Universidade Federal do Espírito Santo - UFES

Centro Universitário Norte do Espírito Santo - CEUNES

Departamento de Computação e Eletrônica

Bacharel em Ciência da Computação

Caio Vianna Rizzo

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

Trabalho de Conclusão de Curso

São Mateus - ES

2019

Caio Vianna Rizzo

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

Trabalho de Conclusão de Curso apresentado ao curso de Curso de Graduação de Ciência da Computação, como parte dos requisitos necessários à obtenção do título de Graduando de Ciência da Computação.

Orientador: Wilian Hiroshi Hisatugu

São Mateus - ES
2019

Caio Vianna Rizzo

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

IMPORTANTE: ESSE É APENAS UM TEXTO DE EXEMPLO DE FOLHA DE APROVAÇÃO. VOCÊ DEVERÁ SOLICITAR UMA FOLHA DE APROVAÇÃO PARA SEU TRABALHO NA SECRETARIA DO SEU CURSO (OU DEPARTAMENTO).

Trabalho aprovado. São Mateus - ES:

Wilian Hiroshi Hisatugu
Orientador

Professor
Convidado 1

Professor
Convidado 2

São Mateus - ES

2019

Dedico este trabalho a todos aqueles que me ajudaram de alguma forma, as vezes apenas uma palavra ou pequeno gesto é o suficiente para motivar alguém.

Agradecimentos

Quero agradecer a minha família, minha namorada, meus amigos e professores que me incentivaram durante o processo.

Em especial ao meu Orientador professor doutor WIlian que me deu todo o suporte e ao professor doutor Flávio pela ideia do trabalho.

“Os limites só existem se você os deixar existir.”
(Son Goku)

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 na 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.

Palavras-chave: Blockchain, proof-of-stake, servidor distribuído, jogo.

Abstract

Blockchains and games are two markets that are gaining more space each year. Since its inception in 2009, Blockchain has been a disruptive technology in the payments and virtual currencies industry with Bitcoin. It is a fully distributed and secure system, without the need for transaction validation entities. Its hash pointer structure, together with the proof-of-work consensus system, forces the computers that make up the system to perform very costly computational proofing, making saved data virtually unchanged. This paper suggests a different approach to this revolutionary technology using it as an online multiplayer game server. Each player becomes a server in this network and thus eliminates the need for central servers controlled by private companies. Games created on this type of server remain active with at least one player connected. In this implementation, we used a consensus protocol that is gaining more notoriety today, the proof-of-stake. This uses a variable block difficulty value depending on the total score of the block, eliminating the high computational expense of proof-of-work and giving greater security considering the dedication of nodes to the system when choosing the validator. An application-specific communication protocol was developed, which was developed in Golang and a test game in Unity.

Keywords: Blockchain, proof-of-stake, distributed server, game.

Lista de ilustrações

Sumário

1 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 PWC (2018) na 19ª Pesquisa Global de Entretenimento e Mídia, 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. 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 (Denied of Service), o qual é bem comum, os jogadores ficarão impossibilitados de jogar (YAHYAVI; KEMME, 2013; CONTI et al., 2018).

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 Yahyavi e Kemme (2013) aborda um conjunto de desafios de projeto como, problemas em escalabilidade, atraso da rede, segurança dos dados, entre outros.

Em 2008, foi proposto no *white paper* (NAKAMOTO, 2008), sob pseudônimo de Satoshi 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. Para um sistema que necessita que todos os usuários visualizem o mesmo estado, como é o caso de um jogo multiplayer, ele é crucial. 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, o chamado *proof-of-work* (PoW). Embora ofereça segurança à integridade de dados, o PoW tem um alto custo computacional (TANENBAUM; STEEN, 2007; YAHYAVI; KEMME, 2013; NAKAMOTO, 2008).

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, o qual está sendo implementado para substituir o PoW no principal concorrente do Blockchain, o ETHEREUM (2019). 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*) (TSCHORSCH; SCHEUERMANN, 2016).

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. Este sistema 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 (TSCHORSCH; SCHEUERMANN, 2016).

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.

2 Objetivos

Este trabalho irá conter, após uma breve revisão bibliográfica, uma implementação de Blockchain como servidor, protocolo de consenso *proof-of-stake* e um um jogo multiplayer simples em Unity como caso de uso.

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. 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.

2.1 Objetivos Específicos

- Implementar uma blockchain com sistema de consenso proof of stake (PoS) para ser utilizada como servidor.
- Implementar um jogo teste multiplayer online (MMO) de estratégia.
- Analisar a viabilidade da solução em termos de armazenamento, integridade, ordenação e recuperação dos dados.

3 Revisão Bibliográfica

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.

3.1 Sistemas Distribuídos

> "Um sistema distribuído é um conjunto de computadores independentes que se apresenta a seus usuários como um sistema único e coerente". (TANENBAUM; STEEN, 2007)

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; STEEN, 2007):

- 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 (YAHYAVI; KEMME, 2013; TANENBAUM; STEEN, 2007).

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.

3.1.1 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

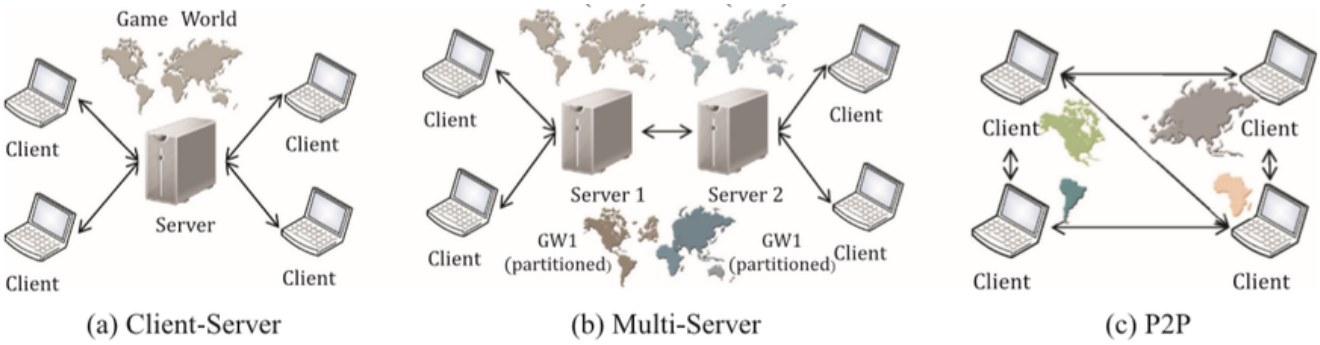
protocolo são sintaxe, semântica e temporização (*timing*) e serão explicados abaixo (FOROUZAN, 2010):

- 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, 2009; FOROUZAN, 2010).

3.1.2 Arquitetura de servidores

Figura 1 – Arquiteturas possíveis



Fonte: Amir Yahyavi (2013)

Existem diversas arquiteturas possíveis um servidor, os principais tipos são os mostrados na 1: Client-Server (cliente-servidor), Multi-Server e peer-to-peer (P2P) (YAHYAVI; KEMME, 2013).

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 (YAHYAVI; KEMME, 2013).

3.1.3 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 2 (TANENBAUM; STEEN, 2007).

Figura 2 – Modelo de envio de mensagens Cliente-Servidor

Fonte: Tanenbaum (2007)

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; STEEN, 2007).

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 (YAHYAVI; KEMME, 2013).

3.1.4 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 (TANENBAUM; STEEN, 2007).

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 (TANENBAUM; STEEN, 2007).

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; STEEN, 2007).

As fornas de disseminação de atualizações em servidores *peer-to-peer* são diversas, abaixo segundo Yahyavi e Kemme (2013), 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.

3.1.5 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 (KAKAVAND; SEVRES; CHILTON, 1, 2017).

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 (KAKAVAND; SEVRES; CHILTON, 1, 2017).

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 (KAKAVAND; SEVRES; CHILTON, 1, 2017).

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 (KAKAVAND; SEVRES; CHILTON, 1, 2017; NAKAMOTO, 2008).

3.2 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. 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 (NAKAMOTO, 2008; TSCHORSCH; SCHEUERMANN, 2016; KAKAVAND; SEVRES; CHILTON, 1, 2017).

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 . 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 (KRAFT, 2016; KAKAVAND; SEVRES; CHILTON, 1, 2017; WRIGHT; FILIPPI, 10, 2015).

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, 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. 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 (NAKAMOTO, 2008; DWYER, 2015; KRAFT, 2016).

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. 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 (NAKAMOTO, 2008; KAKAVAND; SEVRES; CHILTON, 1, 2017).

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.

3.2.1 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..., 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 (TSCHORSCH; SCHEUERMANN, 2016; NAKAMOTO, 2008; BANKING..., 2016).

O acerto 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 (DWYER, 2015; BANKING..., 2016).

O modelo bancário resolve este problema utilizando números de série controlados pela instituição e proibindo o processamento de transações concorrentes. 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. 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 (MEIKLEJOHN et al., 2016; TSCHORSCH; SCHEUERMANN, 2016; STRASSEL, 1996).

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 (TSCHORSCH; SCHEUERMANN, 2016; NAKAMOTO, 2008).

3.2.2 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 na Blockchain. E protocolos de autenticação, para verificar identidades de entidades (NARAYANAN et al., 2016; STALLINGS, 2008).

3.2.2.1 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), três 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, 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 (STALLINGS, 2015).

3.2.2.2 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 (STALLINGS, 2015).

3.2.2.2.1 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. Neste tipo de encriptação a chave secreta é compartilhada pelo emissor e receptor da mensagem, pois esta deve ser a mesma para ambos. Segundo Stallings (2015), 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.

Figura 3 – Esquema de criptografia simétrica com chave compartilhada

Fonte: Forouzan (2010)

3.2.2.2.2 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 4, 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 4 – Esquema de criptografia de chave assimétrica

Fonte: Forouzan (2010)

Algoritmos de encriptação assimétrica, segundo Stallings (2015) possuem a seguintes característica principais:

- 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.

Ainda segundo Stallings (2015), para a utilização destes algoritmos, devem ser seguidas as seguintes etapas:

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).

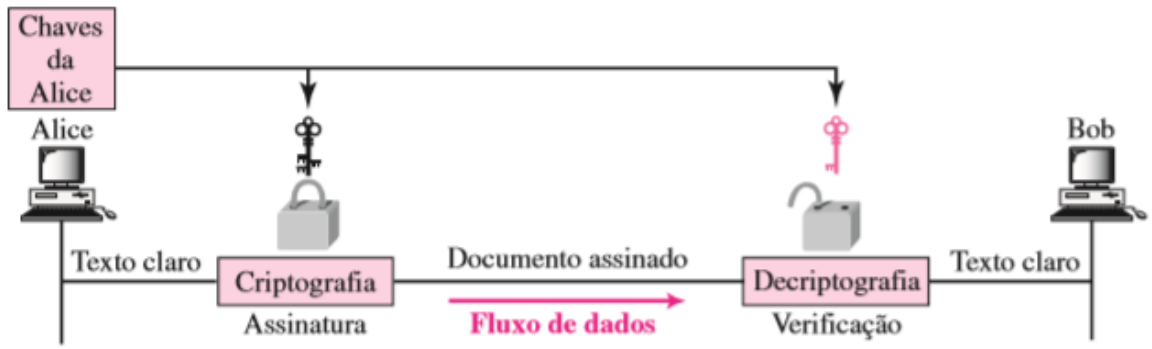
A grande vantagem de se utilizar esta técnica está no fato de todas as chaves terem sido geradas localmente, e apenas as chaves públicas precisam ser distribuídas.

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 (STALLINGS, 2015).

3.2.2.3 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 5 – Esquema de assinatura digital utilizando criptografia assimétrica



Fonte: Forouzan (2010)

3.2.2.4 *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 (DWYER, 2015):

- Dado um M, é fácil computar h.
- Dado um h, é difícil computar M, onde H(M) = h.
- 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 segundo Stallings (2015) são:

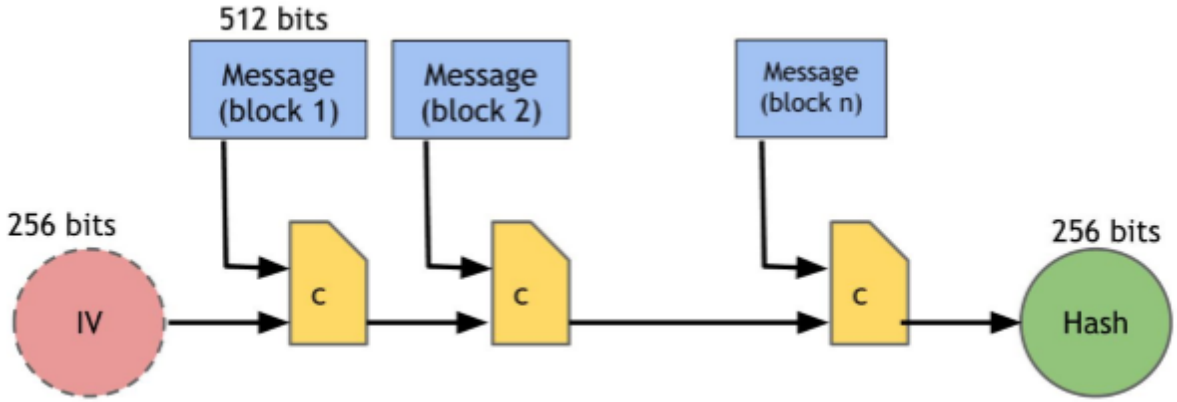
- Autenticação de mensagem: é um mecanismo usado para verificar a integridade de uma mensagem quando esta chega ao receptor, garantindo que os dados recebidos não foram alterados durante o transporte. Nesses casos o valor da função de hash é chamado de resumo da mensagem. Este resumo é calculado pelo 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 2ˆ69 operações, menos do que as 2ˆ80 previstos, no SHA-1 (STALLINGS, 2015).

Figura 6 – Funcionamento resumido do SHA-256



Fonte: Narayanan et al (2016)

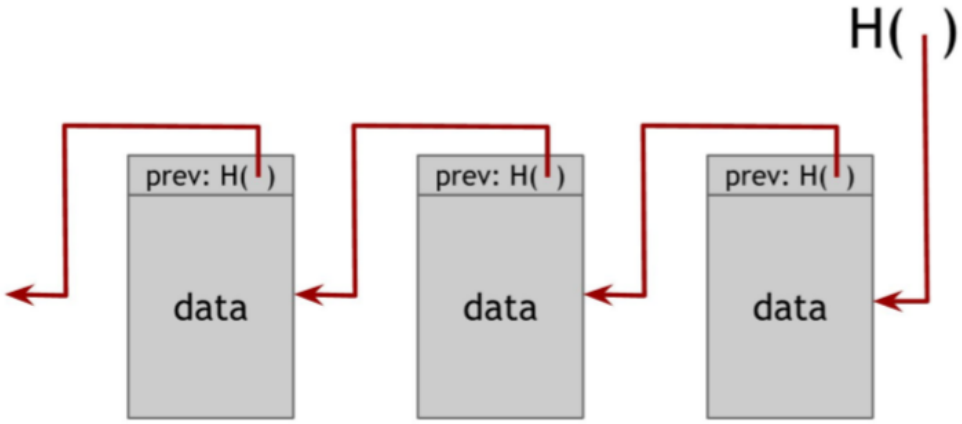
A 6 apresenta o funcionamento do SHA-256. De forma resumida, a mensagem inicial, que pode ser de qualquer tamanho, é dividida em blocos de 512 bits. Cada bloco é passado juntamente com a saída do bloco anterior a uma função de compressão até a saída final em um hash de 256 bits. No bloco inicial, como 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 (NARAYANAN et al., 2016).

3.2.3 *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 pointers. 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” (NARAYANAN et al., 2016).

A 7 ilustra um exemplo de lista de blocos com *hash pointers*, uma estrutura de blockchain básica.

Figura 7 – Estrutura de *blockchain*



Fonte: Narayanan et al (2016)

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* (NARAYANAN et al., 2016).

3.2.4 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* (NAKAMOTO, 2008).

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 (DWYER, 2015).

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 do Bitcoin 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 (TSCHORSCH; SCHEUERMANN, 2016).

3.2.5 Proof of Work (PoW)

Como Nakamoto (2008) explica no whitepaper do Bitcoin, a *proof of work*, descrita por ele é similar ao Hashcash do Back (2002) . 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*. 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. 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 (DAI, 1998; NAKAMOTO, 2008; TSCHORSCH; SCHEUERMANN, 2016).

Com isso a dificuldade dos mineradores não é computar o *hash*, que é fácil, mas encontrar o *hash* adequado através de uma busca de força bruta (*brute-force*). 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* (DWYER, 2015; KRAFT, 2016).

A função de hash utilizada pelo bitcoin é a seguinte (BASHIR, 2017):

$$H(N||P_{_{hash||Tx||Tx||...Tx)<Target}} \tag{3.1}$$

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 (BASHIR, 2017).

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 (BASHIR, 2017):

$$Target = Previoustarget * time/2016 * 10min \tag{3.2}$$

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 (BASHIR, 2017).

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 (KRAFT, 2016).

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* (DWYER, 2015).

