UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIA DA COMPUTAÇÃO – BACHARELADO

D-EMAIL: CORREIO ELETRÔNICO BASEADO EM BLOCKCHAIN

RUAN SCHUARTZ RUSSI

BLUMENAU
2021

RUAN SCHUARTZ RUSSI

D-EMAIL: CORREIO ELETRÔNICO BASEADO EM BLOCKCHAIN

Trabalho de Conclusão de Curso apresentado ao curso de graduação em Ciência da Computação do Centro de Ciências Exatas e Naturais da Universidade Regional de Blumenau como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação.

Prof. Francisco Adell Péricas, Ms. - Orientador

BLUMENAU
2021

D-EMAIL: CORREIO ELETRÔNICO BASEADO EM BLOCKCHAIN

Por

RUAN SCHUARTZ RUSSI

Trabalho de Conclusão de Curso aprovado para obtenção dos créditos na disciplina de Trabalho de Conclusão de Curso II pela banca examinadora formada por:

Presidente:	Prof(a). Francisco Adell Péricas – Orientador(a), FURB
Membro:	Prof(a). Luciana Pereira de Araújo Kohler – FURB
Membro:	Prof(a). Everaldo Artur Grahl – FURB

Dedico este trabalho à minha família, amigos e professores que me apoiaram em minha jornada até este momento.

AGRADECIMENTOS

Aos meus pais, Gislane Schuartz e José Maurício Russi, que sempre me apoiaram em todas as fases da minha vida.

Ao meu orientador Francisco Adell Péricas pela dedicação na orientação deste trabalho.

Não sabendo que era impossível, foi lá e fez.

Jean Cocteau

RESUMO

Este trabalho apresenta a implementação de um sistema de correio eletrônico baseado em *blockchain*. O sistema foi dividido em três componentes, sendo eles uma *blockchain,* uma camada de armazenamento externo e um aplicativo cliente. Na *blockchain* são armazenados os dados essenciais para o funcionamento do sistema. Esses dados incluem os usuários cadastrados, os endereços de e-mail e as informações básicas dos e-mails. O desenvolvimento da *blockchain* foi feito com base no *framework* Cosmos-SDK. A camada de armazenamento externo foi implementada com o objetivo de diminuir o tamanho da *blockchain*. Ela armazena os textos enviados nas mensagens de e-mail. A tecnologia utilizada para fazer o armazenamento externo dos dados foi o IPFS. Já o aplicativo cliente foi desenvolvido com base no *framework* Flutter e tem como objetivo facilitar o uso do sistema. Os resultados obtidos a partir dos testes e experimentos executados demonstram que é viável implementar soluções de correio eletrônico utilizando *blockchain*. Foi observado que o uso de *blockchain* consegue entregar maior privacidade, segurança e descentralização para os usuários finais. Apesar disso, foram observados alguns problemas. Em um comparativo com os atuais sistemas de correio eletrônico, foi identificado que a blockchain precisaria de uma capacidade de armazenamento de 5541 TB para armazenar todos os e-mails enviados mundialmente em um período de 10 dias. Além do alto custo com armazenamento, foram identificados gargalos com relação ao desempenho da *blockchain*. Com a popularização do sistema e o aumento do número de usuários, estima-se que a *blockchain* não deve conseguir processar mais do que 4000 e-mails por segundo.

Palavras-chave: Blockchain. E-mail. IPFS. Cosmos-SDK.

ABSTRACT

This paper presents the implementation of a blockchain-based email system. The system was divided into three components, being a blockchain, an external storage layer, and a client application. In the blockchain, the essential data for the operation of the system is stored. This data includes registered users, email addresses, and basic email information. The development of the blockchain was based on the Cosmos-SDK framework. The external storage layer was implemented in order to reduce the size of the blockchain. It stores the texts sent in the email messages. The technology used to store the external data was IPFS. The client application was developed based on the Flutter framework, and aims to facilitate the use of the system. The results obtained from the tests and experiments performed demonstrate that it is feasible to implement email solutions using blockchain. It was observed that the use of blockchain is able to deliver greater privacy, security, and decentralization to end users. Nevertheless, some problems have been observed. In a comparison with current email systems, it was identified that blockchain would need a storage capacity of 5541 TB to store all emails sent worldwide in a 10-day period. In addition to the high cost of storage, bottlenecks were identified with regard to the performance of the blockchain. As the system becomes more popular and the number of users increases, it is estimated that the blockchain should not be able to process more than 4000 emails per second.

Key-words: Blockchain. Email. IPFS. Cosmos-SDK.

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

ARPANET – Advanced Research Projects Agency Network

DHT – Distributed Hash Table

HTTP – Hypertext Transfer Protocol

IBC – Interblockchain Communication Protocol

IPFS – InterPlanetary File System

P2P – Peer-to-Peer

SMTP – Simple Mail Transfer Protocol

URL – Uniform Resource Locator

SUMÁRIO

1 INTRODUÇÃO

O correio eletrônico é uma das tecnologias mais populares no meio digital. Segundo Radicati (2020), o número total de usuários ativos de e-mail é superior a 4 bilhões, sendo estimado que esse valor aumente para 4,4 bilhões até o fim de 2024. No mundo, mais de 306 bilhões de e-mails são enviados diariamente. Essa popularidade pode ser atribuída a diversos fatores, sendo um deles a maturidade da tecnologia. O primeiro e-mail da história foi enviado no ano de 1971 (BLOOM, 2017).

Na época em que os protocolos de correio eletrônico foram desenhados, a computação ainda estava nos primórdios da Advanced Research Projects Agency Network (ARPANET), progenitor da internet. O acesso a ARPANET era restrito a militares e pesquisadores (TRAININI; CARISSIMI, 2005). Neste ambiente controlado, a preocupação com a segurança da informação era mínima. O cenário mudou com o advento e popularização da internet. Neste novo cenário, os protocolos para correio eletrônico apresentaram diversas vulnerabilidades, algumas inerentes ao modo como os protocolos foram desenhados. Exemplos de vulnerabilidades são envio de *spam*, vazamentos de mensagens e falsificação de identidade (TRAININI; CARISSIMI, 2005).

A maioria dos sistemas de correio eletrônico utilizam o Simple Mail Transfer Protocol (SMTP) para o envio das mensagens (RIABOV, 2005). O SMTP é um protocolo aberto. Isso permite com que sejam criadas implementações independentes de plataforma e evita que exista alguma entidade com controle sobre a tecnologia. Apesar disso, atualmente existe uma tendência na centralização dos usuários em serviços de e-mail de um pequeno grupo de empresas. Um exemplo é o serviço de e-mail da Google, o Gmail, que já soma mais de 1,8 bilhões de usuários (PETROV, 2021). O problema dessa centralização é que ela exige que os usuários tenham plena confiança nas empresas pois nada impede que elas modifiquem, excluam ou exponham mensagens privadas.

Um modo de resolver o problema da centralização é através da utilização de uma tecnologia de armazenamento descentralizado. Essa abordagem garantiria que todos os usuários tivessem a mesma visão sobre todos os dados, garantindo assim a integridade das mensagens. Um modo de se implementar uma base de dados descentralizada é através da utilização de *blockchain*. Segundo Crosby *et al.* (2016, p. 8, tradução nossa), “uma *blockchain* é essencialmente um banco de dados distribuído de registros, ou livro-razão público de todas as transações ou eventos digitais que foram executados e compartilhados entre as partes participantes”.

Diante do exposto, esse trabalho visa propor uma implementação de correio eletrônico baseada em *blockchain*. Conjectura-se que o armazenamento descentralizado de dados conseguirá resolver grande parte dos problemas encontrados nas atuais implementações de e-mail.

1.1 OBJETIVOS

O objetivo deste trabalho é disponibilizar uma implementação de correio eletrônico baseada em *blockchain*. Os objetivos específicos são:

a) disponibilizar uma API de comunicação com a *blockchain* para o desenvolvimento de aplicações clientes;
b) disponibilizar um protótipo de cliente para validar a funcionalidade da *blockchain*.

1.2 ESTRUTURA

A estrutura deste trabalho é apresentada em quatro capítulos. O primeiro apresenta a introdução e os objetivos geral e específicos. O segundo capítulo apresenta a fundamentação teórica que serve de base para o trabalho. No terceiro capítulo são demonstradas questões referentes à implementação do sistema. São apresentados os requisitos da implementação e diagramas do sistema. São apresentadas ainda imagens do sistema, demonstrando suas funcionalidades e alguns trechos de código quando necessário para complementar a explicação da lógica utilizada. Por fim, neste capítulo são apresentadas as análises dos resultados, realizando comentários sobre os testes executados e uma comparação com os trabalhos correlatos. No quarto capítulo é apresentada a conclusão do trabalho e sugestões para trabalhos futuros.

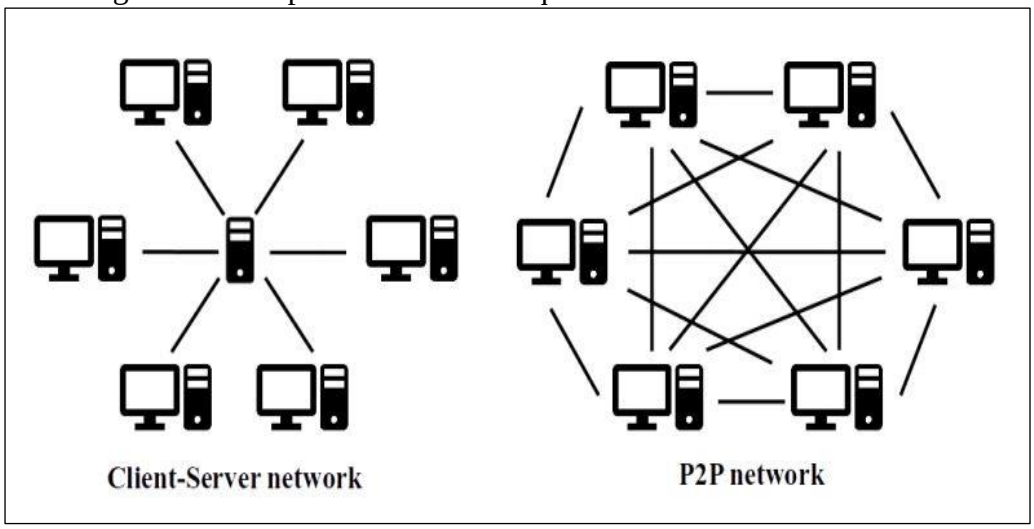
2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo serão apresentados os conceitos e fundamentos mais importantes para a realização desse trabalho. A seção 2.1 trará noções básicas de Peer-to-Peer (P2P) e a seção 2.2 explicara o que é uma *merkle tree*. Na sessão 2.3, será apresentado o InterPlanetary File System (IPFS) e na seção 2.4 será abordado o tema *blockchain*. Por fim, a seção 2.5 apresentará trabalhos correlatos com a aplicação desenvolvida.

2.1 REDES P2P

De acordo com Rocha *et al.* (2004), a arquitetura Peer-to-Peer (P2P) é uma opção ao modelo cliente-servidor. No modelo cliente-servidor, os participantes da rede possuem responsabilidades diferentes: os servidores assumem o papel de fornecedores dos serviços, enquanto os clientes assumem o papel de consumidores. Devido a essa divisão, grande parte dos participantes de um sistema cliente-servidor são apenas coadjuvantes, pois são totalmente dependentes do servidor. Já na arquitetura P2P não existe diferenciação entre os participantes, pois todos possuem as mesmas capacidades e responsabilidades (ROCHA *et al.*, 2004). A Figura 1 mostra um comparativo entre as arquiteturas cliente-servidor e P2P.

Figura 1 – Comparativo entre as arquiteturas cliente-servidor e P2P



Fonte: Veeramani e Jaganathan (2019).

Os dispositivos integrantes de uma rede P2P são chamados de *peers*. Em uma aplicação P2P, a comunicação entre os *peers* é feita através de mensagens enviadas pela internet ou qualquer outro tipo de rede (BUFORD; YU; LUA, 2009). As mensagens precisam seguir algum protocolo que permita o seu entendimento por todos os usuários da aplicação. De acordo com Buford, Yu e Lua (2009), os protocolos utilizados em aplicações P2P possuem as seguintes características:

a) são construídos na camada de aplicação do modelo de rede;
b) conseguem identificar um *peer* através de um endereço único;
c) seguem uma padronização em relação aos tipos de mensagens;
d) suportam mecanismos de roteamento de mensagens.

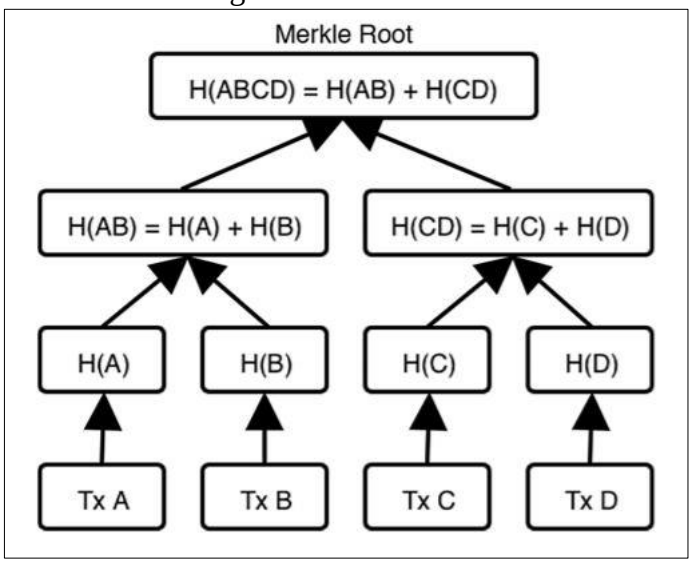
As conexões entre os *peers* em uma rede P2P formam uma rede sobreposta. Conforme Rocha *et al.* (2004), redes sobrepostas são redes virtuais construídas sobre redes existentes. Elas criam uma camada de abstração sobre a rede que possibilita a adição de novos recursos. As redes P2P podem ser classificadas em redes sobrepostas estruturadas e redes sobrepostas não estruturadas (BUFORD; YU; LUA, 2009). As redes não estruturadas não definem nenhuma regra em relação a como as conexões entre os *peers* devem ser feitas. Já nas redes estruturadas, a conexão entre os *peers* é feita de modo determinístico.

Conforme Buford, Yu e Lua (2009), aplicações P2P tendem a ser auto escaláveis. Em aplicações cliente-servidor, a capacidade de processamento do sistema é limitada a quantidade de servidores configurados, o que implica que o sistema é capaz de atender apenas a um determinado número de usuários. Já em sistemas P2P, a capacidade de processamento total é a soma do compartilhamento de recursos entre todos os usuários. Desse modo, a adição de um novo usuário em uma aplicação faz com que o poder de processamento dessa aplicação aumente.

2.2 MERKLE TREE

Uma *merkle tree* é uma estrutura de dados que se organiza como uma árvore binária. Essa árvore é construída a partir dos nós folhas através da aplicação recursiva de funções *hash* sobre pares de valores (GAMAGE; WEERASINGHE; DIAS, 2020). Utilizando-se como exemplo o cenário apresentado na Figura 2, tem-se inicialmente quatro valores: Tx A, Tx B, Tx C e Tx D. Primeiramente é calculado o *hash* de cada um dos valores (H(A), H(B), H(C), H(D)). Os valores gerados são armazenados nos nós folhas da *merkle tree*. Após isso, se inicia o processo recursivo. Todos os nós que não possuem um nó pai são agrupados em pares. A partir de cada par, é gerado um novo nó que assume o papel de nó pai do par. Esse novo nó armazena o *hash* gerado sobre a concatenação dos valores de cada nó do par (H(AB) = H(A) + H(B), H(CD) = H(C) + H(D)). Esse processo recursivo continua até que o nó raiz da árvore seja alcançado (H(ABCD) = H(AB) + H(CD)). O nó raiz de uma *merkle tree* é chamado de *merkle root*.

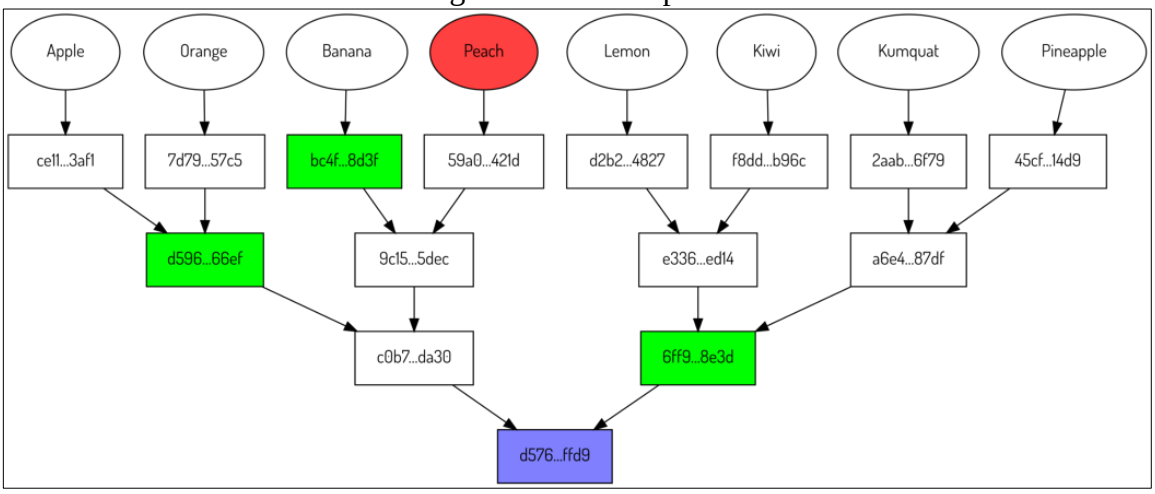
Figura 2 – Merkle tree



Fonte: Gamage, Weerasinghe e Dias (2020).

Uma das vantagens na utilização de *merkle trees* é a possibilidade de verificar dados através de *merkle proofs*. Uma *merkle proof* é uma forma de provar que um determinado valor faz parte de um conjunto de dados sem exigir que nenhum dos outros valores seja exposto (MCDONALD, 2019). Um exemplo de utilização de uma *merkle proof* pode ser visto na Figura 3. No cenário apresentado na Figura 3, deseja-se provar que o valor “Peach” (em vermelho) faz parte da *merkle tree*. Para aplicar a *merkle proof*, basta se ter a posse do *merkle root* (em azul) e dos nós intermediários (em verde). Os nós intermediários são um conjunto formado pelo nó par do valor sendo verificado e todos os nós que são raízes de sub-árvores não ascendentes ao nó sendo verificado. Com estes dados, basta calcular os *hashes* da sub-árvore do nó que armazena o valor “Peach” e validar o *merkle root* gerado.

