida (Globosat, 2019).

Dentro deste cenário, a indústria de jogos tem se adaptado a realidade de seus consumidores, se aproveitando de diversas tecnologias emergentes, como os *smartphones*. Como apontado pela Newzoo, cerca de 2.4 Bilhões de pessoas ao redor do mundo estarão jogando em dispositivos móveis em 2019 (Newzoo, 2019).

É possível mencionar também o uso da internet para criar experiências online onde os diversos jogadores são capazes de compartilhar, cooperar e/ou competir, permitindo que jogos como League of Legend e PlayerUnknown's BattleGrounds possam alcançar marcas de 8 e 3,2 milhões de jogadores simultâneos respectivamente, segundo a página oficial de League of Legends[1] e Steam Charts[2].

Nesse sentido, diversas soluções para tornar as experiências online melhores foram desenvolvidas ao longo dos anos, sendo uma delas o uso de arquitetura cliente-servidor, como apontado por Mark Claypool e Kajal Claypool:

> "O espectro dos jogos online tem mudado de algumas pessoas colaborando ou competindo em redes locais (LAN) em jogos com perspectiva de primeira pessoa como Doom da Id's software no início dos anos 90, para milhares de jogadores interagindo através da internet em uma grande variedade de jogos que vão de jogos de tiro em primeira pessoa e jogos de interpretação de papéis até jogos de estratégia e esportes. Este crescimento na popularidade dos jogos online é também refletida na correspondentemente grande quantidade de servidores espalhados através do mundo que dão suporte e hospeda milhares de jogadores jogando estes jogos. "(Claypool, M.; Claypool, K; 2006. 1f. tradução nossa).

[1] Disponível em <https://na.leagueoflegends.com/en/news/game-updates/special-event/join-us-oct-15th-celebrate-10-years-league> Acesso em: 17 Nov. 2019.

[2] Disponível em <https://steamcharts.com/app/578080> Acesso em: 17 Nov. 2019.

Segundo Cronin, Filstrup e Kurc (2001) “Jogos comerciais são ambos projetados sobre arquiteturas cliente servidor ou, menos frequentemente, sobre arquiteturas nó a nó”.[3]

Com o surgimento da tecnologia conhecida como *blockchain*, novas portas se abrem para as mais diversas áreas, e para os jogos não é diferente. Sendo uma tecnologia distribuída e segura (Nakamoto, 2008), pode oferecer recursos interessantes para experiências gamificadas online.

Um exemplo de como a *blockchain* pode ser um divisor de águas no mundo dos jogos online vem de uma obra de ficção cinematográfica chamada *Ready Player One* ou em português, Jogador Número Um, dirigido por Steven Spielberg (Warner Bros, 2018).

No filme, todo um universo digital online conhecido como OASIS é cenário de milhares de interações sociais entre jogadores, que cooperam e/ou disputam em diversos jogos e desafios, sendo recompensados com moedas que podem ser utilizadas tanto no jogo quanto no mundo real.

As tecnologias para tal existem hoje, capazes de tornar ações digitais (como série de cálculos feitos por um computador) um ativo que pode ser gasto tanto no ambiente digital quanto no mundo real, operando sobre uma *blockchain*. Um exemplo disso no mundo real é o vídeo “Pegando um uber com uma criptomoeda ganha em um video game”[4] (Enjin, 2019).

A trama do filme gira em torno da promessa do falecido criador do jogo Halliday, que diz que aquele que encontrar as três chaves que ele escondeu no universo do jogo, passaria a ter controle total sobre este universo.

Segundo Ruff Chain (2019), para que uma ação fosse executada automaticamente, mesmo após a falência dos interessados, seria necessária uma tecnologia que, uma vez escrito uma regra ou acordo, este seria imutável e automaticamente executado assim que condições pré-estabelecidas fossem satisfeitas. Esta tecnologia existe e é conhecida como *smart contracts*, onde segundo Atzei, Bartoletti e Cimoli (2017) são “programas de computador que podem ser executados corretamente por uma rede de nós mutuamente não confiáveis, sem a necessidade de uma autoridade externa confiável”.

---

[3] “Commercial games are either designed on top of client-server architectures or, less frequently, on top of peer-to-peer architectures”

[4] “Catching an Uber With Cryptocurrency Won in a Video Game”

Sendo assim, tanto a *blockchain* quanto as tecnologias relativas a ela (como os *smart contracts* por exemplo) apresentam um potencial considerável na criação de experiências *gamificadas* online ainda mais interessantes, onde este trabalho aprofunda este tema e mostra, de maneira superficial, como criar um jogo capaz de utilizar recursos disponibilizados pela *blockchain*.

## 1.2. Justificativa

No cenário atual, existem duas arquiteturas de rede mais comuns, como apontado por Huynh e Valarino (2019), sendo elas a arquitetura cliente servidor e a arquitetura nó-a-nó, também conhecida como *peer-to-peer* ou P2P.

Em uma típica implementação da arquitetura cliente servidor, a consistência do jogo é mantida pelo servidor onde, segundo Gambetta (2017) opera com um “servidor autoritativo e clientes burros”, onde o servidor central controla toda a simulação e os clientes são meros observadores com capacidade de envio de inputs, como mostrado na figura 1.

A existência desse terceiro confiável que controla a simulação representa um ponto central de falha, permitindo que um ataque bem sucedido elimine a disponibilidade deste servidor e;ou danifique a integridade dos dados armazenados e transmitidos por este servidor.

Huynh e Valarino (2019) apontam como alguns pontos negativos como a alta latência de rede para o servidor, perda de pacotes, congestionamento na rede ou até mesmo fatores externos fora do controle, afetando o desempenho do servidor.

Como apontado por Oluwatosin, os servidores precisam ser projetados sobre altos padrões a fim de serem confiáveis e performáticos, contribuindo para que eles sejam caros (Oluwatosin, 2014), assim limitando o acesso de desenvolvedores com pouco recurso financeiro.

Outro ponto levantado por Oluwatosin (2014) nas arquiteturas cliente servidor diz respeito a segurança, onde o cliente é facilmente acessado pelo servidor, expondo o sistema cliente a vários problemas, além das mensagens trocadas entre cliente e servidor também acarretarem em uma série de desafios a fim de garantir a segurança.

Por fim, todo o progresso dos jogadores (traduzido em posses de itens, avatares, habilidades, etc) na prática pertencem a empresa que criou e mantém o

jogo. Se um dia esta empresa desligar os servidores ou estes forem danificados, todo esse progresso e tempo investido no jogo será perdido, algo que é extremamente frustrante para os jogadores, como aconteceu com o jogo online Grand Chase, que teve seus servidores desligado pela KOG, distribuidora do jogo (Vieira, 2015).

Como apontado pela revista Forbes (2018), uma *blockchain* não é 100% segura, assim como qualquer outra tecnologia. Entretanto, ela é projetada para ser imutável, a prova de violações e democrática, se baseando em três pilares, sendo eles a descentralização, a criptografia e o consenso.

A descentralização permite que, por exemplo, os desenvolvedores não tenham que gastar nada com a manutenção ou aluguel de um servidor dedicado para atender as requisições dos usuários pois, já que os responsáveis por manter a rede são os nós participantes, ou, analogamente ao modelo cliente servidor, os clientes.

Além disso, para que os ativos digitais dos jogadores viessem a ser destruídos, seria necessário que toda a rede de nós fosse abaixo, em outras palavras, enquanto houver pelo menos um nó disponível, todo o progresso do jogador estará a salvo.

Sendo uma *blockchain*, todas as transações são auditáveis, possibilitando que, caso exista dúvidas sobre a veracidade de um item por exemplo, seria possível verificar todas as transações cujas quais este item foi incluído e verificar a sua origem.

Esse recurso é excelente para evitar problemas como o enfrentado por Fallout 76 (2019) conhecido como duplicação de itens. Em um *Hotfix*[5] *Notes* publicado pela Bethesda - criadora do jogo - uma série de *exploits*[6] que permitiam jogadores duplicar itens sobre a sua posse foi corrigido. Sendo capaz de rastrear a origem de um item, é possível determinar se ele é fruto de ações legítimas do jogo ou do uso de algum *exploit*, permitindo que outros jogadores não aceitem o item durante um troca.

---

[5] Segundo Chopra e col. (2014), são pequenos pacotes de software com correções para problemas como defeitos no código fonte, usabilidade e; ou performance, disponibilizados para os usuários de um sistema.

[6] “Descreve uma condição da programação, situação e/ou aberração do jogo e jogador padrão que poderia dar a um membro trapaceiro uma vantagem sobre um jogador honesto” (Pryor, 2009, tradução nossa)

Além disso, o processo pelo qual uma blockchain cresce e novas transações são inseridas é conhecido por mineração. Este processo é importante pois, através dele, novas moedas são geradas e postas em circulação, moedas estas que possuem valor no mundo real, como explicado no tópico 2.3 Funcionamento de uma blockchain.

Sendo assim, mecanismos de mineração que utilizam de ações lúdicas para comprovar o esforço empregado nesta tarefa, como proposto pelo grupo Xaya (2019), permitem que, de forma simplória, jogadores sejam pagos por jogar.

## 1.3. Objetivo

O objetivo deste trabalho é criar uma aplicação gamificada produzida na *game engine* Unity capaz de utilizar recursos de uma *blockchain*, a fim de desenvolver e disponibilizar uma biblioteca que permita qualquer desenvolvedor independente com interesse em utilizar *blockchain* em seus jogos, seja capaz de fazer isso de maneira simplificada.

Para tal, além de uma aplicação que implementa todas as funcionalidades de uma *blockchain*, é necessária também uma aplicação gamificada que utilize recursos da *blockchain* para persistir os seus dados e de uma aplicação que forneça serviços de rede para que os nós da *blockchain* se comuniquem.

# 2. BLOCKCHAIN

## 2.1. A história das tecnologias precursoras do blockchain

*Blockchain* é um conceito que se popularizou 2008 com o manifesto do *bitcoin*, escrito por Satoshi Nakamoto, onde descreve uma série de procedimentos e contramedidas que tornam possível a existência de um livro razão eletrônico, armazenado em uma rede distribuída, capaz de ser seguro em um meio naturalmente inseguro.

O trabalho de Nakamoto (2008) é evidentemente a evolução de diversas tentativas e conceitos elaborados anos antes, por diversos pesquisadores, criptógrafos, acadêmicos, desenvolvedores e/ou ativistas que valorizavam a privacidade, como Cynthia Dwork e Moni Naor, além de um grupo chamado *Cypherpunk*, onde David Chaum, Wei Dai, Adam Back e Nick Szabo discutiram sobre privacidade digital, anonimidade e proporem ao longo dos anos 90 e princípio dos 2000 diferentes modelos de sistemas monetários digitais.

Além deles, existe o trabalho dos pesquisadores Stuart Harber e Scott Stornetta (1991), que em contraste com os nomes citados acima, elaboraram uma abordagem mais técnica para um outro tipo de problema que também foi fundamental para a criação do *blockchain*.

### 2.1.1. Pagamentos anônimos e seguros na rede

Para fundamentar o *blockchain*, é necessário falar de economia e tecnologia. Sendo assim, o primeiro a ser mencionado é o Dr. David Chaum, criador do eCash (Wirdum, 2018).

Antes mesmo das pessoas terem computadores pessoais em suas casas, David já se preocupava com o futuro da privacidade *online*, e mais do que isso, ele criou dinheiro digital, capaz de preservar a privacidade das partes envolvidas.

A sua grande contribuição para a economia digital fica por conta do seu artigo “Assinaturas cegas para pagamentos não rastreáveis” onde, segundo David Chaum (1998) uma unidade central (um banco por exemplo) é incapaz de saber quem é o dono do dinheiro, mas é capaz de garantir que o dinheiro é verdadeiro pois a própria unidade central já o validou uma vez.

Imaginando um cenário onde cada solicitação de transação representa o débito de uma unidade monetária daquele que efetuou a solicitação, e que um cliente A deseja efetuar uma transação para um cliente B.

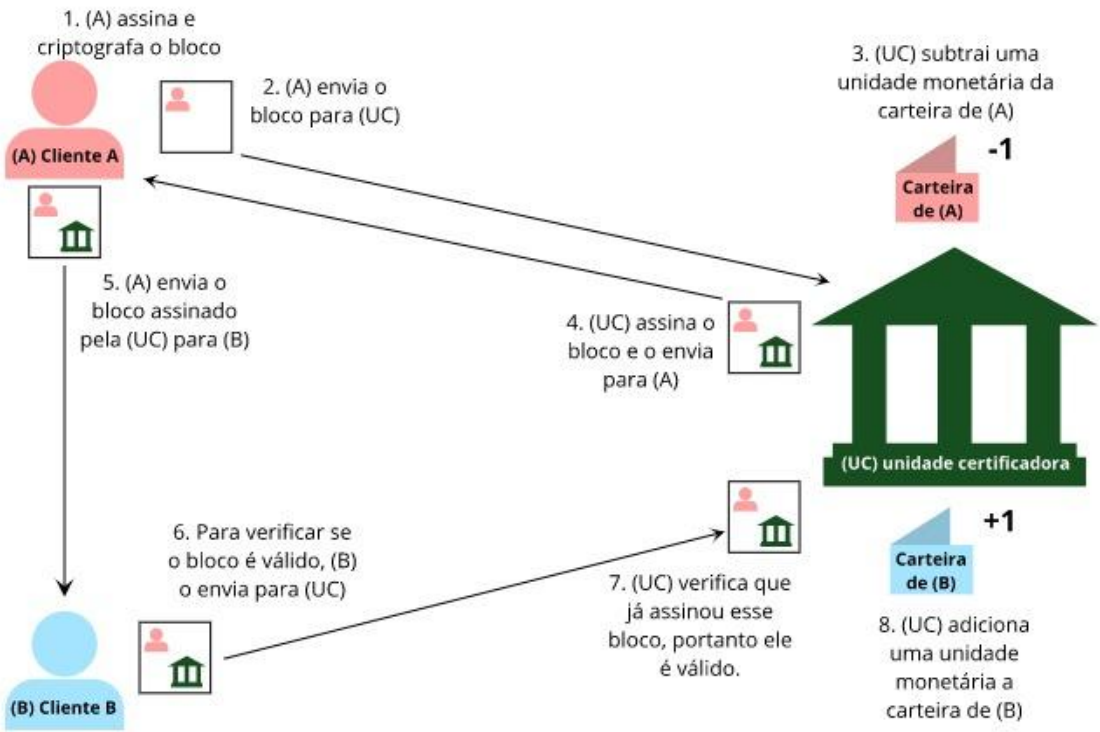


Figura 1: Processo de assinatura às cegas.
Fonte: Própria (2019)

No processo apresentado na figura 1, um cliente A deseja efetuar uma transação anônima de uma unidade monetária para o cliente B, logo envia um pacote criptografado para uma unidade certificadora (UC).

A unidade certificadora assina a solicitação às cegas, efetua o débito de uma unidade monetária da carteira de A e retorna o pacote assinado para A.

O cliente A envia o pacote assinado pela unidade certificadora (UC) para B, que para saber se o pacote é verídico, o submete a unidade certificadora.

A unidade certificadora recebe o pacote de B, verifica que ele está assinado pela própria unidade certificadora e credita uma unidade monetária a carteira de B.

A unidade certificadora sabe apenas que A efetuou o saque de uma unidade monetária e que B recebeu um pacote. A unidade certificadora é incapaz de determinar a origem do pacote submetido por B, é capaz somente de confirmar que ela mesma assinou o pacote e que, portanto, alguém foi debitado.

Apesar deste processo precisar de um terceiro confiável, permite que as partes da transação não tenham que se identificar, e permite a verificação do duplo gasto da moeda, uma vez que, se um pacote é submetido novamente a unidade certificadora, ela é capaz de determinar que aquele pacote já foi processado e que as partes já foram debitadas e creditadas.

A independência de um terceiro confiável e a possibilidade de se realizar transações entre nós anônimos de forma segura são característica vitais na *blockchain* dos dias atuais, sendo essas características possíveis graças às contribuições de David Chaum.

Além disso, foi o ponto de partida para a criação da *DigiCash*, a empresa que administrou o primeiro dinheiro virtual entre a década de 80 e 90, e que após a falência, foi uma importante inspiração para o *Cypherpunks*.

### 2.1.2. Livro razão distribuído

Entre os membros dos *Cypherpunks*, Wei Dai empenhou um papel importante na evolução das *cryptocurrencies,* que foi a criação do *b-money* em 1998, um protocolo para trocas financeiras e execução de contratos por pseudônimos.

Dentro de uma rede *peer-to-peer*, todos os nós possuem uma cópia do livro-razão, e sempre que uma nova transação é feita, todos os nós atualizam seus registros.

O funcionamento é baseado em criptografia de chaves assimétricas, onde sempre que um cliente A deseja enviar um montante para o cliente B, o cliente constrói a transação especificando o montante e usando a chave pública de B, em seguida assina a transação com a sua chave privada e por fim envia a transação para todos os nós.

Tal abordagem descentralizada impediria que qualquer entidade pública fosse capaz de bloquear ou taxar as transações.

O problema desta abordagem é que ela não é imune ao problema do gasto duplo, onde segundo Rosenfeld Meni (2012) “é de fato uma tentativa bem sucedida de primeiramente convencer um mercador que a transação foi confirmada, e então convencer toda a rede de receber uma outra transação” (Rosenfeld, 2012).

Em outras palavras, um indivíduo é capaz de gastar uma mesma moeda duas ou mais vezes. Para contornar esse problema, Wei criou o B-Money versão 2.

Nesta versão, os usuários do sistema seriam divididos em duas categorias - usuários regulares e usuário servidores.

Usuários servidores seriam responsáveis por manter o livro razão, enquanto os usuários regulares efetuariam transações e, caso seja necessário verificar alguma, basta consultar algum dos usuários servidores.

Em caso de conflito com alguma das transações, os clientes a rejeitariam e esta seria descartada.

Além disso, os servidores teriam que se comprometer criptograficamente e publicar periodicamente os bancos de dados sob sua responsabilidade.

Essa proposta é similar ao sistema de *proof of stake*, que será explicado na seção 2.4.2 - Algoritmos de consenso, e além desse sistema, Dai também propôs uma versão primitiva de *contratos inteligentes* - contratos virtuais capazes de serem executados de forma automática, sem a necessidade de um terceiro para efetuar a transação entre as partes (Nicola et al., 2017).’

As propostas de Wei Dai nunca foram implementadas, e em função disso, foram incapazes de prever uma série de problemas que impossibilitaram o B-Money de viralizar tanto quanto o *bitcoin*, como a falta de privacidade ou à possível centralização da moeda, entretanto, mostrou que a descentralização era necessária, sendo esta sua contribuição para o que hoje é conhecido como *blockchain*,

### 2.1.3. Proof of work para combate de spam

Antes de falar das contribuições de Adam Back para o que hoje é conhecido como *blockchain* e criptomoedas, temos de falar da criação do primeiro sistema que, anos depois, seria conhecido como *proof of work*, criado por Cynthia Dwork e Moni Naor, pesquisadores da IBM, a fim de reprimir o *spam*.

Através de artigo publicado, Dwork. C e Naor. M explicam que com “a facilidade e o baixo custo de enviar e-mail eletrônico, e em particular a simplicidade de enviar a mesma mensagem para muitos grupos, quase sempre é convidativo ao abuso”. (Dwork. C e Naor. M, 1992, p. 2).

A proposta era voltada para serviços de e-mail, com foco no combate ao *spam*, onde o remetente de um e-mail tem que anexar dados em todo envio, dados esses que seriam o resultado de um problema matemático exclusivo do e-mail em questão.

A proposta não afetaria o uso comum do e-mail, uma vez que o problema matemático seria projetado para consumir não mais que poucos segundos do computador. Porém, quando a quantidade de e-mails a ser enviada cresce de maneira exponencial, assim como é feito por anunciantes, hackers e todos aqueles que se beneficiam do *spam*, a quantidade de poder de processamento necessário para enviar esses e-mails também cresce de forma exponencial.

Posteriormente, o Dr. Adam Back seria responsável por trazer ao mundo o *HashCash*, incluindo sua descrição e implementação, semelhantes ao trabalho de Dwork e Naor, mas baseado em colisão parcial de *hash*[7] (Back, 2002), processo criptográfico onde uma entrada X é transformada em uma saída Y através de procedimentos matemáticos.

A única forma de se conseguir a entrada X é através da força bruta, inserindo diversos *inputs* na função *hash* até se encontrar aquele que gerou a saída Y.

