Leonardo Dantas Gouveia

Blockchain

Campina Grande - PB

Março de 2021

Leonardo Dantas Gouveia

Blockchain

Trabalho de Conclusão de Curso submetido à Unidade Acadêmica de Engenharia Elétrica da Universidade Federal de Campina Grande como parte dos requisitos necessários para a obtenção do grau de Bacharel em Engenharia Elétrica. Área de Concentração: Computação Distribuída e Segurança da Informação.

Universidade Federal de Campina Grande – UFCG

Centro de Engenharia Elétrica e Informática

Graduação em Engenharia Elétrica

Orientador: Prof. Dr. José Ewerton Pombo de Farias

Coorientador: Prof. Dr. Leocarlos Bezerra da Silva Lima

Campina Grande - PB

Março de 2021

Resumo

O presente Trabalho de Conclusão de Curso apresenta uma visão geral de sistemas *blockchain* [1], bem como uma discussão sobre aplicações dessa tecnologia, com destaque para as criptomoedas. A análise desenvolvida contém: (a) o funcionamento de sistemas *blockchain*, (b) suas vantagens e desvantagens e (c) um exemplo simples de mineração — atividade importante no funcionamento de várias criptomoedas.

Palavras-chaves: *blockchain*, criptomoedas.

[1] Ao pé da letra: cadeia de blocos. No contexto deste TCC: sistemas constituídos por uma sucessão de registros ou conjunto de informações registradas que é ligado a blocos de informações anteriores e sucessores ou processo que ocorre em sucessão, de forma sequencial e interligada.

Lista de ilustrações

Sumário

Introdução

O conceito de *Blockchain* não é novo, foi proposto inicialmente na década de 90 por criptógrafos como uma forma de validar transações em um grupo de pessoas e verificar a legitimidade de documentos compartilhados. Apesar do desenvolvimento inicial na área, muito pouca coisa surgiu que se utilizasse desse conceito até que no ano de 2008 foi feita uma postagem na lista de correio eletrônico metzdowd.com contendo um endereço para um domínio (bitcoin.com) que por sua vez continha um artigo pdf com as bases para um sistema monetário descentralizado. O artigo se intitulava *"Bitcoin: A Peer-to-Peer Electronic Cash System"* e seu autor utilizava o pseudônimo Satoshi Nakamoto.

Desde então o Bitcoin cresceu exponencialmente, de um uso tímido no início da década de 2010 e valor quase nulo para centenas de milhares de transações por dia e uma massiva apreciação de valor sendo cotada atualmente em mais de dez mil dólares americanos. Após o sucesso inicial, outras criptomoedas tais como Nano e Ethereum começam a surgir como alternativas se propondo a resolver problemas técnicos associados ao Bitcoin estabelecendo as criptomoedas como uma alternativa ao dinheiro tradicional.

Não há dúvidas de que hoje as criptomoedas são um uso legítimo da tecnologia *Blockchain* mas elas representam apenas uma das muitas aplicações possíveis. *Blockchain* é um termo utilizado para se referir a um registro compartilhado por vários usuários de uma rede onde a verificação da validade de cada entrada (ou bloco) é feito com base nos blocos anteriores da cadeia. A segurança do processo é garantida por meio da criptografia que é utilizada não só para verificação da integridade de cada bloco da cadeia como para validar as transações dos usuários e impedir interferências maliciosas.

Esta tecnologia tem aplicação em qualquer organização preocupada com a segurança de uma quantidade crescente de dados (governos ou empresas). A solução padrão é centralizar esses dados em um administrador que se encarrega de garantir a validade/registro dos dados inseridos e a segurança dos usuários participantes da rede. Esta solução, no entanto, apresenta uma série de problemas característicos como a sobrecarga do servidor central em caso de uso simultâneo ou a possibilidade de fraude caso o administrador do sistema fosse comprometido. A solução do *Blockchain* é, paradoxalmente, distribuir a informação na rede, tornando milhões de usuários em verificadores individuais.

1 A Função *Hash*

> "Yao Fu não escreve poesia só por diversão"
>
> *Shao Yung em "Poemas escritos enquanto se bate no chão".*

Uma função *hash* é uma função matemática que transforma uma mensagem de tamanho qualquer em um número de tamanho fixo que chamamos de *hash*(6). Por exemplo, a função MD5 transforma a mensagem "Ola mundo" na *hash* "eea5849b67308e7cfb3b0bfab52d30ea". O tamanho da *hash* depende do padrão utilizado, no caso do MD5 a *hash* gerada sempre tem tamanho igual a 128 bits. A seguir temos uma pequena tabela expondo alguns algoritmos de *hash* comuns:

Padrão	Tamanho da Hash
MD5	128 bits
SHA-1	160 bits
SHA-224	224 bits
SHA-256	256 bits
SHA-384	384 bits
SHA-512	512 bits
SHA-3	arbitrário

Tabela 1 – Funções *Hash*.

Uma característica interessante destas funções é que, caso um único caractere seja removido, adicionado ou alterado na mensagem, a saída gerada para esta nova mensagem será diferente da saída da mensagem original. Por exemplo, vamos alterar uma única letra na mensagem do exemplo anterior fazendo com que a mensagem "Ola mundo" passe a ser "ola mundo", a *hash* da nova mensagem calculada pela função MD5 passa a ser "3b2613ff007c695c2d560d0e9c9ccbcf" que é completamente diferente do resultado obtido anteriormente, isso deve se verificar para qualquer mudança por mais diminuta que seja.

As funções *hash* possuem, graças às suas caracteristicas, várias aplicações de segurança na rede tais como:

1. Verificação da Integridade de Arquivos: Quando uma informação qualquer atravessa a rede mundial de computadores, sempre vai haver possibilidade dessa informação ser corrompida na viagem, certos pacotes serem perdidos ou até mesmo um programa malicioso ser enviado tentando se passar por um programa legítimo. Uma forma de

evitar tais complicações é o provedor do programa fornecer a *hash* desse programa de forma que o destinatário possa verificar por si só a integridade e a autenticidade da informação recebida.

2. Armazenamento de Senhas: Em muitos sites é comum a utilização de senhas para autenticação do usuário. Seria, no entanto, um grande risco para estes usuários caso os servidores deste site armazenassem suas senhas em texto aberto uma vez que um atacante ou qualquer pessoa que tivesse acesso a estas informações poderia facilmente utilizá-las para invadir as contas e causar todo tipo de problemas. Assim sendo, é comum que servidores de email, redes sociais e qualquer outro serviço que requerem login, armazenem as senhas de seus usuários em uma "forma hasheada", ou seja, a *hash* de cada senha. Nesta situação, mesmo que os dados do servidor se tornem públicos, um criminoso só teria acesso às *hashes* das senhas dos usuários e não às senhas em si.

3. Confecção de Redes *Blockchain*: As funções *hash* também são essenciais na confecção de sistemas *blockchain*, isto ficará bastante evidente ao longo do trabalho. Uma implementação da função SHA-256 em linguagem C++, que será utilizada nos exemplos pode ser encontrada no anexo.

É importante ressaltar que o processo de "*hashing*" é irreversível, ou seja, não existe uma função "*hash* reversa" que possamos aplicar em uma *hash* para obter de volta a mensagem original(6). Para todos os efeitos práticos, esta foi completamente destruída pelo algoritmo do *hash*. Outro ponto importante é que o *hash* de uma mensagem é imprevisível. Isto significa que não é possível saber características e muito menos prever a *hash* de uma mensagem andes de ela ser calculada(6). Esta última propriedade faz com que um atacante que busque gerar uma mensagem falsa que possua a mesma *hash* da mensagem verdadeira não tenha outra escolha a não ser tentativa e erro até que sua mensagem fraudulenta gere a *hash* desejada. Como existem muitas *hashes* possíveis, tal processo levaria muito mais tempo do que uma existência humana.

2 O que é uma *Blockchain*

Atualmente é bastante comum que nossas informações estejam armazenadas em servidores ao redor do mundo. Os registros de nossas compras online são armazenados nos servidores do site em que a compra foi realizada, os registros bancarios são armazenados nos servidores do banco e nossas informações pessoais são armazenadas em servidores do governo (ou em cartórios, para o caso de países menos civilizados). Todos estes casos têm em comum o fato de que os dados de vários usuários estão armazenados em um lugar da rede controlado por uma entidade que pode, caso lhe convenha, se utilizar dessa informação para benefício próprio em detrimento do usuário.

Sistemas centralizados são, por definição, extremamente dependentes de uma única entidade que se supõe confiável. Se um sistema central tal como um banco for comprometido, seja por fraude, ciberataque ou um mero erro todo o sistema será afetado. Portanto faz-se necessária uma nova forma de armazenamento de dados importantes. A arquitetura *blockchain* dá aos usuários a posse de um registro que pode conter desde transações financeiras até contratos trabalhistas ou transferência de bens. Tal registro não é armazenado em nenhum servidor central. No lugar disso cada nó possui uma cópia idêntica do registro e compara seu registro com outros nós de maneira *peer-to-peer*.

Para entender o porquê de se usar uma *blockchain*, é preciso primeiro entender o que é uma *blockchain*. Uma *blockchain* é um conjunto de dados ligados por um esquema de *hashes*(2) com cada unidade de informação na *blockchain* sendo chamado de bloco. O primeiro bloco de uma *blockchain* poderia ser algo do tipo:

Figura 1 – Bloco Gênesis.

Esse bloco contém em seu interior um conjunto de transações quaisquer que representamos por linhas de asteriscos “*********”. Por fim, passamos o conteúdo do nosso bloco (ie. as transações) em uma função *hash* previamente definida (MD5, SHA-2, etc) e a *hash* resultante é inserida no final do bloco(7). A este primeiro bloco damos o nome de Bloco Gênesis, todos os demais blocos estarão ligados a ele por uma longa cadeia.

Agora que temos o primeiro bloco podemos iniciar a construção da cadeia. O nosso segundo bloco vai ter uma composição bastante similiar ao primeiro, exceto que também iremos adicionar em seu início a *hash* do bloco anterior além, é claro, das transações nele contidas serem (muito provavelmente, mas não necessariamente) diferentes das do bloco anterior(7).

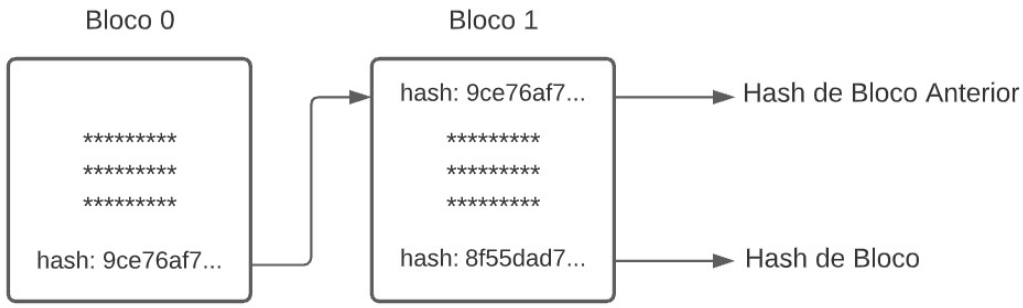

Figura 2 – Cadeia de dois blocos.

Perceba que a partir do segundo bloco a *hash* é calculada usando o conteúdo do bloco e a *hash* do bloco anterior. Isso ocorre para evitar que existam dois blocos consecutivos com *hashes* idênticas caso estes blocos apresentem a mesma lista de transações. Outro motivo pelo qual isso é feito é para dificultar as bifurcações que são ramificações da *blockchain*, que produzem uma cadeia paralela que causam problemas à rede(3). O processo pelo qual isso ocorre ficará mais evidente nos capítulos subsequentes.

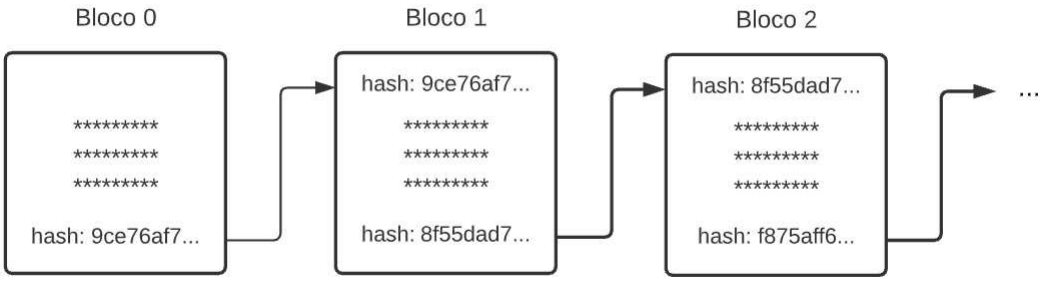

Figura 3 – Estrutura da *blockchain*.

3 Topologias de Rede

Agora que sabemos como se constrói uma *blockchain* básica, precisamos saber como ela se organiza. A topologia de uma rede define quem é autorizado a adicionar blocos à cadeia e sugere como o consenso é atingido caso um conflito ocorra. Ao longo deste capítulo iremos utilizar círculos amarelos para indicar nós de maior importância para a rede e os nós azuis representam nós periféricos cuja única função é servir de repositório para a *blockchain* e executar tarefas simples para os usuários além de verificar a validade das transações. . Existem três topologias básicas que diferenciam os participantes da rede.

3.1 Rede Centralizada

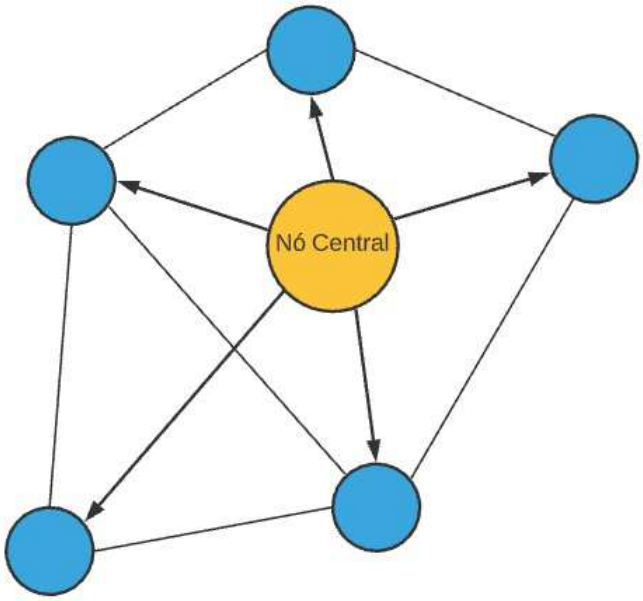

Figura 4 – Rede Centralizada. As setas indicam o fluxo de informação.

Uma rede *blockchain* centralizada se refere a um tipo de topologia de rede que põe um único nó responsável pela atualização da *blockchain* e pela confirmação das transações feitas pelos nós periféricos(1). Esse estilo de rede é relativamente raro porque oferece poucas vantagens em relação ao sistema de armazenamento tradicional, os pontos de falha aqui também serão extremamente limitados sendo necessário somente que o nó central seja comprometido para que toda a rede fique inoperante.