Figura 3 – Merkle proof



Fonte: Mcdonald (2019).

De acordo com Kansal (2020), *merkle trees* são úteis principalmente em cenários distribuídos que exigem verificação e sincronização de dados. Como exemplo, pode ser

utilizado um sistema distribuído de compartilhamento de arquivos. Neste sistema, cada arquivo é dividido em blocos e compartilhado entre os *peers* de uma rede P2P. Para recuperar um arquivo, é necessário, por questões de performance, baixar blocos de vários *peers* ao mesmo tempo. Neste cenário, uma *merkle tree* pode ser utilizada para garantir que os arquivos não foram modificados. Inicialmente, é necessário gerar a *merkle tree* na criação do arquivo. Os nós folhas armazenam os *hashes* gerados a partir de cada bloco do arquivo. Com a *merkle tree* gerada, bastaria, a cada bloco baixado, fazer a verificação da *merkle proof* para garantir que o arquivo não foi modificado.

2.3 IPFS

O Hypertext Transfer Protocol (HTTP) é o protocolo padrão para compartilhamento de arquivos na web. Devido a sua idade e ao seu acoplamento a basicamente qualquer navegador de internet, o HTTP é utilizado pela vasta maioria das aplicações que rodam na internet. Apesar de sua popularidade, o protocolo não evoluiu de acordo com as novas técnicas referentes a distribuição de arquivos. No HTTP, o compartilhamento de arquivos é feito de modo centralizado, onde cada arquivo precisa ser armazenado em servidores próprios que possuem o poder para deletar ou modificar qualquer arquivo. Além disso, a centralização em servidores torna qualquer aplicação muito mais vulnerável a falhas, pois basta uma falha no servidor para que todos os arquivos armazenados nele fiquem inacessíveis. Neste cenário, Benet (2014) apresentou o InterPlanetary File System (IPFS), um sistema de compartilhamento de arquivos P2P que visa, de acordo com as mais avanças técnicas de distribuição de arquivos, resolver parte dos problemas presentes na web.

O protocolo HTTP faz o endereçamento dos objetos com base em sua localização. Através do Uniform Resource Locator (URL) é possível identificar em qual servidor um determinado arquivo está armazenado. Esta abordagem baseada em localização impossibilita identificar um arquivo pelo seu conteúdo. Por exemplo, caso um usuário tenha feito o *upload* de um vídeo em uma plataforma online e após algum tempo decida mover o vídeo para outra plataforma, obrigatoriamente o identificador do vídeo (URL) vai ser alterado, pois está estritamente ligado a plataforma que o armazena. Já o IPFS faz o endereçamento dos arquivos através de um identificador único gerado através da aplicação de uma função *hash* sobre o conteúdo do arquivo (BENET, 2014). Essa abordagem, além de ser independente de plataformas, elimina a existência de arquivos duplicados e permite identificar se um arquivo teve o seu conteúdo alterado por algum *peer*.

Uma das estruturas base do IPFS é a Distributed Hash Table (DHT). De acordo com Benet (2014), as DHTs são largamente utilizadas para armazenar metadados em sistemas P2P. No geral, uma DHT se trata de uma estrutura de dados chave-valor distribuída entre vários *peers* de uma rede P2P. No IPFS, as DHTs são utilizadas como mecanismo de roteamento: através delas é possível identificar os *peers* que armazenam um determinado objeto e identificar o endereço de rede de cada *peer*.

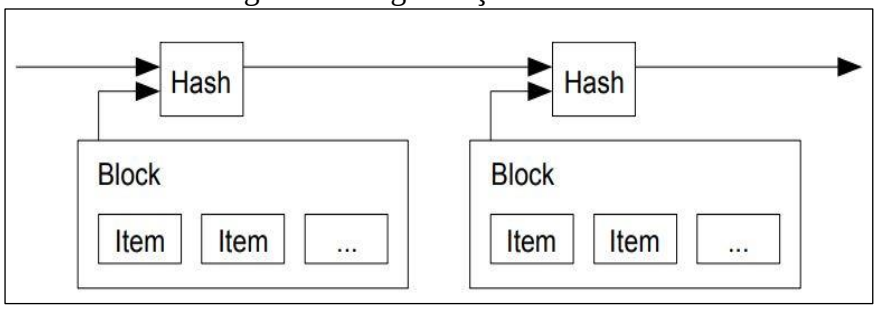
Cada objeto armazenado no IPFS possui um valor e uma lista de links (ponteiros para outros objetos). O conjunto de objetos armazenados no IPFS formam um grafo acíclico dirigido chamado de *merkle dag*. A *merkle dag* é uma estrutura de dados similar a uma *merkle tree*, porém, diferentemente da *merkle tree*, ela não precisa ser balanceada e pode armazenar dados em todos os nós (BENET, 2014). Para armazenar um arquivo pequeno no IPFS, é criado apenas um objeto tendo como valor o conteúdo do arquivo. Já para arquivos maiores, é necessário dividir o arquivo em vários blocos menores e armazenar cada bloco em um objeto separado. Por fim, é criado um objeto que faz a junção de todos os blocos através do armazenamento de links para os objetos criados.

2.4 BLOCKCHAIN

Uma *blockchain* pode ser definida como uma base de dados distribuída e descentralizada (GAMAGE; WEERASINGHE; DIAS, 2020). Os dados são distribuídos através de uma rede P2P e são protegidos através da utilização de criptografia e algoritmos de consenso. Cada *peer* da rede mantém uma cópia local dos dados. As operações de busca e adição de dados são feitas através da comunicação entre os *peers*. Cada novo dado deve ser replicado entre todos os *peers* da rede

Conforme Crosby *et al.* (2016), os dados de uma *blockchain* são adicionados em transações. De tempos em tempos, um grupo de transações é acumulado e adicionado em um bloco. Cada bloco possui obrigatoriamente uma lista de transações, um identificador único (*hash*) e o identificador do último bloco adicionado na *blockchain*. Conforme pode ser observado na Figura 4, a lista de blocos é armazenada em uma estrutura de lista encadeada, sendo cada bloco diretamente ligado ao seu antecessor. Como cada *peer* armazena essa lista encadeada localmente, cada usuário pode escolher que tipo de sistema de armazenamento deseja utilizar. Já para a replicação dos blocos, todos os usuários devem seguir estritamente um protocolo previamente definido pela aplicação, garantindo assim que todos os *peers* entendam o formato das mensagens enviadas.

Figura 4 – Organização dos blocos



Fonte: Nakamoto (2008).

Sistemas baseados em *blockchain* utilizam conceitos de criptografia de chave pública para verificar a integridade dos dados de um usuário (GAMAGE; WEERASINGHE; DIAS, 2020). Cada usuário possui uma chave privada e uma chave pública. Todas as transações criadas por um usuário devem ser assinadas com a sua chave privada. A veracidade de uma transação pode ser facilmente verificada através da chave pública do usuário, garantindo assim que não seja possível criar transações em nome de outro usuário. Esse mesmo par de chaves pode ser utilizado para criptografar as mensagens enviadas. Por exemplo, caso se deseje enviar uma mensagem privada para o usuário B, basta criptografar a mensagem com a chave pública do usuário B e adicionar a transação na *blockchain*. Por mais que todos os *peers* tenham acesso a transação, apenas o usuário B, por meio de sua chave privada, conseguirá descriptografar a mensagem e ler o seu conteúdo.

A grande vantagem da *blockchain* é a sua natureza descentralizada. Um exemplo é o seu uso no sistema financeiro. Segundo Crosby *et al.* (2016), a economia digital atual necessita de uma autoridade confiável que tem a tarefa de validar as transações financeiras e garantir a autenticidade delas. Por mais que as pessoas confiem nessas entidades centrais, ainda assim elas se apresentam como um ponto de falha, sendo suscetíveis a invasões e manipulações. Essas autoridades centrais existem em diversos domínios além do setor financeiro, como serviços de e-mail, redes sociais, sistemas de cartório etc. Com a utilização de *blockchain*, a necessidade da existência de uma entidade central é eliminada.

É possível categorizar as *blockchains* em dois tipos: *blockchains* públicas e *blockchains* privadas (GAMAGE; WEERASINGHE; DIAS, 2020). As *blockchains* públicas são acessíveis por qualquer usuário que deseje fazer parte da rede. Qualquer um tem permissão para ler os dados e participar do processo de criação dos blocos. Já as *blockchains* privadas são restritas a um grupo fechado de usuários. Esse tipo de *blockchain* é geralmente utilizada no meio corporativo. As *blockchains* privadas podem ser divididas em dois tipos: parcialmente descentralizada e centralizada (GAMAGE; WEERASINGHE; DIAS, 2020). As *blockchains* privadas parcialmente descentralizadas restringem quem pode fazer a inclusão de novos blocos,

porém, permitem que qualquer um faça a leitura dos dados. Já as *blockchains* privadas centralizadas não permitem que partes não autorizadas façam a escrita ou leitura dos dados. O Quadro 1 mostra uma comparação das características dos diferentes tipos de *blockchain* com uma base de dados distribuída privada tradicional.

Quadro 1 – Comparativo entre tipos de *blockchain*

Características/ Tipos	*Blockchain* pública	*Blockchain* privada parcialmente descentraliza	*Blockchain* privada centralizada	Base de dados distribuída privada tradicional
Acesso público para leitura	Permitido	Permitido	Não permitido	Não permitido
Acesso público para escrita	Permitido	Não permitido	Não permitido	Não permitido
Imutabilidade	Alta	Média	Baixa	Baixa
Taxa de transferência	Baixa	Média	Alta	Alta
Escalabilidade	Baixa	Média	Alta	Alta
Descentralização	Alta	Média	Baixa	Baixa
Distribuição	Alta	Média	Baixa	Baixa
Auditabilidade	Alta	Alta	Baixa	Baixa

Fonte: Gamage, Weerasinghe e Dias (2020, tradução nossa).

2.4.1 Bitcoin

Em 2008, um indivíduo utilizando o pseudônimo Satoshi Nakamoto publicou um artigo chamado “Bitcoin: A Peer-to-Peer Electronic Cash System” (CROSBY *et al.*, 2016). Neste artigo, o autor apresenta a estrutura de um sistema financeiro digital que não necessita de uma entidade terceira para o envio de dinheiro entre duas partes. O trabalho de Nakamoto (2008) apresentou um conjunto de técnicas e algoritmos que resultaram na tecnologia conhecida como *blockchain*. Ao longo do tempo, o conceito de *blockchain* foi separado do conceito de Bitcoin, sendo que a *blockchain* pode ser utilizada em diversos outros domínios além do setor financeiro. Entretanto, o entendimento do Bitcoin, devido a sua origem, se torna fundamental para o entendimento prático dos conceitos envolvidos em sistemas baseados em *blockchain*.

No Bitcoin, cada usuário possui uma carteira digital que serve para armazenar suas chaves pública e privada. De acordo com Crosby *et al.* (2016), para um usuário fazer a transferência de bitcoins, ele cria uma transação e assina a mesma com a sua chave privada. Por último, ele define como destino a chave pública do destinatário e envia a transação para os outros *peers* da rede. Cada *peer,* ao receber a transação, verifica se ela é válida. A verificação envolve confirmar que a assinatura da transação é autêntica e garantir que o remetente não está enviando mais bitcoins do que o total que ele possui. Eventualmente, todos os *peers* recebem a transação,

porém, neste ponto, a transação ainda não foi confirmada. Para uma transação ser considerada autêntica, ela precisa ser adicionada em um bloco da *blockchain*.

A necessidade do agrupamento das transações em blocos surgiu como resolução do problema de consenso distribuído. Por se tratar de uma moeda virtual, é necessário que todos os *peers* da rede concordem em relação a quanto dinheiro cada usuário possui. Segundo Nakamoto (2008), esse problema pode ser resolvido com a existência de um sistema que possibilite que todos os usuários armazenem um histórico ordenado de todas as transações realizadas. As transações não confirmadas são agrupadas em um bloco e adicionadas na *blockchain* através de um processo chamado mineração. O primeiro *peer* que conseguir minerar o bloco faz a replicação dele para toda a rede.

O processo de mineração envolve aplicar uma função *hash* (SHA-256 no caso do Bitcoin) sobre o conteúdo do bloco (NAKAMOTO, 2008), cujo objetivo é encontrar um *hash* que atenda a certos pré-requisitos. Atualmente, para um *hash* ser válido, ele precisa iniciar com uma determinada quantidade de zeros a esquerda. Essa quantidade é calculada dinamicamente de um modo que cada bloco demore em média dez minutos para ser minerado. Através desse processo, o único modo de criar um bloco válido é provando que uma determinada capacidade de processamento computacional foi utilizada.

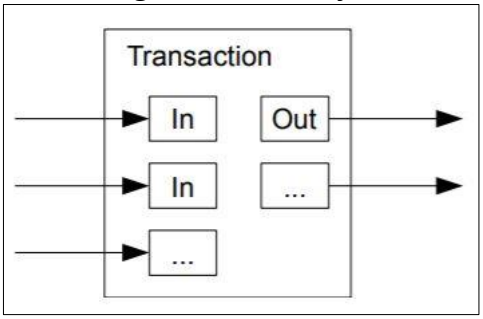
No processo de mineração é possível que o mesmo bloco seja minerado por dois usuários ao mesmo tempo. Ao terminar a mineração, ambos os usuários iriam enviar o bloco para os outros *peers*. Cada *peer* vai aceitar como válido o primeiro bloco que receber. Neste cenário, a rede não possui consenso, pois não consegue determinar qual o último bloco válido. De acordo com Nakamoto (2008), os *peers*, ao receberem diferentes versões da *blockchain*, irão aceitar a que tiver a maior quantidade de blocos. Eventualmente, os *peers* entram em consenso pois algum minerador irá conseguir finalizar o processo de mineração sozinho e replicar a sua versão para a rede.

A mineração faz com que os dados da *blockchain* sejam praticamente imutáveis. Essa garantia de imutabilidade existe porque um dos dados utilizados para a geração do *hash* de um bloco é o *hash* do bloco anterior. Desse modo, para alterar um bloco antigo é necessário recalcular o seu *hash* e o *hash* de todos os blocos subsequentes a ele. O único modo de um fraudador conseguir alterar uma *blockchain* é caso ele possua sozinho 51% de todo o poder computacional da rede (NAKAMOTO, 2008). Neste cenário, o fraudador conseguiria criar blocos de modo mais rápido que todos os outros *peers* e eventualmente conseguiria gerar uma *blockchain* maior que a gerada pelos *peers* honestos.

São os mineradores que garantem a integridade da rede, tornando-se obrigatória a presença de mineradores para o correto funcionamento do Bitcoin. O problema é que o processo de mineração é caro. Os mineradores precisam investir em hardwares de última geração e acabam gastando muito dinheiro com eletricidade para manter esses hardwares funcionando. Devido a isso, é necessário que os mineradores recebam algum incentivo. A solução proposta por Nakamoto (2008) foi permitir que, a cada bloco gerado, o minerador receba alguns bitcoins como recompensa. Parte desses bitcoins são gerados junto com o bloco e a outra parte são as taxas de transação pagas pelos remetentes.

O Bitcoin não armazena diretamente o valor que cada usuário possui. Conforme pode ser visto na Figura 5, cada transação armazenada na *blockchain* do Bitcoin possui uma lista de *inputs* e uma lista de *outputs* (NAKAMOTO, 2008). Cada *output* de uma transação possui um valor e o endereço do destinatário. O total de bitcoins que um determinado usuário possui é representado pela soma de todos os valores dos *outputs* criados para o endereço dele que ainda não foram gastos. A partir do momento que um *output* é referenciado por um *input*, ele não pode mais ser utilizado em outra transação. A soma dos *outputs* de uma transação deve ser menor ou igual ao total referenciado nos *inputs*. Caso a soma dos *inputs* seja maior que o total que se deseje enviar, o usuário pode adicionar na transação um *output* que retorna o valor excedido para a sua conta. De acordo com Nakamoto (2008), esse modelo de armazenamento facilita a gestão das transações, pois elimina a necessidade de criar uma transação diferente para cada centavo enviado.

Figura 5 – Transação



Fonte: Nakamoto (2008).

Cada bloco da *blockchain* do Bitcoin possui uma lista de cabeçalhos e uma lista de transações. Todos os *peers* que desejam atuar como mineradores ou que desejam ser independentes de qualquer entidade externa devem manter armazenados tanto os cabeçalhos quanto as transações. O problema dessa abordagem é o alto custo com armazenamento, pois cada novo bloco gerado deve ser armazenado junto com suas transações. Com novos blocos sendo criados a cada dez minutos, o tamanho total da rede do Bitcoin já é maior que 300GB

(FROST, 2020). Esse tamanho dificulta a utilização do Bitcoin em dispositivos pessoais com menor capacidade de armazenamento. Para resolver esse problema, a proposta de Nakamoto (2008) foi armazenar nos cabeçalhos do bloco um *merkle root* gerado a partir de todas as transações do bloco. Desse modo, é possível que usuários com menor capacidade de armazenamento mantenham salvo apenas os cabeçalhos dos blocos. A única ação que esses usuários conseguem executar é a realização da verificação de pagamentos.

A verificação de pagamentos é a ação de validar se um determinado pagamento já foi confirmado no Bitcoin. Sendo assim, ela envolve descobrir se uma determinada transação foi incluída em um determinado bloco. Essa validação é feita através de uma *merkle proof*. O usuário, através do *hash* da transação que deseja validar, faz buscas na rede P2P para recuperar os nós intermediários da *merkle tree*. Com os nós intermediários é possível gerar a *merkle tree* e comparar o *merkle root* gerado com o *merkle root* armazenado no cabeçalho do bloco. Caso sejam iguais, o usuário consegue afirmar que um determinado pagamento foi confirmado. De acordo com Nakamoto (2008), essa abordagem pode apresentar problemas de segurança caso *peers* maliciosos entrem em grande número e de modo coordenado na rede. Esses *peers*, ao receberem buscas, poderiam enganar o usuário através do retorno de dados incorretos.

