UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
DEPARTAMENTO DE ENGENHARIA ELÉTRICA E ELETRÔNICA

Protocolo para *Marketplace* de *Tokens* em Redes Descentralizadas

Trabalho de Conclusão de Curso submetido ao curso de Engenharia Eletrônica da Universidade Federal de Santa Catarina como requisito para obtenção do título de Bacharel em Engenharia Eletrônica.

Claudio Felipe Carvalho Vilas Boas

Orientador: Eduardo Luiz Ortiz Batista
Co-orientador: Walter Antônio Gontijo

Florianópolis
Março de 2022

Ficha de identificação da obra elaborada pelo autor,
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Carvalho Vilas Boas, Claudio Felipe
Protocolo para Marketplace de Tokens em Redes Descentralizadas / Claudio Felipe Carvalho Vilas Boas ; orientador, Eduardo Luiz Ortiz Batista, coorientador, Walter Antônio Gontijo, 2022.
103 p.

Trabalho de Conclusão de Curso (graduação) - Universidade Federal de Santa Catarina, Centro Tecnológico, Graduação em Engenharia Eletrônica, Florianópolis, 2022.

Inclui referências.

1. Engenharia Eletrônica. 2. Blockchain. 3. Ethereum Virtual Machine. 4. Smart Contracts. 5. Marketplace de NFT. I. Ortiz Batista, Eduardo Luiz. II. Gontijo, Walter Antônio. III. Universidade Federal de Santa Catarina. Graduação em Engenharia Eletrônica. IV. Título.

Claudio Felipe Carvalho Vilas Boas

Protocolo para *Marketplace* de *Tokens* em Redes Descentralizadas

Este Trabalho Conclusão de Curso foi julgado adequado para obtenção do Título de Bacharel em Engenharia Eletrônica e aprovado em sua forma final pelo Curso de Engenharia Eletrônica.

Florianópolis, 28 de marco de 2022

Documento assinado digitalmente
Fernando Rangel de Sousa
Data: 13/04/2022 12:07:18-0300
CPF: 884.649.114-91
Verifique as assinaturas em https://v.ufsc.br

Prof. Fernando Rangel de Sousa, PhD.
Coordenador do Curso

Banca Examinadora:

Documento assinado digitalmente
Eduardo Luiz Ortiz Batista
Data: 13/04/2022 10:20:07-0300
CPF: 036.521.889-85
Verifique as assinaturas em https://v.ufsc.br

Prof. Eduardo Luiz Ortiz Batista, PhD.
Orientador, UFSC

Documento assinado digitalmente
WALTER ANTONIO GONTIJO
Data: 13/04/2022 14:17:39-0300
CPF: 536.287.996-00
Verifique as assinaturas em https://v.ufsc.br

Walter Antônio Gontijo, MSc.
Co-orientador, UFSC

Documento assinado digitalmente
Richard Demo Souza
Data: 14/04/2022 08:59:22-0300
CPF: 004.267.379-89
Verifique as assinaturas em https://v.ufsc.br

Prof. Richard Demo Souza, PhD.
Avaliador, UFSC

DocuSigned by:
Natan Votre
9A1FF9A2B338470...

Eng. Natan Votre
Avaliador, UFSC

Agradecimentos

Deixo aqui em especial os agradecimentos a minha família pelo constante apoio e motivação para seguir em frente e confrontar os desafios do cotidiano, em especial ao meu pai Jefferson e ao meu avô Claudio que sempre estiveram ao meu lado. Também à Marina por todo o carinho, amor e advertências sem os quais esse trabalho muito dificilmente teria prosseguido.

Ao LINSE que me proporcionou muitos conhecimentos e técnicas para conseguir enfrentar problemas complexos de engenharia. Em especial ao Walter e ao professor Eduardo que se aventuraram comigo na escrita deste trabalho me orientando e conduzindo constantemente.

Também a diversos amigos que me apoiaram e ensinaram ao longo da graduação sempre me puxando para o próximo nível. Em especial à Syscoin, ao Kaue e aos sócios na Pollum, que junto comigo, propuseram e deram a oportunidade para o desenvolvimento de aplicações novas com Blockchain e sem eles esse trabalho também não teria sido possível.