Existem, no entanto, algumas vantagens em se utilizar uma *blockchain* dessa forma. A primeira grande vantagem é a transparência. Em uma *blockchain*, todas as transações

são públicas e cada nó possui uma cópia de todos os registros, isso significa que o nó central não pode alterar nada na *blockchain* sem que a alteração fique óbvia para toda a rede. A segunda vantagem é a segurança. Como vimos, um bloco da *blockchain* uma vez inserido não é facilmente removido ou modificado. Além de ser óbvia, qualquer alteração no registro iria requerer enorme esforço computacional. A terceira e última vantagem é a redundância. Mesmo para o caso onde o nó central seja completamente destruído e seu registro excluído, os demais membros da rede poderiam facilmente fornecer um *backup*. Isso não seria possível em sistemas tradicionais, não sem que o administrador do sistema tivesse feito *backup* ele mesmo.

Esta topologia é ideal para organizações, empresas ou governos que desejam possuir alguma forma de controle do sistema e, ainda ter todas a segurança que uma *blockchain* proporciona. Neste modelo uma autoridade central pode unilateralmente decidir quais usuários podem participar da rede e quais transações serão incluídas e em qual ordem. Um banco central pode utilizar este modelo para implementar uma versão digital da moeda de um país. De fato, a digitalização da economia é uma tendência observada no mundo inteiro(4).

Um sistema *blockchain* centralizado também é útil para evitar os chamados ataques de repúdio. Este tipo de ataque ocorre quando o usuário executa uma ação no sistema e depois alega que não realizou tal ação com o objetivo de obter algum tipo de conpensação financeira. Se registrada em uma *blockchain*, as ações dos usuários tornam-se permanentes já que, para alegar que não fez uma ação, o usuário estaria alegando que o bloco onde aquela ação foi inserida é inválido, ou seja, foi inserido posteriormente sem o seu consentimento. Sabemos que isso é impossível, porque alegar que um bloco é invalido é o mesmo que alegar que todos os blocos que o seguem foram produzidos a partir daquele bloco inválido e consequentemente, teriam *hashes* totalmente diferentes(6).

3.2 Rede Hierárquica

Nessa topologia, certos nós são especializados em exercer determinadas funções na rede. Um exemplo de aplicação de redes hierárquicas é o Ripple. No Ripple, os nós são divididos em *tracking* e validação. Os nós de *tracking* são responsáveis pela coleta de transações da rede ou executar funções no registro, tais como recuperar o balanço de um determinado cliente. Eles também são responsáveis por compartilhar as transações com a rede em geral. Os nós de validação exercem as mesmas funções que os nós de *tracking*, mas com direito de voto e autoridade para tomar decisões relativas a taxas(1).

3.3 Rede Descentralizada

Este é o tipo mais comum de topologia de rede. Ela tira toda a vantagem da tecnologia *blockchain* e apresenta o maior nível de segurança dentre as listadas anteriormente. Uma rede descentralizada é extremamente resiliente e segura se projetada corretamente e essas características tornam-se mais fortes à medida que a rede cresce. Por motivos óbvios, quase todas as criptomoedas se utilizam de *blockchains* descentralizadas na sua implementação.

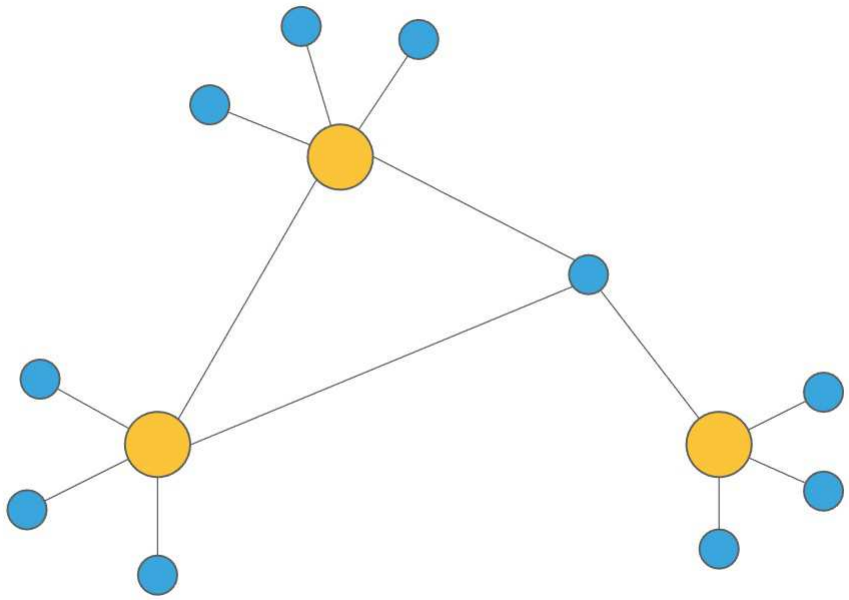

Figura 5 – Rede Descentralizada.

Primeiramente, é importante fazermos uma distinção entre sistemas distribuídos e descentralizados. Um sistema distribuído é um sistema em que os nós estão conectados de maneira aleatória e exercem essencialmente a mesma função sobre a rede que os demais nós, um exemplo disso é o *torrent*. Ao baixar um arquivo pelo *torrent*, o usuário que fez o download tem a opção de se tornar um contribuinte da rede ajudando outras pessoas que desejam baixar o mesmo arquivo. Não há em sistemas distribuidos nenhuma hierarquia entre os nós, todos são iguais. Sistemas descentralizados, por outro lado, podem apresentar certo nível de hierarquia entre seus nós, mas isso emerge não por pré determinação, mas por simples organização expontânea da rede. No caso do Bitcoin por exemplo, os nós com maior poder computacional tornam-se mineradores enquanto que os demais exercem a função de "*full nodes*", que são nós que realizam a verificação e retransmissão das transações além de manter uma cópia da *blockchain*.

As vantagens da rede descentralizada ficarão evidentes quando estudarmos as criptomoedas, mas já podemos adiantar que um atacante precisaria dedicar uma enorme

quantidade de recursos para, no máximo, causar um distúrbio momentâneo na rede. As desvantagens, por outro lado, são bem mais evidentes em redes descentralizadas que nas demais. Isso ocorre porque qualquer mudança nas regras de interação entre os nós para corrigir um problema qualquer deve ter a aprovação da esmagadora maioria dos usuários desse sistema. O consenso está no cerne da rede. Felizmente, no caso das criptomoedas, a maioria dos participantes da rede são os usuários dessa criptomoeda, o que faz com que quase todas as mudanças que ocorrem nas regras da *blockchain* favorecem mais aos usuários do que aqueles que ganham diretamente com a operação do sistema (mineradores, por exemplo).

4 O Processo de Validação

Quando falamos em validação, nos referimos ao processo que permite a uma transação ou um bloco ser considerado seguro para ser incluído na *blockchain*. Existem dois tipos principais de validação: de transação e de bloco. As transações, no geral, são validadas por meio de criptografia assimétrica também conhecida como criptografia de chave pública. A matemática por trás dessa criptografia está fora do escopo deste trabalho, mas podemos dizer que em um sistema *blockchain* os usuários são identificados por meio de chaves públicas. Em sistemas descentralizados, isso é normalmente tudo o que se sabe sobre o usuário, não é possível determinar sequer sua identidade. Só a chave pública é conhecida e mais nada.

A chave pública, por sua vez, é gerada a partir de uma chave privada que é utilizada juntamente com uma função matemática para "assinar" uma transação. A chave privada tem o poder de validar qualquer transação associada à sua chave pública e, portanto, deve ser conhecida somente pelo proprietário dela. O processo de "assinatura" é feito de tal forma que é impossível determinar a chave privada por meio da transação assinada por ela. Esta portanto, é segura para divulgação na rede. A assinatura também deixa óbvio para todos que o detentor da chave privada relacionada àquela chave pública é o único responsável possível pela transação. A transação, uma vez assinada e divulgada, é então combinada pelo verificador com a chave pública do usuário para produzir uma transação válida legível que é então transmitida a outros verificadores até ser aceita pela rede(3)(7).

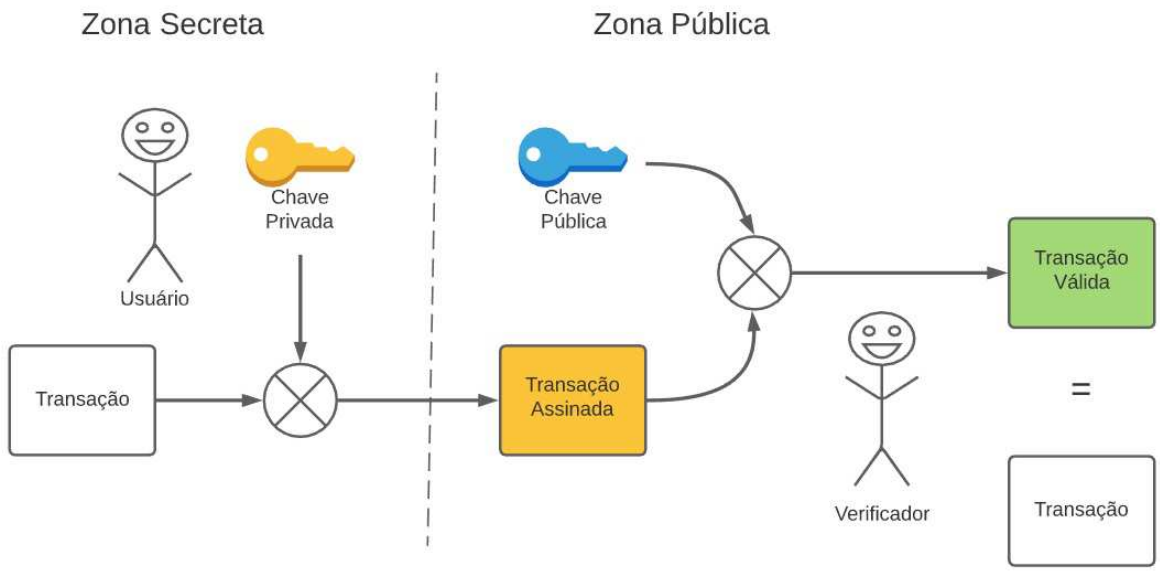

Figura 6 – Verificação de Transação.

Criptografia assimétrica recebe esse nome porque a chave usada na encriptação é diferente da usada na decriptação. Quando falamos em "assinatura", nos referimos a esse processo. Começamos por encriptar a transação com a chave privada e depois transmiti-la à rede. Os demais nós irão assim tentar decriptar a transação usando a chave pública do usuário que supostamente transmitiu a transação. Caso a decriptação seja bem sucedida podemos concluir que a transação foi, de fato, feita pelo usuário e é portanto válida[1].

A validação de blocos é mais dependente do sistema e sua aplicação. Enquanto redes centralizadas raramente precisam de uma validação de bloco robusta (o nó central desempenha o papel de validador), redes descentralizadas requerem um nível de validação mais independentes de nós individuais. A solução encontrada é fazer com que para validar um bloco o nó validador tem que provar algo para a rede. A prova a ser fornecida depende do sistema e geralmente está implícita no bloco que foi adicionado à cadeia. As formas de validação de bloco comumente usadas são:

1. ***Proof of Work(PoW)***: A mais comum, *proof of work* faz com que um verificador que deseja inserir um novo bloco na cadeia precise gerar um bloco cuja *hash* atenda um certo critério que, normalmente, exige muito poder computacional. No caso do Bitcoin, a *hash* gerada tem que ser menor que um valor pré estabelecido e é obtida inserindo no bloco um número aleatório qualquer, calculando sua *hash*, e testando a validade dessa *hash*. Em outras palavras, tentativa e erro. *Proof-of-work* apresenta algumas desvantagens, a mais óbvia é o desperdício da energia elétrica usada nas CPUs, que calculam inúmeras *hashes* até que uma delas, a *golden hash*, seja válida.

   Além do desperdício de energia na solução de um problema matemático inútil, *proof of work* é sujeito a ataques coordenados de validadores que podem usar seu poder dentro da rede para negar serviço a um determinado usuário. Mesmo assim, fraudes seriam impossíveis de serem feitas mesmo pelo esforço combinado da maioria dos verificadores de blocos, pois os blocos e transações fraudulentas seriam prontamente rejeitadas pelo restante da rede.

2. ***Proof of Stake(PoS)***: Leva em consideração a quantidade de recursos pertencentes a cada um dos nós e o quanto o usuário por trás de cada nó perderia caso houvesse uma falha no sistema. Para o caso de uma criptomoeda assume-se que aqueles nós que possuem a maior quantidade de criptomoeda são justamente os mais interessados no correto funcionamento do sistema, uma vez que sofrerão maior prejuízo em caso de falha. Na *proof of stake*, os usuários mais "ricos" votam com maior peso na validação de blocos na *blockchain*. Sistemas *proof of stake* são vulneráveis ao ataque *nothing at stake*, que será discutido posteriormente.

[1] Na prática, é a hash da transação que é encriptada enquanto que a transação em si é divulgada em texto aberto

Apesar de resolver o desperdício observado na *proof of work*, o *proof of stake* apresenta problemas sérios de confiabilidade, uma vez que um usuário que possui grande quantidade de recursos na *blockchain* não necessariamente possui a maior parte de seus recursos totais investidos na rede e pode se utilizar de sua posição privilegiada para realizar operações fraudulentas ou para negar transações válidas de usuários específicos. Por este motivo, é comum que sistemas que utilizem *proof-of-stake* também implementem punições por comportamento malicioso. Um verificador que auxiliar outro usuário em um ataque de *double spending* (ver capítulo 8) pode perder parte de seu saldo de criptomoedas tornando o ataque extremamente arriscado para o atacante.

3. ***Proof of Authority***: Atribui maior autoridade a certos nós, esta autoridade pode ser delegada pelo próprio responsável por uma transação ou até por outra autoridade previamente determinada. Esse esquema de delegar autoridade a certos nós é mais comum em redes centralizadas ou hierárquicas, onde as autoridades podem ser simplesmente inseridas no código do sistema. Estas autoridades podem receber um conjunto de chaves privadas especiais com as quais "assinar" os blocos válidos, isso torna o processo de validação de bloco com *proof of authority* praticamente idêntico à validação de transação.

4. ***Proof of Storage***: Também conhecida como *proof of space* ou *proof of capacity*, *proof of storage* funciona de maneira muito similar ao *proof of work*, exceto que, ao invés de tentar computar *hashes* válidas, o validador tem que dedicar espaço na memória para a solução de um problema. A quantidade de memória dedicada ao processo de validação aumenta a probabilidade do minerador obter um bloco válido. *Proof of storage* se comporta da mesma maneira que *proof of work* e apresenta as mesmas desvantagens do ponto de vista de segurança e escalabilidade. O que ocorre é a simples troca de poder computacional por espaço na memória. Essa é uma alternativa mais "verde" se comparada ao enorme desperdício de energia observado em *proof of work*, já que espaço de memória é um recurso menos crítico para outras atividades que energia elétrica.