O *HashCash* foi projetado inicialmente em 1997 com o mesmo objetivo do trabalho de Dwork e Naor de combater o *spam*. Entretanto, sua implementação inclui, além dos dados como remetente, destinatário e horário, um valor conhecido na criptografia por “*nonce*”.

O *nonce* é uma redução para “*number once*”, sendo definido como um valor criado de forma aleatória e usado apenas uma vez.

Todos os dados, incluindo o *nonce*, quando submetidos ao algoritmo de *hash*, são anexados ao *e-mail* e enviados ao destinatário, que aceita o *e-mail* caso o *hash* seja válido.

---

[7] Importante mencionar propriedades como a irreversibilidade (dada a saída Y, é impossível se executar um procedimento que retorne a entrada X), a capacidade de gerar a saída Y sempre que o *input* X for inserido na função e a impossibilidade de se prever qual será o resultado Y de uma entrada X.

Repare, entretanto, que para o *hash* ser válido, sua versão binária deve conter um número pré-determinado de zeros no início. Como o resultado da função *hash* é impossível de ser previsto, a única maneira de se gerar um *hash* válido é alterando o valor do *nonce* e executando o algoritmo novamente - ou seja, força bruta - comprovando assim o uso de poder de processamento.

Enquanto o sistema de Dwork e Naor permite que computadores com maior poder computacional sejam capazes de resolver o quebra cabeça matemático mais rápido que computadores mais modestos, a abordagem de Back não permite esse cenário, dada a aleatoriedade do resultado correto para o envio da mensagem.

Essa característica é interessante pois na abordagem de Dwork e Naor, aqueles que desejam praticar algum tipo de atividade abusiva, poderiam executa-la baseado no poder computacional que possuem. A abordagem de Back garante que todos têm as mesmas capacidades independentemente de poder computacional.

Como descrito por Wirdum na sua série de artigos chamados "*The Genesis Files*" (Wirdum, 2018), publicado na *Bitcoin Magazin*e, de forma análoga, o sistema de Dwork e Naor funcionaria como uma corrida, onde caso um corredor fosse melhor preparado fisicamente, ele sempre ganharia, já o sistema de Back seria comparado com uma loteria, onde alguém que compra mais bilhetes não necessariamente ganharia com mais frequência que os outros competidores.

É importante mencionar que mesmo com a aleatoriedade, o *HashCash* de Back pode ser calculado cada vez mais rápido, conforme o poder de processamento dos computadores aumenta em função do tempo, permitindo que os usuários gerassem cada vez mais provas de trabalho a longo prazo.

Isso foi o fator chave para que o *HashCash* não viesse a se tornar uma moeda digital, como descrito por Back em 2002 através do paper público "*HashCash - A Denial of Service Counter-Measure*" (Back, 2002). Apesar do conceito de *proof of work* garantir que existisse escassez, uma vez que seja necessário efetuar uma tarefa computacional para gerar uma prova, o fato dessa prova tender a ser gerada com mais facilidade conforme o tempo passa indica que a moeda irá inflacionar no futuro.

### 2.1.4. Moeda digital descentralizada

Em 1998 Nick Szabo efetuou o primeiro trabalho relativo a uma moeda digital descentralizada, conhecida como Bit Gold. Nunca foi implementada, mas considera-se que esta seja o precursor do conhecido *bitcoin*.

O caminho que Szabo percorreu inclui participação no *Cypherpunks*, um ensaio chamado “Shelling Out: The Origins of Money”* (Szabo, 2002), um interesse por sistemas bancários livres, onde cada banco emitia a sua moeda e o livre mercado se encarregava de escolher qual delas usaria e uma período trabalhando na *DigiCash*, empresa criada por David Chaum, onde aprendeu com a ascensão e queda do *eCash*.

“O problema, resumidamente, é que o nosso dinheiro atualmente depende de um terceiro confiável para ter valor.” (N. Szabo, 2005, *Unenumerated*[8])

Seus estudos o levaram a conceituar como seria o dinheiro ideal. Deveria ser invulnerável a perda e roubo, ser um bem valioso através do seu alto valor e precisaria ser facilmente mensurado através de processos simples, tal como observações ou medições simples.

Apesar de nascer em 1998, o Bit Gold só veio a se tornar público em 2005, quando Szabo publicou sobre em seu blog conhecido com *Unenumerated*, se tratando de uma moeda que usava de vários artifícios para se concretizar no meio digital.

Para garantir a escassez e, tal qual o ouro, garantir o seu valor, Szabo usou o conceito de *proof of work*, elaborado por Adam Back e seu *HashCash* (Back, 2002) - uma vez que este modelo exigia recursos do mundo real (processamento) para provar a existência das moedas.

O mesmo conceito de um *hash* com um número específico de zeros permanece na implementação de Szabo mas a diferença está no fato de, após se descobrir um *hash* válido, este seria utilizado para a próxima transação. Logo, o *BitGold* se tornaria uma cadeia de blocos conectados por *hashes*.

Um usuário que encontrou o *hash* teria posse desse *hash*, tal qual alguém que encontra uma pepita de ouro tem. Esta relação de pertinência seria estabelecida através da chave pública do seu respectivo criador, que é inserida em um registro de propriedade digital, permitindo também a transferência de propriedades para outros membros da rede.

---

[8] Blog escrito e mantido pelo próprio Nick Szabo chamado unenumerated (Szabo, 2005).

O livro razão do Bit Gold também seria distribuído, onde um grupo de usuário servidores armazenam uma relação dos *hashes* existentes e suas respectivas chaves públicas, semelhante a proposta de Wei Dai.

Entretanto, é no processo de manter os servidores atualizados que reside a diferença entre a proposta dos dois. Enquanto Dai utiliza o sistema de prova de participação (*proof of stake* - explicado na seção “algoritmos de consenso”), Szabo usa um “Sistema Quorum Bizantino”.

Quorum é definido como o “Número mínimo de membros necessários em uma assembleia para que as decisões tomadas sejam válidas: o *quorum* de uma assembleia legislativa é de metade mais um do total de seus membros.” (DICIO, 2019).

Dentro do contexto da ciência da computação, existem os sistemas Quorum, que são “[...] comumente usados para manter a consistência de dados replicados em um sistema distribuído.” (Gilbertz, S.; Malewicz, G.; 2004).

E do problema dos generais bizantinos, descrito como:

> “[...] um grupo de generais bizantinos de um exército bizantino acampados com suas tropas ao redor de uma cidade inimiga. Se comunicando apenas por mensageiros, os generais devem concordar com um plano de batalha comum. Entretanto, um ou mais deles podem ser traidores que vão tentar confundir os outros. O problema é encontrar um algoritmo para garantir que os generais leais vão chegar a um acordo.” (Lamport, L.; Shostak, R.; Pease, M., 1982 , tradução nossa)

O problema dos generais bizantinos descreve um cenário comum em aplicações distribuídas, onde é necessário se estabelecer um consenso entre as partes, de forma que todas as unidades participantes estejam de acordo em relação a algum aspecto do sistema.

Neste cenário, integrantes da rede distribuída mal intencionados podem executar ações que levem dados errados para os nós. Portanto, encontrar um algoritmo que seja imune às ações dos integrantes mal intencionados é um desafio.

Soluções que envolvem Quorum permitem que, se uma quantidade X menor que metade da rede estiver *offline* ou falhar, o sistema permanecerá operando satisfatoriamente, enquanto o problema dos generais bizantinos descreve o cenário de comunicação entre servidores através de uma rede não confiável - como a internet por exemplo.

Apesar da proposta não ser 100% a prova de falhas, como exibido no artigo de Aaron van Wirdum (2018) - apontando que um ataque Sybil onde uma pessoa ou grupo tenta assumir o controle de uma rede criando diversas contas, poderia comprometer a rede - Szabo acreditava que em um cenário onde uma maioria desonesta poderia estar na rede, uma minoria honesta continuaria a gerar os registros corretamente e os usuários escolheriam os registros corretos.

Szabo também previu a possibilidade de inflação, uma vez que com o avanço do poder computacional, encontrar um *hash* válido seria cada vez mais fácil, permitindo que cada vez mais moedas entrassem em circulação, e a inflação aumentasse. A solução inicial é que quando um *hash* válido fosse descoberto, ele seria "carimbado" com a data atual, e isso definiria o seu valor. Um *hash* mais atual valeria menos que um *hash* mais antigo, uma vez que o *hash* mais atual foi encontrado mais facilmente.

Esse *design* adicionava um novo problema, relativo a fungibilidade[9], como o próprio Szabo apontou em seu blog em 2008.

"...os bits (a solução dos quebra cabeças) de um período (qualquer lugar entre segundos e semanas, digamos semanas) para o próximo não são fungíveis. " (Szabo. N; unenumerated, 2008)[10].

Em outras palavras, o dinheiro precisa ter hoje o mesmo valor que ele terá amanhã. Se uma unidade monetária é equivalente a cinco unidades de um bem hoje, essa mesma unidade monetária precisará ser equivalente as mesmas cinco unidades do mesmo bem, a fim de manter a ordem e praticidade do sistema financeiro.

A solução encontrada consistia em uma espécie de banco altamente auditável, que tinha o trabalho de reunir os *hashes* de diferentes períodos de tempo e com base no valor desses *hashes*, os reuniria em blocos até que esses blocos atingissem um valor pré determinado - blocos mais antigos incluiriam menos *hashes* que blocos mais atuais, mas ambos os blocos representam o mesmo valor - e por fim, estes blocos seriam divididos em unidades menores, que poderiam ser emitidas por esses bancos.

---

[9] "Coisas móveis que, por convenção das partes, podem ser substituídas por outras da mesma espécie, qualidade e quantidade, como o dinheiro, os cereais, o vinho etc". (Michaelis, 2019)

[10] "the bits (the puzzle solutions) from one period (anywhere from seconds to weeks, let's say a week) to the next are not fungible." (Szabo, 2008)

O *bitgold* de Szabo[11] foi o trabalho que mais se aproximou do *bitcoin* de Nakamoto, sendo necessário algum polimento para que esta moeda digital tecnicamente chegasse ao patamar do bitcoin. É válido mencionar que o trabalho de Szabo não foi mencionado no White Paper de Nakamoto, porém é evidente a semelhança entre ambos os conceitos.

### 2.1.5. Carimbo cronológico de dados

Outro conceito importante se fundamentou com Stuart Harber e Scott Stornetta em 1991, quando publicaram o artigo acadêmico “Como carimbar o tempo em um documento digital” (Harber, S.; Stornetta, W.S; 1991) descrevendo uma corrente de blocos conectados de forma criptológica, mais conhecido hoje por *blockchain*.

O estudo focava na solução de um problema específico: certificar-se de quando um dado qualquer (áudio, imagem, vídeo, texto) foi criado, além de se verificar se e quando este dado foi adulterado.

Eles alegaram que o problema é marcar o tempo atual no dado, e não no meio pelo qual ele existe ou é transmitido, e sendo assim, ambos propuseram meios computacionais práticos de se solucionar o problema, descritos a seguir:

> “Sempre que um cliente possui um documento para ser carimbado com a data atual, ele ou ela transmite o documento para um serviço que carimbe a data (**TSS**). O serviço registra a data e o tempo em que o documento foi recebido e retém uma cópia do documento por segurança. Se a integridade do documento do cliente for questionada, ele pode ser comparado com a cópia armazenada pelo **TSS**. Se são idênticos, é evidente que o documento não foi adulterado com uma data posterior a contida no registo **TSS.** Este procedimento de fato está de acordo com o requerimento central para o carimbo na data de um documento.” (Harber, S.; Stornetta, S; How to Time-Stamp a Digital Document. 1991. 13f. Artigo de jornal. tradução nossa). [12]

Entretanto, o estudo aponta que esta abordagem levanta uma série de preocupações, a saber:

---

[11] segundo Wirdum (2018), Szabo propôs em 1990 a criação de uma tecnologia conhecida hoje como Contratos inteligentes.

[12] “Whenever a client has a document to be time-stamped, he or she transmits the document to a time-stamping service …”

- **Privacidade** - Esta pode ser comprometida quando uma terceira entidade está interceptando os dados enquanto são transmitidos, e após a transmissão, eles estão disponíveis indefinidamente na TSS;
- **Largura de banda e armazenamento** - O tempo de upload necessário e o tamanho do arquivo em si podem ser proibitivos em termos de performance e tempo de resposta;
- **Competência** - Os dados podem ser corrompidos durante a transmissão para o TSS, ou podem ser corrompidos e/ou perdidos a qualquer momento dentro do TSS; e
- **Credibilidade** - Nada previne o TSS de ser influenciado por um ou um grupo de clientes a fim de carimbar no tempo um documento com data e tempo diferentes do que efetivamente ocorreu o carimbo.

O problema de privacidade e largura de banda e armazenamento são resolvidos através das funções *hash*. Como descrito por Haber e Stornetta, ao invés de enviar os dados x diretamente para o TSS, seria enviado o resultado da função *hash*(x) = y, onde o carimbo de tempo de x é equivalente ao de y.

Em seguida descrevem o uso de assinaturas, onde, ao enviar o *hash* para o TSS, este por sua vez insere o carimbo de tempo, assina os dados e por fim envia esse conjunto para o cliente, que pode validar todas as informações e saber se sua requisição foi processada.

Este procedimento resolve o problema de competência, e elimina a necessidade do TSS de armazenar registro.

Por fim, eles dão um par de soluções para o problema de credibilidade, onde uma inclui um TSS centralizado e inconfiável para produzir carimbos de tempo genuínos, de tal forma que carimbos de tempo falsos sejam difíceis de produzir, já a outra visa distribuir a confiança requerida entre os clientes do serviço.

O trabalho de Stornetta e Harber reforçam a necessidade de se ter uma corrente de blocos conectados através de hashes, mas inclui também o ato de carimbar a data e hora nos dados, permitindo que fosse possível se verificar quando um bloco foi criado.

Recurso este importante para definir, por exemplo, qual bloco deve ser inserido na blockchain quando mais de um bloco é encontrado e enviado para a

rede. No caso da blockchain, o bloco com o carimbo de tempo menor (mais antigo) é o escolhido.

## 2.2. A Blockchain e o Bitcoin

A princípio é importante mencionar que *bitcoin* e *Blockchain* representam conceitos diferentes, onde o *bitcoin* é a moeda digital proposta por Satoshi Nakamoto, que se popularizou em 2014 e *blockchain* é parte da solução proposta pelo mesmo para implementar o *bitcoin*.

"Uma versão puramente *peer-to-peer* de dinheiro eletrônico poderia permitir que pagamentos online fossem enviados diretamente de uma entidade para outra sem passar por uma instituição financeira." (Nakamoto .S; *bitcoin*: A Peer-to-Peer Electronic Cash System, 2008).

Um dos benefícios do *bitcoin* apontados por Nakamoto na sua implementação é a ausência de um terceiro confiável, como um banco ou uma instituição financeira por exemplo, que nos moldes atuais, é responsável por:

- Armazenar as informações dos correntistas;
- Manter o livro razão, que armazena o montante que cada pessoa possui;
- Acabar com o problema do gasto duplo;
- Oferecer serviços; e
- Garantir que as medidas legais do país de atuação sejam cumpridas.

Em contrapartida, os problemas envolvendo um terceiro confiável são:

- Esta entidade não necessariamente é confiável, uma vez que indivíduos internos podem manipular os dados para atingir interesses particulares;
- Taxas geralmente são cobradas por cada transação feita;
- A existência dessa entidade garante um ponto central de ataque e/ou falha; e
- Ações governamentais de censura ou controle podem ser executadas em função do caráter centralizado.

Pensando nas vulnerabilidades existentes no modelo centralizado e no modelo distribuído - o gasto duplo - Nakamoto propôs "uma solução para o problema

de gasto duplo usando um servidor distribuído que carimba no tempo para gerar computacionalmente prova cronológica da ordem das transações".

## 2.3. Funcionamento de uma blockchain

Por ser uma moeda, a operação fundamental do *bitcoin* é a transação, que é feita da seguinte maneira:

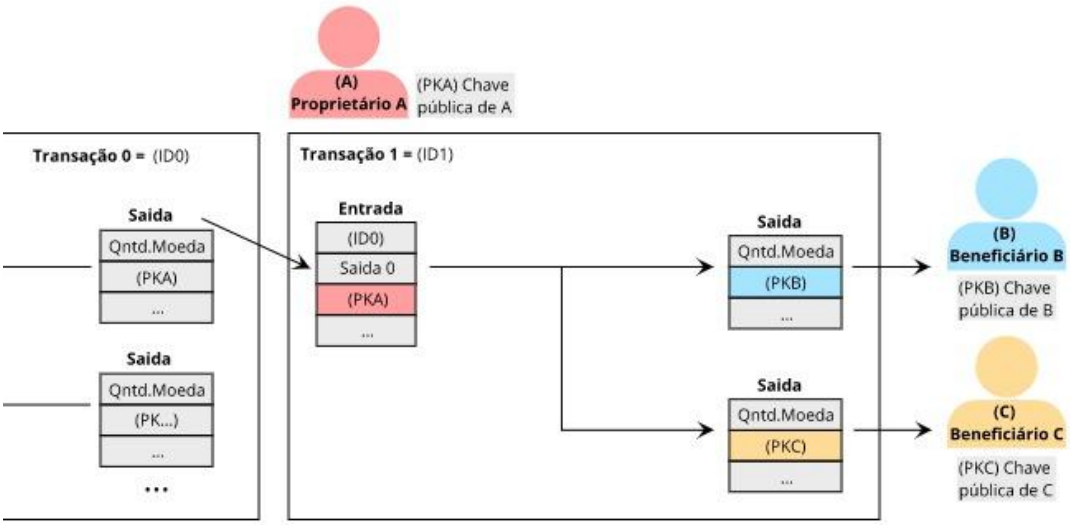


Figura 2:Transação no bitcoin.
Fonte: Própria (2019)

Cada transação recebe uma **entrada** da transação anterior, que na figura 2 é representada pela **transação 0**. O proprietário (**A**) gera uma nova transação com n **saídas** , cada uma representando uma **quantidade de moedas** a ser enviada para um **beneficiário** (B e C), identificado pela sua **chave pública** (PKB e PKC).

O total de moedas das saídas não pode ultrapassar a quantidade estabelecida na entrada, ou seja, o proprietário pode enviar em cada saída uma fração do total que recebeu na **entrada**, para diferentes beneficiários.

Essa abordagem não resolve o problema do gasto duplo. Em cenários comuns, uma entidade certificadora (terceiro confiável) seria responsável por

verificar se o dinheiro não foi gasto anteriormente verificando todas as transações anteriores.

Para prevenir o duplo gasto em uma rede sem o terceiro confiável, suponhamos a tentativa de efetuar uma transação com uma mesma moeda duas vezes, transação A efetuada antes da transação B.

As transações precisam ser anunciadas publicamente, para todos os nós da rede e é preciso um sistema onde todos os participantes da rede concordem a partir de um histórico único qual das transações chegou primeiro, A ou B, e assim aceitar a transação que veio primeiro - no caso a transação A - e ignorando a transação B. O beneficiário então precisa de uma prova de que a maioria dos nós aceitou a transação A.

A implementação do caso acima se inicia com um servidor[13] que carimbe o tempo nas transações, onde esse servidor escolhe um bloco (que contém vários itens), a data e hora atuais e o carimbo de tempo do bloco anterior e gera um *hash* com esses dados, gerando uma espécie de corrente, onde cada novo bloco reforça a existência do bloco anterior, uma vez que mantém o carimbo no tempo do anterior.

“Para implementar um servidor distribuído de carimbo, nós vamos precisar usar um sistema de prova de trabalho, semelhante ao *HashCash* do Adam Back“. (Nakamoto, S; *bitcoin* - a peer to peer …, tradução nossa)

A implementação da prova de trabalho do *bitcoin* usa um *nonce* que é incrementado no bloco até que o *hash* comece com um número de zeros requerido pela rede - processo este conhecido como minerar.

“Uma vez que o esforço de CPU foi gasto para satisfazer a prova de trabalho, o bloco não pode ser modificado sem refazer a prova de trabalho” (Nakamoto, S.; 2008).

Sendo assim, uma vez que os blocos estejam todos “acorrentados”, refazer um bloco inclui refazer todos os outros. O processamento necessário para refazer um bloco é equivalente a refazer este bloco e todos os outros anteriores.

Esta abordagem também resolve o problema de determinar qual é a decisão da maioria. Nakamoto aponta que em uma rede onde os votos são definidos por IP, alguém capaz de alocar um número significativo de *IPs* poderia manipular a votação. Com essa abordagem, que ele chama de “*one-CPU-one-vote*”, a decisão da maioria

[13] Entenda servidor como um grupo de nós da rede, não um servidor centralizado.