"*[Blockchain] is the biggest opportunity set we can think of over the next decade or so.*"
(Robert Greifeld, Nasdaq Chief Executive)

RESUMO

Este trabalho é dedicado ao desenvolvimento de um protocolo para *marketplace* de *NFTs* em redes descentralizadas para *blockchains* compatíveis com *EVMs*. A *blockchain* alvo foi a *Polygon*. O *marketplace* desenvolvido é capaz de efetuar trocas de *tokens* de maneira automática e segura entre dois participantes de uma *blockchain*. Nesse contexto são desenvolvidos todos os blocos necessários para uma *exchange* com um *orderbook offchain*. O protocolo desenvolvido neste trabalho passou por auditorias, todas as regras de negócio foram validadas e estão disponíveis na aplicação *web Luxy*.

Palavras-Chave: *Polygon*, *Ethereum*, *Blockchain*, *Marketplace de NFT*, *Tokens*, *Protocolos descentralizados*.

ABSTRACT

This work is dedicated to the development of a protocol for NFT marketplaces in decentralized networks for blockchains compatible with EVMs. The target blockchain was Polygon. The developed marketplace is able to exchange tokens automatically and securely between two participants in a blockchain. In this context, all the blocks necessary for an exchange with an offchain orderbook were developed. The protocol developed in this work underwent audits, all business rules were validated and are available for use at Luxy web application.

Keywords: Polygon, Ethereum, Blockchain, NFT Marketplace, Tokens, Decentralized Protocols.

Lista de Figuras

Listagens

Sumário

CAPÍTULO 1

Introdução

Os NFTs (*non-fungible tokens*) são elementos digitais criptográficos que representam o direito de posse a algo único, normalmente um conteúdo digital ou, em alguns casos, conteúdos ou objetos físicos reais. Os *marketplaces* de NFTs, plataformas que permitem a comercialização dos NFTs, obtiveram um crescimento exponencial desde sua criação em 2014 [1]. No começo de 2021 os NFTs atingiram uma capitalização de mercado de aproximadamente 7 biliões de dólares, de acordo com um estudo do banco *JPMorgan* [2]. O comércio de NFTs foi muito acelerado ao longo da pandemia do COVID-19, sendo que casas de leilão de arte tradicionais como a *Sotheby's* [3], *Christie's*[4] e a famosa feira *Hamptons Fine Arts* [5] começaram a digitalizar suas obras físicas, leiloar obras exclusivamente digitais e criar *marketplaces* digitais para a venda e compra de itens não fungíveis online. Além do mundo da arte, muitos outros *marketplaces* têm ganhado espaço no mercado. A maior plataforma deste tipo é o *OpenSea* [6], um mercado de colecionáveis que indexa a maioria dos NFTs disponíveis em diferentes *blockchains*.

O *OpenSea* movimenta um volume considerável, efetuando milhões de negociações de NFTs todos os dias [7]. Por exemplo, ao longo do mês de agosto de 2021, essa plataforma atingiu um pico no volume de negociações de 3.4 biliões de dólares [8]. A procura por *NFTS* é volátil e não previsível, mas vem crescendo como constatou *JPMorgan*:

> "Ao criar mercados para ativos ilíquidos, como arte digital, colecionáveis, música, jogos e outros ativos, o universo NFT certamente continuará a crescer fortemente nos próximos anos porque ajuda a resolver o problema de injetar liquidez em ativos naturalmente ilíquidos, como o de colecionáveis"[2].

Como essa já indica, esses *marketplaces* podem comercializar uma infinidade de diferentes ativos não fungíveis em suas operações diárias, apesar da maior divulgação dos casos referentes a arte digital pelos valores envolvidos nessas transações [9].

Para entender como diversos ativos ilíquidos são comercializado dentro dos *marketplaces* deve-se primeiro compreender o que é um NFT. De acordo com o *Investopedia*:

> "*Tokens* não fungíveis ou *NFTs* são ativos criptográficos em blockchain com códigos de identificação exclusivos e metadados que os distinguem uns dos outros. Ao contrário das criptomoedas, eles não podem ser negociados ou trocados em equivalência. Isso difere de *tokens* fungíveis como criptomoedas, que são idênticos entre si e, portanto, podem ser usados como um meio para transações comerciais"[10].