2.4.2 Algoritmos de consenso

No contexto de *blockchains*, consenso é o mecanismo no qual os *peers* concordam sobre a validade, autenticidade e ordem das transações (GAMAGE; WEERASINGHE; DIAS, 2020). Como todos os dados da *blockchain* são descentralizados, um mecanismo de consenso é fundamental pois é o único modo de garantir que todos os *peers* tenham a mesma visão referente aos dados. Os algoritmos de consenso resolvem um famoso problema em sistemas distribuídos conhecido como o problema dos generais bizantinos.

No problema dos generais bizantinos, um grupo de generais precisa organizar um ataque coordenado contra uma cidade (GAMAGE; WEERASINGHE; DIAS, 2020). O único modo de os generais não serem derrotados é, ou todos atacarem ao mesmo tempo, ou todos decidirem não atacar. Os generais, por estarem posicionados em lugares diferentes, precisam se comunicar através do envio de mensageiros. O problema é que podem existir generais traidores que enviam mensagens falsas com o objetivo de confundir a todos. É impossível garantir consenso entre os generais caso pelo menos um terço desses generais sejam traidores (GAMAGE; WEERASINGHE; DIAS, 2020). Levando o problema para o cenário de *blockchains*, cada general representa um *peer* da rede e os mensageiros representam o envio de mensagens entre os *peers*. Os algoritmos de consenso têm o objetivo de tornar a *blockchain* tolerante ao problema

dos generais bizantinos, ou seja, eles garantem que a *blockchain* funcione corretamente mesmo que alguns *peers* falhem ou se comportem de modo malicioso.

O algoritmo de consenso mais famoso utilizado em sistemas baseados em *blockchain* é o *proof of work*. Esse algoritmo obriga os *peers* resolverem um problema computacional complexo antes de adicionarem um bloco na *blockchain* (GAMAGE; WEERASINGHE; DIAS, 2020). O primeiro *peer* da rede que conseguir resolver o problema é o único que consegue adicionar o bloco. Desse modo, quanto mais poder computacional um *peer* tiver, maior sua chance de criar um bloco válido. Um ponto negativo do *proof of work* é seu grande gasto de energia. Estima-se que o uso do *proof of work* no Bitcoin consuma energia suficiente para abastecer todo o país da Suíça (UMLAUF, 2019).

Devido ao alto gasto energético do *proof of work*, várias pesquisas foram realizadas na tentativa de desenvolver novos algoritmos de consenso. Um dos algoritmos criados foi o *proof of stake*. Neste algoritmo, indivíduos chamados validadores são selecionados para fazerem a criação dos blocos (GAMAGE; WEERASINGHE; DIAS, 2020). O critério para a seleção dos validadores leva em conta o interesse econômico que os *peers* tem para manter a rede funcionando corretamente. Cada *peer* que deseja se tornar um validador aposta uma determinada quantidade de moedas. As moedas apostadas são bloqueadas e não podem ser utilizadas pelo *peer* enquanto um validador não for selecionado. Quanto maior o valor apostado por um *peer*, maior a sua chance de ser selecionado para criar o bloco. O *proof of stake* implementa mecanismos para punir validadores que tentem fraudar a rede. Geralmente os fraudadores perdem as moedas que apostaram e são eliminados do processo de seleção de validadores. Um problema da estratégia baseada em aposta é que, por mais que dois *peers* apostem valores semelhantes, o total de moedas que eles possuem pode ser completamente diferente. Desse modo, o valor apostado não representa o mesmo risco para todos os *peers*, possibilitando que usuários mais ricos tenham um maior incentivo a não se comportarem corretamente.

Uma abordagem semelhante ao *proof of stake* é o *proof of authority*. Neste algoritmo, os usuários com interesse em se tornarem validadores, ao invés de apostarem moedas, precisam revelar a sua identidade (CURRAN, 2018). O número total de *peers* que podem se tornar validadores é controlado. Apenas *peers* pré-aprovados podem ser selecionados para criarem os blocos. Caso um validador se comporte de modo malicioso, ele é automaticamente excluído da lista de validadores e tem sua reputação manchada. O *proof of authority* é o algoritmo ideal para ser utilizado em *blockchains* privadas, pois permite que as partes confiáveis sejam as únicas com permissão para criarem os blocos. O algoritmo também pode ser utilizado em

blockchains públicas, porém ele não apresenta o mesmo nível de descentralização do *proof of work* ou do *proof of stake*.

2.4.3 IBC

Após o surgimento do Bitcoin, muitas aplicações baseadas em *blockchain* foram criadas. Inicialmente, todas as aplicações eram construídas de modo isolado, cada uma rodando em sua própria *blockchain* e sem a possibilidade de se comunicar com outras aplicações. Em uma segunda geração surgiram os *smart contracts*, que possibilitam criar diversas aplicações diferentes em uma mesma *blockchain*. Os *smart contracts* tornaram muito simples o desenvolvimento de novas aplicações, pois permitem que os desenvolvedores foquem apenas na aplicação, sendo a *blockchain* mantida e gerenciada por uma comunidade independente. O problema dessa abordagem é que os desenvolvedores da aplicação não têm soberania sobre a *blockchain* e dependem da comunidade que a mantém para a implementação de melhorias e correções de bugs. Foi neste cenário que foi criado o Interblockchain Communication Protocol (IBC).

De acordo com Goes (2020), o IBC é um protocolo que possibilita *blockchains* independentes se comunicarem de modo seguro e voluntário. Essa comunicação seria útil, por exemplo, para fazer a troca entre diferentes moedas sem a necessidade de uma entidade central. O objetivo do IBC é criar uma internet de *blockchains*, possibilitando diferentes aplicações compartilharem recursos e serviços. O protocolo gerencia a autenticação, transporte e ordenação dos pacotes de dados enviados entre as diferentes *blockchains*. A comunicação feita pelo protocolo é realizada entre módulos. Os módulos são os componentes de uma *blockchain* e geralmente são divididos com base na divisão de responsabilidades. Os dados trafegados não possuem nenhum significado para o protocolo, sendo responsabilidade de cada módulo atribuir a devida semântica.

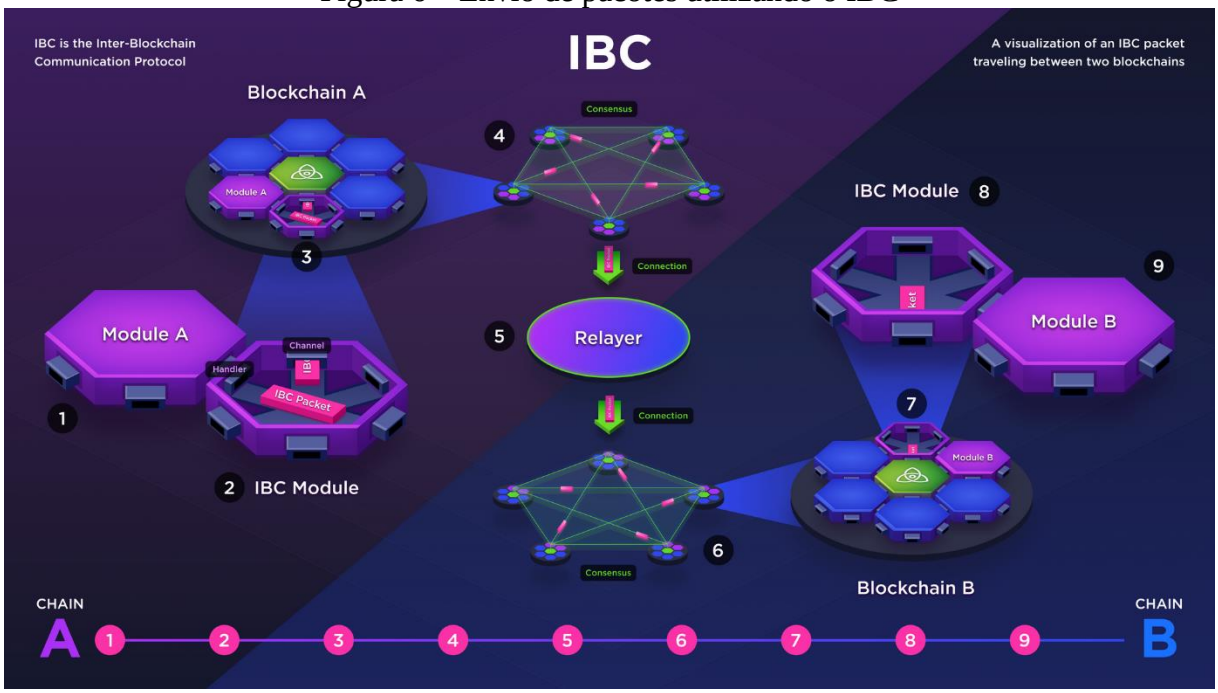
Um exemplo do fluxo de envio de um pacote utilizando o IBC pode ser visto na Figura 6. Segundo Goes (2020), a implementação do protocolo IBC envolve a implementação das seguintes abstrações:

a) *clients*: possibilitam uma *blockchain* identificar atualizações de estado de outra *blockchain*. Os *clients* devem aceitar apenas atualizações que foram aprovadas pelo algoritmo de consenso da *blockchain* de origem;
b) *connections*: fazem a ligação entre dois *clients* de diferentes *blockchains*;
c) *channels*: canais para o envio de pacotes entre módulos de diferentes *blockchains*. Os *channels* garantem que os dados sejam enviados apenas uma vez e que sejam

recebidos na ordem correta. Cada *channel* precisa estar ligado a uma *connection*, podendo uma *connection* estar ligado a diversos *channels*;

d) *relayers*: processos separados que fazem a conexão física entre as diferentes *blockchains* através de protocolos de rede (TCP/IP, UDP/IP). Os *relayers* monitoram os estados das *blockchains* e fazem o envio de datagramas com as devidas atualizações.

Figura 6 – Envio de pacotes utilizando o IBC



Fonte: Waugh (2020).

2.5 TRABALHOS CORRELATOS

Neste capítulo são apresentados trabalhos com características semelhantes aos principais objetivos do trabalho desenvolvido. Na seção 2.5.1 é apresentado o trabalho de Warren (2012), que desenvolveu uma aplicação para o envio de mensagens eletrônicas utilizando a arquitetura P2P. A seção 2.5.2 detalha o trabalho de Mislove *et al.* (2003), que desenvolveram uma infraestrutura para o desenvolvimento de aplicações P2P. Por fim, a seção 2.5.3 descreve o artigo de Kageyama, Maziero e Santin (2008), que desenvolveram uma aplicação P2P para envio de correio eletrônico baseado em *polling* de mensagens.

2.5.1 Bitmessage

O sistema proposto por Warren (2012) tem como objetivo permitir que os usuários enviem e recebam mensagens eletrônicas de modo seguro e descentralizado. O sistema foi desenvolvido com base na arquitetura P2P, o que elimina a necessidade de uma entidade central.

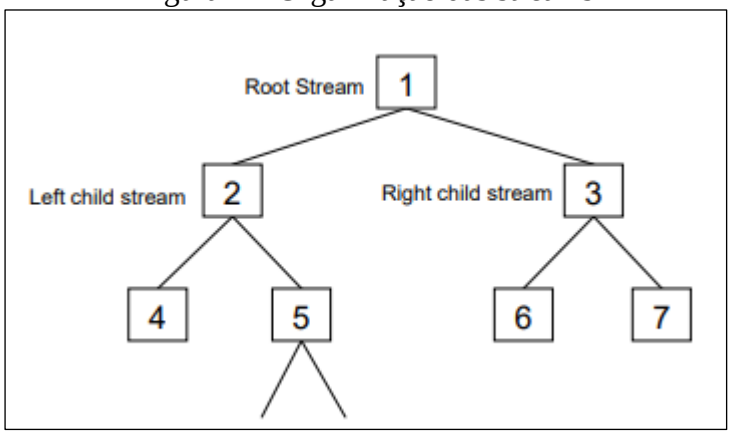
Para manter a privacidade dos usuários, todas as mensagens são criptografadas através de técnicas de criptografia de chave pública. Além disso, o sistema garante a anonimidade das partes envolvidas em uma mensagem.

Para utilizar o sistema, cada usuário precisa fazer um cadastro. Esse processo é realizado localmente no dispositivo do usuário e envolve unicamente a geração de um par de chaves que serão utilizados nos algoritmos de criptografia. A chave pública representa o endereço do usuário na rede (WARREN, 2012), ou seja, para enviar uma mensagem para um usuário, é necessário ter acesso a sua chave pública.

O envio das mensagens é feito através da comunicação direta entre os *peers* da rede P2P. De acordo com Warren (2012), por se tratar de uma arquitetura descentralizada, é necessário que todos os usuários recebam todas as mensagens enviadas. Cada usuário, ao receber uma mensagem, deve tentar descriptografar a mesma com a sua chave privada. Apenas o destinatário vai conseguir realizar a descriptografia com sucesso.

O problema de todos os *peers* receberem todas as mensagens é que isso pode sobrecarregar a rede. Devido a isso, Warren (2012) propôs a divisão da rede em várias sub-redes independentes chamadas de *streams*. Conforme pode ser visto na Figura 7, as *streams* são estruturadas de modo hierárquico, o que permite uma comunicação eficiente entre elas. Todos os usuários são por padrão criados na *stream* 1. Caso o usuário deseje utilizar outra *stream*, basta que ele cadastre um par de chaves nela.

Figura 7 – Organização das streams



Fonte: Warren (2012).

Um problema comum em sistemas de mensagens eletrônicas é o envio de *spam*. Para resolver isso, Warren (2012) propôs que, para enviar uma mensagem, o usuário precise resolver um problema computacional de colisão de *hash*. Esse problema se resume a encontrar um *hash* que atenda alguns requisitos e é configurado para levar em média 4 minutos para ser resolvido. O *hash* encontrado é enviado junto com a mensagem. Os *peers* da rede só aceitam mensagens

que possuam um *hash* válido. Essa abordagem mitiga o envio de *spam*, pois garante que um usuário possa enviar apenas uma mensagem a cada 4 minutos.

Como conclusão, Warren (2012) compara a solução desenvolvida com os sistemas de e-mail padrões. O autor afirma que o sistema consegue preencher a lacuna entre a facilidade de uso do e-mail e a segurança no uso de técnicas de criptografia, enquanto consegue garantir a anonimidade dos usuários.

2.5.2 POST

Mislove *et al.* (2003) tiveram como objetivo criar uma infraestrutura que permite o desenvolvimento de aplicações colaborativas descentralizadas, como e-mail, chat, aplicativos de notícias etc. Essa infraestrutura, nomeada de POST, não depende de nenhum servidor dedicado para funcionar, pois tem como base uma rede P2P. Além da descentralização, a utilização da arquitetura P2P traz algumas vantagens como resiliência, segurança e escalabilidade (MISLOVE *et al.* 2003).

A infraestrutura desenvolvida utilizou como base o Pastry, uma rede sobreposta arquitetada para ser escalável, tolerante a falhas e auto-organizada (MISLOVE *et al.* 2003). O Pastry implementa uma DHT. Cada *peer* e objeto armazenado na rede possui um identificador único. Dado um objeto e o seu identificador, o Pastry consegue de modo eficiente armazenar o objeto e recuperar o mesmo futuramente.

O POST fornece três serviços básicos: um repositório seguro de mensagens, serviços de notificação e um repositório de metadados (MISLOVE *et al.*, 2003). O repositório de mensagens é utilizado para armazenar de modo seguro todas as mensagens que são enviadas. Os serviços de notificação possibilitam notificar os usuários sobre eventos que ocorreram no sistema, como, por exemplo, o envio de uma mensagem. Já o repositório de metadados permite adaptar as mensagens de acordo com o aplicativo cliente utilizado (MISLOVE *et al.*, 2003).

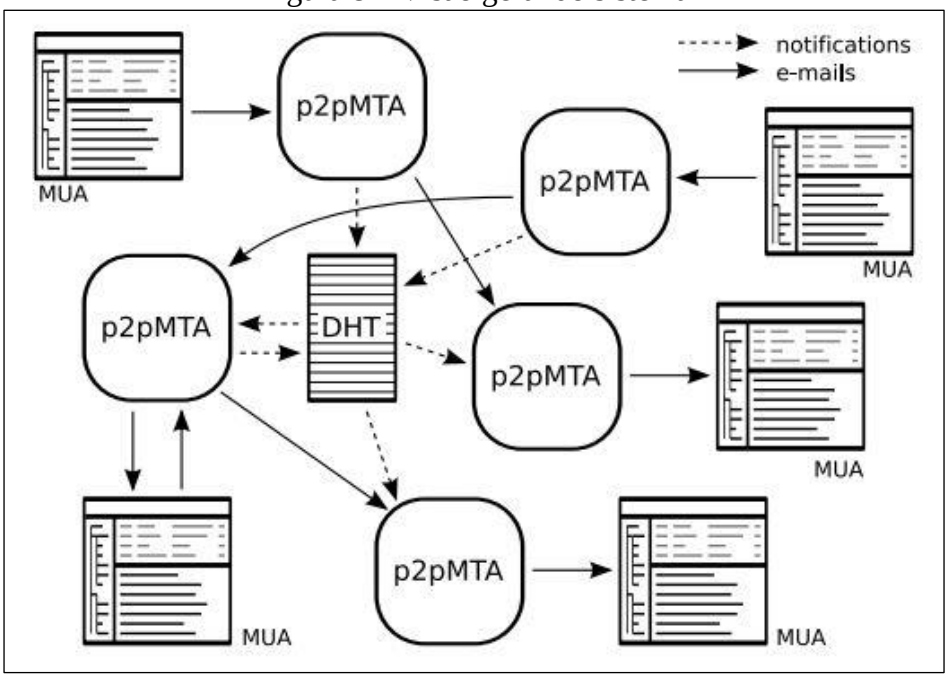
Como exemplo de uso do POST, Mislove *et al.* (2003) desenvolveram o ePOST, um aplicativo de e-mail descentralizado. Os usuários desse sistema precisam executar uma daemon em segundo plano. Essa daemon, além de implementar os serviços do POST, funciona como um servidor SMTP e implementa o Internet Message Access Protocol (IMAP), o que possibilita a compatibilização com os clientes de e-mail convencionais. As mensagens do usuário são armazenadas no repositório de mensagens do POST e são enviadas ao destinatário através de uma notificação. O controle do estado da caixa de e-mail é feito através do repositório de metadados.

