Blockchain como servidor de um jogo online: uma análise com proof of stake

Resumo

Blockchains e jogos são dois mercados que vem ganhando mais espaço a cada ano. Desde a sua criação, em 2009, a Blockchain se mostrou uma tecnologia disruptiva no setor de pagamentos e moedas virtuais, com o Bitcoin. É um sistema totalmente distribuído e seguro, sem a necessidade de entidades para validação de transações. A sua estrutura de *hash pointer*, em conjunto com o sistema de consenso *proof-of-work*, força os computadores que compõem o sistema a realizarem uma prova computacional muito custosa, tornando os dados salvos virtualmente imutáveis. Este trabalho, sugere uma abordagem diferente para esta tecnologia revolucionária, utilizando esta, como um servidor de jogo *multiplayer* online. Cada jogador se torna um servidor nesta rede e assim elimina a necessidade de servidores centrais controladas por empresas privadas. Jogos criados neste tipo de servidor se mantém ativos com ao menos um jogador conectado. Nesta implementação, foi utilizado um protocolo de consenso que vêm ganhando mais notoriedade atualmente, o *proof-of-stake*. Este, usa um valor variável de dificuldade de bloco a depender da pontuação total do mesmo, eliminando o alto gasto computacional do *proof-of-work* e atribuindo maior segurança, considerando a dedicação dos nós ao sistema no momento da escolha do validador. Foi desenvolvido também um protocolo de comunicação específico para a aplicação, a qual foi desenvolvida em Golang e um jogo teste em Unity.

Introdução

O desenvolvimento da internet tem aumentado a sua extensão e capilaridade de alcance, o qual tem possibilitado que as pessoas possam interagir mesmo estando separados por grandes distâncias. Além de serviços, a internet tem sido a base para o desenvolvimento de jogos eletrônicos, onde os jogadores competem podendo estar geograficamente distantes entre si. Segundo a 19ª Pesquisa Global de Entretenimento e Mídia, da pwc, o mercado de jogos deve crescer cerca de 5,3% até 2022, atingindo 1,5 bilhão de dólares no país, 13º colocação global e líder latino-americano no setor. O grande crescimento deste mercado também se deve pelos avanços tecnológicos de redes e computadores, crescimento de vendas de smartphones e tablets e a popularização dos jogos criados ou portados para estes dispositivos.

Tais jogos online, em especial os chamados jogos *multiplayer*, no qual vários jogadores estão atuando simultaneamente), possuem em geral um ou mais servidores que centralizam as informações de uma região em que o serviço do jogo é oferecido e onde os jogadores se conectam para poder jogarem simultaneamente. Para atingirem o objetivo de colocar todos os jogadores em um mesmo ambiente ou partida (dependendo do tipo de jogo) estes possuem, predominantemente, arquiteturas cliente-servidor, onde os dados em comum ficam centralizados e este controla o andamento do jogo [Peer-to-Peer Architectures for]. Estes servidores centralizados são controlados, em sua maior parte, pela empresa dona do jogo e se este vier por algum motivo a ficar offline, podendo ser por decisão de encerramento do serviço pela empresa, falhas, ou até mesmo ataques por hackers, como o DOS (Denial of Service) [22], o qual é bem comum, os jogadores ficarão impossibilitados de jogar.

Este trabalho propõe um jogo sem a necessidade de um servidor central. Dessa maneira, os próprios usuários do jogo assumem a função de servidor e o jogo sempre funcionará independente de alguma instituição, desde que exista pelo menos um jogador ativo. Para construir uma aplicação que sem o uso de um servidor central, usando uma arquitetura peer-to-peer, o trabalho descrito em [Peer-to-Peer Architectures for] aborda um conjunto de desafios de projeto como, problemas em escalabilidade, atraso da rede, segurança dos dados, entre outros.

Em 2008, foi proposto em [4], sob pseudônimo de Satori Nakamoto, a criptomoeda Bitcoin, que usa uma estrutura de dados voltada para um sistema peer-to-peer totalmente distribuído, oferecendo segurança e transparência, a qual é chamada blockchain. Os dados gravados em uma blockchain são organizados em blocos de transações e ligados entre si por uma cadeia de hashes, no qual cada bloco possui o valor hash do bloco anterior à ele.

Manter os dados consistentes, ou seja iguais entre as suas réplicas, em sistemas distribuídos é um grande desafio, se não o maior, devido a distância e processos paralalelos que podem existir entre as entidades que o compõem (Tanenbaum). Para um sistema que necessita que todos os usuários visualizem o mesmo estado, como é o caso de um jogo multiplayer, ele é crucial [Peer-to-Peer Architectures for]. Dessa forma uma blockchain mostra-se uma alternativa bastante interessante, pois seu método de consenso é um sistema que também age como mecanismo de segurança, mantendo todos os dados nos nós consistentes com uma cadeia de dados computacional em que um dado gravado não pode ser alterado sem que se refaça todo o trabalho [4], o chamado proof-of-work (PoW). Embora ofereça segurança à integridade de dados, o PoW tem um alto custo computacional.

Existem alternativas ao uso de PoW que visam, principalmente, reduzir o custo computacional que este impõe ao seus utilizadores. A mais utilizada pelo mercado e que será utilizada neste trabalho é o proof-of-stake (PoS). Como descrito, a Blockchain foi inicialmente desenvolvida para o funcionamento do Bitcoin. Desde o surgimento do Bitcoin, tem sido desenvolvidas outras propostas semelhantes ao bitcoin e à blockchain, onde uma das mudanças mais bem sucedidas é a mudança deste sistema de consenso para o PoS [3], o qual está sendo implementado para substituir o PoW no principal concorrente do Blockchain, o Ethereum [referência sobre Ethereum alterar o PoW]. Uma vantagem do PoS sobre o PoW é uma redução significativa do custo computacional por não haver uma busca exaustiva pelo hash ideal, como ocorre no PoW, mas um desempate por um sistema de pontuação baseado na quantidade em criptomoedas e tempo que o validador as possuí (*coin age*) [3].

Este sistema sofre uma pequena mudança no contexto do jogo, uma vez que não tratamos de uma implementação de criptomoeda comum, mas com o valor do *coin age* sendo calculado a partir da pontuação (dinheiro) do jogador, aqui chamado de Ether. Portanto aqueles que têm mais pontos, tem por consequência mais tempo de jogo e provavelmente tem uma dedicação maior, o que os torna mais confiáveis na hora de gerar novos blocos para rede, uma vez que estes seriam os mais prejudicados em um eventual ataque [2]. PoS traz segurança para o jogo, fazendo com que os jogadores (nós) mais ricos e por conseguintes mais confiáveis, gerem mais blocos, mas ainda fazendo com que todos possam contribuir com o sistema, pois o *coin age* do nó é consumido a cada validação de bloco.

Este trabalho aborda todos estes temas citados, com a implementação de um jogo teste e todo o servidor em Blockchain, utilizando o sistema de *PoW*. Incluindo também um protocolo de comunicação feito especificamente para esta aplicação.

-Porque trabalhar com estes temas

A ideia de retirar o controle dos dados de um servidor central, fazendo um sistema totalmente distribuído em que não existisse a figura de uma empresa específica administrando e assim poderia ser administrado e hosteado por todos aqueles que o jogam.

-Objetivos a serem alcançados

Geral:

-Criar um servidor baseado em conceitos de *blockchain* e utilizá-lo como aplicação de servidor de jogo *multiplayer* online, utilizando o sistema de consenso *proof of stake*.

Específicos:

-Implementar uma *blockchain* com sistema de consenso *proof of stake* (PoS) para ser utilizada como servidor.

-Implementar um jogo *multiplayer* online (MMO) de estratégia.

-Analisar a viabilidade da solução em termos de segurança dos dados.

-Metodologia

Revisão bibliográfica da *blockchain* e do sistema de *proof of stake*.

Implementação da *blockchain* utilizando Golang.

Implementação do jogo utilizando Unity.

Será feita primeiramente uma revisão bibliográfica sobre o sistema de blockchain, surgimento, problema que foi solucionado com sua existência e aplicações diferentes que surgiram posteriormente.

A implementação da blockchain será feita utilizando a linguagem de programação Golang, desenvolvida pela Google e que já conta com paralelismo nativo, bibliotecas padrão de server web e criptografia (SHA256), o que facilita o desenvolvimento da rede e demais componentes que formam a blockchain [https://golang.org/doc/]. Possui também a vantagem de integrar códigos em linguagens C e C++, em que o jogo será implementado[<https://golang.org/cmd/cgo/>]. As trocas de mensagem serão feitas com a implementação de um protocolo de comunicação dos processos que por sua vez utilizará sockets TCP, que estará explicitado nesse trabalho, feito para rodar sobre o TCP e especificamente para esta aplicação, com intuito de tornar todo o funcionamento da rede mais simples.

O jogo será desenvolvido na ferramenta Unity, que utiliza a linguagem C# como padrão para os *scripts* e programação orientada a componentes.

== LER A PARTIR DAQUI

Fundamentação teórica

Este capítulo abordará, de forma resumida, os principais conceitos necessários para o entendimento deste trabalho. Com explicações sobre o funcionamento da *blockchain* e demais tecnologias envolvidas. Primeiramente será feita uma rápida introdução ao Bitcoin que deu luz a esta nova tecnologia no mercado.

Sistemas Distribuídos

“*Um sistema distribuído é um conjunto de computadores independentes que se apresenta a seus usuários como um sistema único e coerente*”. (Tanenbaum)

Seguindo essa definição, um sistema distribuído é composto de componentes, computadores, autônomos. Essa separação do sistema não é percebida por aqueles que o usarem, funcionam como uma entidade única aos olhos do público. Portanto a grande dificuldade a ser vencida em um sistema distribuído é fazer todos os seus componentes, espalhados fisicamente, se comunicarem de forma eficiente e colaborarem uns com os outros.

Um sistema distribuído, portanto, deve ter algumas características principais (Tanenbaum):

* Fácil acesso aos recursos oferecidos.
* Transparência: ocultar razoavelmente bem dos utilizadores a sua distribuição pela rede.
* Escalabilidade: o sistema deve poder ser expandido.
* Abertura: os serviços oferecidos pelo sistema devem ter sua sintaxe e semântica padronizadas e bem definidas.

Um grande problema enfrentado ao se utilizar um sistema distribuído é manter os dados que estão em cada computador que compõe o sistema igual, a chamada consistência de dados. Em um sistema distribuído, ao exemplo de um servidor de jogo, atualizações podem ocorrer de forma concorrente e com isso gerar mudanças que conflitem entre si, resultando em um estado inconsistente dos dados.

Para poder lidar com essas inconsistências, provenientes de execuções de atualizações paralelas, os sistemas distribuídos utilizam mecanismos de consistência, os quais geralmente garantem que todas as cópias irão executar as atualizações na mesma ordem e com isso, chegarem ao mesmo estado final dos dados. Infelizmente, apenas executar na mesma ordem não garante o estado consistente. Devido a problemas de rede, mensagens podem chegar em ordem diferente da que foi enviada ao sistema. As mensagens que eram causalmente dependentes, necessitavam de uma mensagem anterior para terem sua execução corretas, tornem o estado dos dados inconsistente. Outros tipos de inconsistência tem a ver com a perda de mensagens. Uma forma de evitar estes problemas é o uso do protocolo confiável TCP, ou como em jogos que precisam de uma latência mais baixa, UDP, apesar deste último não garantir a entrega das mensagens [Peer-to-Peer Architectures for] (Tanenbaum).

Sistemas distribuídos estão por toda a parte, em servidores, bancos de dados, sistemas financeiros, jogos online, redes sociais e em tantas outras aplicações. Veremos um pouco mais sobre alguns conceitos e usos úteis sobre o tema para este trabalho nas seções abaixo.

Protocolo de Comunicação

Dados são informações apresentadas de uma forma qualquer, acordada entre as partes que irão criar e utilizar estes dados. Portanto comunicação de dados se trata da troca de informações, em formato de dados, entre dois dispositivos. A forma como essa transmissão é feita dependerá do sistema de comunicação que o dispositivo faz parte. Todo sistema de comunicação é composto por hardware (físico) e software (programas). Cinco componentes formam um sistema de comunicação de dados, são eles: mensagem, emissor, receptor, meio de transmissão e protocolo. A mensagem se trata dos dados que estão sendo transmitidos. Emissor e Receptor, são os dispositivos de envio e recebimento da mensagem, respectivamente. O meio de transmissão se trata do caminho físico pelo qual a mensagem irá ser transportada até o receptor, podendo ser cabos, de rádio, entre outras. E os protocolos, garantem que os dispositivos possam se comunicar e se entenderem, são um padrão de comunicação que veremos mais a seguir. (FOROUZAN, 2010)

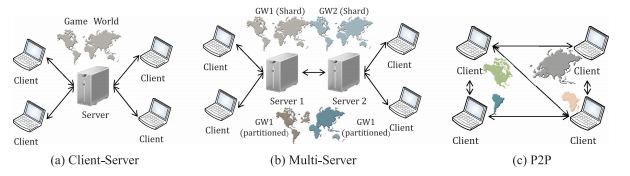
Protocolo também pode ser sinônimo de regra. Mais especificamente é uma convenção, acordos, que permite uma conexão, comunicação e transferência de dados entre computadores ou sistemas (FOROUZAN, 2010).

A comunicação em redes de computadores ocorre entre entidades em sistemas diferentes. Essas entidades podem ser qualquer coisa capaz de realizar uma comunicação com troca de informações, enviar e receber dados. Porém, para que haja de fato a comunicação entre essas entidades, é necessário o estabelecimento de um protocolo, que é acordado entre as partes, pois sem o qual as mensagens transmitidas não poderiam ser compreendidas. O protocolo é formado por um conjunto de regras que servem para determinar e controlar a comunicação de dados, definindo o que é comunicado, como é e quando deve ser feito. Os elementos-chaves que compõem um protocolo são sintaxe, semântica e temporização (*timing*) e serão explicados abaixo (FOROUZAN, 2010):

* Sintaxe: Se refere à estrutura ou o formato dos dados transmitidos, a ordem em que estes são apresentados. Por exemplo, dedicar os 8 primeiros bits a serem o endereço do emissor e os próximos 8 do receptor, seguido da mensagem.
* Semântica: Esta tem relação ao significado de cada parte dos dados, definindo como algo deve ser interpretado e a ação a ser tomada diante daquela informação. Por exemplo, um campo de endereço pode identificar tanto a rota, quanto o destino final, a depender da semântica definida.
* Temporização (*timing*): Tem a ver com o sincronismo entre as partes, quando os dados devem ser enviados e com que velocidade podem ser transmitidos. Por exemplo, um sistema pode ter um emissor muito mais rápido que o receptor, se os dados forem produzidos e enviados a uma taxa maior do que o receptor pode suportar, este será sobrecarregado e parte da informação pode ser perdida.

Um exemplo de protocolo amplamente utilizado para comunicação confiável é o TCP (*Transmission Control Protocol*), um protocolo complexo de camada de transporte. Este é orientado a fluxo de dados, usando número de portas e criando uma conexão virtual entre dois processos TCP para a transmissão dos dados (orientado a conexão). Também implementado um mecanismo para controle do fluxo e erros provenientes da camada de transporte (FOROUZAN, 2010).

Arquitetura de servidores



Arquiteturas possíveis [Peer-to-Peer Architectures for]

Existem diversas arquiteturas possíveis um servidor, os principais tipos são os mostrados na figura (acima): *Client-Server* (cliente-servidor), *Multi-Server* e *peer-to-peer* (P2P) [Peer-to-Peer Architectures for].

