Centro de Engenharia Elétrica e Informática
Departamento de Engenharia Elétrica

George Henrique Corcino Camboim

Contratos Inteligentes na Rede Ethereum

Campina Grande
Outubro de 2021

George Henrique Corcino Camboim

Contratos Inteligentes na Rede Ethereum

Trabalho de conclusão de curso submetido à Coordenação de Graduação em Engenharia Elétrica da Universidade Federal de Campina Grande como parte dos requisitos necessários para a obtenção do grau de Bacharel em Ciências no Domínio da Engenharia Elétrica.

Orientador:
Marcos Ricardo de Alcântara Morais, D. Sc.

Campina Grande
Outubro de 2021

George Henrique Corcino Camboim

Contratos Inteligentes na Rede Ethereum

Trabalho de conclusão de curso submetido à Coordenação de Graduação em Engenharia Elétrica da Universidade Federal de Campina Grande como parte dos requisitos necessários para a obtenção do grau de Bacharel em Ciências no Domínio da Engenharia Elétrica.

Aprovado em / /

Marcos Ricardo de Alcântara Morais, D. Sc.
UFCG

Gutemberg Gonçalves dos Santos Júnior, D. Sc.
UFCG

Campina Grande
Outubro de 2021

AGRADECIMENTOS

Devo antes de tudo agradecer aos meus pais, Silvana e José Maria, pois sem eles sequer estaria estudando em uma universidade. Agradeço também a minha madrinha Lenita e a meu primo Filipe, que muito me ajudaram durante toda a jornada em Campina Grande.

Agradeço aos professores Gutemberg Junior, Marcos Morais e Elmar Melcher, pela oportunidade de trabalhar e me encontrar profissionalmente no laboratório XMEN. Agradeço também aos colegas, aos líderes e engenheiros, assim como os gerentes do projeto e todos aqueles que me ajudaram e me inspiraram nesse trabalho.

Agradeço também de forma geral a toda equipe do departamento de Engenharia Elétrica: professores, coordenadores e técnicos, em especial Adail Ferreira e Tchai Oliveira, que muitas vezes me ajudaram durante a graduação.

Por fim, agradeço a todos os amigos que conheci no pensionato, na universidade e de forma geral em Campina Grande. Em especial, agradeço a Thiago, Lucas, Elvis Wendel e Vinicius; ainda tenho muito a aprender com todos vocês.

"Agora você sabe, e saber é metade da batalha."
G.I. Joe.

RESUMO

Este Trabalho de Conclusão de Curso introduz o conceito de contratos inteligentes e demonstra seu uso e fundamentos a partir da plataforma Ethereum, utilizando a linguagem Solidity como ferramenta de desenvolvimento. Os contratos são contextualizados como mecanismos computacionais capazes de fornecer serviços *web* de forma descentralizada. Além disso, esse trabalho relata o desenvolvimento de um *token* compatível com o padrão ERC-20, com o nome de DEECoin.

Palavras-chave: Ethereum, Contratos Inteligentes, Aplicações Descentralizadas, *Blockchain*.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE CÓDIGOS

LISTA DE ABREVIATURAS E SIGLAS

DApps	Aplicações Descentralizadas
DEE	Departamento de Engenharia Elétrica
EIP	Proposta de Melhoria Ethereum
ERC	Requisição por Comentários Ethereum
EVM	Máquina Virtual Ethereum
ICO	Oferta Inicial de Moedas
IDE	Ambiente de Desenvolvimento Integrado
UFCG	Universidade Federal de Campina Grande

SUMÁRIO

1 INTRODUÇÃO

Bitcoin foi inventado em 2008 com a publicação de um artigo intitulado "*Bitcoin: A Peer-to-Peer Electronic Cash System*" (NAKAMOTO, 2008), escrito por uma pessoa anônima sob a alcunha de Satoshi Nakamoto. Em seu artigo, Nakamoto descreve a capacidade dessa rede de permitir que pagamentos possam ser enviados entre duas pessoas sem a necessidade de autorização de terceiros.

A *blockchain* é uma estrutura de dados concebida por Nakamoto, e pode ser entendida como um banco de dados público que é distribuído, ou duplicado, através de muitos computadores em uma rede. *Block* refere ao fato que as transações são armazenadas em grupos sequenciais de informação chamados de blocos. *Chain* se refere ao fato de que cada bloco referencia criptograficamente o bloco anterior de forma encadeada. Essa estrutura é a pedra fundamental sobre a qual todas as criptomoedas são construídas.

O protocolo Ethereum, assim como o Bitcoin, é construído a partir do uso da *blockchain* e do um mecanismo de consenso descentralizado baseado em prova de trabalho. Seu componente chave de diferenciação em relação ao Bitcoin é a ideia de uma *blockchain* Turing-completa, com uma máquina virtual embutida na cadeia de blocos. Essa máquina, junto a uma linguagem de programação, permite o desenvolvimento de contratos inteligentes; programas armazenados e executados de forma descentralizada junto à rede.

1.1 OBJETIVOS

1.1.1 OBJETIVOS GERAIS

Esse trabalho busca trazer luz a um tópico recheado de dúvidas. Inspirado pelo trabalho de conclusão de curso de Leonardo Gouveia no período 2020.1, entitulado "*Blockchain*", o texto contido nesse documento busca uma extensão a respeito do tema.

Ao invés de comentar sobre Bitcoin e sua estrutura, o trabalho visa explorar os conceitos da rede Ethereum, suas propostas e aplicações, introduzindo conceitos de computação distribuída, passando por contratos inteligentes para enfim chegar a ideia de aplicações descentralizadas em um cenário de Web 3.0. De forma geral, o objetivo desse trabalho é fazer com que pessoas pouco familiarizadas com a rede Ethereum possam ser capazes de desenvolver seus primeiros contratos inteligentes.

1.1.2 OBJETIVOS ESPECÍFICOS

- Produção de uma guia teórico sintetizado e atualizado a respeito dos aspectos da rede Ethereum;
- Implementação de um contrato compatível com o padrão ERC-20;
- Implementação de testes de unidade do contrato.

1.2 METODOLOGIA

A metodologia empregada neste trabalho envolveu a realização de pesquisa bibliográfica sobre o tema proposto e o desenvolvimento de códigos e testes de contratos. Os contratos descritos neste texto foram desenvolvidos sob autoria própria; compilados e validados a partir do uso IDE Remix.

1.3 ESTRUTURA

Para se atingir os objetivos desse trabalho, os capítulos foram estruturados da seguinte forma:

- O capítulo 2 busca contextualizar a história e motivação por trás da criação das criptomoedas, iniciando pelos primeiros desenvolvimentos na área e passando pela criação e contexto do Bitcoin e Ethereum;
- O capítulo 3 introduz os conceitos da rede Ethereum e seus contrastes com a rede Bitcoin, explorando ideias de computação descentralizada, custos operacionais e limitações da *blockchain*;
- O capítulo 4 busca conceitualizar os fundamentos dos contratos inteligentes, introduzindo a linguagem Solidity, suas características e padrões de *design*;
- O capítulo 5 expõe as ideias e motivações por trás das aplicações descentralizadas e suas diferenças em comparação com aplicações web tradicionais.
- O capítulo 6 especifica o processo de desenvolvimento e testes de um contrato de criptomoeda;

2 CONTEXTO HISTÓRICO

O surgimento de dinheiro digital viável está correlacionado com o desenvolvimento no ramo da criptografia. Não há nenhuma surpresa nisso quando se considera os desafios que envolvem o uso de bits, que podem ser copiados e manipulados, para representar valores que podem ser trocados por bens e serviços.

Em 1983 o criptógrafo e cientista da computação americano David Chaum começou a experimentar com conceitos de dinheiro eletrônico. Ele concebeu uma moeda eletrônica que poderia ser trocada de forma segura e anônima através de uma fórmula para encriptar as transações a partir de assinaturas digitais. Chaum fundou o DigiCash alguns anos depois, porém, em 1998 sua empresa foi levada à falência.

Durante a década de 90, duas outras invenções, o B-money de Wei Dai e o Bit Gold de Nick Szabo, foram concebidas. Apesar das inovações introduzidas na área, como o uso de computadores distribuídos e o algoritmo de prova de trabalho, ambas as invenções falharam em obter alcance de público e sustentar os custos de operações.

De acordo com Antonopoulos (ANTONOPOULOS, 2017), o problema que esses cientistas estavam tentando solucionar pode ser sintetizado em torno de três questões básicas que qualquer indivíduo se perguntaria antes de aceitar uma nova forma de dinheiro:

1. É possível confiar que o dinheiro não seja falso?
2. É possível confiar que o mesmo dinheiro só poderá ser gasto uma única vez?
3. É possível ter certeza que uma pessoa não poderá alegar propriedade sobre o dinheiro de forma ilegítima?

Instituições emissoras de moeda estão constantemente lutando contra a falsificação monetária utilizando papéis especiais e técnicas de impressão sofisticadas. Dinheiro físico resolve naturalmente o problema de gasto duplo devido ao fato que um único papel-moeda não pode estar fisicamente em dois lugares ao mesmo tempo. Porém, nos dias de hoje, mesmo as moedas fiduciárias impressas por bancos centrais nacionais, títulos de dívida e derivativos financeiros são frequentemente trocados e armazenados de forma completamente digital. Nestes casos, servidores centralizados são responsáveis pela legitimidade dos ativos e de suas transações.

Neste tipo de sistema, a confiança sobre o sistema financeiro está totalmente concentrada na crença de que a instituição responsável pelo servidor central agirá de boa fé, e que a infraestrutura digital responsável pelo sistema será robusta e a prova de falhas.

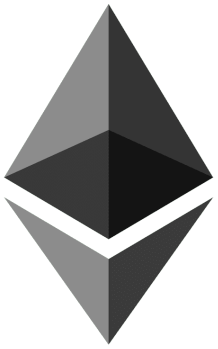
Curiosamente, não foi uma falha técnica que resultou na criação e popularidade das criptomoedas, mas sim, uma crise de confiança. A crise financeira de 2008 atingiu o mundo abruptamente a partir de um dos setores mais estáveis e de maior confiança: o imobiliário. Uma bolha de crédito fácil e de alto volume estourou, e requereu intervenção direta de governos do mundo todo para evitar um total colapso do sistema monetário internacional. Os governos centrais, e por consequência seus contribuintes, tiveram que gerar pacotes de estímulos para os bancos prestes a falir, cobrindo as dívidas e os riscos que estas instituições se expuseram. Um ano depois da crise, surgiu o Bitcoin.

Bitcoin foi criado em um contexto de alta desconfiança em relação ao sistema bancário internacional. Em poucos anos, teve um crescimento exponencial de valor e de número de usuários. Com a inovação de uma estrutura de dados chamada de *blockchain* e um mecanismo descentralizado especial de consenso que garante a validade de transações financeiras, sem a necessidade de autoridades centrais para regular e garantir seu funcionamento, pesquisadores começaram a ser perguntar como é possível utilizar essa tecnologia para resolver outros tipos de problema.