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 (NAKAMOTO, 2008; TSCHORSCH; SCHEUERMANN, 2016).

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 (TSCHORSCH; SCHEUERMANN, 2016).

3.2.6 *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 (BASHIR, 2017; TSCHORSCH; SCHEUERMANN, 2016).

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. 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 (TSCHORSCH; SCHEUERMANN, 2016; KING; SCOTT NADAL, 2012).

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 (TSCHORSCH; SCHEUERMANN, 2016).

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, como é no PoW. 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, não existindo vantagens em sabotar e com isso, diminuir a confiança na criptomoedas em questão (KING; SCOTT NADAL, 2012; TSCHORSCH; SCHEUERMANN, 2016).

4 Arquitetura Proposta

4.1 Arquitetura da Blockchain

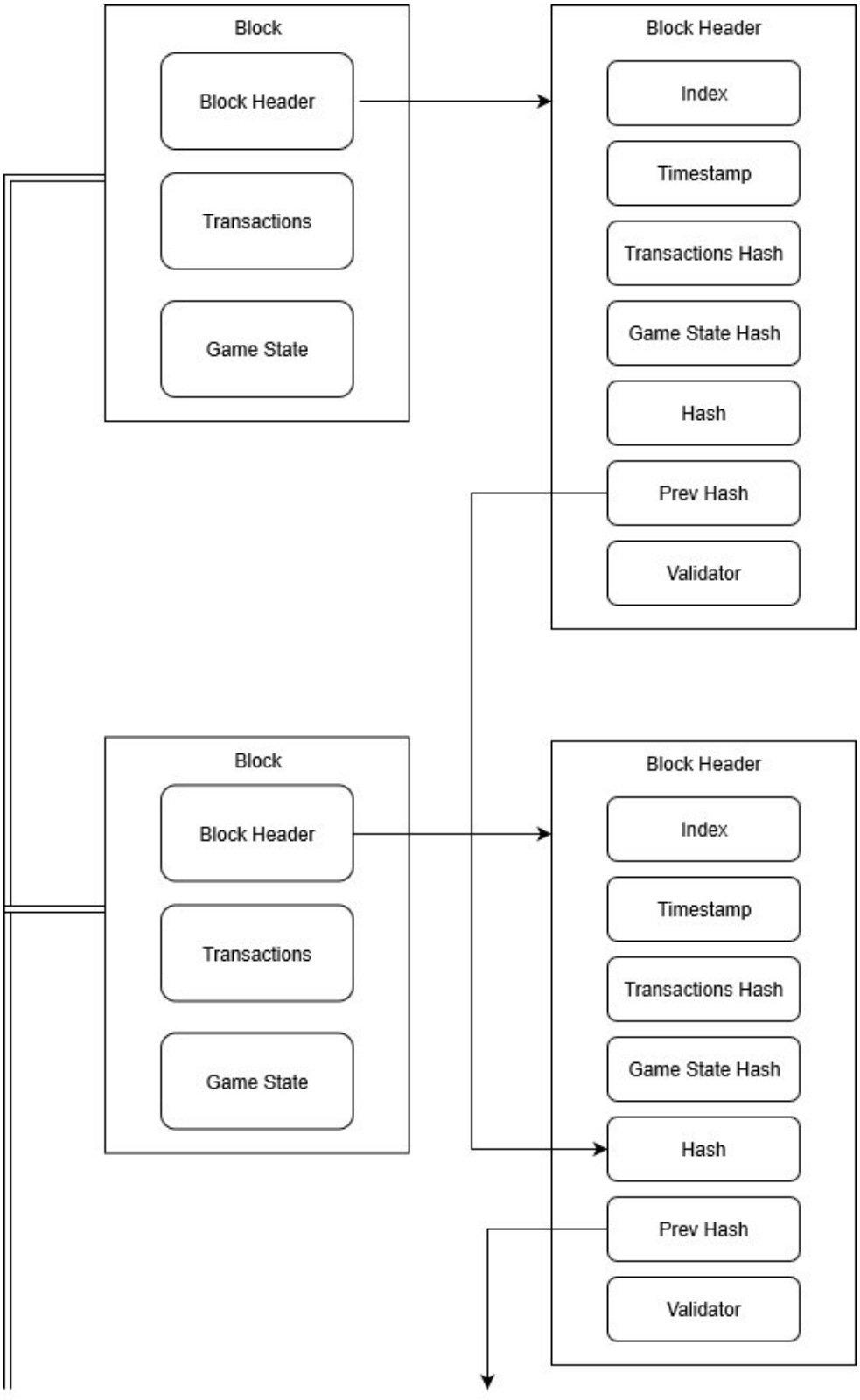
A Blockchain proposta é formada por duas estruturas principais, o BlockHeader e o Block, como pode ser visto na 8. O BlockHeader é onde são inseridas informações de cabeçalho do bloco e as hashes das informações contidas no Bloco. Os campos da BlockHeader consistem em:

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.
4) TransactionsHash: um hash que consiste em uma codificação dupla, primeiro o vetor de transações é codificado em JSON, 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.
5) GameStateHash: consiste no hash, em SHA-256, do JSON 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, 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.
3) GameState: JSON contendo o estado do jogo no início da criação daquele bloco.

Figura 8 – Arquitetura dos blocos



Fonte: Caio Rizzo (2019).

Uma transação no modelo foi estabelecida como uma estrutura para agrupar os comandos de um determinado jogador (*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 assim evitar que 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:

1) Player: campo contendo o nome do player que gerou os comandos que serão inseridos na Blockchain.
2) SeqNumber: campo com o número de sequência daquela transação.
3) 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.

4.2 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.

4.2.1 Processos gerais da Blockchain

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, calculo que está descrito na seção 4.2.2.

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 jogador esteja registrado no jogo e seja válido;
2) Se a mensagem não é do próprio jogador logado.
3) O número de sequência ser maior ou igual ao último conhecido pelo servidor ou 0 caso seja um comando de registro;

4.2.2 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:

$$Dificuldadedobloco = const - s - s * \frac{stack}{10} \tag{4.1}$$

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, realizadas a cada segundo, um novo hash seja encontrado por um servidor.

Assim como em um PoW clássico, visto na seção 3.2.5, 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:

$$H(Index||GameStateHash||TransactionsHash||PrevHash||Validator||Seconds) < Dificuldadedo \tag{4.2}$$

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

Utiliza para os cálculos os campos estáticos do cabeçalho, já anteriormente especificados na seção 4.1 e Seconds que representa os segundos do campo de *timestamp*, sendo o único campo variável de todo o cálculo do *hash*.

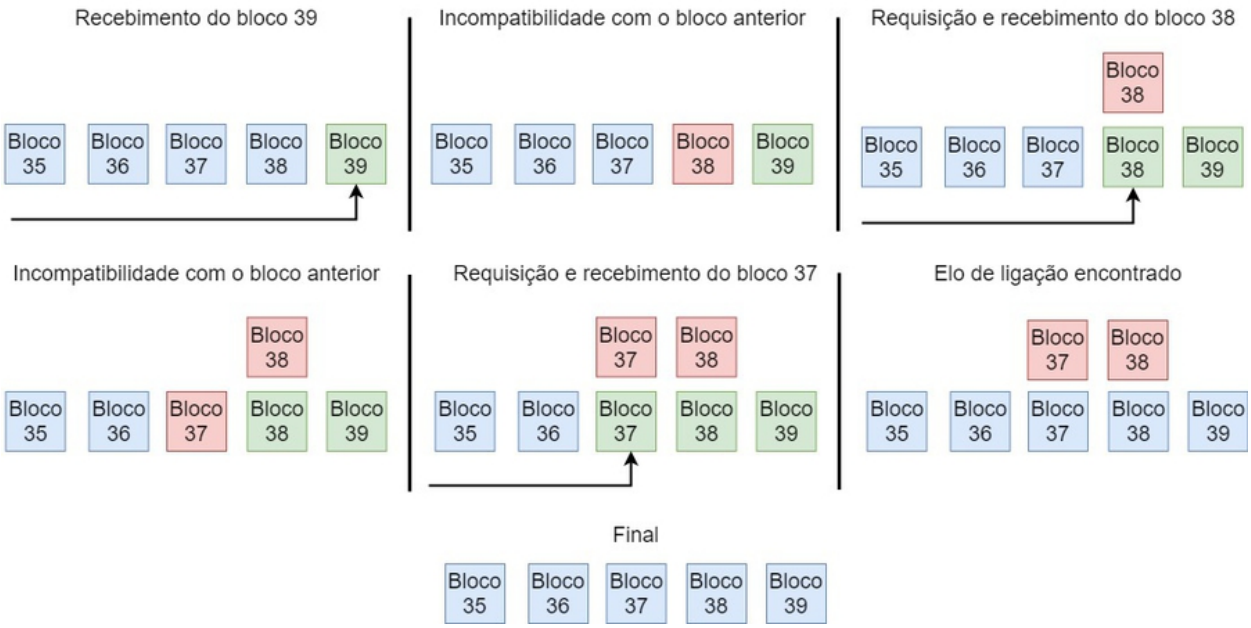
4.2.3 Processo de recuperação da cadeia principal

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 *chain* (cadeia) 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 9, onde neste exemplo o elo de ligação é o bloco 37.

Figura 9 – Inserção do bloco 39 em uma Blockchain com processo de recuperação da cadeia principal.



Fonte: Caio Rizzo (2019)

As transações contidas nos blocos substituídos são inseridas novamente na *pool* de transações e 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.

4.3 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.

4.3.1 Processo com troca de mensagem entre servidores

4.3.1.1 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 4.4.2.2.1 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 4.4.2.2.3 do protocolo, para solicitar todos os blocos à partir do último bloco salvo, se existirem, ou uma mensagem BLOCK ALL, também na seção 4.4.2.2.3 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, seção 4.4.2.2.1 do protocolo, aos servidores conectados e dessa forma será retirado do map de servidores conectados.

4.3.1.2 Flooding de transações e blocos

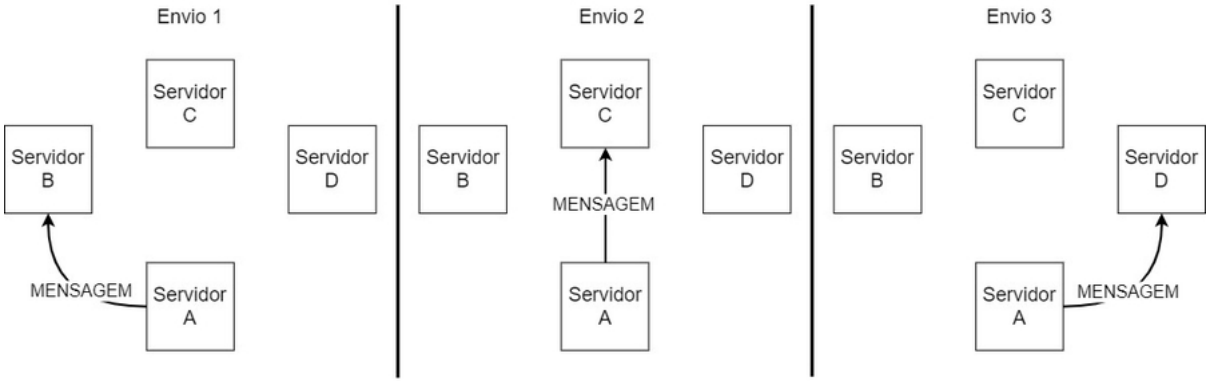
A inundação, ou flooding em inglês, é feita através de um broadcast para todos os servidores conectados, enviando uma mensagem, que no caso da implementação pode ser do tipo TRANSACTION, para o caso de uma transação e para blocos ou uma mensagem tipo WIN que anuncia um bloco vencedor.

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 de sintaxe descrito no protocolo de comunicação seção 4.4.2.2.4, e enviadas para os

servidores conectados uma a uma, fazendo um broadcast de mensagens através de uma conexão TCP/IP, conforme a figura 10 mostra. Onde o servidor A faz o broadcast de uma mensagem para os servidores B, C e D conectados a ele.

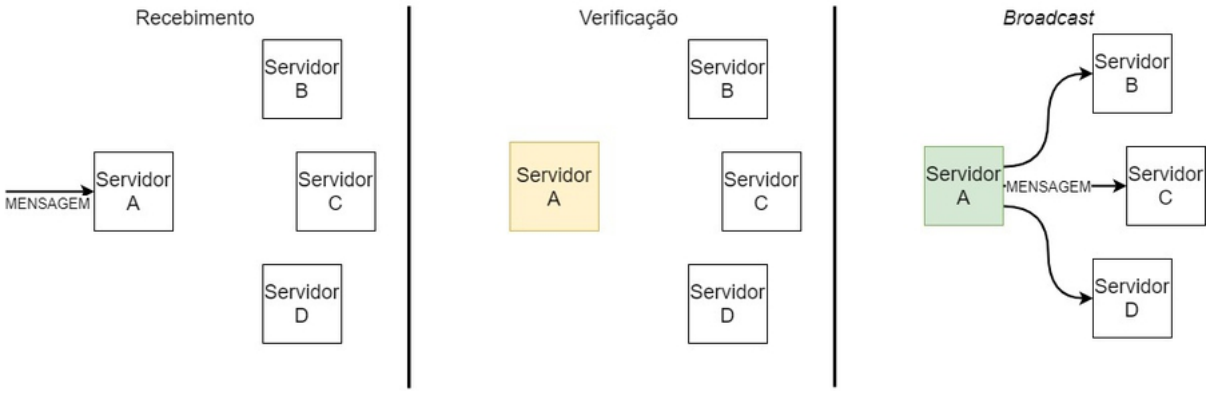
Figura 10 – Envio de mensagens por *broadcast*



Fonte: Caio Rizzo (2019)

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. A figura 11 ilustra este processo.

Figura 11 – Visão geral do processo de inundação



Fonte: Caio Rizzo (2019)

4.3.2 Processos com troca de mensagens entre jogo e servidor

4.3.2.1 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 4.4.2.1.4. Caso exista um game state salvo, ele será enviado com uma mensagem de GAMESTATE, também na seção 4.4.2.1.4, caso contrário uma mensagem NOT FOUND, seção do protocolo 4.4.2.1.2, 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.

4.3.2.2 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 4.4.2.1.1 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.

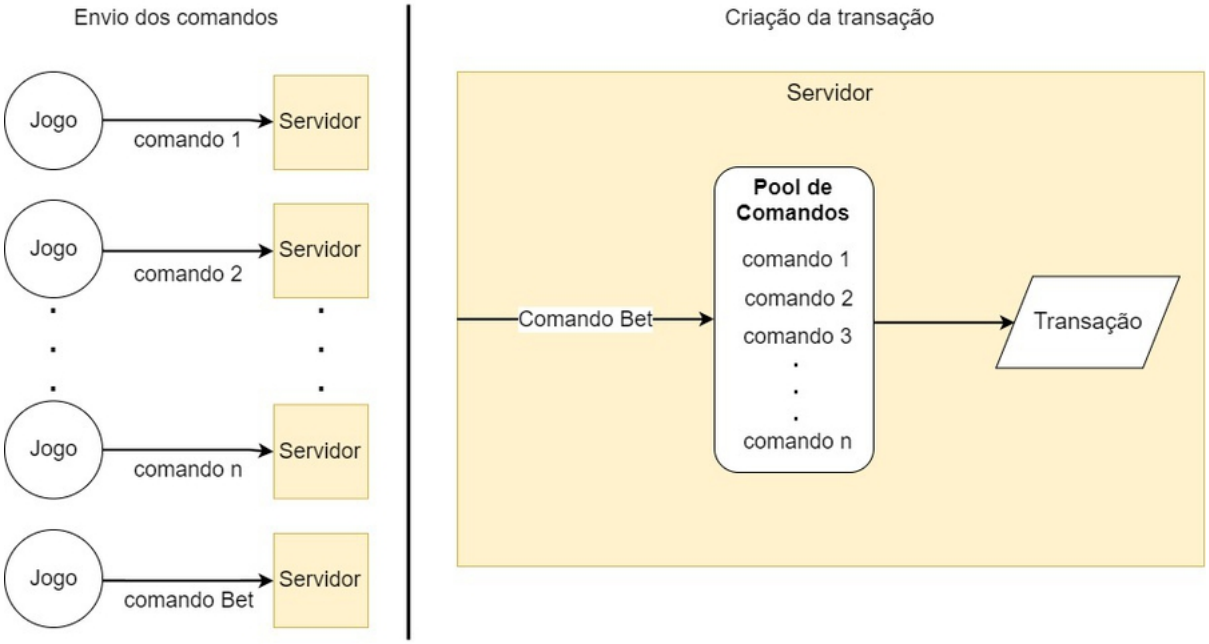
O registro de novos jogadores é um pouco mais complexo, a mensagem padrão de envio está descrita na seção 4.4.2.1.1, em vez de dados cadastrais, o processo jogo cria um novo objeto Player, codifica em JSON e então envia para o servidor. Ao receber a mensagem o servidor irá verificar algum erro na codificação 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.