Em um contexto de servidor de jogo, o estado atual do jogo (*game state*) pode ser a informação mais valiosa, principalmente naquele em que o estado do mundo (*game world*) é persistente, ou seja não é reinicializado a cada uso. Manter o controle desse mundo é bem mais simples utilizando arquiteturas distribuídas, como o *client-server*, na verdade toda a programação e gerenciamento de consistência em arquiteturas baseadas em servidores (*cliente-servidor*, *multi-servidor*) é mais fácil do que as P2P, sendo a principal razão da sua maior popularidade. Arquiteturas distribuídas são utilizadas, principalmente, pela sua escalabilidade, exatamente o ponto negativo de uma arquitetura de servidor único, os quais não são capazes de lidar com milhares de players simultâneos [Peer-to-Peer Architectures for].

Cliente - Servidor

Pensar em sistemas em termos de cliente-servidor ajuda a entender e gerenciar toda a complexidade de um sistema distribuído. O básico deste modelo consiste em dividir os processos em dois grupos principais, clientes e servidores. Servidores são processos que implementam serviços e lidam com requisições de serviços de clientes, por exemplo um serviço de banco de dados. Clientes são processos que fazem as requisições, enviando uma mensagem, para utilizarem esses serviços prestados pelos servidores. Essa interação também é conhecida como comportamento de requisição-resposta e é mostrado na figura Z (Tanenbaum).

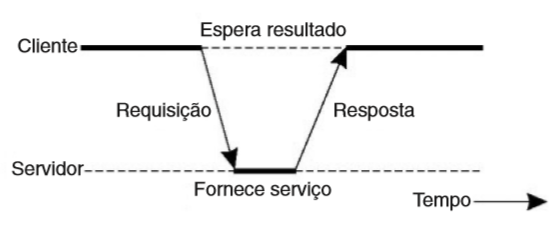


Figura Z

É possível implementar uma comunicação entre os processos clientes e os processos servidores de forma mais simples, sem conexão, quando em uma rede confiável. A mensagem de requisição, que deve descrever o serviço requisitado e conter os dados de entrada, é simplesmente empacotada e enviada ao servidor. O servidor recebe a mensagem, desempacota, identifica o serviço requerido, executa com os comandos recebidos e empacota os resultados da execução dos mesmos e os envia ao cliente. Apesar do ganho de eficiência em não utilizar uma conexão, há grandes riscos de perda ou corrompimento das mensagens. Para evitar esses problemas, muitos sistemas cliente-servidor utilizam um protocolo confiável e orientado a conexão, como o TCP/IP. Esta decisão adiciona uma nova etapa a todo o processo, pois sempre que um cliente quiser enviar uma mensagem ele primeiro terá de iniciar uma conexão com o servidor e só aí poderá enviar a requisição. O servidor utilizará a mesma conexão para enviar a mensagem de resposta e encerrará a conexão. O único problema é o custo em se estabelecer e encerrar conexões, especialmente quando as requisições são pequenas (Tanenbaum).

Em servidores de jogo, as arquiteturas deste tipo normalmente guardam no servidor cópias de todos os objetos que são mutáveis e as informações do *game world*. Clientes se conectam para receberem as informações necessárias sobre o estado do mundo e todas as atualizações sobre os players são enviadas para o servidor para serem executadas e terem seus possíveis conflitos resolvidos. O server fica responsável por enviar atualizações de objetos a todos os players interessados. Como já dito anteriormente, o grande problema desta arquitetura é o limite no número de players suportados, soluções que adicionam mais de um servidor, apesar de aumentar a complexidade, melhoram a escalabilidade [Peer-to-Peer Architectures for].

Arquitetura Peer-To-Peer

Enquanto na arquitetura cliente-servidor havia uma diferenciação entre as funções de cada processo que constituía o sistema, processos que fazem parte de um sistema peer-to-peer (P2P) são iguais, olhando de uma perspectiva alto nível. Essa igualdade torna todas as interações entre eles simétricas, agindo como um cliente e um servidor ao mesmo tempo.

Uma questão importante deve ser resolvida em arquiteturas *peer-to-peer*, como organizar a chamada rede de sobreposição (*overlay*). Essas redes de sobreposição é uma rede lógica, na qual os nós são formados pelos processos e os enlaces pelos canais de comunicação, normalmente usando TCP. Dessa forma os processos devem sempre enviar mensagens por este canal de comunicação existente, não se comunicando diretamente uns com os outros.

Os sistemas *peer-to-peer* podem ser classificados em dois tipos, quanto a sua rede de sobreposição: estruturadas, aquelas em que a rede de sobreposição é construídas com base em um procedimento determinístico, e não estruturadas. No caso dos não estruturados, os nós mantêm uma lista de n vizinhos (conhecido como visão parcial), que foi escolhida de forma aleatória dentro dos nós vigentes, e façam conexão com alguns, ou todos eles dependendo da implementação. A ideia é construir a rede de forma que esta fique parecida com um grafo aleatório e que não deixem nós desconectados, ou seja grupos de nós isolados que não alcançaram outros nós da rede. Uma boa forma de evitar essa desconexão é deixar os nós trocarem entradas de suas visões parciais entre si (Tanenbaum, 2ª edição).

As fornas de disseminação de atualizações em servidores *peer-to-peer* são diversas, abaixo segundo [peer-to-peer], alguns dos mecanismos mais utilizados:

* Comunicação direta ou *broadcast*: uma cópia da atualização é enviada para todos os nós conectados. Esta forma mais simples possui a vantagem de ter uma latência baixa, ou seja chega aos nós conectados rapidamente, porém possui um grande custo de rede, por realizar muito tráfego de mensagens.
* Árvores de *multicast*: árvores lógicas feitas de conexões de nós são os *peers* do sistema, e com isso uma mensagem é enviada para a raiz desta árvore, que por sua vez repassa a mensagem para seus filhos conectados. A grande vantagem é o pequeno custo de rede, uma vez que as árvores normalmente possuem alguns poucos filhos.
* NAT e *Firewalls*: neste caso, clientes estão conectados a redes que estão por trás de NATs (do inglês *Network Address Translation*) e *firewalls*, que protegem as conexões da rede interna. Isto dificulta as mensagens a adentrarem os nós internos destas redes, necessitando de modificações em protocolos para lidarem com essas limitações.

*Distributed Ledger Technology* - DLT

Tecnologia de *ledger* distribuído ou a sigla em inglês DLT (*Distributed Ledger Technology* é definido como um banco de dados distribuído, compartilhado e encriptado, o qual serve como um repositório de informações irreversível e incorruptível [10].

O uso desta tecnologia permite aos usuários armazenarem e acessarem informações em um banco de dados compartilhado, que opera sem existir uma figura central de um sistema validador, ou seja ele é mantido por todos os participantes (nós) da rede distribuída que o compõe. As DLTs são famosas por normalmente utilizarem criptografia como forma de validação das transações e armazenamento de ativos [10].

Apesar de existirem várias aplicações para as DLTs, a mais importante é a aplicação na área financeira, pois essa tecnologia permite aos usuários acesso direto aos banco de dados compartilhados, podendo liquidar seus valores imobiliários e realizar transferências de dinheiro sem necessitarem de um intermediário. Como toda a informação presente nestes bancos é compartilhada por todos os usuários da rede, as transações realizadas podem ser liquidadas quase instantaneamente, a depender do sistema utilizado. As DLTs tem o potencial de fazer os sistemas de pagamento serem independentes de figuras centrais como bancos e totalmente distribuídos [10].

Uma solução utilizando DLTs como sistemas de pagamento, são as chamadas criptomoedas, as quais pertencem a um subconjunto de moedas virtuais. Moedas virtuais são geralmente entendidas como uma representação virtual de valor que possuem características de moeda. Portanto uma criptomoeda pode ser entendida como uma moeda virtual que utiliza protocolos *peer-to-peer* e criptografia como forma de validação das transferências de valor, uma DLT. O Bitcoin, a criptomoeda mais famosa, utiliza um tipo de *ledger* distribuído chamado Blockchain para a validação e armazenamento de suas transações sem a necessidade de um intermediário [10] [4].

Blockchain

Blockchain é uma tecnologia de servidor de dados totalmente distribuído, um *ledger* distribuído, proposto inicialmente por um autor de pseudônimo Satoshi Nakamoto no whitepaper do Bitcoin em 2008 [4][2][10]. Utiliza o conceito de uma rede peer-to-peer onde cada nó que compõe esta rede também atua como o próprio servidor de dados, mantendo uma réplica completa do banco [2].

Os dados gravados em uma blockchain são separados em blocos de transações em ordem cronológica e ligados entre si por uma “corrente” de *hashes*, no qual cada bloco possui o valor *hash* do bloco anterior à ele [6][10]. Depois de adicionado, um bloco jamais pode ser deletado e uma vez que cada nó que compõe a rede guarda uma cópia da Blockchain, suas transações podem ser vistas por todos. Este registro permanente pode ser utilizado por qualquer computador da rede para coordenar suas ações ou verificar eventos [11].

O mecanismo de consenso mais utilizado numa blockchain também age como mecanismo de segurança, mantendo todos os dados nos nós consistentes com uma espécie de puzzle computacional em que um dado gravado não pode ser alterado sem que se refaça todo o trabalho [4], o chamado proof-of-work. Neste sistema um valor de *hash* ideal deve ser encontrado para que um bloco possa ser inserido na corrente de blocos que forma a blockchain. Tal valor só consegue ser encontrado por meio de *brute force* variando um *nonce* até se obter um valor que seja mais baixo que o determinado para aquele bloco [5]. Um *broadcast* para a rede é feito pelo nó que encontrar o valor de *hash* primeiramente e todos os nós devem aceitar a cadeia de blocos válidos mais longa como sendo o estado consistente do banco de dados [6].

É utilizado principalmente pelas criptomoedas, como armazenamento seguro das transações envolvendo essas moedas virtuais, como assim foi idealizada para ser por Satoshi no *whitepaper* do Bitcoin [4]. Porém hoje podemos observar que a tecnologia da Blockchain vai muito além das criptomoedas, alcançando aplicações que podem mudar significativamente a forma de lidar com mercados financeiros, inteligência artificial, computadores e toda a tecnologia em geral [10].

Veremos nos tópicos a seguir um pouco sobre a origem dessa tecnologia no Bitcoin e os principais conceitos que fazem parte dos sistemas de Blockchain mais aprofundadamente.

Bitcoin

“*A purely peer-to-peer version of electronic cash would allow online payments to be sent directly from one party to another without going through a financial institution*”, ou seja, uma moeda virtual descentralizada, que permitirá pagamentos serem feitos sem a necessidade de uma instituição financeira. Foi assim que Satoshi Nakamoto (2008), um pseudônimo cujo verdadeiro autor permanece um mistério, se referiu a sua criação no *whitepaper* do Bitcoin (NAKAMOTO, 2008).

“The trust machine ... technology behind Bitcoin lets people who do not know or trust each other build a dependable ledger. This has implications far beyond the crypto-currency ... could transform how the economy works.”

The Economist, Oct 2015

As moedas virtuais que precederam o Bitcoin ainda utilizavam alguma figura central (geralmente um banco), ou seja, não eram completamente distribuídas. Esse sistema pode ser considerado um *ledger* (livro-razão) distribuído, que reflete todas as transações e seus envolvidos, tornando todos no sistema o próprio banco. No Bitcoin é o blockchain que assume essa função, permitindo a ocorrência de diversas transações sem a necessidade de um mediador de segurança [2][4][13].

O objetivo da nova tecnologia da Bitcoin foi conseguir, através da Blockchain, resolver um dos principais problemas dos métodos de pagamento digital e que mantinham a necessidade de uma instituição financeira centralizadora, o gasto duplo (DOUBLE SPENDING). Este pode ser definido como a utilização de uma mesma moeda para realizar dois, ou mais, pagamentos distintos. [8][9]

O modelo bancário resolve este problema utilizando números de série controlados pela instituição e proibindo o processamento de transações concorrentes.[7] O Bitcoin consegue este feito com toda uma rede distribuída, cada usuário está ciente das transações que ocorreram, mesmo se concorrentes, podendo verificar sua legitimidade, de forma que os gastos duplos são percebidos pelos participantes e evitados. Uma transação nessa rede só é aceita se a maioria dos participantes concordar com a sua inserção no ledger [6][7]. Este sistema de quórum garante ao Bitcoin que mesmo que existam informações incorretas e entidades maliciosas na rede, enquanto a maioria dos participantes for honesta o estado dos dados continuará correto. Não necessitando que os nós da rede confiem totalmente uns nos outros, podendo ser uma rede pública, como é o caso do Bitcoin. [12][7]

O funcionamento da Blockchain do Bitcoin ocorre sobre uma rede *peer-to-peer* não estruturada, baseada em conexões persistentes TCP, onde as transações são gravadas em uma cadeia contínua de *proof-of-work* [2][4].

Criptografia

“Criptografia, palavra de origem grega, significa “escrita secreta”. Entretanto, usamos o termo para nos referirmos à ciência e à arte de transformar mensagens de modo a torná-las seguras e imunes a ataques”.(FOROUZAN, 2010)

Protocolos e algoritmos de criptografia possuem uma ampla área de aplicações, principalmente quando se trata de segurança de rede e internet. Estes podem ser agrupados em: Algoritmos de encriptação, que podem ser simétricos ou assimétricos, este último amplamente utilizado na Blockchain no sistema de chaves públicas e privadas. Algoritmos de integridade de dados, para proteger blocos de dados de alterações, como os *hash pointers* (Narayanan et al, 2015) na Blockchain. E protocolos de autenticação, para verificar identidades de entidades. (STALLINGS, 2015)

Segurança dos dados

A segurança dos dados se tornou uma grande preocupação com a evolução dos computadores e da comunicação entre estes. Existindo por tanto, segundo (STALLINGS, 2015), objetivos principais na segurança dos computadores, conhecido como tríade CIA (do inglês *confidentiality, integrity and availability*):

* Confidencialidade, que assegura que as informações só sejam reveladas para aqueles indivíduos autorizados.
* Integridade, que pode ser de dados ou do sistema, de dados assegura que informações serão sempre modificadas de uma maneira específica e previamente autorizada, de sistema que assegura que um sistema não sofra manipulações durante a execução de seus serviços.
* Disponibilidade que assegura o acesso rápido e confiável à informação.
* Autenticidade, tendo a ver com a capacidade de poder ser verificada e portanto considerado genuíno. Verificando por exemplo se um usuário é realmente quem diz ser e se as entradas de um sistema vem de fontes confiáveis.
* Responsabilização, trata-se de atribuir ações de uma entidade exclusivamente à ela, podendo com isso atribuir violações de segurança a uma parte responsável.

Criptografia é de extrema importância na segurança dos dados, sendo utilizada para garantir principalmente, as propriedades confidencialidade e autenticidade. A encriptação mensagens transmitidas evita que estas sofram ataques, como os chamados ataques passivos (STALLINGS, 2015), onde informações sigilosas podem ser “bisbilhotadas” por indivíduos não autorizados, garantindo a confidencialidade na transmissão de informações. Além existirem métodos para garantir a legitimidade de documentos, pessoas, etc.

Tipos de encriptação

O aspecto de maior importância na segurança da informação hoje é a criptografia (STALLINGS, 2015).

Existem dois tipos de encriptação, simétrico e assimétrico. E a diferença entre estes está nas chamadas chaves secretas, utilizadas pelos algoritmos de encriptação para realizar o embaralhamento do texto. Quando as chaves do emissor e receptor são as mesmas, este é considerado um sistema de encriptação simétrica. Se utilizam chaves diferentes, é considerado um sistema de encriptação assimétrica.