5. ***Proof of Burn***: Este método de verificação é bastante raro. Basicamente, ele exige que, para um bloco ser verificado, uma determinada quantidade de um recurso digital (uma criptomoeda, por exemplo) tem que ser enviado a um endereço público inválido, que por sua vez destroi o recurso e o torna inacessível para sempre. Embora a motivação para *proof-of-burn* seja nobre (resolver os problemas de gasto energético em sistemas *proof of work*). Frequentemente os recursos digitais "queimados" no processo são exatamente criptomoedas que usam *proof of work*, o que dá aos sistemas *proof of burn* os exatos mesmos problemas de sistemas *proof-of-work*.

6. **Híbridos**: Sistemas híbridos são sistemas que utilizam mais de um método na validação de blocos. Um exemplo seria a criptomoeda Nano, que utiliza uma combinação de *proof of stake* para validação de blocos, mas exige do usuário a execução de uma *proof of work* para validar transações. Sistemas *proof of stake* frequentemente são híbridos para evitar certos ataques que serão discutidos posteriormente.

5 Finalidade e Consenso

> "Por que você não está usando um chapéu de festa, quando todo mundo está usando um?"
>
> *Ali Almosawi em "Um Livro Ilustrado de Maus Argumentos"*

Finalidade caracteriza a qualidade de uma informação de ser gravada na *blockchain* permanentemente. Em outras palavras, a finalidade define se uma informação registrada na *blockchain* pode ser considerada eternamente armazenada a partir do momento do seu registro(1).

5.1 Finalidade Não-Determinística

Chamamos de finalidade não-determinística quando o consenso não é atingido de maneira unânime de uma vez só mas ao longo do tempo, graças a regras previamente estabelecidas e a forma como novos blocos são adicionados ao registro. Isto ocorre geralmente em sistemas descentralizados, onde a propagação da informação se dá de maneira assíncrona, dando possibilidade à produção de bifurcações. Bifurcações em uma *blockchain* ocorrem quando dois agentes tentam atualizar a *blockchain* ao mesmo tempo com dois blocos válidos. Estes blocos deveriam ocupar a mesma posição na cadeia criando assim duas versões válidas da *blockchain*.

Tomemos como exemplo uma criptomoeda que se utilize de *proof of work*. Ela atinge consenso de maneira assíncrona e não-determinística tendo em sua rede dois mineradores que, por sua vez, competem entre si para inserir seus respectivos blocos na cadeia. O minerador A encontra um bloco com válido primeiro, e o transmite à rede. Antes da informação sobre o bloco minerado por A chegar ao minerador B, o minerador B encontra um segundo bloco válido e o transmite à rede. Neste momento ocorre uma bifurcação: se tem duas versões da cadeia terminadas em blocos diferentes.

Temos agora um impasse, temos duas versões reconhecidas e válidas da cadeia, uma terminada pelo bloco inserido por A, e a outra terminada pelo bloco inserido por B, como a rede chega ao consenso?. Por se tratar de uma rede descentralizada que, por definição, não possui autoridade central, os mineradores não têm a quem consultar para resolver o problema. Sendo assim, sabendo que há uma recompensa para quem adicionar o bloco, tanto A quanto B têm interesse que o seu respectivo bloco seja o novo bloco

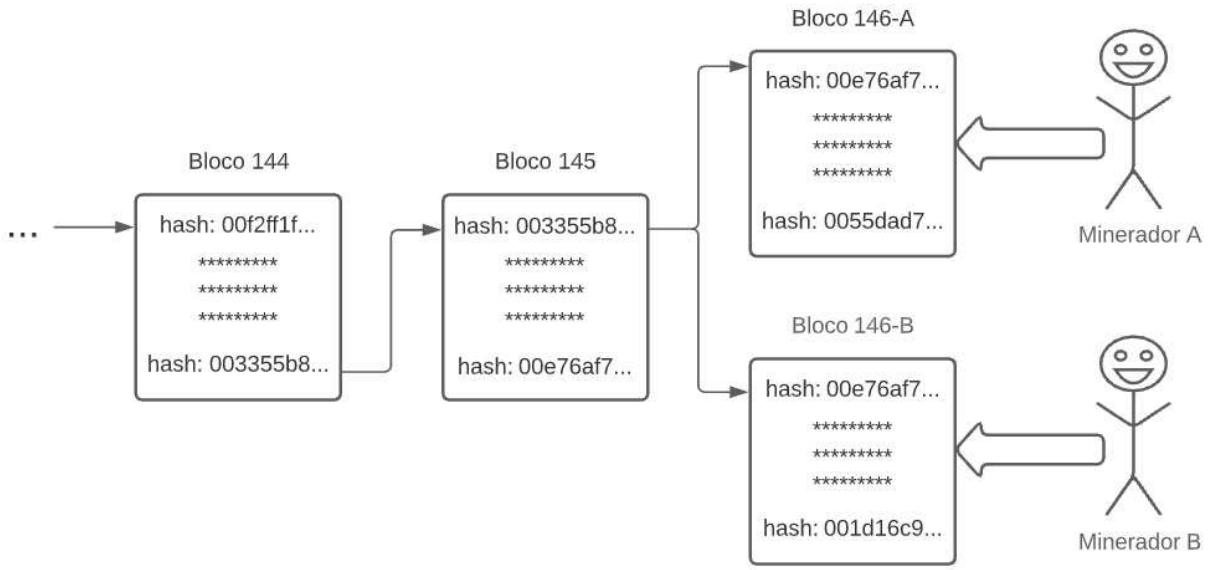

Figura 7 – Cadeia Bifurcada.

acrescentado. Com isso, ambos os mineradores continuam a trabalhar em suas respectivas versões da *blockchain*.

Quando isso ocorre, os demais mineradores escolhem uma das versões da *blockchain* na qual trabalhar. No nosso exemplo os mineradores A e B passariam a trabalhar em suas respectivas versões, uma terminada pelo bloco A e outra pelo bloco B. Eventualmente um deles, digamos B, em algum momento iria encontrar o bloco seguinte e o adicionaria na sua versão da *blockchain*. A partir daí a cadeia de B ficaria mais longa do que aquela mantida por A. Sabemos de capítulos anteriores que quanto mais longa for a cadeia, mais "permanentes" se tornam as transações nela registradas, daí temos que, em redes descentralizadas, confia-se na versão mais longa da *blockchain* e a versão de B seria adotada como a versão oficial por ser mais longa(7).

5.2 Finalidade Determinística

Neste caso, o consenso ocorre com certeza e quase que instantâneamente assim que uma nova transação (ou bloco) válido é lançado na rede. O consenso nesse tipo de sistema é geralmente atingido por meio de votação, seja por autoridades pré-estabelecidas ou por certos usuários da rede que atendem certos critérios. A decisão em si, fica por conta do princípio de Lamport ou por maioria simples(5).

Sistemas determinísticos nunca produzem bifurcações duradouras e costumam ser mais rápidos para resolver conflitos, mas podem apresentar problemas de escalabilidade e/ou são vulneráveis a certos ataques e, por isso, são mais comuns em sistemas centralizados

ou hierárquicos. Apesar dos pontos negativos, algumas aplicações atuais se utilizam de finalidade determinística. Uma delas é a rede cardano que utiliza um sistema de votação baseado em *proof of stake* para seleção do minerador e *proof of work* para autenticar blocos.

6 Criptomoedas

> "O problema principal com a moeda tradicional é toda a confiança necessária para que ela funcione. Confiamos que o Banco Central não irá desvalorizar a moeda. No entanto a história das moedas fiduciárias está cheia de quebras dessa confiança. Bancos tem que ser confiáveis para guardar nosso dinheiro e transferi-lo eletronicamente, mas eles o emprestam aos montes para bolhas de crédito e mantém quase nenhuma reserva"
>
> *Satoshi Nakamoto*

Imagine que um grupo de três indivíduos Alice, Bob e Carlos desejam enviar e receber dinheiro dos demais membros do grupo. Eles poderiam se encontrar e realizar as trocas pessoalmente ou até mesmo realizar transferências bancárias entre si mas, ao invés disso, eles decidiram que seria mais simples registrar as transações em uma lista pública e acertar o balanço da conta de cada um periodicamente. Essa lista poderia ser algo como um registro de papel, em um site ou aplicativo de mensagens qualquer. A lista poderia ter a seguinte forma:

Figura 8 – Lista Pública.

Na lista temos que, na primeira linha, Alice enviou a Bob o valor de 10 R$. Na segunda temos que Carlos enviou a Bob o valor de 15 R$ e assim por diante. Este não é um sistema muito seguro já que não impede, por exemplo, Carlos de simplesmente adicionar à

lista “Alice 500 R$ Carlos”. Alice obviamente não concordaria com essa transação portanto, para impedir fraudes como essa, é importante que todas as transações da nossa lista sejam assinadas digitalmente pelo responsável pela transação. A lista com as assinaturas ficaria então com a seguinte aparência:

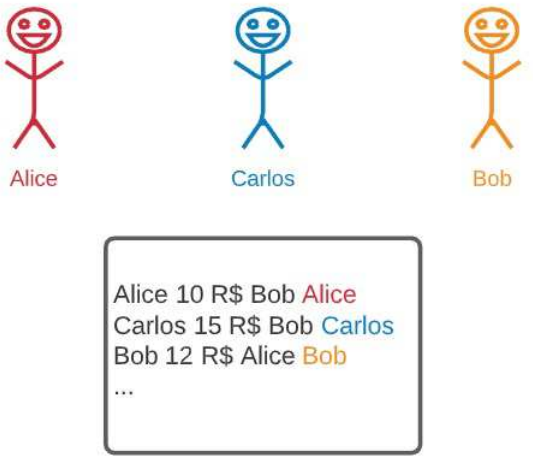

Figura 9 – Lista Assinada.

Com cada transação assinada digitalmente, é impossível que um membro do grupo se comporte como outro adicionando transações em nome de outra pessoa mas, e quanto às transações que já estão registradas?. Vamos supor que Bob queira fraudar a lista para roubar dinheiro de Alice. Como todas as transações têm que ser assinadas pelo seu responsável, não será possível para Bob fingir ser Alice mas note que a primeira transação da lista é uma transação assinada de Alice dando a Bob 10 R$. Bob poderia simplesmente pegar a transação assinada “Alice 10 R$ Bob” e repetir quantas vezes quiser na lista tirando 10R$ de Alice a cada repetição. Por isto é importante que cada transação tenha algo único que identifique a transação, um índice por exemplo, poderia servir este propósito[1]:

Se no nosso exemplo, ao invés de uma lista em um site, utilizarmos uma *blockchain* descentralizada, teremos a estrutura básica de uma criptomoeda. Alice, Bob e Carlos jamais precisariam usar dinheiro tradicional novamente caso o mundo inteiro passasse a usar essa lista como meio de transação com uma moeda própria. Ainda temos porém o problema de como o dinheiro é distribuído inicialmente. No nosso exemplo, os valores seriam acertados entre os usuários do registro no “mundo real” usando “dinheiro de verdade”. Mas, se nosso objetivo aqui é usar esse registro como meio oficial de trocas, com uma moeda única e exclusiva a esse registro, sem correspondência com uma outra moeda qualquer, então precisamos que esse “dinheiro” seja distribuído no início de alguma forma.

Existem duas formas usadas para a distribuição de moedas. Para a maioria das moedas, a validação de um bloco carrega consigo uma recompensa monetária. Neste caso,

[1] Na prática, o que ocorre é o *timestamping* de transações pelos *full nodes*

Figura 10 – Lista Indexada.

os nós que validam blocos são conhecidos como mineradores. Ao validar um bloco por *proof of work*, *proof of storage* ou qualquer outro método *resource intensive*, um minerador pode, ao final do bloco, inserir uma transação do tipo "Minerador recebe 50 moedas", ou seja, uma transação que simplesmente dá ao minerador um valor em criptomoeda. Moedas criadas dessa forma são conhecidas como mineráveis. O Bitcoin, a Ethereum e várias outras moedas utilizam este modelo e é a partir desse processo que novas unidades monetárias entram no mercado.

Outra forma muito menos comum de distribuição de moeda são feitas pelas chamadas moedas não-mineráveis. Para este caso, todas as moedas em circulação já existem no início da operação do sistema. As unidades de moeda são distribuídas aos usuários saindo de uma carteira gênesis, que contém todas as moedas. A forma de distribuição de moeda varia de caso a caso, a *Binance Coin* (BNB), moeda criada pela corretora Binance, foi distribuida por uma ICO (*Initial Coin Offering*) que é a venda de moeda para investidores interessados. A Nano foi distribuída usando o sistema *faucet*, nesse sistema pessoas interessadas entravam em um site e resolviam um *captcha* recebendo em troca uma pequena quantidade de Nano. Aproximadamente 133 milhões de Nano foram distribuídos dessa forma(8).

A adoção de uma criptomoeda vai depender de certos fatores e é fato que, nem sempre a popularidade de uma criptomoeda depende de sua tecnologia. Mesmo assim podemos afirmar que, para ter um mínimo de chance de aceitação, um aspirante a moeda global deve atender a um conjunto de pré-requisitos:

1. **Segurança**: Como em qualquer sistema financeiro, criptomoedas também devem atender certos padrões de segurança, não por motivos regulatórios ou jurídicos já que a adoção de uma criptomoeda é puramente voluntária, e sim porque a segurança funciona como um incentivo à adoção: ninguém compraria um ativo considerado

inseguro. A segurança de um criptoativo se traduz em aspectos como, a imutabilidade da *blockchain*, a transparência, a descentralização da rede e as técnicas de criptografia utilizadas. Outra característica importante que se reflete na maior segurança de moedas criptográficas é o fato de que criptomoedas geralmente, e ao contrário de moeda fiduciária, apresentam quantidade limitada de unidades monetárias. Esta característica faz das criptomoedas

2. **Escalabilidade**: É a capacidade de um sistema de crescer, uma característica essencial no sistema financeiro moderno e um grande problema para criptomoedas mais antigas. O Bitcoin e a rede Ethereum têm sérios problemas de escalabilidade, que são as causas de taxas de transação e tempo de espera elevados.

3. **Conveniência**: Quando falamos em conveniência falamos da facilidade de uso de uma criptomoeda em relação a moedas convencionais. Em geral, criptomoedas tem três desvantagens em relação a moedas convencionais: a necessidade de internet, tempo de espera para confirmar transações e a taxa de transação.

Após o sucesso do Bitcoin, várias criptomoedas alternativas foram criadas vizando a solução de problemas encontrados no Bitcoin. Por si só, o bitcoin é extremamente problemático, sua principal limitação são as altas taxas de transação sempre que a rede está congestionada, o que ocorre com frequência já que o Bitcoin só permite a realização de aproximadamente duas mil transações por bloco, o que dá 200 transações por minuto. Para fins de comparação, a rede de cartões Visa permite até 56 mil transações por segundo(9) em teoria, na prática 1,7 mil transações por segundo. É importante deixar claro que incluir uma taxa de transação é totalmente opcional na maioria das criptomoedas e o usuário pode optar por não incluí-la ou incluir uma taxa com valor mais baixo, mas para esses casos o tempo para confirmação da transação pode se tornar extremamente elevado caso a rede esteja muito congestionada.