4.3.2.3 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 4.4.2.1.3, 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 12 mostra o processo:

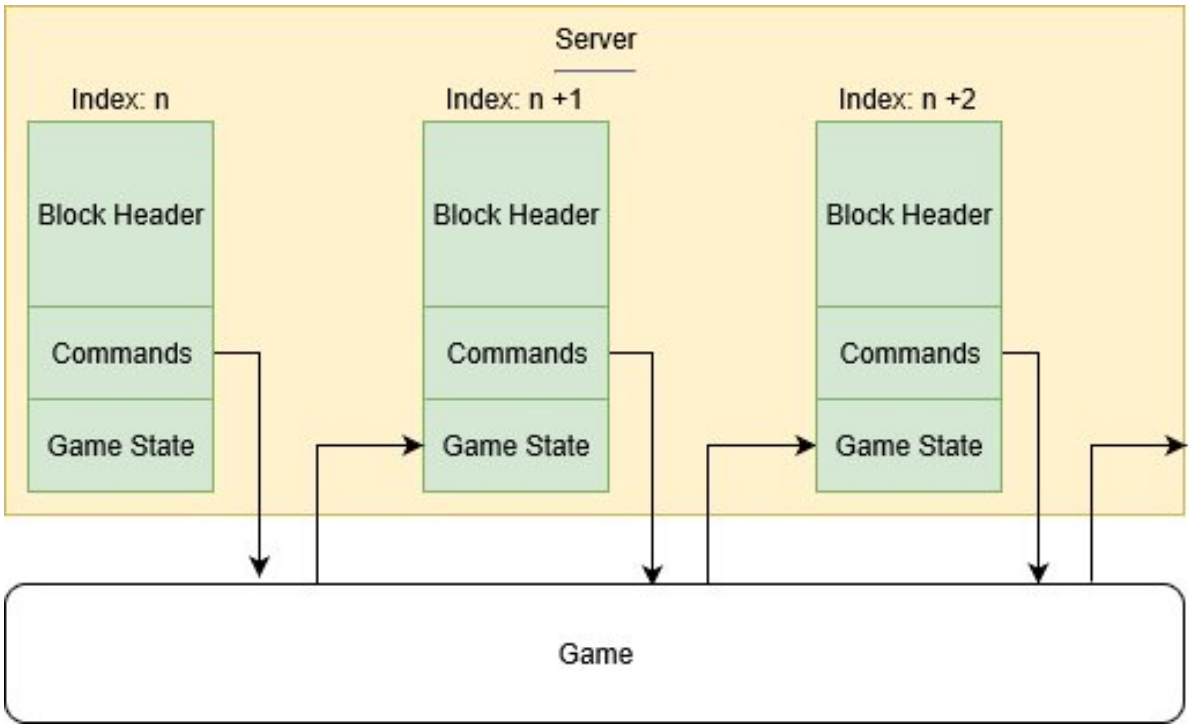
Figura 12 – Processo de recebimentos de comando do servidor



Fonte: Caio Rizzo (2019)

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 4.4.2.1.4 do protocolo, que os executa e envia um Game State de volta ao processo servidor, utilizando a mensagem GAMESTATE especificada também na seção 4.4.2.1.4, 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.

Figura 13 – Envio dos comandos das transações para o processo jogo



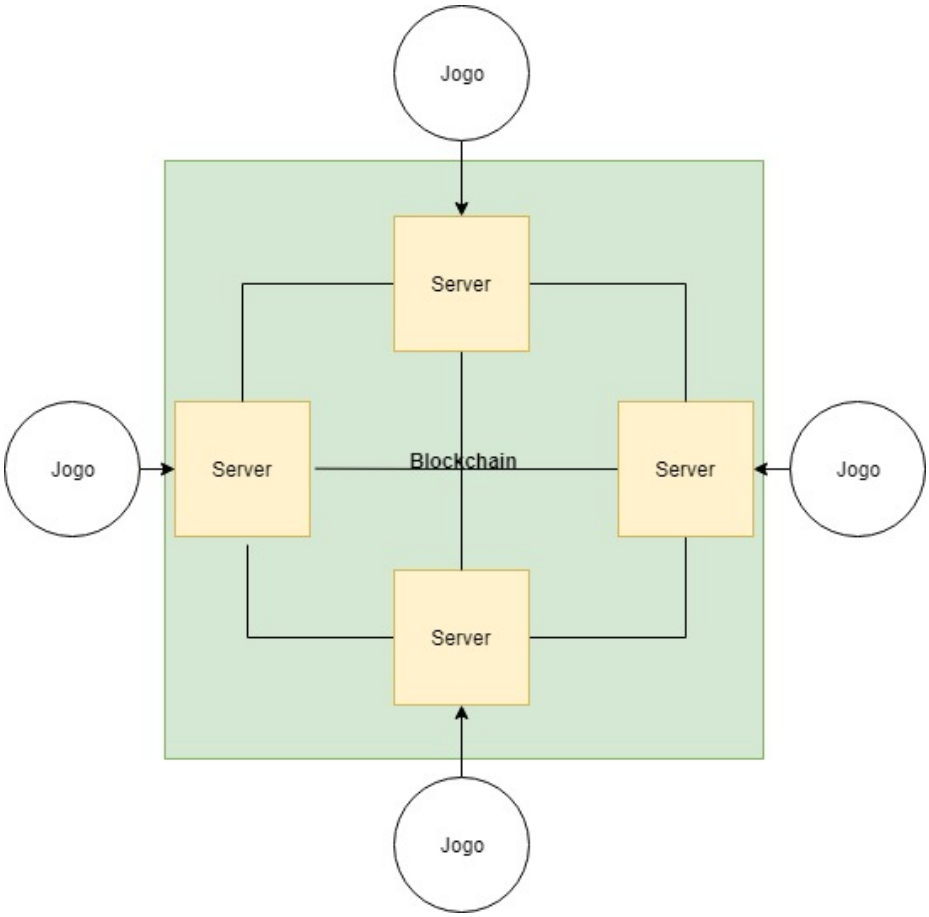
Fonte: Caio Rizzo (2019)

4.4 Protocolo de comunicação

4.4.1 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 14.

Figura 14 – Visão geral dos processos



Fonte: Caio Rizzo (2019)

A próxima subseção descreve as mensagens do protocolo.

4.4.2 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.

4.4.2.1 Comunicação jogo-servidor

4.4.2.1.1 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,“is
- LOGOUT - “LOGOUT\n”
  - A string “LOGOUT\n é enviada pelo servidor que deseja encerrar a conexão e não é esperada mensagem de confirmação.

4.4.2.1.2 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”

4.4.2.1.3 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”:
    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 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.

4.4.2.1.4 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.
- 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”

4.4.2.2 Comunicação servidor-servidor

4.4.2.2.1 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.

4.4.2.2.2 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.

4.4.2.2.3 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”.

4.4.2.2.4 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.

- 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,“maxSto

- 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 4.4.2.1.3 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 4.1. 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,\”popula 0”],“SeqNumber”:24}\n”

5 Caso de uso (jogo)

5.1 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.

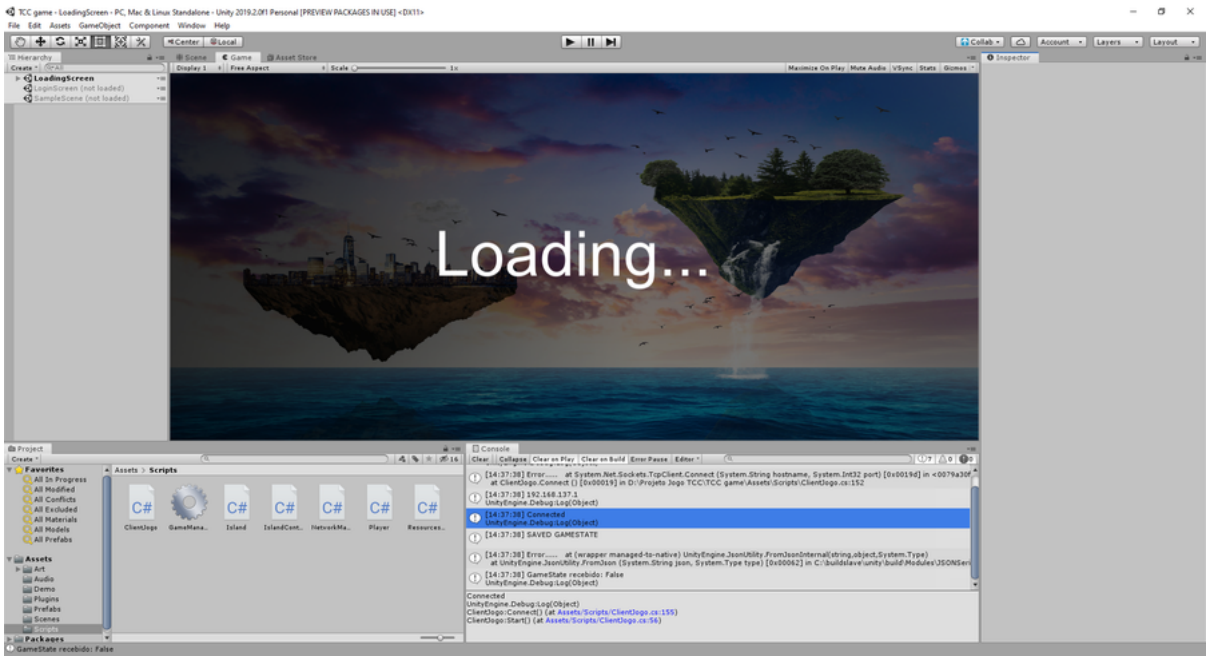
Dessa forma as ações que um jogador pode tomar durante o jogo são:

- Mover a população da ilha entre exército e extração de recursos, os quais podem ser: ouro, 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.

Imagens das telas do jogo foram colocadas na seção de apêndice.

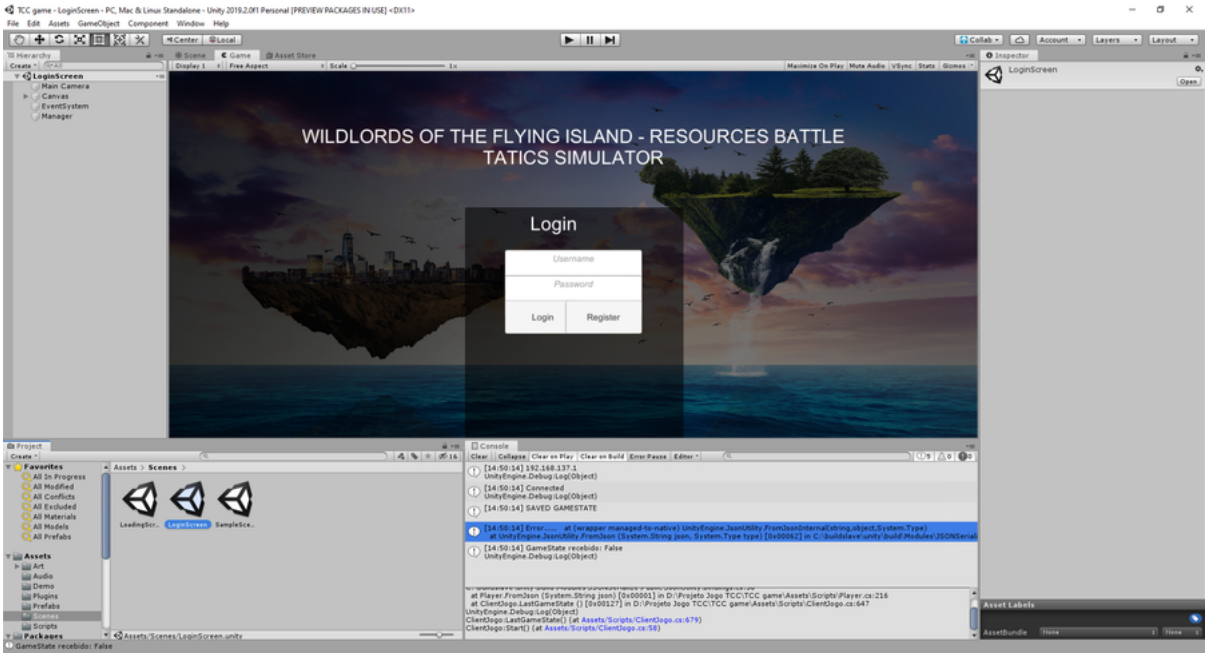
O jogo é composto por três telas principais. A tela de loading, onde ocorrem os processos de conexão com o servidor e requisição de game state (Figura 15). Tela de login (Figura 16), onde o usuário pode realizar o cadastro e se conectar com o jogo. E por último a tela do jogo em si (Figura 17).

Figura 15 – Tela de loading com interface do Unity



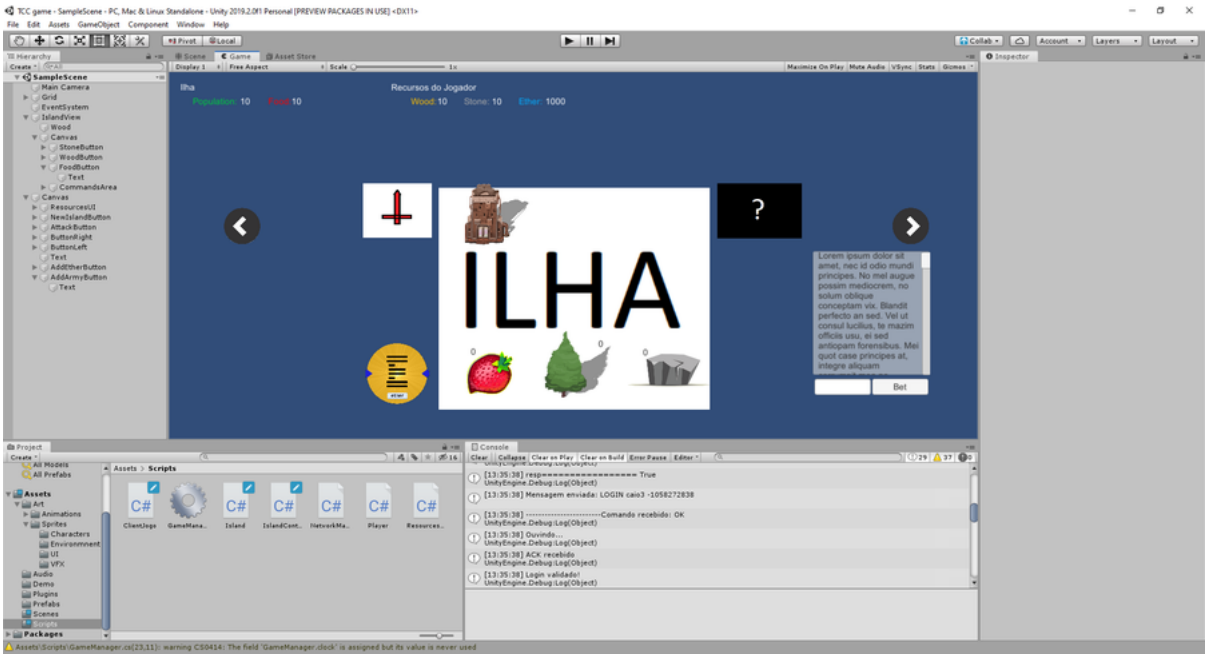
Fonte: Caio Rizzo (2019)

Figura 16 – Tela de login com interface do Unity



Fonte: Caio Rizzo (2019)

Figura 17 – Tela do jogo com interface do Unity



Fonte: Caio Rizzo (2019)

A tela do jogo, que pode ser vista mais detalhadamente na figura **??**, 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 18 – Tela do jogo detelahada

Fonte: Caio Rizzo (2019)

Na figura 18, 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.

5.2 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 (UNITY TECHNOLOGIES,).

- 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, segundo UNITY TECHNOLOGIES ():

- 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.

6 Conclusões e propostas de melhoria

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. 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

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

BACK, A. **Hashcash - A Denial of Service Counter-Measure**. 2002. Disponível em: http://www.hashcash.org/papers/hashcash.pdf.

BANKING on Bitcoin Documentário (90 min.). Christopher Cannucciari. EUA: Christopher Cannucciari, David Guy Levy, 2016.

BASHIR, I. **Mastering Blockchain**: Distributed ledger technology, decentralization, and smart contracts explained. 1. ed. [S.l.]: Packt Publishing, 2017.

CONTI, M. et al. A Survey on Security and Privacy Issues of Bitcoin. **IEEE Communications Surveys & Tutorials**, IEEE, v. 20, n. 4, p. 3416 – 3452, 2018. ISSN 1553-877X. Disponível em: https://ieeexplore.ieee.org/document/8369416.

DAI, W. **B-Money**. 1998. Disponível em: http://www.weidai.com/bmoney.txt.

DWYER, G. P. The Economics of Bitcoin and Similar Private Digital Currencies. **Journal of Financial Stability**, v. 17, p. 81 – 91, Abril 2015.

ETHEREUM. **Proof of Stake FAQ**. 2019. GitHub. Disponível em: https://github.com/ethereum/wiki/wiki/Proof-of-Stake-FAQ.

FOROUZAN, B. A. **TCP/IP Protocol Suite**. 4. ed. New York: McGraw-Hill Education, 2009. (McGraw-Hill Forouzan Networking). ISBN 978-0073376042. Disponível em: http://dl.acm.org/citation.cfm?id=940580. Acesso em: 22/05/2016.