Nos anos que se seguiram, houve diversas propostas de aplicações para esta tecnologia. Em 2014, Vitalik Buterin conceitualizou a rede Ethereum como uma aplicação da *blockchain* para a construção de um sistema global de computadores distribuídos, capaz de garantir a execução de códigos e a legitimidade de seus resultados de forma distribuída. Esse design evoluiu desde sua criação e permitiu a execução de contratos inteligentes, uma ideia inicialmente desenvolvida por Nick Szabo. Recentemente, o sistema de contratos inteligentes da rede Ethereum teve um grande crescimento em sua adoção, resultando na criação de milhares de contratos responsáveis por gerenciar valores de bilhões de dólares em ativos digitais.

3 A REDE ETHEREUM

Figura 1 – Logotipo da rede Ethereum.



Fonte: (ETHEREUM. . . ,)

A rede Ethereum foi concebida em 2013 por Vitalik Buterin, sintetizada em 2014 no artigo "*A next-generation smart contract and decentralized application platform*" (BUTERIN et al., 2014), e lançada em 2015 junto a uma oferta inicial de moedas (ICO). Essa rede foi fortemente inspirada nas inovações e limitações da rede Bitcoin.

O componente chave de diferenciação entre as *blockchains* do Bitcoin e do Ethereum, é a ideia de que a rede Ethereum é uma *blockchain* Turing completa, com uma linguagem de programação embutida. A habilidade de criar interações entre usuários no futuro, ativadas quando certas condições forem alcançadas, é uma poderosa adição a uma *blockchain*. Isso permite que desenvolvedores adicionem fluxos de controle a transações de criptomoedas. Na rede Bitcoin, todas as transações válidas acontecem assim que possível.

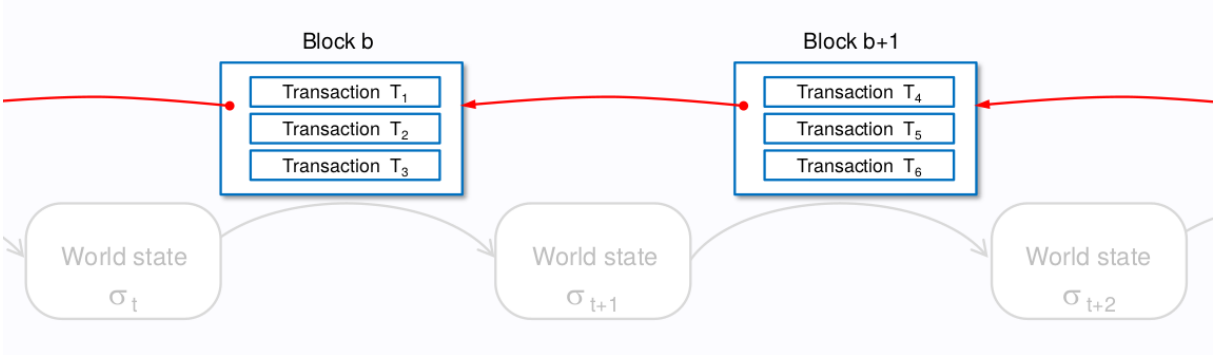
A rede Ethereum utiliza o conceito do registro público de transações, criado pelo Bitcoin, e o propõe como um modelo de computador virtual, dando a instruções computacionais o mesmo nível de certeza e legitimidade que as transações monetárias de bitcoins possuem. A validação de transações nesta rede ocorre de forma similar à rede Bitcoin, através do mecanismo de consenso de prova de trabalho. Junto a validação das transações, é realizada a execução de códigos de máquina através de uma máquina virtual.

O modelo de governança da rede Ethereum é liderado por uma fundação sem fins lucrativos, que mantém uma equipe de desenvolvedores dedicados ao desenvolvimento da rede. O desenvolvimento de aplicações é realizado de forma descentralizada por milhares de companhias e pessoas em todo o globo. A proposta do Ethereum é criar sistemas econômicos descentralizados na forma de puro software, através de contratos inteligentes.

3.1 BLOCKCHAIN COMO UMA MÁQUINA DE ESTADOS

Do ponto de vista técnico, a *blockchain* do Bitcoin pode ser entendida como um sistema de transição de estados. O estado, nesse contexto, se refere à distribuição total de moedas entre todas as carteiras existentes. Ao realizar transferência entre carteiras, um novo bloco na cadeia está essencialmente alterando o estado do sistema. Esse estado não pode ser observado diretamente; para se ter uma completa noção do estado é necessário contabilizar todas as transações da cadeia desde o bloco genesis.

Figura 2 – Blockchain representando uma cadeia de transição de estados.



Fonte: (TANI, 2018)

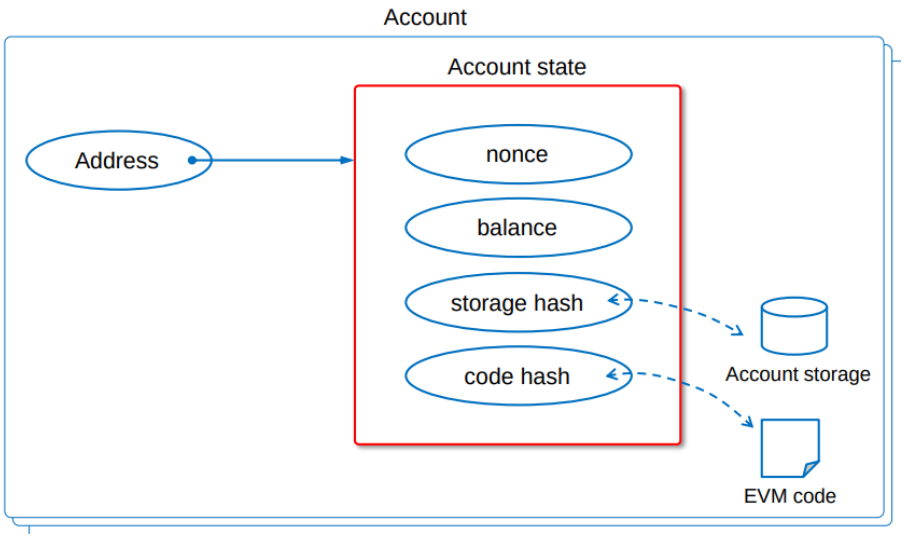
Na rede Ethereum, o estado é composto por objetos chamados de "contas". As transições de estados são caracterizadas por transferência direta de valor e informação entre as contas. Cada conta se caracteriza por um endereço com 20 bytes de largura, e possui quatro campos:

- O nonce, um contador utilizado para garantir que cada transação será processada uma única vez;
- O balanço da conta, em *ether*;
- O código do contrato, se existir;
- O armazenamento da conta, que será vazio por padrão.

Ether é o nome que se dá a moeda nativa da rede Ethereum, e é usada para pagar os custos de transações. Existem dois tipos fundamentais de contas: contas sob propriedade externa e contas de contratos.

Uma conta sob propriedade externa é controlada a partir de chaves privadas, geralmente sob posse de pessoas físicas ou jurídicas, e representam entidades externas à rede. Já uma conta pertencente a um contrato é controlada pelo código de contrato contido em sua conta e representa uma entidade pertencente à *blockchain*. Esse tipo de conta

Figura 3 – Estrutura de dados das contas.



Fonte: (TANI, 2018)

é dormente até o momento que é ativada por uma mensagem da rede, que resultará na execução do código de contrato, permitindo a leitura e armazenamento de informações na conta além do envio de mensagens para outras contas.

Contratos são agentes autônomos armazenados dentro da *blockchain* Ethereum, executando um pedaço de código sempre que são acordados por uma mensagem ou transação. Eles possuem total controle sobre seu balanço monetário, assim como mecanismos para armazenar e transacionar valores e informações. Esses agentes podem se comunicar entre si, assim como com entidades externas da rede.

A troca de valores e informações nesta rede possui um sentido mais amplo que seu equivalente na rede Bitcoin. Na rede Ethereum, existem transações e mensagens. Uma transação é um pacote de dados inserido na *blockchain*, assinado criptograficamente, e enviado por uma conta sob propriedade externa. Uma transação contém:

- O destino da mensagem;
- Uma assinatura identificando quem a enviou;
- Uma quantidade monetária de *ether* a ser enviada a conta destino;
- Um campo opcional de dados;
- Um valor STARTGAS, representando o máximo de passos computacionais que o remetente está disposto a gastar pela transação;
- Um valor GASPRICE, representando o preço em *ether* de cada passo computacional.

Os três primeiros campos são comuns a qualquer criptomoedas. O campo de dados não possui nenhum significado por padrão, mas pode ser utilizado para transferir informações entre contas que podem então ser utilizadas na execução de contratos. Os últimos dois campos são cruciais no modelo de serviços da rede Ethereum. Para evitar o desperdício de recursos computacionais na execução dos códigos de contrato e evitar laços lógicos infinitos, propositais ou acidentais, o remetente deve definir um limite de custos computacionais que está disposto a gastar.

Embora contratos não sejam capazes de realizar transações, eles possuem a habilidade de enviar mensagens para outras contas. Mensagens são objetos virtuais que não existem na *blockchain*, sendo executados apenas no ambiente de execução Ethereum. Uma mensagem contém:

- O remetente da mensagem, que é implícito;
- O destino da mensagem;
- A quantidade monetária de *ether* a ser enviada entre as contas;
- O campo opcional de dados;
- O valor STARTGAS.

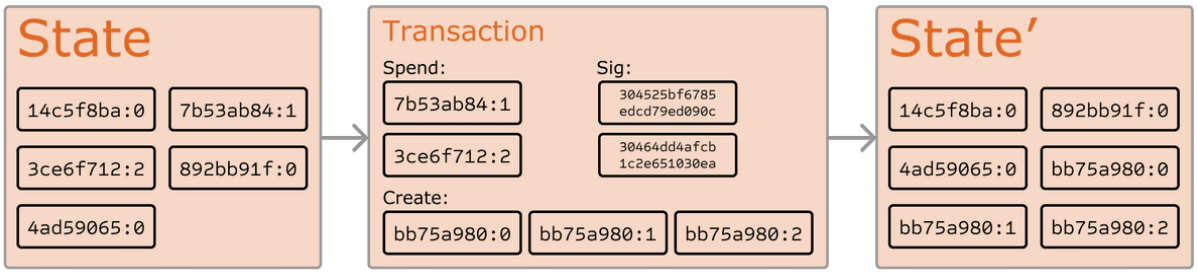
Essencialmente, uma mensagem se comporta como uma transação, porém só pode ser produzida por um contrato e não diretamente por uma entidade externa. Através das mensagens, os contratos podem ter relações com outras contas da mesma forma que agentes externos são capazes. Dessa forma, mensagens são efeitos colaterais de transações destinadas a contratos.

Uma transação destinada a um contrato pode resultar em uma reação em cadeia de mensagens e execuções de código, baseado na relação entre os contratos envolvidos. Para uma transação ser validada junto a *blockchain*, além da transferência de valor e informação contida na transação, todas as transições de estado que resultam da execução em cadeia de contratos precisam ser processadas. Uma transação só poderá ser validada junto ao bloco, quando toda a cadeia de contratos, se existir, for processada.