As *altcoins* são criptomoedas alternativas que surgiram após o Bitcoin com o objetivo de sanar suas deficiências e até adicionar novas funcionalidades à *blockchain*. Inicialmente as *altcoins* eram meras bifurcações da *blockchain* do bitcoin mas operando sob regras diferentes (mais transações por bloco, menor tempo de espera, etc.), algumas criptomoedas criadas dessa forma são bastante populares (Bitcoin Cash, Litecoin, etc.).

Outras *altcoins* tentaram, além de resolver problemas de escalabilidade, adicionar novas funcionalidades à *blockchain*. A rede Ethereum é um bom exemplo disso pois cria, além de uma nova criptomoeda (o Ether), os *smart contracts* que podem representar qualquer tipo de acordo entre indivíduos ou entidades do mundo real e podem até servir como base para outras criptomoedas. Apesar de bastante versátil, a rede Ethereum acabou congestionada devido a enorme quantidade de serviços possíveis por meio dela, mesmo com um tempo de bloco de somente 15 segundos, as taxas são relativamente altas.

7 Mineração por *Proof of Work*

No capítulo anterior caracterizamos as criptomoedas e introduzimos o conceito de mineração. A mineração é o processo pelo qual alguns nós da rede de criptomoedas executam uma tarefa necessária para autenticação de um bloco e recebem em troca um valor em criptomoeda. Um dos esquemas de mineração mais populares é a *proof of work*, usado na mineração do Bitcoin, Ether e em diversas outras criptomoedas.

Uma *blockchain* que usa *proof of work* como mecanismo de resolução de conflito é basicamente um sistema que usa um problema matemático para validar blocos, esse problema geralmente é uma condição específica que a *hash* dos blocos deve atender. No caso do Bitcoin a *hash* tem que ser menor que um valor predeterminado. Na prática, mineradores de Bitcoin utilizam o número de zeros no início da *hash* como condição de validação porque isso simplifica o processo de verificação(ver figura 7 na página seguinte).

Outra particularidade do Bitcoin é que a condição da *hash* é determinada de maneira dinâmica, isso significa que a condição não é constante ao longo da vida da *blockchain*. A condição de *hash* do Bitcoin é alterada para garantir um passo de bloco de 10 minutos, ou seja, a cada 10 minutos um novo bloco é adicionado à *blockchain* do Bitcoin. O número de zeros no início da *hash* de cada bloco é determinada levando em conta o tempo que a rede levou para encontrar o bloco anterior.

Outras criptomoedas utilizam sistemas muito similares ao Bitcoin, a rede Ethereum por exemplo tem um sistema de autenticação de bloco de passo igual a 15 segundos e a rede Cardano possui um sistema de passo variável que se adapta às necessidades da rede (quanto mais congestionada, menor o período de passo). O passo de bloco é uma variável importante para as criptomoedas, um passo pequeno demais pode tornar o sistema inseguro e um passo grande demais limita a escalabilidade.

A fim de construir um sistema de mineração por *proof-of-work*, precisamos primeiramente criar uma *blockchain* básica. Para isso, vamos nos referir a biblioteca block_generator.h (a definição das funções estão na block_generator.cpp) em anexo. Essa biblioteca é utilizada para simular uma criptomoeda simples por meio da função get_block(), essa função tem duas formas. A primeira não recebe nenhum argumento e gera um bloco genesis que não apresenta *hash* em seu início. A segunda forma recebe uma *hash* em formato string e a coloca no início do bloco, essa forma será usada para produzir os demais blocos da cadeia. O conteúdo do bloco produzido pela função get_block() será um conjunto de 5 transações do tipo "[i] - Alice [X] Bob" conforme exemplificado no capítulo "Criptomoedas".

Para a mineração em si iremos nos referir a uma outra biblioteca, a miner.h (a definição das funções estão na miner.cpp) que introduz a função golden_nonce() que recebe

dois argumentos: uma mensagem no formato string e um inteiro sem sinal (*unsigned*) de 16 bits n_zeros. A função golden_nonce() deve retornar um número de 32 bits (unsigned) que, ao ser incluído no final da mensagem, retorna uma *hash* que tem exatamente n_zeros zeros no início dela. A mensagem será o nosso bloco produzido pela função block_generator, e o inteiro n_zeros será a quantidade de zeros que desejamos que a *hash* de cada bloco tenha em seu início como condição para inclusão na cadeia(nosso sistema é bastante similar ao utilizado no Bitcoin, diga-se de passagem).

Figura 11 – O número de zeros no início da *hash* indicam sua validade.

Introduzidas as bibliotecas necessárias para a contrução do nosso exemplo. Podemos iniciar a confecção do código de uma *blockchain* básica, a começar pelo primeiro bloco:

```
#include <iostream>
#include <string>

#include "sha.h"
#include "miner.h"
#include "tempo.h"
#include "block_generator.h"

int main()
{
        using namespace std;
        string block, hash;
        unsigned nonce;
        const int num_zeros = 3; //Numero de zeros no inicio da hash
        const int num_blocos = 3; //Tamanho da blockchain

        //Geracao do bloco genesis
        cout << "BLOCO GENESIS" << endl;
        block = get_block();
        nonce = golden_nonce(block, num_zeros);
        hash = SHA256(add_num32_to_message(block, nonce));
        cout << block << "NONCE: " << hex << nonce << endl
                << "HASH DE BLOCO: " << hash << endl;

        cout << "********************\n";
        return 0;
}
```

O código acima gera o primeiro bloco da nossa *blockchain* usando a função get_block() sem argumentos e em seguida calcula a *golden nonce*. A nonce é um termo utilizado para se referir a um número de 32 bits utilizado somente uma vez no curso da autenticação do bloco, a função golden_nonce() recebe como argumento o bloco e o número de zeros desejados no início da *hash* desse bloco. Começa então o processo de

concatenar ao final do bloco, a *nonce* e calcular sua *hash*. A *hash* então é testada e, caso tenha a quantidade mínima estipulada de zeros no seu início, a *hash* é considerada válida e a *nonce* é retornada e a partir daí é considerada uma *golden nonce*. A *golden nonce* ao ser concatenada no final do bloco resulta em uma *hash* de bloco válida. Caso a *nonce* não resulte em uma *hash* de bloco válida então ela é incrementada e o processo se repete, para melhor entendimento observe o diagrama a seguir.

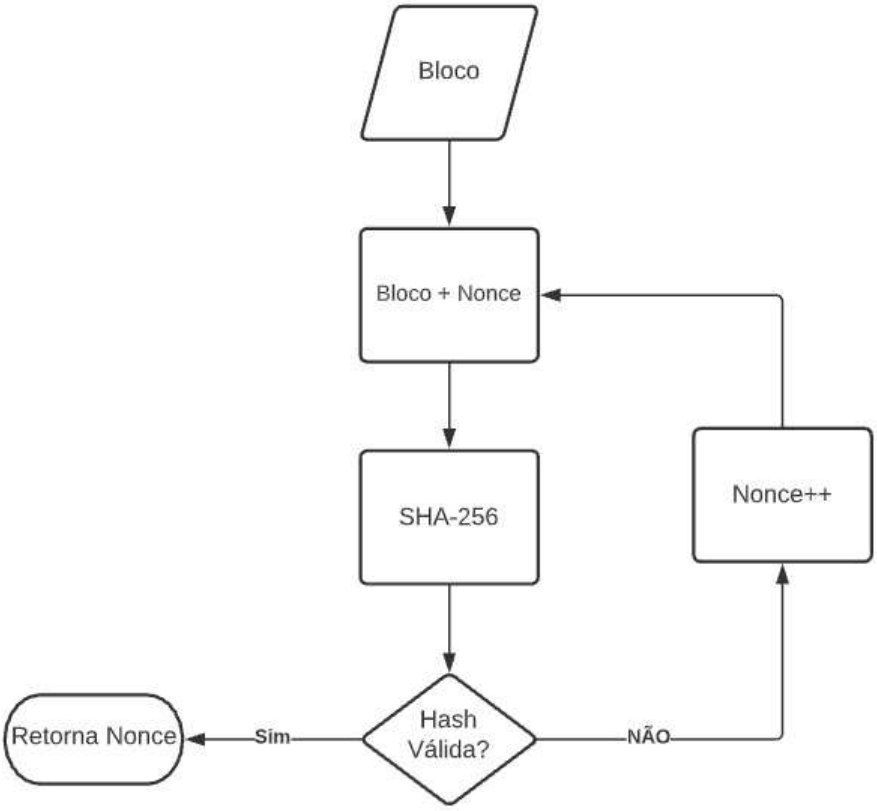

Figura 12 – Funcionamento da função golden_nonce().

O restante da *blockchain* pode ser gerado usando a função block_generator() dentro de um laço for() com a *hash* do bloco anterior colocada como argumento. Os blocos gerados dessa forma terão, além das transações que compõem o bloco, a *hash* do bloco anterior incluída logo no início do bloco conforme descrito no capitulo 2. A variável num_blocos determina o quão longa será a *blockchain* e a num_zeros determina o quão demorada será a produção de novos blocos válidos.

```
#include <iostream>
#include <string>

#include "sha.h"
#include "miner.h"
#include "tempo.h"
#include "block_generator.h"
```

```
int main()
{
        using namespace std;
        string block, hash;
        unsigned nonce;
        const int num_zeros = 4; //Numero de zeros no inicio da hash
        const int num_blocos = 3; //Tamanho da blockchain

        //Geracao do bloco genesis
        cout << "BLOCO␣GENESIS" << endl;
        block = get_block();
        nonce = golden_nonce(block, num_zeros);
        hash = SHA256(add_num32_to_message(block, nonce));
        cout << block << "NONCE:␣" << hex << nonce << endl
                << "HASH␣DE␣BLOCO:␣" << hash << endl;

        cout << "********************\n";

        //Geracao da blockchain
        for (int i = 0; i < num_blocos - 1; i++)
        {
                cout << "Bloco␣" << i + 1 << endl;
                block = get_block(hash);
                nonce = golden_nonce(block, num_zeros);
                hash = SHA256(add_num32_to_message(block, nonce));
                cout << block << "NONCE:␣" << nonce << endl
                        << "HASH␣DE␣BLOCO:␣" << hash << endl;

                cout << "********************\n";
        }

        return 0;
}
```

Sabendo que as transações e portanto as *hashes* dos blocos são aleatórias a execução desse código pode produzir a seguinte saída no prompt de comando:

```
BLOCO GENESIS
        01-Paulo 73 Bob
        02-Douglas 46 Marcos
        03-Maria 84 Paulo
        04-Aline 77 Pedro
        05-Maria 97 Douglas
NONCE: 80001000
HASH DE BLOCO: 0000ed7551d08463a0fa6fc972dbe232f98ebb15233a24de335675063f2ef8bc
********************
Bloco 1
HASH ANTERIOR: 0000ed7551d08463a0fa6fc972dbe232f98ebb15233a24de335675063f2ef8bc
        01-Aline 13 Paulo
        02-Ana 46 Douglas
        03-Ana 55 Aline
        04-Douglas 65 Maria
        05-Laura 86 Douglas
NONCE: 800012f4
HASH DE BLOCO: 00005f421582ce48caf137c9bb9dd15bc805587b4ced40c6dcedf1f4fdcf52b9
********************
Bloco 2
HASH ANTERIOR: 00005f421582ce48caf137c9bb9dd15bc805587b4ced40c6dcedf1f4fdcf52b9
        01-Ana 74 Paulo
```

```
        02-Bob 14 Paulo
        03-Paulo 41 Laura
        04-Douglas 28 Bob
        05-Marcos 25 Marcos
NONCE: a000399b
HASH DE BLOCO: 00004d5253982c81ffb93bbb987f2d2d7646ede4af33eda90061e413aad40bfc
********************
```

Perceba que, como estipulado pela variável num_zeros, todas as *hashes* de blocos tem no mínimo quatro zeros no início. O valor de num_zeros foi escolhido de maneira arbitrária no nosso exemplo. As únicas condições que num_zeros deve atender é ser um valor positivo e menor que 64 levando em conta que um valor mais alto significa uma maior dificuldade de se encontrar *hashes* válidas o que resulta em uma maior demora na execução do programa.

7.1 Desempenho e Eficiência de Mineração.

No nosso exemplo duas funções podem ser consideradas essenciais para a mineração, a função *hash* (SHA-256) e a função golden_nonce(). A importância da função *hash* fica evidente se considerarmos que o processo de mineração é feito basicamente por tentativa e erro com a função SHA sendo executada milhões de vezes até que uma *hash* de bloco válida seja produzida. Esse fato torna essa função crítica para a mineração e qualquer melhora na eficiência de execução dela fará enorme diferença na velocidade da função golden_nonce().

Outro ponto importante é que a mineração de um bloco é um problema divisível. Como explicado anteriormente a mineração consiste no calculo da *hash* de um mesmo bloco concatenado com uma *nonce* inúmeras vezes até que uma *hash* válida seja obtida, se tivermos vários dispositivos capazes de executar essa tarefa podemos dividir o nosso problema entre eles. Por exemplo, a *nonce* usuamente é um inteiro de 32 bits, se tivermos dois computadores podemos dividir o problema entre eles colocando o computador A para testar as *nonces* de 0 a $2^{32}/2$ e o computador B testa de $2^{32}/2$ a 2^{32}-1.

A função golden_nonce() da biblioteca miner.h usada no nosso exemplo tira vantagem da divisibilidade usando a funcionalidade da biblioteca <thread> do C++. As possíveis *nonces* são divididas em 8 conjuntos de aproximadamente 597 milhões de possíveis valores e cada *thread* tenta encontrar a *golden nonce* em seu respectivo conjunto. Este arranjo é feito para tirar total proveito das 8 *threads* disponíveis no processador usado para testar o exemplo (Intel i5-9300H). Outros processadores com maior número de *threads* irão suportar mais processos simultâneos e consequentemente encontrarão blocos válidos mais rapidamente caso utilizem uma versão de golden_nonce() com mais *threads*.

Atualmente, criptomoedas mais populares que usam *proof of work* são mineradas usando *pools* de mineração que tiram vantagem da divisibilidade do problema da validação

de bloco. Em um *pool*, a cada minerador individual é dado um conjunto de *nonces* para teste. Quando um dos mineradores finalmente encontra uma *golden nonce*, o bloco é minerado e o prêmio é então dividido entre todos os participantes do *pool* de acordo com sua contribuição em poder computacional. Este esquema é tão popular que hoje em dia é praticamente impossível para mineradores individuais obter lucro minerando moedas como o Bitcoin ou o Ether, toda mineração é feita por *pools*.