Simétrica

A encriptação convencional, também chamada de simétrica ou de chave única é o método que continua sendo de longe o mais utilizado. Este, assim como todos os algoritmos de encriptação, é baseado na substituição, mapeando os elementos da mensagem em outro elemento, porém sem ocorrer perda de informação neste processo. O mapeamento é baseado chave secreta, produzindo saídas diferentes dependendo da chave utilizada. Para resgatar uma informação encriptada, utiliza-se os algoritmos de decriptação, que é o algoritmo inverso da encriptação, utilizando a chave secreta para retornar a informação para seu estado original (STALLINGS, 2015). Neste tipo de encriptação a chave secreta é compartilhada pelo emissor e receptor da mensagem, pois esta deve ser a mesma para ambos. Os elementos que fazem parte de um modelo de criptografia simétrica são:

* Texto claro: que é a mensagem ou os dados sem alterações.
* Chave secreta: utilizada para fazer a encriptação e decriptação do texto claro. Todo o processo de substituições e transformações é realizado baseando-se nela. Qualquer mudança na chave secreta terá impacto na saída do algoritmo.
* Texto cifrado: é o texto claro após o embaralhamento feito pelo algoritmo, a saída do processo de encriptação.
* Algoritmo de decriptação: algoritmo inverso à encriptação, utiliza o texto cifrado e a chave secreta para retornar o texto claro.



FIg (FOROUZAN, 2010)

Assimétrica

Na criptografia de chave pública ou assimétrica, permanecem praticamente os mesmos elementos do modelo assimétrico, com a diferença de que não haverá apenas uma chave secreta. Neste modelo a chave secreta é substituída por duas chaves distintas, a chave privada, que é guardada pelo receptor da mensagem e a chave pública, a qual pode ser exposta ao público em geral. Na figura 30.4 abaixo, temos um exemplo do funcionamento desse mecanismo. Para Alice enviar uma mensagem a Bob, esta usa a chave do endereçado para criptografar a mensagem antes do envio. Assim que esta mensagem chegar para Bob, se tudo estiver certo e nada foi alterado ou corrompido durante a transmissão, ele conseguirá descriptografar a mensagem utilizando a sua chave pública. (FOROUZAN, 2010)



FIgura 30.4

Algoritmos de encriptação assimétrica, portanto, possuem a seguintes característica principais (STALLINGS, 2015):

* Dada apenas a sua chave pública e o conhecimento do algoritmo de encriptação, é impossível, ou pelo menos impraticável, determinar qual a chave privada associada.
* Nos algoritmos mais utilizados, como RSA, qualquer uma das chaves pode ser utilizada como chave pública ou privada. Independente de qual for escolhida para a encriptação, a outra irá decriptar.

Para a utilização destes algoritmos, devem ser seguidas as seguintes etapas (STALLINGS, 2015):

1. Serão gerados por cada usuário um par de chaves a ser usado na encriptação e decriptação das mensagens.
2. As chaves públicas são colocadas por cada usuário em um arquivo acessível aos demais. Não revelando a sua chave privada. Dessa forma cada usuário terá uma relação de chaves públicas para cada participante.
3. Para o envio de uma mensagem confidencial, esta será encriptada fazendo uso da chave pública do endereçado, que já é conhecida por estar na coleção.
4. Quando a mensagem for recebida, o receptor poderá usar sua chave privada para decriptar a mensagem. Caso esta seja interceptada estará protegida pois apenas o destinatário real tem o conhecimento da chave de decriptação (chave privada).

A grande vantagem de se utilizar esta técnica está no fato de todas as chaves terem sido geradas localmente, e apenas as chaves públicas precisam ser distribuídas. Isto resolve um grande problema da criptografia simétrica, que é a distribuição das chaves secretas entre as partes interessadas. Como a chave secreta nestes algoritmos é única e a mesma para as partes que irão trocar as mensagens, se a chave for exposta a indivíduos mal intencionados durante o processo de transferência, todo o sistema fica comprometido e as mensagens não estarão mais seguras. Outra vantagem é que a qualquer momento um sistema poderá gerar um novo par de chaves e substituir a sua chave pública antiga, garantindo ainda mais segurança.

Assinatura Digital

A assinatura digital se aproveita de uma característica de alguns algoritmos de encriptação pública para utilizar como forma de autenticação. Essa característica é o fato de se poder utilizar qualquer chave do par gerado para realizar a encriptação. Neste caso, dados dois sistemas A e B, A irá preparar uma mensagem e fará a encriptação utilizando a sua própria chave privada, enviando para B a seguir. Assim que a mensagem for recebida, B poderá resgatar a mensagem, decriptando com a chave pública, que é conhecida, de A. Como apenas A tem acesso à chave que faz par com a utilizada na decriptação por B, a recuperação da mensagem prova que A foi realmente o emissor da mensagem que chegou para B. Logo a mensagem encriptada inteira serve como uma assinatura digital e prova a autenticidade, em termos da origem dos dados (STALLINGS, 2015).

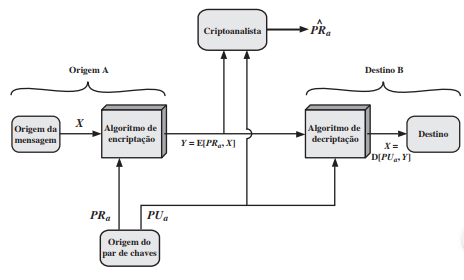


Figura 9.3

Falar sobre assinaturas digitais na blockchain

Hash

Um hash é uma transformação da informação original. Sua função(H), portanto, irá pegar uma mensagem M com um tamanho arbitrário e produzir um valor hash h, o qual h = H(M) [9]. Os resultados de uma função de hash considerada boa são valores distribuídos por igual, dentro do universo de saídas possíveis, e aparentemente de forma aleatória. A propriedade mais importante de um hash é que qualquer alteração de bit na entrada M, produzirá com uma alta chance, uma mudança também na saída h. Servindo assim para a verificação de integridade de dados. Pois caso haja alguma alteração na entrada, intencional ou não, mudará o hash produzido e assim por uma simples comparação entre eles pode-se verificar uma mudança ou corrupção nos dados (STALLINGS, 2015).

A Blockchain do Bitcoin usa um subconjunto das funções hash, chamadas de one-way (mão única) hash ou hash criptográfico, as quais são consideradas unidirecionais, apenas podendo ser revertidas a troco de um custo computacional verdadeiramente alto. As principais características do one-way hash são [9]:

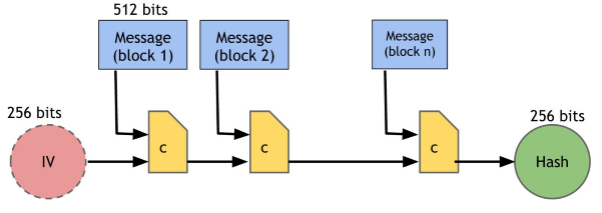
1. Dado um M, é fácil computar h;
2. Dado um h, é difícil computar M, onde H(M) = h;
3. Dado um M1, é difícil achar outra mensagem M2, a qual H(M1) = H(M2) (resistente a colisão).

Por causa dessas características únicas, funções de hash criptográfico são muito versáteis e possuem as mais diversas aplicações de segurança e protocolos da internet. As principais são (STALLINGS, 2015):

* **Autenticação de mensagem:** é um mecanismo usado para verificar a integridade de uma mensagem quando esta chega ao receptor, garantindo que os dados recebidos não foram alterados durante o transporte. Nesses casos o valor da função de hash é chamado de resumo da mensagem. Este resumo é calculado pelo emissor e enviado juntamente da mensagem. Chegando ao destinatário, este irá realizar o mesmo procedimento sobre a mensagem e verificar se o valor de hash é igual ao recebido. Caso haja diferenças, significa que a mensagem, ou o valor de hash, sofreu alteração.
* **Assinaturas digitais:** a grande diferença no procedimento com hash do apresentado no tópico anterior é que desta vez não é a mensagem que é utilizada na encriptação com a chave privada, mas sim o hash da mensagem. Enviando assim a mensagem juntamente a assinatura sobre o hash da mesma. Na decriptação com a chave pública, o valor obtido é o hash, que será comparado com o hash da mensagem recebida para garantir a autenticidade do emissor e a não alteração da mensagem. Outra vantagem é poder utilizar um sistema de criptografia simétrica sobre a mensagem mais o código hash encriptado com a chave privada, com uma chave secreta, se a confidencialidade for necessária.
* **Arquivo de senha de mão única:** o sistema armazena apenas o hash da senha de um usuário, ao em vez de a senha em si. Garantindo que caso ocorra alguma invasão no armazenamento dos dados por hackers, as senhas permaneçam secretas. Quando o usuário tentar acessar o sistema com sua senha, este fará o hash e simplesmente irá comparar com o valor de hash armazenado no momento do cadastro.

Entre outras utilizações como na detecção de intrusão, vírus, geradores de número pseudo aleatórios, etc.

O *Secure Hash Algorithm* (SHA) passou a ser a função de hash mais utilizada a partir de 2005, uma vez que as demais funções de hash mais utilizadas tiveram vulnerabilidades expostas. Esta foi desenvolvida pelo *National Institute of Standards and Technology* (NIST) e publicado no FIPS 180 (Federal Information Processing Standards Publication) em 1993, mas foram descobertos pontos fracos em seu algoritmo. Uma nova versão revisada foi lançada, chamada de SHA-1, a qual produz um hash de 160 bits de saída, além de novas versões em 2002, com tamanhos de hash maiores, 256, 384 e 512 bits. Esses novos algoritmos ficaram conhecidos como SHA-256, SHA-384 e SHA-512, respectivamente, sendo chamados em conjunto de SHA-2. Ganharam força principalmente após uma equipe conseguiu encontrar um mesmo hash, em duas mensagens distintas, em apenas 2^69 operações, menos do que as 2^80 previstos, no SHA-1 (STALLINGS, 2015).



(Narayan)

A figura apresenta o funcionamento do SHA-256, onde a mensagem inicial, que pode ser de qualquer tamanho, é dividida em blocos de 512 bits. Cada bloco é passado juntamente com a saída do bloco anterior a uma função de compressão até a saída final em um *hash* de 256 bits. No bloco inicial, como não há saída anterior, é utilizado o chamado vetor de inicialização no lugar, o qual é reutilizado sempre que a função de *hash* é chamado.

Hash Pointer

O hash pointer é uma estrutura de dados na qual um ponteiro para uma informação é guardado juntamente ao hash criptográfico, como um SHA-256 que é utilizado na Blockchain do Bitcoin, daquela mesma informação. Isto torna essa estrutura muito útil, pois um ponteiro comum permite recuperar uma informação apontada, um hash *pointer* vai além, ele permite também verificar se ocorreu alguma mudança na informação recuperado. *Blockchain*, nada mais é do que uma lista de blocos, que contém diversas informações, ligadas por hash *pointe*r*s*. O fato de cada um destes blocos possuírem um *hash pointer* para o bloco anterior, faz com que em uma estrutura de *blockchain*, cada bloco não apenas diz onde está o bloco anterior, como também contém toda a informação dele resumida em um hash, o chamado *digest*. Dessa forma é possível verificar facilmente se alguma informação em blocos anteriores foram alteradas. Pois, uma vez que ocorra alguma mudança nos dados de um determinado bloco k, surtirá uma mudança também no hash do bloco, fazendo com que o valor de hash do bloco k que está armazenado como um *hash pointer* no bloco k + 1, fiquem diferentes, quebrando a “corrente”.

A figura V ilustra um exemplo de lista de blocos com *hash pointers*, uma estrutura de *blockchain* básica.

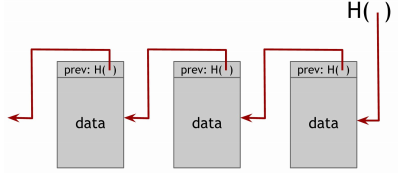


Figura V

Como cada bloco possui o *digest* da informação do bloco anterior, para um adversário alterar algum dado e manter todos os *hash pointers* corretos, este terá que recalcular todos os hashs dos blocos à frente do que continha a informação adulterada. Além disso se ele tentar adicionar um novo bloco em algum lugar que não seja o final da *blockchain*, será obrigado a recalcular todos os hashes dos blocos da estrutura, inclusive o primeiro bloco o qual recebe um nome especial, o chamado *genesis block*.

Transações e TimeStamp

As transações armazenadas em uma Blockchain são organizadas em ordem cronológica. Tanto sua ordem dentro do bloco quanto o bloco que foi inserida na Blockchain, garantem a sequência dos eventos ao longo do tempo. Marcados o instantes pelo campo do TimeStamp. Assim é possível identificar que determinadas transações ocorreram em um dado instante de tempo, identificando o tempo do bloco que está inserido e em uma determinada ordem, verificando a sua ordem de adição no bloco. A Blockchain pode ser então considerada um servidor de TimeStamp (figura N) [4].

Toda moeda, no exemplo do Bitcoin, é associada a um endereço, que é na verdade uma chave pública (*public key*) de criptografia assimétrica (no caso do Bitcoin o hash dela). A chave privada (*private key*), que faz o par com a pública, é utilizada para a criptografia das mensagens (assinatura digital) e dar validade às transações.[8]

Transações são o cerne da tecnologia da Blockchain no Bitcoin, elas especificam como o dinheiro transita de um endereço para outro, utilizando para isso, no caso clássico do Bitcoin, uma linguagem *script* que não é Turing completa. As transações escritas em Script são compostas de basicamente dois elementos principais, os *Inputs* (INs) e os *Outputs* (OUTs) e o *hash* da transação, como pode ser visto na figura B [2].



Figura B

Os IN definem a origem, que deve necessariamente ser de uma transação não gastas, chamadas de UTXOs (do inglês *Ununspentable Transactions*). E os OUTs definem para quais endereços e que quantia será transferida. Os valores de entrada dos endereços IN, que sempre será composto por todo o valor contido naquele endereço, são somados e distribuídos pelas saídas dos OUTs. Sua quantidade não deve ser menor do que a soma dos valores de saída, mas podem ser maiores. O valor excedente fica como *fee* (gorjeta) para o nó que validar o bloco como forma de incentivo para esta transação ser inserida em um bloco.

Os inputs são todos assinados pelo criador da transação, utilizando a chave privada que faz par com a chave pública de onde o input está sendo extraído, para garantir que o verdadeiro dono está transferindo a quantia. A transação é hashada antes de ser enviado para garantir a sua não alteração.

Smart contracts are transactions or contracts converted into code that facilitate, execute and enforce commercial agreements between two or more parties. As discussed in this article, Blockchain-based smart contracts have the potential to streamline financial transactions and operational and counterparty risk associated with monitoring or enforcing contractual obligations.[10]

Transações são o cerne da tecnologia da Blockchain no Bitcoin, elas especificam como o dinheiro transita de um endereço para outro, utilizando para isso, no caso clássico do Bitcoin, uma linguagem script que não é Turing completa. As transações escritas em Script são compostas de basicamente dois elementos principais, os *Input,* que especificam os endereços e valores de entrada e os *Outputs*, que fazem o mesmo mas para os destinos, o *hash* da transação e as assinaturas digitais dos I*nputs*, para provarem sua autenticidade quanto à posse da quantia. Valores de entrada são somados e subtraídos pelas saídas. Sua quantidade não deve ser menor do que a soma dos valores de saída, mas podem ser maiores. O valor excedente fica como fee (gorjeta) para o nó que validar o bloco como forma de incentivo para esta transação ser inserida em um bloco.​​​​​​​ (Florian Tschorsch;Björn Scheuermann (2016)).

Proof of Work (PoW)

Como Nakamoto explica no whitepaper do Bitcoin [4], a *proof of work*, descrita por ele é similar ao Hashcash do Adam Back. Esta envolve a procura por um valor de *hash*, utilizando de uma função criptográfica como SHA-256, e incrementando um nonce até que um hash com uma certa quantidade de bits 0 seja encontrada.