2.5.3 Sistema de e-mail P2P baseado em polling

O trabalho de Kageyama, Maziero e Santin (2008) apresenta a arquitetura de um sistema de e-mail P2P baseado no *polling* de mensagens. Os e-mails são enviados diretamente entre o remetente e o destinatário, não existindo a necessidade de um servidor central. Uma DHT é utilizada para enviar notificações entre os usuários e para armazenar informações de controle (KAGEYAMA; MAZIERO; SANTIN, 2008).

Cada *peer* do sistema roda uma aplicação chamada p2pMTA que funciona como um servidor SMTP/IMAP. O formato das mensagens utilizado pelo p2pMTA segue o formato padrão dos protocolos de e-mail convencionais. Desse modo, é possível o reaproveitamento do Mail User Agent (MUA) utilizado com o SMTP. A Figura 8 mostra uma visão geral da arquitetura do sistema.

Figura 8 – Visão geral do sistema



Fonte: Kageyama, Maziero e Santin (2008).

Para iniciar o uso do sistema, cada usuário deve possuir um par de chaves criptográficas. A chave pública deve ser exposta através de um serviço que possibilite os outros *peers* da rede recuperarem ela. Além disso, cada *peer* deve adicionar na DHT um descritor, utilizando como chave seu endereço de e-mail (KAGEYAMA; MAZIERO; SANTIN, 2008). Esse descritor contém o IP e porta do *peer*. Os descritores possibilitam que os *peers* consigam se comunicar através de um endereço de e-mail.

Após configurar o descritor, o usuário pode realizar o envio de mensagens. Primeiramente é necessário adicionar na DHT uma notificação criptografada com a chave pública do destinatário. Ao receber essa notificação, o destinatário tem o poder de decidir se

deseja ou não receber a mensagem. Caso deseje, ele invoca diretamente o remetente para recuperar o e-mail (KAGEYAMA; MAZIERO; SANTIN, 2008).

Como conclusão, Kageyama, Maziero e Santin (2008) afirmam que a maior contribuição da arquitetura apresentada é colocar a sobrecarga de armazenamento no lado do remetente. Desse modo, usuários que enviam *spam* irão encher suas próprias caixas de e-mail. Outro benefício é que, caso o destinatário não deseje receber uma mensagem, ela não será transferida do remetente, economizando assim sua banda larga (KAGEYAMA; MAZIERO; SANTIN, 2008).

3 DESENVOLVIMENTO

Neste capítulo serão abordadas as etapas de desenvolvimento do sistema. A seção 3.1 apresenta os requisitos funcionais e não funcionais do projeto desenvolvido. Na seção 3.2 são apresentados o diagrama de casos de uso, o diagrama entidade relacionamento, a arquitetura da aplicação e os diagramas de atividades, especificando e detalhando o funcionamento do sistema. A seção 3.3 detalha a implementação da *blockchain* e do protótipo de cliente, destacando os principais pontos. Por último, na seção 3.4 é apresentada a análise dos resultados obtidos no trabalho.

3.1 REQUISITOS

O objetivo deste trabalho é disponibilizar uma implementação de correio eletrônico utilizando como base uma *blockchain*. Sistemas baseados em *blockchain* são geralmente descentralizados, não existindo uma entidade responsável por manter a rede funcionando. Devido a isso, a grande maioria dos sistemas que utilizam *blockchain* implementa um sistema financeiro próprio baseado em criptomoedas. Esse sistema financeiro serve como incentivo para que os próprios usuários mantenham a rede P2P funcionando corretamente.

Além disso, outra característica de sistemas que utilizam *blockchain* é que todos os dados são replicados entre todos os usuários. Essa característica, dependendo do tipo de dado, pode se tornar um problema pois elimina *peers* com menor capacidade de armazenamento computacional. Com menos *peers* funcionando, o sistema consequentemente se torna menos confiável e mais suscetível a ataques. Desse modo, é importante que apenas os dados essenciais sejam armazenados na *blockchain.* Os dados que ocupam muito espaço devem preferencialmente ter apenas uma referência armazenada na *blockchain*. O dado em si pode ser armazenado em alguma outra tecnologia que seja mais eficiente para fazer o seu armazenamento distribuído.

Em função disso, os requisitos do sistema descrito nesse trabalho são:

a) o sistema deve permitir a criação de endereços de e-mail (Requisito Funcional - RF01);
b) o sistema deve possibilitar a consulta de endereços de e-mail (RF02);
c) o sistema deve permitir que o usuário crie uma carteira para o armazenamento de criptomoedas (RF03);
d) o sistema deve permitir que o usuário transfira criptomoedas de sua carteira para outros usuários (RF04);
e) o sistema deve possibilitar que o usuário consulte o saldo de sua carteira (RF05);

f) o sistema deve permitir que o usuário consulte o histórico de todas as transações de criptomoedas envolvendo a sua carteira (RF06);
g) o sistema deve permitir que o usuário envie mensagens de e-mail, podendo enviar uma mensagem para um ou vários destinatários (RF07);
h) o sistema deve possibilitar ao usuário consultar todas as mensagens de e-mail destinadas a ele (RF08);
i) o sistema deve possibilitar que um usuário responda a um e-mail (RF09);
j) o sistema deve permitir que usuários incluídos em uma conversa de e-mail consigam visualizar todas as mensagens envolvidas na conversa (RF10);
k) o sistema deve fornecer um mecanismo de autenticação (RF11);
l) a *blockchain* deve ser desenvolvida utilizando o *framework* Cosmos-SDK (Requisito Não Funcional - RNF01);
m) o protótipo de cliente deve ser compatível com as plataformas Android, IOS, Linux, Windows e Mac OS (RNF02);
n) o corpo das mensagens de e-mail deve ser armazenado no IPFS (RNF03);
o) todos os e-mails devem ser salvos criptografados, permitindo que apenas os envolvidos tenham acesso legível aos dados (RNF04);
p) a comunicação entre o protótipo de cliente e a *blockchain* deve ser feita utilizando o protocolo gRPC (RNF05).

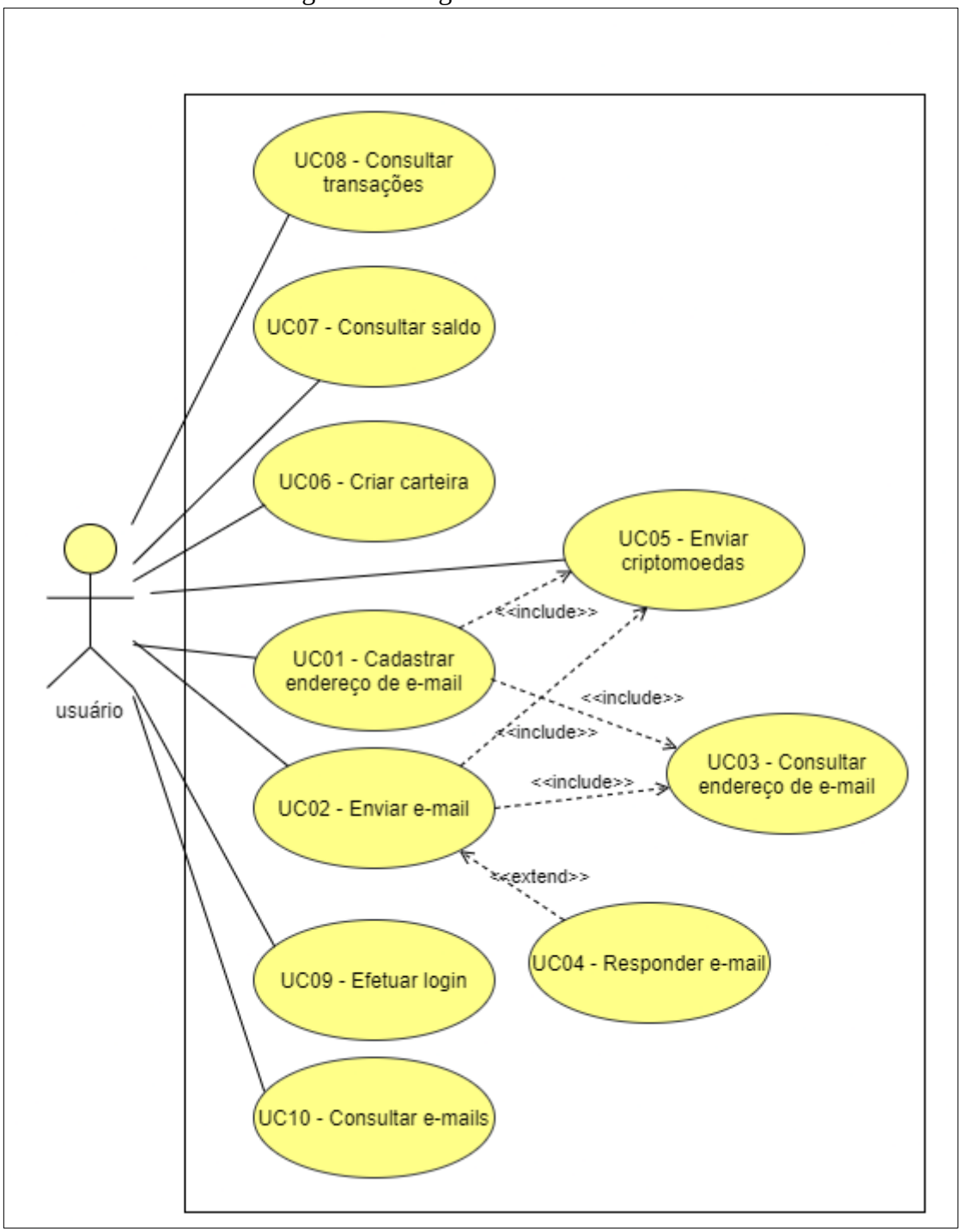
3.2 ESPECIFICAÇÃO

Nesta seção é apresentada a especificação do trabalho desenvolvido. É apresentado o diagrama de casos de uso, o diagrama entidade relacionamento, a arquitetura da aplicação e os diagramas de atividades. Todos os diagramas foram feitos através da ferramenta Draw.io.

3.2.1 Diagrama de casos de uso

Com base nos requisitos funcionais elencados na seção anterior, foi desenvolvido o diagrama de casos de uso apresentado na Figura 9.

Figura 9 – Diagrama de casos de uso



Fonte: elaborado pelo autor.

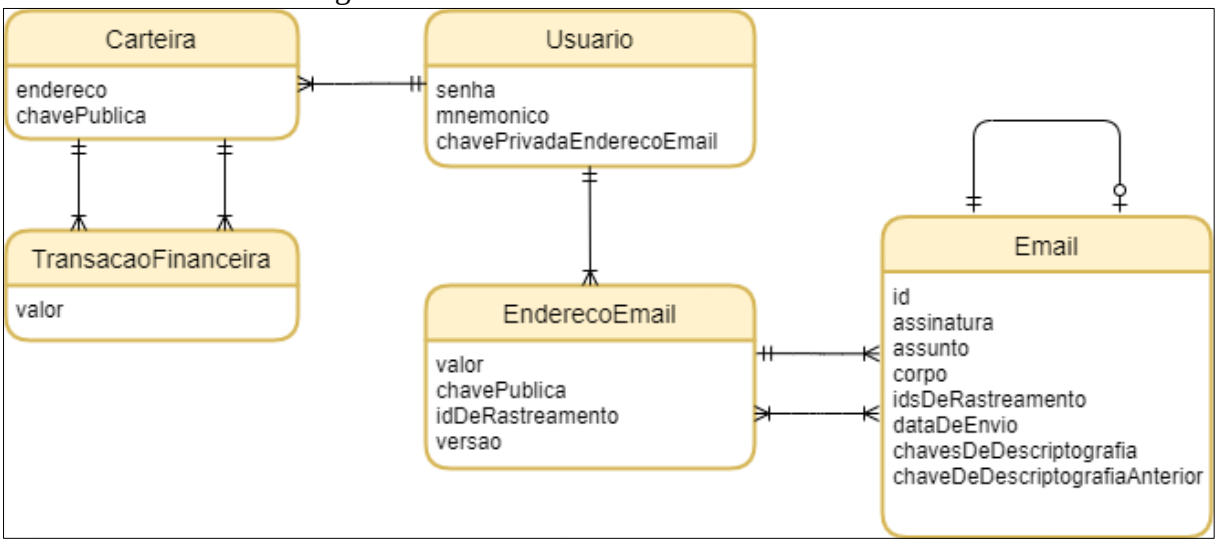
No diagrama de caso de uso apresentado na Figura 9, no caso `UC01 – Cadastrar endereço de e-mail`, o usuário cadastra o seu endereço de e-mail. O cadastro de e-mail tem relação direta com o caso `UC03 – Consultar endereço de e-mail`, pois, antes do cadastro, deve ser validado se o endereço informado não está sendo utilizado por algum outro usuário. O cadastro de e-mail também se relaciona com o caso `UC05 – Enviar criptomoedas`, pois, para o cadastro do e-mail, o usuário precisa fazer o pagamento de uma tarifa. No caso `UC02 –`

`Enviar e-mail`, o usuário faz o envio das mensagens de e-mail para outros usuários. O envio de e-mail se relaciona com o `UC05 – Enviar criptomoedas`, pois, para o envio do e-mail também é cobrada uma tarifa. No envio de e-mail, também é utilizado o `UC03 – Consultar endereço de e-mail` para recuperar informações referentes aos destinatários. O caso `UC04 – Responder e-mail` amplia o envio de e-mail para cenários em que o envio é uma resposta a algum outro e-mail. Nos casos `UC06 – Criar carteira`, `UC07 – Consultar saldo` e `UC08 – Consultar transações`, o usuário faz a gestão de sua carteira de criptomoedas que é utilizada para pagar as tarifas de uso. O caso `UC09 – Efetuar login`, o usuário faz a autenticação no sistema, ganhando acesso a dados referentes a sua conta. Por último, no caso `UC10 – Consultar e-mails`, o usuário faz a consulta dos seus e-mails.

3.2.2 Modelo Entidade Relacionamento

A aplicação desenvolvida é descentralizada e não possui a figura de uma base de dados central. Ao contrário disso, existem diferentes partes que armazenam diferentes dados referentes ao domínio. Na Figura 10 é representado de forma unificada o modelo entidade relacionamento (MER) da aplicação. O MER foi construído com o objetivo de documentar como as entidades envolvidas no sistema se relacionam.

Figura 10 – Modelo Entidade Relacionamento



Fonte: elaborado pelo autor.

A entidade `Usuario` representa os usuários que se cadastram no sistema. Ao se cadastrar, o usuário precisa criar uma carteira (entidade `Carteira`) que representa a carteira de criptomoedas de um usuário. Através do atributo `endereço` da carteira, é possível fazer o envio de criptomoedas para um usuário. As transações financeiras realizadas no sistema são representadas pela entidade `TransacaoFinanceira`. Uma transação financeira sempre estará

relacionada a duas carteiras, sendo uma carteira de origem e uma carteira de destino. A carteira também armazena uma chave pública utilizada para verificar a assinatura de dados feita por um determinado usuário no envio de dados para a *blockchain*. A chave privada referente a chave pública da carteira é gerada a partir do atributo `mnemonico` armazenado na entidade `Usuario`.

A entidade `EnderecoEmail` representa o endereço de e-mail de um usuário. Os endereços de e-mail armazenam uma chave pública de criptografia e um identificador de armazenamento. A chave pública serve para fazer a criptografia de mensagens e o identificador de rastreamento serve para rastrear os e-mails que foram enviados para o endereço. Além disso, cada endereço armazena também um identificador de versão. A versão permite que a chave pública de um e-mail seja alterada sem quebrar a compatibilidade com as chaves antigas.

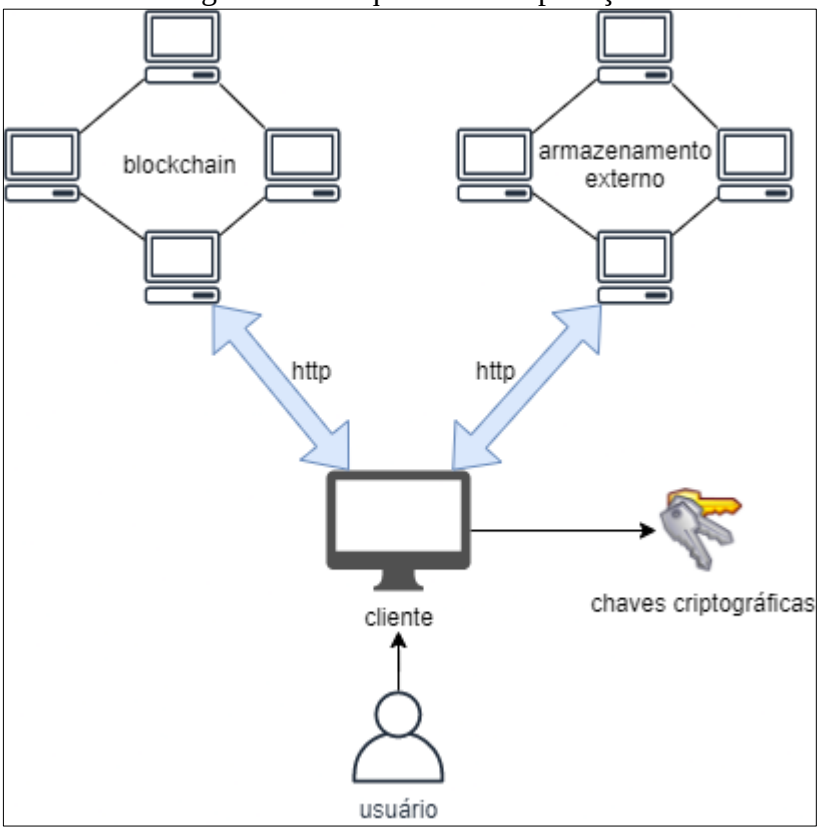
A entidade `Email` representa as mensagens de correio eletrônico que são enviadas no sistema. Ela possui dois relacionamentos com a entidade `EnderecoEmail`: um para representar o remetente e outro para representar os destinatários. A entidade também possui um auto relacionamento opcional que atende a cenários em que o envio de um e-mail é uma resposta a algum outro e-mail. O atributo `chavesDeDescriptografia` armazena a chave de descriptografia que deve ser utilizada para descriptografar o e-mail. O atributo `chaveDeDescriptografiaAnterior` armazena a chave para descriptografar o e-mail que está sendo respondido, em casos que o e-mail enviado é uma resposta a algum outro e-mail. O atributo `assinatura` armazena uma assinatura do e-mail feita pelo remetente utilizando a chave privada do seu endereço de e-mail.