7.2 A Biblioteca tempo.h

Esta biblioteca pode ser encontrada em anexo para ajudar nos testes de eficiência de código. Caso seja de interesse do leitor, o tempo de execução de um código pode ser medido utilizando as funções start_chrono(), end_chrono e periodo() da seguinte forma:

```
#include <iostream>

#include ‘‘tempo.h’’

using namespace std;

int main(){
        start_chrono();

        //Codigo teste

        end_chrono();

        cout << periodo(‘m’);
        return 0;
        }
```

A função periodo() recebe como argumento os caracteres ‘u’, ‘m’ ou ‘s’ indicando a unidade de tempo que deve ser retornada (microssegundos, milisegundos ou segundos) e retorna o tempo de execução do código contido entre start_chrono() e end_chrono() na unidade de medida especificada.

8 Considerações sobre Segurança

> "Uma sala cheia de ouro e jade
> é difícil de ser guardada"
>
> *Lao Tse em "Tao Te Ching"*

Em capítulos anteriores discutimos que uma das principais razões para se usar uma *blockchain* é a segurança que ela proporciona deixando de explicar em detalhes o porquê dessa estrutura de dados ser considerada segura. Esse capítulo visa sanar essa deficiência. Devido a enorme diversidade de sistemas *blockchain*, é de se esperar que cada um deles se comporte diferentemente em relação a diferentes tipos de ataques.

Obviamente não é possível existir um sistema 100% seguro, todo sistema é suscetível a ataques de alguma forma. Quando dizemos que um sistema é seguro, estamos de fato afirmando que o sistema requer uma grande quantidade de recursos para ser atacado, muito maior do que o que poderia ser razoavelmente gasto pelo atacante. Do ponto de vista econômico, podemos dizer que o atacante gastaria muito mais no ato de atacar do que ganharia caso o ataque fosse bem sucedido.

A forma mais simples de se verificar a segurança de um sistema é definindo os possíveis ataques que esse sistema pode sofrer e analisando, caso a caso, como o sistema lidaria com o ataque em questão. A seguir iremos definir alguns ataques comuns em sistemas *blockchain* e clarificar como tais ataques seriam repelidos.

8.1 Mudanças na *Blockchain*

A característica mais notável da *blockchain* é a capacidade de registrar dados de maneira quase que permanente. Essa característica evita a forma mais óbvia de ataque: a mera mudança do registro por uma entidade maliciosa. Mas como essa estrutura de dados consegue isso?. Suponha que um minerador de uma criptomoeda PoW deseje alterar a *blockchain* por meio da insersão de um bloco fraudulento que, por exemplo, mude o valor de uma transação anteriormente realizada.

Ao realizar essa operação, o minerador visa fazer com que um gasto realizado por ele anteriormente se torne menor do que realmente é. Quando definimos a *blockchain* vimos que todo bloco é iniciado com a *hash* do bloco anterior (HA) e que essa *hash* entraria no calculo da *hash* do bloco (HB), na figura a seguir o bloco 145 da cadeia é substituido pelo bloco fabricado pelo minerador.

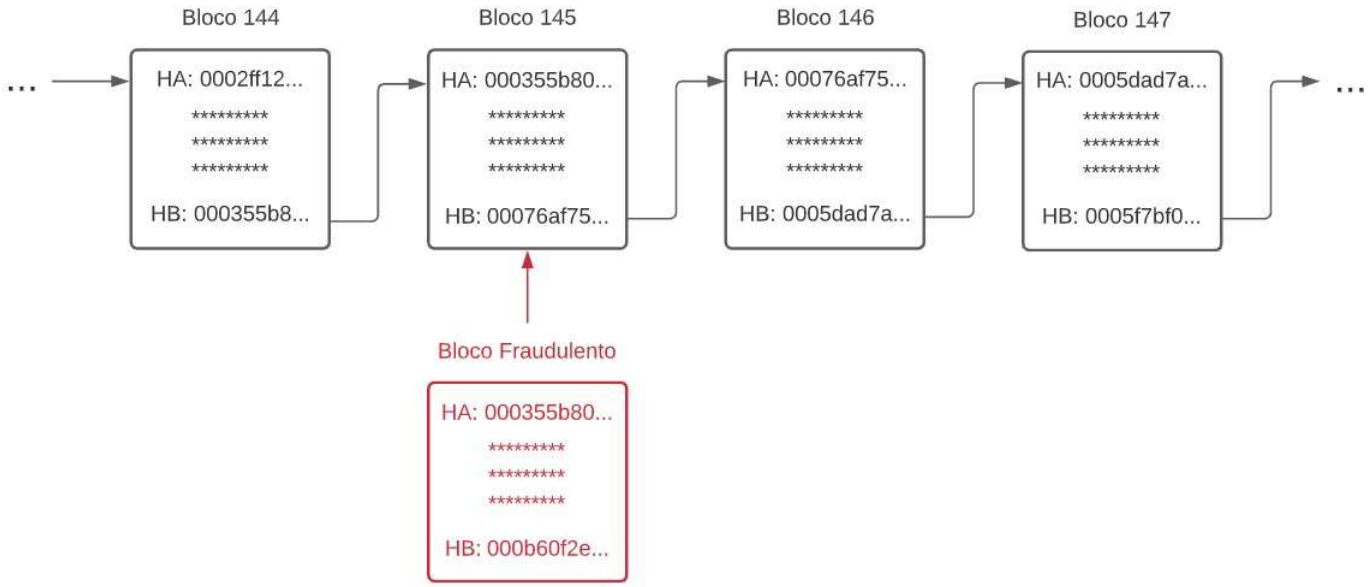

Figura 13 – Inserção ilegal de bloco.

Observe, no entanto, que o novo bloco fraudulento gera uma *hash* de bloco (HB) diferente da do bloco original. Em outras palavras, o bloco fraudulento não poderia fazer parte da cadeia porque a sua *hash* de bloco não coincide com a *hash* anterior (HA) do bloco seguinte da cadeia (o bloco 146). Isto vem da característica da função *hash* discutida no capítulo 1: qualquer mudança na entrada da função *hash* gera um número de *hash* completamente diferente. Por isso o bloco fraudulento não se encaixaria na cadeia.

Para que o bloco fraudulento tome o lugar do bloco válido, ele precisaria ser gerado com uma *hash* de bloco exatamente igual ao bloco que ele irá substituir, não basta só atender a condição de *hash*, a *hash* teria que ser idêntica. Assumindo que todas as *hashes* são igualmente prováveis, temos um espaço amostral de aproximadamente 1.158×10^{77} *hashes* possíveis, das quais apenas uma atende nosso propósito. Para se ter idéia, se todo o poder computacional dedicado a toda a rede do Bitcoin no pico histórico(170×10^{18} *hashes* por segundo) fosse dedicada somente a resolver esse problema, teríamos uma espera média de aproximadamente 2.16×10^{47} séculos até que essa *hash* fosse encontrada(10).

A segunda forma que uma alteração como essa poderia ser realizada é se o atacante, ao invés de tentar gerar um bloco falso com a mesma *hash* do verdadeiro, simplesmente gerasse o bloco falso com uma *hash* válida e, a partir dele, tentar reconstruir toda a *blockchain*. Como vimos, o nosso bloco falso tem uma HB diferente do bloco a ser substituído e não poderia ser inserido na cadeia. O atacante então, ao colocar o bloco falso na posição 145, descarta todo o restante da *blockchain* e tem que refazer a cadeia a partir do bloco 146.

Refazer a cadeia apresenta seus próprios desafios, como vimos no capítulo 5 o

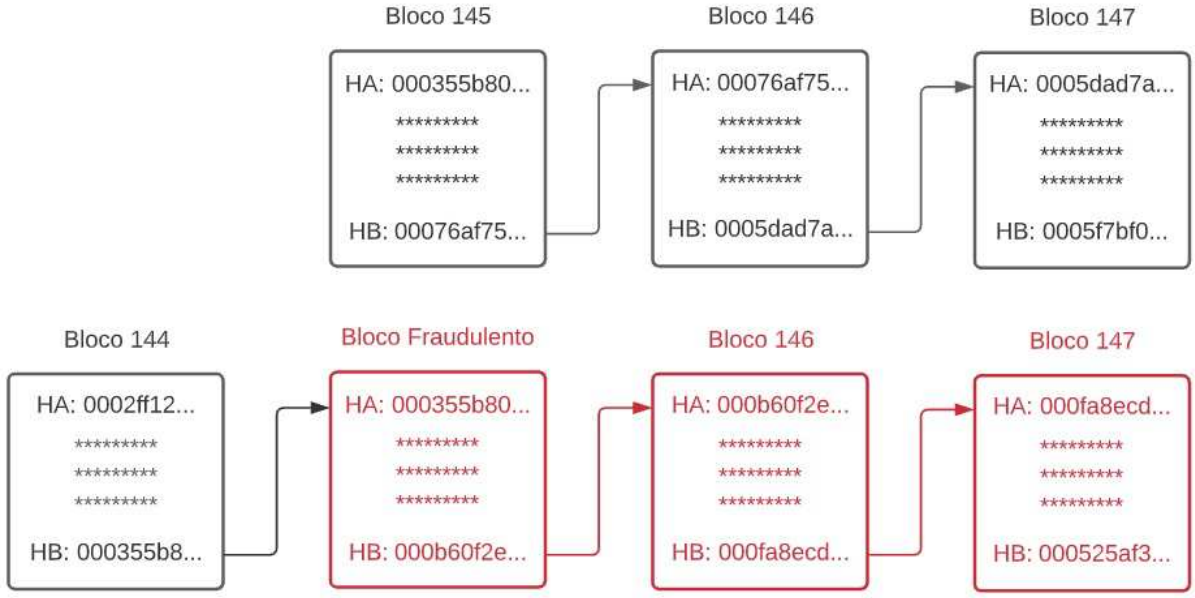

Figura 14 – A substituição de um único bloco no meio da cadeia leva a invalidação dos blocos após esse bloco e requer a remineração do restante da cadeia.

consenso em uma *blockchain* descentralizada com finalidade não-determinística é determinado pela cadeia mais longa. Tendo isso em mente, o atacante não só terá o trabalho de reminerar toda a cadeia a partir do bloco fraudulento, como também terá que fazer isso rápido o bastante para atingir e/ou ultrapassar a cadeia honesta enquanto os demais mineradores continuam a trabalhar nela. Em teoria, esse tipo de ataque só é viável caso o atacante domine mais de 50% do poder computacional da rede.

A explicação matemática para esse caso pode ser obtida usando teoria das probabilidades, a probabilidade da cadeia paralela de um atacante alcançar a cadeia original é obtida de maneira análoga a solução de um problema muito comum em probabilidades: o problema da *Ruína do jogador*(3). Primeiramente vamos definir algumas variáveis importantes:

p = probabilidade de um minerador honesto encontrar o próximo bloco
q = probabilidade do atacante encontrar o próximo bloco
q_z = probabilidade do atacante alcançar a cadeia honesta partindo de z blocos atrás

É possível demonstrar que se $p > q$, a probabilidade q_z cai exponencialmente a medida que o número de blocos que o atacante deve alcançar (z) aumenta.

$$q_z = \begin{cases} 1 & \text{se } p \leq q \\ (q/p)^z & \text{se } p > q \end{cases} \tag{8.1}$$

Assumindo que q é diretamente proporcional ao poder computacional controlado pelo atacante já podemos adiantar que sem um controle majoritário da rede de mineração um atacante tem poucas chances de sucesso.

8.2 Gasto Duplo

O ataque de gasto duplo (*double spend*) é uma variante de tentativa de mudança da *blockchain* onde o atacante tenta gastar a mesma quantidade de criptomoeda duas vezes. Vamos usar como exemplo nossa criptomoeda criada no capítulo 6. Como vimos, o balanço de uma conta nunca é registrado na *blockchain*, o que ocorre de fato é o registro de todas as transações do usuário em uma longa lista. Toda vez que o usuário envia fundos para outra conta o verificador (minerador) tem que verificar suas transações anteriores para saber se o usuário tem saldo suficiente na carteira para efetuar a operação.

Com isso já podemos assegurar que, durante a operação normal da *blockchain*, o gasto duplo é impossível (afinal, o verificador só verifica cada uma das transações passadas uma única vez) exceto em uma situação específica: durante uma bifurcação. Para entender melhor como isso ocorre, observe a figura a seguir:

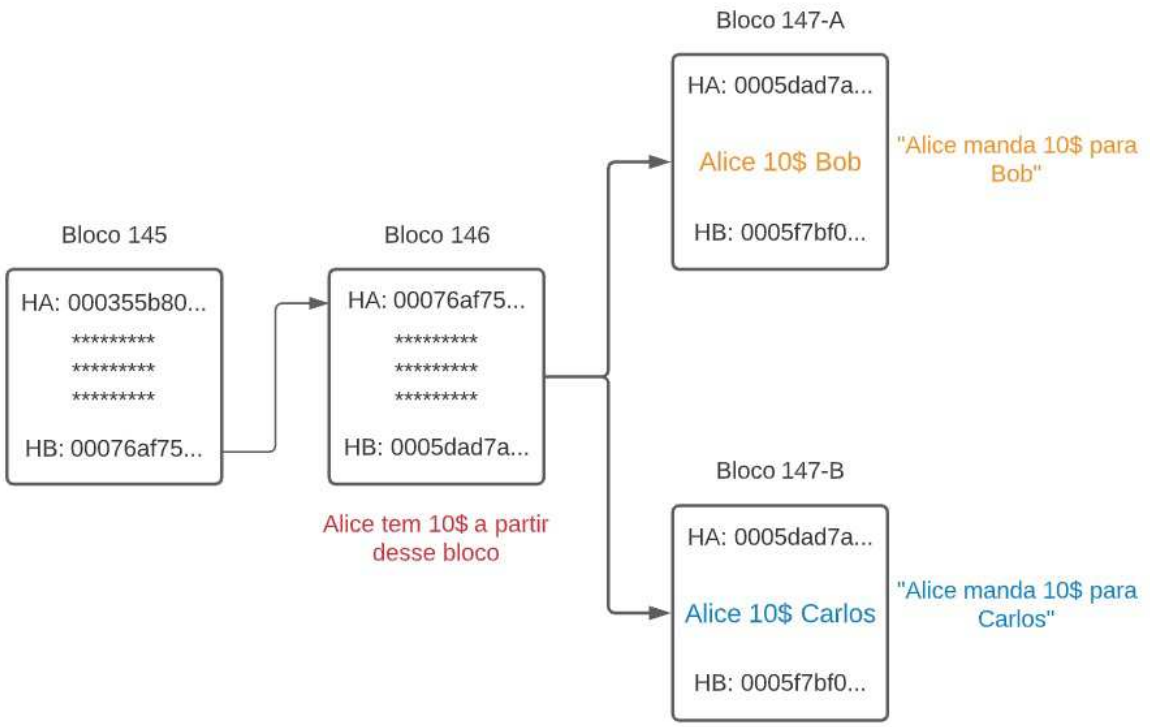

Figura 15 – O gasto duplo só é possível durante uma bifurcação.