No modelo de execução da rede Ethereum, uma transação é uma operação atômica, que não pode ser interrompida ou dividida em sub-operações, e que devem ser executadas de forma serial. Cada transação representa um conjunto de mudanças de estados das contas envolvidas na operação. O estado global é uma soma dos estados individuais de cada conta pertencente à rede.

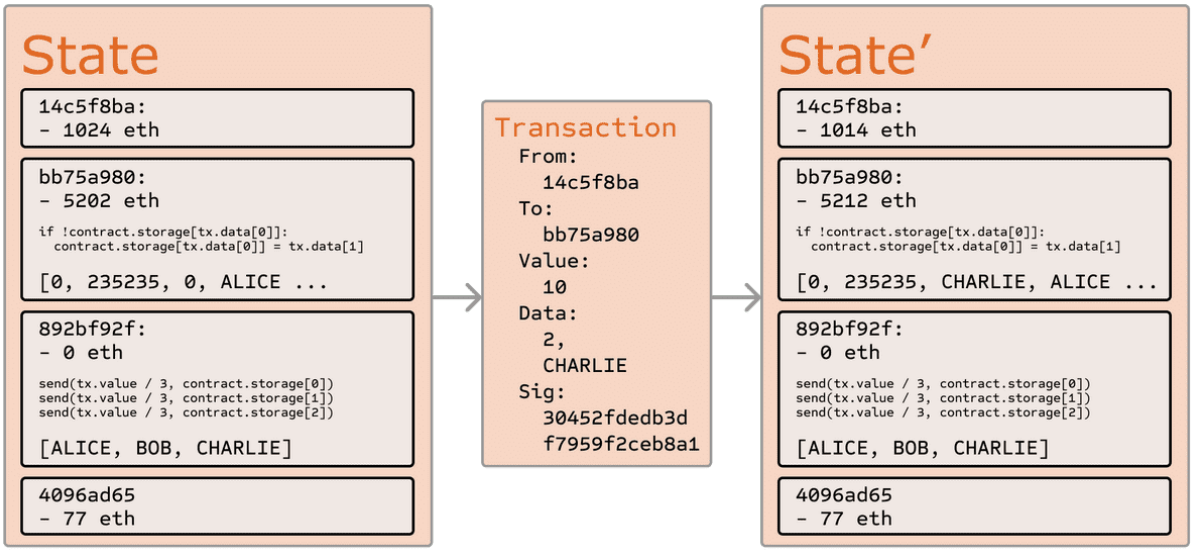
Todas as transações Ethereum são armazenadas na *blockchain*, criando uma história canônica de transições de estados que são armazenadas em cada nó individual da rede.

Figura 4 – Transição de estados na rede Bitcoin.



Fonte: (BUTERIN et al., 2014)

Figura 5 – Transição de estados na rede Ethereum.



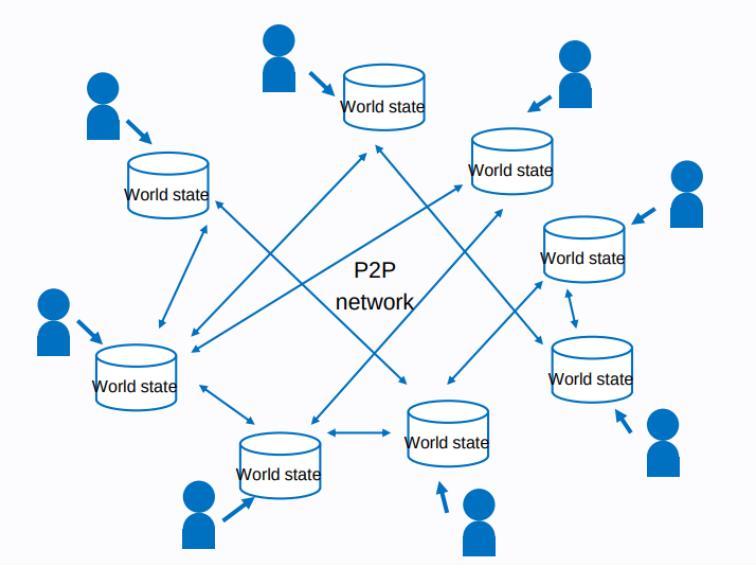
Fonte: (BUTERIN et al., 2014)

Através dos mesmos mecanismos de consenso utilizados na rede Bitcoin, a rede Ethereum garante o consenso do estado global da rede como um todo, a partir de estados globais gerados em cada nó minerador de forma descentralizada. Quanto ao consenso, a maior diferença entre as duas redes está na escolha do algoritmo de prova de trabalho: a função hash utilizada na rede Ethereum é o algoritmo Keccak-256.

3.2 MODELO DE EXECUÇÃO

O modelo de execução específica como o estado do sistema é alterado, dado uma série de instruções e lista de informações do ambiente. Esse modelo é especificado através de um modelo formal de uma máquina virtual, chamada de Ethereum Virtual Machine (EVM). Essa máquina é quase Turing completa; o quase, nesse caso, se refere ao fato de que é intrinsecamente limitada através de um parâmetro chamado de *gas*, que limita a quantidade total de computação a ser exercida (WOOD et al., 2014).

Figura 6 – Composição do estado global de forma distribuída.



Fonte: (TANI, 2018)

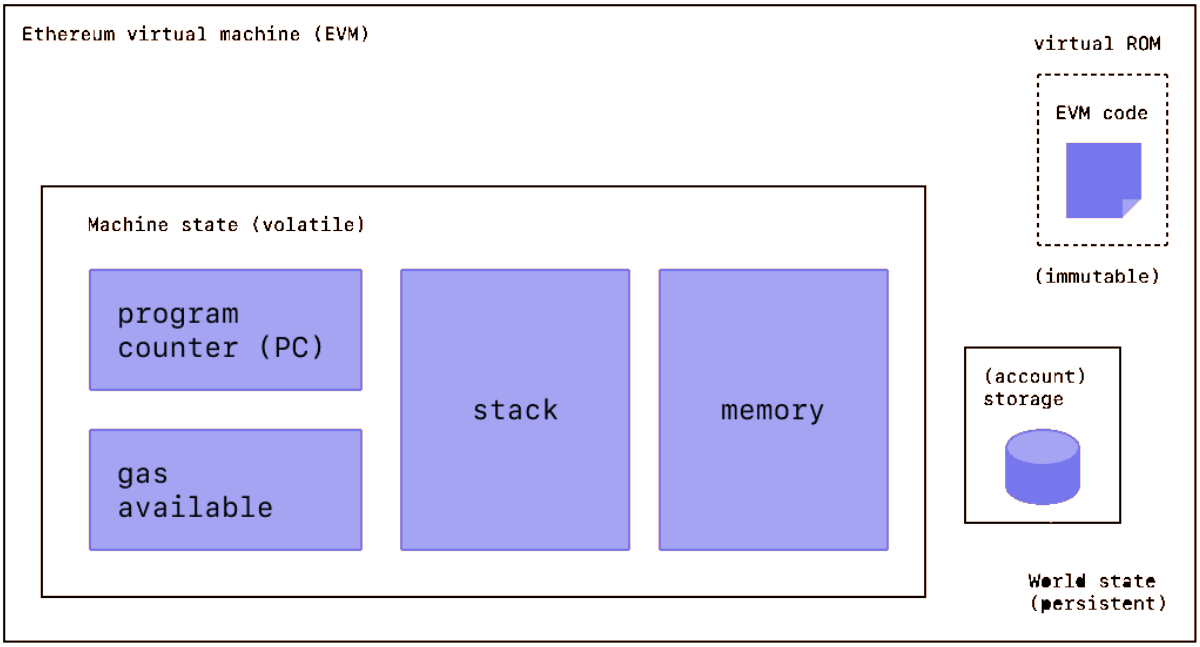
A EVM é um computador global formado por uma grande quantidade de computadores menores, que se estrutura com uma arquitetura de execução de instruções baseada em pilha. Essencialmente, é uma máquina de estados baseada em transações. Essa máquina virtual possui um tamanho de palavra, e por consequência a largura da pilha, de 256 bits. O código presente em uma conta de contrato é armazenado na forma de um bytecode pré-compilado de baixo nível, que é referenciado como código EVM. Esse código é originalmente gerado a partir da compilação de códigos de alto nível.

O código consiste em uma série de bytes, onde cada byte representa uma operação. De forma geral, o código é executado a partir de um loop infinito que executa repetidamente as operações apontadas pelo contador do programa (que sempre se inicia com valor zero). O código chegará ao fim ao se atingir um erro de execução, ou ao identificar uma instrução STOP ou RETURN. As operações têm acesso a três espaços de memória: a pilha, a memória principal e o armazenamento de longo prazo (BUTERIN et al., 2014).

O modelo de memória é uma simples array de bytes endereçado por palavra. A pilha da memória possui um tamanho máximo de 1024 palavras. Diferentemente da memória principal, que é volátil, o armazenamento de longo prazo é não-volátil e faz parte do estado de uma conta, fazendo assim parte do estado global da rede. O código do contrato é armazenado temporariamente na forma de uma memória ROM virtual durante toda a execução do processo.

A EVM possui uma capacidade de execução de instruções excepcional, sendo capaz de lidar com problemas de stack underflow e de instruções inválidas (WOOD et al., 2014). Em casos casos de exceção, a máquina irá interromper a execução e reportar o problema

Figura 7 – Arquitetura de dados da Máquina Virtual Ethereum.



Fonte: (ETHEREUM. . . ,)

para o agente de execução, seja ele o processador de transações ou de forma recursiva o ambiente de execução, que irão lidar com o problema de forma independente. Antes de realizar uma computação, o processador dessa máquina tenta garantir que as seguintes informações estão disponíveis e são válidas:

- O estado do sistema;
- A quantidade restante de *gas*;
- O endereço da conta que possui o código a ser executado;
- O endereço da conta responsável a transação original que resultou na execução do código;
- O endereço da conta que causou a execução do código;
- O preço do *gas* que originou a execução;
- Os dados de entrada;
- Valor, em Wei, a ser transferido para a conta do contrato;
- O código binário do contrato;
- O cabeçalho do bloco atual;
- O tamanho da pilha de chamadas da mensagem.

No início da execução de uma transação, a memória e a pilha estão vazias e o contador do programa é zero. A EVM então executa a transação de forma recursiva, computando o estado do sistema e o estado da máquina em cada ciclo. O estado do sistema é o estado global da rede Ethereum. O estado da máquina é composto por:

- Gas restante;
- Contador do programa;
- Conteúdo da memória;
- Número de palavras ativas na memória;
- Conteúdo da pilha.

Durante cada ciclo, o contador do programa é incrementado, e uma quantidade apropriada de *gas* é subtraída do *gas* restante em relação a operação realizada. Ao fim de cada ciclo, a máquina decide continuar o processo, realizar uma interrupção controlada para sinalização do seu fim, ou atinge uma exceção que interrompe o processo.