O trabalho computacional demandado para encontrar o valor satisfatório é exponencial no número de bits 0 demandado, porém para sua verificação basta o cálculo de um único hash [4, pp. 3]. Trabalhar para gerar novos blocos nesse sistema de *proof-of-work* é chamado de mineração e os nós que realizam este processo são os mineradores [14]. Este termo vem de uma analogia, descrita por Nakamoto 2008, como “*The steady addition of a constant of amount of new coins is analogous to gold miners expending resources to add gold to circulation*”, sendo os recursos gastos neste caso, tempo de CPU e eletricidade e o ouro resgatado por eles é a recompensa, no caso em Bitcoins, da validação de um bloco. Esta recompensa é reivindicada por aquele minerador que primeiro encontrar o *hash* adequado, através de uma transação especial no sistema do Bitcoin, que é inserida no bloco por ele criado, chamada *coinbase transaction*. [14][2][4]

Com isso a dificuldade dos mineradores não é computar o hash, que é fácil, mas encontrar o hash adequado através de uma busca *brute-force* [8][5]. Para proteger a integridade dos dados, cada bloco de transações possuirá um único *hash*, que será derivado da informação contida no mesmo e o valor do hash do bloco anterior, criando assim uma corrente de blocos que não pode ser alterada sem refazer o proof-of-work.

A função de *hash* utilizada pelo bitcoin é a seguinte [1]:

|  |
| --- |
| H (N || P\_hash || Tx || Tx || ... Tx) < Target |

Onde *N* é o *nonce* que varia para alterar os valores do *hash*, *P\_hash* é o *hash* do bloco anterior, *Tx* representa as transações no bloco e *Target* é a dificuldade do *hash* (chamada de dificuldade da rede) que é imposta pela rede a ser satisfeita. Isto significa que o *hash* que deve ser encontrado pelos mineradores deve ser menor que o imposto pelo *Target* para poder ser considerado válido [1].

O bitcoin também utiliza um mecanismo para forçar os blocos a serem gerados aproximadamente a cada 10 minutos, ajustando dinamicamente a dificuldade do hash a cada 2016 blocos (por volta de 2 semanas). Para conseguir esta taxa de adição constante, o *Target* é calculado seguindo a seguinte função [1]:

|  |
| --- |
| Target=Previous target \* Time/2016 \* 10min |

*Previous target* é o antigo valor do *target* e *time* é o tempo que foi gasto até gerar os 2016 blocos anteriores. Portanto a dificuldade da rede basicamente significa o quão difícil vai ser para os mineradores encontrarem o *hash* do próximo, ou seja, o quão difícil o quebra cabeças criptográfico está no momento.

O uso dos *hash pointers* força os possíveis invasores do sistema a terem de refazer todo o trabalho computacional na busca dos *hashes* que já foi realizado pelos mineradores, caso pretendam fazer alguma alteração em dados já gravados. Um complicador ainda maior é o fato do sistema de consenso determina que os nós que compõem a rede apenas irão aceitar, como estado válido da Blockchain, a cadeia de blocos mais longa, ou seja, em caso de conflitos o *branch* (ramo) que possuir o maior número de “provas de trabalho” é o verdadeiro estado do *ledger*. Dessa forma as inserções de novos blocos pelos mineradores sempre serão feitas na cadeia mais longa, forçando nós maliciosos a terem de ultrapassá-la para subjugar a rede [5, pp. 398].

A vantagem deste sistema é o fato de que apesar do grande esforço demandado para encontrar o *hash* adequado à dificuldade da rede no *proof-of-work*, a verificação é de baixo custo e pode ser realizada pelos demais computadores da rede rapidamente, bastando o cálculo de um único *hash*[8].

O problema do consenso é solucionado pelo *proof-of-work* é uma inovação no conceito de decisão por maioria. Como já dito, nós de uma rede Blockchain aceitarão a cadeia mais longa de blocos como o estado verdadeiro, logo, este modelo se torna diferente de um consenso por maioria tradicional, baseado em um-IP-um-voto, o qual poderia ser subvertido alocando IPs suficientes para possuir a maioria da rede, sendo o *proof-of-work* essencialmente um-CPU-um-voto. A maioria é determinada pelo maior esforço computacional, a cadeia com mais “provas de trabalho” é a que gastou mais tempo calculando *hashes*. Como o próprio Nakamoto (2008) ressalta “*If a majority of CPU power is controlled by honest nodes, the honest chain will grow the fastest and outpace any competing chains.*”. Desta forma, a rede continuará honesta, uma vez que a maior parte do poder computacional for composta de usuários honestos [4, pp. 3].

O protocolo também resolve um antigo problema de sistemas distribuídos, conhecido como problema dos generais bizantinos. Este problema se relaciona ao sistema conseguir ser tolerante a informações incorretas, possivelmente propositais, circulando a rede. Como no *proof-of-work* as informações são validadas por toda a rede, onde os nós votam nas transações corretas ao inseri-las em um novo bloco, este problema é solucionado e a rede pode lidar com este tipo de ataque, desde que a maioria dos nós se mantenha honesto.

*Proof-of-stake* (PoS)

O *Proof-of-stake* (PoS) é uma técnica alternativa para o sistema de consenso tradicional utilizado nas blockchains, o proof of work, o qual é extremamente custoso computacionalmente e demanda um gigantesco gasto energético. O PoS visa reduzir drasticamente estes problemas mantendo a característica de ser tolerante à falha bizantina e adicionar também uma maior segurança à rede [1] [2].

O PoS utiliza a ideia de *Coin Age*, como informação fundamental no momento de realizar a validação de um bloco. Quanto maior o *coin age* total de um bloco mais fácil será encontrar sua hash. O cálculo para obter o valor de *coin age* é feito com base em quanto tempo uma quantia ficou sem ser gasta na Blockchain. Por exemplo, ao enviar 2 moedas, as quais estavam paradas em uma transação por 90 dias, o *coin age* referente a essa nova transação estará em 180 *coin-days* e será zerada ao ser transferida para o novo dono. As *coin ages* das transações do bloco são somadas para se obter o *score* (pontuação) do bloco e este valor é usado para definir a dificuldade daquele bloco. Analogamente ao *proof-of-work* o valor do *hash* deve ser igual ou menor o valor da dificuldade obtida para o bloco poder ser válido. Isto limita o espaço de busca pelo *hash* (sendo aqui inversamente proporcional ao *score* do bloco), o qual anteriormente era virtualmente infinito no PoW, além de não haver uma maneira de se obter vantagem utilizando o poder computacional, uma vez que apenas o valor do campo do *timeStamp* (tempo) é variável e com isso cada tentativa de *hash* é feita a cada segundo, reduzindo drasticamente o custo energético [2][3]. Como não há este esforço nas obtenções de moeda, trabalhar em *hashes* em um sistema PoS é chamado de forjar. Novas moedas são forjadas pelos nós a cada bloco e não mineradas como no PoW.

Assim como no PoW existe a *coinbase* *transaction*, na criptomoeda peercoin que implementa o PoS existe a *coinstake* *transaction*, em que os donos enviam moedas de seus endereços para eles mesmos, adicionando juntamente uma porcentagem pré definida de recompensa. Esta transação, além de aumentar a chances do nó de forjar novos blocos de acordo com quanto capital possuí, da a chance aos outros participantes, uma vez que o coin age é zerado ao ser utilizado. Logo mesmo que um nó possua um *stake* (quantia na moeda) muito alto, o rodízio de validadores ainda ocorrerá pois o *coin age* levará vários blocos até ter um valor significativo novamente [2].

Outra vantagem deste sistema é o fato do consenso ser baseado em quantidade de criptomoedas, assim para se obter a maioria e poder comprometer a segurança da Blockchain, seria necessário ter 51% das moedas em circulação, podendo ser ainda mais caro do que se obter 51% do poder computacional. Além do fato de que quanto mais valor investido um nó possuir, mais interesse ele tem de que o sistema funcione corretamente e a moeda valorize e não teria vantagens ao sabotar e diminuir a confiança na criptomoedas em questão [2].

Metodologia

Arquitetura da Blockchain proposta

A Blockchain proposta é formada por duas estruturas principais, o BlockHeader e o Block (Imagem Y). O BlockHeader é onde são inseridas informações de cabeçalho do bloco e as hashes das informações contidas no Bloco. JSON (*JavaScript Object Notation*) foi escolhido como método de codificação padrão, por ser amplamente utilizado por aplicações na internet e poder ser lido por diversos programas. Os campos da *BlockHeader* consistem em:

1. Index: que consiste no número do bloco na sequência da cadeia da Blockchain.
2. TimeStamp: é a data e hora que o bloco foi forjado pelo servidor.
3. *PrevHash*: campo que contém uma string a qual representa o *hash* do bloco anterior, usando SHA-256 [shaREF].
4. *TransactionsHash*: um *hash* que consiste em uma codificação dupla, primeiro o vetor de transações é codificado em JSON [jsonREF], se tornando um único vetor de bytes, o qual será salvo no corpo da Blockchain e sobre este vetor de bytes é aplicado o algoritmo do SHA-256 [shaREF].
5. *GameStateHash*: consiste no hash, em SHA-256 [shaREF], do JSON [jsonREF] contendo o estado do jogo no momento do início da criação daquele bloco.
6. *Hash*: campo contendo o *hash* do bloco atual, também em SHA-256 [shaREF], sendo adicionado ao final do forjamento do mesmo.
7. Validator: campo que guarda o login do player responsável pelo forjamento do bloco, o qual deve ser um player existente no jogo.

O BlockHeader descrito à cima é um campo da estrutura que compõe o bloco. O bloco, então, é formado pelos seguintes campos:

1. BlockHeader: com as informações de cabeçalho e *hashes*.
2. Transactions: campo que consiste em um vetor de transações, codificado em JSON [jsonREF].
3. GameState: JSON contendo o estado do jogo no início da criação daquele bloco.



Imagem Y

Uma transação no modelo foi estabelecida como uma estrutura para agrupar os comandos de um determinado player juntamente com sua aposta (Bet) para aquele conjunto. Foi adicionado um campo de número de sequência, que é controlado para cada usuário (*player*) específico, para permitir evitar transações repetidas sejam adicionadas no bloco e com isso fazer o *flooding* eficiente da informação. O servidor recupera a informação do último número de sequência de cada player ao carregar a Blockchain e sempre que uma nova mensagem é enviada, ou recebida, este é incrementado para aquele player específico. Desta forma uma mensagem antiga ou repetida é facilmente detectada, pois conterá um número de sequência menor ou igual o último conhecido, podendo assim ser descartada. Dito isto, a estrutura de uma transação é formada pelos campos:

* Player: campo contendo o nome do player que gerou os comandos que serão inseridos na Blockchain.
* SeqNumber: campo com o número de sequência daquela transação.
* Commands: um vetor de *strings*, que representam os comandos, definidos na seção do protocolo de comunicação e cujo primeiro comando é necessariamente o Bet daquela transação, que será usado para o cálculo da dificuldade do bloco.

Arquitetura de Processos

Para cada instância do processo de jogo, existe uma instância do processo servidor. O jogo sempre agirá como um cliente do seu servidor local, o qual ficará responsável por fazer a comunicação com os demais servidores em outras máquinas da rede.

Nesta seção serão explicados e ilustrados os processos que são realizados pelo sistema criado. Descrevendo o funcionamento do estabelecimento de conexão entre os servidores, através do método push-pull, os processos de forja e dispersão de novos blocos e recebimento de requisições de outros servidores e do processo jogo.

**Processos gerais da Blockchain**

**Processo de forjamento/recebimento de novos blocos**

O forjamento de um bloco ocorre sempre que houver alguma transação na *pool* de transações do servidor. Uma thread de forjamento fica ativa, monitorando a *pool* de transações e assim que há ao menos uma transação, começa a o forjamento de um novo bloco, seguindo os passos:

1. A thread retira todas as transações do pool, deixando-o vazio.
2. Adiciona as transações e um *hash* em SHA-256 dessas transações no novo bloco.
3. Procura pelo *hash* adequado a dificuldade do bloco definida pela função criada de PoS, a qual é dinâmica.
4. Quando o *puzzle* do PoS for solucionado, e se for solucionado antes que os demais nós, insere na Blockchain local e faz o *flooding* do mesmo na rede.

O fluxo de atividades após o forjamento ou recebimento de um novo bloco é definido pelos seguintes passos:

1. Caso seja recebido um bloco, passa pelo processo de validação. Se ele for inválido é, então, descartado. Caso contrário, segue para passo 2. Em um último caso, onde ele é forjado, passo 3.
2. Verifica se a cadeia principal não foi ultrapassada, se sim realiza o processo de recuperação da cadeia principal. Caso contrário, segue para a etapa 3.
3. Faz o *broadcast* do bloco para os servidores conectados.
4. Envia os comandos das transações do bloco para o processo jogo um a um e na ordem que aparecem no bloco.
5. Anuncia ao processo jogo que o envio de comandos daquele bloco foi finalizado.
6. Recebe o *Game State* com a execução já realizada dos comandos no jogo e salva como o estado atual do jogo.

Para um novo bloco ser considerado válido e poder ser adicionado na cadeia, este precisa atender às seguintes condições:

1. O Index ser o número do último bloco na cadeia mais 1.
2. O *TimeStamp* ser posterior ao do último bloco válido.
3. O *hash* do bloco ser válido e atender a dificuldade da rede proposto para ele.
4. O *hash* anterior ser o do último bloco.
5. O *hash* do *Game State* estar correto.
6. O *hash* das transações estar correto.
7. O Validador existir no jogo.
8. As Transações serem todas válidas.

Uma transação é válida para este modelo caso esta atenda as condições abaixo:

1. O nome do Player esteja registrado no jogo e seja válido;
2. O número de sequência ser maior ou igual ao último conhecido pelo servidor;
3. Os comandos seguem o padrão de comandos de jogo, descrito na seção j.k;

**Cálculo da dificuldade do bloco e *hash***

Uma vez que o protocolo usado nesta implementação é o PoS, a dificuldade do *hash* é calculada dinamicamente e individualmente para cada novo bloco. Este cálculo é baseado no valor de *Bet* total das transações que compõem aquele bloco e nos segundos do valor do *timestamp*, o qual é o único campo variável e dessa forma força as tentativas de busca do *hash* a ocorrerem por segundo, sem uso de poder computacional. O cálculo então é feito utilizando o somatório dos *Bets* e o valor dos segundos na seguinte fórmula:

Onde *stack* é o somatório dos *Bets* do bloco, *const* é um valor constante, s é os segundos do campo de *timestamp*, o qual muda a cada tentativa. A ideia é que o valor de *const* seja subtraído a cada segundo, deixando a dificuldade cada vez mais fácil a medida que o tempo avança. Como nesta fórmula os segundos do *timestamp* estão sendo utilizados, *s* sempre varia dentro do intervalo de valores que vai de 0 a 59, portanto o valor de *const* escolhido foi o de 59 para que desta forma em no máximo 59 tentativas por segundo um novo *hash* seja encontrado por um servidor.