No bloco 146, Alice possuía apenas 10$ em sua conta. Alice então mandou para Bob e para Carlos esses 10$ que ela tinha em sua conta. Em condições normais só uma dessas transações seria considerada válida e a outra seria descartada por falta de saldo, mas em

uma cadeia bifurcada as transações persistem. Isso ocorre porque um nó verificador, ao observar as transações anteriores, verifica apenas aquelas a partir do bloco 146 permitindo que Alice gaste duas vezes o valor do seu saldo.

Essa situação, no entanto, não dura muito tempo. Como vimos no capítulo 5, a bifurcação tende a ser resolvida naturalmente em sistemas não-determinísticos. Isto significa que um dos recipientes do dinheiro de Alice terá sua transação cancelada. Esse é um dos motivos pelos quais é importante se esperar a adição de novos blocos na cadeia antes de considerar a transação estabelecida.

8.3 Cancelamento de Transação

Suponhamos agora que um usuário que detêm uma quantidade significativa de poder computacional envie fundos para outra pessoa. É possível por meio da criação de uma bifurcação na *blockchain* que essa transação seja cancelada e substituída por uma de menor valor? Resposta: Sim, e quanto mais recente a transação, mais fácil será esse tipo de ataque.

Como vimos anteriormente neste capítulo, quando criamos uma *blockchain* paralela mais longa que a original a cadeia original é descartada em favor da cadeia paralela. Isto permite que um pagador possa "cancelar" seu pagamento simplesmente construindo uma *blockchain* paralela longa o bastante. Para evitar esse tipo de operação, como já deixamos claro previamente, receptores de fundos são aconselhados a esperar um certo número de blocos confirmados antes de considerar a transação permanente.

O número de blocos a ser esperado depende de vários fatores, em especial da capacidade computacional do atacante. O sucesso ou fracasso do atacante, assim como em outros tipos de ataque, é dado por uma probabilidade. Mas o quanto exatamente o recipiente de uma transação deve esperar até estar suficientemente certo de que a transação é permanente?.

Suponhamos que o recipiente espere até que z blocos sejam adicionados à frente do bloco da transação. Ele não tem como saber o quanto de progresso o atacante já obteve, mas assumindo que a cadeia honesta progrediu no tempo médio esperado, o progresso potencial do atacante será dado por uma distribuição de Poisson de valor esperado(3):

$$\lambda = z\frac{q}{p} \tag{8.2}$$

Onde q é a probabilidade do atacante avançar um bloco na cadeia e p é a probabilidade da cadeia honesta. A probabilidade do atacante alcançar a cadeia honesta será dado ao se multiplicar a densidade de Poisson para cada progresso feito pela probabilidade de ele alcançar a partir daquele ponto:

$$\sum_{k=0}^{\infty} \frac{\lambda^k e^{-\lambda}}{k!} \cdot \begin{cases} (q/p)^{z-k} & \text{se } k \leq z \\ 1 & \text{se } k > z \end{cases} \tag{8.3}$$

Rearranjando a equação, obtemos:

$$1 - \sum_{k=0}^{z} \frac{\lambda^k e^{-\lambda}}{k!}(1 - (q/p)^{z-k}) \tag{8.4}$$

Com essa equação, podemos calcular a probabilidade de um atacante reverter uma transação feita anteriormente. Como exemplo, vamos considerar que um atacante com o mesmo poder de computação que a média calculada para o maior minerador de Bitcoin do ano de 2020 (aproximadamente 17,2% da rede)(12) tente realizar esse ataque. O código Python abaixo foi adaptado do *whitepaper* de Satoshi Nakamoto e nos fornece as probabilidades neste caso:

```
import math

def attackprob(q, z):
    p = 1.0 - q
    lamb = z * (q/p)
    sum = 1.0
    for k in range(0, z+1):
        poisson = math.exp(-lamb)
        for i in range(1, k+1):
            poisson *= lamb/i
        sum -= poisson * (1 - pow(q / p, z - k))
    return sum

q = 0.172
print('q␣=␣' + str(q))
for z in range(0, 11):
    palavra = ('z␣=␣' + str(z)).ljust(10) + 'P␣=␣'\
              + str(round(attackprob(q, z) * 100, 2)).rjust(6) + '␣%'
    print(palavra)
```

O resultado obtido no *prompt* ao rodar o código é uma lista de probabilidades para cada um dos valores de z listados. Pode-se ver claramente que a chance de uma ataque bem sucedido se torna mínima (<1%) após o quinto bloco:

```
q = 0.172
z = 0      P =  100.0 %
z = 1      P =  35.63 %
z = 2      P =  15.12 %
z = 3      P =   6.63 %
z = 4      P =   2.95 %
z = 5      P =   1.32 %
z = 6      P =   0.60 %
```

```
z = 7      P =   0.27 %
z = 8      P =   0.12 %
z = 9      P =   0.06 %
z = 10     P =   0.03 %
```

8.4 Considerações Finais

Esse capítulo teve por objetivo exemplificar os ataques mais tradicionais sofridos por criptomoedas, em especial àquelas que usam PoW como método de validação de bloco. É importante deixar claro que, devida a enorme diversidade possível entre sistemas, *blockchain* uma lista completa seria impossível.

Blockchains centralizadas tem um ponto de falha óbvio que requer cuidados próprios. *Blockchains* que utilizem *proof of stake* estão sujeitos ao ataque conhecido como "*Nothing at Stake*", que é quando, ao ocorrer uma bifurcação no registro, os mineradores passam a investir nas duas versões da *ledger* para que, quando o conflito for resolvido, não haver risco de ele ficar sem uma recompensa. Isto no entanto torna resolução de conflitos mais difícil nesse tipo de sistema e desenvolvedores tem de dar atenção especial a essa possibilidade.

Outro ataque comum, especialmente em sistemas mais escaláveis ou em criptomoedas conhecidas como "*feeless*" tais como a Nano, é o clássico ataque de *spam*. Neste ataque o agente malicioso executa um grande número de transações com grande frequência durante um certo período de tempo. No caso da Nano, em março de 2021, um atacante conseguiu, com um enorme ataque de spam, tornar a rede extremamente lenta obrigando os nós "representantes" a reduzir artificialmente o número de transações por segundo fazendo com que uma criptomoeda conhecida por ser praticamente instantânea levar dias para confirmar certas transações(13).

Sistemas determinísticos, quando descentralizados, frequentemente se baseiam em algum tipo de votação para atingir consenso. Nestes casos temos o ataque *sybil* que ocorre quando o atacante finge ser mais de uma entidade votante, muitas vezes partindo de uma mesma máquina, para atribuir mais votos para si esperando obter a maioria dos votos. Esse tipo de ataque pode ser prevenido por meio de sistemas de reputação ou, mais frequentemente, com um peso de voto baseado em *proof of stake* ou *proof of work*.

Perceba que os ataques descritos geralmente não permitem a execução de uma transação inválida. Como uma transação que gera um grande número de criptomoeda por exemplo. Isto se deve ao caráter público da *ledger*. O maior assegurador que existe neste caso é a divulgação do registro para todos os nós do sistema. Isto faz com que transações fraudulentas possam ser imediatamente identificadas e ignoradas pela rede.

9 Conclusão

Blockchain, juntamente com *Internet of Things*, tornou-se um tema tradicional de convenções, palestras e jornais científicos. Sua primeira aplicação famosa, no entanto, foi informal e inicialmente pouco notada por agentes acadêmicos mais sérios. Atualmente o tema vive uma explosão de popularidade. A rápida valorização do Bitcoin e o surgimento de milhares de criptomoedas no mercado e outras aplicações com as mais variadas características fez com que o assunto entrasse nos círculos acadêmicos e corporativos.

A "febre" das *blockchains* levou a muitos avanços na tecnologia e a estudos mais sérios do potencial desse tipo de modelo, bem como na implementação (às vezes imprópria frente às alternativas) de vários sistemas *blockchain* na indústria. A fama também trouxe a tona um problema grave, especialmente observado em criptomoedas: a baixa escalabilidade desse tipo de sistema. Problemas de escalabilidade assombram praticamente todas as criptomoedas e tentativas de resolver o problema geralmente levam a uma maior centralização da rede ou a uma menor segurança para seus usuários.

Alternativas a criptomoedas baseadas unicamente em *blockchain* existem, mas são poucas e assombradas por seus próprios problemas. Criptomoedas tais como a Nano e IOTA que fazem uso de DAG (*Directed Acyclic Graph*) têm problemas relacionados a *spam*, no caso da Nano, e sérias falhas de segurança e centralização, no caso da IOTA. A verdade é que unicórios não existem. No momento, a solução para a falta de escalabilidade das criptomoedas individuais parece ser usar várias criptomoedas e, graças ao enorme interesse apresentado por desenvolvedores, não faltam alternativas no mercado.

Referências

1 TASCA, Paolo; TESSONE, Claudio J. Taxonomy of blockchain technologies. Principles of identification and classification. arXiv preprint arXiv:1708.04872, 2017. Citado 3 vezes nas páginas 13, 14 e 21.

2 MANAV, G. Blockchain for Dummies (3rd IBM Limited Edition). Hoboken. 2018. Citado na página 11.

3 NAKAMOTO, Satoshi. Bitcoin: A peer-to-peer electronic cash system. Manubot, 2019. Citado 5 vezes nas páginas 12, 17, 37, 39 e 49.

4 ALBRECHT, Chad; HAWKINS, Steven; DUFFIN, Kristopher McKay. Legitimizing Bitcoin as a Currency and Store of Value: Using Discrete Monetary Units to Consolidate Value and Drive Market Growth. Ledger, v. 5, 2020. Citado na página 14.

5 LAMPORT, Leslie; SHOSTAK, Robert; PEASE, Marshall. The Byzantine generals problem. In: Concurrency: the Works of Leslie Lamport. 2019. p. 203-226. Citado na página 22.

6 BRYSON, J. NIST: FIPS PUB 180-4: Secure Hash Standard SHS. National Institute of Standards and Technology, 2015. Citado 3 vezes nas páginas 9, 10 e 14.

7 How Bitcoin Works Under the Hood, 2021. Disponivel em: <http://www.imponderablethings.com/2013/07/how-bitcoin-works-under-hood.html>. Acesso em: 23 jan. 2021. Citado 4 vezes nas páginas 11, 12, 17 e 22.

8 LEMAHIEU, Colin. Nano: a feeless distributed cryptocurrency network (2018). Citado na página 27.

9 Bitcoin and Ethereum vs Visa and PayPal – Transactions per second. Disponível em: <https://mybroadband.co.za/news/banking/206742-bitcoin-and-ethereum-vs-visa-and-paypal-transactions-per-second.html>. Acesso em: 17 fev. 2021. Citado na página 28.

10 Taxa total de hash (TH / s). Disponivel em: <https://www.blockchain.com/charts/hash-rate>. Acesso em: 12 mar. 2021. Citado na página 36.

11 OZISIK, A. Pinar; LEVINE, Brian Neil. An explanation of Nakamoto's analysis of double-spend attacks. arXiv preprint arXiv:1701.03977, 2017. Citado na página 51.

12 Pool Distribution. Disponivel em: <https://btc.com/stats/pool?pool_mode=year>. Acesso em: 25 mar. 2021. Citado na página 40.

13 Nano's Network Flooded With Spam, Nodes Out of Sync. Disponivel em: <https://www.coindesk.com/nanos-network-flooded-spam-nodes-out-of-sync>. Acesso em: 01 abr. 2021. Citado na página 41.

Apêndices

APÊNDICE A – Demonstração Matemática da Improbabilidade de um Atacante obter a Liderança em uma *Blockchain* bifurcada

A.1 O Problema da Ruína do Jogador

De acordo com Nakamoto a probabilidade de um atacante obter a liderança em uma cadeia bifurcada é análoga a uma versão específica do problema da Ruína do Jogador(3). Suponha um jogador entra em um cassino com i dólares. O objetivo desse jogador é apostar esse dinheiro até obter uma quantidade N de dólares ou até perder todo o dinheiro. A cada aposta o jogador pode ganhar 1$ com probabilidade q ou perder 1$ com probabilidade $p = 1 - q$. Qual a probabilidade do jogador alcançar seu objetivo de obter N dólares antes de perder todo o dinheiro? O jogo termina somente em duas situações:

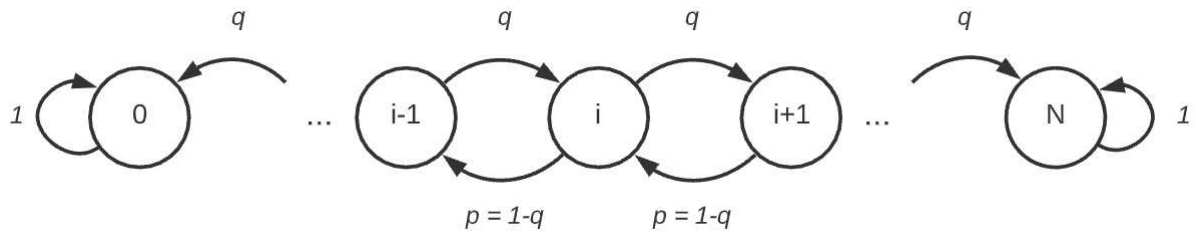

Figura 16 – Diagrama de estados do problema da Ruína do Jogador.

1. O jogador atingiu o seu objetivo de N dólares e parou de jogar, representado pelo estado N da cadeia.
2. O jogador pedeu todo o seu dinheiro e não pode mais fazer apostas, representado pelo estado 0 da cadeia

Ambos os estados (0 ou N) tem probabilidade 1, ou seja, uma vez atingidos não há mais mudança de estado, o jogo termina e esses estados são finais.