FOROUZAN, B. A. **Comunicação de Dados e Redes de Computadores**. [S.l.]: McGraw Hill, 2010.

KAKAVAND, H.; SEVRES, N. K. D.; CHILTON, B. The Blockchain Revolution: An Analysis of Regulation and Technology Related to Distributed Ledger Technologies. Janeiro 1, 2017. Disponível em: https://ssrn.com/abstract=2849251orhttp://dx.doi.org/10.2139/ssrn.2849251.

KING, S.; SCOTT NADAL. PPCoin: Peer-to-Peer Crypto-Currency with Proof-of-Stake. Agosto 2012. Disponível em: https://decred.org/research/king2012.pdf.

KRAFT, D. Difficulty control for blockchain-based consensus systems. **Peer-to-Peer Networking and Applications**, v. 9, n. 2, p. 397 – 413, 2016. Disponível em: http://dx.doi.org/10.1007/s12083-015-0347-x.

MEIKLEJOHN, S. et al. A fistful of Bitcoins: characterizing payments among men with no names. **Commun. ACM**, v. 59, n. 4, p. 86 – 93, 2016. Disponível em: http://doi.acm.org/10.1145/2896384.

NAKAMOTO, S. **Bitcoin**: a peer-to-peer electronic cash system. 2008. Disponível em: https://bitcoin.org/bitcoin.pdf. Acesso em: 11/10/2018.

NARAYANAN, A. et al. **Bitcoin and Cryptocurrency Technologies – A Comprehensive Introduction**. 1. ed. [S.l.]: Princeton University Press, 2016. ISBN 978-0-691-17169-2.

PWC. **19° Pesquisa Global de Entretenimento e Mídia 2018-2022**. 2018. Disponível em: https://www.pwc.com.br/pt/outlook-18.html. Acesso em: 06 dez 2019.

STALLINGS, W. **Criptografia e Segurança de Redes**. 4. ed. [S.l.]: Pearson Education do Brasil, 2008.

STALLINGS, W. **Criptografia e Segurança de Redes**: Princípios e Práticas. 6. ed. [S.l.]: Pearson, 2015. ISBN 978-85-430-0589-8.

STRASSEL, K. A. **Deutsche Bank to Test 'E-Cash' With DigiCash in Pilot Project‖ on The Wall Street Journal**. 1996. Disponível em: https://www.wsj.com/articles/SB831416067295410500.

TANENBAUM, A. S.; STEEN, M. V. **Sistemas distribuídos**: princípios e paradigmas. 2. ed. São Paulo: Pearson Prentice Hall, 2007. ISBN 978-85-7605-142-8.

TSCHORSCH, F.; SCHEUERMANN, B. Bitcoin and Beyond: A Technical Survey on Decentralized Digital Currencies. **IEEE Communications Surveys and Tutorials**, v. 18, n. 3, p. 2084 – 2123, 2016. Disponível em: http://dx.doi.org/10.1109/COMST.2016.2535718.

UNITY TECHNOLOGIES. **Unity3D**. Disponível em: https://unity3d.com/pt/. Acesso em: 10/12/2017.

WRIGHT, A.; FILIPPI, P. D. Decentralized Blockchain Technology and the Rise of Lex Cryptographia. Março 10, 2015. Disponível em: https://ssrn.com/abstract=2580664.

YAHYAVI, A.; KEMME, B. Peer-to-peer architectures for massively multiplayer online games: A Survey. **ACM Comput. Surv.**, v. 46, n. 1, p. 9 –, 2013. Disponível em: http://doi.acm.org/10.1145/2522968.2522977.

THE trust machine. The Economist Group, Reino Unido, Outubro 2015.