é representada pela cadeia de blocos mais longa, ou seja, que tem a maior quantidade de esforço de CPU alocada nela.

Relativo ao problema do aumento de poder computacional ao longo do tempo, a dificuldade da prova de trabalho é capaz de ser alterada posteriormente, baseado na média de blocos gerados por hora. Se a quantidade crescer muito, a dificuldade aumenta.

Em termos de rede, ela funciona da seguinte maneira:

- Novos transações são transmitidas para todos na rede;
- Cada nó da rede insere as transações no bloco;
- Cada nó trabalha na prova de trabalho do nó (minera o bloco);
- Quando alguém encontra a prova de trabalho, emite o bloco para todos os nós;
- Outros nós só aceitam o bloco se todos as transações forem válidas e não houver nenhum dinheiro já gasto; e
- Os nós expõem sua aceitação daquele bloco ao começar a trabalhar no próximo bloco na corrente, usando o *hash* do bloco aceito posteriormente.

O ato de minerar um bloco se refere ao trabalho computacional gasto para se encontrar um *hash* que esteja de acordo com a regra de aceitação da *blockchain* (por exemplo, no caso do *bitcoin* se trata de um número específico de zeros no início do *hash*).

O termo minerar é uma alusão ao ato de minerar metais preciosos, onde é necessário um esforço considerável por parte do minerador para encontrar algum minério precioso, e no caso das *cryptocurrencies*, esse esforço é traduzido em gasto de poder computacional.

O que mantém os nós minerando os blocos é a taxa que eles ganham sobre o bloco minerado, que é referente a diferença entre o valor da saída da transação e o valor da entrada.

Uma característica interessante é referente a otimização de espaço usada pelo *bitcoin*, a partir de um certo ponto, caso a última transação de uma moeda esteja sob uma quantidade suficiente de blocos, ela pode ser descartada. Para que esse processo de descarte não quebre a cadeia de blocos, o *hash* das transações anteriores são reunidos em um novo *hash* “raiz”, usando o conceito de árvore de

merkle, definida como “uma árvore binária completa com um valor de k bits associados a cada nó, onde o valor de cada nó interno é uma função unidirecional dos valores dos nós filhos” (Szydlo, 2004).

Com o conceito de otimização do espaço, a verificação de um pagamento também pode ser simplificada, onde ao invés de verificar toda a corrente de blocos para averiguar cada transação feita com aquela moeda, o nó pode obter o galho da árvore merkle que faz o link da transação com o bloco. Assim, ele é capaz de ver que a rede aceitou aquela transação e que, portanto, a transação é válida.

A privacidade dos usuários é mantida através da anonimidade das chaves públicas. Em outras palavras, as pessoas sabem que os usuários identificados por uma chave A e outra B então efetuando uma transação entre sí, mas ninguém sabe quem exatamente é o usuário proprietário da chave A e da chave B, as chaves não são explicitamente ligadas a ninguém no mundo real.

## 2.4. Algoritmos de consenso

Um algoritmo de consenso é o recurso utilizado para resolver o problema dos generais bizantinos, mencionado na seção 2.1.4, capaz de garantir que todos os generais do exército bizantino estão de acordo sobre qual plano seguir.

No cenário do *blockchain*, esses algoritmos são responsáveis por garantir que todos os nós da rede concordam sobre uma informação crucial: qual bloco deve ser adicionado na *blockchain*.

Após minerar um bloco, o minerador o submete para a rede que valida se as transações inseridas nele são válidas, e então cada nó da rede, após validar o bloco, o insere nos seus registros.

Para que todos os nós adicionem em seu registro o mesmo bloco, é necessário a existência de regras pré definidas entre todos eles, a qual chamamos de algoritmos de consenso.

Alguns dos exemplos de algoritmos de consenso mais comuns são o *Proof of Work* (Pow) e *Proof of Stake* (PoS), entretanto é importante saber que existe uma série de estudos sobre possíveis algoritmos de consenso, como o *Delegated Proof of Stake* (DPoS), *Proof of Burn* (PoB), além de outros menos conhecidos e que não serão mencionados neste trabalho.

Por serem algoritmos utilizados em moedas digitais já consolidadas e em circulação, o Proof-of-Work (bitcoin) e Proof-of-Stake (Ethereum 2.0) serão aprofundados nas seções a seguir, enquanto o Human Mining será aprofundado por ser o único algoritmo de consenso voltado para jogos encontrado durante as pesquisas.

### 2.4.1. Proof of work

Para garantir que todos os nós da rede vão adicionar o mesmo bloco em seus registros, o *bitcoin* por exemplo, utiliza o algoritmo de consenso conhecido como *proof of work*.

O próprio nome do algoritmo entrega seu princípio, se tratando de um algoritmo feito para provar que algum tipo e quantidade de esforço foi investido em alguma tarefa.

No caso do *bitcoin*, é utilizado para garantir que uma certa quantidade de trabalho tenha sido empregada na criação do bloco, garantindo que os blocos não sejam criados por qualquer um a qualquer momento, e assim garantindo a escassez da moeda.

Tendo em vista a imprevisibilidade do resultado de funções *hash*, o *bitcoin* utiliza essa função exigindo um número pré determinado de zeros, que determinam a dificuldade de se gerar um bloco, e a única forma de gerar um *hash* com esse número inicial de zeros é através da força bruta, utilizando tentativa e erro.

Quanto mais zeros iniciais são exigidos, mais difícil se torna encontra um *hash* apropriado e, portanto, mais trabalho foi empenhado na solução desse problema.

Um dos exemplos é o trabalho de Dwork e Naor (1992), que usam a solução de uma função matemática para garantir que algum trabalho computacional foi feito antes de se enviar um *e-mail*.

Nesse algoritmo, nós conhecidos como mineradores usam seu poder computacional para validar as transações e gerar um bloco, sendo recompensados com as taxas das transações que estão sendo processadas.

Uma das desvantagens desse algoritmo de consenso é o uso extensivo do hardware, acarretando em um alto consumo elétrico. Além disso, como mencionado por Naoki Shibata (2019) - “Bitcoin requer que os mineradores usem seus recursos computacionais para a PoW, que é basicamente o cálculo repetido de Hashes. Isto é um desperdício de recursos computacionais e eletricidade.”.

Trabalhos acadêmicos como o de Naoki Shiabata propõe soluções para que este desperdício de recurso computacional seja aplicado em algo que agregue valor, como por exemplo o *Proof-of-Search* (PoS) que “permite que o poder computacional desperdiçado no *Proof-of-Work* seja utilizado na busca por uma solução aproximada de uma instância de um problema de otimização” (Shibata, 2019).

### 2.4.2. Proof of stake

Um algoritmo que também serve para garantir que todos os nós da rede estão de acordo e sincronizados com as mesmas informações, porém, ao invés usar poder computacional para validar as transações e gerar os blocos, é usado a própria moeda.

Segundo a página no *Github* do Ethereum , os nós responsáveis por validar as transações são conhecidos como validadores, e estes são responsáveis por validar as transações, propor e votar qual bloco deve ser o próximo.

Para se tornar um validador, um nó que possua algumas moedas precisa efetuar uma transação específica para a rede, enviando uma quantia dessas moedas. Essas moedas são usadas como “garantia” de que o no atuará de forma honesta e caso contrário, perde as moedas depositadas.

Os votos tem peso variável em função da quantidade de moedas que foram depositadas pelo nó.

O que incentiva os nós a se tornarem validadores é o ganho da taxa das transações, que é repartido em função de diferentes implementações desse algoritmo, onde os dois mais conhecidos são o *chain-based proof of stake* e o *BFT-style proof of stake*.

### 2.4.3. Human Mining

Um outro algoritmo de consenso criado pelo grupo Xaya, opera de maneira mais lúdica. Segundo o próprio site da companhia, “blockchain são propagadas e prêmios são distribuídos com base no sucesso dos jogadores em uma competição online” (Xaya, 2018).

Segundo o blog do grupo Xaya, “neste sistema, desenvolvedores podem criar ambientes com recursos distribuídos finitos e justos. Em outras palavras, itens virtuais podem ser implementados de maneira que sejam escassos e ainda sim disponíveis para quem os desejar” (Xaya, 2018).

Sendo assim, ações lúdicas nos jogos como encontrar um item raro, enfrentar um inimigo poderoso ou lutar contra outros jogadores podem compensar o jogador com moedas, itens e recursos que tem sua escassez (e consequentemente seu valor) expostos na *blockchain*.

As ações empregadas pelo jogador para adquirir algum recurso são a prova do esforço necessário para se adquirir tal item e que, portanto, este jogador tem posse sobre este item. Como o próprio grupo menciona, “jogadores passam a ser pagos para jogar” (Xaya, 2017).

“Este valor é relativo a dificuldade requerida para se adquirir esta coleção, assim como a quantidade e nível de esforço desempenhado por outros jogadores competindo para adquirir o mesmo recurso” (Xaya, 2018).

Outro ponto importante é que, pelo fato desses itens serem persistidos em uma blockchain que pode abrigar diversos jogos, os ativos digitais adquiridos pelos jogadores podem ser migrados entre diferentes jogos.

# 3. GAME ENGINES

Como mencionado na seção 1.3, o foco deste trabalho é aplicar os conceitos de *blockchain* em uma aplicação gamificada. Esta seção é responsável por descrever todos os aspectos relevantes referentes a jogos digitais e *game engines*.

Segundo DICIO, jogo é um exercício ou divertimento sujeito a certas regras, enquanto MICHAELIS define como atividade onde diferentes indivíduos ou grupos se submetem a competições com um conjunto de regras que determina quem ganha ou perde.

Para este trabalho, jogos eletrônicos são o foco onde, segundo a Britannica Escola, é todo jogo que utiliza tecnologia eletrônica, como o computador, consoles, fliperamas, smartphones e outros dispositivos eletrônicos, onde uma interface de input e uma de output são utilizados para jogar, sendo uma forma de entretenimento e, nos últimos anos, tendo sido explorada como uma forma de educar.

Produzir jogos eletrônicos, assim como produzir *software*, não é uma tarefa trivial. Da mesma forma que *softwares* convencionais, jogos eletrônicos exigem que os jogadores - análogo aos usuários dos *softwares* - forneçam *inputs*, que são processados pelo jogo, seguindo as regras impostas por ele, e então gere um ou mais *outputs*, muitas das vezes visual e sonoro.

Como apontado pelo Game Studio Report 2018, da Unity Technologies, cerca de 64% dos membros dos estúdios da pesquisa estão envolvidos com a área de produção, sendo eles divididos em programadores, artistas e designers, e os demais

36% integrantes da parte de suporte, como administradores, produtores, engenheiros de áudio e outros.

Tendo em vista o foco técnico deste trabalho, as atividades de produção serão aprofundadas nos parágrafos a seguir, segundo Shylenok (2019), para fins de contextualização.

Inicialmente, um profissional da parte de Design, mais especificamente um *game designer* elabora todas as regras do jogo, assim como outros aspectos importantes, como personagens, ambientes, itens, equipamentos, enfim, tudo pertinente aquele jogo. Todas essas informações são reunidas em diversos documentos, sendo um dos principais o *game design document*, conhecido também pela sigla GDD, responsável por compilar diversas informações técnicas sobre o jogo, a fim de guiar os desenvolvedores no processo de criação do game.

Paralelamente, temos os artistas cuidando de toda a parte visual do jogo, elaborando as imagens que representam os personagens, inimigos, itens, cenários, equipamentos, *HUD*[14] sendo esses e outros elementos conhecidos como *assets*, que nada mais são que os recursos utilizados pelo jogo.

Por fim, o programador é responsável por gerar todo o código fonte do jogo, aplicando as regras definidas pelo game designer no meio eletrônico. É pertinente mencionar a existência de recursos que são comuns entre diversos jogos, como simulação de física, emissão de partículas, renderização, testes de colisão, gerenciamento de *assets*, recebimento de *inputs*, sistema de eventos entre outros que são implementados pelo programador.

Por serem recursos comuns no desenvolvimento de jogos, grupos e empresas desenvolveram e desenvolvem o que hoje é chamado de *game engines*, um ambiente de desenvolvimento jogos, que possui ferramentas e bibliotecas com recursos prontos, a fim de reduzir a complexidade do desenvolvimento e tornar mais fácil e prático todas as tarefas relativas à produção de jogos, permitindo assim que o programador hoje, se preocupe apenas em desenvolver código relativo às regras do jogo.

## 3.1. Comparativo entre game engines

[14] HUD ou *heads-up-display*, é o grupo de elementos da interface visual do player que transmitem informações relevantes para a jogada, como pontos de vida (HP), munição de armas, mini mapa entre outros.

Decidir qual ferramenta será utilizada para alguma tarefa é essencial quando se deseja alcançar o sucesso. E o desenvolvimento de jogos não difere disso. Para se escolher uma game *engine* a ser utilizada por um projeto, é necessário se conhecer bem as necessidades e requisitos deste projeto.

Isso porque não existe game *engine* definitiva, todas apresentam pontos positivos e negativos que precisam ser levados em consideração quando se trabalha com um projeto.

Para este trabalho, iremos analisar três game *engines* muito utilizadas segundo Toftedahl M. e Engström H. (2019), que consultaram os jogos disponíveis na Steam e Itch.io, duas grandes lojas de jogos, e buscaram quais *engines* foram usadas na produção desses jogos.

Segundo seus infográficos, Unity e Construct 2 são os mais utilizados pelos jogos da Itch.io, enquanto Unreal e Unity são as mais utilizadas nos jogos disponíveis na steam.

Em termos de comparação, Hill-Whittall escreveu o livro *The Indie Game Developer Handbook* (2015) onde ele compara diversas game *engines*, incluindo as três citadas acima, com seus prós e contras apresentados abaixo:

### 3.1.1. Unreal 4

Como apontado por Hill-Whitall, “visualmente é difícil superar a Unreal Engine”. Jogos como Injustice 2, Days Gone e Fortnite são estruturados sobre a *engine*, como mostrado no seu site oficial.

Hill-Whitall aponta a grande variedade de sistemas suportados para lançamento e um bom suporte da comunidade, com uma quantidade razoável de tutoriais pela rede.

Ainda sim, pontos negativos levantados incluem uma taxa relativamente cara se o seu jogo vender bem, fora o relato de alguns usuários sobre a curva de aprendizado difícil, mostrando que certamente a *Unreal* é um dos kits de desenvolvimento mais complexos atualmente.

Válido mencionar que para todo produto lançado, 5% do valor arrecadado após os primeiros $3.000 por trimestre é destinado a Epic Games, desenvolvedora da *Engine*.

Por fim, uma loja de *assets* relativamente pequena em relação a lojas como a da Unity.

### 3.1.2. Unity 3D

A versão gratuita da *engine* possui todos os recursos disponíveis na *engine*, com a versão paga adicionando alguns recursos de suporte, como um número maior de colaboradores na plataforma da Unity e atendimento prioritário no suporte da *engine*.

Jogos como Kerbal Space Program, Deus Ex: The Fall e Call Of Duty: Strike Team foram feitos utilizando a *engine*.

Além disso, a *engine* é capaz de trabalhar tanto com jogos 3D quanto 2D, permitindo que jogos com estilo simples até jogos de altíssima qualidade possam ser desenvolvidos.

Possui uma comunidade enorme e ativa, garantindo que material de qualidade e diversificado esteja disponível, além de uma loja de *assets* com um volume considerável de produtos.

Entretanto, é necessário que o usuário saiba programar para tirar proveito de todos os recursos da *engine*, e a versão paga não é barata.

A partir de $100.000 anuais (algo em torno de $25.000 por trimestre) a Unity Technologies exige que uma licença seja adquirida, como informado no site oficial da *engine*, com a licença Plus, a mais barata atualmente, por cerca de USD $40 mensais a partir de 2020.

### 3.1.3. Construct 2

Segundo Hill-Whitall, é uma *engine* eficiente para jogos 2D simples, incluindo uma série de funcionalidades prontas que agilizam o desenvolvimento, como um controle de jogo em plataforma pronto, plataformas móveis, pular através de plataformas e outros recursos.

Ideal para aqueles que não possuem conhecimentos em programação e para prototipagem rápida de projetos.

Quanto os pontos negativos, se trata de uma solução não muito efetiva em jogos complexos e o sistema pelo qual se programa utilizando recursos visuais é limitado e por vezes apresenta funcionamento inesperado.

Além destes pontos, a versão gratuita não permite a criação de projetos comerciais e monetizados, sendo necessário a compra da *engine*, cujo valor mais baixo é a da licença pessoal, por R$700.

### 3.1.4. Tabelas Comparativas

As tabelas a seguir foram montadas analisando dados do livro The indie Developer HandBook, Gamasutra e dos sites oficias das *engines*, a fim de fornecer uma visão mais simples dos recursos e diferenças das *engines* apresentadas neste trabalho:

**Quadro 1 -** Comparação dos recursos das game engines

| Características | Unreal | Unity | Construct 2 |
|---|---|---|---|
| Versão gratuíta | Sim | Com todos os recursos da engine. | Sim |
| Taxas | A partir de $3.000 trimestrais | A partir de $100.000 anuais | Necessário a compra para se vender um jogo feito. |
| Suporte a jogos 2D | Sim | Sim | Sim |
| Suporte a jogos 3D | Sim | Sim | Não |
| Necessário conhecimento em alguma linguagem de programação | Não | Sim - C# | Não |
| Loja de assets | Sim | Sim | Sim |
| Outros | Alta qualidade visual por padrão. | Uma comunidade grande e ativa, com diversos tutoriais pela rede. | Recursos prontos que facilitam a prototipagem. |

Fonte: The indie Developer HandBook, Gamasutra

**Quadro 2 -** Plataformas de lançamento suportadas

| Plataforma | Unreal | Unity | Construct 2 |
|---|---|---|---|
| Windows | Sim | Sim | Sim |
| Mac | Sim | Sim | Sim |
| Linux | Não | Sim | Sim |
| iOS | Sim | Sim | Sim |
| Android | Sim | Sim | Sim |
| Windows Phone | Não | Sim | Não |
| Blackberry | Não | Sim | Sim |
| Web | Não | Sim | Sim |
| Playstation | Playstation 4 | Através de pagamento. | Não |
| Xbox | Xbox One | Através de pagamento. | Não |
| Wii U | Não | Através de pagamento. | Não |

Fonte: The indie Developer HandBook, Gamasutra

Levando em consideração que apenas a Unreal e a Unity possuem suporte para jogos em 2D e 3D, o Construct 2 deixa de ser levado em consideração.

Outro ponto a ser avaliado é que as *engines* apresentam alguns recursos não disponíveis na versão grátis, excetuando a Unity, que libera todas as ferramentas da *engine* tanto na versão grátis quanto na versão paga.

A comunidade de usuários ativa e grande permite que um projeto como este atinja uma quantidade maior de desenvolvedores de jogos, ajudando na propagação e evolução do projeto.

Portanto, Unity é a *engine* escolhida para o desenvolvimento deste trabalho.

## 3.2. Unity Engine e sua História

Uma das *engines* mais famosas da atualidade surgiu em 2002, em um fórum de *OpenGL* para Mac, onde Nicholas Francis pediu ajuda com o sistema de *shaders*[15] que estava desenvolvendo para a sua *game engine.* Joachim Ante também estava desenvolvendo um *software* semelhante, entrou em contato e juntos começaram a desenvolver um sistema de *shaders* que sería utilizado por ambos - como descrito por Haas, J. em seu trabalho “A History of the Unity Game Engine” (HAAS, 2014).

Posteriormente, decidiram que seria mais interessante desenvolverem juntos uma única *engine*, e sabendo dessa informação, David Helgason se juntou a equipe de desenvolvedores.

Sem saber por onde começar, depois de dois anos o fundadores decidiram que sua *engine* seria a ferramenta definitiva para 3D na web, solicitaram alguns empréstimos e reuniram alguns engenheiros para abrir um escritório. Posteriormente passaram a procurar por um CEO, sem muito sucesso, acabaram por colocar o próprio Helgason no posto, por ser o mais sociável dos três.

Assim, formaram a OTEE (*Over the Edge Entertainment*) e começaram a desenvolver um plano de negócios, se espelhando na empresa britânica de

[15] “Um shader é um programa de computador que executa cálculos de gráficos - por exemplo, transformações de vértice ou cor do pixel — e normalmente é executado em uma unidade de processamento gráfico (GPU) em vez da CPU.” (Warren, 2016).

desenvolvimento Criterion, que estava se tornando muito conhecida no mercado de middlewares na época do playstation 2.