3.3 ETHER E GAS

Ether, denominado pelo símbolo ETH, é a moeda nativa da rede Ethereum. Assim como bitcoin, esta moeda é utilizada por pessoas, governos e corporações para transferir valor e comprar produtos e serviços. Fundamentalmente, é a única forma de pagamento de comissões aceita pela rede. O *ether* pode ser subdividido e fracionado até a sua unidade mínima, chamada de Wei.

Entretanto, diferentemente do bitcoin, o *ether* foi pensado não puramente como uma reserva de valor e moeda de troca, mas também como um commodity utilizado para realizar computações dentro da rede Ethereum, tal como um combustível computacional para execução de aplicações e serviços. Quando se paga por recursos computacionais na rede Ethereum, isso inclui o custo de execução das transações, de armazenamento de informações em memória e do uso de banda de internet.

A unidade fundamental de computação é chamada de *gas*. Comumente, um passo computacional custa 1 *gas*, porém algumas operações custam quantidades maiores pois são computacionalmente mais intensivas, ou utilizam uma quantidade de dados que precisam ser armazenadas junto ao estado da rede. Existe também um custo de 5 unidades de *gas* para cada byte no pacote da transação. O preço do *gas*, em *Gwei*, flutua em função da demanda de serviços e oferta de recursos da rede. Um aumento no números de transações requeridas resulta em um aumento do preço do combustível, enquanto um aumento no

Tabela 1 – Denominações da moeda *ether*.

Unidade	Valor em Wei	Valor em Ether
wei	1 wei	10^{-18} ether
Kwei (babbage)	10^{3} wei	10^{-15} ether
Mwei (lovelace)	10^{6} wei	10^{-12} ether
Gwei (shannon)	10^{9} wei	10^{-9} ether
microether (szabo)	10^{12} wei	10^{-6} ether
milliether (finney)	10^{15} wei	10^{-3} ether
ether	10^{18} wei	1 ether

Fonte: (ETHEREUM. . . ,)

número de nós mineradores e do poder computacional da rede resulta em uma diminuição deste preço.

A intenção por trás desse sistema de taxas é fazer com que um indivíduo que ataque a rede, pague proporcionalmente por cada recurso consumido, incluindo processamento, armazenamento e largura de banda. Assim, cada transação que leva a rede a consumir uma maior quantidade de recursos, precisa pagar taxas proporcionais a essa carga computacional.

Devido ao fato que sistemas distribuídos não possuem donos per se, qualquer entidade com computadores podem participar da rede Ethereum e validar transações, tal como a mineração na rede Bitcoin. O sistema de consenso e mineração funciona de forma similar nas duas redes, com poucas diferenças. Nós mineradores fornecem consenso a rede a respeito do estado e da ordem das transações realizadas no sistema. Esse processo é computacionalmente intensivo e consome muita eletricidade, que custa dinheiro. Portanto os mineiros são recompensados com uma premiação por cada bloco que mineram: aproximadamente 5 *ethers*.

Além deste prêmio, os nós mineradores são recompensados pela execução de contratos na rede através do sistema de *gas*. O custo associado com os gastos elétricos dos servidores rodando nós da rede Ethereum é um dos fatores que dá ao *ether* seu valor intrínseco. Isto é, o fato de que alguém gastou dinheiro para comprar computadores, pagar pela sua manutenção e consumo elétrico com a finalidade de obter *ether*, garante a criptomoeda um valor correlacionado com o mercado energético e computacional.

3.4 REDES

Visto que o Ethereum é na verdade um protocolo, é possível que existam múltiplas redes independentes seguindo as mesmas regras do protocolo, sem interagir umas com as outras. Essas redes são ambientes Ethereum distintos, que podem ser acessadas para diferentes casos de uso, como desenvolvimento, testes ou produção.

Redes públicas são acessíveis para qualquer pessoa no mundo com acesso a internet. Qualquer usuário pode ler ou criar transações em *blockchains* públicas, assim como validar as transações sendo executadas. A *Mainnet* é a rede pública primária, onde transações com valores monetários reais ocorrem através do consenso. Quando pessoas e corretoras discutem os preços do *ether*, elas estão se referindo ao valor do *ether* da *Mainnet*.

Além da *Mainnet*, existem redes públicas voltadas para testes. Essas redes são utilizadas pelos desenvolvedores do protocolo e dos contratos para testar tanto atualizações de protocolo como contratos inteligentes em um ambiente similar ao de produção. De forma geral, é importante testar qualquer código de contrato em uma rede de testes antes do seu lançamento na rede principal. A maioria das redes de teste utilizam um mecanismo de consenso baseado em prova de autoridade. Isso significa que um pequeno número de nós são escolhidos para validar as transações e criar novos blocos.

Ethers, nas redes de teste, não possuem real valor monetário, não havendo um mercado para sua criação e transação. Devido a necessidade do uso de *ethers* para se comunicar no protocolo Ethereum, usuários devem obter moedas através do uso de faucets. Faucets são aplicações web que enviam *ethers* de teste para qualquer endereço solicitado.

Alguns exemplos de redes públicas de testes são (ETHEREUM. . . ,):

- Görli: *testnet* com consenso baseado em prova de autoridade que funciona com múltiplos clientes;
- Kovan: *testnet* com consenso baseado em prova de autoridade para usuários utilizando o cliente OpenEthereum;
- Rinkeby: *testnet* com consenso baseado em prova de autoridade para usuários utilizando o cliente Geth;
- Ropsten: *testnet* com consenso baseado em prova de trabalho, sendo a rede com comportamento mais próximo da *Mainnet*.

Redes Ethereum também podem ser privadas, sendo assim isoladas ou protegidas da rede principal. Desenvolvedores podem criar nós locais, utilizando o protocolo Ethereum, com a finalidade de permitir iterações de testes mais rápidas. Redes privadas também podem ser utilizadas para garantir privacidade de transações junto a rede através do seu isolamento.

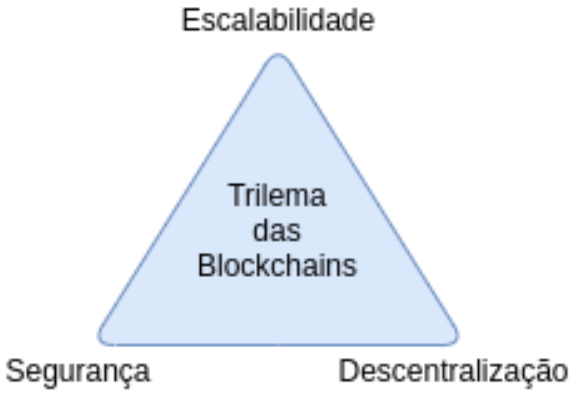
Objetos digitais transacionados em redes privadas podem ser transferidos de volta à rede principal a partir de *bridges*. Essas *bridges* também podem ser utilizadas para conectar diferentes *blockchains* com diferentes protocolos e características junto a rede Ethereum, criando um sistema misto de *blockchains* interconectadas.

3.5 DESAFIOS

Normalmente, soluções baseadas em *blockchain* buscam equilibrar três características distintas: segurança, descentralização e escalabilidade. Isso tem se mostrado uma impossibilidade até o momento; as tentativas de construir soluções distribuídas tiveram que fazer um compromisso, sacrificando um dos três fatores.

Cunhado por Vitalik Buterin, o trilema das *blockchains* aponta para os desafios que desenvolvedores enfrentam na tentativa de criar uma *blockchain* que atenda os três fatores sem comprometer nenhum. Esse trilema faz parte de um problema herdado e não resolvido dentro da ciência computacional chamado de teorema de Brewer. O teorema de Brewer, ou teorema do CAP, diz que é impossível que o armazenamento de dados de forma distribuída possua ao mesmo tempo consistência, disponibilidade e particionamento tolerante a falhas.

Figura 8 – Trilema das blockchains.



Fonte: Autor

As redes Ethereum e Bitcoin foram bem sucedidas em garantir a descentralização e segurança de suas redes, sob um alto custo na escalabilidade. Os problemas de escalabilidade da rede Ethereum podem ser sintetizados em três sintomas: a instabilidade e altos preços do *gas*, os custos energéticos associados a mineração dos blocos, e o grande aumento em espaço de memória da *blockchain*.

Embora o preço do *gas* seja modelado de forma a garantir um equilíbrio entre a demanda por transações e a oferta de recursos computacionais, esses dois aspectos não possuem a mesma agilidade. Quando ocorre um surto de transações requisitadas, a rede é incapaz de fornecer de imediato um aumento na taxa de transações validadas, fazendo com que o preço do *gas* suba abruptamente. Esse descasamento entre oferta e demanda resulta em grandes flutuações nos custos de transações durante os dias, fazendo com que os preços de execuções dos contratos no futuro se tornem imprevisíveis. Além disso, uma rede com as ambições de fornecimento de serviços que a Ethereum possui, deve ter como

meta a manutenção dos custos de transações em torno dos centavos de dólar. Essa faixa é completamente fora da realidade que a rede vive hoje.

Outra implicação negativa desses problemas de escalabilidade é que o grande consumo energético por parte dos nós responsáveis pela mineração e consenso da rede é um problema a nível de sociedade. Quando levamos em consideração a escassez desse recurso, e que a maioria dos países ainda possui sua malha energética majoritariamente composta por queima de combustíveis fósseis, esse nível de consumo se torna inaceitável para a manutenção de um sistema computacional.

Ambos esses problemas estão intrinsecamente relacionados ao algoritmo de prova de trabalho, que é fundamentalmente lento e custoso. É por causa disso, que a rede Ethereum está no processo de lançar diversas atualizações programadas, cujo resultado final será chamado de Ethereum 2.0. Essa nova versão, também chamada de Serenity, inclui, entre diversas alterações, a modificação do algoritmo central de consenso em favor de um algoritmo de *Proof-of-Stake*.

A proposta desse mecanismo de consenso é que o poder de validação dos nós não seja mais proporcional ao poder computacional, e sim, a quantidade monetária sob posse das carteiras associadas ao nó. Esse método de consenso se baseia na ideia de que, aqueles que são mais expostos aos riscos de segurança da rede são os mesmos que mais tem interesse em garantir que a rede não sofra ataques. Fraudes e colapsos de segurança poderiam impactar drasticamente o valor monetário do *ether*, o que significa que os indivíduos que possuem grandes quantidades dessa moeda são os que mais tem interesse que a moeda mantenha seu valor.

Esse mecanismo busca de uma vez só resolver o problema do alto custo energético da rede, do alto preço do *gas* e das baixas taxas de transmissão. Apesar disso, críticos do mecanismo argumentam que essa mudança representa uma diminuição da segurança da rede. Assim, voltamos ao trilema das *blockchains*.

Como forma de compromisso, a atualização 2.0 da rede Ethereum busca implementar também o conceito de *sharding*. Em banco de dados, o conceito de *sharding* é o processo de dividir bancos de dados horizontalmente, compartilhando a carga total. Em sua atualização, Ethereum busca dividir a *blockchain* em subcadeias independentes, cada uma com seu estado e possuindo até mesmo protocolos diferentes. Sharding permite que os mecanismos *Proof-of-Work* e *Proof-of-Stake* possam conviver na mesma rede, ao mesmo tempo que soluciona o problema de armazenamento dos dados da cadeia.