Assim como em um PoW clássico, visto na seção l.m, na validação dos blocos desta implementação a quantidade de bits 0 no início do *hash* deve ser igual ou menor do que a especificada pela dificuldade do bloco. Como o cálculo da dificuldade diminui a cada segundo, ao chegar em 0 qualquer *hash* é aceito e um bloco é forjado com cem por cento de chance. Assim a fórmula de cálculo do *hash* pelo *proof-of-stake* implementado é:

H (*Index* || *GameStateHash* || *TransactionsHash* || *PrevHash* || *Validator* || *Seconds*) < Dificuldade do Bloco

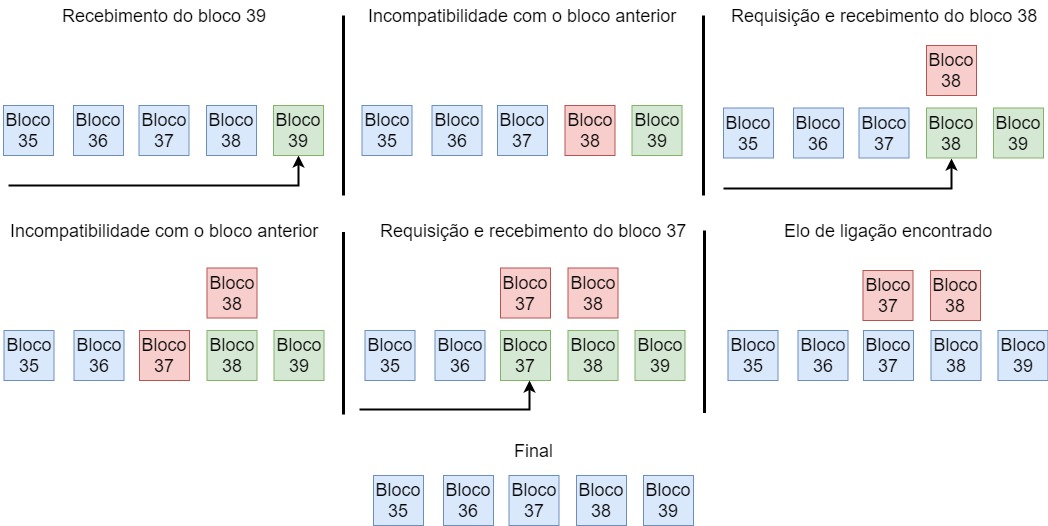
Utiliza para os cálculos os campos estáticos do cabeçalho, já anteriormente especificados e *Seconds* que representa os segundos do campo de *timestamp*, sendo o único campo variável de todo o cálculo do *hash*.

**Processo de recuperação da cadeia principal**

Quando um bloco é recebido e é julgado inválido, algumas situações podem ocorrer:

* Bloco inválido *hash* incorreto: neste caso alguma informação no bloco não coincide,e portanto o *hash* fica incorreto. O bloco é então descartado e o servidor retorna o processo de busca pelo *hash*.
* Bloco inválido, porém *hash* do bloco está correto e *hash* do bloco anterior não coincide ou *index* é superior ao esperado: neste caso o bloco está correto mas ele provém de uma cadeia principal (*main chain*) diferente da salva local, logo a cadeia local foi ultrapassada e deve-se iniciar o processo de recuperação.

O processo de recuperação da cadeia principal ocorre quando o bloco é inválido por pertencer a uma *main chain* diferente, portanto ele não é realmente inválido. O que ocorreu de fato, foi uma cadeia de blocos diferentes da que o servidor que recebeu o bloco havia considerado principal se tornou a maior. Dessa forma o servidor deve aceitar este bloco como válido, para manter a característica da Blockchain de aceitar a cadeia mais longa como válida e requisitar os blocos anteriores até encontrar o elo de ligação entre a sua cadeia local e a nova que se tornou maior. Por exemplo, um servidor A está com uma cadeia de tamanho n, e portanto trabalhando no *hash* do bloco n+1, ao receber um bloco n+1 de outra cadeia principal e aceitá-la como novo estado verdadeiro e supondo que as duas cadeias possuem k blocos diferentes, este irá requisitar um a um, utilizando a mensagem padrão de requisição de blocos especificada no protocolo, até encontrar o bloco n-k+1 o qual tenha o *hash pointer* compatível com sua cadeia local. Após encontrar esse elo, e caso todos os k blocos recebidos estejam válidos, irá substituir sua cadeia principal por está nova recebida. Caso haja algum bloco recebido inválido, todos os blocos recebidos anteriormente são descartados e a cadeia permanece a mesma. O processo pode ser visto na figura abaixo3, onde neste exemplo o elo de ligação é o bloco 37.



Inserção do bloco 39 em uma Blockchain com processo de recuperação da cadeia principal (figura abaixo3)

As transações contidas nos blocos substituídos são inseridas novamente na *pool* de transaçõese uma verificação é feita, com base nos números de sequência, para identificar aquelas que já foram inseridas nos novos blocos que formam a nova maior cadeia. Aquelas que estiverem repetidas são apagadas da *pool*, para não ocorrerem problemas de transações serem inseridas duas vezes em blocos diferentes.

**Processos de comunicação**

Nesta seção serão abordados os processos de comunicação entre as entidades que compõem esse sistema distribuído. Explicitando as principais trocas de mensagem entre os servidores que compõem a Blockchain e o processo jogo e sua comunicação com o servidor local associado.

**Processo com troca de mensagem entre servidores**

**Estabelecendo e encerrando conexão com demais servidores**

Para facilitar o estabelecimento de uma conexão com um outro servidor da rede, foi criado uma tabela em arquivo .csv local, que é lida assim que o servidor é inicializado, com alguns IPs de servidores na rede. São selecionados 3, que estiverem online, aleatoriamente desta lista local e aberta uma conexão TCP persistente. Uma mensagem de início de conexão padrão OPEN deve ser enviada, como está definida na seção protocolo do protocolo de comunicação criado e o servidor assim que aceitar a conexão, enviará como resposta a sua tabela de IPs. Caso a tabela recebida for maior (em número de IPs), a local é substituída por ela (apenas a maior tabela é mantida). Todo servidor tentará realizar o mesmo processo, tentando sempre estabelecer a conexão com 3 clientes, ao mesmo tempo manterá uma *thread* para ouvir novas solicitações de conexão de outros servidores, sempre na porta 8080. Criando assim uma rede de sobreposição não estruturada aleatória.

Assim que um novo servidor é conectado, tanto faz se por *push* ou por *pull*, este é adicionado a um *hash map* de conexões ativas, que relaciona o IP do servidor com a sua conexão, para facilitar a gestão dos servidores conectados e poder realizar o broadcast das mensagens à eles. Após isso é enviado uma mensagem de requisição para atualizar a Blockchain local, podendo ser BLOCK SINCE, especificada na seção i do protocolo, para solicitar todos os blocos à partir do último bloco salvo, se existirem, ou uma mensagem BLOCK ALL, seção j do protocolo, requerendo toda a Blockchain do servidor conectado, para quando for a primeira inicialização do servidor.

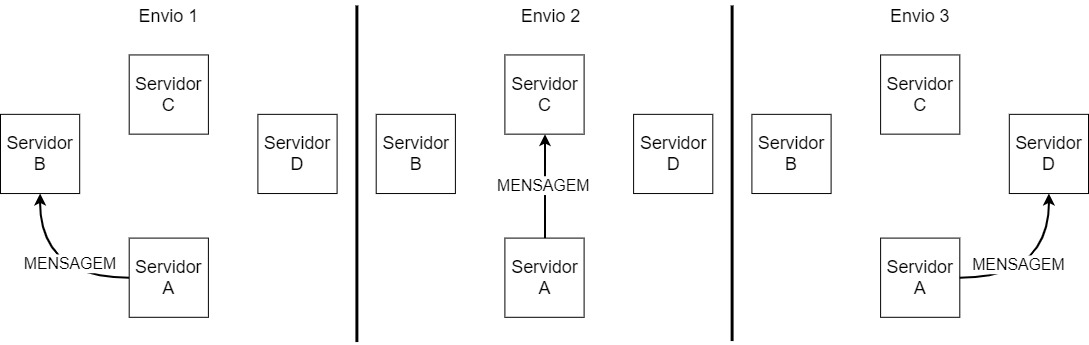
Caso algum servidor vá ser desligado e precise se desconectar da rede, este irá enviar uma mensagem CLOSE aos servidores conectados e dessa forma será retirado do *map* de servidores conectados.

**Flooding de transações e blocos**

A inundação, ou *flooding* em inglês, é feita através de um *broadcast* para todos os servidores conectados, enviando uma mensagem, que no caso da implementação pode ser do tipo TRANSACTION, para o caso de uma transação e para blocos ou uma mensagem tipo WIN que anuncia um bloco vencedor. O padrão da sintaxe das mensagens deste tipo é especificado na seção x do protocolo de comunicação.

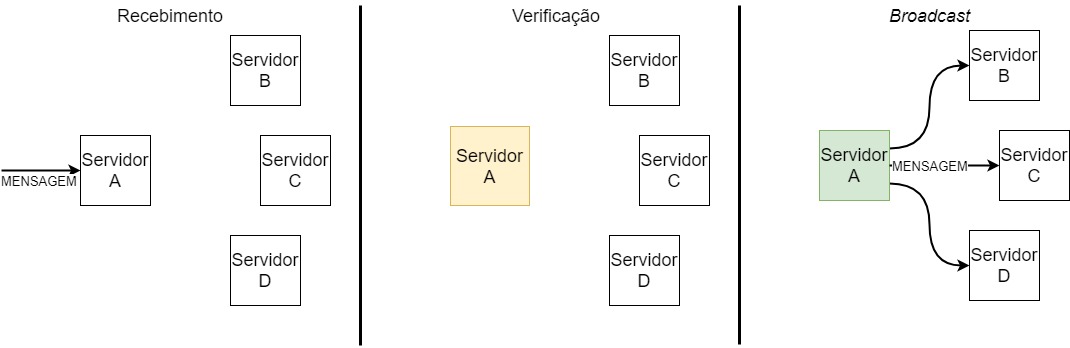
Transações e mensagens de forjamento de bloco (WIN) tem tratamento diferente quanto ao problema das mensagens repetidas e antigas na rede. As transações, como já foi explicitado, possuem um campo de número de sequência na sua estrutura, assim caso uma mensagem antiga chegue a algum servidor ela pode ser facilmente percebida e descartada, uma vez que seu número de sequência será menor do que o último recebido pelo servidor. Nas mensagens WIN, no entanto, não há a necessidade de campos especiais na mensagem, contendo apenas a estrutura do bloco codificada em formato JSON, já que cada bloco possui um número de índice (*index*) em seu cabeçalho (*Block Header*), blocos com índices menores que o esperado, que podem ser provenientes de mensagens repetidas, são automaticamente descartados da mesma forma.

Em ambos os casos as mensagens são formatadas, conforme o padrão no protocolo de comunicação seção j.k, e enviadas para os servidores conectados uma a uma, fazendo um *broadcast* de mensagens através de uma conexão TCP/IP, conforme a figura abaixo mostra. Onde o servidor A faz o *broadcast* de uma mensagem para os servidores B, C e D conectados a ele.



Envio de mensagens por *broadcast* (figura abaixo)

Servidores podem também receber uma destas mensagens de disseminação, neste caso, a mensagem passará pelas verificações de acordo com seu tipo, se não for repetida (pelo número de sequência), fora do padrão ou bloco inválido, os servidores então irão realizar o mesmo processo, um *broadcast* para suas conexões, com exceção daquela ao qual a mensagem foi recebida (Figura abaixo2).



Visão geral do processo de inundação (figura abaixo2)

**Processos com troca de mensagens entre jogo e servidor**

**Estabelecendo conexão entre jogo e servidor**

A conexão é iniciada assim que o jogo é inicializado. O processo jogo irá tentar estabelecer uma conexão TCP com o processo servidor através da porta 9090, apenas passando para a próximo estado quando a conexão estiver estabelecida. Após esse procedimento, o processo jogo irá requisitar do servidor o último *game state* salvo na Blockchain, para realizar o carregamento dos dados do mundo, utilizando a mensagem padrão SAVED GAMESTATE, descrita na seção m.n. Caso exista um *game state* salvo, ele será enviado com uma mensagem de GAMESTATE, seção m.l, caso contrário uma mensagem NOT FOUND, seção do protocolo m.k, será enviada e ambos a requisição do jogo será terminada. Se a mensagem contendo o estado do jogo (*game state*) foi recebida, o processo jogo então fará o carregamento dos dados dos jogadores e do estado do mundo. Independente de haver dados na Blockchain ou não, este então aguardará o *input* dos dados por parte de um usuário para prosseguir com o processo de *login* ou registro no sistema do jogo.

**Login ou registro no sistema**

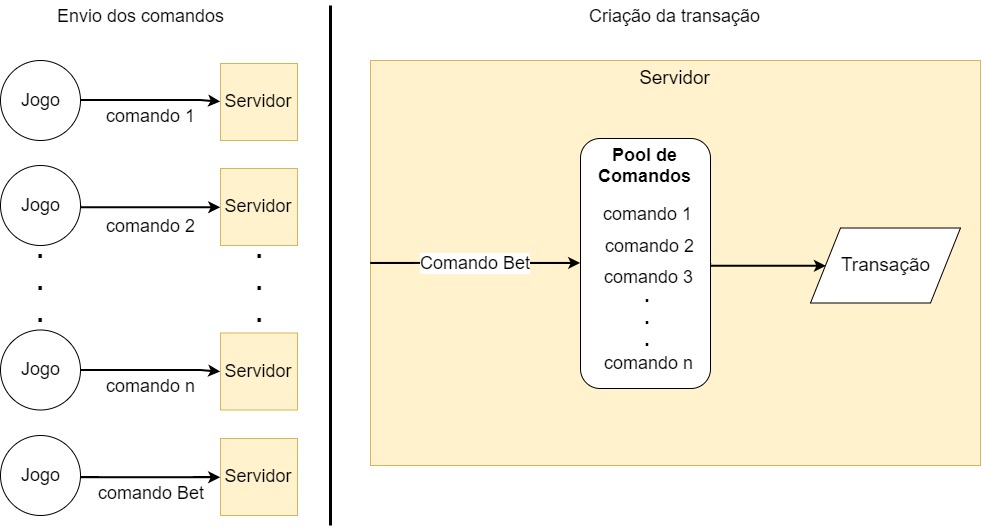
Este processo ocorre assim que o jogo termina o processo de estabelecimento de conexão, o sistema aguardará por uma entrada de *login* e senha do usuário com duas funções principais, o *login*, para usuários já cadastrados e que foram previamente carregados do *game state* salvo na Blockchain e o registro, para aqueles que ainda não estão cadastrados.

A mensagem padrão para *login* enviada pelo processo jogo ao servidor é descrita na seção j.k e para ser confirmado, o servidor deve concordar com o acesso, se os dados de *login* coincidirem. Neste ponto o processo jogo e o processo servidor devem ter o mesmo *game state* salvo para que não haja erros. Sempre que um novo jogador é logado no servidor, este ficará salvo como o jogador ativo, o qual só pode haver um, e todos os blocos validados neste período que permanecer logado levam seu nome de *login* como o validador do bloco.

O registro de novos jogadores é um pouco mais complexo, a mensagem padrão de envio está descrita na seção j.l, em vez de dados cadastrais, o processo jogo cria um novo objeto *Player*, codifica em JSON e então envia para o servidor. Ao receber a mensagem o servidor irá verificar algum erro na codificação ou na mensagem, resgatar os dados e adicionar à sua lista de jogadores conhecidos, criará uma transação, a qual sempre terá o número de sequência 0 (por ser a primeira mensagem daquele jogador), irá inserir essa transação na primeira posição da *pool* de transações e então por fim distribui a mensagem sem da forma com que foi recebida por *broadcast* para os servidores conectados. Os demais servidores que receberem tomarão as mesmas ações, com exceção de que também enviarão ao processo jogo conectado à eles, para que com isso todos os jogadores tenham conhecimento da existência deste novo jogador. Quando esta mensagem é recebida pelo jogo, o JSON é novamente transformado em objeto e adicionado aos jogadores existentes no jogo. Este processo de registro é diferente de um tradicional, devido ao fato de quando um novo jogador é criado no jogo, este recebe uma ilha com recursos aleatórios, portanto para manter todos os servidores e instâncias do jogo com os mesmo dados sobre a ilha e o jogador, todo o objeto é enviado para eles.