Eles perceberam que os jogos casuais e online iriam crescer, e decidiram focar nesse mercado, além de desenvolverem um jogo comercial completo usando a *engine*, a fim de testar os limites dela e conseguir alguma renda para manter o desenvolvimento da ferramenta.

*Gooball* foi o primeiro jogo comercial feito na Unity, lançado pela *Ambrosia Software*. O time usou esse projeto como oportunidade para encontrar bugs e outros problemas antes de lançar oficialmente a *engine*.

Com o lançamento de Gooball, a OTEE conseguiu juntar mais desenvolvedores para refinar a Unity para seu lançamento inicial em Junho de 2005. Com esse apoio, foram capazes de eliminar falhas restantes, documentar de maneira extensiva a ferramenta e oferecer suporte aos usuários.

Após o lançamento, eles começaram a trabalhar na próxima iteração da *engine*:

> "Nós somos desenvolvedores desenvolvendo para outros desenvolvedores, nós entendemos os problemas deles e tentamos conserta-los; nos falamos com eles, seria o caso de não estar conseguindo dormir à noite pensando, `o que ser[a que está acontecendo nos fóruns. Isso era constante ao longo do dia. Eu penso que eles reconheceram isso, e então esses clientes que tínhamos começaram a evangelizar para a gente" (HAAS apud Brodkin, 2014)

Muitas companhias de desenvolvimento faziam jogos que não vendiam muito bem e consequentemente jogavam fora sua tecnologia. Clientes potenciais temiam que isso pudesse acontecer com a Unity, e assim levaram alguns anos até provarem que a Unity era capaz de ser atualizada e ter um suporte adequado.

A primeira versão publicada da Unity era capaz de gerar *builds* apenas para o Mac OS X, até que na versão 1.1 passou a ser capaz de exportar para o Windows e para navegadores web. Tendo em vista que o cenário de jogos casuais web estava crescendo e existiam algumas poucas opções no mercado, como o adobe flash player, a Unity estava em um cenário favorável.

Outra adição importante e com efeitos até os dias atuais foi o suporte a plugins externos em C e C++, permitindo os desenvolvedores a estender a *engine*.

A versão 2.0 trouxe mais estabilidade para os ambientes que a *engine* suportava, além de uma performance melhor, uma vez que o DirectX que já existia nas máquinas com windows, passou a ser utilizado para projetos compilados para o sistema operacional da Microsoft, ao invés de usar o OpenGL que precisava ser baixado separadamente.

Outros recursos como *streaming web*, sombras suaves em tempo real, conexão com a internet, uma *engine* de terreno, o Unity Asset Server e um sistema de interface codificada (GUI) também foram adicionados.

Com o advento dos *smartphones* e mais precisamente do Apple Iphone e da Apple Store, a agora conhecida como *Unity Technologies* decidiu desenvolver uma versão da Unity capaz de publicar para o Iphone, mas separada do núcleo principal da Unity, um software separado.

Uma reestruturação completa do software se fez necessária quando a Unity Technologies se deu conta que seus clientes estavam adquirindo Macintosh apenas para usar o *software*, uma vez que a *engine* existia apenas para os sistemas Apple, apesar de ser capaz de gerar builds de jogos para outros sistemas operacionais, como o windows. Sendo assim, eles desenvolveram o núcleo de forma independente de sistema operacional, e criaram editores diferentes para cada plataforma.

A versão 3.0 trouxe uma série de melhorias, tais como um *lightmapping* melhor, *deferred rendering*, Umbra Occlusion culling, debug de baixo nível, efeitos de lentes entre outras melhorias menores. “com a Unity 3, nós estamos demonstrando que nós podemos nos mover mais rápido que companhias de *middleware*, que nós somos sérios em relação a longo prazo e que alta tecnologia feita simples é uma força transformadora” disse Helgason em entrevista. Nessa versão a equipe da Unity também pegou todos as plataformas de publicação que precisavam de editores diferentes e a uniram em um único editor.

Na época do lançamento da versão 3.0, a Unity somava mais que 200.000 usuários registrados, tornando-a a *engine* número 1 para propósitos educacionais e a mais usada em plataformas mobile.

A versão 3.5 adicionou suporte a *deploy* para o Flash, um recurso esperado a muito tempo, além do sistema de partículas “Shuriken” e de um sistema de *pathfinding* nativo.

A versão 4 veio em Junho de 2012 e trouxe o sistema de animação conhecido como *mecanim*, responsável por permitir que personagens fossem animados na *engine* de uma forma mais visual e simples. Sombras em tempo real em todas as plataformas, melhoras no sistema de partículas *shuriken* e *deploy* para linux e adobe flash foram outros recursos adicionados.

A versão 4.3 trouxe suporte nativo a jogos 2D, um recurso também muito aguardado. Antes desse update, projetos 2D eram feitos usando um projeto 3D com objetos planos posicionados em 2 eixos apenas, passando a falsa impressão de 2D. Com esta atualização, uma série de recursos nativos, como o componente “sprite”, foram adicionados.

A versão 4.6 veio com novas ferramentas de UI, incluindo uma série de componentes que poderiam ser facilmente inseridos nas cenas, tornando obsoleto o sistema de GUI que exigia muito mais código.

A versão 5 da Unity foi a maior até então, trazendo muitas melhorias no aspecto visual da *engine*, principalmente na parte de iluminação, adicionando uma janela dedicada para esse recurso. Iluminação global em tempo real e Reflexão HDR foram alguns dos recursos que vieram com essa nova versão.

Após a versão 5, a Unity passou a usar o ano ao invés de números sequenciais, iniciando pela versão 2017.1.

## 3.3. Unity Editor

O editor da Unity é a interface visual pela qual todas as interações entre desenvolvedores e *engine* são feitas, podendo ser estendida pelos próprios desenvolvedores baseado nas suas necessidades.

Existem alguns conceitos importantes que precisam ser comentados antes de aprofundar o editor.

Um desses conceitos é o de cenas, como explicado pelo próprio site da Unity “pense em cada cena individual como um nível único”. Cada cena é como uma tela do jogo, sendo possível ter diversas cenas, com uma para o menu principal, uma para os créditos finais e uma com o jogo em si por exemplo.

Outro conceito importante é o de *assets*. Estes nada mais são que qualquer recurso usado no jogo, desde imagens até modelos 3D e arquivos de áudio, todos eles são considerados *assets*. Repare que não necessariamente só arquivos

gerados fora da *engine* são *assets*. *Prefabs* também são um tipo de *asset* e são gerados dentro da *engine*.

*Prefabs* podem ser entendidos como “pré fabricados”. Todo objeto (também conhecido como *GameObject*) que é criado dentro da Unity nada mais é que um conjunto de componentes agrupados para dar aquele objeto um comportamento esperado.

Montar esses objetos exige trabalho manual e, por vezes, é necessário criar uma quantidade considerável deles, como por exemplo inimigos. Um inimigo é formado por uma série de componentes agrupados, como um *sprite renderer*[16], um *rigidbody2D*[17] e um *script*. Ao criar um *prefab* deste inimigo, é possível copiá-lo e ter várias cópias deste inimigos.

Alguns nomes citados acima, como *rigidbody2D* e *sprite renderer* são alguns dos componentes que a *engine* possui por padrão, e exercem alguma função específica.

Os componentes refletem um *design pattern* conhecido como *component pattern*, conhecido no desenvolvimento de software. Esse padrão de desenvolvimento diz que cada script existe para desempenhar uma função específica, agregando um comportamento específico a um objeto. Caso seja necessário gerar comportamentos complexos, estes seriam gerados a partir da adição de vários componentes em um objeto.

[16] Componente responsável por renderizar (desenhar) um objeto na tela. Ele recebe como um dos inputs a imagem que representa aquele objeto.

[17] Componente responsável por adicionar propriedades físicas aos objetos, permitindo que eles passem a ser influenciados pela física da engine.

# 4. MODELAGEM E IMPLEMENTAÇÃO

## 4.1. Introdução e regras do jogo

Para apresentar o conceito de jogo funcionando com uma blockchain, uma aplicação gamificada foi desenvolvida e integrada com uma blockchain também desenvolvida para este trabalho.

É válido mencionar que, pelo caráter experimental deste trabalho, toda a implementação é simplória e exige melhorias extensivas antes do projeto ser disponibilizado, sendo estas melhorias descritas na seção 4.9 – Análise Crítica.

O jogo escolhido para ser implementado é o dominó, tendo em vista que, por ser jogado em turnos, sua implementação é mais simples, e suas regras sendo aprofundadas nos parágrafos a seguir.

Um dos jogos que faz parte da história humana é o dominó, possivelmente criado entre 243 A.C e 182 A.C na China. Entretanto, a primeira versão do dominó difere da versão conhecida hoje em diversos fatores.

Segundo o museu de jogos Elliot Avedon, citado por Mara Ludmila (2009), podem ser vistas evidências indicando que o dominó surgiu do uso de um par de dados, sendo as peças originárias da combinação dos dois pares de dados. Este modelo possui 32 peças sendo 21 diferentes e 11 repetidas.

A versão européia conta com um número menor de peças e não usa os dados onde, segundo o mesmo museu, foi inserido na europa no século XIX inicialmente na Alemanha, partindo para França e Itália posteriormente.

A versão ocidental é constituída de 28 peças, com cada peça sendo dividida em duas partes, cada uma com um valor de 0 a 6. Nenhuma combinação de números se repete entre as 28 peças. (Mara, 2009)

Vale mencionar a existência de seis peças chamadas ‘doble’, que são aquelas cujo valor é o mesmo nas duas partes da peça, e devem ser postas de forma perpendicular às outras peças sempre que postas na mesa.

A versão brasileira pode ter 4 jogadores jogando em duplas, onde cada um recebe 5 peças, ou 2 jogadores, onde cada jogador recebe 7 peças.

### 4.1.1. Regras de negócio

A fim de priorizar a implementação e a experiência digital e online do clássico dominó, algumas regras foram modificadas a fim de beneficiar essa experiência.

Protagonizado por dois jogadores que não se conhecem e são conectados por uma rede *peer-to-peer*, as seguintes regras são aplicadas:

- Das 28 peças existentes, cada jogador receberá 7 sorteadas de forma aleatória, restando 14 peças;
    - Cada peça possui duas partes, preenchidas com números de 0 a 6;
    - Cada combinação de peças é única; e
    - Das 14 peças restantes:
        - Uma será utilizada para iniciar o jogo; e
        - As demais 13 estarão disponíveis para compra.
- O jogador que tiver feito o pedido de conexão será responsável por:
    - Gerar todas as 28 peças;
    - Sortear quais peças ficam com ele e com o adversário;
    - Sortear a peça inicial;
    - Embaralhar a pilha de peças a serem compradas;
    - Enviar todo o conjunto de peças sorteadas para o adversário; e
    - Efetuar a primeira jogada.
- Cada jogador tenta encaixar suas peças em uma das duas extremidades existentes.
    - Para que uma jogada seja válida, um dos dois valores da peça precisa ser igual a um dos dois valores disponíveis nas extremidades.

        - Caso a jogada seja válida, a peça é movida para a extremidade com o mesmo valor da peça; e
        - O outro valor da peça se torna o novo valor da extremidade.
    - Caso o jogador não tenha nenhuma peça que se encaixe com alguma das extremidades, terá de comprar uma peça das 13 na pilha de compra:
        - Caso a peça se encaixe com alguma extremidade, poderá jogá-la; e
        - Caso não, terá de comprar outras peças até conseguir uma que se encaixe.
- O jogo acaba quando um dos jogadores pressionar o botão "*EndGame*".

#### 4.1.2. Requisitos funcionais

- Efetuar *login*;
    - Armazenar apelido do jogador;
    - Armazenar os dados de rede do nó;
- Consultar jogadas;
- Consultar bloco;
- Consultar transação;
- Desafiar outro jogador;
    - Conectar com outro jogador;
    - Obter informações de outro jogador;
    - Armazenar dados de conexão do outro jogador;
- Iniciar Jogo;
    - Sortear peça inicial;
    - Sortear peças dos jogadores;
    - Embaralhar peças para comprar;
- Enviar jogada;
    - Enviar peça comprada;
    - Enviar peça jogada;
    - Criar transação;
    - Determinar quem deve jogar a cada momento;
- Encerrar o jogo;
    - Adicionar as jogadas ao bloco atual; e

- Minerar o bloco com transações do jogo;

### 4.1.3. Requisitos não funcionais

- O jogo deve ser escrito em C#;
- O jogo deve funcionar em rede (seja ela local ou não);
  - Os usuários devem ser reconhecidos através de um apelido;
  - Os usuários precisam determinar o seu apelido antes de iniciar um jogo;
- O jogo deve guardar os dados de login em memória volátil (RAM) enquanto o jogo estiver sendo executado;
- Ao efetuar login, a aplicação deve abrir um socket[18] para comunicação em rede;
- O jogo deve usar a UI da Unity para exibir dados relevantes:
  - Exibição dos dados da *blockchain* deve ser através da UI;
  - Exibição das peças do jogo deve ser através da UI;
  - Exibição do formulário de login deve ser através da UI; e
  - Exibição do formulário de conexão deve ser através da UI.
- O jogo deve expor algum recurso de rede para estabelecer a conexão entre os nós;
- O jogo deve gerenciar o ato de criação e distribuição das peças entre os jogadores;
- O jogo deve gerenciar qual jogador joga a cada momento;
- O ato de sortear as peças deve usar as funções de randomização da *Unity;*

[18] “Um *socket* é um *endpoint* de um link de comunicação de duas vias entre programas rodando na rede” (Oracle)

- O jogo deve serializar[19] os dados de rede no formato *Json;*
- O jogo deve utilizar a *blockchain* para garantir que os blocos com as jogadas estejam seguros:
    - O jogo deve garantir que qualquer usuário deva ser capaz de visualizar qualquer transação na *blockchain; e*
    - O jogo deve permitir que qualquer jogador possa adicionar jogadas ao bloco atual.

## 4.2. Diagramas

### 4.2.1. Diagrama de caso de uso

[19] “Serialização de objeto transforma os dados de um objeto em um byteStream que representa o estado dos dados. A forma serializada dos dados contém informação suficiente para recriar um objeto com seus dados em um estado similar ao que foi salvo. ” (Oracle)

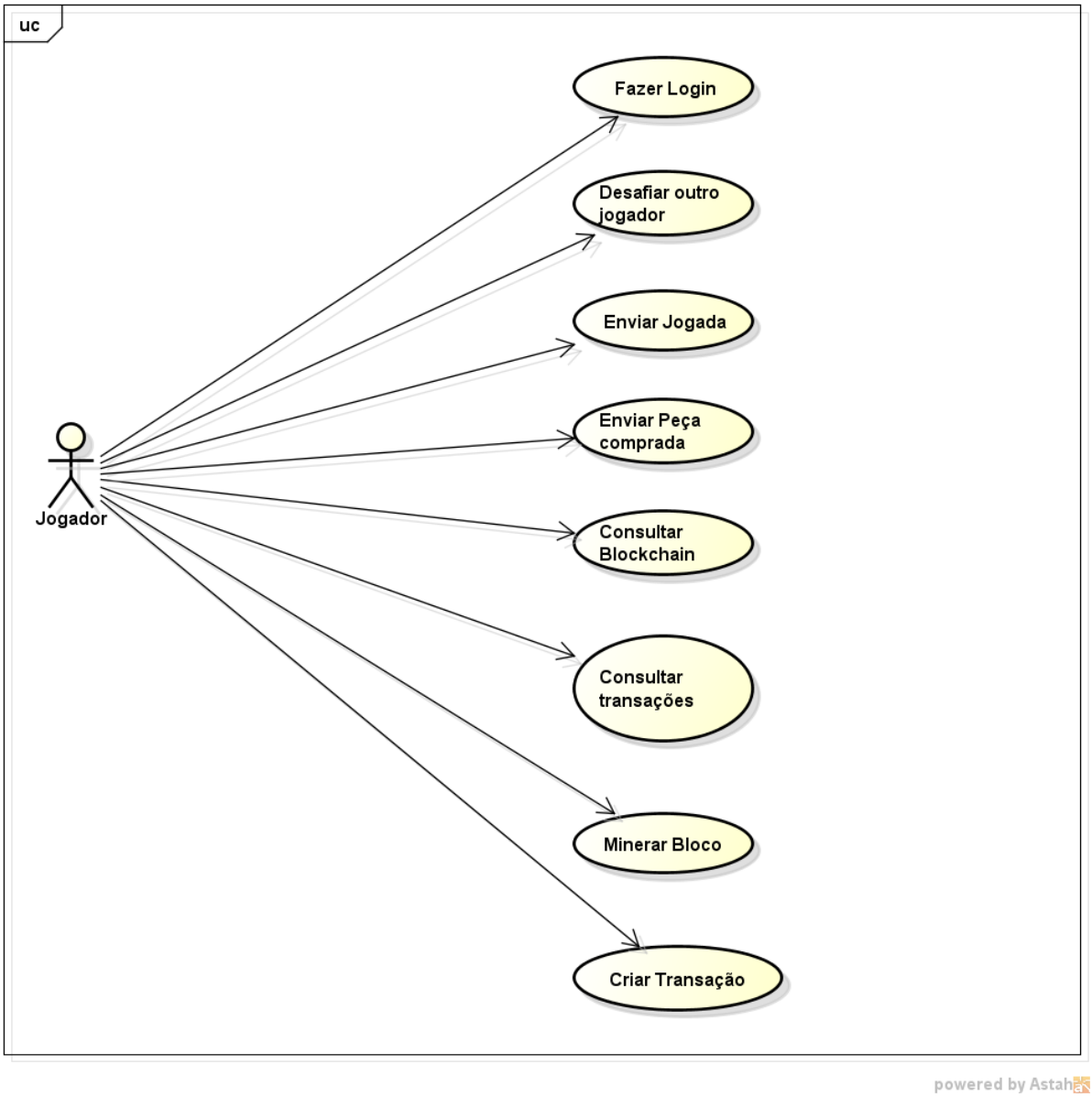


Figura 3: Diagrama de caso de uso

O diagrama de caso de uso acima exibe todas as ações que podem ser executadas pelo jogador, incluindo ações dependentes umas das outras.

O Jogador pode fazer login, passando o número da porta e o apelido pelo qual será chamado.

Pode desafiar outro jogador ao informar a porta na qual este jogador está conectado.

O ato de clicar em uma peça durante o jogo representa o envio de uma jogada, que pode ser a compra de uma peça, o ato de desistência ou a peça escolhida pelo jogador.

O ato de consultar a *blockchain* só é possível antes do jogo, no espaço chamado *Lobby*, onde o player também pode efetuar o pedido de conexão com outro jogador.

Durante o jogo, é possível consultar apenas as transações do jogo em andamento.

O ato de minerar o bloco também é solicitado pelo jogador, e ocorre quando ele seleciona o botão “*End Game*” durante o jogo.

### 4.2.2. Diagrama de classes

Por contar com um número considerável de classes para o pleno funcionamento da solução, o diagrama de classes foi dividido em algumas partes. É possível que alguma classe apareça em mais de um diagrama, tendo em vista as múltiplas relações que essas classes podem ter.

É importante mencionar que a solução foi feita sobre o conceito de *ViewModel*, portanto, existem classes de modelo, responsáveis apenas por armazenar propriedades daquele modelo em si, como a peça de dominó, e classes de *ViewModel*, responsáveis por armazenar uma referência à classe modelo e todas as propriedades de exibição deste modelo na UI da Unity.

Esta abordagem é melhor explicada na seção 5.4 - Análise Crítica, que explicita o porquê esta abordagem foi adotada.