4 CONTRATOS INTELIGENTES

O conceito de *Smart Contracts*, ou contratos inteligentes, precede a criação do Bitcoin e Ethereum. O termo foi inicialmente cunhado pelo cientista da computação e criptografo Nick Szabo, que o definia como “contratos computáveis e digitais, cuja a execução e aplicação dos termos contratuais possam ser cumpridos de forma automática, sem a necessidade de intervenção humana” (SZABO, 1997). Szabo comparou o conceito de contratos inteligentes com o de uma máquina automática de vendas de refrigerante:

> "Quando o dinheiro é pago, um conjunto irrevogável de ações são executadas. O dinheiro é retido e o refrigerante é fornecido. A transação não pode ser interrompida na metade. O dinheiro não pode ser retornado uma vez que o refrigerante é entregue. Os termos da transação estão de certa forma inseridos no hardware e no software que roda na máquina".

No contexto Ethereum, contratos inteligentes são agentes autônomos armazenados na blockchain. São acordos especificados na forma de software, e armazenados sob a propriedade de uma conta endereçada, criados com o objetivo de garantir transferências de moedas ou dados quando determinadas condições são atingidas. Esses contratos podem funcionar e ter validade por tempo indeterminado depois que foram escritos e lançados, assumindo que a rede prossiga com seu funcionamento, e mesmo que agentes mau intencionados tentem interferir.

Contratos são criados a partir de uma transação, que cria uma nova carteira, armazena o código EVM e executa seus métodos construtores. Uma vez que é criado, um contrato inteligente é identificado por um endereço de contrato. Cada contrato armazena uma quantidade de moedas *ether*, assim como um espaço interno para armazenamento de dados. Códigos de contrato podem manipular variáveis e exercer fluxos de controle tal como programas tradicionais. Usuários definem contratos utilizando linguagens de programação de alto nível, que são então compiladas em um código binário.

Os contratos são executados de forma isolada. Eles podem observar dados disponíveis publicamente na blockchain e se comunicar através de mensagens com outros contratos, porém são incapazes de se comunicar com qualquer serviço externo à rede. Essa é uma decisão de design; a dependência em informações externas poderia ser capaz de afetar o consenso e o estado do sistema, criando riscos para a segurança e descentralização.

Outra característica dos contratos na rede Ethereum é a imutabilidade. Sua definição, isto é, seu código, não pode ser modificado ou atualizado uma vez que é inserido na

blockchain. Para se atualizar um contrato, é necessário inserir a nova versão sob um novo endereço. É por esse motivo que se deve ter muito cuidado e atenção com a qualidade do código, e garantir uma verificação prévia do seu funcionamento para ter a certeza que não irá introduzir bugs que nunca poderão ser consertados. Somado a isso, contratos implementados na rede Ethereum tem seu código binário limitado a 24 *kilobytes*. Apesar desses fatores, existem padrões de projeto voltados ao desenvolvimento em blockchain que permitem a atualização e modificação de contratos em funcionamento, assim como mecanismos para contornar a limitação de tamanho.

O ciclo de desenvolvimento de contratos inteligentes de forma geral é composto de três passos:

1. Implementação: A escrita do contrato na forma de código;
2. Testes: A escrita de códigos e ambientes de teste para garantir que o contrato irá funcionar de forma apropriada. Embora esse passo não é obrigatório, ele é altamente recomendável devido a natureza imutável dos contratos na rede;
3. Deployment: O lançamento do contrato na blockchain na forma de uma transação.

4.1 SOLIDITY

Uma das vantagens do desenvolvimento de contratos na rede Ethereum é que os mesmos podem ser programados utilizando uma gama de diferentes linguagens. O código de contrato é armazenado na blockchain na forma de um código binário chamado de código EVM, que pode ser gerado a partir da compilação de códigos de alto nivel com melhor legibilidade. Algumas das linguagens dedicadas para essa finalidade são Solidity, Vyper, Yul, DAML e Fe. Linguagens de propósito geral, como C++, Javascript e Golang também podem ser utilizadas nesse tipo de desenvolvimento através do uso de bibliotecas. Qualquer linguagem de programação pode ser utilizada no desenvolvimento de contratos, desde que exista um compilador para realizar sua conversão.

A linguagem Solidity é a mais popular para o desenvolvimento de contratos na rede Ethereum. É uma linguagem orientada a objetos e foi criada por alguns dos desenvolvedores do projeto Ethereum. É estaticamente tipada, o que significa que os tipos das variáveis já são conhecidos e definidos no momento da compilação. Sintaticamente é baseada em colchetes e fortemente inspirada em C++ e Javascript.

Em Solidity, variáveis podem ser catalogadas de três formas: locais, globais e de estado (SOLIDITY...,). Variáveis locais são definidas dentro de funções e não são armazenadas na *blockchain*, sendo voláteis por definição. As variáveis globais armazenam informações a respeito do ambiente de execução EVM, de parâmetros da transação e do

Tabela 2 – Tipos de dados na linguagem Solidity.

Tipo	Descrição
bool	Valor booleano, sendo verdadeiro ou falso
int	Valor inteiro positivo ou negativo
unint	Valor inteiro não negativo
fixed	Valor real baseado em ponto fixo, positivo ou negativo
ufixed	Valor real baseado em ponto fixo, não negativo
string	Sequência de caracteres
address	Representa o endereço (20 bytes) de uma conta, possuindo métodos associados para transferência de dados

Fonte: (SOLIDITY...,)

bloco. Já as variáveis de estado representam o armazenamento interno de dados junto a carteira de contrato e sendo assim, ocupa espaço junto a *blockchain*. O uso de variáveis de estado deve ser minimizado em um contrato devido ao seu custo monetário.

A tabela 2 demonstra os tipos de variáveis permitidos na linguagem e suas descrições. Os tipos *int* e *uint* podem possuir sufixos representando a largura, entre 8 e 256 bits. O código 5 exemplifica e contextualiza os tipos descritos. A linguagem também permite o uso de objetos mais complexos, como *structs*, *enums*, *mappings* e *arrays* estáticas e dinâmicas.

Código 1 – Demonstração de tipos de variáveis na linguagem Solidity

```
1  // SPDX-License-Identifier: GPL-3.0
2  pragma solidity >=0.7.0 <0.9.0;
3
4  contract DataTypes {
5
6      // Variaveis de estado
7      string public greeting = "Hello!";
8      address public owner = 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4;
9      ufixed public pi = 3.14;
10
11     // Funcao
12     function Foo() public {
13         // Variaveis locais
14         uint luckyNumber = 7;
15         bool isContract = true;
16
17         // Variaveis globais
18         address msgSender = msg.sender; // Remetente da mensagem
19         address txSender = tx.origin; // Remetente da transacao
20         uint diff = block.difficulty; // Dificuldade do bloco
21     }
22 }
```

Os contratos podem interagir externamente através da definição de funções. A visibilidade é definida a partir dos tipos de funções. Por questões de legibilidade e boas

Tabela 3 – Tipos de funções na linguagem Solidity.

Tipo	Descrição
public	Qualquer objeto ou método pode chamar essa função
external	Essa função só pode ser chamada por elementos externos
private	Apenas métodos pertencentes ao contrato podem chamar essa função
internal	Similar ao *private*, porém contratos derivados a partir de herança também podem utilizar essa função
view	Essa função retorna dados sem modificar o estado. Além disso, a utilização dessa função não consome *gas*
pure	Essa função não irá ler nem modificar o estado do contrato
payable	Essa função pode requerer o pagamento de *ether* para sua utilização

Fonte: (SOLIDITY. . . ,)

práticas, a utilização de tipos de funções é recomendada em todos os casos, mesmo quando o tipo pode ser determinado a partir do contexto. O objetivo dessa metodologia é obter clareza na intenção do código. A tabela 3 define e descreve os tipos de funções permitidos dentro da linguagem. O conjunto de todas as funções, exceto aquelas do tipo *private* ou *internal*, formam uma interface de comunicação utilizada para chamadas externas.

O ambiente de execução EVM inclui uma funcionalidade de registro de eventos. Eventos são utilizados para informar usuários externos que algo aconteceu na blockchain; os contratos em si, são incapazes de observar eventos. A linguagem Solidity permite se comunicar com essa funcionalidade de registro através do uso da *keyword* “Event”, usada como forma de interface.

4.2 PADRÕES DE DESIGN

O desenvolvimento de contratos inteligentes que funcionem de forma correta e eficiente não é uma tarefa fácil. Os desenvolvedores precisam antes de tudo levar em consideração a segurança, assim como aspectos do desenvolvimento de software como autorização, manutenção e controle. O nível de consumo de *gas* também está relacionado a esse problema, visto que a eficiência de combustível computacional é um fator crucial. Qualquer modificação na blockchain requer um gasto em *ether*, que está associado a um custo monetário.

Alguns dos exemplos de padrões de design, catalogados por Bartoletti e Pompianu

são (BARTOLETTI; POMPIANU, 2017):

- **Token**: Esse padrão é utilizado para distribuir bens fungíveis, na forma de moeda. Fungibilidade é um conceito econômico; elementos fungíveis podem ser trocados entre si, não havendo distinção entre suas unidades. *Tokens* podem representar uma variedade de bens, moedas, ações, bilhetes, ou qualquer coisa que possa ser transferível e contável. As implicações de ser *tokens* depende do protocolo e dos casos de uso que o *tokens* representa. *Tokens* podem ser utilizados para rastrear a propriedade de objetos físicos, como ouro e commodities, assim como objetos digitais (como criptomoedas).

- **Autorização**: Esse padrão é usado para restringir a execução de código de acordo com o endereço da conta que convocou o contrato. A maioria dos contratos costumam checar se o endereço invocador é o mesmo do dono do contrato, a fim de performar operações críticas como a autodestruição do contrato ou emissão de bens digitais.

- **Oráculo**: Alguns contratos podem necessitar de dados de fora da blockchain, como por exemplo o resultado de uma aposta ou a previsão do tempo. A rede Ethereum não permite que contratos acessem dados externos de forma direta, pois se permitisse, comprometeria o determinismo das computações, pois nós diferentes poderiam calcular diferentes resultados a partir de uma mesma transação. Oráculos funcionam como uma interface entre contratos e o mundo externo. Tecnicamente, funcionam como contratos normais, porém possuem seus estados atualizados a partir de transações. Esse estado pode ser então utilizado para alimentar outros contratos com informações externas. Na prática, ao invés de consultar a internet diretamente para obter esses dados, contratos devem consultar oráculos.

- **Terminação**: Visto que a blockchain é imutável, um contrato não pode ser deletado mesmo quando seu uso chega ao fim. Sendo assim, desenvolvedores devem prever formas de desativá-los no futuro, fazendo com o que embora os contratos continuem presentes na rede, os mesmos se tornem irresponsivos. Isso pode ser feito na forma de adição de métodos de auto-destruição. Geralmente, apenas o dono do contrato deve possuir permissão para tais ações.