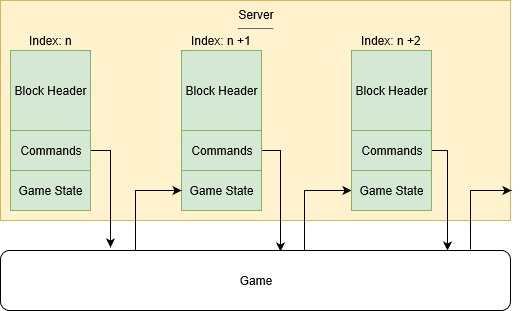
**Envio e Recebimento de comandos entre os processos**

Comandos no jogo são agrupados e enviados juntamente à um valor de aposta, chamado de *Bet*, que será utilizado no cálculo da dificuldade do bloco. Os comandos do processo jogo são enviados, através das mensagens especificadas na seção k.l, um a um para o servidor, que irá armazená-los na *pool* de comandos até receber um comando BET, com o valor da aposta, e então irá transformar aquele grupo de comandos recebidos até então em uma nova transação. O esquema da figura abaixo4 mostra o processo:



Processo de recebimentos de comando do servidor (Figura abaixo4)

Após a criação da nova transação, *broadcast* para a rede, inserção desta em algum bloco da Blockchain. As transações contidas no novo bloco são percorridas na ordem em que foram adicionadas, e os comandos são enviados ao processo do jogo, seguindo o padrão especificado na seção k.l do protocolo, que os executa e envia um *Game State* de volta ao processo servidor, utilizando a mensagem GAMESTATE especificada na seção k.j, com o resultado dessa execução. Dessa forma o servidor sempre terá o *Game State* referente a execução dos comandos do último bloco válido da cadeia e usará esta informação no forjamento do próximo bloco da cadeia, como mostra a figura x.

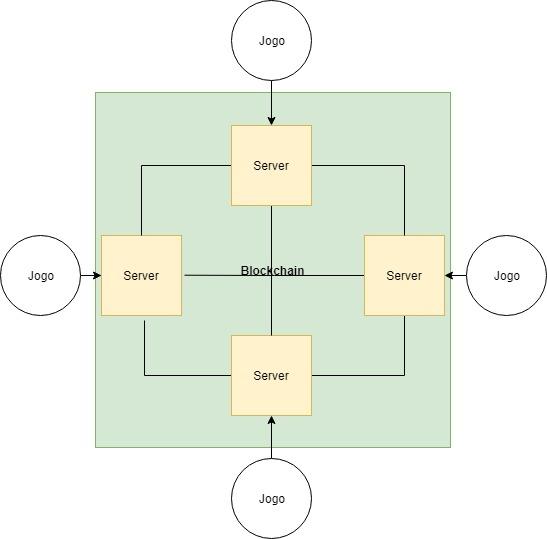


Envio dos comandos das transações para o processo jogo (Figura x)

Protocolo de comunicação

Descrição Geral

Devido à característica cliente servidor da aplicação, foi implementado um protocolo de comunicação dessa aplicação. Este protocolo tem o *stop-and-wait* [SAW] como método para controle de fluxo. Foi construído sobre o protocolo TCP/IP. Este é dividido em duas partes principais. A primeira parte consiste na comunicação entre o processo que constitui o jogo em execução e o processo que funciona como um servidor local para esta instância do jogo. E a segunda parte é a comunicação entre a instância local de servidor com os demais processos servidores que constituem os nós da rede *Blockchain*. Estes serão mais detalhadamente explicados na seção posterior e a visão global pode ser vista na Figura...



Visão geral dos processos (Figura…)

A próxima subseção descreve as mensagens do protocolo.

O protocolo

As mensagens são classificadas em três categorias. A primeira categoria é composta pelas mensagens de eventos que são utilizadas quando um jogador realiza algum tipo ação no jogo. A ação é requisitada junto ao servidor local e este envia para os demais servidores dos demais jogadores. As mensagens de controle que são usadas quando um novo jogador é adicionado ou um jogador já existente se conecta ao jogo e portanto, conecta-se a sua instância de servidor local. Por último, tem-se as mensagens de requisição e espalhamento de blocos, esta ocorre entre servidores: Quando um servidor se conecta a rede, este receberá os blocos desde que foram validados desde a última vez que se conectou; Quando um bloco é validado, neste caso o bloco validado é espalhado para os demais servidores conectados; Quando um novo jogador adentra o jogo e o seu servidor local deve se conectar a primeira vez com a Blockchain e portanto receberá toda a cadeia de blocos.

1. Comunicação jogo-servidor
2. Estabelecendo/Encerrando conexão

Sempre que uma instância cliente do jogo é criada, o novo usuário faz autenticação ao jogo, uma conexão com o servidor local deve ser estabelecida. O cliente comunica-se com o servidor local através de troca de mensagens. Portanto as mensagens de estabelecimento de conexão são realizadas quando a autenticação do usuário é realizada, e de finalização quando o usuário se desconecta do jogo. Seguem os seguintes padrões:

* LOGIN USER KEY - “LOGIN USER KEY\n”
  + Onde USER é uma string que representa o *login* que o usuário está cadastrado no jogo e KEY é outra string contendo o hash da senha relacionada aquele usuário de USER. Ambas informações são concatenadas a “LOGIN” e separadas por 1 espaço cada, finalizando a mensagem com um “\n”.
  + Ex: “LOGIN FULANO 123456\n”, “LOGIN CAIO 244466666\n”, etc.
* REGISTER JSON - “REGISTER JSON\n”
  + Onde JSON é uma string que representa o objeto Player que o usuário está cadastrado no jogo. O JSON é concatenado a “REGISTER” e separados por 1 espaço, finalizando a mensagem com um “\n”.
  + Ex.: “REGISTER {"username":"caio","passwordHash":"-1058272838","ether":1000,"islands":[{"maxPopulation":78,"maxWood":1301,"maxStone":1208,"maxFood":1363,"population":10,"extractedWood":0,"extractedStone":0,"food":0,"woodWorker":10,"stoneWorker":20,"foodWorker":30,"army":0,"attacking":false,"underAttack":false}],"wood":0,"stone":0,"maxStorage":1000}\n”
* LOGOUT - “LOGOUT\n”
  + A string “LOGOUT\n é enviada pelo servidor que deseja encerrar a conexão e não é esperada mensagem de confirmação.

1. Mensagem de confirmação/erro

Mensagens de confirmação são enviadas sempre depois de receber algum evento de jogador, login ou registro. Podendo também indicar fim de um bloco ou um eventual erro. Não esperam mensagem em retorno e seguem o padrão abaixo:

* OK - “OK\n”
  + Uma string “OK\n” que indica o recebimento correto da mensagem enviada anteriormente.
* END BLOCK - “END BLOCK\n”
  + String “END BLOCK\n” que indica o fim do envio de comandos de um bloco por parte do servidor.
* ERROR COD - “ERROR COD\n”
  + Onde COD representa um inteiro que está relacionado ao erro ocorrido. Sendo portanto concatenado na string “ERROR”, separados por um espaço e finaliza a mensagem com “\n”.
  + Ex.: “ERROR 0\n”, “ERROR 127\n”, “ERROR 7\n”, etc.
* NOT FOUND - “NOT FOUND\n”
  + Mensagem enviada em resposta a mensagem SAVED GAMESTATE, no caso em que nenhum *game state* salvo foi encontrado na Blockchain. É formada por uma string única “NOT FOUND” concatenada ao terminador “\n”.
  + Ex.: “NOT FOUND\n”

1. Eventos de Jogador

Toda ação realizada por um jogador interferirá no jogo de todos os demais, portanto estas devem ser mensagens enviadas ao servidor. Este deverá garantir que todos vejam as mudanças e que estas ocorram na mesma ordem. Todos os comandos são concatenados a uma string inicial “COMMAND ” antes de serem enviadas ao servidor. As ações possíveis por parte do jogador e a estrutura das suas mensagens seguem os padrões abaixo:

* ATTACK PLAYER ISLAND FROM MYISLAND - “ATTACK PLAYER ISLAND FROM MYISLAND\n”
  + Mensagem padrão de ataque, devem ser informados o destino (usuário que irá ser atacado). Onde PLAYER é uma string que representa o *login* que o usuário que está sendo atacado está cadastrado no jogo. ISLAND que representa o ID da ilha do player atacado. E MYISLAND é o ID da ilha atacante. A informação é concatenada a string “ATTACK” e string “FROM” e separadas por 1 espaço finalizando a mensagem com um “\n”. Uma mensagem de confirmação é aguardada.
  + Ex.: “COMMAND ATTACK CAIOVR 0 FROM 1\n”, “ COMMAND ATTACK FULANO 3 FROM 0\n”, etc.
* EXPLORE ISLAND ID - “EXPLORE ISLAND ID\n”
  + Mensagem de Exploração de ilha, uma nova ilha é gerada randomicamente e o exército da ilha que tem o ID especificado na mensagem é enviado para a exploração. ISLAND é uma string JSON que representa a ilha que será criada. ID é um inteiro que representa o id da ilha que o jogador que está fazendo a exploração possui e de onde o exército partirá para a exploração. O inteiro ID é concatenado a string JSON “ISLAND” e a string “EXPLORE”, separado por um espaço e finalizado com um “\n”. Uma mensagem de confirmação é aguardada.
  + Ex.: “EXPLORE {"maxPopulation":97,"maxWood":1044,"maxStone":1116,"maxFood":1313,"population":10,"extractedWood":0,"extractedStone":0,"food":0,"woodWorker":10,"stoneWorker":20,"foodWorker":30,"army":0,"attacking":false,"underAttack":false} 0\n”, etc.
* CHANGE OLD\_ROLE QTD TO NEW\_ROLE IN MYISLAND - “CHANGE OLD\_ROLE QTD TO NEW\_ROLE IN MYISLAND\n”
  + Mensagem que tem função de trocar as classes dos cidadãos da ilha. Onde OLD\_ROLE e NEW\_ROLE são strings que podem assumir os seguintes valores: “NOT DEFINED”, “SOLDIER”, “WORKER\_WOOD”, “WORKER\_STONE”, “WORKER\_FOOD”. E QTD um valor inteiro positivo maior que 0. Estas são concatenadas a string “CHANGE” e “TO”, separados por um espaço cada e finalizando a mensagem com “\n”. Aguarda uma mensagem de confirmação do servidor.
  + Ex.: ”CHANGE WORKER\_WOOD 30 TO SOLDIER\n”, “CHANGE WORKER\_FOOD 2 TO SOLDIER\n”, “CHANGE SOLDIER 20 TO WORKER\_GOLD\n”, etc.
* CREATE ARMY TYPE QTD - “CREATE ARMY TYPE QTD\n”.
  + Mensagem para a criação de exército na ilha. TYPE é um tipo string que representa o tipo de exército que será criado, podendo ser do tipo: SOLDADO,e QTD é um inteiro que estabelece a quantidade que será criada. Ambos são concatenados a string “CREATE ARMY” e separados por um caractere de espaço vazio entre eles, finalizando a mensagem com o caractere “\n”. Uma mensagem de confirmação é aguardada do vinda do servidor é aguardada.
  + Ex.: “COMMAND CREATE ARMY SOLDADO 1\n”.
* CREATE MACHINE QTD - “CREATE MACHINE QTD\n”
  + Mensagem para criação das máquinas de guerra. MACHINE é uma string que indica qual a máquina que será construída. E QTD é um inteiro maior que 0, que indica a quantidade. São concatenados a string “CREATE”, separados por um espaço cada e finalizando a string com “\n”. Uma mensagem de confirmação é enviada, caso a solicitação de explorar seja aceita e uma mensagem de erro caso contrário.
  + Ex.: “CREATE FLYER 10\n”, etc.
* CREATE ETHER QTD - “CREATE ETHER QTD\n"
  + Mensagem para criação de Ethers, a moeda do jogo, consumindo recursos quando feito. ETHER é uma string que indica que um Ether será criado. E QTD é um inteiro maior que 0, que indica a quantidade. São concatenados a string “CREATE”, separados por um espaço cada e finalizando a string com “\n”. Uma mensagem de confirmação é enviada, caso a solicitação de explorar seja aceita e uma mensagem de erro caso contrário.
  + Ex.: “CREATE ETHER 1\n”, etc.
* BET VALUE - “BET VALUE\n”
  + Mensagem para anúncio da quantia de aposta que um jogador fará para aquele conjunto de comandos. BET é uma string que é concatenada a VALUE, que por sua vez é um inteiro positivo que indica a quantia de Ether que será apostado. São separados por um espaço e concatenados a uma string “\n” para finalizar a mensagem. Aguarda uma mensagem de confirmação padrão ou de erro.
  + Ex.: “BET 0\n”, “BET 100\n”, etc.

4. Eventos de jogo

Existem trocas de informações entre o cliente-jogo e o servidor da instância do jogo que independem de ações do jogador para serem executadas. Estas se relacionam com o salvamento /carregamento do estado do jogo e envio de comandos por parte do servidor quando algum bloco é forjado. Com isto é possível manter os estados de jogo de todos os jogadores conectados bem próximo um do outro e manter a ordem de ações globalmente. As mensagens de evento de jogo seguem o padrão abaixo:

* SAVED GAMESTATE - “SAVED GAMESTATE\n”
  + Mensagem enviada pelo jogo para requerir o último estado do jogo (*game state*) salvo na Blockchain. Onde SAVED GAMESTATE é uma string de requerimento do último JSON de jogo salvo seguro da Blockchain. Uma mensagem de GAMESTATE (Seção x.y) é aguardada pelo cliente como resposta.
* GAMESTATE JSON - “GAMESTATE JSON\n”
  + Mensagem enviada pelo processo servidor contendo o último estado do jogo (*game state*) salvo. JSON é uma string que representa o estado do jogo, codificado no formato de json. Está é concatenada a string “GAMESTATE”, separados por um espaço e finalizando a mensagem com um “\n”. Não espera mensagem de confirmação em retorno.
  + Ex.: “GAMESTATE {"keys":["caio"],"values":[{"username":"caio","passwordHash":"-1058272838","ether":1000,"islands":[{"maxPopulation":78,"maxWood":1301,"maxStone":1208,"maxFood":1363,"population":10,"extractedWood":0,"extractedStone":0,"food":0,"woodWorker":10,"stoneWorker":20,"foodWorker":30,"army":0,"attacking":false,"underAttack":false}],"wood":0,"stone":0,"maxStorage":1000}]}\n”
* COMMAND PLAYER COMM - “COMMAND PLAYER COMM\n”
  + Mensagem enviada pelo servidor ao jogo sempre que um novo bloco é validado e adicionado a Blockchain. É enviada uma mensagem de command para cada comando presente em cada transação do novo bloco adicionado. PLAYER é o nome de usuário do jogador a quem pertence a transação do comando e COMM um comando que estava armazenado no bloco. São concatenados a string “COMMAND “ no início e finaliza a mensagem com “\n”.
  + Ex.: “COMMAND caio BET 0\n”

1. Comunicação servidor-servidor
2. Estabelecendo/Encerrando conexão

Todas as conexões com os servidores da rede são estabelecidas assim que a instância do servidor local é inicializada. Portanto ao iniciar o servidor local, este irá se conectar com os demais de sua lista de endereços (tentando manter 3 conexões) e manterá conexões persistentes TCP. A mensagem de estabelecimento e encerramento de conexão seguem os seguintes formatos:

* OPEN ID - “OPEN ID\n”
  + Onde “OPEN ID” é uma string finalizada com um fim de linha ‘\n’, que é a mensagem padrão para conexão com servidores. É esperado uma mensagem contendo a tabela de servidores de volta. Caso seja um novo jogador/servidor para entrar na rede o endereço IP é adicionado na tabela do servidor antes da tabela ser enviada. O cliente substituirá sua tabela de servidores local, caso a recebida seja maior. Em caso de erro, retorna erro.
* CLOSE - “CLOSE\n”
  + A string “CLOSE\n é enviada pelo servidor que deseja encerrar a conexão e não é esperada mensagem de confirmação.