Ou seja, um NFT é basicamente um registro criptográfico gerado com uma assinatura única de um usuário com uma carteira em alguma *blockchain* que garante posse sobre o item. Nos metadados, tem-se a informação sobre o tipo de conteúdo sendo assegurado pelo registro, seja a *skin* de um jogo, uma arte digital, uma arte física de alguma casa de leilão, propriedade e etc. Esse registro pode ser trocado por alguma criptomoeda ou dinheiro de forma eficiente, rápida e sem intermédio de pessoas dentro de um *marketplace* integrado com a *blockchain* na qual o registro do item foi feito. Esses *marketplaces* são descentralizados e funcionam em tempo integral(24/7), o que é possível graças a tecnologia de *blockchain* que permite o desenvolvimento de protocolos com funções determinísticas e seguras para este tipo de transação.

Esses protocolos que possibilitam a operação em *blockchain* de um *marketplace* são escritos com *smart contracts* ou contratos inteligentes.

"Contratos inteligentes são simplesmente programas armazenados em uma blockchain que são executados quando condições predeterminadas são atendidas. Eles normalmente são usados para automatizar a execução de um acordo, de modo que todos os participantes possam ter a certeza imediata do resultado, sem o envolvimento de qualquer intermediário ou perda de tempo. Eles também podem automatizar um fluxo de trabalho, disparando a próxima ação quando as condições forem atendidas"[11].

Esses contratos são escritos como código e publicados em alguma *blockchain* se tornando indestrutíveis, ou seja não é possível apagá-los e nem bloquear as chamadas das funções definidas, enquanto a rede na qual foram armazenados exista. A *blockchain* é sempre atualizada após a conclusão de uma transação, no caso de um contrato o *output* de resultado da função chamada, garantindo que a transação não pode ser alterada.

Os contratos que estão armazenados na *blockchain* na qual foram publicados são escritos e definidos pelas mesmas regras de programação que qualquer outro código, isto é, utilizam *loops* como *for* e *while*, operações ternárias e condicionais, possuem operações aritméticas, definição de tipos(*int*, *string* e etc) e estruturas de dados. No entanto, não é possível compilar ou escrever códigos em linguagens convencionais para a máquina virtual do *Ethereum* ou de outras *blockchains*, como a Solana [12], uma vez que a estrutura de *assembly* e *OP_CODES* dessas máquinas virtuais é diferente que a de um processador (ARM, Intel e etc) ou de uma máquina virtual do Java. Para a escrita e produção desses contratos foram desenvolvidos compiladores, *frameworks* e linguagens próprias. Dentre essas linguagens a primeira a ser criada foi *Solidity* junto do compilador *Solc*, que por serem as primeiras contam com muito suporte *online*. Essas duas ferramentas permitem gerar o *bytecode* que a *EVM* consegue interpretar. Considerando o *OpenSea*, o *core* de sua infraestrutura é um protocolo de *marketplace* para comercializar *NFTS* escrito em *Solidity* e publicado em algumas *blockchains* compatíveis com a máquina virtual do *Ethereum* (*EVM*).

No presente Trabalho de Conclusão de Curso (TCC), pretende-se desenvolver um protocolo de *marketplace* para NFTs, a partir de protocolos bem estabelecidos como os utilizados pelo *OpenSea*. O protocolo desenvolvido será publicado nas redes de teste e a oficial da *Polygon* (os motivos para a escolha da rede ficam evidentes no Capítulo 4). Espera-se que o protocolo desenvolvido customize algumas regras de negócio já existentes nos protocolos já estabelecidos de forma a permitir maior flexibilidade para os usuários.

1.1 Objetivos

1.1.1 Objetivo geral

Especificamente, o objetivo geral deste trabalho é a implementação dos *smart contracts* de um protocolo para um *marketplace* descentralizado que permite a comercialização de itens não fungíveis(*NFTS*) em *blockchains* compatíveis com *EVMS* (Ethereum Virtual Machines).