Alguns padrões são tão importantes que se tornam *standards*. Na comunidade Ethereum, padrões oficiais e mudanças nos protocolos são inicialmente sugeridos a partir de EIPs (*Ethereum Improvement Proposals*). Depois de várias iterações de *feedback* e desenvolvimento, essas propostas podem ser elevadas à condição de ERC (*Ethereum Request for Comments*).

O padrão ERC mais utilizado e comum é o ERC-20, que define uma interface comum de métodos e eventos para a criação de tokens fungíveis (VOGELSTELLER, 2015). A utilização em larga escala deste padrão pela maioria dos contratos que implementam esse tipo de dado, resultam em um ecossistema de contratos e aplicações compatíveis. Qualquer novo contrato, ao implementar corretamente a interface proposta, garante uma interoperabilidade com carteiras e corretoras de imediato. Os códigos 2 e 3 demonstram as especificações desse padrão.

Código 2 – Especificação dos métodos para implementação de um *token* ERC20.

```
function name() public view returns (string)
function symbol() public view returns (string)
function decimals() public view returns (uint8)
function totalSupply() public view returns (uint256)
function balanceOf(address _owner) public view returns (uint256 balance)
function transfer(address _to, uint256 _value) public returns (bool success)
function transferFrom(address _from, address _to, uint256 _value) public returns
    (bool success)
function approve(address _spender, uint256 _value) public returns (bool success)
function allowance(address _owner, address _spender) public view returns (uint256
     remaining)
```

Código 3 – Especificação dos eventos para implementação de um *token* ERC20.

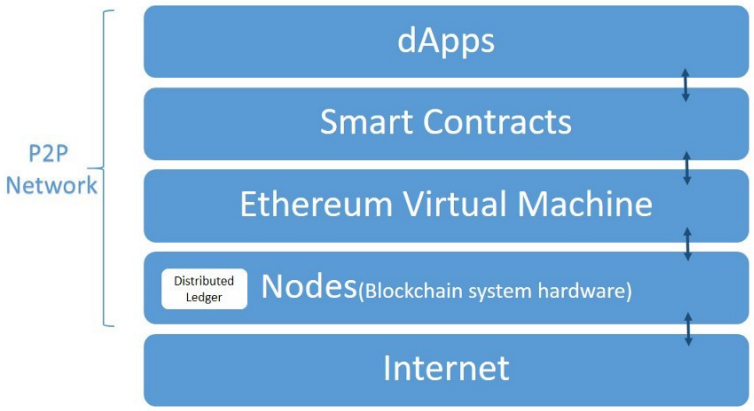
```
event Transfer(address indexed _from, address indexed _to, uint256 _value)
event Approval(address indexed _owner, address indexed _spender, uint256 _value)
```

5 APLICAÇÕES DESCENTRALIZADAS

Uma aplicação descentralizado (*DApp*) é uma aplicação construída combinando contratos inteligentes junto a uma interface web para usuários. Em Ethereum, todos os contratos são públicos tal como a rede, e são acessíveis e transparentes tal como uma API. DApps contrastam aplicativos tradicionais, que tipicamente executam a partir de servidores centralizados.

A estrutura de aplicações webs tradicionais é baseada em dois elementos: frontend e backend. Dispositivos e servidores se comunicam com o uso de mensagens codificadas e transmitidas através do protocolo HTTP. Quando se acessa uma rede social, tal como o Instagram, todo o conteúdo a ser exibido no browser na forma de interface gráfica (*frontend*) é construído a partir de dados armazenados nos servidores web do Facebook (*backend*). Os *DApps* são similares, porém seu *backend* é composto por um conjunto de contratos inteligentes armazenados na *blockchain*, fazendo uso de suas características distribuídas e imutáveis.

Figura 9 – Camadas de infraestrutura das aplicações descentralizadas.



Fonte: (XBT, 2018)

Contratos inteligentes são o coração das aplicações descentralizadas. Esses programas auto executáveis são usados para definir a lógica das aplicações através de um sistema de computação entre pares. Um contrato já é essencialmente um *DApp*, visto que ele por si só já fornece funcionalidades de computação distribuída. A introdução de uma interface web acessível busca levar esses serviços a um maior público, utilizando conceitos de interface e experiência de usuário da web moderna.

Algumas das vantagens dos serviços descentralizados em comparação com suas alternativas tradicionais são (ETHEREUM. . . ,):

- **Uptime**: Visto que contratos inteligentes residem dentro da *blockchain*, a rede Ethereum sempre garantirá a possibilidade de acesso à leitura, uso e escrita de contratos, independente da hora ou localização. Agentes mal intencionados são portante incapazes de realizar ataques de negação de serviço em aplicações descentralizadas individuais;
- **Privacidade**: Nenhum usuário necessita prover sua identidade para interagir com aplicações descentralizadas. Suas identidades são caracterizadas dentro da rede por um endereço ;
- **Imunidade a Censura**: Nenhuma entidade na rede é capaz de bloquear transações de usuários ou impedir a leitura de dados da *blockchain*;
- **Integridade dos Dados**: Dados armazenados na *blockchain* são imutáveis e indiscutíveis devido a conceitos criptográficos. Agentes mal intencionados não podem forjar nem manipular transações ou dados que já façam parte da rede;
- **Computação Confiável**: Contratos inteligentes podem ser analisados e auditados. Sua execução ocorre sempre de maneira deterministica, baseando-se no estado da rede e nos parâmetros de transação fornecidos.

Podemos contrastar com as seguintes desvantagens:

- **Manutenção**: Aplicativos descentralizados são mais difíceis de manter porque os contratos localizados na *blockchain* são imutáveis. É difícil realizar atualizações nas aplicações uma vez que elas estão ativas, sendo necessário o uso de padrões específicos de proejto;
- **Overhead**: Existe um grande *overhead* de performance que resulta em dificuldades na escalabilidade das aplicações. Para atingir o nível de segurança, integridade, transparência e confiabilidade que a rede aspira ter, cada nó executa e armazena todas as transações. Além disso, o algoritmo de consenso através da prova de trabalho é demorado e custoso.
- **Congestão de Rede**: Atualmente a rede processa em torno de 15 transações por segundo. Se as transações são requisitadas em taxas maiores que essa, o número de transações não confirmadas tende a subir rapidamente;

- **Centralização**: Soluções simples e agradáveis construídas a partir do Ethereum podem facilmente acabarem se tornando semelhantes a serviços centralizados. Alguns serviços podem armazenar chaves, senhas e outras informações sensíveis ou executar lógicas importantes em um servidor centralizado antes mesmo de escrever na *blockchain*. A centralização elimina muitas das vantagens que uma solução descentralizada possui sobre os modelos tradicionais.

De acordo com o artigo "The General Theory of Decentralized Applications" (JOHNSTON et al., 2014), um aplicativo pode ser considerado um DApp quando obedece a quatro critérios :

1. O aplicativo deve possuir código aberto, operar de maneira autônoma sem que alguma entidade controle a maioria de seus tokens. Adaptações ao aplicativo devem ocorrer mediante a consenso entre seus usuários.
2. Dados relativos ao aplicativo devem ser armazenados em uma *blockchain* pública.
3. O aplicativo utiliza algum token criptográfico, necessário para obter acesso ao *DApp* e para recompensar contribuições de seus usuários.
4. O aplicativo deve gerar tokens de acordo com algum algoritmo criptográfico que atuam como prova do valor da contribuição de usuários do sistema.

A partir desses quatro critérios, é possível identificar que *tokens* são uma parte fundamental do modelo econômico e estrutural das aplicações descentralizadas. *Tokens* são usados para mediar os interesses de usuários e desenvolvedores, além de pagar os custos de manutenção do sistema. Fundamentalmente, a utilização de serviços da rede devem ser custeados a partir de transações e emissões de *tokens*. Idealmente, um aplicativo descentralizado seguindo os quatro critérios descritos não necessita da manutenção e governança dos desenvolvedores originais do projeto. Devido a natureza do modelo, os lucros e os custos de organizações descentralizadas são divididos entre todos os participantes (CAI et al., 2018).

5.1 WEB3

A web moderna, comumente chamada de *web* 2.0, é drasticamente diferente do que era 15 anos atrás. A internet em seus primeiros anos era mantida por desenvolvedores que criaram websites contendo informações na forma de imagem ou texto, semelhante ao compartilhamento de documentos. Tipicamente, servidores centralizados e monolíticos distribuiam conteúdos estáticos quando usuários acessavam seus endereços. Essencialmente, era uma internet baseada em leitura.

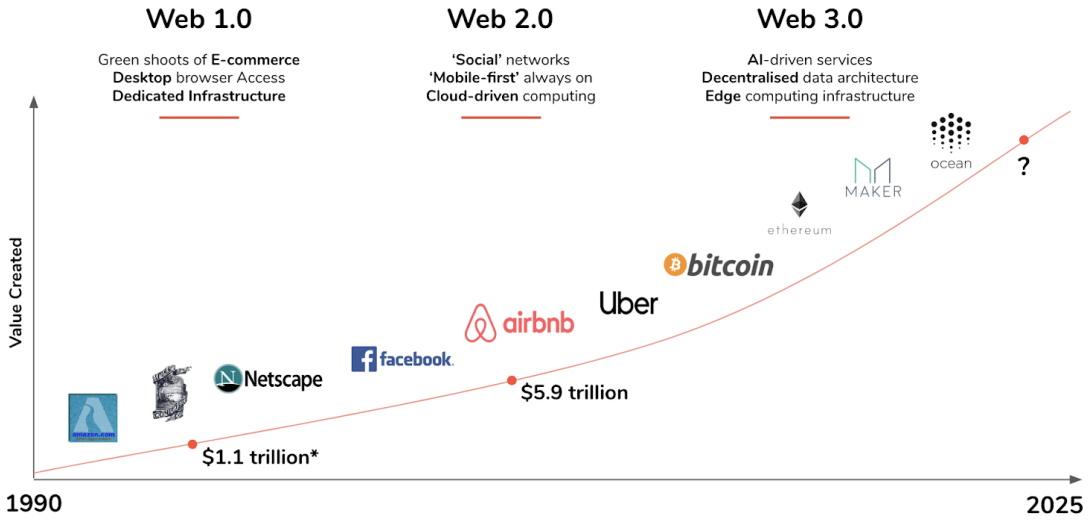
A *web* 2.0 introduziu o conceito de uma internet interativa e social, e essa mudança começou a partir da forma de distribuição do conteúdo. Websites estáticos passaram a se tornar dinâmicos, alterando o conteúdo a partir do contexto do usuário com o uso de ferramentas de backend como PHP e ASP.NET. Blogs e fóruns se popularizaram a partir da interação de usuários através de comentários e posts, que acabou resultando no surgimento das primeiras redes sociais.

Em 2008, com a introdução dos dispositivos móveis e da evolução na complexidade dos navegadores de internet, as páginas dinâmicas começaram a ser trocadas por aplicações; softwares rodando do lado do cliente, sendo responsáveis por criar uma interface gráfica e fornecer um serviço. Os servidores passaram a focar na distribuição de dados através de uma API, distribuindo conteúdo para aplicações móveis e scripts de navegadores, independente das características físicas dos hardwares utilizados pelos usuários.