1. Mensagem de confirmação/erro

Mensagem de confirmação enviada sempre depois de receber alguma requisição/entrega e de finalização de envio (OK/END), exceto no caso de encerramento CLOSE. E as mensagens de ERRO, quando algum problema ocorrer, sendo seguidas do código relacionado ao erro que ocorreu.

* OK - “OK\n”
  + Uma string “OK\n” que indica o recebimento correto da mensagem enviada anteriormente.
* END - “END\n”
  + Uma string “END\n” que indica que o último bloco da requisição já foi enviado, ou seja o fim do envio de blocos, em casos onde mais de um bloco é enviado. É formada por uma string “END” concatenada à um “\n”.
  + Ex.: “END\n”
* ERROR COD - “ERROR COD\n”
  + Onde COD representa um inteiro que está relacionado ao erro ocorrido. Sendo portanto concatenado na string “ERROR”, separados por um espaço e finaliza a mensagem com “\n”.
  + Ex.: “ERROR 0\n”, “ERROR 127\n”, “ERROR 7\n”, etc.

1. Requisição de blocos ou toda a Blockchain

As mensagens de requisição de blocos são enviadas para servidores conectados, sempre que um servidor que estava desligado se conecta a rede, um servidor novo entra na Blockchain, ou uma cadeia de blocos foi ultrapassada. Os blocos são enviados um a um e para cada recebimento de bloco uma mensagem de confirmação (OK) é enviada pelo servidor que fez a requisição. Ao final, o servidor que está enviando os blocos, envia uma mensagem de finalização de envio (END) para sinalizar que todos os blocos requisitados já foram enviados e encerrar aquela requisição.

* BLOCK ALL - “BLOCK ALL\n”
  + Uma string única “BLOCK ALL\n” onde o “ALL”, separado por um espaço, indica que todos os blocos são requeridos.
* BLOCK SINCE I - “BLOCK SINCE I\n”
  + A string “BLOCK SINCE” é concatenada, juntamente com um espaço em branco, à um inteiro I (“i”) seguido de fim de linha. Este inteiro I representa o index do último bloco que o servidor que fez a requisição possui. Todos os blocos à partir deste index, até o último já validado pela rede serão enviados.
  + Ex: “BLOCK SINCE 30\n”, “BLOCK SINCE 0\n”, “BLOCK SINCE 123\n”, etc.
* BLOCK BEFORE I - “BLOCK BEFORE I\N”
  + A string “BLOCK BEFORE” é concatenada, juntamente com um espaço em branco, à um inteiro I (“i”) seguido de fim de linha. Este inteiro I representa o index do maior bloco atual que o servidor possui, e é usada quando algum servidor tem sua cadeia mais longa ultrapassada e necessita recuperar blocos da nova cadeia principal. Um bloco de index I - 1 é enviado ao servidor que fez a requisição.
  + Ex: “BLOCK BEFORE 30\n”, “BLOCK BEFORE 200\n”, “BLOCK BEFORE 123\n”, etc.
* BLOCK I - “BLOCK I\n”
  + Mensagem enviada quando um requerimento de blocos é feita, pelo servidor que está atendendo a requisição. Onde BLOCK é um vetor de bytes que representa a codificação do bloco. Finaliza a mensagem com um “\n”.

1. Disseminação de comandos e blocos

Mensagens de disseminação de comandos e blocos são aquelas que serão enviadas por meio de mensagens broadcast para toda a rede quando ocorrerem. O método utilizado para o envio é a inundação, garantindo que todos os nós que possuem conexão vigente receberão a mensagem, apesar de aumentar significativamente o tráfego da rede.

O tratamento de mensagens repetidas é realizado de duas formas, na mensagem de vitória (WIN BLOCK), não é feita nenhuma tratativa, uma vez que cada bloco possui um número de identificação único e sequencial, logo mensagens com blocos de index repetidos serão automaticamente descartadas. Nas transações foi colocado na estrutura da mensagem um campo de número de sequência (SeqNumber) e do Player que enviou a mensagem, logo mensagens com o SeqNumber menor que o último recebido, para aquele player, serão consideradas repetidas e descartadas.

* WIN BLOCK\n - “WIN BLOCK\n”
  + BLOCK é um vetor de bytes em JSON que representa a codificação do bloco. Finaliza a mensagem com um “\n”.
  + Ex.:”WIN {"keys":["caio"],"values":[{"username":"caio","passwordHash":"-1058272838","ether":1000,"islands":[{"maxPopulation":78,"maxWood":1301,"maxStone":1208,"maxFood":1363,"population":10,"extractedWood":0,"extractedStone":0,"food":0,"woodWorker":10,"stoneWorker":20,"foodWorker":30,"army":0,"attacking":false,"underAttack":false}],"wood":0,"stone":0,"maxStorage":1000}]}\n”
* COMMAND PLAYER MSG\n - “COMMAND PLAYER MSG\n”
  + PLAYER é uma string que representa o *login* do player que realizou o comando. MSG é uma string que indica o comando realizado pelo player, que segue alguma das especificações descritas na seção I.2 deste documento. Finaliza a mensagem com um “\n”. Espera uma mensagem de confirmação padrão.
* TRANSACTION JSON\n - “TRANSACTION JSON\n”
  + Mensagem de espalhamento de transações. Esta é enviada sempre que um servidor recebe uma nova transação de um servidor conectado, ou cria uma com os comandos recebidos do jogo. JSON é uma string que representa a codificação de uma transação em formato JSON, seguindo o padrão de transações abordados em (z.a). Esta codificação é concatenada ao final da string "TRANSACTION", separados por um espaço e concatenado a um "\n" para finalizar a mensagem. É aguardado um ACK de confirmação padrão ou erro como retorno.
  + Ex.:”TRANSACTION {"Username":"caio","Commands":["BET 0","EXPLORE {\"maxPopulation\":57,\"maxWood\":971,\"maxStone\":1424,\"maxFood\":933,\"population\":10,\"extractedWood\":0,\"extractedStone\":0,\"food\":0,\"woodWorker\":10,\"stoneWorker\":20,\"foodWorker\":30,\"army\":0,\"attacking\":false,\"underAttack\":false} 0"],"SeqNumber":24}\n”

Caso de uso (jogo)

Descrição do jogo e suas regras

O jogo é composto por diversas ilhas, onde cada jogador que ingressar, receberá inicialmente uma ou mais ilhas, podendo expandir para outras com o avanço de seu império. Cada ilha possui uma série de recursos, de quantidades aleatórias e limitadas, portanto com o passar do tempo estes se esgotam e forçam jogadores a procurarem novas ilhas e/ou roubarem de outros players. Ilhas possuem um limite de população, assim caso esta seja alcançado, o jogador terá de expandir seu império em outra ilha para continuar crescendo sua população. Os recursos explorados na ilha serão recolhidos com velocidade dependente do número de pessoas que foram alocadas para função, como é comum em jogos de RTS (Estratégia em tempo real). Quanto maior a comida, maior a taxa de crescimento da população, que também possuem um limite de pessoas, independente da função que esta ocupe.

Uma pessoa só pode exercer uma função da ilha de cada vez, podendo ser guerreiro ou operário. Caso seja guerreiro este terá seus status de acordo com o nível de exército de seu império, cada nível do exército poderá ser comprado com ouro e mais alguns recursos que podem ser extraídos da ilha e isto irá fortalecer todo o exército. Um exército de nível maior possuíra melhores armas e armaduras e com isso a chance de vencer um ataque ou defesa aumentam.

Os ataques podem ser realizados a qualquer momento e levam um turno até serem concluídos, durante o tempo do ataque as tropas alocadas ficarão indisponíveis, deixando o jogador vulnerável. Podemos entender como turno um conjunto de comandos em transações de um bloco validado. Caso o ataque tenha sucesso, algumas tropas do jogador atacante e defensor serão perdidas em combate e uma porcentagem de recursos serão transferidos do perdedor para o vencedor, deixando o perdedor em um estado de não poder ser atacado por outros jogadores durante um certo tempo. Enquanto houverem recursos e espaço disponíveis, a população irá e em algum momento o limite populacional será atingido, fazendo o jogador ter de conquistar uma nova ilha. O processo de conquista de nova ilha também envolve o exército e este também ficará indisponível durante o ataque que também leva um turno.

Todas os comandos dados pelo jogador ficam armazenadas numa lista de ações até o momento que este adicionar o valor de Ethers que pretende gastar para que seus comandos sejam executados mais rapidamente. Por padrão todos os jogadores começam com 10 trabalhadores em cada recurso e uma quantia de 1000 Ethers. Ether é o recurso mais valioso do jogo, é criado a partir dos outros recursos base, necessitando de 10 de madeira e 10 de pedra para criação de um único Ether, um custo bastante alto. Não foi estabelecido um limite no número de ações por cada aposta (*Bet*) em Ether, vai da estratégia de cada jogador como melhor gastar seus recursos, com a mecânica clara de quanto mais Ethers apostados, mais rapidamente e primeiro os comandos serão executados.

Dessa forma as ações que um jogador pode tomar durante o jogo são:

* Mover a população da ilha entre exército e extração de recursos, os quais podem ser: comida, madeira e pedra.
* Atacar ilhas de outros jogadores, o qual como já dito anteriormente sempre usará todo o exército disponível.
* Explorar novas ilhas, ação que consome gastando recursos e também utilizando todo o exército.
* Criar Ethers, gastando muito recurso para isso.

O jogo é composto por três telas principais. A tela de *loading*, onde ocorrem os processos de conexão com o servidor e requisição de *game state* (Figura x). Tela de login (Figura y), onde o usuário pode realizar o cadastro e se conectar com o jogo. E por último a tela do jogo em si (Figura z).