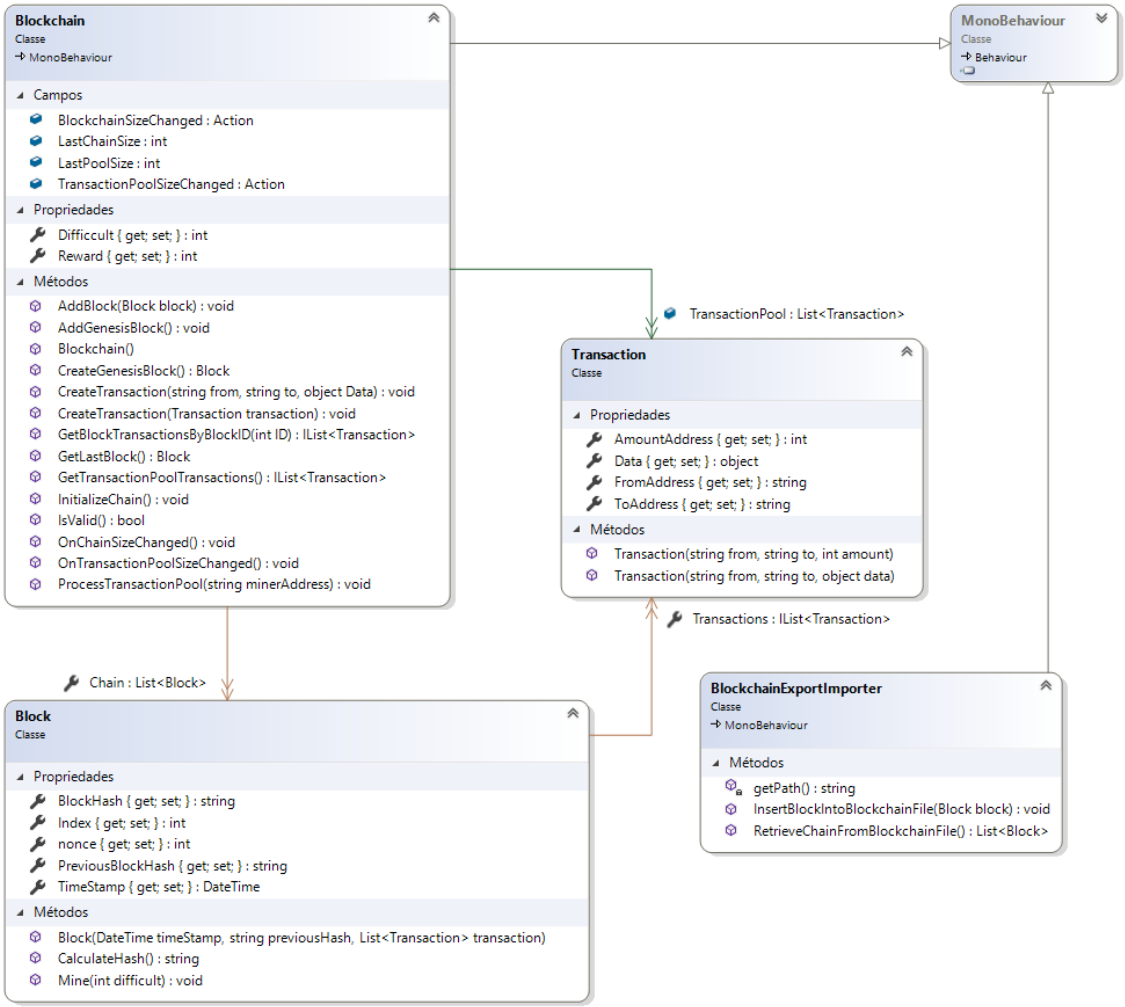


Figura 4: Diagrama de classes

*Monobehaviour* é a classe da qual todo componente da Unity precisa derivar, caso tenha a intenção de ser exibido e manipulado pela *engine*. É válido mencionar que, mesmo que algumas classes como a própria *blockchain* não tenham a necessidade de ser exibida no editor da *engine*, esta classe ser derivada de Monobehaviour facilita a comunicação dela com outros componentes da *engine*.

A classe *Blockchain* armazena todas as propriedades e métodos referentes a *blockchain*, incluindo eventos chamados quando um novo bloco ou uma nova transação são adicionados, métodos para criação do bloco Gênesis, criação de novos blocos e métodos para resgatar blocos específicos da corrente para visualização.

Nela pode-se notar a existência de uma lista da classe *Block*, representando o conceito fundamental da *blockchain*, que é uma série de blocos sequenciais conectados por um *Hash*.

Cada bloco possui o seu *Hash*, o *hash* do bloco anterior, o momento em que foi criado o seu *nonce* e a sua dificuldade, que representa o número de zeros iniciais necessários para que o *Hash* encontrado seja aceito. Para fins de praticidade, a implementação em questão utiliza uma dificuldade de apenas dois dígitos, permitindo que os blocos sejam criados de maneira rápida e fácil.

Além disso, existe também uma lista de transações, que representa o conceito de *pool* de transações, armazenando as transações atuais que estão para ser adicionadas no bloco atual.

Cada transação possui uma quantidade de valor monetário a ser enviada, dados, que nesta implementação representa os valores da peça de dominó jogada, um nó de origem (*player*) e um nó de destino (adversário), além de seus respectivos construtores.

BlockchainExportImporter é a classe responsável por resgatar e inserir os dados da *blockchain* em um arquivo externo gravado no disco rígido, a fim de permitir que a *blockhain* continue existindo mesmo após o encerramento da aplicação.

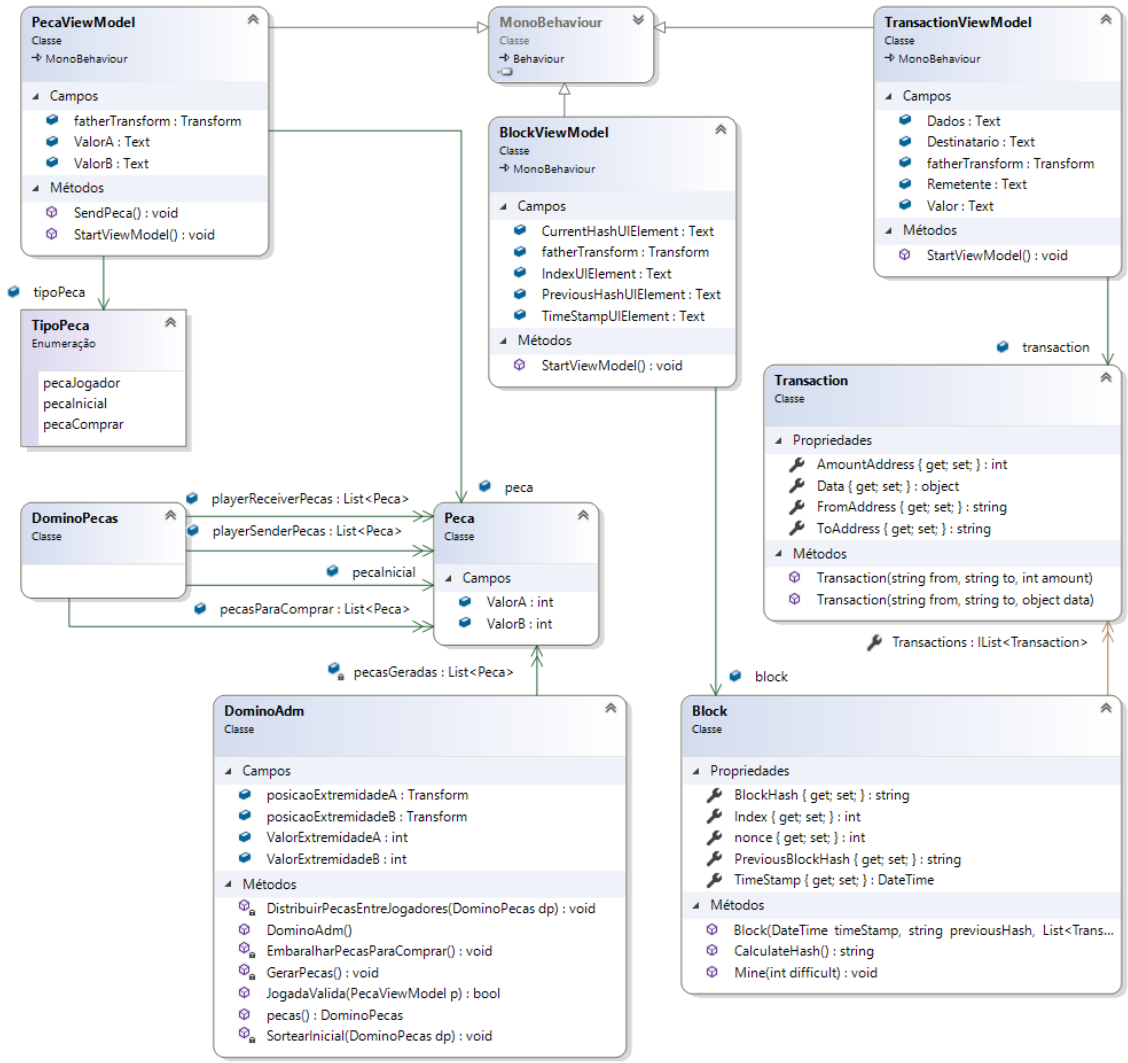


Figura 5: Diagrama de classes

Este diagrama de classes é a primeira parte das classes de *Gamelogic*, referentes a implementação do jogo de dominó.

A classe *BlockViewModel* e *TransactionViewModel* são responsáveis por armazenar todas as propriedades de exibição (texto, painel e posição) de um bloco e de uma transação, representados pelas classes *Block* e *Transaction*, respectivamente.

A classe de modelo Peca representa a peça do dominó, armazenando apenas seus dois valores (A e B). É importante mencionar que por ser tratar de uma classe de modelo, criada para armazenar as propriedades da peça especificamente, ela não deriva de *monobehaviour*.

Para ser exibida pela Unity, a classe *PecaViewModel* é responsável por armazenar uma referência à classe peca com seus valores, e todas as propriedades referentes a Unity e a exibição da peça.

Também possui um enumerador que representa qual tipo de peça ela é, sendo peça de jogador, peça para comprar e peça inicial as possibilidades, mostrando assim onde a peça deve ser colocada durante o jogo.

DominoPecas representa o baralho do jogo, sorteado pelo jogador que fez o pedido de conexão. Armazena uma lista de peca para as peças do Player, do adversário, da pilha de compra e uma única peça que é a inicial. Esta classe é serializada e enviada pela rede para o adversário.

Por fim, a classe *DominoAdm* armazena toda a lógica de validação e validação do jogo, responsável por armazenar qual o valor das extremidades do jogo e validar se uma jogada é válida ou não, além de embaralhar e distribuir as peças do dominó.

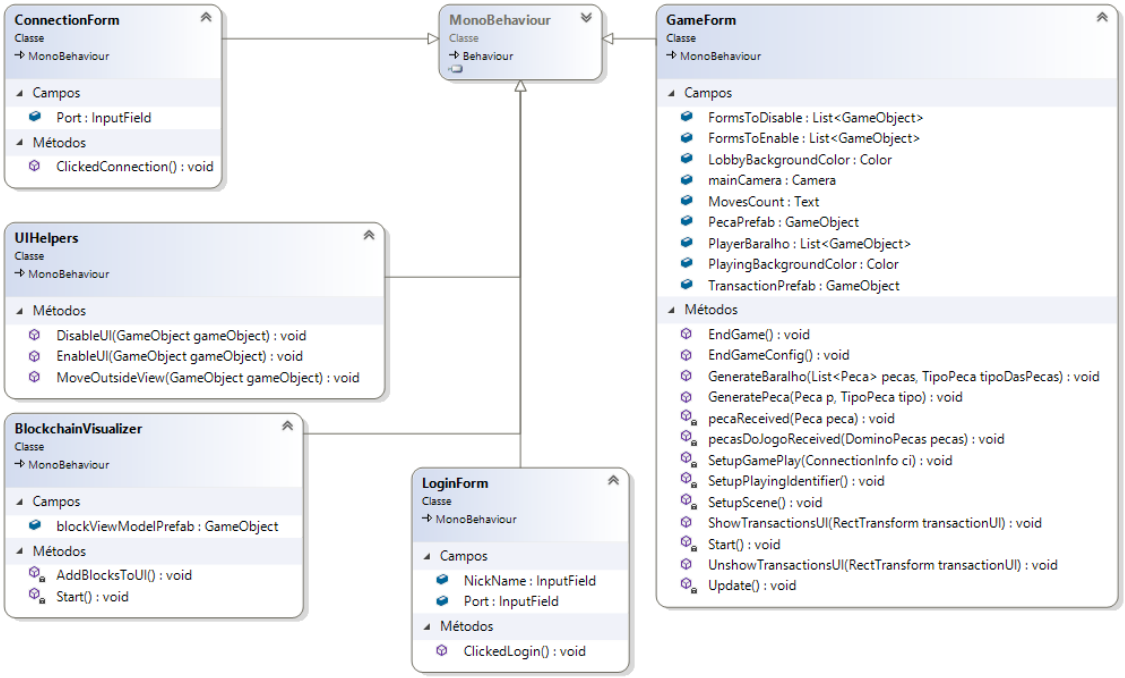


Figura 6: Diagrama de classes

Este é a o diagrama com a segunda parte das classes da *GameLogic*.

O *LoginForm* é responsável por receber o input do usuário no ato de conexão e criar o socket de rede com essas informações.

A classe *ConnectionForm* é responsável por receber o valor da porta do adversário na UI da Unity e transmitir ela para o módulo de Network, responsável por estabelecer a conexão entre o jogador e o adversário.

*UIHelpers* é uma classe com métodos para ativar e desativar objetos visuais da UI, facilitando essas ações.

*BlockchainVisualizer* é uma classe que administra o painel de exibição dos blocos da *Blockchain* na UI, a fim de atualizar a visualização dos mesmos.

Por fim, o *GameForm* controla qual formulário deve ser exibido a cada momento, além de armazenar os métodos chamados por cada botão de cada formulário.

O método que estabelece a conexão entre os jogadores, o que cria o socket no ato de login e o que permite a criação do bloco ao pressionar o botão “EndGame” se encontram aqui.

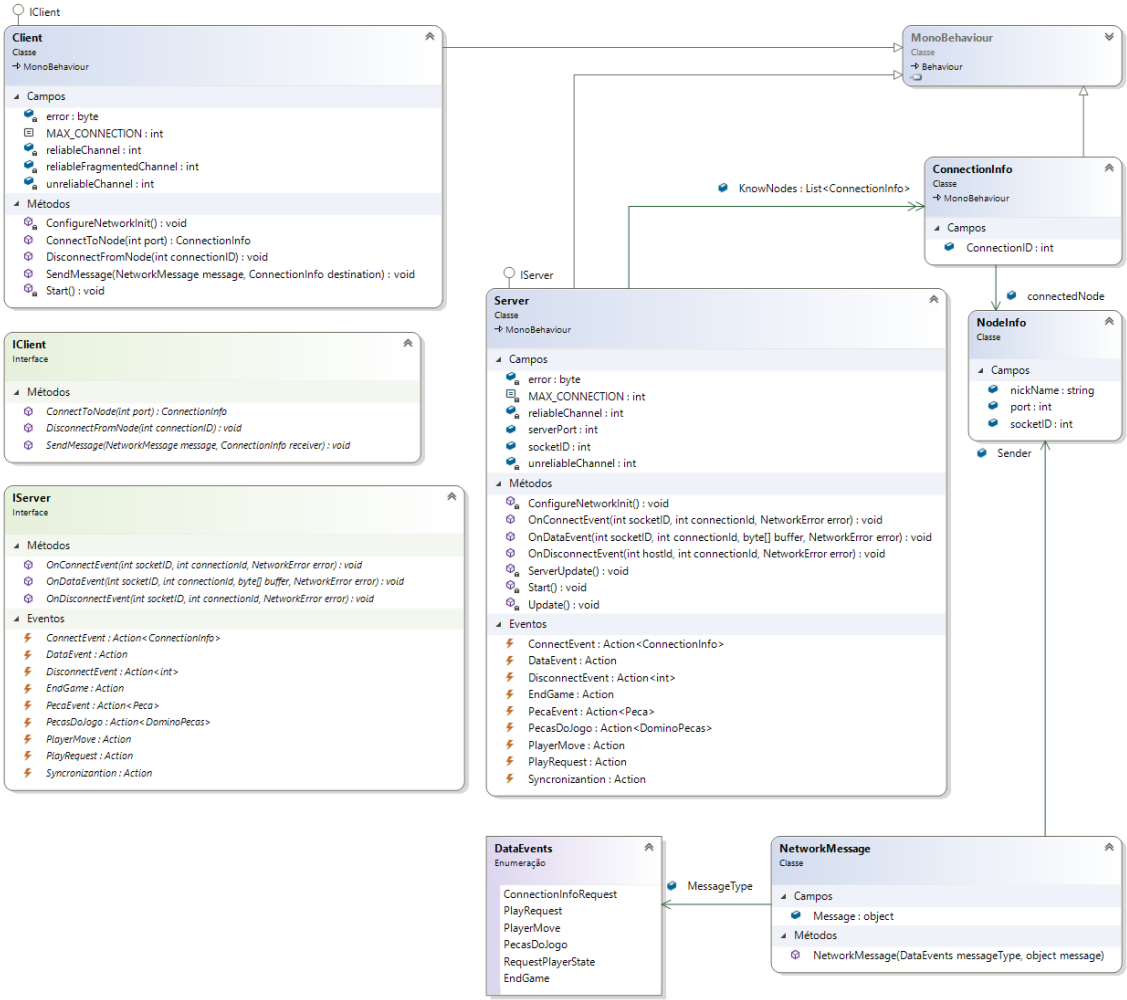


Figura 7: Diagrama de classes

Este Diagrama apresenta todas as classes do módulo de networking.

As classes *IServer* e *IClient* são as interfaces acessadas pelos demais módulos para utilizar os recursos de rede.

A interface *IServer* expõe métodos para quando um evento de rede ocorre como uma conexão ou um dado recebido e quando alguém se desconecta.

A classe que implementa esta interface é a classe Server, que utiliza a biblioteca *Unet* para implementar recursos de rede. Esta classe armazena a porta do player, o ID do canal confiável e não confiável de envio de dados (explicado em mais detalhes na seção 5.3 - Networking) assim como o ID do socket aberto para comunicação e eventos para cada tipo de mensagem que pode ser recebida pelo nó, como pedidos de conexão, recebimento de dados e desconexão de nós.

O método *ServerUpdate* é responsável por ficar analisando o socket aberto pelo nó para detectar o recebimento de alguma mensagem.

A Classe server também possui uma lista da classe *ConnectionInfo*, responsável por armazenar os dados de cada jogador que já se conectou com este nó.

A classe connection Info armazena o ID de uma conexão e uma instancia da classe *NodeInfo*, que contém os dados do nó conectado, como apelido e porta do socket.

A interface *IClient* por sua vez, expõe métodos para se conectar, enviar dados e se desconectar de um nó, sendo implementada pela classe *Client*, que expõe métodos para se conectar a um nó, enviar mensagens para ele e se desconectar do mesmo.

A classe *networkMessage* é responsável por armazenar a mensagem a ser enviada pela rede, assim como o tipo de mensagem e os dados do remetente, representado pelo *NodeInfo*.

O enumerador *DataEvents* armazena os tipos possíveis de mensagens a serem enviados pela rede.

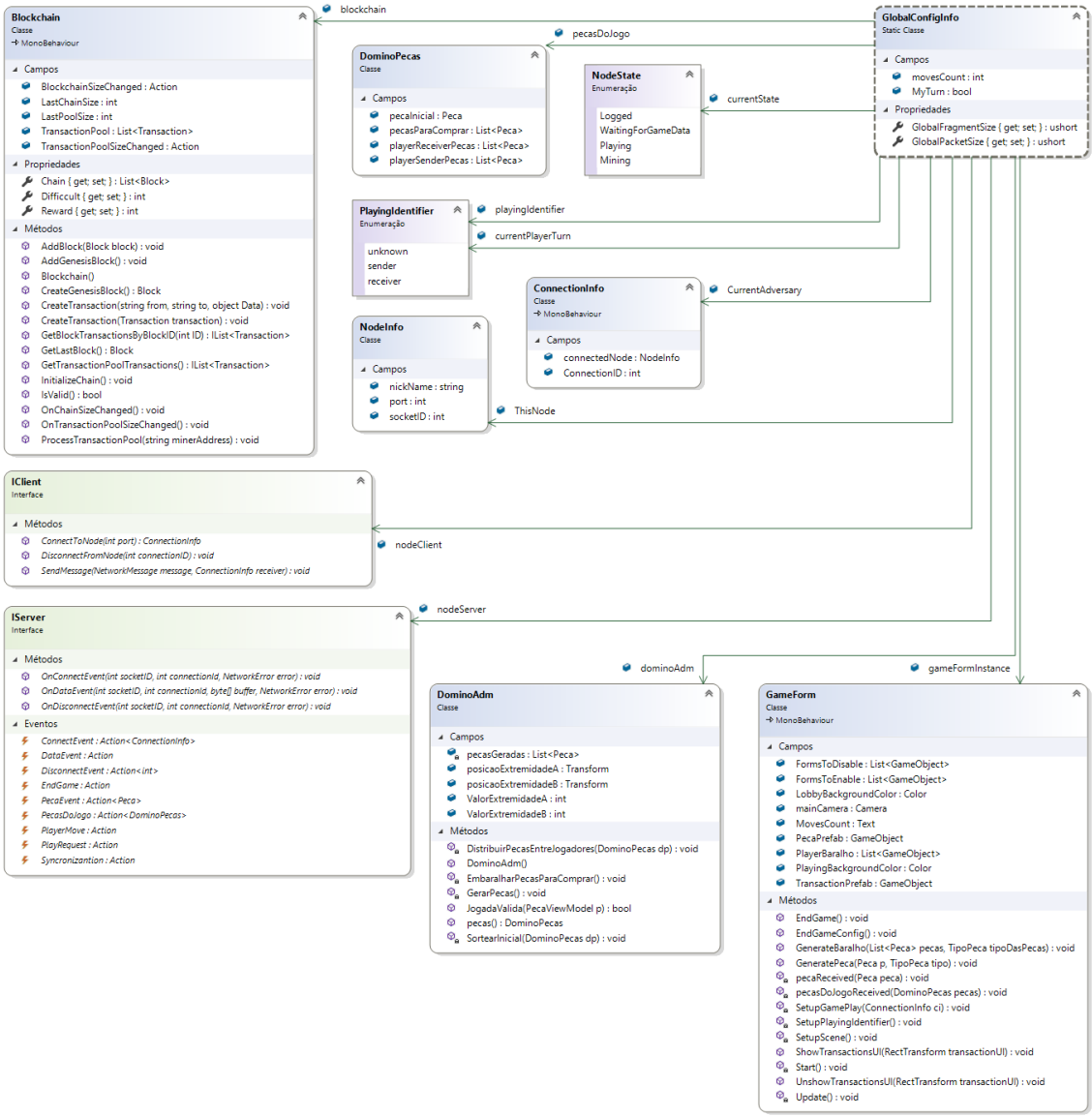


Figura 8: Diagrama de classes

Este último diagrama mostra a relação das classes apresentadas anteriormente com a classe *GlobalInfo*, que é o ponto central da aplicação.