Chamamos q_i a probabilidade do jogador ganhar após jogar um certo número de vezes sabendo que ele começa a jogar com i dólares de saldo. Podemos definir q_i a partir da seguinte relação recursiva:

$$q_i = q * q_{i+1} + p * q_{i-1} \tag{A.1}$$

Sabendo que $p + q = 1$ também podemos escrever:

$$q_i = q * q_i + p * q_i \tag{A.2}$$

Substituindo essa relação na equação A.1 obtemos:

$$q_{i+1} - q_i = \frac{p}{b}(q_i - q_{i-1}) \tag{A.3}$$

Podemos chegar a uma relação geral a partir dos seguintes casos específicos:

$$q_2 - q_1 = \frac{p}{b}(q_1 - q_0) \tag{A.4}$$

$$q_3 - q_2 = \frac{p}{b}(q_2 - q_1) \tag{A.5}$$

Isolando q_2 na equação A.4 e substituindo na segunda parte da equação A.5 obtemos:

$$q_3 - q_2 = \left(\frac{p}{b}\right)^2(q_1 - q_0) \tag{A.6}$$

Mas q_0 representa a probabilidade do jogador atingir o seu objetivo (ganhar N dólares) sem ter nenhum dinheiro para apostar. Da descrição do problema podemos concluir que $q_0 = 0$ e portanto:

$$q_3 - q_2 = \left(\frac{p}{b}\right)^2 q_1 \tag{A.7}$$

Usando esse truque para os demais valores é fácil perceber que:

$$q_{i+1} - q_i = \left(\frac{p}{b}\right)^i q_1 \tag{A.8}$$

Usando a equação A.8 na relação matemática abaixo e desenvolvendo obtemos:

$$\begin{aligned} q_{i+1} - q_1 &= q_{i+1} + (-q_i + q_i) + (-q_{i-1} + q_{i-1}) + ... + (-q_3 + q_3) + (-q_2 + q_2) - q1 \\ &= (q_{i+1} - q_i) + (q_i - q_{i-1}) + ... + (q_3 - q_2) + (q_2 - q_1) \\ &= \sum_{k=1}^{i}(q_{k+1} - q_k) \\ &= q_1 \sum_{k=1}^{i}\left(\frac{p}{q}\right)^k \end{aligned} \tag{A.9}$$

Ou seja,

$$q_{i+1} = q_1 \sum_{k=0}^{i} \left(\frac{p}{q}\right)^k \tag{A.10}$$

Mas a equação A.10 é uma soma de uma série geométrica que pode ser reescrita como:

$$q_{i+1} = \begin{cases} q_1 \frac{1-(\frac{p}{q})^{i+1}}{1-(\frac{p}{q})} & , \text{se } p \neq q \\ q_1(i+1) & , \text{se } p = q = 0,5 \end{cases} \tag{A.11}$$

Agora precisamos nos livrar do termos q_1. Sabemos que quando o jogador atingir N dólares o jogo acaba porque o jogador atingiu seu objetivo. Logo, nesse estado, temos que $q_N = 1$. Reescrevendo a equação A.11 nesse estado e igualando a 1 obtemos:

$$q_N = \begin{cases} q_1 \frac{1-(\frac{p}{q})^N}{1-(\frac{p}{q})} = 1 & , \text{se } p \neq q \\ q_1(N) = 1 & , \text{se } p = q = 0,5 \end{cases} \tag{A.12}$$

Resolvendo para q_1:

$$q_1 = \begin{cases} \frac{1-(\frac{p}{q})}{1-(\frac{p}{q})^N} & , \text{se } p \neq q \\ \frac{1}{N} & , \text{se } p = q = 0,5 \end{cases} \tag{A.13}$$

Colocando o valor de q_1 obtido em A.13 de volta na equação geral A.11 teremos:

$$q_{i+1} = \begin{cases} \frac{1-(\frac{p}{q})^{i+1}}{1-(\frac{p}{q})^N} & , \text{se } p \neq q \\ \frac{i+1}{N} & , \text{se } p = q = 0,5 \end{cases} \tag{A.14}$$

E com isso podemos calcular q_i para qualquer valor inteiro positivo de i usando a equação:

$$q_i = \begin{cases} \frac{1-(\frac{p}{q})^{i}}{1-(\frac{p}{q})^N} & , \text{se } p \neq q \\ \frac{i}{N} & , \text{se } p = q = 0,5 \end{cases} \tag{A.15}$$

A.2 Solução Adaptada para o Problema das *Blockchains* Paralelas

Apesar de bela, nossa solução da Ruína do Jogador ainda não resolve o problema das *blockchains* paralelas já que não sabemos qual seria o q_0 do atacante. Em outras palavras, em que ponto atrás, na corrida com os demais mineradores, o atacante iria desistir do ataque. Por motivos teóricos assumimos que esse ponto não existe, e que o atacante está disposto a continuar seu ataque indefinidamente(11).

Para adaptar a solução, vamos assumir que o jogador perderia y dólares antes de desistir e então faremos y tender ao infinito. Essa situação seria análoga a um jogador que tem crédito infinito e pode pegar emprestado e apostar o quanto quiser. Para converter o nosso problema utilizamos uma variação do problema da Ruína do Jogador aonde o jogador começa com $i = y$ dólares e a aposta termina quando o jogador atinge 0$ (perder) ou no valor $N = y + z$ dólares (ganhar). Nessas condições ainda podemos dizer que $q_0 = 0$ e $q_N = 1$; substituindo na equação A.15 obtemos:

$$q_l = \begin{cases} \frac{1-(\frac{p}{q})^y}{1-(\frac{p}{q})^{y+z}} & \text{, se } p \neq q \\ \frac{y}{y+z} & \text{, se } p = q = 0,5 \end{cases} \tag{A.16}$$

Assumindo que o jogador esteja disposto a perder uma infinita quantidade de dinheiro para vencer jogo. Para $p < q$ temos que $(p/q)^y \to 0$ a medida que $y \to \infty$.

$$\lim_{y\to\infty} \frac{1-(\frac{p}{q})^y}{1-(\frac{p}{q})^{y+z}} = 1, \text{ quando } p < q \tag{A.17}$$

Assumindo $q > p$, calculamos o limite isolando o fator $(\frac{p}{q})^y$ no numerador e no denominador:

$$\frac{1-(\frac{p}{q})^y}{1-(\frac{p}{q})^{z+y}} = \frac{(\frac{p}{q})^y\left((\frac{p}{q})^{-y}-1\right)}{(\frac{p}{q})^y\left((\frac{p}{q})^{-y}-(\frac{p}{q})^z\right)} = \frac{(\frac{p}{q})^{-y}-1}{(\frac{p}{q})^{-y}-(\frac{p}{q})^z} \tag{A.18}$$

Quando $p > q$, $(\frac{p}{q})^{-y} = (\frac{q}{p})^y \to 0$ a medida que $y \to \infty$:

$$\lim_{y\to\infty} \frac{(\frac{p}{q})^{-y}-1}{(\frac{p}{q})^{-y}-(\frac{p}{q})^z} = \frac{-1}{-(\frac{p}{q})^z} = \left(\frac{q}{p}\right)^z \text{quando } p > q \tag{A.19}$$

Daí, quando o atacante tem recursos ilimitados, a probabilidade da *blockchain* atacante alcançar a *blockchain* real estando atrasada em z blocos é dada por:

$$Q_z = \begin{cases} 1 & \text{, se } p < q \\ (\frac{q}{p})^z & \text{, se } p > q \end{cases} \tag{A.20}$$

A.3 Tempo de Espera para Confirmação de Transação

Para saber o quanto se deve esperar antes de assumir que uma transação registrada na *blockchain* não pode ser revertida, utilizamos o modelo de Poisson. O modelo de Poisson se aplica a um experimento aleatório que envolve a probabilidade de um certo número de sucessos em um determinado intervalo de tempo. Neste modelo assumimos:

1. O número de sucessos em cada intervalo é independente do resultado nos demais intervalos

2. A probabilidade de sucesso em um pequeno intervalo de tempo é proporcional ao tamanho do intervalo

3. Para intervalos muito pequenos a chance de mais de um sucesso por intervalo é insignificante

Nossa primeira tarefa ao usar o modelo de Poisson é determinar o valor de λ, que é o número de sucessos esperados em cada intervalo(sucessos/intervalos). Para nós, sucessos serão o número de blocos que esperamos que o atacante descubra. O intervalo será o tempo que o recipiente da transação espera para adição de z blocos à cadeia honesta, ou seja, λ é dado por blocos/intervalo.

Assumimos uma rede configurada para um novo bloco a cada período de T minutos, sendo 1 bloco descoberto com 100% do poder de mineração logo, na cadeia honesta, serão descobertos p blocos a cada T minutos. Para produzir z blocos, precisamos de um intervalo de:

$$z \text{ blocos} \cdot \frac{T \text{ minutos}}{p \text{ blocos}} = \frac{zT}{p} \text{ minutos} \tag{A.21}$$

Para o atacante, q blocos serão descobertos a cada T minutos logo, durante esse intervalo, o atacante produz blocos a seguinte taxa:

$$\lambda = \Big(\frac{zT}{p} \text{ minutos/intervalo}\Big) \cdot \frac{q \text{ blocos}}{T \text{ minutos}} = \frac{zq}{p} \text{ blocos/intervalo} \tag{A.22}$$

$\lambda = zq/p$ é uma média dos blocos produzidos. Para saber quantos sucessos realmente ocorreram usamos a distribuição de Poisson. Se X for o número de sucessos esperados em um determinado experimento. A probabilidade de $X = k$ (número de blocos minerados secretamente pelo atacante) sucessos ocorrerem duranto o intervalo onde $k \geq 0$ é dada por:

$$P(X = k; \lambda) = \frac{\lambda^k e^{-\lambda}}{k!} \tag{A.23}$$

A equação acima é conhecida como a densidade de Poisson, usaremos ela para responder a seguinte pergunta: Qual a probabilidade de um atacante com poder computacional q produzir mais blocos do que os mineradores honestos naquele momento ou no futuro?. Se X for uma variável aleatória representando o número de blocos que o atacante descobre durante o período que os mineradores honestos descobrem z blocos e se $P(X; \lambda)$ é a probabilidade de um atacante produzir X blocos e se a probabilidade do atacante alcançar, partindo de uma diferença de $z - k$ é q_{z-k}. A probabilidade do atacante alcançar é dada pela soma de todas as probabilidades de X:

$$
\begin{aligned}
&= P(X = 0; \lambda)Q_z + P(X = 1)Q_{z-1} + \dots \\
&= \sum_{k=0}^{\infty} P(X = k; \lambda)Q_{z-k} \\
&= \sum_{k=0}^{\infty} \frac{\lambda^k e^{-\lambda}}{k!} Q_{z-k}
\end{aligned}
\tag{A.24}
$$

Se $k > z$, então a probabilidade do atacante alcançar a cadeia honesta é 1. Tendo isto em mente a equação fica:

$$
\sum_{k=0}^{\infty} \frac{\lambda^k e^{-\lambda}}{k!} \cdot \begin{Bmatrix} (q/p)^{z-k} & , \text{se } k \leq z \\ 1 & , \text{se } k > z \end{Bmatrix}
\tag{A.25}
$$

E por fim, já que a probabilidade de alguma coisa acontecer é 1 menos a probabilidade dessa coisa não acontecer, podemos rearranjar para o resultado anterior e obter a probabilidade do atacante minerar k blocos e não alcançar a cadeia honesta é dado por:

$$
\begin{aligned}
&= \sum_{k=0}^{\infty} \frac{\lambda^k e^{-\lambda}}{k!} \cdot \begin{Bmatrix} (q/p)^{z-k} & , \text{se } k \leq z \\ 1 & , \text{se } k > z \end{Bmatrix} \\
&= 1 - \sum_{k=0}^{\infty} \frac{\lambda^k e^{-\lambda}}{k!} \cdot \begin{Bmatrix} 1 - (q/p)^{z-k} & , \text{se } k \leq z \\ 1 - 1 & , \text{se } k > z \end{Bmatrix} \\
&= 1 - \sum_{k=0}^{z} \frac{\lambda^k e^{-\lambda}}{k!} \cdot \left(1 - \left(\frac{q}{p}\right)^{z-k}\right) + \sum_{k=z+1}^{\infty} \frac{\lambda^k e^{-\lambda}}{k!} \cdot (0) \\
&= 1 - \sum_{k=0}^{z} \frac{\lambda^k e^{-\lambda}}{k!} \cdot \left(1 - \left(\frac{q}{p}\right)^{z-k}\right)
\end{aligned}
\tag{A.26}
$$

De maneira mais direta, podemos dizer que a probabilidade de um atacante reverter uma transação após o recipiente da transação esperar que z blocos tenham sido adicionados a frente do bloco da transação é igual a:

$$
= 1 - \sum_{k=0}^{z} \frac{\lambda^k e^{-\lambda}}{k!} \cdot \left(1 - \left(\frac{q}{p}\right)^{z-k}\right)
\tag{A.27}
$$

A.4 Por que Simplesmente Alcançar e não Ultrapassar?

Alguns autores argumentam que, nos exemplos anteriormente apresentados, sendo o objetivo final do atacante anular a cadeia honesta, não faria mais sentido ele buscar ultrapassar a cadeia honesta ao invés de meramente alcançá-la?. Neste caso, a probabilidade

de um atacante substituir a cadeia original produzindo uma blockchain paralela partindo de z blocos atrás deveria ser dada por:

$$Q_{z+1} = \begin{cases} 1 & \text{, se } p < q \\ (\frac{q}{p})^{z+1} & \text{, se } p > q \end{cases} \tag{A.28}$$

E a probabilidade do atacante reverter uma transação após o recipiente da transação esperar z blocos é dada por:

$$= 1 - \sum_{k=0}^{z+1} \frac{\lambda^k e^{-\lambda}}{k!} \cdot \left(1 - \left(\frac{q}{p}\right)^{z+1-k}\right) \tag{A.29}$$

Ambas as equações dão a probabilidade do atacante ultrapassar a rede em ambos os exemplos. É importante deixar claro, no entanto, que a partir do momento em que o atacante atinge a cadeia honesta, ocorre uma ambiguidade na rede no sentido que vai existir a possibilidade de outros mineradores assumirem a cadeia paralela como original e passarem a trabalhar nela. Caso isso ocorra, ambas essas equações não farão mais sentido porque o poder computacional do atacante iria se somar ao dos mineradores enganados, ou seja, haveria uma mudança dos valores de p e q e, consequentemente, uma mudança no modelo obtido para as probabilidades.