Apêndices

Código do servidor em Golang

```
 1 package main
 2
 3 import (
 4         "bufio"
 5         "crypto/sha256"
 6         "encoding/hex"
 7         "encoding/csv"
 8         "encoding/json"
 9         "strconv"
10         "container/list"
11         "fmt"
12         "io"
13         "log"
14         "net"
15         "os"
16         "sync"
17         "time"
18         "errors"
19     "strings"
20 )
21
22 //Estrutura do BlockHeader
23 type BlockHeader struct {
24         Index     int
25         Timestamp string
26         GameStateHash string
27         TransactionsHash string
28         Hash      string
29         PrevHash  string
30         Validator string
31 }
32
33 //Estrutura de cada Bloco da Blockchain
34 type Block struct {
35         Head BlockHeader
36         Transactions []Transaction
37         GameState string
38 }
39
40 //Estrutura das Transações do jogo
41 type Transaction struct {
42         Username string
43         Commands []string
44         SeqNumber int
45 }
46
47 //Estrutura do Cliente
48 type Client struct {
49         Conn net.Conn
50         Mutex sync.Mutex
51         IP string
```

```
52          Messages chan string
53 }
54
55 //Estrutura do Cliente-jogador
56 type Player struct {
57          //Campos iguais no jogo
58          Username string
59          PasswordHash string
60          Ether int
61
62          //Campos extras do server
63          LastSeqNumber int
64          C *Client
65 }
66
67 //Estrutura das mensagens
68 type JSONMessage struct {
69          Keys []string
70          Values []Player
71 }
72
73 //Estrutura para comunicação com jogo
74 type JSONPlayer struct {
75      Username string
76      PasswordHash string
77      Ether int
78 }
79
80 //Constantes e Variaveis globais
81 const TIMEFORM = "2006-01-02 15:04:05.999999999 -0700"
82 const MAXCONNS = 100
83 const OPENCONNS = 3
84 const dCoinage = 61
85 var difficulty = 3
86 var TIMEOUT = 5 * time.Second
87 var secureBlock = 1 //Qual bloco (len - secureBlock) é
       considerado seguro
88 var gameState string //Game state do jogo
89
90 //Vetores/listas
91 var Blockchain []Block
92 var commandPool = list.New()
93 var transactionPool = list.New()
94 var tabelaServidores [2][100]string
95
96 //Flags
97 var forging bool
98 var logged bool
99
100 //Mutex para evitar condição de corrida
101 var mutex = &sync.Mutex{}
102 var mutexConnList = &sync.Mutex{}
103 var mutexTransactionPool = &sync.Mutex{}
104 var mutexCommandPool = &sync.Mutex{}
```

```
105
106 //Jogador conectado
107 var player Player
108
109 //MAPS
110 var localAddrs = make(map[string]int) //Guarda os endereços
      IP locais e os ja conectados para evitar conexões
      repetidas
111
112 var connectionsMap = make(map[string]*Client)
113
114 var transactionPoolMap = make(map[string]Transaction)
115
116 var playersMap = make(map[string]*Player)
117
118 //Channels
119 var unlockServer = make(chan int, 1)
120
121 var disconnect = make(chan int, 1)
122
123 var bet = make(chan int, 1) //Canal de anúncio de nova aposta
124
125 var ACK = make(chan int, 1) //Canal de sincronização de ACK
126
127 var att = make(chan int, 1) //Cria o canal de saída
128
129 var lose = make(chan Block) //Canal para anunciar que outro
      usuario validou o bloco
130
131 var loginChan = make(chan bool, 1) //Canal de anúncio de
      login de usuário
132
133 //----------------------------------Inicio Funções
      --------------------------------
134
135 func fillLocalAddrs() {
136         //Pega a lista de enreços locais
137         addrs, err := net.InterfaceAddrs()
138         if err != nil {
139             panic(err)
140         }
141         //Adiciona no Map
142         fmt.Println("Resgatando Ips locais")
143         for i, addr := range addrs {
144                 ip := strings.SplitN(addr.String(), "/", 2)
145                 args := strings.SplitN(ip[0], ".", 3)
146         if len(args) == 3 {
147             fmt.Println("Ip local adicionado: ", ip[0])
148             localAddrs[ip[0]] = i
149         }
150         }
151 }
152
153 //Função que cria um novo cliente, caso este ainda não exista
```

```
    no map de conexões
154 func newClient(conn net.Conn) (*Client) {
155         //Verifica se a conexão não é nula
156         if conn == nil {
157         return nil
158     }
159
160         //Resgata o ip
161         aux := conn.RemoteAddr().String()
162     remoteIP := strings.SplitN(aux, ":", 2)
163         ip := remoteIP[0]
164
165         //Verifica se não é o ip de uma conexão vigente
166     _, ok := connectionsMap[ip]
167     if ok {
168       fmt.Println("Conexão já estabelecida! Pulando")
169       return nil
170     }
171
172         //Cria o cliente
173         c := new(Client)
174         c.Conn = conn
175         c.IP = ip
176         c.Messages = make(chan string)
177
178         //Adiciona no map
179         mutexConnList.Lock()
180         connectionsMap[c.IP] = c
181         mutexConnList.Unlock()
182
183         return c
184 }
185
186 func closeClient(client *Client) {
187         if client == nil  {
188                 return
189         }
190         ip := client.IP
191         _, ok := connectionsMap[ip]
192     if ok {
193       //Se for o player
194       if connectionsMap[ip] == player.C {
195           //OBS: O ideal é obrigar a fazer login novamente
196           logged = false
197           }
198           client.Conn.Close()
199       //Remove da lista quem foi desconectado
200       mutexConnList.Lock()
201       delete(connectionsMap, ip)
202       mutexConnList.Unlock()
203       fmt.Println("Cliente desconectado!")
204       fmt.Println("Número de clientes conectados: ", len(
            connectionsMap))
205
```

```
206       //Se o número de clientes conectados for máximo
207       if len(connectionsMap) == (MAXCONNS-1) {
208         unlockServer <- 1
209       }
210
211     } else {
212         fmt.Println("Erro: Cliente não encontrado para a
              desconexão!")
213     }
214   }
215
216 //Verifica se o Hash atende a dificuldade da rede
217 func isHashValid(hash string, difficulty int) bool {
218     prefix := strings.Repeat("0", difficulty)
219     return strings.HasPrefix(hash, prefix)
220 }
221
222 //Função de validação dos blocos
223 func isBlockValid(newBlock, oldBlock Block) bool {
224         if oldBlock.Head.Index+1 != newBlock.Head.Index {
225                 fmt.Println("Here 1")
226                 return false
227         }
228
229         if oldBlock.Head.Hash != newBlock.Head.PrevHash {
230                 fmt.Println("Here 2")
231                 return false
232         }
233
234         if calculateBlockHash(newBlock) != newBlock.Head.Hash
              {
235                 fmt.Println("Here 3")
236                 return false
237         }
238
239         //Caso não tenha validador
240         if newBlock.Head.Validator == "" {
241                 fmt.Println("Here 4")
242                 return false
243         }
244
245         if isHashValid(newBlock.Head.Hash,
              calculateDifficultyPOS(newBlock, oldBlock)) ==
              false {
246                 fmt.Println("Here 6")
247                 return false
248         }
249
250         return true
251 }
252
253 func verificaTransacao(t Transaction) (bool) {
254         //Verifica se o player consta na lista de players
              existentes
```

```
255         p, ok := playersMap[t.Username]
256         if !ok {
257                 if t.SeqNumber == 0 { //Indica ser um novo
                       player
258                         args := strings.SplitN(t.Commands[0],
                            " ", 2)
259                         if len(t.Commands) == 1 && args[0] ==
                            "REGISTER" {
260                             return true
261                         }
262                         return false
263                 } else {
264                         fmt.Println("Player não existe")
265                         return false
266                 }
267         }
268         //Se não é vc mesmo (??!!)
269         if (*p) == player {
270                 fmt.Println("Seria um clone maligno?! Ou
                       apenas um eco da mensagem...")
271                 return false
272         }
273         //Se o número de sequência está correto e em ordem
274         if p.LastSeqNumber >= t.SeqNumber {
275                 fmt.Println("Mensagem antiga perdida pela
                       rede")
276                 return false
277         }
278
279         //Como a mensagem é menor, recebe o número da nova
               mensagem
280         if p.LastSeqNumber < t.SeqNumber {
281                 playersMap[t.Username].LastSeqNumber = t.
                       SeqNumber
282         }
283
284         return true
285 }
286
287 //Calcula a dificuldade do bloco, baseando-se no coinage
288 func calculateDifficultyPOS(newBlock Block, oldBlock Block) (
       int) {
289         p := fmt.Println
290         t, err1 := time.Parse(TIMEFORM, newBlock.Head.
               Timestamp)
291         tOB, err2 := time.Parse(TIMEFORM, oldBlock.Head.
               Timestamp)
292         var stack = 0
293
294         //O validador do bloco deve ser sempre o primeiro
295         if newBlock.Transactions != nil && newBlock.
               Transactions[0].Commands != nil {
296                 //Captura o Ether
297                 //Sempre a primeira transação do bloco
```

```
298                 args := strings.SplitN(newBlock.Transactions
                       [0].Commands[0], " ", 2) //Para uma string
                        BET NUMERO\n
299                 stack, _ = strconv.Atoi(strings.TrimSpace(
                       args[1]))
300                 p("O Bet deste bloco é de: ", stack)
301         }
302
303         if err1 != nil && err2 != nil {
304                 p("Erro ocorrido no tempo")
305         }
306
307         //Faz a conta do coinage (adicionar score do
               validador)
308         target := dCoinage - int(t.Sub(tOB).Seconds()) - (int
               (t.Sub(tOB).Seconds() * float64(stack)/10.0)) //
               Olhar essa conta
309         //p(target)
310         if target < 0 {
311                 target = 0
312         }
313         //p("A dificulade do bloco esta em: ", target)
314
315         return target
316 }
317
318 //Retorna o ultimo estado salvo, a partir de um i passado
319 func getLastSavedGame(i int) (string){
320         var json string
321         json = ""
322         for ; i < len(Blockchain) && i >= 0; i-- {
323                 if Blockchain[i].GameState != "" {
324                         json = Blockchain[i].GameState
325                         break
326                 }
327         }
328         return json
329 }
330
331 //Retorna o último numero de sequencia conhecido para aquele
       player
332 func getLastSeqNumber(username string) (int){
333         //Percorre a Blockchain
334         for i := len(Blockchain) - 1; i >= 0; i-- {
335                 //Percorre as transações de cada bloco
336                 for j := 0; j < len(Blockchain[i].
                       Transactions); j++ {
337                         if Blockchain[i].Transactions[j].
                               Username == username {
338                                 return Blockchain[i].
                                       Transactions[j].SeqNumber
339                         }
340                 }
341         }
```

```
342         return -1
343 }
344
345 //Carrega lista de players de um JSON
346 func carregaPlayersFromJson(jsonGS string) bool {
347         var m JSONMessage
348
349         //Decodifica JSON
350         err := json.Unmarshal([]byte(jsonGS), &m)
351     if err != nil {
352                 fmt.Println("Error na decodificação do Json")
353                 return false
354         } else {
355                 //Salva os players no vetor global de players
356                 for i := 0; i < len(m.Values); i++ {
357                         m.Values[i].LastSeqNumber =
                            getLastSeqNumber(m.Values[i].
                            Username)
358
359                         //OBS sujeito a alteração
360                         //Verifica possivel erro
361                         if m.Values[i].LastSeqNumber == -1{
362                                 //log.Fatal("Erro Fatal:
                                    Player não encontrado,
                                    Blockchain inconsistente!\
                                    n(Para um player existir
                                    deve haver uma primeira
                                    transação de Registro de
                                    numero de sequencia 0)")
363                         }
364
365                         //Salva no map de players
366                         playersMap[m.Values[i].Username] = &m
                            .Values[i]
367                 }
368         }
369         return true
370 }
371
372 //Carrega a lista de players extraída do último bloco e salva
        no Map
373 func carregaPlayersJson() {
374         var m JSONMessage
375
376         if len(Blockchain) - 1 == 0 {
377                 return
378         }
379         //Captura ultimo estado de jogo salvo na blockchain
380         saved := getLastSavedGame(len(Blockchain) - 1)
381
382         //Decodifica JSON
383         err := json.Unmarshal([]byte(saved), &m)
384     if err != nil {
385                 fmt.Println("Error na decodificação do Json")
```

```
386         } else {
387                 //Salva os players no vetor global de players
388                 for i := 0; i < len(m.Values); i++ {
389                         m.Values[i].LastSeqNumber =
                               getLastSeqNumber(m.Values[i].
                               Username)
390
391                         //OBS sujeito a alteração
392                         //Verifica possivel erro
393                         if m.Values[i].LastSeqNumber == -1{
394                                 //log.Fatal("Erro Fatal:
                                       Player não encontrado,
                                       Blockchain inconsistente!\
                                       n(Para um player existir
                                       deve haver uma primeira
                                       transação de Registro de
                                       numero de sequencia 0)")
395                         }
396
397                         //Salva no map de players
398                         playersMap[m.Values[i].Username] = &m
                               .Values[i]
399                 }
400         }
401 }
402
403 //Lê a blockchain salva no disco (se existir)
404 func carregaBlockchain() []Block{
405         var BlockchainTemp []Block
406         BlockchainTemp = nil
407         buffer := make([]byte, 4096)
408         //Abre os arquivos de bloco locais, decodifica,
               verifica integridade e salva na memória
409         for i := 0; ; i++ {
410                 f, err := os.Open("Block_" + strconv.Itoa(i))
411                 if err == nil {
412                         n, err2 := f.Read(buffer)
413                         if err2 != nil{
414                                 fmt.Println("Error: leitura
                                       de arquivo blockchain")
415                                 return nil
416                         }
417                         //fmt.Println("lidos ", n)
418                         bloco := decodifica(buffer[:n])
419                         if i == 0 || isBlockValid(bloco,
                               BlockchainTemp[i-1]) {
420                                 BlockchainTemp = append(
                                       BlockchainTemp, bloco)
421                                 imprimeBloco(bloco)
422                         } else {
423                                 fmt.Println("Error:
                                       blockchain local salva
                                       inconsistente")
424                                 break
```

```
425                         }
426                 } else {
427                         break
428                 }
429                 f.Close()
430         }
431         //Se i == 0, não existe blockchain local e sera
               retornado nil
432         return BlockchainTemp
433 }
434
435 //Função que salva o bloco em arquivo
436 func salvaBloco(b Block) (error){
437         vet := codifica(b)
438         f, err := os.Create("Block_" + strconv.Itoa(b.Head.
               Index))
439         defer f.Close()
440         if err != nil {
441                 fmt.Println("Error: file creation fail in
                       block saving")
442                 return err
443         }
444
445         _, err = f.Write(vet)
446         if err != nil {
447                 fmt.Println("Error: write in file fail in
                       block saving")
448                 return err
449         }
450
451         return nil
452 }
453
454 //Lê o arquivo csv e retorna uma matriz de strings
455 //TODO tratar para quando não há arquivo criado ainda
456 func parseData(file string) ([][]string, error) {
457         f, err := os.Open(file)
458     if err != nil {
459         return nil, err
460     }
461     defer f.Close()
462
463     data, err := csv.NewReader(f).ReadAll()
464     if err != nil {
465         return nil, err
466     }
467
468     return data, nil
469 }
470
471 //Gera um vetor com todas as transações da pool
472 func fromPoolToVectorT() ([]Transaction, bool) {
473         var tVet []Transaction
474         mutexTransactionPool.Lock()
```

```
475         //Copia e converte os comandos para vetor
476         for _, t := range transactionPoolMap {
477         tVet = append(tVet, t)
478     }
479     if len(tVet) == len(transactionPoolMap) {
480                 transactionPoolMap = make(map[string]
                       Transaction) //Zera a pool
481                 mutexTransactionPool.Unlock()
482         } else {
483                 fmt.Println("Erro: Comandos não foram
                       convertidos e copiados, abortando!")
484                 mutexTransactionPool.Unlock()
485                 return tVet, false //TODO verificar os
                       impactos deste return
486         }
487
488         return tVet, true
489 }
490
491 //Cria uma transação a partir dos comandos na pool de
       comandos
492 func transactionFromCommPool() (Transaction, bool) {
493         var t Transaction
494         fmt.Println("Player: ", player.Username)
495         if player.Username == "" {
496                 return t, false
497         }
498         t.Username = player.Username
499         t.SeqNumber = player.LastSeqNumber + 1
500         player.LastSeqNumber++
501         playersMap[player.Username].LastSeqNumber = player.
               LastSeqNumber
502
503         mutexCommandPool.Lock()
504         //Copia e converte os comandos para vetor
505         for e := commandPool.Front(); e != nil; e = e.Next()
               {
506         t.Commands = append(t.Commands, e.Value.(string))
507     }
508     if (len(t.Commands) == commandPool.Len()) {
509                 commandPool.Init() //Zera a pool
510                 mutexCommandPool.Unlock()
511         } else {
512                 fmt.Println("Erro: Comandos não foram
                       convertidos e copiados, abortando!")
513                 mutexCommandPool.Unlock()
514                 return t, false //TODO verificar os impactos
                       deste return
515         }
516
517         return t, true
518 }
519
520 //Resgata transações que estão perdidas
```

```
521 func salvaTransacoesPerdidas(newBlock Block, oldBlock Block){
522         var hash string
523         var hash2 string
524         flag := false
525         for _, t := range newBlock.Transactions {
526                 hash = calculateTransactionHash(t)
527                 _, ok := transactionPoolMap[hash]
528                 //Se a transação do novo bloco está na pool,
                      apaga
529                 if ok {
530                         delete(transactionPoolMap, hash)
531                 }
532         }
533         for _, t := range oldBlock.Transactions {
534                 flag = false
535                 for _, t2 := range newBlock.Transactions {
536                         hash = calculateTransactionHash(t)
537                         hash2 = calculateTransactionHash(t2)
538                         if hash == hash2 {
539                                 flag = true
540                                 break
541                         }
542                 }
543                 if flag == false {
544                         transactionPoolMap[hash] = t
545                 }
546         }
547 }
548
549 //----------------------------Funções de codificação/
      Decodificação--------------------------//
550
551 //Função do cálculo do hash em SHA256
552 func calculateHash(s string) string {
553         h := sha256.New()
554         h.Write([]byte(s))
555         hashed := h.Sum(nil)
556         return hex.EncodeToString(hashed)
557 }
558
559 //Função que calcula o hash do bloco, reunindo a informação
      necessária
560 func calculateBlockHash(block Block) string {
561         t, _ := time.Parse(TIMEFORM, block.Head.Timestamp)
562         record := string(block.Head.Index) + block.Head.
              GameStateHash + block.Head.TransactionsHash +
              block.Head.PrevHash + block.Head.Validator +
              string(t.Second())
563         return calculateHash(record)
564 }
565
566 func calculateCommandsHash(commands []string) (string) {
567         //Transforma os comandos em um vetor de bytes
568         bytes := codificaComandos(commands)
```

```
569         //Converte para string
570         record := string(bytes)
571         //Calcula o hash da string e retorna
572         return calculateHash(record)
573 }
574
575 func calculateTransactionHash(t Transaction) (string){
576     return calculateHash(string(codificaTransaction(t)))
577 }
578
579 func calculateTransactionsHash(t []Transaction) (string){
580     //Codifica e verifica erro
581         b, err := json.Marshal(t)
582     if err != nil {
583                 log.Fatal("Error na decodificação do Json
                       Transaction")
584         }
585     return calculateHash(string(b))
586 }
587
588 func codificaTransaction(t Transaction) ([]byte){
589         //Codifica e verifica erro
590         b, err := json.Marshal(t)
591     if err != nil {
592                 log.Fatal("Error na decodificação do Json
                       Transaction")
593         }
594         return b
595 }
596
597 func decodificaTransaction(b []byte) (Transaction){
598         var t Transaction
599         err := json.Unmarshal(b, &t)
600     if err != nil {
601                 log.Fatal("Error na decodificação do Json
                       Transaction")
602         }
603     return t
604 }
605
606 func codificaComandos(commands []string) ([]byte){
607         //Codifica e verifica erro
608         b, err := json.Marshal(commands)
609     if err != nil {
610                 fmt.Println("Error na decodificação do Json
                       Comandos")
611         }
612         return b
613 }
614
615 func decodificaComandos(b []byte) (*list.List){
616         var respList = list.New()
617         var commands []string
618         err := json.Unmarshal(b, &commands)
```

```
619     if err != nil {
620                 fmt.Println("Error na decodificação do Json
                       Comandos")
621         }
622     for _, command := range commands {
623         //fmt.Println(command)
624         respList.PushBack(command)
625     }
626 
627     return respList
628 }
629 
630 func codificaTabela(mat [][]string) ([]byte){
631         //Codifica e verifica erro
632         b, err := json.Marshal(mat)
633     if err != nil {
634                 fmt.Println("Error na decodificação do Json
                       Tabela")
635         }
636         return b
637 }
638 
639 func decodificaTabela(b []byte) ([][]string){
640         var mat [][]string
641         err := json.Unmarshal(b, &mat)
642     if err != nil {
643                 fmt.Println("Error na decodificação do Json
                       Tabela")
644         }
645     return mat
646 }
647 
648 func codifica(block Block) ([]byte){
649         b, err := json.Marshal(block)
650     if err != nil {
651                 fmt.Println("Error na decodificação do Json
                       Bloco")
652         }
653         return b
654 }
655 
656 func decodifica(b []byte) (Block){
657         var bloco Block
658         err := json.Unmarshal(b, &bloco)
659     if err != nil {
660                 fmt.Println("Error na decodificação do Json
                       Bloco")
661                 fmt.Println(err)
662         }
663         return bloco
664 }
665 
666 func imprimeBloco(b Block){
667         fmt.Println("Bloco: ", b.Head.Index)
```

```
668         fmt.Println("Timestamp: ", b.Head.Timestamp)
669         fmt.Println("GameState Hash: ", b.Head.GameStateHash)
670         fmt.Println("Transactions Hash: ", b.Head.
               TransactionsHash)
671         fmt.Println("Hash: ", b.Head.Hash)
672         fmt.Println("Previous Hash: ", b.Head.PrevHash)
673         fmt.Println("Validador: ", b.Head.Validator)
674         fmt.Println("GameState(JSON): ", b.GameState)
675         if b.Transactions != nil {
676                 fmt.Println("Transactions in Block: ")
677                 for _, t := range b.Transactions {
678                         fmt.Println("   Transaction")
679                         fmt.Println("           Username: ",
                           t.Username)
680                         fmt.Println("           SeqNumber: ",
                            t.SeqNumber)
681                         fmt.Println("           Commands: ")
682                         if t.Commands != nil {
683                                 for _, c := range t.Commands
                                    {
684                                         fmt.Println("
                                                    ",
                                             c)
685                                 }
686                         }
687                 }
688         } else {
689                 fmt.Println("Transactions in Block: Empty")
690         }
691         fmt.Println()
692         fmt.Println()
693 }
694
695 //--------------------Funções de envio de mensagem
       ----------------------//