Nesta classe, temos a referência às interfaces de rede (*ICLient* e *IServer*), ao *DominoAdm*, a instância única do *GameForm*, a informação do nó atual, a informação do adversário atual e o baralho gerado.

Também há referência a *blockchain*, ao estado atual do nó (jogando, aguardando, minerando) e ao identificador de jogador (se o nó atual foi quem enviou ou recebeu a solicitação de jogo).

## 4.3. Diagramas de sequência

Com o objetivo de mostrar como é o fluxo da aplicação construída, os diagramas de classes a seguir apresentam as relações entre os três módulos construídos (*Gamelogic*, *Network* e *Blockchain*).

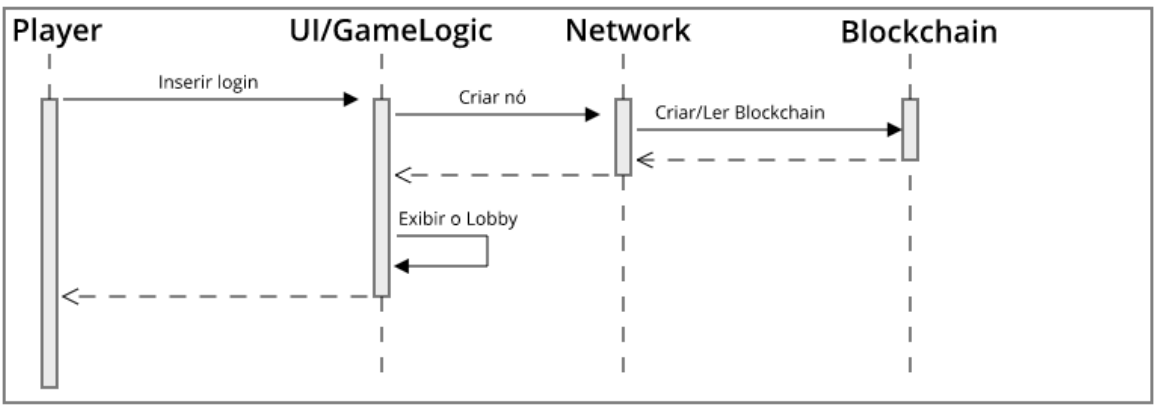


Figura 9 – Diagrama de Sequência para Login

O diagrama de sequência da figura 9 representa o caso de uso “Fazer Login”.

O *Player* interage com a aplicação através da interface visual (UI) que faz parte da *Gamelogic* que, ao inserir os dados de login (porta e apelido), cria um nó na rede local com essas informações.

O módulo de rede (*Network*) então busca pela *blockchain* em um diretório pré definido. Se não a encontrar, cria a *blockchain* com um bloco gênesis.

Por fim, a *Gamelogic* exibe a próxima tela (*Lobby*) para o player, onde ele é capaz de consultar a *blockchain* via interface visual da Unity e estabelecer uma nova conexão com outro jogador.

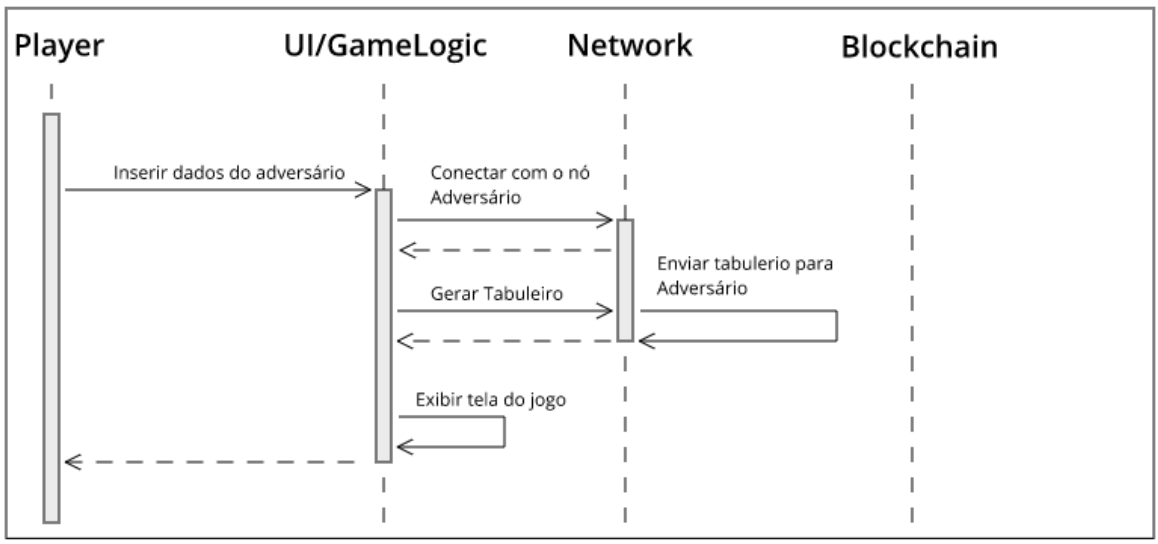


Figura 10 – Diagrama de Sequência para Desafiar outro jogador

O diagrama de sequência da figura 10 representa o caso de uso “Desafiar outro Player”.

Uma vez no *Lobby*, o player insere os dados (na implementação atual, a porta do adversário), a *Gamelogic* passa esses dados informados para o módulo de rede, que estabelece uma conexão com o nó adversário.

Em seguida, a *Gamelogic* gera o tabuleiro de jogo (sorteia as peças de cada jogador, a peça inicial e as peças da pilha de compra) e entrega para o módulo de rede, que envia esses dados ao adversário.

Em seguida, a *Gamelogic* exibe a tela do jogo, com a peça inicial, a pilha de comprar, as peças do player e as peças do adversário.

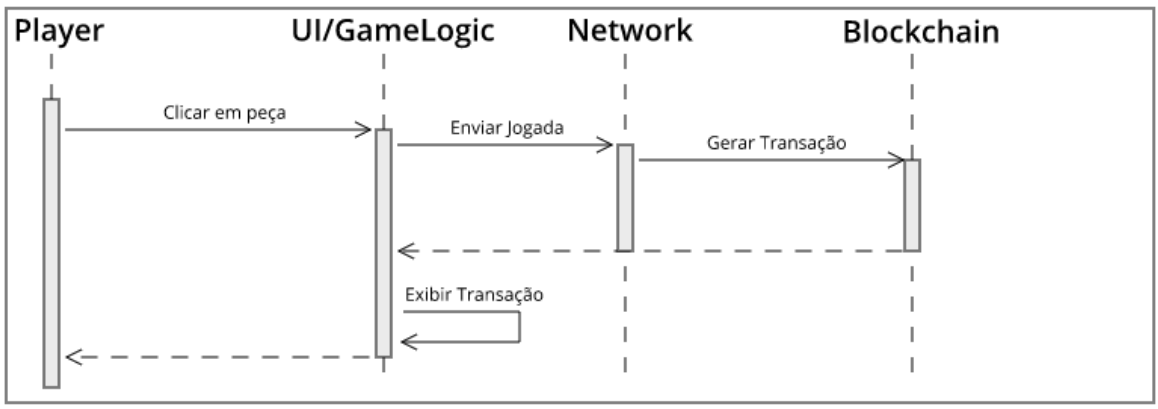


Figura 11 – Diagrama de Sequência para realizar jogada

O diagrama de sequência da figura 11 representa o caso de uso “Enviar jogada” e “Envia peça comprada”.

A próxima ação do player é uma ação do jogo em si, onde ele clica na peça que ele deseja jogar.

A *gamelogic* é responsável por reconhecer qual peça foi clicada e passar essa informação para o módulo de rede, que envia os dados dessa peça para o adversário, para que o jogo dele possa exibir qual peça o player jogou.

No ato de selecionar a peça, o módulo de rede solicita o módulo de *blockchain* a criação de uma transação, a ser adicionada no bloco atual.

Após isso, a camada de *Gamelogic* atualiza a visualização do player e do adversário, colocando a peça selecionada na mesa.

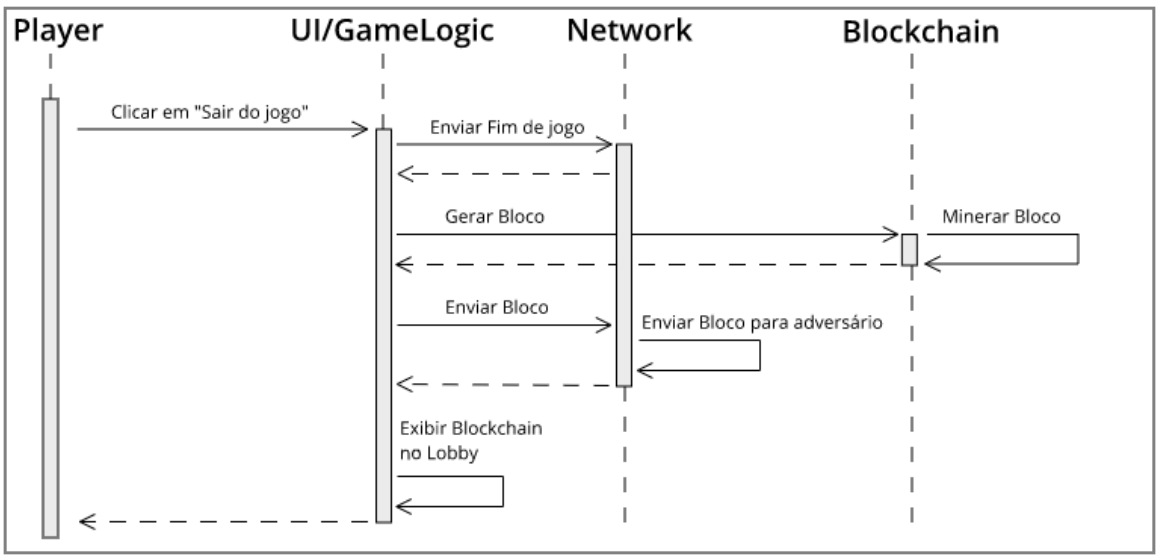


Figura 12 – Diagrama de sequência para encerrar o jogo

O diagrama de sequência da figura 12 representa o caso de uso “Minerar Bloco”.

No momento, para se encerrar um jogo, basta que o Player pressione o botão “Sair do jogo” para terminar a partida.

Ao pressionar este botão, o módulo *GameLogic* passa a informação de fim de jogo para o módulo de Network que envia essa informação para o adversário.

Em seguida, a *Gamelogic* solicita ao módulo de *blockchain* a criação de um bloco com as transações (jogadas feitas na partida) e minera esse bloco, o enviado após o processo de mineração.

Por fim, a tela de Lobby é exibida, permitindo que o jogador inicie um novo jogo com outro ou o mesmo adversário.

## 4.4. Aplicação Final


Figura 13 – Tela de Login

O caso de uso “Fazer Login” ocorre na tela da figura 13. O *player* precisa inserir um valor numérico no campo A e um valor alfanumérico no campo B.

O campo A representa a porta que será utilizado para abrir o socket de comunicação de rede, enquanto o campo B será utilizado para informar a aplicação qual o apelido do player.

Por fim, ao pressionar o botão “Login”, a aplicação abre um socket com o valor informado para a porta e define o apelido do *player,* que será utilizado no envio das transações na rede.

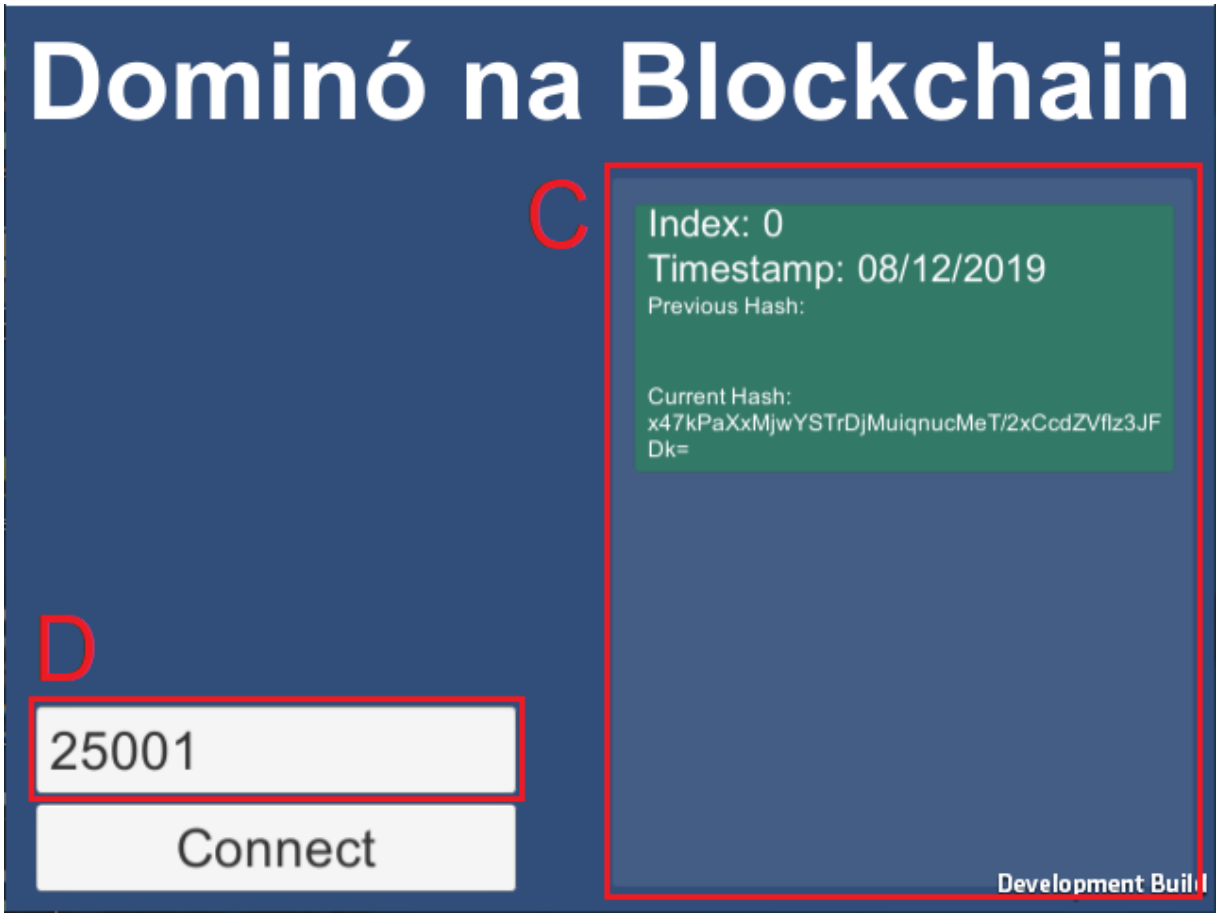


Figura 14 – Lobby

A tela da figura 14 é chamada de Lobby. Na versão atual da aplicação, ela é responsável pelos casos de uso “Desafiar outro jogador” e “Consultar blockchain”.

O painel C, na lateral direita da imagem, é por onde o *player* pode ver os blocos da *blockchain* e suas informações mais importantes, como o index, a data de criação, o *hash* anterior[20] e o *hash* do bloco em sí.

O caso de uso “desafiar outro player” se consolida através do formulário do lado esquerdo da tela, onde o *player* insere no campo D o número da porta do adversário (valor informado na tela de login do adversário) e ao pressionar o botão *connect,* estabelece uma conexão com o *player* adversário e ambos mudam para a tela de jogo.

Importante mencionar que ao pressionar o botão de *connect*, o *player* gera o baralho do jogo, definindo quais peças são dele, do adversário, qual será a peça inicial e as demais estarão disponíveis na pilha de compra.

---

[20] É válido mencionar que, pelo fato de o bloco da imagem ser o primeiro bloco da *blockchain*, conhecido como bloco gênesis, ele não possui hash anterior.

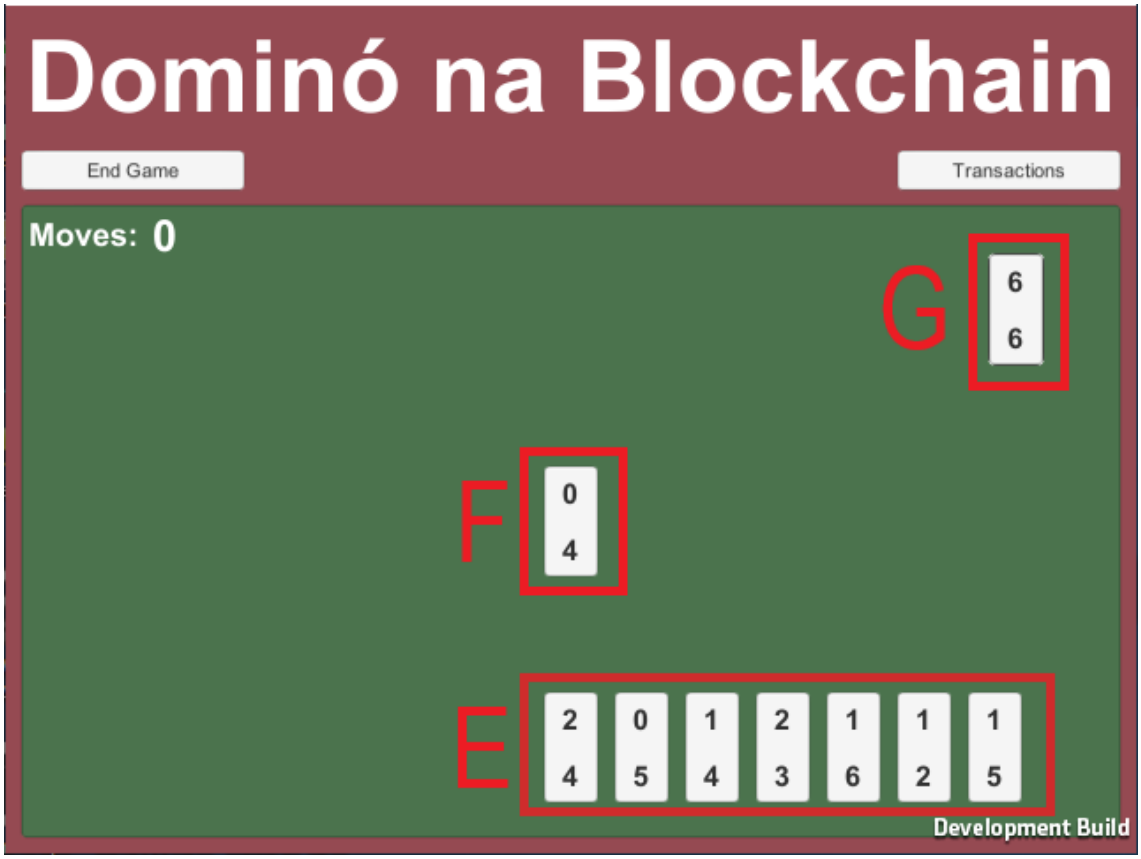


Figura 15 – Tela de jogo

Os casos de uso “enviar jogada”, “enviar peça comprada”, “criar transação”, “consultar transações” e “Minerar o bloco” são todos realizados através da tela de game, exibida na figura 15.