O remetente, destinatário, assunto e corpo de um e-mail são salvos de modo criptografado. O único modo de ler esses dados é através da chave de descriptografia armazenada no e-mail. Como essa chave é criptografada com as chaves públicas dos destinatários, eles são os únicos que conseguem ler os dados da mensagem. O problema dessa abordagem é que, para um usuário saber que um e-mail foi enviado para ele, seria necessário que ele tentasse descriptografar todos os e-mails que são enviados no sistema. Isso acaba gerando um problema de desempenho pois, dependendo do volume de mensagens, boa parte do processamento do dispositivo do usuário seria gasto nas tentativas de descriptografia. Devido a isso, na entidade `Email`, é armazenado o atributo `idsDeRastreamento`. Esse atributo armazena os identificadores de rastreamento dos endereços de e-mail de todos os destinatários. Desse modo, um usuário precisa filtrar apenas os e-mails que possuem o seu identificador de rastreamento para identificar as mensagens que foram enviadas para ele.

3.2.3 Arquitetura da aplicação

Esta seção apresenta a arquitetura proposta para a aplicação. Conforme pode ser observado na Figura 11, a aplicação foi dividida em três partes: *blockchain*, armazenamento externo e cliente.

Figura 11 – Arquitetura da aplicação



Fonte: elaborado pelo autor.

A *blockchain* é o principal componente da aplicação. Nela serão armazenados todos os dados de controle necessários para o funcionamento de um sistema de correio eletrônico. Além disso, será salvo também na *blockchain* todos os dados referentes a carteiras e transações financeiras. Cada *peer* da rede P2P executando a *blockchain* faz a exposição de uma API HTTP que permite a comunicação dos aplicativos clientes. As APIs que servem unicamente para a consulta de dados são abertas e não necessitam de nenhum tipo de identificação por parte do cliente. Já as APIs de criação de dados necessitam que o usuário envie uma assinatura feita com a chave privada da sua carteira. A chave privada deve ficar armazenada no aplicativo cliente, sendo salva na *blockchain* apenas a chave pública da carteira.

Conforme foi observado no levantamento de requisitos, a prática de salvar todas as mensagens de e-mail diretamente na *blockchain* não é escalável. Devido a isso, foi adicionada na arquitetura uma segunda camada de armazenamento externo que serve unicamente para armazenar o corpo das mensagens de e-mail. Nessa camada, o foco está na utilização de alguma

tecnologia eficiente e robusta para armazenamento de dados. Um ponto importante dessa camada é que ela também deve ser distribuída e descentralizada, pois, não seguindo essa abordagem, a aplicação como um todo perderia as principais vantagens ganhas com a utilização de *blockchain*. Por exemplo, um serviço de armazenamento muito popular é o Amazon Simple Storage Service (S3). Com o S3, qualquer pessoa pode criar uma conta e usufruir de toda a infraestrutura dos serviços em *cloud* da Amazon. O problema de utilizar tal abordagem é que o dono da conta acaba tendo controle sobre os arquivos, podendo censurar usuários ou remover dados. Mesmo que fosse adotada uma solução de armazenar os dados em várias contas diferentes, ainda assim a aplicação estaria sujeita ao controle da própria Amazon.

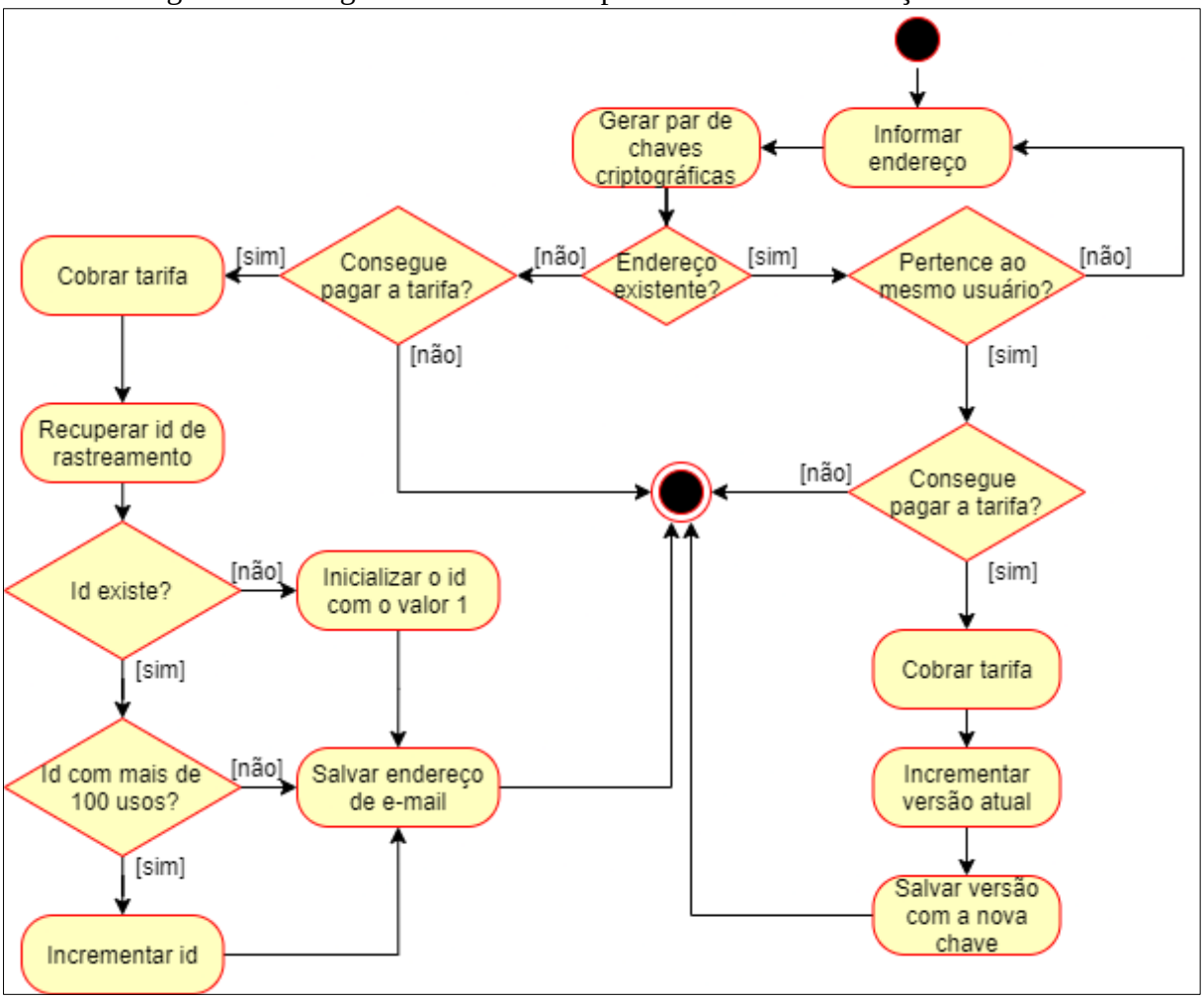
O último componente presente na arquitetura é o cliente. Ele será o ponto de entrada para que os usuários finais utilizem a aplicação. Os clientes são independentes de plataforma, podendo ser desenvolvidos para qualquer plataforma que tenha suporte ao protocolo TCP/IP. Boa parte das regras de negócio da aplicação estão presentes no cliente. Ele deve saber como criar transações assinadas para a *blockchain* e fazer a consulta de dados na mesma. Além disso, o cliente precisa se comunicar com a camada de armazenamento externo para recuperar o corpo das mensagens de e-mail. Outro ponto tratado pelo cliente é a criptografia dos dados: os clientes precisam implementar toda a lógica para criptografar os e-mails enviados e descriptografar os e-mails recebidos.

3.2.4 Diagramas de atividades

Na Figura 12 é apresentado o diagrama de atividades referente ao fluxo de cadastro de um novo endereço de e-mail. Primeiramente, o usuário informa o endereço de e-mail que deseja utilizar. Após isso, é gerado um par de chaves criptográficas, sendo a chave pública enviada para a *blockchain* junto com o endereço informado. Caso o endereço informado já exista e pertença ao mesmo usuário, o e-mail é cadastrado com uma nova versão. No cenário em que o endereço ainda não foi cadastrado, é necessário gerar o identificador de rastreamento, que é um contador que é incrementado a cada cem novos e-mails. Desse modo, existem no máximo cem endereços que utilizam o mesmo identificador de rastreamento. Essa abordagem de agrupamento foi feita para manter a anonimidade dos usuários. Mesmo que o identificador de rastreamento de um usuário seja público, não é possível identificar quais e-mails foram enviados para ele pois o e-mail pode ter sido enviado para qualquer um dos cem usuários pertencentes ao grupo. Por fim, o cadastro de endereço exige o pagamento de uma tarifa. Essa tarifa é paga através da carteira de criptomoedas do usuário e tem como função dificultar práticas maliciosos de usuários. Sem a tarifa, um único usuário poderia cadastrar diversos

endereços sem nenhum custo, fazendo com que os endereços ficassem indisponíveis para outros usuários.

Figura 12 – Diagrama de atividades para cadastro de endereço de e-mail

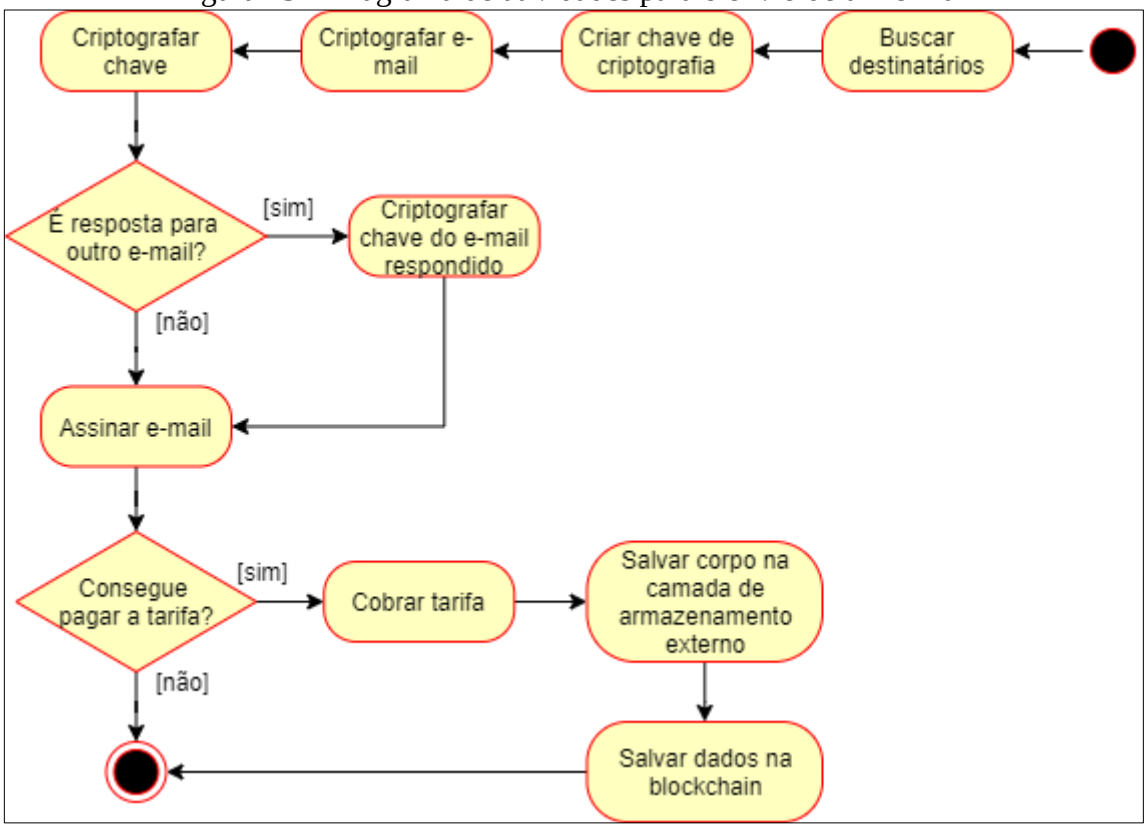


Fonte: elaborado pelo autor.

Após realizar o cadastro do endereço, o usuário pode começar a enviar e-mails. Na Figura 13 é apresentado o diagrama de atividades referente ao fluxo de envio de um e-mail. O primeiro passo é buscar as chaves públicas dos endereços de e-mail de todos os destinatários informados. Após isso, é gerada uma chave de criptografia simétrica que será utilizada para criptografar os campos privados do e-mail. Para que apenas os destinatários tenham acesso à chave de descriptografia, ela é salva também de modo criptografado. Para cada destinatário, é gerada uma versão criptografada da chave simétrica. Caso o e-mail enviado seja uma resposta a algum outro e-mail, a chave simétrica do e-mail sendo respondido também é salva na mensagem. Essa chave deve ser salva de modo criptografado utilizando a chave simétrica gerada para o e-mail que está sendo enviado. Desse modo, mesmo que um destinatário do e-mail sendo enviado não seja um destinatário do e-mail que está sendo respondido, ainda assim ele consegue visualizar o conteúdo de ambos os e-mails. Após todos os campos do e-mail serem

devidamente configurados, o remetente deve adicionar uma assinatura criada com a chave privada do seu endereço de e-mail. O objetivo dessa assinatura é possibilitar que os destinatários consigam verificar a autenticidade de um e-mail. Após a geração da assinatura, o e-mail é enviado para a *blockchain*. Na *blockchain* foi implementado o pagamento de uma tarifa para o envio de e-mails. Essa tarifa visa principalmente diminuir possíveis casos de envio de *spam*, pois, caso um usuário deseje enviar *spam*, ele terá que arcar com o custo financeiro dessa ação. Tendo o usuário criptomoedas suficientes para realizar o pagamento da tarifa, o e-mail é salvo na *blockchain*, sendo o corpo do e-mail salvo na camada de armazenamento externo.

Figura 13 – Diagrama de atividades para o envio de um e-mail



Fonte: elaborado pelo autor.

3.3 IMPLEMENTAÇÃO

A seguir são mostradas as técnicas e ferramentas utilizadas e a operacionalidade da implementação.

3.3.1 Técnicas e ferramentas utilizadas

O trabalho desenvolvido foi divido em três partes: *blockchain*, aplicativo cliente e camada de armazenamento externo. Para o desenvolvimento da *blockchain*, utilizou-se a linguagem de programação Golang na versão 1.15 em conjunto com o *framework* Cosmos-SDK na versão 0.42.1. Para a persistência dos dados na *blockchain*, utilizou-se o banco de dados

chave-valor LevelDB. Para o desenvolvimento do aplicativo cliente, utilizou-se a linguagem de programação Dart na versão 2.12 em conjunto com o *framework* Flutter na versão 2.2.1. A comunicação entre o cliente e a *blockchain* foi implementada através de uma API gRPC exposta pela *blockchain*. Como camada de armazenamento externo, utilizou-se o IPFS. O aplicativo cliente se comunica com o IPFS através de uma API HTTP.

3.3.1.1 Implementação da *blockchain*

A *blockchain* é a responsável por armazenar os dados e manter o consenso entre todos os *peers* da rede. O desenvolvimento de uma *blockchain* pode ser divido em três camadas, sendo elas a camada de rede para comunicação entre os *peers*, a camada de consenso e a camada das regras de negócio específicas da aplicação. A implementação das três camadas é um processo árduo, que exige bastante tempo e pesquisa. Como o objetivo deste trabalho não envolve a pesquisa de novos métodos de desenvolvimento de *blockchains*, optou-se por utilizar um *framework* que faça a abstração das duas primeiras camadas, possibilitando o foco nas regras específicas da aplicação.

Após uma pesquisa aprofundada, optou-se pela utilização do Tendermint. O Tendermint é um software que faz a abstração das camadas de consenso e rede da *blockchain*. Ele gerencia a criação de transações, geração dos blocos, execução de consultas e propagação de dados entre os *peers*. Para a execução das regras específicas da aplicação, o Tendermint implementa uma camada de comunicação utilizando o protocolo Application Blockchain Interface (ABCI). Desse modo, as regras da *blockchain* podem ser desenvolvidas em qualquer linguagem de programação, bastando a implementação do ABCI.

Com o Tendermint é possível configurar qual algoritmo de consenso deve ser utilizado pela *blockchain*. É possível escolher entre uma implementação de *proof of stake* e uma implementação de *proof of authority*. Como o objetivo do trabalho desenvolvido é a criação de uma *blockchain* pública, optou-se pela utilização do *proof of stake*. Apesar disso, devido a facilidade para realizar a troca entre os algoritmos, a aplicação poderia facilmente ser alterada para funcionar como uma *blockchain* privada utilizando o *proof of authority*.

Por mais que o Tendermint abstraia boa parte da complexidade no desenvolvimento de *blockchains*, ele exige que o desenvolvedor implemente toda a camada envolvendo o protocolo ABCI. Para simplificar esse processo, foi utilizado o *framework* Cosmos-SDK. O Cosmos-SDK é um *framework* desenvolvido para facilitar a implementação de *blockchains* que utilizam a linguagem de programação Golang. Utilizando o Cosmos-SDK, é possível abstrair também a

implementação do protocolo ABCI, sendo necessário apenas implementar as regras de negócio da aplicação.

Além de implementar o ABCI, o Cosmos-SDK também disponibiliza uma série de funcionalidades comuns para aplicações baseadas em *blockchains*. Para o desenvolvimento da aplicação, foi utilizada a implementação do Cosmos-SDK que faz a gestão automática de carteiras e criptomoedas. Com base nesta implementação, foi criada uma criptomoeda própria chamada demail. Essa criptomoeda é utilizada pelos usuários para pagar as tarifas necessárias para executar as ações na *blockchain*. Quando o usuário cria uma carteira, ela inicia zerada. Para adicionar demail em sua carteira, o usuário precisa receber transações de criptomoedas de usuários que possuem demail.