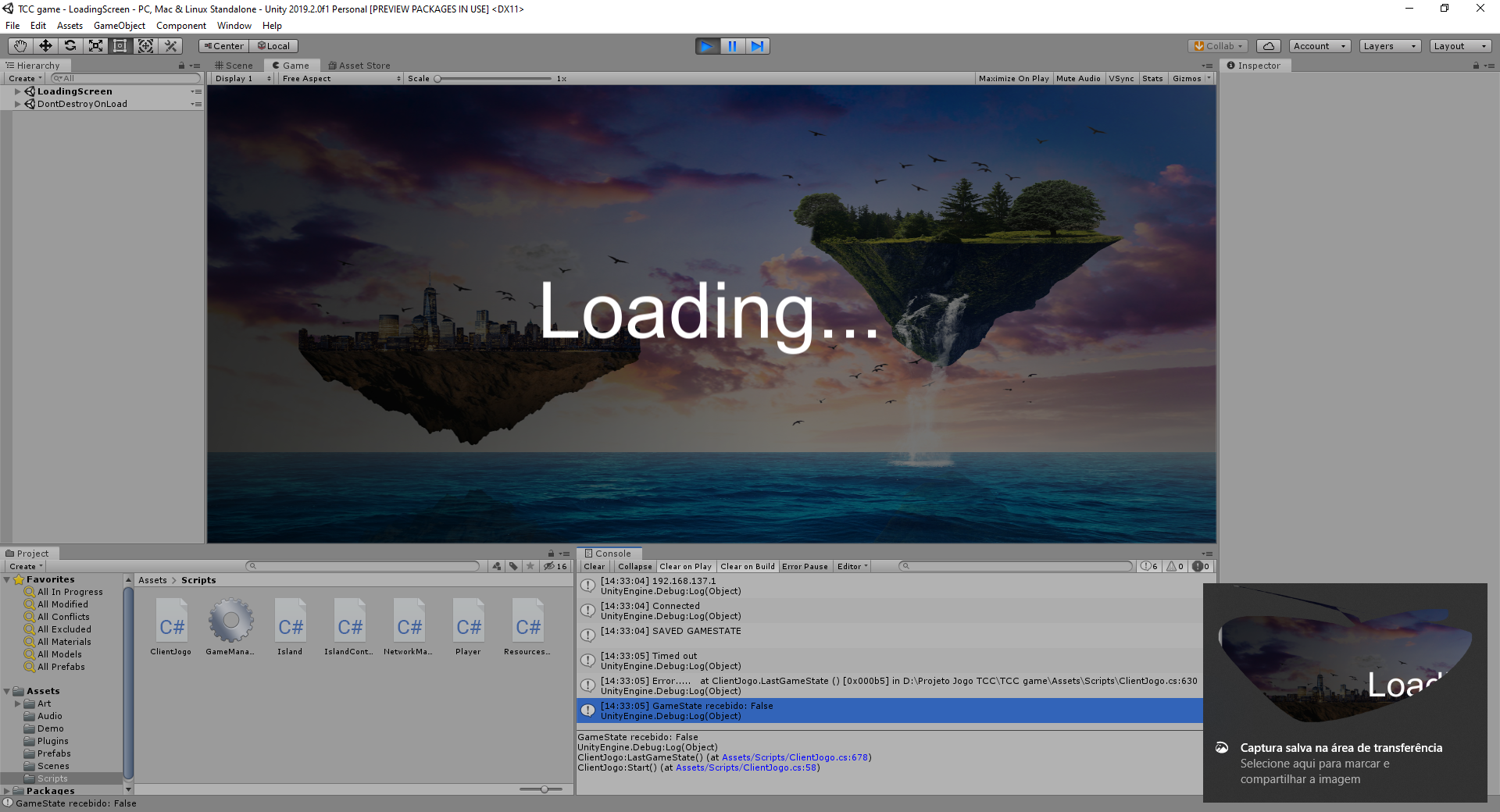


Figura x

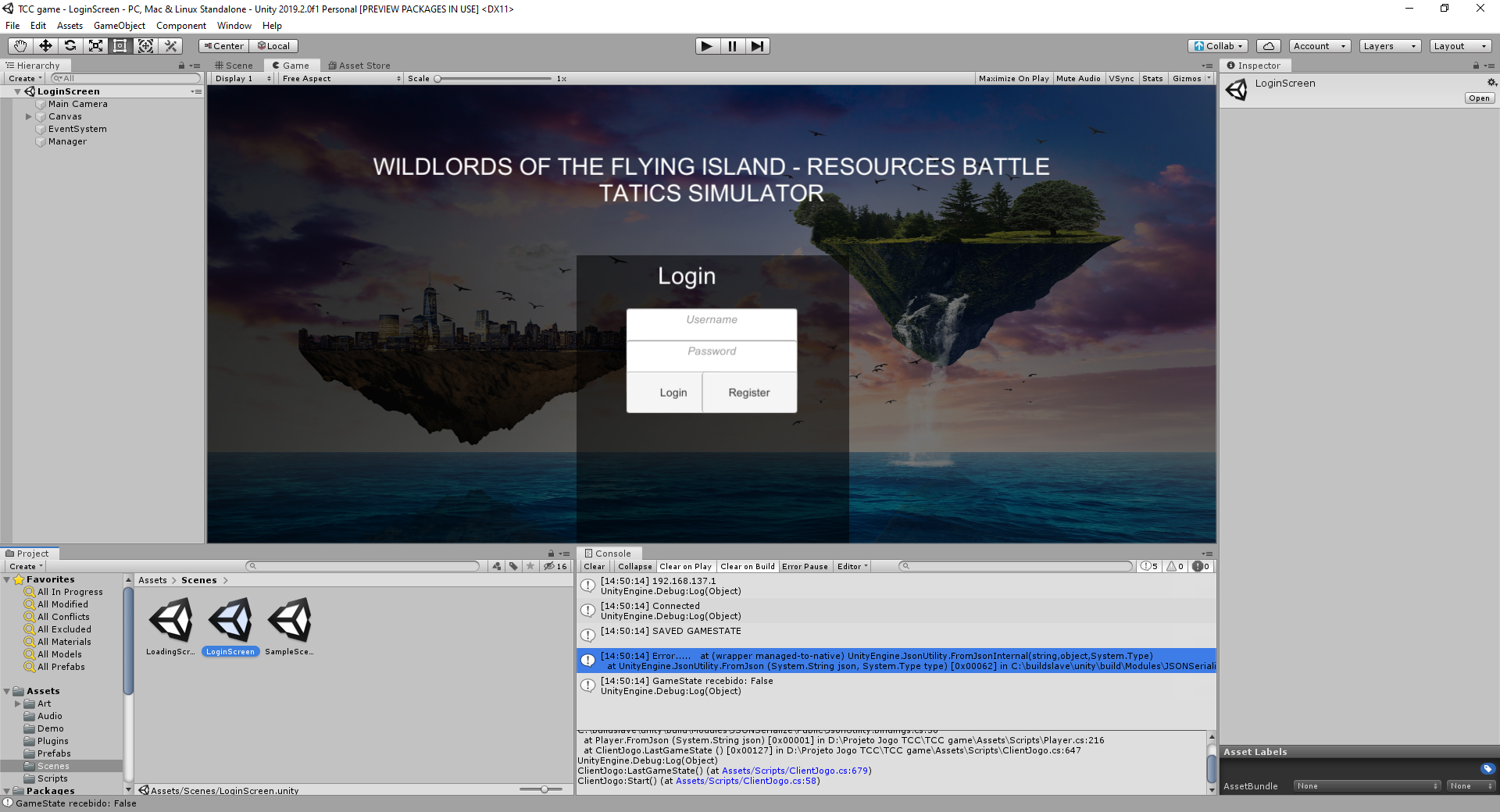


Figura y

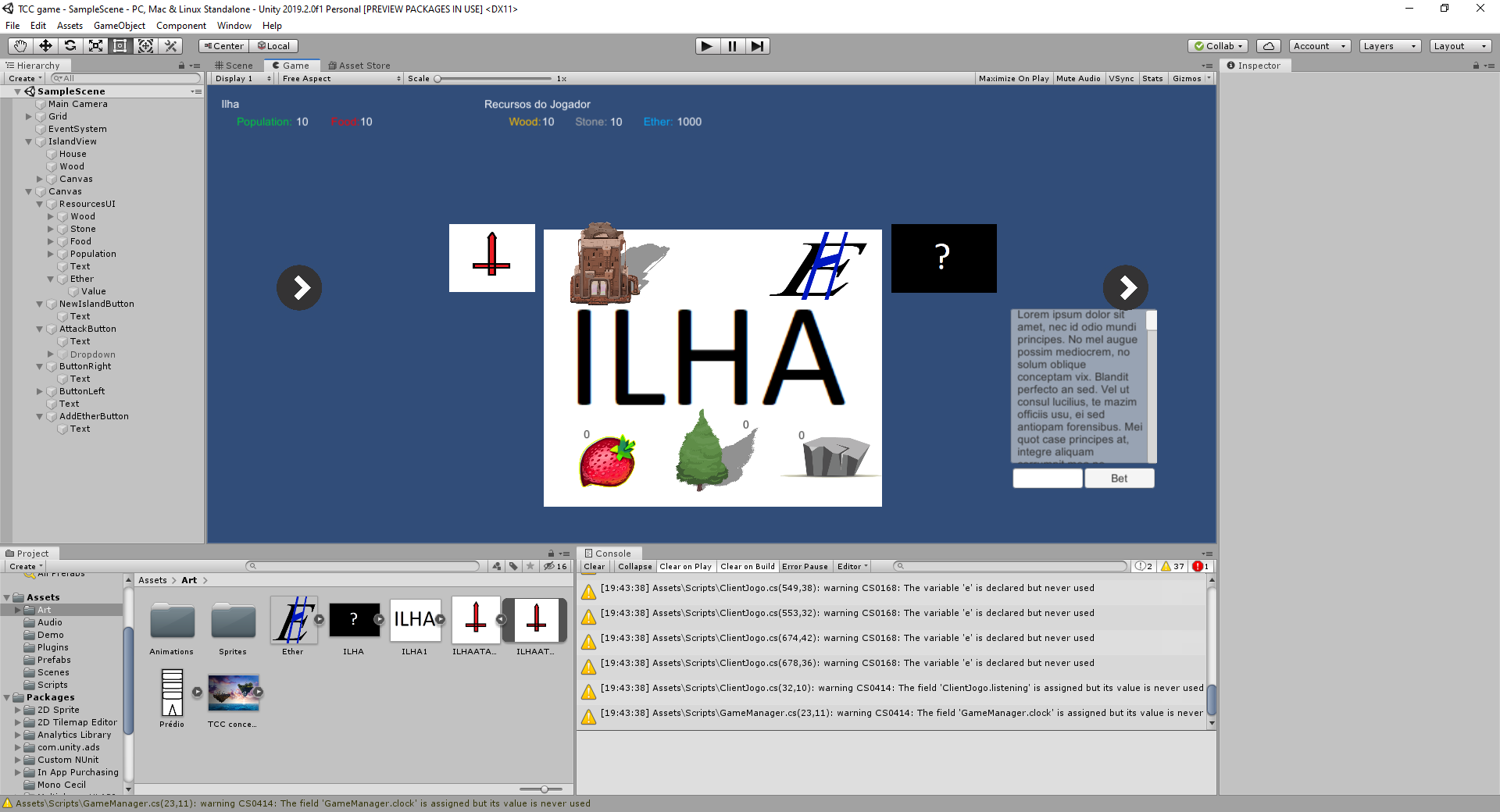


Figura z

A tela do jogo, que pode ser vista mais detalhadamente na figura w, possui botões para cada um dos comandos de jogador que foram detalhados nas seções anteriores, com uma limitação de que a quantidade é em 1 para os comandos que alteram trabalhadores e criam Ethers, com retângulos vermelhos, para fins de simplificação.



Figura w

Na figura w, o retângulo em azul é o botão executa o comando de ataque à uma ilha de algum outro jogador registrado. Os retângulos vermelhos comandos relacionados aos recursos, mudança de função da população e criação de Ethers. No retângulo branco a lista onde os comandos ficam armazenados e o campo para especificar o valor do Bet, juntamente ao botão de enviá-lo. Retângulo verde ressalta o botão que aciona o comando de procura de adição de nova ilha. Os retângulos roxos são os botões de troca de ilha, para o jogador ter a possibilidade administrar suas demais posses. E por último o retângulo amarelo representa os recursos atuais do jogador, que são compartilhados por todas as suas ilhas e da ilha, que fica restrito a cada ilha específica.

Descrição das ferramentas de implementação

Foi utilizado para criação do jogo a plataforma de desenvolvimento de jogos Unity, que utiliza como linguagem o C# e programação orientada a componentes. Esta *game engine* é uma das mais famosas do mercado, possuindo versões pagas e gratuitas, sendo utilizada por grandes desenvolvedoras como Blizzard, em seu jogo de tabuleiro Hearthstone e outros jogos que ficaram bastante populares, como Inside e Cuphead. (Referenciar Site <https://unity3d.com/pt/games-made-with-unity>).

A Unity possui várias ferramentas permitindo um rápido desenvolvimento com seus modos de Play para pré-visualização do projeto em tempo real. Agumas das principais possibilidades oferecidas pela Unity são (https://unity3d.com/pt/unity):

* Tudo em-um: Possuí diversas ferramentas artísticas para design de mundos de jogos, ferramentas de desenvolvedor para lógica e jogabilidade, além de ser multiplataforma, disponível para Windows, Mac e Linux.
* 2D e 3D: Suporta o desenvolvimento tanto de jogos 2D como 3D.
* Ferramentas de IA: Permite a criação de NPCs (personagens controlados pela máquina), que se movem e tomam ação de forma inteligente.
* Fluxo de trabalho: Os chamados Unity *prefabs* são objetos de jogo (*Game Objects*) previamente configurados, minimizando eventuais erros e agilizando o desenvolvimento.
* Interfaces de usuário: Sistema de UI (Interface de Usuário) auxilia na criação das interfaces de forma mais rápida.
* *Engine* de física: Sistema de física já implementado, o Box2D, baseado em DOTS, com suporte para NVIDIA PhysX.
* *Asset* *Store*: Possibilidade de encontrar recursos, ferramentas e extensões na loja disponível na plataforma.

Resultados de testes

Conclusões e trabalhos futuros

A implementação realizada atingiu o seu objetivo. O servidor foi reduzido para rodar em uma rede local com algumas máquinas que conseguiram sincronizar seus dados e funcionarem simultaneamente sem a necessidade do servidor central.

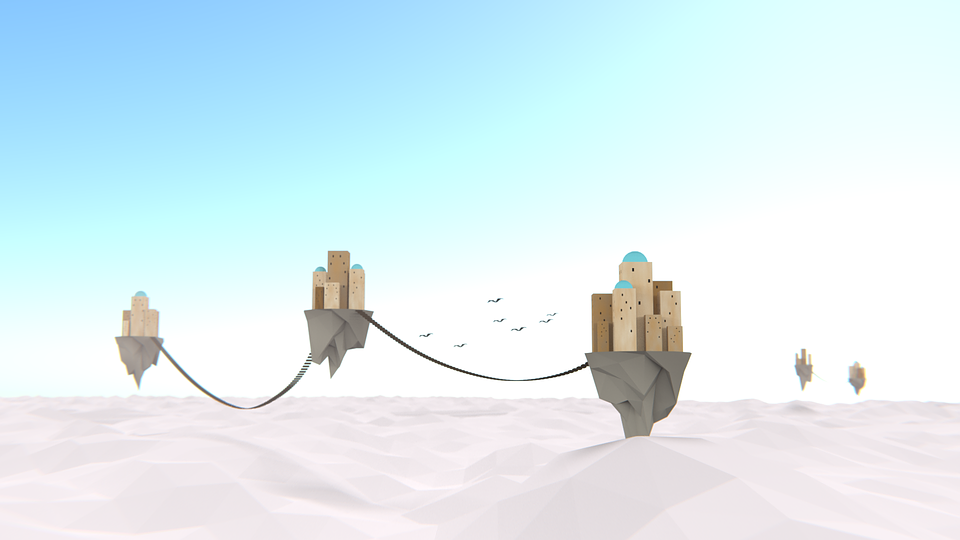
O trabalho apresentou um grande desafio no momento de modelar o esquema de um jogo em tempo real para um sistema da Blockchain, que possui um atraso na validação de seus blocos. Desta forma o tempo real é simulado, todas as ações podem ser feitas a qualquer momento pelo jogador, porém levam um tempo até serem concluídas, que vai depender da taxa de forjamento dos blocos. Sendo portanto uma mistura de jogo de turnos e tempo real. Muitas dificuldades surgiram no desenvolvimento por se tratar de uma nova linguagem de programação e falta de experiência com a ferramenta da Unity, principalmente no controle de tráfego de mensagens do servidor *peer-to-peer*. Contudo foi possível concluir a implementação de forma satisfatória e a solução parece viável para alguns tipos de jogos, como o feito para o caso de uso. Jogos que necessitam de um consenso rápido dos dados, como jogos de tiro, não são viáveis nesta implementação por este atraso comprometer a experiência de jogo.

Uma melhoria que poderia ser feita na implementação é a retirada dos *gamestates* de dentro do bloco, criando assim duas estruturas, a Blockchain com os comandos dos jogadores, e um arquivo, ou banco de dados de qualquer tipo, contendo o *gamestate* atual do jogo. Com isso o tamanho do bloco, e o armazenamento da solução no geral, seria reduzido drasticamente e como haveria o último *gamestate* salvo, o *loading* do mundo do processo jogo não seria prejudicado. Apenas sendo problemático em caso de ser necessário fazer a regressão do jogo até um estado anterior na cadeia, sendo necessário reprocessar todos os comandos desde o *genesis block*.

Outro incremento que poderia ser adicionado seria o uso de criptografia assimétrica para o cadastro dos jogadores, enviando a chave pública para os demais servidores ao invés da string de usuário como é feito atualmente. Além de adicionar uma certa anonimidade, assegurar a não repetição de nomes de jogadores, poderia ser feito a assinatura das transações criadas, e das mensagens enviadas, adicionando muito mais segurança na aplicação.

Na parte do jogo muitas melhorias poderiam ser realizadas, sendo criado apenas para ser um caso de teste. O jogo é praticamente um pré-alfa, opções como escolher quantidade de trabalhadores para alternar de função, quantidade de Ethers que serão criados, melhorar os botões e *layout* da interface, criar novos e melhores *assets* que se adequem à ideia do jogo, seriam apenas algumas das muitas melhorias.





Referências

1. I. Bashir, *Mastering Blockchain,* Birmingham, UK: Packt Publishing, Março 2017
2. F. Tschorsch, B. Scheuermann, “Bitcoin and Beyond: A Technical Survey on Decentralized Digital Currencies”, *IEEE Communications Surveys & Tutorials Journal*, vol.18, no. 3, pp. 2084-2123, Agosto 2016.
3. S. King e S. Nadal, “PPCoin: Peer-to-Peer Crypto-Currency with Proof-of-Stake”, pp. 1-6, Agosto 2012
4. S. Nakamoto, (2008) “Bitcoin: A Peer-to-Peer Electronic Cash System,”, [Online]. Available: https://bitcoin.org/bitcoin.pdf.
5. P. G. Dwyer, “The Economics of Bitcoin and Similar Private Digital Currencies”, *Journal of Financial Stability*, Vol. 17, pp. 81-91, Abril 2015.
6. D. Kraft, “Difficulty Control for Blockchain-Based Consensus Systems”, *Peer-to-Peer Networking and Applications*, vol. 9, pp. 397–413, Abril 2015.
7. <https://golang.org/doc/>
8. N. Houy, “It will cost you nothing to ‘kill’ a proof-of-stake cryptocurrency,” Econ. Bull., vol. 34, no. 2, pp. 1038–1044, 2014.
9. P. G. Dwyer, ―The Economics of Bitcoin and Similar Private Digital Currencies‖, Journal of Financial Stability, Vol. 17, pp. 81-91, Abril 2015.
10. Kakavand, Hossein and Kost De Sevres, Nicolette and Chilton, Bart, The Blockchain Revolution: An Analysis of Regulation and Technology Related to Distributed Ledger Technologies (January 1, 2017). Available at SSRN: <https://ssrn.com/abstract=2849251> or [http://dx.doi.org/10.2139/ssrn.2849251](https://dx.doi.org/10.2139/ssrn.2849251)
11. Wright, Aaron and De Filippi, Primavera, Decentralized Blockchain Technology and the Rise of Lex Cryptographia (March 10, 2015). Available at SSRN: <https://ssrn.com/abstract=2580664> or [http://dx.doi.org/10.2139/ssrn.2580664](https://dx.doi.org/10.2139/ssrn.2580664)
12. S. Meiklejohn, M.i Pomarole, G. Jordan, K.l Levchenko, D. McCoy, G. M. Voelker, S. Savage, ―A Fistful of Bitcoins: Characterizing Payments among Men with No Names‖, Communications of the ACM, vol.. 59, No. 4, pp. 86-93, abril 2016.
13. Banking on Bitcoin. Direção: Christopher Cannucciari. Produção: Christopher Cannucciari, David Guy Levy. EUA, 2016.Documentário (90 min.).
14. X. Li, C. A. Wang, “The Technology and Economic Determinants of Cryptocurrency Exchange Rates: The Case of Bitcoin”, *Decision Support Systems*, vol. 95, pp. 49-60, Março 2017. A1 (CC)