A peça F representa a peça inicial, sorteada pelo jogador que fez o desafio.

As peças no espaço E são as peças disponíveis para jogar. É válido mencionar que estas peças são diferentes entre o *player* desafiante e o seu adversário. Ao clicar nelas, o *player* executa o caso de uso “enviar jogada”, onde ele enviar o valor da peça clicada para o adversário.

A peça encontrada no espaço G é a pilha de peças para comprar. Ao clicar na peça desta pilha, o *player* executa o caso de uso “comprar peça”, onde ele recebe a peça da pilha e é capaz de joga-la.

Toda vez que o player clica em uma peça do espaço E ou do Espaço G, ele também executa o caso de uso “criar transação”, inserindo os dados da jogada para o bloco atual e enviando esses mesmos dados para o adversário, permitindo que ele atualize seu registro de jogadas.

Por fim, ao pressionar o botão “End Game”, o *player* encerra o jogo, iniciando assim o ato de minerar o bloco atual com todas as transações que foram criadas durante a partida. Após minerar o bloco, o jogador é enviado para a tela de Lobby, representada pela figura 11.

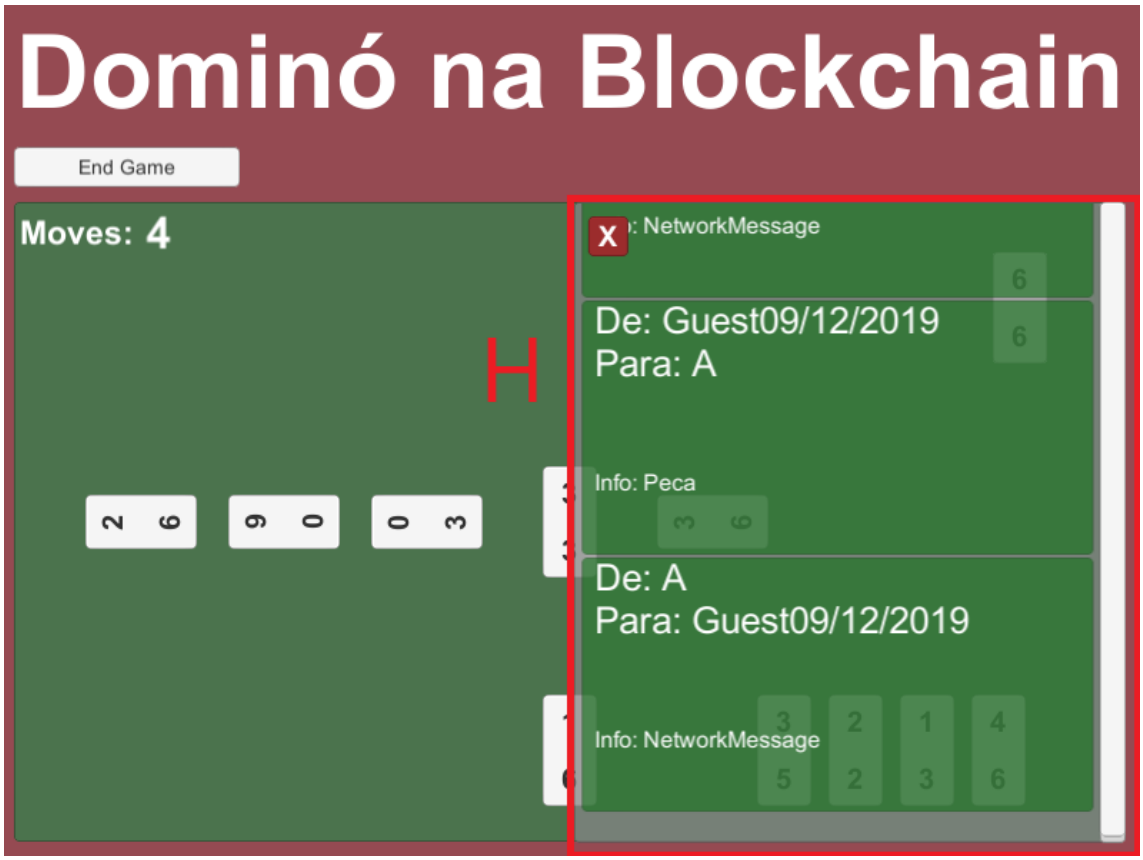


Figura 16 – Consultar transações

O caso de uso “consultar transações” é realizado através da tela exibida na figura 16. Ao pressionar o botão “Transactions”, o menu presente no espaço H é exibido, mostrando todas as transações da partida atual.

É válido mencionar que as transações armazenam o seu remetente, seu destinatário, e o tipo de dado que foi enviado.

No exemplo atual, existe a mensagem do tipo *NetworkMessage*, que serve para enviar dados genéricos pela rede. No caso atual, o baralho gerado pelo Player 0, que é o desafiante e, portanto, o responsável por gerar o baralho.

A segunda transação é uma transação do tipo Peca, responsável por enviar pela rede os dados de peça jogado por um dos *players*.

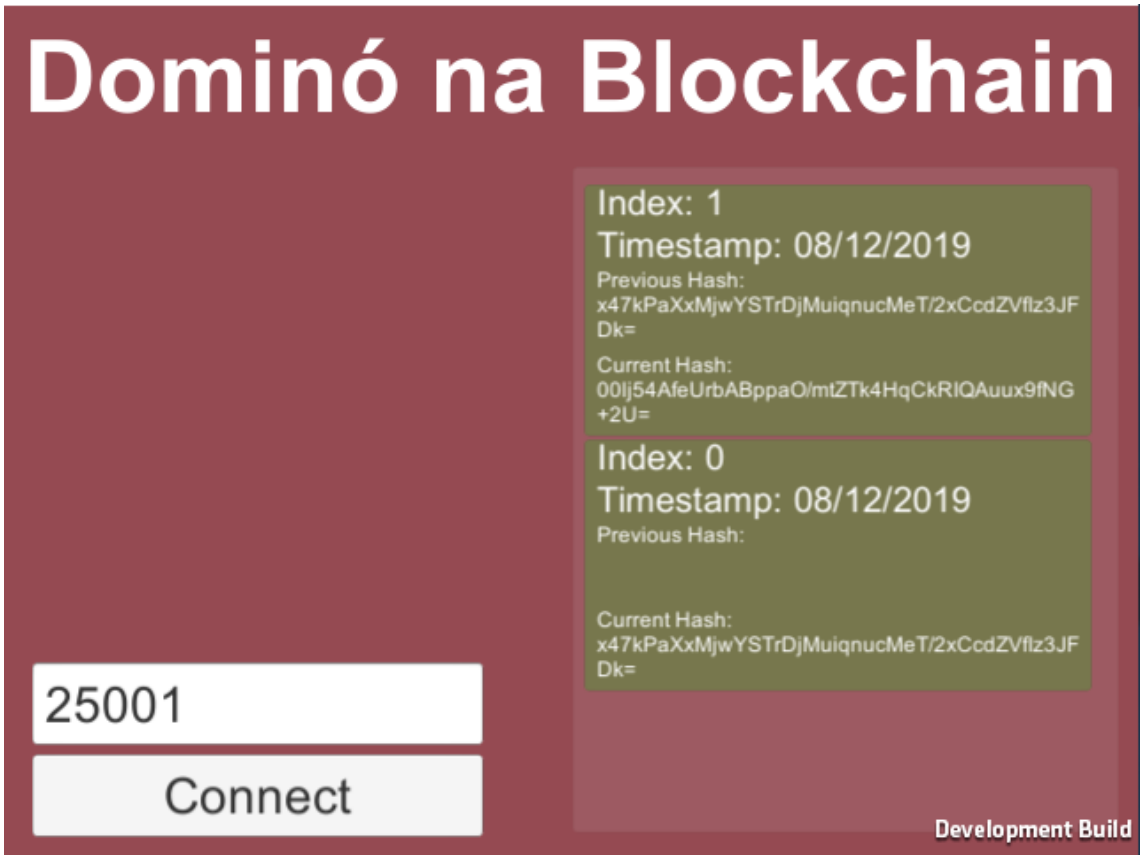


Figura 17 – Lobby após jogo

A tela da figura 17 mostra a aplicação após o primeiro jogo. É importante da ênfase ao fato de existir um segundo bloco na imagem, caracterizando o sucesso da ação de minerar o bloco.

O segundo bloco possui no campo “*Previous Hash*” o mesmo valor do *hash* do primeiro bloco, aplicando o conceito de blocos criptograficamente acorrentados pelo *hash*.

### 4.5. Detalhes da Implementação

Visando alcançar o objetivo proposto, a aplicação se divide em três partes responsáveis por aspectos distintos do produto final, nomeadas *Gamelogic*, *Blockchain* e *Networking*.

Essas partes foram desenvolvidas pensando no desacoplamento, a fim de garantir que possam ser substituídas por implementações desenvolvidas pelas

equipes que estiverem usando esta biblioteca, permitindo que apliquem suas próprias regras de negócio.

### 4.6. GameLogic

*Gamelogic* é a parte responsável por implementar a lógica do jogo que será persistido na *blockchain* e, no contexto deste trabalho, representa a implementação de um jogo de dominó. Entretanto, pode ser substituída por qualquer projeto *gamificado* que precise utilizar os recursos das outras partes da aplicação (conexão *peer-to-peer* e *blockchain*).

Mesmo em um cenário desacoplado, esta é a parte unificadora das demais, pois ela que utiliza os recursos de rede para enviar e receber dados dos nós, também sendo responsável por definir o que faz parte das transações e quando um novo bloco é minerado.

É válido mencionar que a *blockchain* deste projeto é privada, como definido por Jayachandran (2017), uma *blockchain* pode ser pública ou privada, onde uma *blockchain* pública permite que qualquer um entre e participe da rede, onde geralmente existe um incentivo para que cada vez mais participantes venham fazer parte da rede.

*Blockchains* privadas exigem um convite para se fazer parte, e a validação das transações fica por conta do nó inicial ou por uma série de regras definidas pelo nó inicial, sendo assim, algoritmos de validação como o *proof of work* não se fazem necessários, uma vez que os nós que mantém a *blockchain* são também os que validam as transações.

Levando em consideração que a implementação atual funciona com mineração, para fins de validação do conceito, jogar dominó colocando cada transação atômica na *blockchain* seria análogo a jogar dominó através de um serviço postal. Sendo assim, cada jogador enviaria sua peça por carta para seu adversário, que receberia a peça e retornaria a carta com a sua jogada, algo a princípio inviável em termos de tempo.

Neste caso, apesar das peças enviadas entre os jogadores ser um dado importante e que será posteriormente transacionado para a *blockchain*, eles precisam ser enviados de forma rápida para que o fluxo do jogo não seja interrompido.

É importante mencionar que a *blockchain* não é responsável pela segurança da informação em termos de envio e recebimento pela rede, não sendo função dela definir como um dado é enviado, garantir que os dados cheguem em ordem no destino ou como deve ser a reação à perda de pacotes durante o envio.

Para tal, a Unity provê um recurso chamado *Quality of service*[21] (explicado em maior detalhes na seção 5.1.3 - Networking) que é utilizado para garantir que a informação enviada a partir de um nó seja recebida íntegra no nó destino.

A implementação do dominó foi feita sobre o sistema de UI da Unity, onde todas as peças na verdade são botões da UI e ao serem clicados, executam um código de validação da jogada e enviam as informações da peça selecionada para o nó adversário.

## 4.7. Blockchain

Esta parte é a *blockchain* em si, que armazena todos os métodos e propriedades necessários para o funcionamento pleno da *blockchain*, feita baseando-se na implementação de Henry He (2018), que mostra como implementar uma *blockchain* no ambiente .Net.

A classe *Blockchain* armazena a quantidade de zeros que são necessários para que um novo bloco seja aceito, quantas moedas o minerador recebe ao processar um novo bloco e eventos que são acionados sempre que um novo bloco ou transação surge.

Também é responsável por armazenar o *pool* de transações que serão inseridas no próximo bloco e a cadeia de blocos gerados (*blockchain*). Atualmente o *pool* é preenchido a cada peça que os jogadores enviam um para o outro durante a partida.

Além disso, métodos que executam ações auxiliares, como criar o bloco *gênesis* e validar a corrente são encontrados aqui.

É válido mencionar a existência da classe auxiliar chamada *BlockchainImporterExporter* responsável por gerar um arquivo com extensão *txt* contendo os blocos da *blockchain*, além de ler esse mesmo arquivo quando necessário.

---

[21] define como a *engine* deve enviar os dados através da rede e como reagir em casos de perda de pacotes.

As classes de modelo da transação e do bloco também se encontram nesse projeto, incluindo a função responsável por minerar o bloco que se encontra na classe de modelo do bloco.

Levando em consideração que o algoritmo de *proof-of-stake* exige uma rede distribuída funcional para seu pleno funcionamento e teste, e que a aplicação atual funciona apenas em um contexto local, este algoritmo não será utilizado nesta implementação.

O algoritmo de human mining por sua vez, possui uma natureza que é mais difícil de se implementar, principalmente por ser um algoritmo novo, possuindo um número limitado de fontes de pesquisa.

Por fim, o algoritmo de consenso utilizado nesta solução foi o *proof-of-work*, pela facilidade da implementação e de teste de seu funcionamento, além da existência do trabalho de Henry Re que facilita a consulta de como deveria funcionar este algoritmo.

Esta implementação do *proof-of-work* utiliza apenas dois zeros como critério de validação do bloco, para fins de teste e verificabilidade.

## 4.8. Networking

Apesar de existir uma implementação de rede neste trabalho, o objetivo é aplicar recursos da *blockchain* em uma aplicação gamificada, sendo esta implementação de rede apenas para fins de teste e apresentação do conceito. Para aqueles que pretendem utilizar este código fonte em seus jogos, é recomendado que uma implementação de recursos de rede mais robusta esteja disponível.

O projeto nomeado *Networking* é responsável por toda a parte de comunicação entre os nós. Tendo em vista que a aplicação opera de forma *peer-to-peer*, existem as classes servidor e cliente, responsáveis respectivamente por receber mensagens de outros nós e enviar mensagens para outros nós.

É válido mencionar que a parte de rede foi implementada utilizando uma biblioteca nativa da Unity chamada *UNet*, que possui uma série de recursos comumente utilizados em jogos online, incluindo uma API de baixo nível, conhecida como *Low Level API* ou ainda *Transport Layer API*, utilizada extensivamente neste projeto, utilizando como referência os códigos fonte disponíveis na página “Using The Transport Layer API” (Unity).

Para permitir o desacoplamento e a modularidade da solução, tanto o servidor quanto o cliente são representados por interfaces (*IServer* e *IClient*) que são implementados pelas classes Server e *Client*. Desta forma, esta implementação que utiliza a LLAPI se torna dispensável, podendo ser substituída por qualquer implementação que os desenvolvedores criem.

Um ponto a ser levantado é relativo a segurança das informações transitadas. Em circunstâncias comuns, dados que precisam ser recebidos integralmente pelo receptor seriam enviados em um canal TCP, que garante a entrega dos dados ao destino. Porém, uma das características da UNet é operar sobre UDP exclusivamente, que é rápido, mas incapaz de garantir que a mensagem foi enviada integralmente e em ordem por exemplo.

No cenário atual, é necessário garantir que o nó adversário recebeu a peça que foi enviada, e para tal, o fato da *UNet* expor uma série de propriedades internas que podem ser modificadas conforme as necessidades da aplicação, como mencionado por Abramychev (2014) “Se você der uma olhada na implementação de um socket TCP você pode encontrar toneladas de parâmetros (*timeouts, Buffer lenght etc.)* que você pode alterar. Nós escolhemos seguir uma abordagem similar e permitir que usuários mudem praticamente todos os parâmetros de nossa biblioteca [...]” se mostra uma característica valiosa.

A UNet permite que o desenvolvedor defina o que chamam de “canal”, que define uma série de parâmetros utilizados no envio de dados entre os nós. Um desses parâmetros chama-se *Quality-of-Service* (QoS), que define como este canal deve se comportar em relação ao reenvio de uma mensagem. Um desses QoS garante a entrega da mensagem, não necessariamente em ordem, nomeado *Reliable*, entregando a garantia de que a peça enviada foi entregue ao destino.

Tendo em vista que apenas uma peça pode ser enviada por vez, o risco de duas peças sequenciais serem enviadas em ordens diferentes, acarretando em uma inconsistência no nó adversário é eliminada.

## 4.9. Análise crítica

A Unity exige que todos os componentes que venham a ser utilizados anexados a *GameObjects* na janela *Hierarchy* sejam derivados de *monobehaviour*. Isso acaba exigindo que o código gerado seja derivado de *monobehaviour*,

adicionando uma série de funções e propriedades exclusivas da Unity, fora o fato do código não poder ser reutilizado fora da *engine*.

Além disso, um problema é a impossibilidade de utilizar a palavra reservada "*new*" para criar novas instâncias. O ato de criar uma instância na Unity é feito pelo método *instantiate* que obrigatoriamente cria um *GameObject* na hierarquia do projeto, o que nem sempre é o desejado. Por vezes queremos instanciar uma classe de modelo, sem a necessidade de explicitá-la na *engine* no momento de sua criação.

Outro ponto a ser levantado sobre a execução deste projeto foi a impossibilidade de utilizar construtores para iniciar objetos. Apesar da existência dos métodos *Start* e *Awake*, por vezes se faz necessário criar um novo objeto passando valores pré-definidos no construtor.

Uma possível solução para estes problemas que foi implementada no projeto envolve um padrão de projeto chamado MVVM, ou *model-view-viewModel*. Apesar de não ser uma implementação completa do padrão, a ideia principal está presente.

Para cada classe de modelo que precisa ser exibida na UI da Unity, como por exemplo as transações da *blockchain* ou o próprio bloco da *blockchain*, uma classe que deriva de *monobehaviour* é criada, armazenando uma referência do modelo que ela representa e das estruturas exclusivas do *monobehaviour* que serão utilizadas na UI, como *Text*, *Input Field* e *Scrollview* por exemplo.

Essa classe é conhecida como *ViewModel*, e é responsável por armazenar tantos dados de visualização (Text, Input Field, ...) como o próprio modelo. Nesta implementação, levando em consideração que a Unity possui um método chamado *Start*, executado sempre que o objeto seja instanciado, todos os dados da *Model* são passados para os elementos de UI da Unity nesta chamada.

Em termos de rede, o desafio de se utilizar a *UNet* é a documentação antiga e escassa, por vezes imprecisa, dificultando a identificação de erros, o entendimento sobre como funciona e dificultando a extensão de suas funcionalidades. Cenário este compreensível tendo em vista que a Unity pretende abandonar esta biblioteca, como mencionado no site oficial da *engine* (House, 2018).

Quanto a *blockchain*, a implementação é primitiva, seguindo os tutoriais do Henry He (2018). Ele apresenta os conceitos mais básicos de uma *blockchain*, como blocos conectados através de um *hash* e a implementação do *proof of work*.

Tendo em vista que a *blockchain* criada reside apenas em uma máquina, esta é altamente suscetível ao ataque dos 51%, pois o poder de processamento empregado atualmente não passa a de um *desktop* caseiro.

Além disso, caso este projeto se torne público no estado atual, um ataque *sybil* a fim de tomar o controle da rede é possível, tendo em vista que executar múltiplas instâncias desta aplicação é uma tarefa trivial.

Nesse contexto, o produto atual só funciona em uma rede local, sendo o ideal que a aplicação seja capaz de se comunicar com dispositivos em redes externas, o que caracteriza um ponto a ser aprimorado no futuro.

Um bug conhecido é a incapacidade de serializar as transações dos blocos já minerados na UI da Unity, onde a solução encontrada para fins de demonstração é a exclusão do arquivo com a *blockchain* e a criação dele toda vez que a aplicação for executada.

Melhorias na implementação de rede também se fazem necessárias. O ato de conexão com outro jogador não faz troca de informações entre os nós, impedindo a aplicação de utilizar o apelido do outro jogador nos campos das transações.

Além disso, um problema de *serialização* existe no ato de ler o arquivo com a *blockchain* que se encontra na pasta de *resourcers* da solução, impedindo que o jogo inicie caso já exista um arquivo de *blockchain* dentro da pasta *resourcers*.