696
697 //Envia mensagens que esperam uma mensagem de volta
698 func enviaMsg(conn net.Conn, msg string) (string, error){
699         c1 := make(chan string, 1)
700         var err error
701         var resp string
702         res := ""
703         if conn != nil {
704                 fmt.Println("Enviando msg1: ", msg)
705                 rw := bufio.NewReadWriter(bufio.NewReader(
                       conn), bufio.NewWriter(conn))
706                 rw.WriteString(msg)
707                 rw.Flush()
708
709                 err2 := &err
710
711                 go func() {
712                         //Aguarda confirmação de recebimento
713                 fmt.Println("Aguardando recebimendo de ACK
```

```
                    ...1")
714                         resp, *err2 = rw.ReadString('\n')
715                         c1 <- resp
716                 }()
717                 if err != nil {
718                         fmt.Println("Error! ", err)
719                         return "", err
720                 }
721                 //Espera o retorno no tempo, caso contrário
                       ocorre o timeout
722                 select {
723         case res = <-c1:
724                 args := strings.SplitN(res, " ", 2)
725                         op := strings.TrimSpace(args[0])
726                 if op == "OK" { //Tudo ocorreu corretamente
727                                 fmt.Println(res)
728
729                         } else if strings.TrimSpace(res) == "
                           ERRO" {
730                                 fmt.Println("Erro: ", res)
731                                 return "", errors.New(res)
732
733                         } else {
734                                 fmt.Println(res)
735                         }
736         case <-time.After(TIMEOUT):
737                 fmt.Println("Erro: client timed out!")
738                 return "", errors.New("Client timed out!")
739         }
740
741         } else {
742                 return "", errors.New("Cliente desconectado
                       !")
743         }
744
745         return res, err
746 }
747
748 //Envia mensagens que esperam um ACK de OK ou ERRO no canal
      em vez de reader
749 func enviaMsg2(client *Client, msg string) (string, error){
750         var err error
751         err = nil
752         res := ""
753         if client == nil {
754                 fmt.Println("Erro: Cliente inexistente!")
755                 return res, errors.New("Cliente inexistente
                       !")
756         }
757         if client.Conn != nil {
758                 fmt.Println("Enviando msg2: ", msg)
759                 rw := bufio.NewReadWriter(bufio.NewReader(
                       client.Conn), bufio.NewWriter(client.Conn)
                       )
```

```
760                 rw.WriteString(msg)
761                 rw.Flush()
762
763         //Avisa que está esperando uma menssagem
764         client.Messages <- "ACK"
765
766                 fmt.Println("Aguardando recebimendo de ACK
                       ...2")
767                 //Espera o retorno no tempo, caso contrário
                       ocorre o timeout
768                 select {
769
770         case res = <- client.Messages:
771                 args := strings.SplitN(res, " ", 2)
772                         op := strings.TrimSpace(args[0])
773                 if op == "OK" { //Tudo ocorreu corretamente
774                                 fmt.Println(res)
775                         } else if strings.TrimSpace(res) == "
                           ERRO" {
776                                 fmt.Println("Erro: ", res)
777                         }
778         case <-time.After(TIMEOUT):
779                 fmt.Println("Erro: client timed out!")
780             closeClient(client)
781                 return "", errors.New("Client timed out!")
782         }
783
784         } else {
785         closeClient(client)
786                 return "", errors.New("Cliente desconectado
                       !")
787         }
788
789         return res, err
790 }
791
792 //Envia mensagens que não esperam um ACK de volta
793 func enviaMsg3(client *Client, msg string) (error){
794         var err error
795         if client == nil {
796                 fmt.Println("Erro: Cliente inexistente!")
797                 return errors.New("Cliente inexistente!")
798         }
799         if client.Conn != nil {
800                 fmt.Println("Enviando msg3: ", msg)
801                 w := bufio.NewWriter(client.Conn)
802                 w.Reset(client.Conn)
803                 fmt.Println("Enviando3...")
804                 _, err := w.WriteString(msg)
805                 if err != nil {
806                         fmt.Println("Erro ao escrever string:
                             " + msg)
807                 }
808                 w.Flush()
```

```
809         } else {
810         closeClient(client)
811                 return errors.New("Cliente desconectado!")
812         }
813
814         return err
815 }
816
817 //Distribui mensagem para clientes conectados
818 func distribuiMsg(client *Client, msg string) {
819         fmt.Println("Distribuindo mensagem para os servidores
              ")
820         for key, enviar := range connectionsMap {
821         if enviar != client && enviar != player.C {
822                 //Trava o Mutex para distribuição
823                 enviar.Mutex.Lock()
824                 if enviar.Conn != nil {
825                fmt.Println("Enviando mensagem para o servidor:
                    ", key)
826                                 enviaMsg2(enviar, msg)
827                     } else {
828                         fmt.Println("Cliente está
                            desconectado")
829                     }
830                 enviar.Mutex.Unlock()
831         }
832         }
833         fmt.Println("Distribuição finalizada!")
834 }
835
836 //Pode tb so receber a mensagem e repassa-la
837 func distribuiBloco(client *Client, bloco Block) {
838         fmt.Println("Distribuindo bloco para os servidores")
839         for key, enviar := range connectionsMap {
840         if enviar != client && enviar != player.C {
841                 enviar.Mutex.Lock()
842                 if enviar.Conn != nil {
843                                 fmt.Println("Enviando bloco
                                    para o servidor: ", key)
844                 msg := "WIN " + string(codifica(bloco)) + "\n
                       "
845                 enviaMsg3(enviar, msg)
846                     } else {
847                         fmt.Println("Cliente está
                            desconectado")
848                     }
849                 enviar.Mutex.Unlock()
850         }
851         }
852         fmt.Println("Distribuição finalizada!")
853 }
854
855 //Flooda uma transação pela rede
856 func floodTransaction(client *Client, t Transaction) {
```

```
857         //TRANSACTION BYTES\n
858         bytes := codificaTransaction(t)
859         msg := "TRANSACTION " + string(bytes) + "\n"
860         distribuiMsg(client, msg)
861 }
862
863 //Envia os comando para o jogo e aguarda ACK nesta função
864 //OBS não pode ter lock
865 func enviaComandosParaJogo(bloco Block) {
866         if len(Blockchain)-1 == 0 || player.C == nil{
867                 return
868         }
869         fmt.Println("Enviando comandos do Bloco " + strconv.
               Itoa(bloco.Head.Index) +" para o Jogo")
870         for i := 0; i < len(bloco.Transactions); i++ {
871                 for j := 0; j < len(bloco.Transactions[i].
                       Commands); j++ {
872                         if bloco.Transactions[i].SeqNumber
                               !=0 { //Se é o registro muda
873                                 msg := "COMMAND " + bloco.
                                       Transactions[i].Username +
                                        " " + bloco.Transactions[
                                       i].Commands[j] + "\n"
874                                 enviaMsg2(player.C, msg)
875                         } else {
876                                 msg := bloco.Transactions[i].
                                       Commands[j] + "\n"
877                                 enviaMsg2(player.C, msg)
878                         }
879                 }
880         }
881         msg := "END BLOCK\n"
882         enviaMsg3(player.C, msg)
883         fmt.Println("Envio de comandos do Bloco finalizado!")
884 }
885
886 //Envia os comando para o jogo e aguarda ACK na função HANDLE
887 func enviaComandosParaJogo2(bloco Block) {
888         if len(Blockchain)-1 == 0{
889                 return
890         }
891         fmt.Println("Enviando comandos do Bloco " + strconv.
               Itoa(bloco.Head.Index) +" para o Jogo")
892         fmt.Println("----------LOCK em enviaComandosParaJogo2
               ")
893         player.C.Mutex.Lock()
894         for i := 0; i < len(bloco.Transactions); i++ {
895                 for j := 0; j < len(bloco.Transactions[i].
                       Commands); j++ {
896                         if bloco.Transactions[i].SeqNumber
                               !=0 { //Se não é o registro
897                                 msg := "COMMAND " + bloco.
                                       Transactions[i].Username +
                                        " " + strings.TrimSpace(
```

```
                                        bloco.Transactions[i].
                                        Commands[j]) + "\n"
898                                 enviaMsg2(player.C, msg)
899                         } else {
900                                 msg := strings.TrimSpace(
                                        bloco.Transactions[i].
                                        Commands[j]) + "\n"
901                                 enviaMsg2(player.C, msg)
902                         }
903                 }
904         }
905         enviaMsg3(player.C, "END BLOCK\n")
906         player.C.Mutex.Unlock()
907         fmt.Println("==========UNLOCK em
               enviaComandosParaJogo2")
908         fmt.Println("Envio de comandos do Bloco finalizado!")
909 }
910
911 //--------------------Funções de forjamento
       ----------------------//
912
913 func generateGenesisBlock() {
914         //Criando o genesisBlock
915         var genesisBlock Block
916         t := time.Now()
917         var tr Transaction
918         genesisBlock.Transactions = []Transaction{tr}
919         genesisBlock.Transactions[0].Commands = []string{"let
               there be light"}
920         //genesisBlock.GameState = "JSON"
921
922         genesisBlock.Head.Index = 0
923         genesisBlock.Head.Timestamp = t.Format(TIMEFORM)
924         genesisBlock.Head.GameStateHash = "First of Many"
925         genesisBlock.Head.TransactionsHash = "Create Genesis"
926         genesisBlock.Head.PrevHash = "1"
927         genesisBlock.Head.Validator = "God"
928         genesisBlock.Head.Hash = calculateBlockHash(
               genesisBlock)
929         Blockchain = []Block{genesisBlock};
930         salvaBloco(genesisBlock)
931 }
932
933 // Cria um novo bloco para a blockchain
934 func generateBlock(oldBlock Block, name string, transactions
       []Transaction) (Block, error) {
935
936         var newBlock Block
937
938         if transactions == nil {
939                 //TODO Não sei oq fazer ainda
940                 //newBlock.Commands = []string{"BET 0\n"}
941         } else {
942                 newBlock.Transactions = transactions
```

```
943                 //newBlock.Commands = commands
944         }
945         newBlock.GameState = gameState
946 
947         t := time.Now()
948         newBlock.Head.Index = oldBlock.Head.Index + 1
949         newBlock.Head.Timestamp = t.Format(TIMEFORM)
950         newBlock.Head.GameStateHash = calculateHash(newBlock.
              GameState)
951         newBlock.Head.TransactionsHash =
              calculateTransactionsHash(newBlock.Transactions)
952         newBlock.Head.PrevHash = oldBlock.Head.Hash
953         newBlock.Head.Validator = name
954         newBlock.Head.Hash = calculateBlockHash(newBlock)
955 
956         return newBlock, nil
957 }
958 
959 //Gera um novo bloco válido
960 func generatePOS() (Block, bool) {
961         p := fmt.Println
962         var newBlock Block
963         var win bool
964         win = true
965         oldBlock := Blockchain[len(Blockchain)-1]
966         var tVet []Transaction
967         var ok bool
968 
969         //Verifica se existe um player logado
970         if player.Username == "" {
971                 p("Erro: Não existe um player logado para
                      forjar um bloco")
972                 return newBlock, false //TODO verificar os
                      impactos deste return
973         }
974 
975         for {
976                 //Gera um vetor com todas as transações da
                      pool e zera esta
977                 tVet, ok = fromPoolToVectorT()
978                 if !ok {
979                         p("Erro: Transações não foram
                              convertidos e copiados, abortando
                              !")
980                         return newBlock, false //TODO
                              verificar os impactos deste return
981                 }
982                 //Enquanto não houverem comandos (Sujeito a
                      mudança)
983                 if len(tVet) > 0 {
984                         break
985                 }
986         }
987 
```

```
988         p("Procurando Hash válido...")
989         //Tenta encontrar o hash que satisfaça a dificulade
               da rede
990         tempoI := time.Now()
991         for tentativas := 1; ; tentativas++ {
992
993                 //Verifica se algum outro servidor já
                       encontrou o bloco válido
994                 select {
995           case bWinner := <-lose:
996                                   p("
                                      XXXXXXXXXXXXXXXXXXXXXXXXXX
                                       PERDEU
                                      XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
                                      ")
997                         salvaTransacoesPerdidas(bWinner,
                               newBlock)
998                                   return newBlock, false
999                         default:
1000                }
1001
1002                //TODO adicionar ID do jogador na geração dos
                        blocos
1003                newBlock, _ = generateBlock(oldBlock, player.
                       Username, tVet)
1004
1005                target := calculateDifficultyPOS(newBlock,
                       oldBlock)
1006
1007                if isHashValid(newBlock.Head.Hash, target){
1008                        //p("Hash satisfatório encontrado")
1009                        //p(newBlock.Head.Hash)
1010                        //p("Tentativas: ", tentativas)
1011                        break
1012                } else {
1013                        time.Sleep(time.Second - 100)
1014                }
1015        }
1016
1017        if !isBlockValid(newBlock, oldBlock) {
1018                log.Fatal("Forjando blocos invalidos")
1019                return newBlock, false
1020        }
1021
1022        p("--------------------------Bloco Forjado
                ---------------------------")
1023        p("Levou: ", int(time.Now().Sub(tempoI).Seconds()))
1024
1025        imprimeBloco(newBlock)
1026        return newBlock, win
1027 }
1028
1029 //Função que mantém a geração de blocos
1030 func forge() {
```

```
1031         //Verifica se já não está ativa
1032         if forging {
1033         return
1034         }
1035         forging = true
1036         for j := 1; ; j++ {
1037                 novo, ok := generatePOS()
1038                 if ok {
1039                         mutex.Lock()
1040             if isBlockValid(novo, Blockchain[len(Blockchain)
                    -1]) == true {
1041                                 Blockchain = append(
                                       Blockchain, novo)
1042                                 salvaBloco(novo)
1043                                 distribuiBloco(nil,novo)
1044                                 enviaComandosParaJogo2(novo)
1045                         }
1046                         mutex.Unlock()
1047         } else {
1048                 forging = false
1049                 return
1050         }
1051         }
1052 }
1053
1054 //--------------------Principais funções do Cliente
       ----------------------//
1055
1056 //Requisita toda a blockchain de outro servidor
1057 func solicitaBlockchain(client *Client){
1058         var err error
1059         var res string
1060
1061         //Envia o comando de solicitar todos os blocos ao
              servidor
1062         res, err = enviaMsg2(client, "BLOCK ALL\n")
1063         if err != nil {
1064                 fmt.Println("Erouuu! ", err)
1065         }
1066
1067         for {
1068                 //Le o total enviado
1069                 fmt.Println("Esperando próximo bloco ")
1070
1071                 if err != nil {
1072                         fmt.Println("Erro: resposta fora do
                                padrão, erro não tratado ou nula\n
                                ", err)
1073                         return
1074                 }
1075                 //Espera o retorno no tempo, caso contrário
                      ocorre o timeout
1076                 //Quebra os argumentos
1077                 args := strings.SplitN(res, " ", 2)
```

```
1078                    //Limpa a msg
1079                    op := strings.TrimSpace(args[0])
1080                    //fmt.Println(op)
1081                    if op == "END" {
1082                            //Verifica se é a mensagem de
                                   confirmação de fim de operação e
                                   retorna
1083                            fmt.Println("Blockchain sucessfully
                                   received! ")
1084                            att <- 1
1085                            return
1086                    } else if op == "ERRO" {
1087                            //TODO Verificar isso
1088                            fmt.Println("Erro: erro de número ",
                                   args[1])
1089                            return
1090                    } else {
1091                            bloco := decodifica([]byte(strings.
                                   TrimSpace(res)))
1092                            fmt.Println("Bloco recebido:")
1093                            imprimeBloco(bloco)
1094                            //Decodifica o bloco e coloca na
                                   blockchain
1095                            //mutex.Lock()
1096                            if Blockchain == nil || isBlockValid(
                                   bloco, Blockchain[len(Blockchain)
                                   -1]) {
1097                                    fmt.Println("Bloco válido e
                                           inserido: " + strconv.Itoa
                                           (bloco.Head.Index))
1098                                    Blockchain = append(
                                           Blockchain, bloco)
1099                                    salvaBloco(bloco)
1100                                    res, err = enviaMsg2(client,
                                           "OK\n")
1101                            } else {
1102                                    //Erro de recebimento de
                                           blockchain inconsistente
1103                                    fmt.Println("Erro de
                                           recebimento de blockchain
                                           inconsistente")
1104                                    enviaMsg3(client, "ERROR 0\n
                                           ")
1105                                    return
1106                            }
1107                            //mutex.Unlock()
1108                    }
1109            }
1110 }
1111
1112 //Recupera a cadeia principal do servidor em caso onde estão
       incompatíveis
1113 //O inteiro i representa a partir de qual index onde serão
       pedidos os blocos
```

```
1114 func recuperaCadeiaPrincipal(client *Client, i int) (error) {
1115         var err error
1116         var res string
1117         err = nil
1118         fmt.Println("Entrando na função de recuperar cadeia
                ...")
1119
1120         //Blockchain temporária para realizar possíveis row
                backs
1121         var BlockchainTemp []Block
1122         var bloco Block
1123         res, err = enviaMsg2(client, "BLOCK BEFORE " +
                strconv.Itoa(i) + "\n")
1124         //Requisita blocos até encontrar a ligação com a
                blockchain local
1125         for ;i > 0 && i <= len(Blockchain); i-- {
1126                 if err != nil {
1127                         fmt.Println("Erro: ", err)
1128                         break
1129                 }
1130                 args := strings.SplitN(res, " ", 2)
1131                 op := strings.TrimSpace(args[0])
1132                 if op == "OK" {
1133                         //Continua executando o próximo bloco
1134                         continue
1135                 } else if op == "ERRO" {
1136                         //TODO Verificar isso
1137                         fmt.Println("Erro: erro de número ",
                                args[1])
1138                         return errors.New("erro de número " +
                                 args[1])
1139                 }
1140                 //Decodifica blcoo
1141                 bloco = decodifica([]byte(op))
1142
1143                 res, err = enviaMsg2(client,"OK\n")
1144
1145                 //Salva o último bloco requerido na
                        blockchain temporaria
1146                 BlockchainTemp = append(BlockchainTemp, bloco
                        )
1147
1148                 //Se verdade, encontramos de volta o elo de
                        ligação com a cadeia existente local
1149                 if isBlockValid(bloco, Blockchain[i-1]) ==
                        true {
1150                         fmt.Println("Elo de ligação
                                encontrado!")
1151                         imprimeBloco(bloco)
1152                         break
1153                 }
1154         }
1155
1156         //Após terminar o for, ou o bloco de ligação foi
```

```
            encontrado ou toda blockchain estava errada (pior
            caso)
1157
1158        //Caso especial onde é encontrado de cara
1159        if len(Blockchain) == i {
1160                Blockchain = append(Blockchain, bloco)
1161
1162        } else if i > 0 {
1163                //Retira da blockchain temporaria e coloca na
                        local
1164                for j := len(BlockchainTemp) - 1; i < len(
                    Blockchain) && j >= 0; j-- {
1165                        if isBlockValid(BlockchainTemp[j],
                            Blockchain[i-1]) == true {
1166                salvaTransacoesPerdidas(BlockchainTemp[j],
                    Blockchain[i])
1167                                Blockchain[i] =
                                    BlockchainTemp[j]
1168                                imprimeBloco(BlockchainTemp[j
                                    ])
1169                                salvaBloco(Blockchain[i])
1170                                i++
1171                        } else {
1172                                fmt.Println("Erro: blockchain
                                     temporaria inconsistente
                                    ")
1173                                return err
1174                        }
1175                }
1176        } else {
1177                //Neste caso mantém a blockchain local, pois
                    algo errado ocorreu
1178                fmt.Println("Erro: algo muito errado esta
                    acontecendo! (possível ataque)")
1179                fmt.Println("Erro: gênesis block recebido não
                     coincide")
1180                err = errors.New("Erro: gênesis block
                    recebido não coincide")
1181
1182        }
1183        return err
1184 }
1185
1186 //Faz a requisição de novos blocos ao servidor conectado
1187 //Chamada sempre que um servidor inicializa para fazer a
       atualização da blockchain
1188 func atualizaBlockchain(client *Client){
1189        var err error
1190        var res string
1191
1192        // Requisita do bloco de maior indice local até o
            mais recente validado no servidor
1193        //Caso exista blockchain local
1194        if Blockchain != nil && len(Blockchain) > 0 {
```

```
1195                inicio := Blockchain[len(Blockchain)-1].Head.
                       Index + 1
1196                fmt.Println("Sera pedido a partir de ",
                       inicio)
1197                res, err = enviaMsg2(client, "BLOCK SINCE " +
                        strconv.Itoa(inicio) + "\n")
1198                if err != nil {
1199                        fmt.Println("Erro: resposta fora do
                           padrão, erro não tratado ou nula\n
                           ", err)
1200                        closeClient(client)
1201                        return
1202                }
1203                //Recebe os blocos, valida e os adicina na
                       blockchain local
1204                for {
1205                //Quebra os argumentos
1206                args := strings.SplitN(res, " ", 2)
1207                //Limpa a msg
1208                        op := strings.TrimSpace(args[0])
1209                        fmt.Println(op)
1210                        if op == "END" {
1211                                //Verifica se o envio já foi
                                   concluído
1212                                //Neste caso retorna
1213                                //fmt.Println("Entrou no END
                                   ")
1214                                fmt.Println("Recebimento de
                                   blocos realizado com
                                   sucesso!")
1215                                return
1216                        } else if op == "ERRO" {
1217                                //TODO Verificar isso