Para adicionar regras de negócio em uma aplicação desenvolvida com o Cosmos-SDK, é necessário dividir a aplicação em módulos. Os módulos são os diferentes domínios da aplicação. Cada módulo é configurado para tratar diferentes tipos de mensagens, podendo uma mensagem ser de escrita ou de leitura. As mensagens de escrita permitem o usuário modificar o estado da aplicação, enquanto as mensagens de leitura servem apenas para consultar o estado. O estado da aplicação é armazenado no banco de dados chave-valor LevelDB.

Para se comunicar com os módulos, o usuário precisa enviar mensagens para o Tendermint através de uma API gRPC. Para uma mensagem de escrita ser processada, ela deve ser incluída pelo usuário em uma transação. A transação precisa ser assinada e enviada para o Tendermint. Ao receber uma transação, o Tendermint faz a inclusão dela em um bloco através da execução do algoritmo de consenso. Cada transação aceita tem suas mensagens extraídas e enviadas para a aplicação através do ABCI. Já as mensagens de leitura podem ser enviadas diretamente ao Tendermint. Ao receber uma mensagem de leitura, o Tendermint faz o envio da mensagem diretamente para a aplicação através do ABCI. Fica a cargo da aplicação identificar quais módulos devem tratar cada mensagem e fazer a execução das suas regras.

Para a implementação do trabalho, foi necessária a criação de dois módulos: um para fazer a gestão dos endereções de e-mail e outro para fazer a gestão dos e-mails. As seções 3.3.1.1.1 e 3.3.1.1.2 introduzem as regras implementadas por estes módulos.

3.3.1.1.1 Módulo para gestão dos endereços de e-mail

O módulo `address` implementa as regras relacionadas à persistência dos endereços de e-mail. Ele aceita duas mensagens de escrita: adição de um novo endereço de e-mail e alteração da chave pública de um endereço existente. No Quadro 2 é apresentado parte do método que faz a tratativa da mensagem de criação de um novo endereço. Primeiramente, é validado se o

endereço já está cadastrado (linha 01). Após isso, é validado se o usuário consegue pagar a tarifa para a criação do endereço (linha 06). Se o usuário possuir criptomoedas suficientes para realizar o pagamento, o valor da tarifa é descontado de sua carteira. Por fim, é gerado o id de rastreamento (linha 10) e o endereço é persistido no banco de dados (linha 18).

Quadro 2 – Trecho de código para criação do endereço de e-mail

```
01 current := k.GetAddressByName(ctx, msg.Name)
02 if current != (types.Address{}) {
03     return nil, sdkerrors.Wrap(types.ErrAddressAlreadyExists,
04         fmt.Sprintf("address %s already exists", msg.Name))
05 }
06 err := k.sendCoinsToModule(ctx, msg.Creator, "50token")
07 if err != nil {
08     return nil, err
09 }
10 trackID := k.GenerateTrackID(ctx)
11 address := types.Address{
12     Creator: msg.Creator,
13     Name:    msg.Name,
14     PubKey:  msg.PubKey,
15     TrackID: trackID,
16     Version: 1,
17 }
18 k.SetAddress(ctx, address)
```

Fonte: elaborado pelo autor.

A tratativa da mensagem de atualização de um endereço de e-mail é muito similar ao cadastro. Conforme pode ser observado no Quadro 3, primeiramente é validado se o endereço sendo editado realmente existe (linha 01) e se o usuário tentando editar é o mesmo que criou o endereço (linha 06). Após essa etapa de validação, é feito o pagamento da tarifa (linha 10). O valor cobrado é menor na edição do que no cadastro, pois, como o usuário está editando um e-mail que já pertence a ele, a chance de ele estar tentando realizar alguma ação maliciosa é muito menor. Caso o pagamento da tarifa seja realizado com sucesso, é salvo no banco de dados uma nova versão do endereço contendo a nova chave (linha 21).

Quadro 3 – Trecho de código para atualização do endereço de e-mail

```
01 current := k.GetAddressByName(ctx, msg.Name)
02 if current == (types.Address{}) {
03     return nil, sdkerrors.Wrap(sdkerrors.ErrKeyNotFound,
04 fmt.Sprintf("address %s doesn't exist", msg.Name))
05 }
06 if msg.Creator != current.Creator {
07     return nil, sdkerrors.
08         Wrap(sdkerrors.ErrUnauthorized, "incorrect owner")
09 }
10 err := k.sendCoinsToModule(ctx, msg.Creator, "20token")
11 if err != nil {
12     return nil, err
13 }
14 address := types.Address{
15     Creator: msg.Creator,
16     Name:    current.Name,
17     PubKey:  msg.PubKey,
18     TrackID: current.TrackID,
19     Version: current.Version + 1,
20 }
21 k.SetAddress(ctx, address)
```

Fonte: elaborado pelo autor.

Além das mensagens de escrita, o módulo `address` também implementa três mensagens de leitura: buscar todos os endereços, buscar endereço pelo e-mail e buscar endereço pela versão. A busca de todos os endereços é a operação mais lenta e tende a ficar cada vez menos performática à medida que o número de endereços cadastrados aumente. A busca pelo e-mail retorna sempre a última versão cadastrada de um endereço. Essa consulta deve ser a utilizada para recuperar a chave pública de um endereço antes de fazer o envio de um e-mail. Por fim, a busca pela versão possibilita recuperar a chave de um endereço de e-mail em uma determinada versão. Essa consulta permite validar a assinatura de mensagens antigas, evitando que a alteração da chave de um endereço de e-mail torne a base inconsistente.

3.3.1.1.2 Módulo para gestão dos e-mails

O módulo `email` implementa as regras relacionadas a gestão dos e-mails. Este módulo implementa apenas uma mensagem de escrita que possibilita a criação de um e-mail. Parte do código fonte que faz a criação do e-mail pode ser observado no Quadro 4. Primeiramente é validado se o e-mail é uma resposta a algum outro e-mail (linha 01). Caso seja, é verificado no banco de dados se o e-mail sendo respondido realmente existe (linha 02). Após isso, é validado se o identificador do e-mail já não está cadastrado no banco (linha 07). O identificador é uma chave de identificação única do e-mail que deve ser gerado pelo cliente e incluído na mensagem.

Caso o identificador seja válido, é realizado o processo de pagamento da tarifa (linha 11). A tarifa foi implementada no envio de e-mail como uma ferramenta para dificultar o envio de *spam*. Se o usuário possuir saldo suficiente para pagar a tarifa, o e-mail é incluído no banco de dados (linha 30).

Quadro 4 – Trecho de código para a criação do e-mail

```
01 if msg.ReplyTo != "" {
02     if !k.HasEmail(ctx, msg.ReplyTo) {
03         return nil, sdkerrors.Wrap(sdkerrors.ErrKeyNotFound,
04        fmt.Sprintf("cannot find replied email: %s", msg.ReplyTo))
05     }
06 }
07 if k.HasEmail(ctx, msg.Id) {
08     return nil, sdkerrors.Wrap(types.ErrInvalidEmailId,
09              fmt.Sprintf("invalid id: %s", msg.Id))
10 }
11 err := k.sendCoinsToModule(ctx, msg.Creator, "20token")
12 if err != nil {
13     return nil, err
14 }
15 email := types.Email{
16     Id:                    msg.Id,
17     Creator:               msg.Creator,
18     From:                  msg.From,
19     To:                    msg.To,
20     SenderSignature:       msg.SenderSignature,
21     Subject:               msg.Subject,
22     Body:                  msg.Body,
23     ReplyTo:               msg.ReplyTo,
24     TrackIds:              msg.TrackIds,
25     SendedAt:              msg.SendedAt,
26     DecryptionKeys:        msg.DecryptionKeys,
27     PreviousDecryptionKey: msg.PreviousDecryptionKey,
28     SenderAddressVersion:  msg.SenderAddressVersion,
29 }
30 k.SetEmail(ctx, email)
```

Fonte: elaborado pelo autor.

Além da mensagem de escrita, o módulo `email` implementa duas mensagens de leitura: buscar todos os e-mails e buscar um e-mail pelo seu identificador. A busca de todos os e-mails permite ao usuário consultar todos os e-mails que estão armazenados na *blockchain*. Essa busca tende a se tornar menos performática à medida que o número de e-mails cadastrados aumente. A busca pelo identificador possibilita realizar a consulta de um e-mail de modo mais performático. Ela exige que seja informado qual o identificador único do e-mail que se deseja consultar.

3.3.1.2 Implementação do protótipo de cliente

O cliente é o software utilizado pelo usuário final da aplicação. A ideia de separar a implementação do cliente da implementação da *blockchain* é possibilitar o desenvolvimento de vários clientes diferentes, mantendo como núcleo a *blockchain*. Visando atender o maior número de dispositivos possível, o protótipo de cliente desenvolvido neste trabalho foi implementado utilizando o *framework* Flutter. Desse modo, é possível executar o cliente em várias plataformas diferentes como Linux, Windows, MacOS, IOS, Web e Android.

O cliente precisa se comunicar com a *blockchain* através da API gRPC exposta pelo Tendermint. Para utilizar essa API, foi utilizada a biblioteca alan na versão 0.40.2. Essa biblioteca abstrai o processo de comunicação com a API gRPC, facilitando a criação e consulta de transações. Além da comunicação com a *blockchain*, o cliente desenvolvido implementa facilitadores para os usuários. Alguns desses facilitadores são permitir o login com endereço de e-mail e senha e realizar a geração de novas carteiras.

Cada usuário na *blockchain* possui uma carteira. A carteira é atrelada a um par de chaves criptográficas. A chave privada fica em posse do usuário e é utilizada para fazer a assinatura das transações. Já a chave pública é armazenada na *blockchain* e permite que qualquer um valide a assinatura feita por um determinado usuário. Essa verificação é feita automaticamente pelo Tendermint e serve para validar se uma transação é autêntica. O método que faz a geração da carteira pode ser observado no Quadro 5. A geração da carteira é feita com base em um mnemônico. O mnemônico é uma frase com o qual é possível derivar um par de chaves criptográficas. Desse modo, ao invés de armazenar as chaves, basta que o usuário armazene o mnemônico para ter acesso a sua carteira.

Quadro 5 – Método para geração da carteira

```
01 Future<Wallet> generateNewWallet() async {
02     var mnemonic = alanLib.Bip39.generateMnemonic(strength: 256);
03     final wallet = alanLib.Wallet.
04             derive(mnemonic, this.demailNetworkInfo);
05     return Wallet(wallet.bech32Address, mnemonic.join(' '));
06 }
```

Fonte: elaborado pelo autor.

Após conseguir gerar a sua carteira, o usuário precisa cadastrar um endereço de e-mail. Para o cadastro do endereço, é necessário enviar para a *blockchain* uma transação com a mensagem de criação de endereço. O Quadro 6 apresenta o método que executa esse processo. Com base no mnemônico do usuário, é gerada a carteira (linha 03). Com a carteira, é possível assinar a transação e enviar ela para o Tendermint. Após fazer o envio da transação (linha 13),

o método aguarda a execução do algoritmo de consenso na *blockchain*. Tendo a transação sido incluída em um bloco, é retornado para o cliente o id de rastreamento gerado para o endereço cadastrado (linha 15). Esse id de rastreamento é armazenado no cliente para facilitar a identificação de e-mails enviados para o usuário.

Quadro 6 – Método do cliente para criação do endereço de e-mail

```
01 Future<Address> createAddress(
02       String mnemonic, String name, String pubKey) async {
03     Wallet wallet = generateWallet(mnemonic);
04
05     final message = MsgCreateAddress.create()
06       ..creator = wallet.bech32Address
07       ..name = name
08       ..pubKey = pubKey;
09
10     int trackID = 1;
11
12     try {
13       final response = await sendTransaction(wallet, message, true);
14       if (response.isSuccessful) {
15         trackID = getTrackIDFromCreateResponse(response);
16       } else {
17         throw Exception("Cannot generate address");
18       }
19     } catch (e) {
20       throw e;
21     }
22     if (trackID == -1) {
23       throw Exception("Cannot find trackID");
24     }
25     return Address(name, pubKey, trackID, 1);
26 }
```

Fonte: elaborado pelo autor.

Conforme pode ser observado no Quadro 6, o método que cria o endereço de e-mail recebe como parâmetro uma variável chamada `pubKey`. Essa variável armazena a chave pública pertencente a um par de chaves criptográficas geradas para a utilização do algoritmo RSA. A chave pública é salva na blockchain junto com o endereço de e-mail. Já a chave privada é armazenada no aplicativo cliente. O algoritmo RSA é aplicado sobre o e-mail para permitir que apenas usuários válidos tenham acesso ao conteúdo das mensagens.

Após o usuário criar o seu endereço de e-mail, ele consegue fazer o envio de e-mails para outros usuários. O primeiro passo para enviar um e-mail é aplicar o algoritmo de criptografia sobre o seu conteúdo. Na primeira linha do Quadro 7, é feita a geração de uma chave simétrica que possibilita a utilização do algoritmo AES. Após isso, é feita uma consulta

na *blockchain* para recuperar os endereços de e-mail de todos os destinatários (linha 04). Para cada destinatário, é adicionado um valor na lista `decryptionKeys` (linha 12). O valor adicionado representa a chave simétrica criptografada com a chave pública do destinatário através do algoritmo RSA. Após a geração da chave simétrica, os demais campos do e-mail são criptografados utilizando o algoritmo AES.

Quadro 7 – Criptografia dos campos do e-mail

```
01 AESImpl aesImpl = AESImpl.generateKey();
02 List<String> trackIDs = [user.trackID.toString()];
03 List<String> decryptionKeys = [];
04 List<Address> addresses = await this.addressProvider.getAddresses(to);
05
06 addresses.forEach((address) {
07   if (!trackIDs.contains(address.trackID.toString())) {
08     trackIDs.add(address.trackID.toString());
09   }
10   RSAImpl rsaImpl = RSAImpl();
11   rsaImpl.setKeyPair(address.pubKey, null);
12   decryptionKeys.add(encryptAesKeyWithRsa(aesImpl, rsaImpl));
13 });
14
15 RSAImpl senderRsaImpl = RSAImpl();
16 senderRsaImpl.setKeyPair(user.rsaPublicKey, user.rsaPrivateKey);
17 decryptionKeys.add(encryptAesKeyWithRsa(aesImpl, senderRsaImpl));
18 DateTime now = DateTime.now();
19 String sendedAtStr = DateUtils.getCurrentISOTimeString(dateTime: now);
20 String bodyIpfsCid = await saveBodyInIpfs(aesImpl, body);
21 String encryptedFrom = aesImpl.encrypt(user.email);
22 String encryptedSubject = aesImpl.encrypt(subject);
23 String signature = generateSignature(
24         senderRsaImpl, user.email, bodyIpfsCid, sendedAtStr,
25         to, subject);
26 String toStr = aesImpl.encrypt(to.join(";"));
```

Fonte: elaborado pelo autor.

Antes de enviar o e-mail para a *blockchain*, é necessário armazenar o corpo da mensagem na camada de armazenamento externo. A tecnologia escolhida para armazenar os dados fora da *blockchain* foi o IPFS. No Quadro 8 é apresentado o método que salva o corpo do e-mail no IPFS. Primeiramente, é criptografado o corpo do e-mail com a chave simétrica utilizando o algoritmo AES (linha 02). Após isso, é feito o *encoding* do valor criptografado e enviado o resultado para o IPFS (linha 03). Ao receber o valor, o IPFS retorna o identificador da mensagem (*hash*). Esse identificador é enviado junto com a mensagem de criação de e-mail para a *blockchain* para possibilitar que o corpo do e-mail seja recuperado pelos destinatários.

Quadro 8 – Método para salvar o corpo do e-mail no IPFS

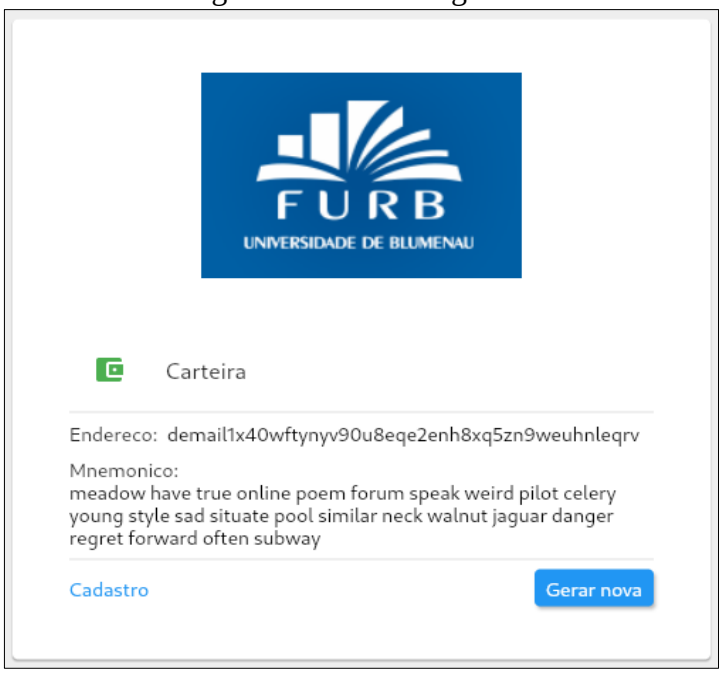
```
01 Future<String> saveBodyInIpfs(AESImpl aesImpl, String body) async {
02     var encryptedBody = aesImpl.encrypt(body);
03     var ipfsAddBodyResponse = await this.ipfs.add(utf8.
04                 encode(encryptedBody));
05     if (!ipfsAddBodyResponse.isSuccessful) {
06       throw Exception("Cannot save body in ipfs");
07     }
08     return ipfsAddBodyResponse.body!.hash!;
09 }
```

Fonte: elaborado pelo autor.

3.3.2 Operacionalidade da implementação

Ao iniciar o aplicativo cliente é requisitado ao usuário que ele informe o seu endereço de e-mail e sua senha. Caso seja o primeiro acesso do usuário, ele precisa criar uma conta. A criação da conta exige que o usuário possua uma carteira com criptomoedas. A Figura 14 mostra a tela que faz a geração automática de uma carteira para o usuário.

Figura 14 – Carteira gerada



Fonte: elaborado pelo autor.