A internet se tornou bem mais heterogênea, distribuída e interativa, focada na geração de conteúdo pelos seus usuários e no fornecimento de serviços sob várias formas. Essa mudança de paradigma resultou em uma explosão de companhias bem sucedidas baseadas principalmente no vale do silício, como Facebook, Uber e Amazon. Hoje, as quatro empresas com maior valor de mercado no mundo oferecem produtos e serviços relacionados à tecnologia *web*.

A Web3 repropõe a internet em dois aspectos: infraestrutura e monetização. Contratos inteligentes permitem a substituição do modelo de microserviços executados em servidores centralizados. Enquanto os mecanismos econômicos da Web2 incentivam a concentração de dados em silos de informação na forma de *data centers*, os mecanismos da Web3 incentivam o espalhamento e a negociação dos dados através da rede.

Figura 10 – Evolução do paradigma da internet.



Fonte: (VENTURES, 2020)

6 PROVA DE CONCEITO: DEECOIN

O padrão de *tokens* fungíveis é o constructo mais importante na modelagem de contratos inteligentes. São utilizados para representar moedas, itens, ações, *vouchers* e objetos similares, digitais ou físicos. Devido a sua importância, o padrão ERC20 foi definido com o objetivo de garantir que as diferentes implementações de *tokens*, com diversas finalidades, possuam compatibilidade a nível de interface. O objetivo desse padrão é garantir uma interoperabilidade entre um grande ecossistema de contratos e aplicações descentralizadas.

Quando um *token* é implementado utilizando a interface de funções e eventos especificada pela ERC20, ao ser lançada, esse *token* já será compativel com uma gama de serviços disponíveis na rede, podendo até mesmo ser comercializado e trocado por outros *token*.

Como prova de conceito, foi definido o *design* e desenvolvimento de uma moeda compatível com esse padrão. Utilizando o ambiente de desenvolvimento Remix, esta moeda foi implementada com as seguintes caracteristicas:

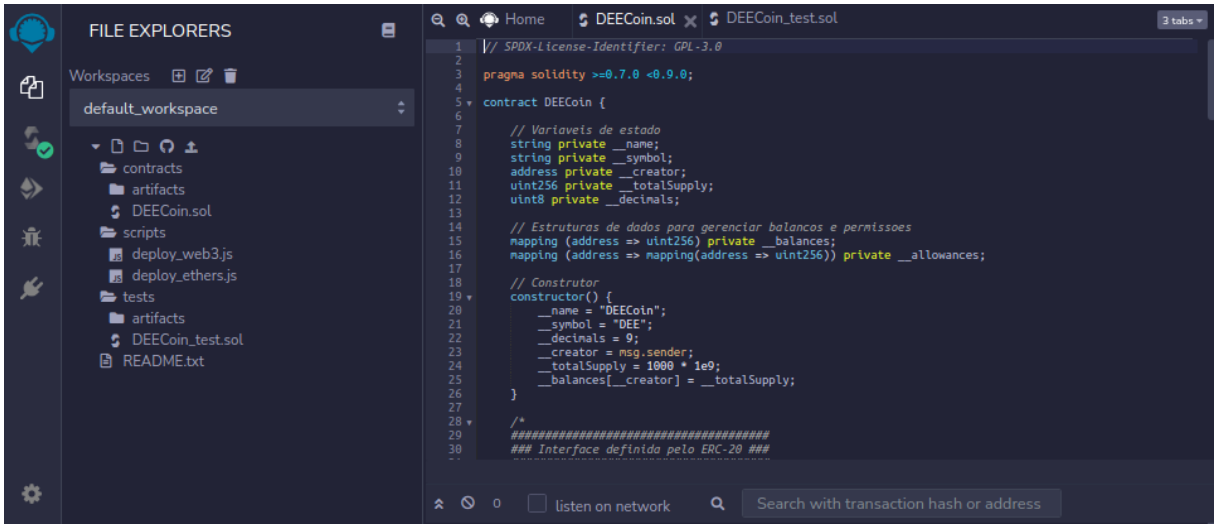
- Nome: DEECoin;
- Símbolo: DEE;
- Oferta inicial: 1000 moedas;
- Minima unidade: 10^{-9} DEE;

Comumente, a maioria das *tokens* implementadas possuem mecanismos automáticos de manipulação da oferta monetária, com os mais diversos objetivos. Por decisão de design e simplificação de implementação, a emissão e destruição de moedas do *token* descrito nesse capitulo é absolutamente controlada pela conta responsável pela sua criação. O código do contrato implementado pode ser encontrado no anexo .

O contrato foi compilado utilizando o compilador oficial, implementado em NodeJS, em sua versão 0.8.7. O código binário do contrato foi então executado em uma *blockchain* local, seguindo o protocolo definido pelo *hard fork* London. A partir do uso da interface fornecida pelo ambiente de desenvolvimento, demonstrado na figura 12, foi realizado os primeiros testes de validação.

Uma bateria de testes de unidade foi desenvolvida a partir do encapsulamento da DEECoin dentro de um contrato de testes chamado de DEECoinTest. Cada unidade de teste é definido sob uma função pública e possuem lógica para testar as interfaces GET,

Figura 11 – Ambiente de desenvolvimento Remix.



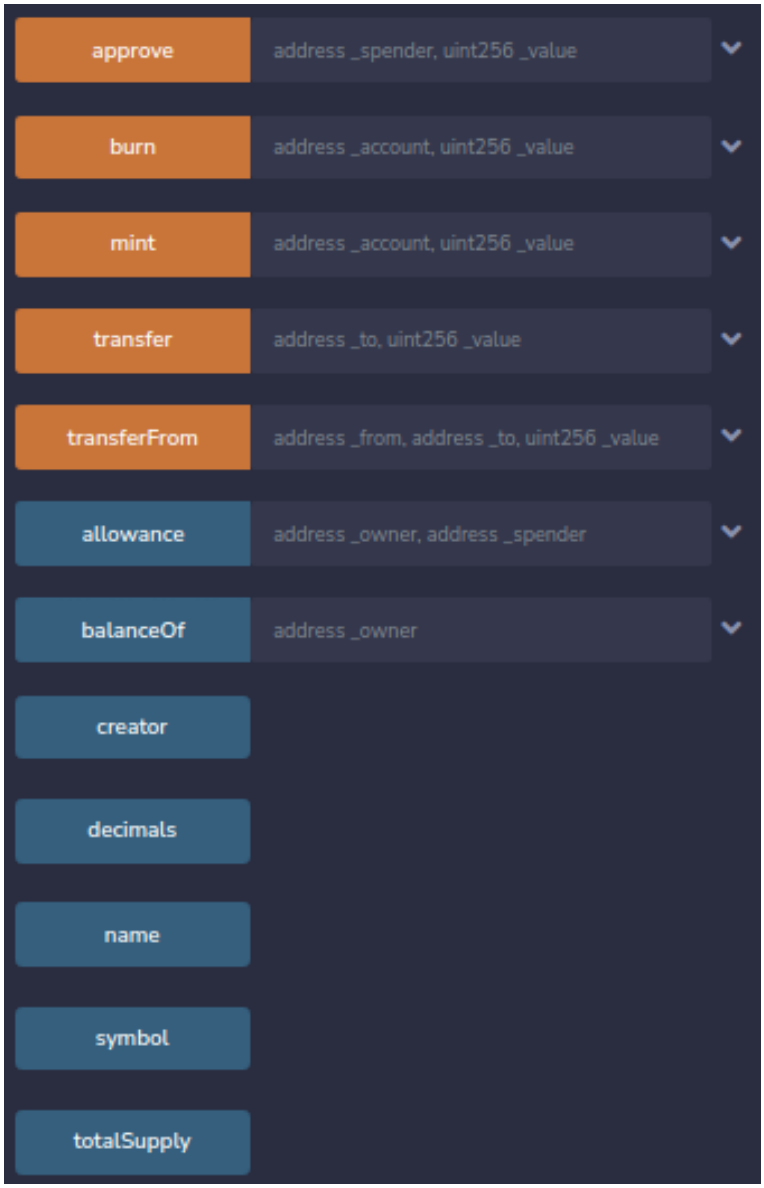
Fonte: Autor

assim como transferência, criação e destruição de *tokens*. Seguindo o mesmo fluxo descrito anteriormente, o contrato de testes demonstrado no anexo B foi lançado e validado junto a rede local.

O custo total do lançamento do código EVM do contrato DEECoin na rede de testes foi de exatas 912428 unidades de *gas*. Considerando o preço do *gas* na faixa dos 60 *Gwei*, e o preço do *ether* na faixa dos 3500 dólares, podemos estimar que o custo do lançamento do contrato na rede principal será de:

$$\text{Custo} = 912428 \cdot (60 \cdot 10^{-9}) \cdot \$3500 = \$191.61.$$

Figura 12 – Interface dos métodos públicos da DEECoin.



Fonte: Autor

7 CONSIDERAÇÕES FINAIS

Ethereum foi construído com a ideia de que muitas blockchains, moedas e tipos de rede podem existir, e que portanto deve existir um conjunto de protocolos para que essas redes mistas possam se comunicar. Com uma perspectiva diferente do criador do Bitcoin, os criadores da rede Ethereum tomaram a posição de que, criptomoedas, se existirem no futuro, não farão parte de um sistema descentralizado único. Ao invés disso, essa rede tomaria a forma de uma rede de sistemas descentralizados independentes, permitindo que diferentes ideias criptográficas de valor, com diversos propósitos e interpretações diferentes possam facilmente ser definidas, criadas e transacionadas.

Os problemas que a rede enfrenta no momento são frutos de um cedo estágio de desenvolvimento. Ethereum, seus contratos e aplicações, foram concebidos nos últimos anos e ainda possuem uma longa jornada pela frente. Muitas das soluções possíveis que essas ferramentas podem oferecer ainda estão a ser inventadas.

É verdade que Bitcoin e Ethereum adicionam uma complexidade econômica ao ato de escrever programas de computadores; mas eles também são simples em alguns aspectos. Trabalhar com redes descentralizadas é similar ao trabalho com computadores na década de 70; são caros e necessitam de compartilhamento de recursos para serem melhor aproveitados. A história nos mostra o potencial que pode ser atingido a partir de soluções incompletas, porém engenhosas e com visão de longo prazo.