1218                                fmt.Println("Erro: erro de nú
                                   mero ", args[1])
1219                                closeClient(client)
1220                                return
1221                        } else {
1222                                fmt.Println("Bloco recebido")
1223                        }
1224                        //Caso não, valida e adiciona o bloco
1225                bloco := decodifica([]byte(strings.TrimSpace(
                       res)))
1226
1227                        if isBlockValid(bloco, Blockchain[len
                           (Blockchain)-1]) == true {
1228                                Blockchain = append(
                                   Blockchain, bloco)
1229                                salvaBloco(bloco)
1230                                imprimeBloco(bloco)
1231                                res, err = enviaMsg2(client,
                                   "OK\n")
1232                                if err != nil {
1233                                        fmt.Println("Erro:
```

```
                                                    resposta fora do
                                                       padrão, erro não
                                                       tratado ou nula\n
                                                       ", err)
1234                                                closeClient(client)
1235                                                return
1236                                        }
1237                                } else {
1238                                        fmt.Println("Entrou no ELSE")
1239                                        //OBS: O único momento em que
                                                this else deve ocorrer é
                                               na primeira iteração do
                                               for
1240
1241                                        //Trata erro de blockchain
                                               incompatível com a versão
                                               do servidor
1242                                        //Envia um erro para
                                               sinalizar o problema e
                                               encerra esta requisição (
                                               ERRO ou END? estou na
                                               duvida)
1243                                        //TODO: Fazer mapeamento
                                               correto de códigos de erro
1244                                        enviaMsg3(client, "END\n")
1245
1246                                        //Faz a requisição de blocos
                                               anteriores até obter
                                               consistencia com o
                                               servidor
1247                                        if recuperaCadeiaPrincipal(
                                               client, inicio) == nil {
1248                                                //TODO se for
                                                       problema de
                                                       genesis block,
                                                       deve-se cancelar
                                                       conexão com o
                                                       servidor
1249                                                atualizaBlockchain(
                                                       client)
1250                                        }
1251                                        break
1252                                }
1253                        }
1254                } else {
1255                        fmt.Println("Solicitando toda a Blockchain")
1256                        //Caso a blockchain local não exista (
                               primeira inicialização)
1257                        solicitaBlockchain(client)
1258                }
1259 }
1260
1261 //Conecta com os servidores
1262 //Lê do arquivo contendo servidores conhecidos e o ID
```

```
    cadastrado neles
func openConn(){
        var err error
        flagN := true;
        var qtd int

        //Lê arquivo de servers para a memoria
        servers, err := parseData("servers.csv")
        if err != nil || len(servers) < 2 {
                fmt.Println("Erro: Abertura de arquivo de
                   servers")
                return
        }

        //Cria um map com o números não repetidos, até a
           quantidade de servideores salvos, para testar na
           lista de servidores
        var zero = struct{}{}
        randomList := make(map[int32]struct{}, len(servers))
        for i := int32(0); i < int32(len(servers)); i++ {
        randomList[i] = zero
        }

        //Tenta estabelecer conexão com 3 (inicialmente)
           servers na lista
        if OPENCONNS > (len(servers) - 1) {
                qtd = len(servers) - 1
        } else {
                qtd = OPENCONNS
        }

        //For que tenta conectar até chegar ao qtd desejado
        for i := 0; i < qtd; i++ {
                //Recupera um IP de servidor da tabela e
                   tenta conexão
                for j := range randomList {
                        //Pula o 0
                        if j == 0 {
                                continue
                        }

                        //Separa o ip no ":"
                        ip := strings.SplitN(servers[j][1],
                           ":", 2)
                        if len(ip) < 1 {
                        fmt.Println("Erro: IP lido fora do
                           padrão")
                        return
                    }

                    //Verifica se o ip nao é o local
                        _, ok := localAddrs[ip[0]]
                        //Se for pula esta conexão
                        if ok {
```

```
1309                                fmt.Println("Ip local
                                       encontrado: ", ip[0])
1310                                fmt.Println("Pulando conexão
                                       ")
1311                                continue
1312                        }
1313
1314                        //Verifica se não é o ip de uma conexão
                               vigente
1315                        mutexConnList.Lock()
1316                        _, ok = connectionsMap[ip[0]]
1317                            //Caso seja pula a conexão
1318                            if ok {
1319                                    fmt.Println("Pulando conexão
                                           ")
1320                                    mutexConnList.Unlock()
1321                                    continue
1322                            }
1323                            mutexConnList.Unlock()
1324
1325                        //Tenta a conexão
1326                        conn, err := net.DialTimeout("tcp", ip[0]
                            + ":8080", TIMEOUT)//net.Dial("tcp",
                               "192.168.0.35:8080")
1327                        //conn, err := net.Dial("tcp", servers[j
                               ][1])
1328                            if err != nil {
1329                                netErr, ok := err.(net.Error)
1330                                //Verifica se a conexão não teve
                                       timeout
1331                                if ok && netErr.Timeout() {
1332                                    //fmt.Println("Timed Out!")
1333                                    continue
1334                                } else {
1335                                    fmt.Println("Error: ", err)
1336                                    continue
1337                                }
1338                            }
1339
1340                            //Neste caso a conexão foi bem
                                   sucedida
1341                            //Double Check para conexão não nula
1342                            if conn != nil {
1343                                    fmt.Println("Conected!")
1344                                    //Envia o comando de abertura
1345                                    res, err := enviaMsg(conn, "
                                           OPEN " + servers[j][0] +
                                           "\n")
1346                                    if err != nil {
1347                                            fmt.Println("Error:
                                                   ", err)
1348                                            conn = nil
1349                                    }
1350                                    fmt.Println("Esperando
```

```
                                resposta do servidor...")
1351
1352                        //Quebra os argumentos
1353                        args := strings.SplitN(res, " ", 2)
1354                        //Limpa a msg
1355                                op := strings.TrimSpace(args
                                       [0])
1356                                fmt.Println(op)
1357
1358                                if op == "OK" { //Obsoleto
1359                                        fmt.Println("Validado
                                               no servidor com
                                               sucesso!")
1360
1361                                } else if op == "ERRO" {
1362                                        //TODO Verificar isso
1363                                        fmt.Println("Erro:
                                               erro de número ",
                                               args[1])
1364                                        conn = nil
1365
1366                                } else {
1367                                        //Decodifica a tabela
                                               recebida
1368                                        novaServers :=
                                               decodificaTabela
                                               ([]byte(op))
1369                                        if len(novaServers) >
                                               len(servers) {
1370                                                //Abre
                                                       arquivo
1371                                                f, err := os.
                                                       Create("
                                                       servers.
                                                       csv")
1372                                                if err != nil {
1373                                                        fmt.Println("
                                                               Erro:
                                                               Alterando
                                                               arquivo de
                                                                servers")
1374                                                return
1375                                                }
1376
1377                                                //Sobrescreve o
                                                       arquivo de
                                                       servers local
1378                                                csv.NewWriter(f).
                                                       WriteAll(
                                                       novaServers)
1379
1380                                                f.Close()
1381                                        }
1382                        }
```

```
1383
1384                                    if (conn != nil){
1385                                            //Salva a conexão em
                                                   um cliente do map
1386                                            c := newClient(conn)
1387                                            go clientListener(c,
                                                   1)
1388
1389                                            fmt.Println("
                                                   Atualizando
                                                   blockchain...")
1390                                            //Ver se deixa assim
                                                   msm
1391                                            c.Mutex.Lock()
1392                                            mutex.Lock()
1393                                            atualizaBlockchain(c)
1394                                            mutex.Unlock()
1395                                            c.Mutex.Unlock()
1396
1397                                            flagN = false
1398                                            break
1399                                    }
1400                            }
1401                    }
1402
1403                    //Verifica se conseguiu se conectar a algum
                           server
1404                    if flagN == false {
1405                            flagN = true
1406                            continue
1407                    } else {
1408                            i--
1409                            fmt.Println("Dormindo por 100
                                   segundos")
1410                            time.Sleep(100*time.Second)
1411                    }
1412            }
1413
1414        return
1415 }
1416
1417 //Função que envia toda a Blockchain
1418 func enviaBlockchain(client *Client){
1419            var err error
1420            //Envia blocos
1421            for _, bloco := range Blockchain {
1422                    fmt.Println("Enviando Bloco:")
1423                    imprimeBloco(bloco)
1424                    //Escreve o bloco
1425                    _, err = enviaMsg2(client, string(codifica(
                           bloco)) + "\n")
1426                    if err != nil {
1427                            // handle error
1428                            fmt.Println("Error: ", err)
```

```
1429                        return
1430                }
1431        }
1432
1433        //Escreve a msg de finalização de envio
1434        enviaMsg3(client, "END\n")
1435 }
1436
1437 //Recebe o index de inicio e fim solicitado
1438 func enviaBlocos(client *Client, inicio int, fim int){
1439        var err error
1440        end := fim
1441
1442        if Blockchain == nil {
1443                fmt.Println("Erro: Blockchain local não
                           existe")
1444                return
1445        }
1446
1447        //Caso index fim -1, envia do inicio até o bloco
                   atual
1448        if fim == -1 && inicio < len(Blockchain) {
1449                end = Blockchain[len(Blockchain)-1].Head.
                           Index
1450                fmt.Println(inicio, end)
1451        }
1452        //Verifica possiveis erros de index (caso a
                   blockchain nao esteja exatamente no index do vetor
                   ,
1453        //isto podera ser alterado)
1454        if fim > Blockchain[len(Blockchain)-1].Head.Index ||
                   inicio < Blockchain[0].Head.Index {
1455                fmt.Println("Erro: Requisição de bloco forma
                           do index")
1456                return
1457        }
1458        //Envia os blocos
1459        for i := inicio; i <= end; i++ {
1460                imprimeBloco(Blockchain[i])
1461                //Escreve o bloco
1462                _, err = enviaMsg2(client, string(codifica(
                           Blockchain[i])) + "\n")
1463                if err != nil {
1464                        // handle error
1465                        fmt.Println("Error: ", err)
1466                        return
1467                }
1468
1469        }
1470
1471        //Caso inicio == fim, a mensagem de END não é
                   esperada pelo cliente
1472        if inicio != fim {
1473                //Escreve a msg de finalização de envio
```

```
1474                    enviaMsg3(client, "END\n")
1475            }
1476    }
1477
1478    func open(client *Client, ID int) bool {
1479            flag := 0
1480            //buffer := make([]byte, 4096)
1481
1482            //Lê arquivo de servers para a memoria
1483            data, err := parseData("servers.csv")
1484            if err != nil || len(data) < 2 {
1485                    fmt.Println("Erro: Abertura de arquivo de
                           servers")
1486                    return false
1487            }
1488
1489            //Procura pelo ip na tabela e valida
1490            ender := client.Conn.RemoteAddr().String()
1491            args := strings.SplitN(ender, ":", 2)
1492            ip := strings.TrimSpace(args[0])
1493            for i := 1; i < len(data); i++ {
1494                    args2 := strings.SplitN(data[i][1], ":", 2)
1495                    ipTable := strings.TrimSpace(args2[0])
1496                    if ipTable == ip {
1497                            fmt.Println("Achado na tabela")
1498                            //Muda o flag pra indiacar que foi
                                   encontrado
1499                            flag = 1
1500                            break
1501                    }
1502            }
1503
1504            //Se não achou insere na tabela
1505            if flag == 0 {
1506                    //Captura o ultimo id e incrementa
1507                    newID, _ := strconv.Atoi(data[len(data)
                           -1][0])
1508                    newID++
1509                    fmt.Println(newID)
1510
1511                    //Recupera o ip e porta do cliente
1512                    d := []string{strconv.Itoa(newID), client.
                           Conn.RemoteAddr().String()}
1513                    fmt.Println(d)
1514                    //Insere na tabela carregadada na memória
1515                    data = append(data, d)
1516                    //Abre arquivo
1517                    f, err := os.Create("servers.csv")
1518                if err != nil {
1519                    fmt.Println("Erro: Abrindo arquivo")
1520                    return false
1521                }
1522                    //Salva alterações no arquivo
1523                    fmt.Println(data)
```

```
1524                    csv.NewWriter(f).WriteAll(data)
1525                    f.Close()
1526            }
1527
1528            //Envia tabela para o cliente
1529            enviaMsg3(client, string(codificaTabela(data)) + "\n
                   ")
1530            fmt.Println("Tabela enviada")
1531            return false
1532 }
1533
1534 //------------------------------Principais funções do Player
       -------------------------//
1535
1536 func login(username string, key string, client *Client) {
1537            //Adiciona player na variável global
1538            p, ok := playersMap[username];
1539            if !ok {
1540                    log.Fatal("Erro fatal: player não encontrado
                           ")
1541            }
1542            if p.PasswordHash != key {
1543                    log.Fatal("Erro fatal: senha não confere")
1544            }
1545            player.Username = p.Username
1546            player.PasswordHash = key
1547            player.LastSeqNumber = p.LastSeqNumber
1548            player.C = client
1549
1550            //Envia a mensagem de confirmação
1551            enviaMsg3(client, "OK\n")
1552            //loginChan <- true
1553            logged = true
1554 }
1555
1556 //Função que cuida dos procedimentos de entrada de novos
       players
1557 func register(client *Client, msg string) (error) {
1558            var p JSONPlayer
1559            var err error
1560
1561            args := strings.SplitN(msg, " ", 2)
1562
1563            //Descodifica JSON em player para verificar se está v
                   álido
1564            err = json.Unmarshal([]byte(strings.TrimSpace(args
                   [1])), &p)
1565            if err != nil {
1566                    fmt.Println("Error na decodificação do Json")
1567                    enviaMsg3(client, "ERROR 1\n") //Envia
                           mensagem de erro cancelando o registro
1568                    return err
1569            }
1570
```

```
1571         //Converte de JSONPLayer para Player
1572         var novoP Player
1573         novoP.Username = p.Username
1574         novoP.PasswordHash = p.PasswordHash
1575         novoP.Ether = p.Ether
1576         novoP.LastSeqNumber = 0
1577         //Adiciona player no map
1578         playersMap[p.Username] = &novoP
1579 
1580         //Coloca em uma transação
1581         var t Transaction
1582         t.Username = p.Username
1583         t.SeqNumber = 0
1584         t.Commands = []string{msg}
1585 
1586         //Adiciona no inicio da pool de transações
1587         mutexTransactionPool.Lock()
1588         transactionPool.PushFront(t)
1589         transactionPoolMap[calculateTransactionHash(t)] = t
1590         mutexTransactionPool.Unlock()
1591 
1592         //Mensagem de confirmação
1593         enviaMsg3(client, "OK\n")
1594 
1595         return err
1596 }
1597 
1598 //Envia o último jogo salvo considerado seguro
1599 func lastSavedGameState(client *Client){
1600         json := ""
1601 
1602         //Captura o estado do jogo no bloco seguro
1603         if len(Blockchain) > secureBlock {
1604                 json = Blockchain[len(Blockchain) -
                           secureBlock].GameState
1605         } else {
1606                 json = Blockchain[len(Blockchain) - 1].
                           GameState
1607         }
1608         if (json != ""){
1609                 //Envia a mensagem com o JSON
1610                 enviaMsg2(client, "GAMESTATE " + json + "\n")
                           ;
1611         } else {
1612                 //Caso existam inconsistencias neste bloco (
                           verificar)
1613                 fmt.Println("Erro: GameState inexistente no
                           bloco seguro!")
1614                 //Procura o último bloco que possui um
                           gamestate e envia
1615                 json = getLastSavedGame(len(Blockchain) - 1)
1616                 if json != "" {
1617                         fmt.Println("Encontrado GameState")
1618                         enviaMsg2(client, "GAMESTATE " + json
```

```
                                + "\n");
1619                    } else {
1620                            fmt.Println("Não foi encontrado
                                   GameState")
1621                            enviaMsg3(client, "NOT FOUND\n");
1622                    }
1623            }
1624 }
1625
1626 //------------------------------Funções de listener
       -------------------------//
1627
1628 //Controla as conexões existentes (listener de novos clientes
       )
1629 func clientServer(exit chan string){
1630             //Testanto conexão server
1631             ln, err := net.Listen("tcp", ":8080")
1632             if err != nil {
1633                     // handle erro
1634                     fmt.Println("Erro: Falha estabelecer porta
                           para conexão")
1635             }
1636
1637             for {
1638                     //Aceita conexões até atingir o limite de
                           MAXCONNS
1639                     if len(connectionsMap) == MAXCONNS {
1640                             <- unlockServer
1641                     }
1642
1643                     for i:=0; len(connectionsMap) < MAXCONNS; i++
                           {
1644                             fmt.Println("Esperando novos clientes
                                   ...")
1645
1646                     conn, err := ln.Accept()
1647                     if err != nil {
1648                                     // handle error
1649                                     fmt.Println("Erro: Falha
                                           estabelecer conexão")
1650                                     continue
1651                             }
1652
1653                             aux := conn.RemoteAddr().String()
1654                             fmt.Println("Conexão de: ", aux)
1655                 remoteIP := strings.SplitN(aux, ":", 2)
1656                 ip := remoteIP[0]
1657                     //Verifica se não é o player (conexão local)
1658                     _, ok := localAddrs[ip]
1659                             if ok {
1660                                     fmt.Println("Erro de
                                           tentativa de ip local")
1661                                     continue
1662                 }
```

```
1663
1664                            //Cria novo cliente
1665                    c := newClient(conn)
1666                    if c == nil {
1667                                    fmt.Println("Erro na criação
                                           de cliente")
1668                    closeClient(c)
1669                    continue
1670                }
1671
1672                            //Chama função para cuidar da conexão
1673                            go clientListener(c, i)
1674                            fmt.Println("Cliente Conectado com
                                   sucesso!")
1675                            fmt.Println("Número de clientes
                                   conectados: ", len(connectionsMap)
                                   )
1676                    }
1677
1678                    select {
1679                    case <- exit: {
1680                            return
1681                    }
1682                    default:
1683                            continue
1684                    }
1685            }
1686 }
1687
1688 //Função que aceita e controla players conectados (limitado a
       1)
1689 func playerServer(exit chan string){
1690            //Escuta a porta para conexão local
1691            ln, err := net.Listen("tcp", ":9090")
1692            if err != nil {
1693                    // handle erro
1694                    fmt.Println("Erro: Falha estabelecer porta
                           para conexão")
1695            }
1696
1697        for {
1698            //Aceita conexões
1699            conn, err := ln.Accept()
1700            if err != nil {
1701              // handle error
1702              fmt.Println("Erro: Falha estabelecer conexão")