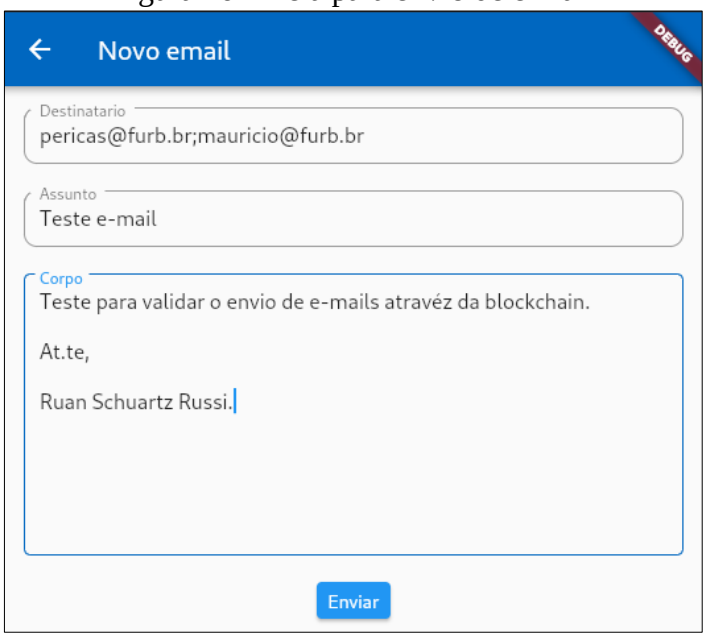
Na tela apresentada na Figura 14, o usuário tem acesso ao endereço da sua carteira. Esse endereço deve ser compartilhado com outros usuários para que seja realizada alguma transação de criptomoedas para a carteira gerada. Após a realização de alguma transação, o usuário pode utilizar a carteira gerada para realizar o seu cadastro. A Figura 15 mostra a tela de cadastro. Nesta tela, o usuário deve informar o seu endereço de e-mail, o mnemônico da sua carteira e a senha que deseja utilizar para fazer o login.

Figura 15 – Tela de cadastro

Fonte: elaborado pelo autor.

Após criar a conta, o usuário pode entrar no sistema e fazer o envio de e-mails para outros usuários. Para enviar um e-mail, o usuário deve informar a lista de destinatários, o assunto e o corpo do e-mail. A Figura 16 apresenta a tela de envio de e-mail. Conforme pode ser observado, caso o e-mail possua mais de um destinatário, os endereços de e-mail dos destinatários devem ser separados por ponto e vírgula.

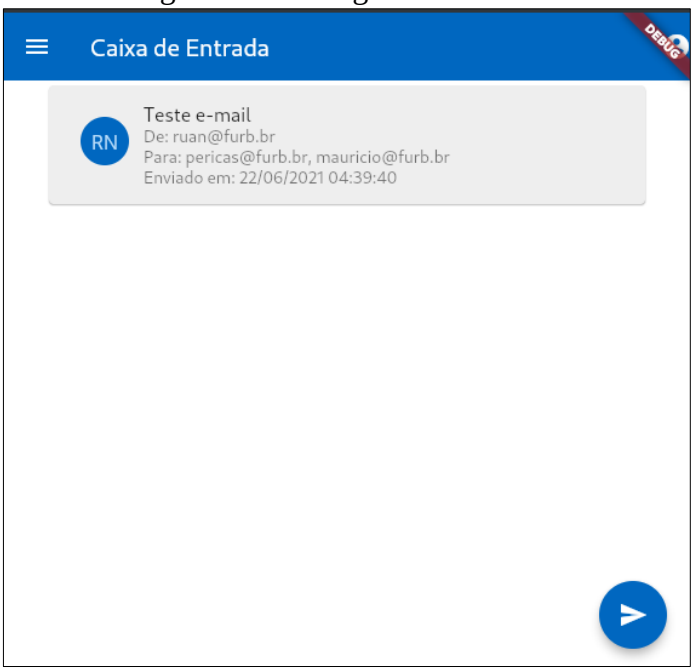
Figura 16 – Tela para envio de e-mail



Fonte: elaborado pelo autor.

Os e-mails enviados para o usuário autenticado são apresentados em uma listagem. Essa listagem apresenta apenas os campos básicos do e-mail. Na Figura 17, é possível observar como o e-mail enviado na tela da Figura 16 é apresentado para o usuário.

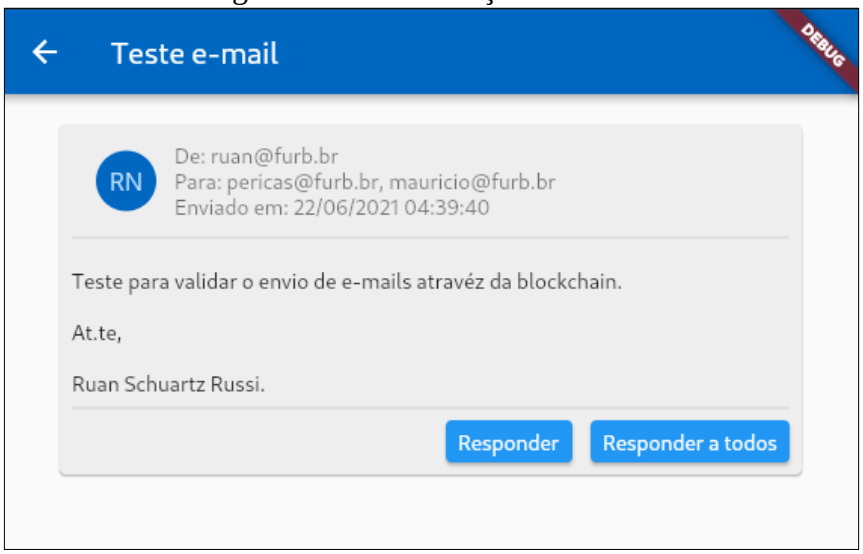
Figura 17 – Listagem dos e-mails



Fonte: elaborado pelo autor.

Na tela apresentada na Figura 17, o usuário pode selecionar, através de um clique, os e-mails que deseja abrir. A seleção do e-mail abre a tela apresentada na Figura 18. Nesta tela, o usuário consegue visualizar o corpo do e-mail, além de ter opções para enviar um e-mail em resposta.

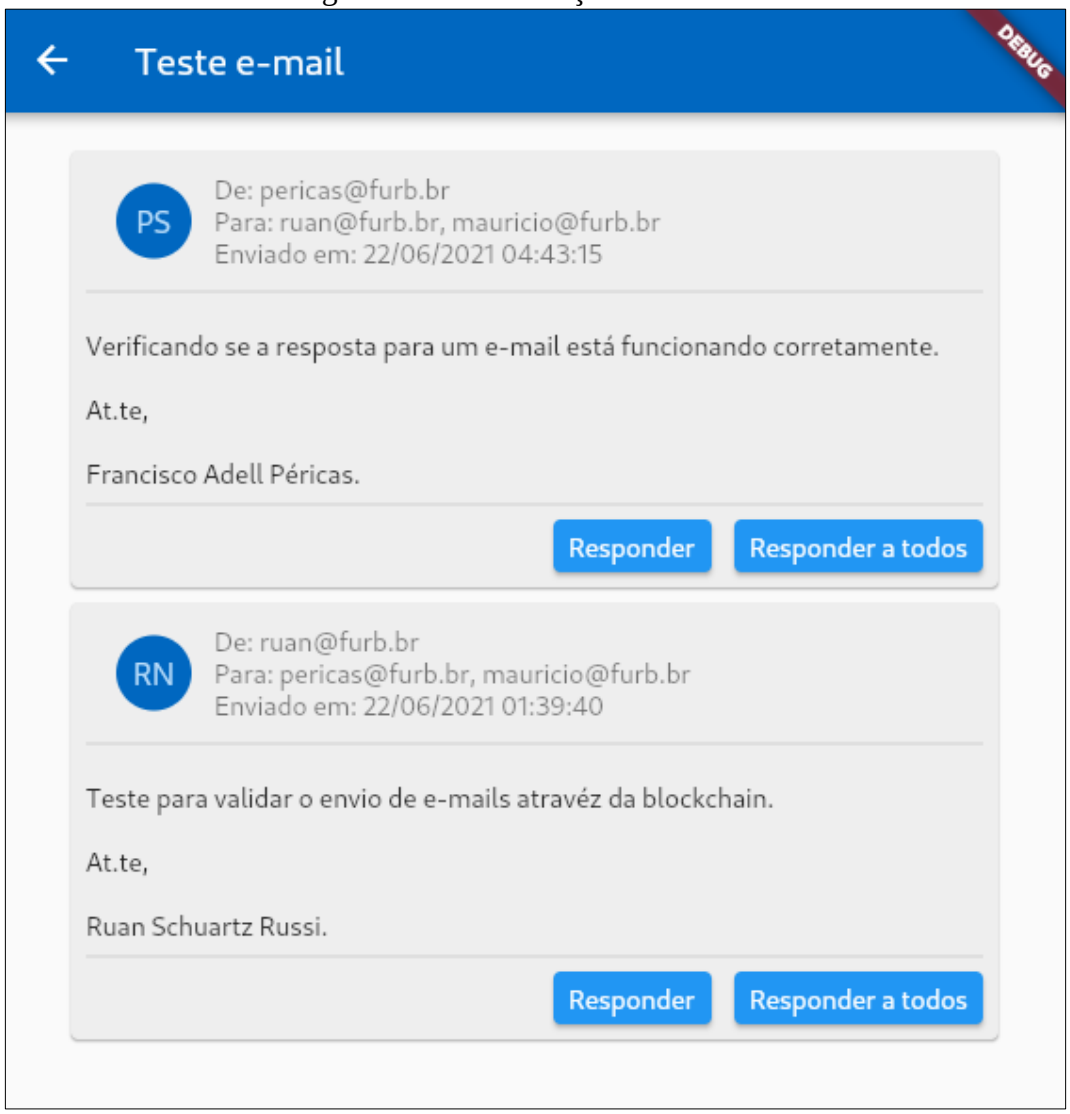
Figura 18 – Visualização do e-mail



Fonte: elaborado pelo autor.

Caso o usuário responda a um e-mail, é criada uma conversa. As conversas são listas encadeadas de vários e-mails. Se o usuário abrir um e-mail que pertence a uma conversa, são mostrados, além do e-mail selecionado, todos os e-mails da conversa anteriores a ele. Esse comportamento pode ser observado na Figura 19.

Figura 19 – Visualização de conversa



Fonte: elaborado pelo autor.

3.4 ANÁLISE DOS RESULTADOS

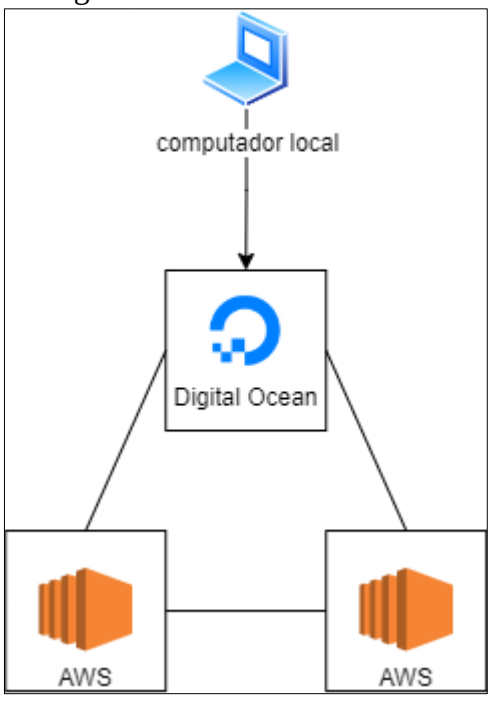
Nesta seção são apresentados os resultados obtidos através dos experimentos realizados com a aplicação desenvolvida. A seção 3.4.1 apresenta o ambiente utilizado para a realização dos testes. Na seção 3.4.2 é feita a análise de desempenho da aplicação, sendo o foco a análise do desempenho da *blockchain*. Na seção 3.4.3 é feita a análise da capacidade de armazenamento necessária para executar a *blockchain*. Na seção 3.4.4, é feita a análise do sistema financeiro da

aplicação. Por fim, a seção 3.4.5 apresenta um comparativo entre o sistema desenvolvido e os trabalhos correlatos.

3.4.1 Ambiente de testes

O sistema desenvolvido, por se tratar de uma aplicação P2P, pode ser executado de diversas formas diferentes. Essa característica dificulta a construção de um ambiente de testes que seja igual ao ambiente utilizado pelos usuários finais. A abordagem mais simples para realizar os testes é executar a *blockchain* apenas em um computador local. Com essa abordagem, é possível validar todas as regras de negócio da aplicação, porém, não é possível testar os pontos que envolvem a comunicação entre os *peers*. Para testar a comunicação entre os *peers*, é necessário criar um ambiente que possua mais de um computador executando a *blockchain.* A Figura 20 apresenta o ambiente utilizado para realizar os testes da aplicação proposta.

Figura 20 – Ambiente de testes



Fonte: elaborado pelo autor.

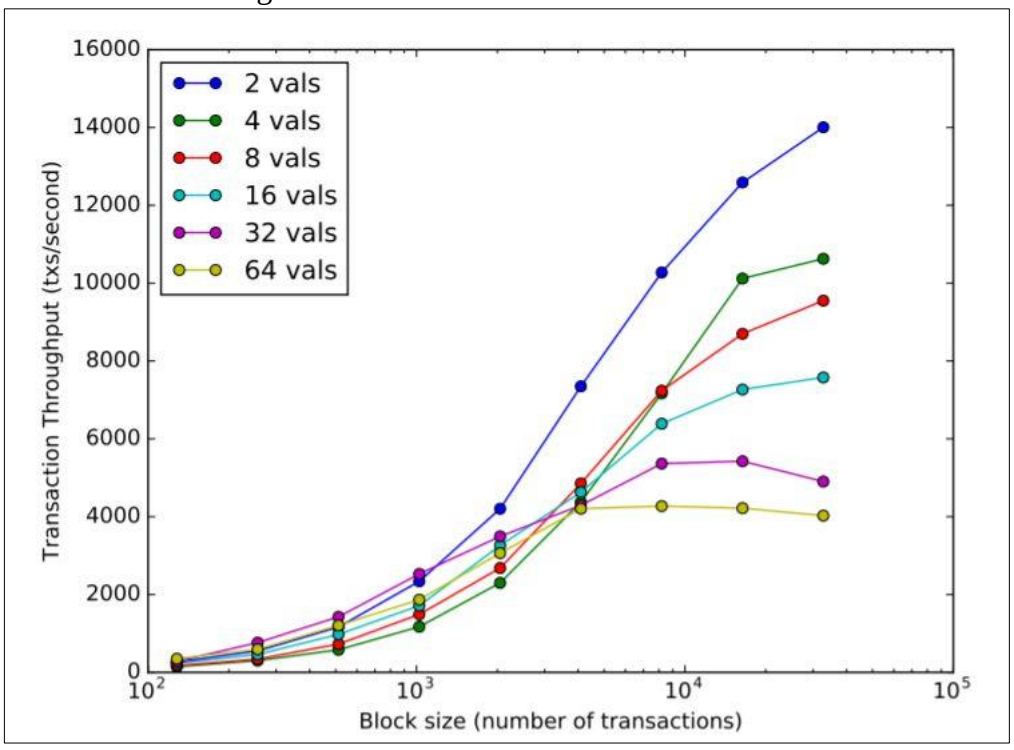
No ambiente apresentado pela Figura 20, a *blockchain* é executada por três *peers* diferentes: um servidor Linux executando na *cloud* da Digital Ocean e dois servidores Linux executando na *cloud* da Amazon. Todos os servidores foram configurados com um núcleo de CPU e 2GB de memória RAM. Tanto o aplicativo cliente quanto o serviço que se comunica com o IPFS foram executados em um computador local. A comunicação entre a *blockchain* e o aplicativo cliente foi feita através do servidor configurado na Digital Ocean. Por mais que o

ambiente criado não permita o teste com um grande número de *peers*, ele possibilita fazer a validação de todos os pontos da aplicação.

3.4.2 Análise de desempenho

A análise de desempenho da *blockchain* foi feita com base na quantidade de transações por segundo (TPS) que a *blockchain* consegue processar. O TPS é diretamente relacionado com o algoritmo de consenso escolhido. Como no trabalho desenvolvido foi utilizado o *proof of stake*, o TPS é medido com base na velocidade que os validadores conseguem entrar em consenso e fazer a criação de um bloco. Quanto mais validadores existirem, maior é o tempo necessário para eles entrarem em consenso e consequentemente menor é o TPS. A implementação de *proof os stake* utilizada no sistema desenvolvido é uma abstração fornecida pelo Tendermint. Com base no algoritmo de *proof of stake* do Tendermint, a Figura 21 mostra a relação entre o número de validadores, a quantidade de transações por bloco e o TPS.

Figura 21 – Análise do TPS do Tendermint



Fonte: Buchman (2016).

Conforme pode ser observado na Figura 21, utilizando 2 validadores e blocos com mais de 10^4 transações, é possível atingir um TPS de 14000. Apesar do alto desempenho, a utilização de poucos validadores é uma má prática, pois torna o sistema mais centralizado e suscetível a ataques. Desse modo, o ideal seria utilizar a maior quantidade de validadores possível. Como a *blockchain* implementada é pública e aberta para qualquer usuário se tornar um validador, à medida que o sistema for se popularizando, o TPS deve ficar menor que 4000. Isso implica que,

caso existam mais de 64 validadores, o sistema é capaz de enviar menos que 4000 e-mails por segundo.

Outra métrica de desempenho importante é o tempo que um e-mail leva para ser enviado ao destinatário. Nos testes realizados, uma transação de criação de e-mail demorou em torno de 2 segundos para ser incluída em um bloco e propagada entre todos os *peers*. Esse tempo pode ser maior dependendo do ambiente de execução. Fatores que podem influenciar no tempo de envio de um e-mail são a largura de banda dos *peers*, a localização física dos *peers* e o tráfego da rede.

3.4.3 Análise da capacidade de armazenamento

Na implementação do sistema desenvolvido, optou-se por armazenar o corpo das mensagens de e-mail no IPFS. Apesar disso, boa parte dos dados, incluindo informações de controle, são armazenados diretamente na *blockchain*. A *blockchain* não é uma tecnologia otimizada para fazer o armazenamento de dados, pois exige que os dados sejam replicados entre todos os *peers* da rede. Desse modo, para executar o sistema, um *peer* precisa ser capaz de armazenar todas as transações enviadas para a *blockchain*.