REFERÊNCIAS

ANTONOPOULOS, A. M. *Mastering Bitcoin: Programming the open blockchain*. [S.l.]: "O'Reilly Media, Inc.", 2017. 13

BARTOLETTI, M.; POMPIANU, L. An empirical analysis of smart contracts: platforms, applications, and design patterns. In: SPRINGER. *International conference on financial cryptography and data security*. [S.l.], 2017. p. 494–509. 31

BUTERIN, V. et al. A next-generation smart contract and decentralized application platform. *white paper*, v. 3, n. 37, 2014. 15, 19, 20

CAI, W. et al. Decentralized applications: The blockchain-empowered software system. *IEEE Access*, v. 6, p. 53019–53033, 2018. 35

ETHEREUM Development Documentation. Disponível em: <https://ethereum.org/en/developers/docs/>. 15, 21, 23, 24, 34

JOHNSTON, D. et al. *The general theory of decentralized applications*. [S.l.]: DApps, 2014. 35

NAKAMOTO, S. Bitcoin: A peer-to-peer electronic cash system. *Decentralized Business Review*, p. 21260, 2008. 11

SOLIDITY Documentation. Disponível em: <https://docs.soliditylang.org/en/v0.3.5/index.html>. 28, 29, 30

SZABO, N. Formalizing and securing relationships on public networks. *First monday*, 1997. 27

TANI, T. *Ethereum EVM illustrated*. 2018. Disponível em: <https://github.com/takenobu-hs/ethereum-evm-illustrated>. 16, 17, 20

VENTURES, F. *What Is Web 3.0 Why It Matters*. Fabric Ventures, 2020. Disponível em: <https://medium.com/fabric-ventures/what-is-web-3-0-why-it-matters-934eb07f3d2b>. 36

VOGELSTELLER, V. B. F. *EIP-20: Token Standard*. 2015. Disponível em: <https://eips.ethereum.org/EIPS/eip-20>. 32

WOOD, G. et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, v. 151, n. 2014, p. 1–32, 2014. 19, 20

XBT. *What is an Ethereum (ETH) Virtual Machine (EVM)?* 2018. Disponível em: <https://xbt.net/blog/ethereum-blog/what-is-an-ethereum-eth-virtual-machine-evm/>. 33

Anexos

ANEXO A – deecoin.sol

Código 4 – Código do contrato DEECoin escrito na linguagem Solidity

```
// SPDX-License-Identifier: GPL-3.0

pragma solidity >=0.7.0 <0.9.0;

contract DEECoin {

    // Variaveis de estado
    string private __name;
    string private __symbol;
    address private __creator;
    uint256 private __totalSupply;
    uint8 private __decimals;

    // Estruturas de dados para gerenciar balancos e permissoes
    mapping (address => uint256) private __balances;
    mapping (address => mapping(address => uint256)) private __allowances;

    // Construtor
    constructor() {
        __name = "DEECoin";
        __symbol = "DEE";
        __decimals = 9;
        __creator = msg.sender;
        __totalSupply = 1000 * 1e9;
        __balances[__creator] = __totalSupply;
    }

    /*
    ########################################
    ### Interface definida pelo ERC-20 ###
    ########################################
    */

    function name() public view returns (string memory){
        return __name;
    }

    function symbol() public view returns (string memory){
        return __symbol;
    }

    function decimals() public view returns (uint8){
        return __decimals;
    }

    function totalSupply() public view returns (uint256){
        return __totalSupply;
    }

    function balanceOf(address _owner) public view returns (uint256 balance){
```

```
51          return __balances[_owner];
52      }
53
54      function transfer(address _to, uint256 _value) public returns (bool balance){
55          address _from = msg.sender;
56          __transferProc(_from, _to, _value);
57
58          return true;
59      }
60
61
62      function transferFrom(address _from, address _to, uint256 _value) public
            returns (bool success){
63          __transferProc(_from, _to, _value);
64
65          uint256 remainingAllowance = __allowances[_from][_to];
66          require(remainingAllowance >= _value, "ERC-20: transfer value is greater
                than allowance");
67          __approveProc(_from, _to, remainingAllowance - _value);
68
69          return true;
70      }
71
72      function approve(address _spender, uint256 _value) public returns (bool
            success){
73          address _owner = msg.sender;
74          __approveProc(_owner, _spender, _value);
75
76          return true;
77      }
78
79      function allowance(address _owner, address _spender) public view returns (
            uint256 remaining){
80          return __allowances[_owner][_spender];
81      }
82
83      /*
84      ########################################
85      ### Eventos definidos pelo ERC-20 ####
86      ########################################
87      */
88
89      event Transfer(address indexed _from, address indexed _to, uint256 _value);
90
91      event Approval(address indexed _owner, address indexed _spender, uint256
            _value);
92
93      /*
94      ########################################
95      ######### Metodos auxiliares #########
96      ########################################
97      */
98
99      // Metodo de transferencia entre contas
100     function __transferProc (address _from, address _to, uint256 _value) internal
            {
```

```
101         require(_from != address(0), "ERC-20: transfer from the zero addres is
                not possible");
102         require(_to != address(0), "ERC-20: transfer to the zero addres is not
                possible");
103
104         require(_value > 0, "ERC-20: transfer of zero values is not possible");
105
106         uint256 fromBalance = __balances[_from];
107         require(fromBalance >= _value, "ERC-20: transfer value is bigger than the
                 sender's balance");
108
109         __balances[_from] = fromBalance - _value;
110         __balances[_to] += _value;
111
112         emit Transfer(_from, _to, _value);
113     }
114
115     // Metodo para permiss o de transferencia
116     function __approveProc (address _owner, address _spender, uint256 _value)
            internal {
117         require(_owner != address(0), "ERC-20: approval from the zero addres is
                not possible");
118         require(_spender != address(0), "ERC-20: approval to the zero addres is
                not possible");
119
120         __allowances[_owner][_spender] = _value;
121
122         emit Approval(_owner, _spender, _value);
123     }
124
125     /*
126     ########################################
127     ######## Interface customizada ########
128     ########################################
129     */
130
131     function creator () public view returns (address){
132         return __creator;
133     }
134
135     function mint (address _account, uint256 _value) public {
136         require(_account != address(0), "ERC-20: mint to the zero address is not
                possible");
137         require(_value > 0, "ERC-20: mint of zero values is not possible");
138         require(__creator == msg.sender, "ERC-20: permission denied");
139
140         __totalSupply += _value;
141         __balances[_account] += _value;
142
143         emit Transfer(address(0), _account, _value);
144     }
145
146     function burn (address _account, uint256 _value) public {
147         require(_account != address(0), "ERC-20: burn in the zero address is not
                possible");
148         require(_value > 0, "ERC-20: burn of zero values is not possible");
149         require(__creator == msg.sender, "ERC-20: permission denied");
```

```
150
151         uint256 accountBalance = __balances[_account];
152         require(accountBalance >= _value, "ERC-20: burn value is bigger than the
                account's balance");
153
154         __totalSupply -= _value;
155         __balances[_account] -= _value;
156
157         emit Transfer(_account, address(0), _value);
158     }
159 }
```

ANEXO B – deecoin_test.sol

Código 5 – Código do contrato de testes DEECoinTest escrito na linguagem Solidity

```
// SPDX-License-Identifier: GPL-3.0

pragma solidity >=0.7.0 <0.9.0;
import "remix_tests.sol";
import "../contracts/deecoin.sol";

contract DEECoinTest {

    address __creatorAddress;
    address __testAddress;

    uint256 __rndNonce;

    DEECoin ContractToTest;

    function beforeAll () public {
        ContractToTest = new DEECoin();
        __creatorAddress = address(this);
        __testAddress = 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4;
        __rndNonce = 0;
    }

    function checkCreator () public {
        Assert.equal(ContractToTest.creator(), __creatorAddress,
                    "Endereco do criador deveria ser o mesmo do contrato
                        DEECoinTest");
    }

    function checkName () public {
        Assert.equal(ContractToTest.name(), "DEECoin",
                    "Nome da moeda deveria ser DEECoin");
    }

    function checkSymbol () public {
        Assert.equal(ContractToTest.symbol(), "DEE",
                    "Simbolo da moeda deveria ser DEE");
    }

    function checkDecimals () public {
        Assert.equal(ContractToTest.decimals(), 9,
                    "Numero de decimais deveria ser 9");
    }

    function checkTransfer () public {
        uint256 creatorBalance = ContractToTest.balanceOf(__creatorAddress);
        uint256 testBalance = ContractToTest.balanceOf(__testAddress);

        uint256 transferAmount = __rndGen() % creatorBalance;

        ContractToTest.transfer(__testAddress, transferAmount);
```

```
50
51          uint256 newCreatorBalance = ContractToTest.balanceOf(__creatorAddress);
52          uint256 newTestBalance = ContractToTest.balanceOf(__testAddress);
53
54          Assert.equal(newCreatorBalance, creatorBalance - transferAmount,
55                       "O balanco da conta criadora nao esta correto apos a
                             transferencia");
56
57          Assert.equal(newTestBalance, testBalance + transferAmount,
58                       "O balanco da conta de testes nao esta correto apos a
                             transferencia");
59      }
60
61      function checkApproval () public {
62          uint256 creatorBalance = ContractToTest.balanceOf(__creatorAddress);
63          uint256 creatorAllowance = ContractToTest.allowance(__creatorAddress,
                __testAddress);
64
65          uint256 allowanceAmount = __rndGen() % creatorBalance;
66
67          ContractToTest.approve(__testAddress, allowanceAmount);
68
69          uint256 newCreatorAllowance = ContractToTest.allowance(__creatorAddress,
                __testAddress);
70
71          Assert.equal(newCreatorAllowance, creatorAllowance + allowanceAmount,
72                       "O balanco da conta criadora nao esta correto apos a
                             transferencia");
73      }
74
75      function checkMint () public {
76          uint256 creatorBalance = ContractToTest.balanceOf(__creatorAddress);
77          uint256 totalSupply = ContractToTest.totalSupply();
78
79          uint256 mintAmount = __rndGen() % creatorBalance;
80
81          ContractToTest.mint(__creatorAddress, mintAmount);
82
83          uint256 newCreatorBalance = ContractToTest.balanceOf(__creatorAddress);
84          uint256 newTotalSupply = ContractToTest.totalSupply();
85
86          Assert.equal(newCreatorBalance, creatorBalance + mintAmount,
87                       "O balanco da conta criadora nao esta correto apos a emissao
                             ");
88          Assert.equal(newTotalSupply, totalSupply + mintAmount,
89                       "A oferta total de moedas nao esta correto apos a emissao");
90      }
91
92      function checkBurn () public {
93          uint256 creatorBalance = ContractToTest.balanceOf(__creatorAddress);
94          uint256 totalSupply = ContractToTest.totalSupply();
95
96          uint256 burnAmount = __rndGen() % creatorBalance;
97
98          ContractToTest.burn(__creatorAddress, burnAmount);
99
100         uint256 newCreatorBalance = ContractToTest.balanceOf(__creatorAddress);
```

```
101         uint256 newTotalSupply = ContractToTest.totalSupply();
102
103         Assert.equal(newCreatorBalance, creatorBalance - burnAmount,
104                     "O balanco da conta criadora nao esta correto apos a queima
                        ");
105         Assert.equal(newTotalSupply, totalSupply - burnAmount,
106                     "A oferta total de moedas nao esta correto apos a queima");
107     }
108
109     function __rndGen () internal returns (uint256) {
110         uint256 rndNum = uint(keccak256(abi.encodePacked(__rndNonce)));
111         __rndNonce += 1;
112         return rndNum;
113     }
114 }
```