1.1.2 Objetivos específicos

- Analisar protocolos de *marketplaces* descentralizados.
- Implementar um protocolo de *marketplace* em *solidity*.
- Testar a implementação do protocolo do *marketplace*.
- Publicar o protocolo implementado na rede da *Polygon*.
- Integrar o protocolo implementado em uma aplicação *Web* de *marketplace* de *NFTs*.
- Avaliar os resultados do protocolo desenvolvido ao longo deste trabalho.

1.1.3 Estrutura do Trabalho

Este trabalho está estruturado da seguinte forma. Ao longo do segundo capítulo é apresentada uma visão geral de sua fundamentação teórica, além das técnicas e ferramentas usadas em sua implementação. No terceiro capítulo, discorre-se sobre o desenvolvimento do projeto e soluções encontradas com as técnicas e ferramentas disponíveis. No quarto capítulo, tem-se uma avaliação dos resultados obtidos e comparações com plataformas concorrentes de mesmo estilo. Por fim, no quinto capítulo são apresentadas as conclusões e visão de futuro do projeto.

CAPÍTULO 2

Fundamentação Teórica

Neste capítulo é apresentada a fundamentação teórica para o desenvolvimento do presente trabalho. Inicialmente, são apresentadas noções gerais de *Blockchains*, seus tipos, estruturas básicas de bloco e algoritmos de consenso. Na sequência, noções sobre o funcionamento de máquinas virtuais, mais especificamente, os modelos de *EVM* (*Ethereum Virtual Machine*) que por serem máquinas de *turing* completas [13] permitem a execução de qualquer código. Também são brevemente apresentadas as especificações e diferenças da rede *Polygon*, rede escolhida para implementação do *marketplace* de NFTs proposto neste trabalho. Posteriormente, são apresentados os itens que compõem um *marketplace* de NFTs, os protocolos utilizados como referência para a implementação deste trabalho. Por fim as ferramentas utilizadas para o desenvolvimento e testes do protocolo de *marketplace* para NFTs proposto.

2.1 *Blockchain*

A tecnologia de *blockchain* emergiu como uma das maiores inovações disruptivas do campo da engenharia e computação desde sua criação em 2008 [14]. O tempo em sistemas digitais gasto pelas pessoas só vem aumentando [15], o que levou dados pessoais se tornaram uma "commoditie"[16]. Então, quando se trata de garantir a privacidade e garantir que não haja usos indevidos por organizações privadas ou governos ditatoriais, a *blockchain* vem se tornando uma solução líder.

A *blockchain* essencialmente é um banco de dados distribuído, considerada uma especialização de DLT (*Distributed Ledger Technology*) [17]. DLT é basicamente uma tecnologia composta de conceitos de contabilidade e computação juntos, envolvendo *ledgers* (livro-razão) que servem para registrar fluxos de caixa e variações de balanço. Assim, um *DLT* faz o mesmo que um livro-razão. No entanto, para assegurar que não haja chance de corrupção ou necessidade de extrema confiança ("somente o tesoureiro do rei pode escrever no livro razão, seu carimbo comprova a verdade"), os *DLTs* contam com a parte distribuída que consiste de registros imutáveis em uma rede P2P (2.1.3) que adiciona segurança ao sistema e também descentralização. Em *DLTs* todos os participantes do sistema validam o próprio sistema e desde que a rede seja composta majoritariamente de pessoas honestas (mais que 51%) os registros nesse banco de dados distribuído estarão corretos [18].

Como o nome sugere, a *blockchain* tem o design de uma cadeia de blocos, onde cada bloco contém informações sobre as transações verificadas (variações de fluxo de caixa) e um cabeçalho de bloco contendo uma *hash*. Os *links* entre os blocos são mantidos pelo armazenamento do *hash* do bloco anterior no cabeçalho do bloco sucessor (a Figura 2.1 mostra esse processo graficamente, usando a *blockchain* do *Bitcoin* como exemplo).