Nos testes realizados, um bloco vazio, que possui apenas informações de controle, utilizou 800 bytes para ser armazenado. Nos blocos são adicionadas as transações. A transação de criação do endereço de e-mail teve o tamanho médio de 288 bytes. Como cada usuário do sistema precisa de um endereço de e-mail, é possível afirmar que no mínimo existira uma transação de criação de endereço para cada usuário do sistema. A Tabela 1 apresenta o custo necessário para armazenar os endereços. A análise foi feita com base em blocos que armazenam 6000 transações.

Tabela 1 – Total de armazenamento necessário para os endereços de e-mail

Quantidade de usuários	Armazenamento necessário
1000	288,8 KB
10000	2,8816 MB
100000	28,8136 MB
1000000	288,1336 MB
1000000000	288,1333 GB

Fonte: elaborado pelo autor.

Durante os testes, a transação de envio de um e-mail teve o tamanho médio de 2031 bytes. A Tabela 2 apresenta o custo necessário para armazenar os e-mails. A análise foi feita com base em blocos que armazenam 6000 transações.

Tabela 2 – Total de armazenamento necessário para os e-mails

Quantidade de e-mails	Armazenamento necessário
1000	2,0318 MB
10000	20,3116 MB
100000	203,1136 MB
1000000	1,8916 GB
1000000000	1,8473 TB

Fonte: elaborado pelo autor.

Com base nos dados apresentados na Tabela 1 e na Tabela 2, é possível verificar como o sistema se comportaria caso fosse utilizado como substituto das atuais implementações de correio eletrônico. Atualmente, existem mais de 4 bilhões de usuários de e-mail (RADICATI, 2020). O custo com armazenamento necessário para armazenar os endereços desses usuários na *blockchain* seria em torno de 1 TB. Já referente ao envio de e-mails, atualmente são enviados em média 306 bilhões de e-mails diariamente (RADICATI, 2020). Para armazenar todos os e-mails enviados no mundo durante 10 dias na *blockchain*, seriam necessários 5541 TB.

3.4.4 Análise do sistema financeiro

A aplicação desenvolvida possui um sistema financeiro próprio baseado em criptomoedas. Esse sistema foi implementado como um modo de incentivar que os *peers* compartilhem os seus recursos computacionais com os outros usuários da rede. Sem a presença desse incentivo, a rede P2P se tornaria menos performática e mais suscetível a ataques, pois os *peers* não teriam nenhum motivo para manter ela funcionando.

As criptomoedas podem ser utilizadas como incentivo através da sua venda para usuários que desejam utilizar o sistema. Os *peers* podem acumular as criptomoedas que recebem para manter o sistema funcionando e, futuramente, trocar elas por outras moedas como real, dólar, Bitcoin etc. Na implementação disponibilizada, não existe um mecanismo de conversão entre moedas. Esta funcionalidade, conforme a popularização do sistema, deverá ser disponibilizada por alguma *exchange*.

Além de serem utilizadas como incentivo, as criptomoedas também servem para dificultar a prática de ações maliciosas. Um exemplo é o envio de *spam*. Com a existência de um custo para fazer o envio de um e-mail, caso alguém deseje enviar e-mails em massa, será necessário arcar com os custos dessa ação.

Uma possível desvantagem da solução proposta com relação as atuais implementações de correio eletrônico é que geralmente os usuários não pagam para utilizar sistemas de e-mail. Soluções como Gmail e Outlook são gratuitas e amplamente utilizadas. Apesar da gratuidade,

é importante entender o que possibilita as empresas disponibilizarem tais serviços de modo gratuito. Na maioria dos casos, as empresas fornecem os serviços gratuitos como forma de capturar dados dos usuários e gerar receita através de anúncios personalizados. Neste caso, mesmo que não exista um custo financeiro, o usuário acaba pagando com a quebra da sua privacidade.

3.4.5 Comparativo com os trabalhos correlatos

O Quadro 9 mostra um comparativo entre a aplicação desenvolvida e os trabalhos correlatos.

Quadro 9 – Comparativo dos trabalhos correlatos e a aplicação desenvolvida

Características/ Trabalhos correlatos	Post	Bitmessage	Sistema de e-mail P2P baseado em *polling*	Aplicação desenvolvida
Arquitetura descentralizada	X	X	X	X
Controle de *spam*	X	X	X	X
Sistema de criptomoedas				X
API para facilitar a implementação de clientes				X
Mensagens imutáveis	X	X	X	X
Compatível com clientes de e-mail tradicionais	X		X	
Histórico de mensagens	X		X	X
Armazenamento em *blockchain*				X

Fonte: elaborado pelo autor.

Por meio das informações do Quadro 9 é possível notar que os trabalhos compartilham várias características. Isso ocorre, pois eles foram desenvolvidos com base nos mesmos objetivos finais, sendo as diferenças entre eles relacionadas apenas as tecnologias utilizadas. O trabalho desenvolvido se diferencia por incentivar os *peers* através de um sistema de criptomoedas. Nos outros trabalhos, o interesse dos *peers* é meramente em utilizar a aplicação, o que acaba não sendo um incentivo tão bom.

Outro ponto diferencial da aplicação desenvolvida é a implementação da API de comunicação. Nos outros trabalhos, para desenvolver um aplicativo cliente, é necessário entender detalhes sobre os diferentes protocolos que são utilizados. Além disso, é necessário

ter um entendimento aprofundado da aplicação como um todo. Já no trabalho desenvolvido, é disponibilizada uma API em um formato amplamente utilizado no mercado (gRPC). Com essa API, é possível desenvolver aplicativos clientes em diversas linguagens e plataformas sem a necessidade de entender como o núcleo da aplicação foi implementado.

Por fim, a aplicação desenvolvida se diferencia por utilizar como núcleo uma implementação baseada em *blockchain*. Essa característica traz uma série de vantagens, sendo grande parte delas relacionada às possibilidades de extensão do sistema. Outra vantagem do uso de *blockchain* é a criação de um sistema mais seguro e tolerante a falhas. Os trabalhos correlatos apresentados, pela forma como foram desenvolvidos, são mais suscetíveis a certos tipos de ataques.

4 CONCLUSÕES

Este trabalho apresentou o desenvolvimento de uma aplicação de correio eletrônico baseada em *blockchain*. Os objetivos específicos para o trabalho foram atingidos, sendo eles o desenvolvimento de uma API para a comunicação com a *blockchain* e o desenvolvimento de um protótipo de cliente para validar a *blockchain*. A API disponibilizada foi desenvolvida utilizando o formato gRPC e possibilita a implementação de aplicativos cliente sem a necessidade de entender como a *blockchain* funciona. Já o protótipo de cliente foi desenvolvido com base na API gRPC e serve como exemplo para a implementação de outros clientes.

Para o desenvolvimento da *blockchain*, optou-se pela utilização do *framework* Tendermint. O Tendermint foi escolhido através de uma pesquisa que envolveu diferentes métodos de desenvolvimento de *blockchains*. O primeiro método pesquisado foi o desenvolvimento da *blockchain* do zero, sem a utilização de nenhum *framework* ou plataforma. Esse método, por mais que traga benefícios como maior controle sobre a implementação, mostrou ser muito custoso. Tendo em mente que o objetivo do trabalho desenvolvido não era a pesquisa de novos métodos de desenvolvimento de *blockchains*, essa abordagem foi abandonada.

A segunda abordagem estudada foi a utilização da plataforma Ethereum. Com o Ethereum, é possível desenvolver aplicações em uma linguagem própria da plataforma e executar elas em uma *blockchain* compartilhada com várias outras aplicações. Essa abordagem traz uma enorme facilidade, pois permite o foco apenas nas lógicas da aplicação, sendo o controle e implementação da *blockchain* responsabilidade do Ethereum. Apesar disso, existem algumas desvantagens. A primeira delas é referente ao desempenho. Como existe uma única *blockchain* que executa diversas aplicações, o desempenho acaba sendo menor, pois a capacidade de processamento da *blockchain* é dividida entre todas as aplicações. Outra desvantagem é a falta de soberania por parte das aplicações. Como a *blockchain* é compartilhada, seu fluxo de desenvolvimento é feito de modo independente das aplicações. Sendo assim, caso uma aplicação precise de alguma otimização por parte da *blockchain*, ela depende do time do Ethereum para aceitar e implementar a otimização. Devido a essas desvantagens, decidiu-se não utilizar o Ethereum.

Como terceira abordagem, foi feita uma pesquisa sobre algum *framework* que facilite o desenvolvimento de aplicações baseadas em *blockchain* da mesma forma que o Ethereum facilita, porém, que permita a soberania das aplicações. Como resultado desta pesquisa, foi encontrado o Tendermint. Com o Tendermint, a aplicação desenvolvida executa em uma

blockchain própria, sem compartilhar recursos com outras aplicações. A única desvantagem do Tendermint em relação ao Ethereum é a necessidade de fazer a gestão manual da *blockchain*. Para facilitar o uso do Tendermint, utilizou-se o *framework* Cosmos-SDK. O Cosmos-SDK foi extremamente útil, pois forneceu uma série de padrões e abstrações que facilitaram o desenvolvimento da *blockchain.*

Além da *blockchain*, o trabalho desenvolvido possui dois outros componentes, sendo eles o protótipo de cliente e a camada de armazenamento externo. Para o desenvolvimento do protótipo de cliente, foi utilizado o *framework* Flutter. O Flutter trouxe uma série de benefícios, sendo o maior deles a capacidade de executar a aplicação em diversas plataformas diferentes como Linux, Windows, MacOS, Android, IOS e WEB. Já para a camada de armazenamento externo, foi utilizado o IPFS. O IPFS foi escolhido com base em dois critérios, sendo eles a descentralização da solução e a otimização para armazenamento distribuído de dados. A configuração do IPFS foi extremamente simples, e seu uso possibilitou diminuir a quantidade de dados armazenados na *blockchain*.

Após a análise dos resultados gerados pelos testes, é possível afirmar que aplicações baseadas em *blockchain* possuem capacidade para substituir as atuais implementações de correio eletrônico. Elas conseguem entregar maior privacidade, segurança e descentralização para os usuários. Apesar disso, existem alguns pontos de melhoria. Com os testes, foi identificado que o armazenamento dos e-mails na *blockchain* exige um grande gasto com armazenamento. Essa característica pode acabar centralizando o sistema, pois apenas as grandes corporações teriam os recursos necessários para executar um *peer*. Além das melhorias relacionadas ao armazenamento, são necessárias também melhorias relacionadas ao desempenho. Através de uma análise de desempenho do Tendermint, foi identificado que, com um número grande de usuários, a aplicação seria capaz de enviar menos de 4000 e-mails por segundo. Esse valor não seria capaz de atender a demanda processada pelos atuais sistemas de correio eletrônico.

De modo geral, este trabalho torna-se relevante pois tem potencial para impactar diretamente os bilhões de usuários de sistemas de correio eletrônico. A solução implementada, além de utilizar tecnologias mais modernas, entrega para os usuários um sistema com maior foco em privacidade e descentralização. Quanto à contribuição científica, este trabalho apresenta uma introdução sobre os principais temas relacionados ao desenvolvimento de aplicações baseadas em *blockchain*. Com isso, espera-se que este trabalho possa ser utilizado como referência bibliográfica por futuras pesquisas relacionadas a *blockchain*.

4.1 EXTENSÕES

Para trabalhos futuros são sugeridos:

a) aumentar o desempenho da *blockchain* através de escalabilidade horizontal utilizando o IBC;
b) possibilitar o pagamento das tarifas com criptomoedas implementadas em outras *blockchains*;
c) implementar uma camada de compatibilidade entre a *blockchain* e o protocolo SMTP com o objetivo de facilitar a migração para o sistema desenvolvido;
d) adicionar suporte para o envio de anexos.

REFERÊNCIAS

BENET, Juan. **IPFS:** Content Addressed, Versioned, P2P File System (DRAFT 3). [S.l.], 2019. Disponível em: https://github.com/ipfs/papers/raw/master/ipfs-cap2pfs/ipfs-p2p-file-system.pdf. Acesso em: 17 mai. 2021.

BLOOM, Zack. **The History of Email**. [S.l.], 2017. Disponível em: https://blog.cloudflare.com/the-history-of-email/. Acesso em: 02 abr. 2021.

BUCHMAN, Ethan. **Tendermint**: Byzantine Fault Tolerance in the Age of Blockchains. 2016. 98 f. Dissertação (Mestrado em Sistemas de Engenharia e Computação) – Escola de Engenharia, Universidade de Guelph, Guelph.

BUFORD, John F.; YU, Heather; LUA, Eng K. **P2P**: Networking and Applications. San Francisco: Morgan Kaufmann, 2009.

CROSBY, Michael *et al.* BlockChain Technology: Beyond Bitcoin. **Applied Innovation Review**, [S.I.], n. 2, p. 33-50, 2016. Disponível em: https://scet.berkeley.edu/wp-content/uploads/AIR-2016-Final-version-Int.pdf. Acesso em: 03 abr. 2021.

CURRAN, Brian. **What is Proof of Authority Consensus? Staking Your Identity on The Blockchain**. [S.l.], 2018. Disponível em: https://blockonomi.com/proof-of-authority/. Acesso em: 10 mai. 2021.

FROST, Liam. **Bitcoin Blockchain Grows to 300 Gigabytes in Size**. [S.l.], 2020. Disponível em: https://decrypt.co/42427/bitcoin-blockchain-grows-to-300-gigabytes-in-size. Acesso em: 14 mai. 2021.

GAMAGE, H.T.M.; WEERASINGHE, H.D.; DIAS, N.G.J. A Survey on Blockchain Technology Concepts, Applications, and Issues. **SN Computer Science**, [S.I.], v. 1, n. 2, 2016. Disponível em: https://doi.org/10.1007/s42979-020-00123-0. Acesso em: 03 abr. 2021.

GOES, Christopher. **The Interblockchain Communication Protocol:** An Overview. Berlin, 2020. Disponível em: https://github.com/cosmos/ibc/raw/old/papers/2020-05/build/paper.pdf. Acesso em: 16 mai. 2021.

KAGEYAMA, Edson; MAZIERO, Carlos; SANTIN. An Experimental Peer-to-Peer E-mail System. In: IEEE INTERNATIONAL CONFERENCE ON COMPUTATIONAL SCIENCE AND ENGINEERING, 11, 2008, São Paulo. **Proceedings...** Massachusetts: IEEE Computer Society, 2008. p 203-208. Disponível em: https://doi.org/10.1109/CSE.2008.9. Acesso em: 05 abr. 2021.

KANSAL, Satwik. **Merkle Trees:** What They Are and the Problems They Solve. [S.l.], 2020. Disponível em: https://www.codementor.io/blog/merkle-trees-5h9arzd3n8. Acesso em: 10 mai. 2021.

MCDONALD, Jim. **Understanding Merkle pollards**. [S.l.], 2019. Disponível em: https://www.wealdtech.com/articles/understanding-merkle-pollards/. Acesso em: 10 mai. 2021.

MISLOVE, Alan *et al.* POST: A secure, resilient, cooperative messaging system. In: WORKSHOP ON HOT TOPICS IN OPERATING SYSTEMS, 9, 2003, Lihue. **Proceedings...** Lihue: USENIX, 2003. p. 61–66. Disponível em: https://www.usenix.org/legacy/events/hotos03/tech/full_papers/mislove/mislove.pdf. Acesso em: 08 abr. 2021.

NAKAMOTO, Satoshi. **Bitcoin:** A Peer-to-Peer Electronic Cash System. [S.l.], 2008. Disponível em: https://bitcoin.org/bitcoin.pdf. Acesso em: 11 abr. 2021.

PETROV, Christo. **50 Gmail Statistics To Show How Big It Is In 2020**. [S.l.], 2021. Disponível em: https://techjury.net/blog/gmail-statistics. Acesso em: 03 abr. 2021.

RADICATI. **Email Statistics Report, 2020-2024**. [S.l.], 2020. Disponível em: https://radicati.com/wp/wp-content/uploads/2020/01/Email_Statistics_Report,_2020-2024_Executive_Summary.pdf. Acesso em: 11 abr. 2021.

RIABOV, Vladimir V. SMTP (Simple Mail Transfer Protocol). In: BIDGOLI, Hossein (Ed.). **The Handbook of Information Security: Key Concepts, Infrastructures, Standards and Protocols**. Califórnia: John Wiley & Sons, 2006. p. 878-900.

ROCHA, João *et al*. Peer-to-Peer**:** Computação Colaborativa na Internet. In: SIMPÓSIO BRASILEIRO DE REDES DE COMPUTADORES, 22, 2004, Gramado. **Minicurso**... SBRC: UFRGS, 2004. p. 3-46.

TRAININI, Paulo R. S., CARISSIMI, Alexandre da S. **Análise das Vulnerabilidades do Sistema de Correio Eletrônico**. Rio Grande do Sul, 2005. Disponível em: https://www.researchgate.net/profile/Alexandre-Carissimi/publication/237499799_Analise_das_Vulnerabilidades_do_Sistema_de_Correio_Eletronico/links/564cd5cb08aeafc2aaaf8985/Analise-das-Vulnerabilidades-do-Sistema-de-Correio-Eletronico.pdf. Acesso em: 02 abr. 2021.

UMLAUF, Fernanda. **Bitcoin consome tanta energia quanto toda a Suíça, afirma estudo**. [S.I.], [2019]. Disponível em: https://www.tecmundo.com.br/mercado/143490-bitcoin-consome-tanta-energia-suica-afirma-estudo.htm. Acesso em: 11 abr. 2021.

VEERAMANI, Karthika; JAGANATHAN, Suresh. A quick synopsis of blockchain technology. **International Journal of Blockchain and Cryptocurencies**, [S.l], v. 1, n. 1, p. 54-66. 2019. Disponível em: https://www.inderscience.com/info/inarticletoc.php?jcode=ijbc&year=2019&vol=1&issue=1. Acesso em 16 mai. 2021.

WARREN, Jonathan. **Bitmessage:** A Peer-to-Peer Message Authentication and Delivery System. [S.l.], 2012. Disponível em: http://kevinriggen.com/files/bitmessage.pdf. Acesso em: 11 abr. 2021.

WAUGH, James. **Secret Hub:** Making Privacy One With the Cosmos. [S.l.], 2020. Disponível em: https://scrt.network/blog/secret-hub/. Acesso em: 16 mai. 2021.