Anexos

ANEXO A – sha.h

```
#pragma once
#include <iostream>
#include <string>

typedef unsigned int int32;
typedef const unsigned int constint32;
typedef unsigned long long int int64;
typedef unsigned char int8;

/*Funcoes matematicas
************************************************/

//Deslocar x para a direita n posicoes bitwise
int32 SHR(int32 x, int32 n);

//Rotacao para a direita n posicoes bitwise
int32 ROTR(int32 x, int32 n);

//Ch faz com que x escolha entre y e z o bit que sera repassado ao resultado
//em cada casa binaria
int32 Ch(int32 x, int32 y, int32 z);

//Maj resulta no maior bit para cada casa binaria entre as mensagens x, y, z
int32 Maj(int32 x, int32 y, int32 z);

int32 soma0(int32 x);
int32 soma1(int32 x);
int32 sigma0(int32 x);
int32 sigma1(int32 x);

/*Fim das funcoes matematicas
************************************************/

//Adiciona no final da mensagem n caracteres do tipo 0b00000000;
std::string addzeros(std::string message, int32 n);

//Calcula a hash de uma mensagem
std::string SHA256(std::string message);

//Converte um valor int32 em uma string representando o mesmo numero em hex
std::string conv_int32_to_hexstring(int32 H);

//Converte um valor int de 4 bits em um char representando seu valor hex
char conv_int4_to_hexchar(int32 num);

//insere num numero de 64 bits no final da mensagem
std::string add_num64_to_message(std::string message, int64 num);
```

ANEXO B – sha.cpp

```
#include "sha.h"
#include <iostream>
#include <string>
#include <iomanip>

using namespace std;

constint32 K[64] = {0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5,
                    0x3956c25b, 0x59f111f1, 0x923f82a4,0xab1c5ed5,
                    0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3,
                    0x72be5d74, 0x80deb1fe, 0x9bdc06a7,0xc19bf174,
                    0xe49b69c1, 0xefbe4786, 0x0fc19dc6, 0x240ca1cc,
                    0x2de92c6f, 0x4a7484aa, 0x5cb0a9dc, 0x76f988da,
                    0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7,
                    0xc6e00bf3, 0xd5a79147, 0x06ca6351, 0x14292967,
                    0x27b70a85, 0x2e1b2138, 0x4d2c6dfc, 0x53380d13,
                    0x650a7354, 0x766a0abb, 0x81c2c92e, 0x92722c85,
                    0xa2bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3,
                    0xd192e819, 0xd6990624, 0xf40e3585, 0x106aa070,
                    0x19a4c116, 0x1e376c08, 0x2748774c, 0x34b0bcb5,
                    0x391c0cb3, 0x4ed8aa4a, 0x5b9cca4f, 0x682e6ff3,
                    0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208,
                    0x90befffa, 0xa4506ceb, 0xbef9a3f7, 0xc67178f2};

/*Funcoes matematicas
***********************************************/
//Deslocar x para a direita n posicoes bitwise
int32 SHR(int32 x, int32 n) {
        return x >> n;
}

//Rotacao para a direita n posicoes bitwise
int32 ROTR(int32 x, int32 n) {
        return (x >> n) | (x << (32 - n));
}

//Ch faz com que x escolha entre y e z o bit que sera repassado ao resultado
//em cada casa binaria
int32 Ch(int32 x, int32 y, int32 z) {
        return (x & y) ^ (~x & z);
}

//Maj resulta no maior bit para cada casa binaria entre as mensagens x, y, z
int32 Maj(int32 x, int32 y, int32 z) {
        return (x & y) ^ (x & z) ^ (y & z);
}


int32 soma0(int32 x) {
        return ROTR(x, 2) ^ ROTR(x, 13) ^ ROTR(x, 22);
}

int32 soma1(int32 x) {
```

```
        return ROTR(x, 6) ^ ROTR(x, 11) ^ ROTR(x, 25);
}

int32 sigma0(int32 x) {
        return ROTR(x, 7) ^ ROTR(x, 18) ^ SHR(x, 3);
}

int32 sigma1(int32 x) {
        return ROTR(x, 17) ^ ROTR(x, 19) ^ SHR(x, 10);
}

/*Fim das funcoes matematicas
************************************************/

//Adiciona no final da mensagem n caracteres do tipo 0b00000000;
string addzeros(string message, int32 n) {
        for (int32 i = 0; i < n; i++) message = message + char(0b00000000);
        return message;
}

//Retorna a mensagem tratada de forma a ter o tamanho de bits necessario
//para execucao da funcao sha
string padding(string message) {
        //Se a mensagem for um multiplo de 64 bytes nenhum padding e realizado.
        if (message.size() % 64 == 0) return message;
        int32 pdspace = 64 - message.size() % 64;

        /*Caso o bloco final da mensagem nao possua a quantidade minima de 9 bytes necessarias para o
        padding entao um novo bloco devera ser adicionado a mensagem, o teste a seguir realiza isso*/
        if (pdspace < 9) pdspace += 64;

        int64 l = int64(message.size()) * 8;//tamanho em bits da mensagem


        message = message + char(0b10000000);
        pdspace--;
        message = addzeros(message, pdspace - 8);//sobram 8 espacos(64 bits) no fim para adicao do l
        message = add_num64_to_message(message, l);
        return message;
}

//Calcula a hash de uma mensagem
string SHA256(string message) {
        int32 H[8] = { 0x6a09e667, 0xbb67ae85, 0x3c6ef372, 0xa54ff53a,
                                0x510e527f, 0x9b05688c, 0x1f83d9ab, 0x5be0cd19 };

        int32 a, b, c, d, e, f, g, h;

        int32 T1, T2;

        int32 W[64] = {};//palavras
        int32 aux = 0;//contador de caracteres da message


        /*PRE PROCESSAMENTO*/
        //padding
        message = padding(message);

        constint32 N = message.size() / 64;//numero de blocos
```

```
/*FIM DO PREPROCESSAMENTO*/

for (size_t n = 0; n < N; n++)
{
        for (size_t i = 0; i < 64; i++) W[i] = 0;//reset array W
        for (int i = 0; i < 16; i++)
        {
                W[i] += (int32(message[aux]) & 0b00000000000000000000000011111111) << 24;
                aux++;
                W[i] += (int32(message[aux]) & 0b00000000000000000000000011111111) << 16;
                aux++;
                W[i] += (int32(message[aux]) & 0b00000000000000000000000011111111) << 8;
                aux++;
                W[i] += (int32(message[aux]) & 0b00000000000000000000000011111111);
                aux++;
        }

        for (size_t i = 16; i < 64; i++)
        {
                W[i] = sigma1(W[i - 2]) + W[i - 7] + sigma0(W[i - 15]) + W[i - 16];
        }

        a = H[0];
        b = H[1];
        c = H[2];
        d = H[3];
        e = H[4];
        f = H[5];
        g = H[6];
        h = H[7];

        for (size_t t = 0; t < 64; t++)
        {
                T1 = h + soma1(e) + Ch(e, f, g) + K[t] + W[t];
                T2 = soma0(a) + Maj(a, b, c);
                h = g;
                g = f;
                f = e;
                e = d + T1;
                d = c;
                c = b;
                b = a;
                a = T1 + T2;
        }

        H[0] = a + H[0];
        H[1] = b + H[1];
        H[2] = c + H[2];
        H[3] = d + H[3];
        H[4] = e + H[4];
        H[5] = f + H[5];
        H[6] = g + H[6];
        H[7] = h + H[7];

}

string HASH;
for (size_t i = 0; i < 8; i++)
{
        HASH = HASH + conv_int32_to_hexstring(H[i]);
```

```
        }


        return HASH;
}

//Converte um valor int32 em uma string representando o mesmo numero em hex
std::string conv_int32_to_hexstring(int32 H) {
        string hexa = "01234567";
        for (int i = 7; i >= 0; i--)
        {
                hexa[i] = conv_int4_to_hexchar(H & 0b00000000000000000000000000001111);
                H = H >> 4;
        }
        return hexa;
}

//Converte um valor int de 4 bits em um char representando seu valor hex
char conv_int4_to_hexchar(int32 num) {
        if (num > 15) {
                cout << "Erro na Conversao de Inteiro" << endl;
                return '\0';
        }
        switch (num)
        {
        case 0:
                return '0';
                break;
        case 1:
                return '1';
                break;
        case 2:
                return '2';
                break;
        case 3:
                return '3';
                break;
        case 4:
                return '4';
                break;
        case 5:
                return '5';
                break;
        case 6:
                return '6';
                break;
        case 7:
                return '7';
                break;
        case 8:
                return '8';
                break;
        case 9:
                return '9';
                break;
        case 10:
                return 'a';
                break;
        case 11:
                return 'b';
```

```
                break;
        case 12:
                return 'c';
                break;
        case 13:
                return 'd';
                break;
        case 14:
                return 'e';
                break;
        case 15:
                return 'f';
                break;
        default:
                return '\\0';
                break;
        }

}

//insere num numero de 64 bits no final da mensagem
std::string add_num64_to_message(std::string message, int64 num) {
        string n;
        int64 aux[8];

        aux[0] = 0b0000000000000000000000000000000000000000000000000000000011111111 & num;
        aux[1] = 0b0000000000000000000000000000000000000000000000001111111100000000 & num;
        aux[2] = 0b0000000000000000000000000000000000000000111111110000000000000000 & num;
        aux[3] = 0b0000000000000000000000000000000011111111000000000000000000000000 & num;
        aux[4] = 0b0000000000000000000000001111111100000000000000000000000000000000 & num;
        aux[5] = 0b0000000000000000111111110000000000000000000000000000000000000000 & num;
        aux[6] = 0b0000000011111111000000000000000000000000000000000000000000000000 & num;
        aux[7] = 0b1111111100000000000000000000000000000000000000000000000000000000 & num;

        aux[1] = aux[1] >> 8;
        aux[2] = aux[2] >> 16;
        aux[3] = aux[3] >> 24;
        aux[4] = aux[4] >> 32;
        aux[5] = aux[5] >> 40;
        aux[6] = aux[6] >> 48;
        aux[7] = aux[7] >> 56;

        string sufixo = { char(aux[7]), char(aux[6]), char(aux[5]), char(aux[4]),
                        char(aux[3]), char(aux[2]), char(aux[1]), char(aux[0]) };


        return message + sufixo;
}
```

ANEXO C – miner.h

```
#pragma once
#include <iostream>
#include <string>

typedef unsigned int int32;
typedef unsigned short int int16;
typedef unsigned long long int int64;

//gera o numero a ser adicionado ao final da mensagem para garantir que
//sua hash tenha n zeros no inicio
int32 golden_nonce(std::string block, int16 n_zeros);

//inclui os bits de num no final da mensagem;
std::string add_num32_to_message(std::string message, int32 num);

//verifica se a hash tem n zeros no inicio
bool verify_hash(std::string hash, int16 n_zeros);

//busca um golden nonce entre n1 e n2
void testa_nonce(std::string message, int16 num_zeros, unsigned n1, unsigned n2, bool* flag,
 int32* gnonce);
```

ANEXO D – miner.cpp

```
#include <iostream>
#include <string>
#include <thread>
#include <mutex>

#include "miner.h"
#include "sha.h"

static std::mutex token_miner;

int32 golden_nonce(std::string block, int16 n_zeros)
{
        using namespace std;
        if (n_zeros > 64) {
                cout << "Erro!␣Numero␣de␣zeros␣excede␣tamanho␣da␣hash" << endl;
                return 0;
        }

        int32 gnonce = 0;
        bool flag = false;
        const int32 N = 536870912;
        thread Processos[8];
        for (size_t i = 0; i < 8; i++)
        {
        Processos[i] = thread(testa_nonce, block, n_zeros, i * N, (i+1) * N - 1, &flag, &gnonce);
        }
        for (size_t i = 0; i < 8; i++)
        {
                Processos[i].join();
                //cout << "Thread␣" << i << "␣finalizada" << endl;
        }
        if (!flag)
        {
                cout << "Numero␣magico␣nao␣encontrado" << endl;
                return 0;
        }
        return gnonce;
}

std::string add_num32_to_message(std::string message, int32 num)
{
        using namespace std;
        string n;
        int32 aux[4];
        aux[0] = 0b00000000000000000000000011111111 & num;
        aux[1] = 0b00000000000000001111111100000000 & num;
        aux[2] = 0b00000000111111110000000000000000 & num;
        aux[3] = 0b11111111000000000000000000000000 & num;

        aux[1] = aux[1] >> 8;
        aux[2] = aux[2] >> 16;
        aux[3] = aux[3] >> 24;
```

```
        string sufixo = { char(aux[3]), char(aux[2]), char(aux[1]), char(aux[0]) };


        return message + sufixo;

}

bool verify_hash(std::string hash, int16 n_zeros) {
        bool answer = true;
        for (size_t i = 0; i < n_zeros; i++)
        {
                if (hash[i] != '0') {
                        answer = false;
                        break;
                }
        }
        return answer;
}

void testa_nonce(std::string message, int16 num_zeros, unsigned n1, unsigned n2,
bool *flag, int32 *gnonce)
{
        using namespace std;
        string hash = SHA256("0");
        bool t_flag;
        for (unsigned nonce = n1; nonce < n2; nonce++)
        {
                token_miner.lock();
                t_flag = *flag;
                token_miner.unlock();
                if (t_flag) break;
                //cout << nonce << endl;
                if (verify_hash(SHA256(add_num32_to_message(message, nonce)), num_zeros))
                {
                        token_miner.lock();
                        *gnonce = nonce;
                        *flag = true;
                        token_miner.unlock();
                        break;
                }
        }
}
```

ANEXO E – block_generator.h

```
#pragma once
#include <string>

std::string get_block(void);//Retorna um bloco genesis

std::string get_block(std::string prev_hash);//Retorna um bloco
```

ANEXO F – block_generator.cpp

```
#include <iostream>
#include <string>
#include <random>

#include "block_generator.h"

using namespace std;

string num2string(unsigned num);

string usuario[10] = { "Aline", "Bob", "Pedro", "Carlos", "Laura",
                                        "Ana", "Maria", "Marcos", "Paulo", "Douglas" };


string get_block(void) {
        random_device rd;
        mt19937 mtrand{ rd() };
        string transaction[5];

        for (int i = 0; i < 5; i++)
        {
                transaction[i] = "\t" +num2string(i + 1) +"-"+ usuario[mtrand() % 10] + "␣" +
                 num2string(mtrand() % 98 + 1) + "␣" + usuario[mtrand() % 10] + "\n";
        }
        string block;
        for (int i = 0; i < 5; i++)
        {
                block = block + transaction[i];
        }

        return block;
}

std::string get_block(std::string prev_hash) {
        string transaction[5];
        random_device rd;
        mt19937 mtrand{ rd() };

        for (int i = 0; i < 5; i++)
        {
                transaction[i] = "\t" + num2string(i + 1) + "-" + usuario[mtrand() % 10] + "␣"
                 + num2string(mtrand() % 98 + 1) + "␣" + usuario[mtrand() % 10] + "\n";
        }
        string block;
        block = "HASH␣ANTERIOR:␣" + prev_hash + "\n";
        for (int i = 0; i < 5; i++)
        {
                block = block + transaction[i];
        }

        return block;
}

char digit2char(unsigned num) {
```

```
        switch (num)
        {
        case 0:
                return '0';
        case 1:
                return '1';
        case 2:
                return '2';
        case 3:
                return '3';
        case 4:
                return '4';
        case 5:
                return '5';
        case 6:
                return '6';
        case 7:
                return '7';
        case 8:
                return '8';
        case 9:
                return '9';
        default:
                return 'x';
        }
}

string num2string(unsigned num) {
        string word;
        unsigned aux1, aux2;
        aux1 = num % 10;
        num = num / 10;
        aux2 = num % 10;
        word = word + digit2char(aux2) + digit2char(aux1);

        return word;
}
```

ANEXO G – tempo.h

```
#pragma once
/*Define funcoes que ajudam na medicao do tempo e tempo de execucao de programas*/

void start_chrono(void);
void end_chrono(void);
unsigned long long periodo(char);
```

ANEXO H – tempo.cpp

```
#include <iostream>
#include <chrono>
#include <string>

#include "tempo.h"

using namespace std;
using namespace chrono;

static high_resolution_clock::time_point start_time;
static high_resolution_clock::time_point end_time;


void start_chrono(void)
{
        start_time = high_resolution_clock::now();
        return;
}

void end_chrono(void)
{
        end_time = high_resolution_clock::now();
        return;
}

unsigned long long periodo(char unidade)
{
        switch (unidade)
        {
        case 'u':
                return duration_cast<microseconds>(end_time - start_time).count();
                break;
        case 'm':
                return duration_cast<milliseconds>(end_time - start_time).count();
                break;
        case 's':
                return duration_cast<seconds>(end_time - start_time).count();
                break;
        default:
                return 0;
                break;
        }
}
```