1703              continue
1704
1705            }
1706
1707            aux := conn.RemoteAddr().String()
1708            remoteIP := strings.SplitN(aux, ":", 2)
1709            ip := remoteIP[0]
1710            //Verifica se é o player (conexão local)
```

```
1711          _, ok := localAddrs[ip]
1712          if !ok {
1713            fmt.Println("Cliente não é local! Pulando conexão
                 !")
1714            continue
1715          }
1716
1717          c := newClient(conn)
1718          if c == nil {
1719            fmt.Println("Erro: Falha na criação de um novo
                 Client")
1720            continue
1721          }
1722
1723                  player.C = c
1724
1725                  //Só aceita 1 por vez
1726                  playerListener()
1727
1728          select {
1729              case <- exit: {
1730                return
1731              }
1732                  default:
1733                          continue
1734          }
1735          }
1736 }
1737
1738 func clientListener(client *Client, i int) {
1739          r := bufio.NewReader(client.Conn)
1740          //Inicializa thread de tratamento de mensagem
1741          for{
1742                  fmt.Println("\nAguardando requisição de
                       cliente...")
1743                  //Aguarda a requisição do cliente
1744
1745                  message, err := r.ReadString('\n')
1746                  if err != nil {
1747                          //Encerra a conexão caso o cliente
                               caia/desconecte
1748                          if err == io.EOF {
1749                                  fmt.Println("Erro: resposta
                                       nula")
1750                          } else {
1751                                  fmt.Println("Erro: ", err)
1752                          }
1753                          //Encerra conexão com cliente
1754                          closeClient(client)
1755                          return
1756                  }
1757
1758          //Verifica se alguma thread está esperando uma
                 mensagem
```

```
1759         select {
1760                 //Caso esteja envia para a thread
1761             case <- client.Messages:
1762                 client.Messages <- message
1763
1764             default:
1765                 //Caso contrario envia mensagem recebida para
                        o handle
1766                     go handleConnectionMsgs(client,
                           message)
1767         }
1768         }
1769 }
1770
1771 func playerListener() {
1772         if player.C == nil{
1773                 fmt.Println("Erro: player desconectado!")
1774                 closeClient(player.C)
1775                 return
1776         }
1777         r := bufio.NewReader(player.C.Conn)
1778         //Inicializa thread de tratamento de mensagem
1779         for{
1780                 fmt.Println("\nAguardando requisição do
                       Player...")
1781                 //Aguarda a requisição do cliente
1782
1783                 message, err := r.ReadString('\n')
1784                 if err != nil {
1785                         //Encerra a conexão caso o cliente
                           caia/desconecte
1786                         if err == io.EOF {
1787                                 fmt.Println("Erro: resposta
                                   nula")
1788                         } else {
1789                                 fmt.Println("Erro: ", err)
1790                         }
1791                         //Encerra conexão com cliente
1792                         closeClient(player.C)
1793                         return
1794                 }
1795
1796         //Verifica se alguma thread está esperando uma
              mensagem
1797         select {
1798                 //Caso esteja envia para a thread
1799             case <- player.C.Messages:
1800                 fmt.Println("Enviando para thread")
1801                 player.C.Messages <- message
1802
1803             default:
1804                 fmt.Println("Criando thread para requisição
                       do player")
1805                 //Caso contrario envia mensagem recebida para
```

```
                        o handle
1806                        go handlePlayerMsgs(message)
1807        }
1808        }
1809 }
1810
1811 func handleConnectionMsgs(client *Client, message string) {
1812        //Trava o mutex para cuidar da requisição
1813        client.Mutex.Lock()
1814    fmt.Println("Mensagem recebida do cliente: ", client.IP)
1815    fmt.Println("Conteúdo: ", message)
1816
1817    //Separa a string no espaço
1818    args := strings.SplitN(message, " ", 2)
1819    if len(args) < 1 {
1820      fmt.Println("Erro: resposta fora do padrão")
1821      closeClient(client)
1822      client.Mutex.Unlock()
1823      return
1824    }
1825
1826    //Remove os espaços e alguns lixos que tenham sobrado na
         string
1827    op := strings.TrimSpace(args[0])
1828
1829    //Verifica a operação pedida pelo cliente e chama a funçã
         o correta para atende-lo
1830    switch op {
1831                case "OPEN":
1832                        fmt.Println("\nMensagem de abertura
                           de conexão recebida!")
1833                        id, _ := strconv.Atoi(strings.
                           TrimSpace(args[1]))
1834                        open(client, id)
1835
1836                case "BLOCK":
1837                        var inicio, fim int
1838                        var conector string
1839                        //Conta o número de espaços na
                           substring restante para saber qual
                            a mensagem
1840                        n := strings.Count(args[1], " ")
1841                        switch n {
1842                                //String: BLOCK ALL\n
1843                                //Caso em que todos os blocos
                                    são requeridos
1844                                case 0:
1845                                        enviaBlockchain(
                                           client)
1846
1847                                case 1:
1848                                        //String: BLOCK SINCE
                                            3\n
1849                                        //Caso peça de um
```

```
                                        determinado bloco
                                        até o atual
1850                                    if strings.SplitN(
                                        args[1], " ", 2)
                                        [0] == "SINCE" {
1851                                    fmt.Sscanf(args[1],
                                        "%s %d", &conector
                                        , &inicio)
1852                                    fmt.Println(message)
1853                                    enviaBlocos(client,
                                        inicio, -1)
1854                                    fmt.Println("Envio de
                                         blocos finalizado
                                        !")
1855                                    } else {
1856                                    //String: BLOCK
                                        BEFORE 3\n
1857                                    fmt.Sscanf(args[1],
                                        "%s %d", &conector
                                        , &inicio)
1858                                    fmt.Println(message)
1859                                    enviaBlocos(client,
                                        inicio, inicio)
1860                                    }
1861
1862                                //String: BLOCK 1 TO 3\n
1863                                //Caso onde uma fatia dos
                                       blocos são requeridos
1864                                case 2:
1865                                        fmt.Sscanf(args[1],
                                        "%d %s %d", &
                                        inicio, &conector,
                                         &fim)
1866                                        enviaBlocos(client,
                                        inicio, fim)
1867                        }
1868
1869                case "TRANSACTION":
1870                        //Envia confirmação
1871                        enviaMsg3(client, "OK\n")
1872
1873                        //Decodifica transação
1874                        t := decodificaTransaction([]byte(
                           strings.TrimSpace(args[1])))
1875
1876                        //Verifica a validade
1877                        if verificaTransacao(t) {
1878                                fmt.Println("Transação
                                   verificada")
1879                                mutexTransactionPool.Lock()
1880                                //Coloca no pool se estiver
                                   tudo correto
1881                                transactionPool.PushBack(t)
1882                                transactionPoolMap[
```

```
                                    calculateTransactionHash(t
                                    )] = t
1883                                mutexTransactionPool.Unlock()
1884                                //Distribui a mensagem para
                                    os demais clientes
1885                                distribuiMsg(client, message)
1886                        }
1887
1888                case "WIN":
1889                        fmt.Println("Recebido novo Bloco
                            forjado")
1890                        fmt.Println
                            ("================================
                             ", args)
1891                        b := decodifica([]byte(strings.
                            TrimSpace(args[1])))
1892
1893                        //Se for o próximo bloco esperado
1894                        mutex.Lock()
1895                        if isBlockValid(b, Blockchain[len(
                            Blockchain)-1]) == true {
1896                                //Se thread de forje viva,
                                    avisa que perdeu
1897                                if forging {
1898                                        fmt.Println("Travado
                                            no Lose")
1899                                        lose <- b
1900                                        fmt.Println("
                                            Destravado no lose
                                            ")
1901                                }
1902                                Blockchain = append(
                                    Blockchain, b)
1903                                salvaBloco(b)
1904                                gameState = getLastSavedGame(
                                    len(Blockchain) - 1)
1905                                carregaPlayersFromJson(
                                    gameState)
1906                                fmt.Println("_____________
                                    ________Bloco Vencedor____
                                    ___________________")
1907                                imprimeBloco(b)
1908                                go forge()
1909
1910                        //Se a cadeia principal local tiver
                            sido ultrapassada
1911                        } else if Blockchain[len(Blockchain)
                            -1].Head.Index < b.Head.Index {
1912                                fmt.Println("Bloco recebido
                                    maior que cadeia local")
1913                                  //Se thread de forje viva,
                                        avisa que perdeu
1914                                if forging {
1915                                        fmt.Println("Travado
```

```
                                                    no Lose")
1916                                              lose <- b
1917                                              fmt.Println("
                                                    Destravado no lose
                                                    ")
1918                                      }
1919                                      atualizaBlockchain(client)
1920                                      go forge()
1921                              } else {
1922                                      fmt.Println("Bloco recebido
                                            menor que a cadeia
                                            existente")
1923                              }
1924                              mutex.Unlock()
1925
1926       //String: CLOSE CONECTION\n
1927       //Caso seja o encerramento da conexão
1928       case "CLOSE":
1929           closeClient(client)
1930
1931       case "OK":
1932           select {
1933             case <- client.Messages:
1934             fmt.Println("WARNING: OK recebido no handle de
                    requisicões!!")
1935             client.Messages <- message
1936
1937             default:
1938             fmt.Println("ERROR: recebido OK fora de sincronia
                    !!")
1939           }
1940     }
1941         client.Mutex.Unlock()
1942 }
1943
1944 func handlePlayerMsgs(message string) {
1945         //Trava o mutex para cuidar da requisição
1946         player.C.Mutex.Lock()
1947     fmt.Println("Mensagem recebida do player")
1948     fmt.Println("Conteúdo: ", message)
1949
1950     //Separa a string no espaço
1951     args := strings.SplitN(message, " ", 2)
1952     if len(args) < 1 {
1953       fmt.Println("Erro: resposta fora do padrão")
1954           closeClient(player.C)
1955           player.C = nil
1956       return
1957     }
1958
1959     //Remove os espaços e alguns lixos que tenham sobrado na
          string
1960     op := strings.TrimSpace(args[0])
1961
```

```
1962    //Verifica a operação pedida pelo cliente e chama a função correta para atende-lo
1963    switch op {
1964
1965                case "LOGIN":
1966                        fmt.Println("Realizando o Login...")
1967                        args2 := strings.SplitN(args[1], " ", 2)
1968                        login(args2[0], strings.TrimSpace(args2[1]), player.C)
1969                        fmt.Println("Login Concluido!")
1970                        go forge()
1971
1972                case "REGISTER":
1973                        fmt.Println("Register")
1974                        register(player.C, strings.TrimSpace(message))
1975                        distribuiMsg(player.C, message)
1976
1977                case "SAVED": //SAVED GAMESTATE\n
1978                        //Envia o último jogo salvo ao cliente jogador
1979                        lastSavedGameState(player.C)
1980                        enviaComandosParaJogo(Blockchain[len(Blockchain)-1])
1981
1982                case "COMMAND":
1983                        fmt.Println("Recebido um novo comando do Jogo")
1984                        if !logged {
1985                                enviaMsg3(player.C, "ERROR 0\n") //Envia mensagem de ERROR não logado
1986                        } else {
1987                                fmt.Println("Enviando ACK de comando recebido")
1988                                enviaMsg3(player.C, "OK\n") //Envia mensagem de confirmação
1989                                mutexCommandPool.Lock()
1990                                commandPool.PushBack(strings.TrimSpace(args[1])) //Coloca o comando no pool
1991                                mutexCommandPool.Unlock()
1992                        }
1993
1994                case "GAMESTATE":
1995                        fmt.Println("Recebido um GAMESTATE")
1996                        var m JSONMessage
1997                        //Decodifica a mensagem em JSON
1998                        jsonS := strings.TrimSpace(args[1])
1999                        err := json.Unmarshal([]byte(jsonS), &m)
2000                        if err != nil {
```

```
2001                                    fmt.Println("Error na
                                           decodificação do Json")
2002                            } else {
2003                                    gameState = jsonS
2004                            }
2005
2006                            if carregaPlayersFromJson(gameState)
                                   {
2007                                    enviaMsg3(player.C, "OK\n")
2008                            } else {
2009                                    fmt.Println("Erro: Erro no
                                           carregamento de players
                                           pelo gamestate recebido\
                                           nEnviando mensagem de erro
                                           ")
2010                                    enviaMsg3(player.C, "ERROR 1\
                                           n")
2011                            }
2012
2013                    case "BET":
2014                            fmt.Println("Recebido um BET")
2015                    if !logged {
2016                            enviaMsg3(player.C, "ERROR 0\n") //
                                   Envia mensagem de ERROR não logado
2017                    } else {
2018                                    //Envia ACK
2019                                    enviaMsg3(player.C, "OK\n")
                                           //Envia mensagem de
                                           confirmação
2020
2021                                    mutexCommandPool.Lock()
2022                                    commandPool.PushFront(strings
                                           .TrimSpace(message)) //
                                           Coloca o Bet no inicio do
                                           pool
2023                                    mutexCommandPool.Unlock()
2024
2025                                    //Cria uma transação com os
                                           comandos
2026                                    t, ok :=
                                           transactionFromCommPool()
2027                                    if !ok {
2028                                            log.Fatal("Erro:
                                                   Comandos não foram
                                                    convertidos e
                                                   copiados,
                                                   abortando!")
2029                                    }
2030                                    //Envia para os demais
                                           clientes
2031                                    fmt.Println("Enviando para
                                           outros players")
2032                                    floodTransaction(player.C, t)
2033                                    //Coloca transação como
```

```
                                        primeira na pool
2034                                    mutexTransactionPool.Lock()
2035                                    transactionPool.PushFront(t)
2036                                    transactionPoolMap[
                                            calculateTransactionHash(t
                                            )] = t
2037                                    mutexTransactionPool.Unlock()
2038
2039                                    //bet <- 1
2040                            }
2041
2042                    case "LOGOUT":
2043                            closeClient(player.C)
2044                            player.C = nil
2045
2046                    default:
2047                fmt.Println("Erro: mensagem inválida")
2048            }
2049            player.C.Mutex.Unlock()
2050 }
2051
2052 func interfaceTeste() {
2053            var i int
2054            for{
2055                    fmt.Scanf("%d", &i)
2056                    switch i {
2057                    case 1 :
2058                            //openConn(conn)
2059                            fmt.Println("Clientes conectados: ")
2060                            for key, _ := range connectionsMap {
2061                                    fmt.Println("Cliente ", key)
2062                            }
2063                            fmt.Println("Fim de Operação")
2064                    case 2 :
2065
2066                    case 3 :
2067                            fmt.Println("Interfaces de rede do
                                    computador: ")
2068                            addrs, err := net.InterfaceAddrs()
2069                            if err != nil {
2070                                panic(err)
2071                            }
2072                            for i, addr := range addrs {
2073                                fmt.Printf("%d %s\n", i, addr.
                                        String())
2074                            }
2075                                                    //lose <-
                                                        true
2076                            fmt.Println("Fim de Operação")
2077                    case 4 :
2078                            fmt.Println("Comandos no pool: ")
2079                            for c := commandPool.Front(); c !=
                                nil; c = c.Next() {
2080                                    fmt.Println(c.Value)
```

```
2081                        }
2082                        fmt.Println("Fim de Operação")
2083
2084                case 5 :
2085                        b := codificaComandos(Blockchain[len(
                               Blockchain)-1].Transactions[0].
                               Commands)
2086                        fmt.Println("Codificação dos comandos
                                do último bloco: ", b)
2087                        fmt.Println("Decodificação: ")
2088                        list := decodificaComandos(b)
2089                        for e := list.Front(); e != nil; e =
                               e.Next() {
2090                        fmt.Println(e.Value.(string))
2091                    }
2092                        fmt.Println("Fim de Operação")
2093
2094                case 6 :
2095                        msg:= "COMMAND " + player.Username +
                               " CHANGE NOT_DEFINED 1 TO WORKER_
                               STONE IN 0\n"
2096                        fmt.Println("Enviando comando: " +
                               msg + " do player: " + player.
                               Username)
2097                        player.C.Mutex.Lock()
2098                        enviaMsg2(player.C, msg)
2099                        player.C.Mutex.Unlock()
2100
2101                case 7 :
2102                        for _, p := range playersMap {
2103                                fmt.Println("Lista de Players
                                    connhecidos:")
2104                                fmt.Println("   Login ", p.
                                   Username)
2105                                fmt.Println("   LastSegNumber
                                    ", p.LastSeqNumber)
2106                        }
2107
2108                case 8 :
2109                        for t := transactionPool.Front(); t
                               != nil; t = t.Next() {
2110                                fmt.Println("Lista de Transaç
                                   ões no pool:")
2111                                fmt.Println("   Transaction")
2112                                fmt.Println("
                                   Username: ", t.Value.(*
                                   Transaction).Username)
2113                                fmt.Println("
                                   SeqNumber: ", t.Value.(*
                                   Transaction).SeqNumber)
2114                                fmt.Println("
                                   Commands: ")
2115                                if t.Value.(Transaction).
                                   Commands != nil {
```

```
2116                                            for _, c := range t.
                                                   Value.(Transaction
                                                   ).Commands {
2117                                                    fmt.Println("


                                                                    ",
                                                                c)
2118                                            }
2119                                    }
2120                            }
2121
2122                    case 9:
2123                            //enviaMsg2(player.C, "OK\n")
2124
2125                            /*
2126                            var tt Transaction
2127                            tt.Username = "caio"
2128                            tt.SeqNumber = 9
2129                            floodTransaction(player.C, Blockchain
                                   [len(Blockchain)-2].Transactions
                                   [0])
2130                            */
2131                            distribuiBloco(player.C, Blockchain[
                                   len(Blockchain)-2])
2132
2133                    default:
2134
2135                    }
2136            }
2137 }
2138
2139 func main(){
2140            Blockchain = nil
2141            logged = false
2142            forging = false
2143
2144
2145            //generateGenesisBlock()
2146
2147            //Cria um canal de saída
2148            exit := make(chan string)
2149
2150            //Carrega blockchain local na memória
2151            Blockchain = carregaBlockchain()
2152
2153            fillLocalAddrs()
2154
2155            go interfaceTeste()
2156
2157            go openConn()
2158
2159            //Caso a blockchain seja nula, nada pode ser feito at
                   é conseguir baixá-la de algum servidor
```

```
2160         if Blockchain == nil {
2161                 //Função de solicitar blockchain escreve no
                        channel
2162                 <-att
2163         }
2164
2165         go clientServer(exit)
2166
2167         go playerServer(exit)
2168
2169         go forge()
2170
2171         //Verifica se a rotina terminou e encessa
2172         for {
2173           select {
2174             case <- exit: {
2175               os.Exit(0)
2176             }
2177           }
2178         }
2179
2180 }
```