Quando qualquer participante na *blockchain* precisa fazer uma nova transação, sua validade é checada pelos outros participantes (isso só se aplica aos nodos participantes da rede *P2P* da qual é feita a *blockchain*. Hoje em dia existem nodos públicos com *APIs* expostas criados por empresas que possibilitam diversas pessoas terem acesso as redes da *blockchain* sem necessitar rodar um nodo próprio através de um servidor *JSON-RPC*). Somente se a transação é considerada válida ela é adicionada em um bloco. Cada atualização em um bloco deve ser comunicada aos outros nodos para eles atualizaram suas cópias locais, existe um mecanismo seguro para garantir qual cadeia de blocos é a válida, esse mecanismo é denominado algoritmo de consenso.

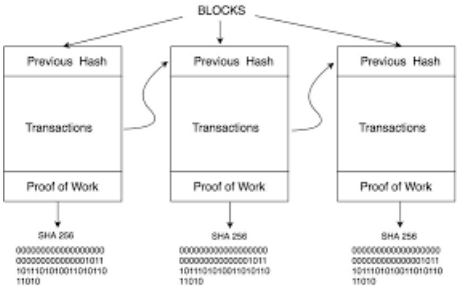


Figura 2.1: Blocos da *blockchain* do bitcoin [19]

A presente discussão mostra que existem diversos conceitos, algoritmos e estruturas de dados envolvidos na estrutura de uma rede *blockchain*, como funções de *hash*, nodos ou *peers* de uma rede *P2P*, encadeamento similar a *linked lists* (na Figura 2.1 é possível ver a semelhança), as árvores de *merkle* para *hash* de *hashes*, dentre outros. Todos esses tópios serão detalhados nessa seção.

2.1.1 Funções de *Hash*

É impossível falar de *blockchain* sem antes termos uma pequena revisão sobre funções de *hash* e assinaturas digitais. Isso se deve ao fato que qualquer sistema de *blockchain* é fundamentado em funções de criptografia avançada e assinaturas digitais consideradas confiáveis, impossíveis de falsificação ou repetição.

Começando da definição de *hash*, apresentada no artigo [20].:

> "Uma função *hash* é um algoritmo matemático que pega dados de comprimento arbitrário como entrada e mapeia-o para um texto cifrado de comprimento fixo como saída. Essa saída é chamada de resumo da mensagem, um valor *hash*, um código *hash* ou simplesmente um *hash*. Mais formalmente, uma função *hash* é uma função matemática H: D → R, onde o domínio D = 0,1 * e R = 0,1 n para algum n maior ou igual a 1, que mapeia um valor de entrada numérica m de comprimento arbitrário em uma saída de valor numérico condensado h de comprimento fixo [1]. Ou seja: h = H (m). Uma função *hash* que satisfaz alguns requisitos adicionais para que possa ser usada para os aplicativos criptográficos são conhecidos como função *hash*

> criptográfica. Essas funções são construções essenciais que possuem uma variedade de casos de uso. Os principais campos de aplicação são proteção de senhas armazenadas, autenticação de mensagens, assinaturas digitais e, portanto, certificados"[20].

Algoritmos de *hashing* são algoritmos relativamente simples que estão presentes intensivamente em diversas aplicações modernas, como a internet, bancos digitais e *blockchains* [21].

Como mencionado na citação acima, um algoritmo de *hashing* mapeia qualquer arquivo, texto, imagem ou conteúdo digital em uma sequência de números aleatórios de tamanho fixo.

Essa função de mapeamento apresenta também algumas propriedades interessantes, tais como irreversibilidade (dada a saída da função é impossível voltar ao resultado original), resistência a colisões (dada uma entrada existe uma probabilidade muito baixa de existir mais de uma saída possível) e determinismo (com a mesma entrada tem-se sempre a mesma saída).

Em sistemas digitais, a *hash* é uma forma muito eficiente de comprovar que determinada pessoa tem algum dado específico ou quem assinou algum documento. Isso é consequência direta do fato, mencionado no parágrafo anterior, que uma entrada qualquer gera uma sequência de um número aleatório fixo e não existem duas entradas diferentes que gerem a mesma saída. Logo, se alguém alterar apenas um *byte* no documento assinado é possível detectar e ainda se um *hash* de um arquivo bater com o *hash* que uma pessoa possui tem-se uma forma segura e discreta de confirmar posse (*ownership*) [22].