## 4.10. Trabalhos relacionados

Válido mencionar que Jacob Eberhardt fez uma série de estudos a fim de mostrar “como mover processamento e dados para fora da corrente, sem comprometer as propriedades introduzidas e os benefícios ganhos ao se utilizar *blockchains*” (Eberhardt, 2017).

Além de Jacob, Paul Grau publicou em 2016 um estudo sobre as lições aprendidas ao se desenvolver um jogo de xadrez para o *Ethereum* (Grau, 2016), supervisionado pelo departamento de engenharia de sistemas da informação do *Institut Wirtschaftsinformatik und Quantittative Methoden* e pelo Doutor Christian Reitwiessner da Fundação Ethereum, mostrando que uma aplicação gamificada é capaz de fazer uso de um canal que não seja validado pela *blockchain*, intitulado “*offchain*” para dados que não precisem ser persistidos e que devam ser enviados de forma rápida.

Também existe o trabalho de grupos como o Enjin, fundado em 2009, que mantém uma plataforma com produtos baseados em *blockchain* para que qualquer um possa facilmente criar, gerenciar, distribuir e integrar recursos de *blockchain* em seus jogos, utilizando a moeda digital ENJ, baseado no Ethereum, e garantindo uma série de vantagens como transformar os itens obtidos em jogo em ENJ e a escassez de itens que, conforme diminui a sua disponibilidade, seu valor aumenta.

Além do Enjin, o grupo Xaya mantém uma plataforma que permite a adição de recursos de *blockchain* em produtos gamificados de forma robusta e altamente escalável, incluindo a elaboração de um novo conceito de mineração, conhecido como *Human Mining* (Xaya, 2018), explicado na seção 2.4 – Algoritmos de consenso, permitindo que jogos transformem as ações lúdicas executadas pelos jogadores em moedas digitais, que possuem valor no mundo real.

# 5. CONCLUSÃO

O presente trabalho apresenta um jogo de dominó implementado na *game engine* Unity, que utiliza uma *blockchain* implementado e capaz de operar em rede local, entregando um passo inicial para se alcançar um jogo que utiliza uma *blockchain* pública.

Em cenários onde produtoras deixam de dar manutenção ou até mesmo desligue os servidores de um jogo online (Vieira, 2015), acarretando na perda de itens, moedas e todo tipo de ativo digital por parte dos jogadores, a *blockchain* se apresenta como uma alternativa promissora para garantir que os ativos conquistados por eles estejam em sua posse, independente de ações como essas.

Além disso, as próprias produtoras são capazes de se beneficiar deste cenário, uma vez que os custos de se manter um servidor para armazenar os dados dos jogadores e gerenciar as partidas é reduzido (quando não eliminado), tendo em vista que a comunicação durante as partidas é feita em uma arquitetura peer-to-peer, sem a necessidade de alocação de um servidor dedicado.

Além disso, os dados dos jogadores são mantidos pelos próprios jogadores, analogamente ao livro razão do bitcoin, mantido pelos próprios nós participantes da rede do bitcoin.

Este trabalho se mostra um passo inicial para o desenvolvimento de *blockchains* melhor preparadas para jogos online e capazes de agregar todos os pontos citados acima.

Em estudos futuros, a busca de algoritmos de consenso mais rápidos e com menor desperdício de recursos computacionais é uma possível linha de pesquisa, além de uma implementação

## 6. REFERÊNCIAS


ABRAMYCHEV, A.; **All about the Unity networking transport layer**. 2014. Disponível em: <https://blogs.unity3d.com/pt/2014/06/11/all-about-the-unity-networking-transport-layer/>. Acesso em: 17 Nov. 2019.

ATZEI, N.; BARTOLETTI, M.; CIMOLI, T. **A Survey of Attacks on Ethereum Smart Contracts (SoK)**. 2017. Disponível em: <https://link.springer.com/chapter/10.1007/978-3-662-54455-6_8>. Acesso em: 08 Maio. 2019.

BACK, A.; **HASHCASH - A DENIAL OF SERVICES COUNTER-MEASURE.** 2002. Disponível em: <http://www.hashcash.org/papers/hashcash.pdf>. Acesso em: 08 Abr. 2019.

BBC News. **Gaming worth more than video and music combined.** 2019. Disponível em: <https://www.bbc.com/news/technology-46746593>. Acesso em: 30 Out. 2019

Bethesda Game Studios. **Fallout 76: Hotfix Notes - November 12, 2019**. Disponível em: <https://fallout.bethesda.net/en/article/3aDmdi6Rk91U832uk2o7BW/fallout-76-hotfix-notes-november-12-2019>. Acesso em: 30 Nov. 2019.

Britannica Escola. **Jogo Eletrônico.** Disponível em: <https://escola.britannica.com.br/artigo/jogo-eletr%C3%B4nico/481214>. Acesso em: 18 Nov. 2019.

CHAUM, D. **BLIND SIGNATURES FOR UNTRACEABLE PAYMENTS.** 1998. Disponível em: <https://sceweb.sce.uhcl.edu/yang/teaching/csci5234WebSecurityFall2011/Chaum-blind-signatures.PDF>. Acesso em: 07 Abr. 2019.

CHAIN, R. **The biggest Easter egg in Ready Player One is blockchain.** 2018. Disponível em: <https://medium.com/ruffchain/the-biggest-easter-egg-in-ready-player-one-is-blockchain-234bd5ec90e0>. Acesso em: 30 Out. 2019

CHOPRA, S.; SCOTT, W.; SEARS, C.; VASANAD, S.; **AUTOMATED HOTFIX HANDLING MODEL.** 2014. Disponível em: <https://patentimages.storage.googleapis.com/cb/ef/36/5f6c82aff6fb6c/US8713554.pdf>. Acesso em: 17 Nov. 2019.

CRONIN, E.; FILSTRUP, B.; KURC, A. **A distributed Multiplayer Game Server System.** 2001. Disponível em: < https://svn.sable.mcgill.ca/sable/courses/COMP763/oldpapers/cronin-01-distributed.pdf>. Acesso em 02 Dez. 2019.

CLAYPOOL, M.; CLAYPOOL, J. **On Latency and Player Actions in Online Games.** 2006. Disponível em: <https://digitalcommons.wpi.edu/cgi/viewcontent.cgi?article=1053&context=computerscience-pubs>. Acesso em: 30 Out. 2019

DWORK, C.; NAOR, M; **PRICING VIA PROCESSING OR COMBATTING JUNK MAIL.** 1992. Disponível em: <http://www.wisdom.weizmann.ac.il/~naor/PAPERS/pvp.pdf>. Acesso em: 07 Abr. 2019.

EBERHART, J.; TAI, S; **On or Off the Blockchain? Insights on Off-Chaining Computation and Data.** Disponível em: <http://www.ise.tu-berlin.de/fileadmin/fg308/publications/2017/2017-eberhardt-tai-offchaining-patterns.pdf>. Acesso em: 30 Out. 2019.

ENJIN. **Catching a Uber with cryptocurrency Won in a Video Game.** 2019. (2m38s). Disponível em: <https://www.youtube.com/watch?v=Xn-o5hfzE7Y>. Acesso em: 30 Out. 2019

ETHEREUM. **"Ethereum/Wiki."** GitHub. Disponível em: <https://github.com/ethereum/wiki/wiki/Proof-of-Stake-FAQ#what-is-proof-of-stake>. Acesso em 08 Maio. 2019

FORBES. **How Secure Is Blockchain Technology?.** 2018. Disponível em: <https://www.forbes.com/sites/forbestechcouncil/2018/10/12/how-secure-is-blockchain-technology/#5152065872f0>. Acesso em: 17 Nov. 2019.

Fungível. In: **Michaelis, Dicionário Brasileiro da Língua Portuguesa.** Disponível em:<https://michaelis.uol.com.br/moderno-portugues/busca/portugues-brasileiro/fungível>. Acesso em: 08 Abr. 2019.

GAMBETTA, G; **Fast-Paced Multiplayer (Part I): Client-Server Game Architecture.** 2017 . Disponível em: <https://www.gabrielgambetta.com/client-server-game-architecture.html>. Acesso em 02 Dez. 2019.

GILBERT, S.; MALEWICZ, G. (2005) **The Quorum Deployment Problem.** Disponivel em: <https://groups.csail.mit.edu/tds/papers/Gilbert/TR-972.pdf>. Acesso em: 08 Abr. 2019.

GLOBOSAT.; **Além dos mitos: o perfil dos gamers no Brasil e no mundo.** 2019. Disponível em: <http://gente.globosat.com.br/esports/>. Acesso em: 17 Nov. 2019.

GRAU, P. **Lessons learned from making a Chess game for Ethereum.** 2016. Disponível em: <https://medium.com/@graycoding/lessons-learned-from-making-a-chess-game-for-ethereum-6917c01178b6>. Acesso em: 30 Out. 2019

HARBER, S.; Stornetta, W.S; **How to Time-Stamp a Digital Document.** 1991. Disponível em: <https://link.springer.com/article/10.1007/BF00196791>. Acesso em: 07 Abr. 2019.

HAAS, J. **A history of the Unity Game Engine.** Disponível em: <https://web.wpi.edu/Pubs/E-project/Available/E-project-030614-143124/unrestricted/Haas_IQP_Final.pdf>. Acesso em: 24 Abr. 2019.

HE, H. **Building a Blockchain in .NET Core.** 2018. Disponível em: <https://www.c-sharpcorner.com/article/blockchain-basics-building-a-blockchain-in-net-core>. Acesso em 26 Nov. 2019.

HILL-WHITTALL, R. **THE INDIE GAME DEVELOPER HANDBOOK.** 2015. Disponível em: <https://content.taylorfrancis.com/books/download?dac=C2014-0-32390-7&isbn=9781317573654&format=googlePreviewPdf>. Acesso em: 02 Dez. 2019.

HOUSE, B. **Evolving multiplayer games beyond Unet.** Unity Technologies. 2018. Disponível em: <https://blogs.unity3d.com/2018/08/02/evolving-multiplayer-games-beyond-unet/?_ga=2.106814659.157859476.1574023534-554063184.1564542217>. Acesso em: 17 Nov. 2019.

HUYNH, M. VALARINO, F**. An analysis of continuous consistency models in real time peer-to-peer fighting games.** 2019. Disponível em: <http://www.diva-portal.org/smash/record.jsf?pid=diva2%3A1322881&dswid=-8245>. Acesso em: 27 Nov. 2019.

JAYACHANDRAN, P. **The difference between public and private blockchain.** 2017. Disponível em: <https://www.ibm.com/blogs/blockchain/2017/05/the-difference-between-public-and-private-blockchain/>. Acesso em: 26 Nov. 2019.

JOGO. In: **Michaelis, Dicionário Brasileiro da Língua Portuguesa.** Disponível em: <http://michaelis.uol.com.br/busca?r=0&f=0&t=0&palavra=jogo>. Acesso em: 18 Nov. 2019.

JOGO. In: **DICIO, Dicionário Online de Português.** Porto: 7Graus, 2018. Disponível em: <https://www.dicio.com.br/jogo/>. Acesso em: 18 Nov. 2019.

LAMPORT, L.; SHOSTAK, R.; Pease, M. **The Byzantine Generals Problem.** 1982. Disponível em: <https://www-inst.eecs.berkeley.edu/~cs162/sp16/static/readings/Original_Byzantine.pdf >. Acesso em: 08 Abr. 2019.

LEAGUE OF LEGENDS; **Join Us Oct. 15th to Celebrate 10 Years of League.** 2019.
Disponível em: <https://na.leagueoflegends.com/en/news/game-updates/special-event/join-us-oct-15th-celebrate-10-years-league>. Acesso em: 17 Nov. 2019.

LOPES, L. **Jogos de mesa para idosos: análise e consideração sobre o dominó.** 2009. Disponível em: <http://www.teses.usp.br/teses/disponiveis/16/16134/tde-24032010-103339/pt-br.php>. Acesso em: 24 Abr. 2019.

MCDONALD, E.; Bosman, S. **Betting on Billions: Unlocking the Power of Mobile Gamers.** Newzoo. 2019. Disponível em: <https://cdn2.hubspot.net/hubfs/4963442/Whitepapers/ABM_Newzoo_Betting_on_Billions_Unlocking_Power_of_Mobile_Gamers_March2019.pdf>. Acesso em: 16 Nov. 2019.

MERCURI, L.; **Game Studio Report 2018.** Unity Technologies. Disponível em: <http://images.response.unity3d.com/Web/Unity/%7Bb9551473-6798-4308-b0b0-7e3fe5d01781%7D_The_Unity_Game_Studio_Report_2018.pdf?utm_campaign=___&utm_content=2018-08-global-game-studio-report-download&utm_medium=email&utm_source=Eloqua>. Acesso em 18 Nov. 2019.

NAKAMOTO, S. **bitcoin: A Peer-to-Peer Electronic Cash System. bitcoin.org.** [S.l.] [2008?]. Disponível em: <https://bitcoin.org/bitcoin.pdf>. Acesso em: 07 Abr. 2019.

OLUWATOSIN, H.; **Client-Server Model.** 2014. Disponível em: <https://pdfs.semanticscholar.org/e1d2/133541a5d22d0ee60ee39a0fece970a4ddbf.pdf>. Acesso em: 17 Nov. 2019.

Oracle. **What is a Socket?.** [S.l] Disponível em: < https://docs.oracle.com/javase/tutorial/networking/sockets/definition.html>. Acesso em: 28 Nov. 2019

Oracle. **Lesson 4: Serialization.** [S.l]. Disponível em: <https://www.oracle.com/technetwork/java/serial-137074.html>. Acesso em: 28 Nov. 2019

PRYOR, L.; **METHODS AND SYSTEMS FOR MONITORING A GAME TO DETERMINE A PLAYER-EXPLOITABLE GAME CONDITION.** 2009. Disponível em: <https://patentimages.storage.googleapis.com/da/20/3b/34e44de8d49641/US7517282.pdf>. Acesso em: 17 Nov. 2019.

QUORUM. In: DICIO, Dicionário Online de Português. Porto: 7Graus, 2018. Disponível em: <https://www.dicio.com.br/quorum/>. Acesso em: 08 Abr. 2019.

ROCHA, L.; eCash: **como a criação de David Chaum deu início ao sonho cypherpunk.** 2018. Disponível em: <https://www.criptofacil.com/ecash-como-a-criacao-de-david-chaum-deu-inicio-ao-sonho-cypherpunk/>. Acesso em: 08 Abr. 2019.

ROCHA, L.; Hashcash: **como Adam Back projetou o motor do bitcoin.** 2018. Disponível em: <https://bitcoinmagazine.com/articles/genesis-files-bit-gold-szabo-was-inches-away-inventing-bitcoin>. Acesso em: 08 Abr. 2019.

ROCHA, L.; Wei Dai: **como o seu B-Money inspirou a criação do bitcoin.** 2018. Disponível em: <https://bitcoinmagazine.com/articles/genesis-files-bit-gold-szabo-was-inches-away-inventing-bitcoin>. Acesso em: 08 Abr. 2019.

ROCHA, L.; Bit Gold: **Nick Szabo esteve a poucos passos de inventar o bitcoin.** 2018. Disponível em: <https://bitcoinmagazine.com/articles/genesis-files-bit-gold-szabo-was-inches-away-inventing-bitcoin>. Acesso em: 08 Abr. 2019.

ROSENFELD, M. **ANALYSIS OF HASHRATE-BASED DOUBLE-SPENDING.** [2012]. Disponível em: <https://www.bitcoil.co.il/Doublespend.pdf> Acesso em: 27 Out. 2019.

SCIENCEDIRECT. **Cryptocurrency.** Disponível em: <https://www.sciencedirect.com/topics/economics-econometrics-and-finance/cryptocurrency>. Acesso em 17 Nov. 2019.

SHIBATA, N. **Blockchain Consensus Formation while Solving Optimization Problems.** 2019. Disponível em: <https://arxiv.org/pdf/1908.01915.pdf>. Acesso em 28 Out. 2019.

SHYLENOK, P. **Understanding the roles of game dev professionals.** Gamasutra. 2019. Disponível em: <https://www.gamasutra.com/blogs/PavelShylenok/20190115/334322/Understanding_the_Roles_of_Game_Dev_Professionals.php>. Acesso em 26 nov. 2019.

STEAM CHARTS.; **PLAYERUNKNOWN'S BATTLEGROUNDS.** 2019. Disponível em: <https://steamcharts.com/app/578080> Acesso em: 17 Nov. 2019.

SZABO, N.; **SHELLING OUT: THE ORIGINS OF MONEY.** [2002?] Disponível em: <https://nakamotoinstitute.org/shelling-out/>. Acesso em: 08 Abr. 2019.

SZABO, N.; **Bit Gold Markets**. **Unenumerated**. 2008. Disponível em: <http://unenumerated.blogspot.com/2008/04/bit-gold-markets.html>. Acesso em 29 Nov. 2019

SZYDIO, M. (2004) **Merkle Tree Traversal in Log Space and Time.** 2004. Disponível em: <https://link.springer.com/content/pdf/10.1007%2F978-3-540-24676-3_32.pdf>. Acesso em: 08 Maio. 2019

TOFTEDAHL, M.; ENGSTRÖM, H.; **A Taxonomy of Game Engines and the Tools that Drive the Industry.** 2019. Disponível em: <https://www.diva-portal.org/smash/get/diva2:1352554/FULLTEXT01.pdf>. Acesso em: 18 Nov.2019.

Unity. **Using The Transport Layer API.** Disponível em: <https://docs.unity3d.com/Manual/UNetUsingTransport.html>. Acesso em 09 Dez. 2019.

VIEIRA, D. **Sem atualizações da KOG, Grand Chase terá atividades encerradas no Brasil.** Tecmundo. 2015. Disponível em: <https://www.tecmundo.com.br/video-game-e-jogos/72707-atualizacoes-kog-grand-chase-tera-atividades-encerradas-brasil.htm>. Acesso em: 30 Out. 2019

Warner Bros. **Ready Player One.** Disponível em: <https://www.warnerbros.com/movies/ready-player-one/>. Acesso em: 30 Out. 2019

WARREN, G; Olprod. **Working with shaders.** 2016. Disponível em: <https://docs.microsoft.com/pt-br/visualstudio/designers/working-with-shaders?view=vs-2015>. Acesso em: 24 Abr. 2019.

WIRDUM, A. v.; **The Genesis Files: How David Chaum's eCash Spawned a Cypherpunk Dream.** 2018. Disponível em: <https://bitcoinmagazine.com/articles/genesis-files-how-david-chaums-ecash-spawned-cypherpunk-dream>. Acesso em: 07 Abr. 2019.

WIRDUM, A. V.; **The Genesis Files: Hashcash or How Adam Back Designed bitcoin`s Motor Block.** 2018. Disponível em: <https://bitcoinmagazine.com/articles/genesis-files-hashcash-or-how-adam-back-designed-bitcoins-motor-block> Acesso em: 07 Abr. 2019.

WIRDUM, A. v.; **The Genesis Files: If bitcoin Had a First Draft, Wei Dai`s B-Money Was It.** 2018. Disponível em: <https://bitcoinmagazine.com/articles/genesis-files-if-bitcoin-had-first-draft-wei-dais-b-money-was-it> Acesso em: 07 Abr. 2019.

WIRDUM, A. v.; **The Genesis Files: With Bit Gold, Szabo Was Inches Away From Inventing bitcoin.** 2018. Disponível em: <https://bitcoinmagazine.com/articles/genesis-files-bit-gold-szabo-was-inches-away-inventing-bitcoin> Acesso em: 07 Abr. 2019.

Xaya Blockchain Gaming. [2018?] Disponível em: <https://xaya.io/> Acesso em: 28 Out. 2019.

Xaya Blockchain Gaming. **The Emergence of the human mining genre.** [2018?] Disponível em: <https://xaya.io/the-emergence-of-the-human-mining-genre/> Acesso em: 28 Out. 2019.

Xaya Blockchain Gaming. **What is Human Mining?.** 2017. Disponível em: <https://medium.com/@XAYA/what-is-human-mining-3d08fbd6dfd2>. Acesso em: 28 Out. 2019.

# 7. ANEXOS

**ANEXO A - ATAQUE SYBIL**
**https://www.microsoft.com/en-us/research/publication/the-sybil-attack/**

**ANEXO B - ATAQUE SYBIL**
**https://www.binance.vision/pt/security/sybil-attacks-explained**

**CÓDIGO FONTE DA APLICAÇÃO**
**https://github.com/Andre220/TCCBlockchainDomino**