Qualquer *blockchain* utiliza extensivamente de funções de *hashing* e assinaturas digitais em todas suas camadas, seja para assinar uma transação, criar um contrato, enviar uma mensagem ou formar um bloco novo [23]. *Blockchains* mais antigas, como a do Bitcoin, utilizam do algoritmo SHA-256 (padrão SHA-2), ilustrado na Figura 2.2. Por outro lado, as criptomoedas mais recentes, como o *Ethereum*, utilizam o *Keccak256* (padrão SHA-3), ilustrado na Figura 2.3.

SHA significa *secure hashing algorithm* e é uma forma de padronizar os algoritmos de *hashing* sendo criados e aprovados pela indústria. Essa padronização foi criada devido a importância desses algoritmos para proteção de dados, comprovar autenticidade e validar as informações circulando online [26].

Podemos comparar a probabilidade de colisão dos algoritmos de *hashing* a partir da tabela apresentada na Figura 2.4. Nessa Figura, a cor verde indica que ainda não foram encontradas colisões para os

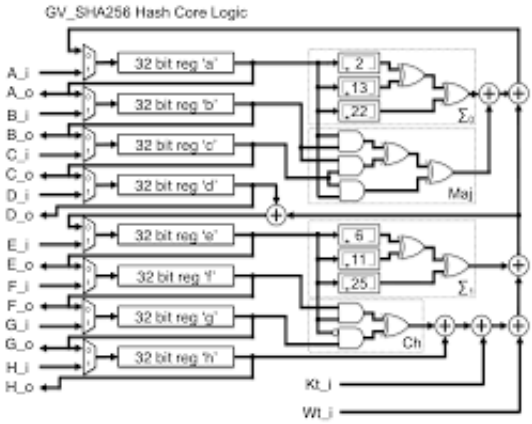


Figura 2.2: Diagrama do algorito SHA256 [24]

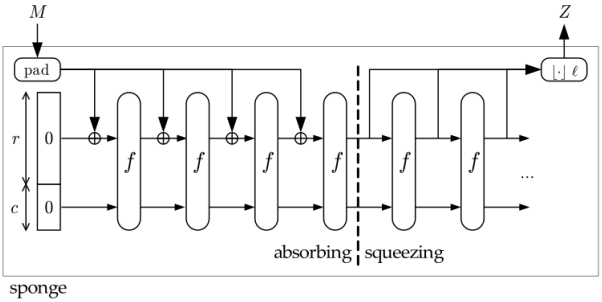


Figura 2.3: Diagrama do kecakk256 [25]

algoritmos (listados na primeira coluna), a cor amarela indica algumas colisões e a cor vermelha indica muitas colisões. Na primeira linha está contabilizado a passagem dos anos desde 1989 até 2017, na segunda coluna o número de bits de saída do algoritmo e na terceira coluna a metrificação de CPB (*cycles per byte*) de *CPU*. Na Figura 2.4, é mostrado que os algoritmos de *hashing* usado pelas *blockchains* atualmente não possuem nenhuma colisão encontrada.

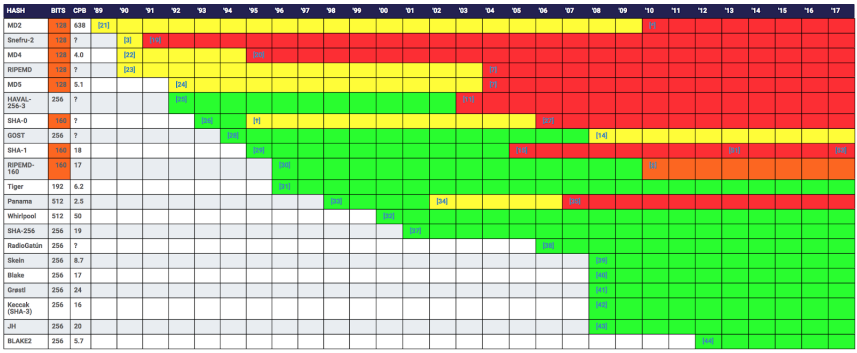


Figura 2.4: Comparação algoritmos de *hash*

2.1.2 Merkle Trees

Outro conceito fundamental quando se fala de *blockchain* são as Árvores de Dispersão ou Árvores de *Merkle*, a *Binance Academy* apresenta a seguinte definição para essas estruturas:

> "Uma *Merkle Tree* é uma estrutura usada para verificar de maneira eficiente a integridade de um conjunto de dados. Essa estrutura é especialmente interessante no contexto de redes *peer-to-peer*, onde os participantes precisam compartilhar e validar informações de forma independente"[27].

As *Merkle Trees* são as estruturas de dados utilizadas para formar os blocos da *blockchain* e também por encadeá-los. Esse tipo de estrutura não é algo novo, tendo sido inventado por *Ralph Merkle* na década de 80 [27]. Trata-se de uma ferramenta comum utilizada por muitos tipos de software populares como o sistema de arquivos dos *Iphones*, sistemas de controle de versionamento distribuído (como o *Github*) ou *Torrents* [28, 29]. As maiores motivações para o uso dessas estruturas de árvore estão na facilidade de identificar corrupção em sistemas e o arquivo ou linha exatos em que a corrupção aconteceu.

A ideia base dessa estrutura surgiu do problema apresentado na sequência. Em qualquer sistema web, normalmente quando se baixa algum *software* ou arquivo, é uma boa prática validar se o *hash* do arquivo baixado é equivalente ao *hash* publicado pelos desenvolvedores de forma a garantir que se tem um arquivo exatamente igual. Ter uma *hash* diferente pode significar uma dentre duas coisas: foi baixado algum arquivo contendo código malicioso ou um arquivo com dados

corrompidos. Nos dois casos é necessário baixar novamente o arquivo/programa para garantir que se tenha dados íntegros, o que é complicado para arquivos grandes [27]. As árvores de *Merkle* trazem uma abordagem mais eficiente para este problema. Nesse tipo de estrutura, os dados são divididos em n bloco menores [27]. Esses blocos passam cada um por uma função de *hashing* que por sua vez passa por outra função de *Hashing* e assim sucessivamente até se ter a *Merkle Root* desse blocos, isto é a ultima *hash* (*Top Hash*) da combinação entre eles. A partir da *Merkle Root* é possível validar que o arquivo está correto e caso haja algum erro, só é necessário ir descendo na arvore até se encontrar o arquivo com erro e requisitar somente o *download* dele. Esse é um processo muito menos custoso que a alternativa inicial e duas vezes menos custoso do que verificar somente a *Hash* de cada pedaço do arquivo [27]. Um exemplo mais visual do funcionamento das *Merkle Trees* pode ser visto na Figura 2.5.

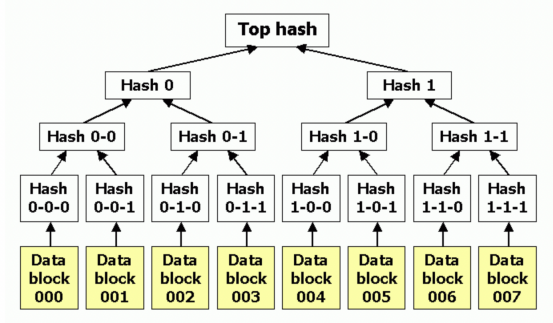


Figura 2.5: Diagrama de uma *Merkle Tree* [30]

2.1.3 Redes P2P

As redes *peer to peer* (P2P) foram idealizadas e também primeiramente usadas na famosa aplicação *Napster*, por *Shawn Fanning* em 1999 [31]. Essa arquitetura de redes possui alta capacidade para transmissão e distribuição de arquivos de forma consideravelmente mais rápida do que a obtida em uma estrutura de cliente/servidor. Essas redes tem uma arquitetura bem diferente dos modelos clássicos de cliente e servidor, conforme ilustrado na Figura 2.6. Nessas redes todas as estações de trabalho, ou nodos, tem os mesmos recursos e responsabilidades, sendo que cada *peer* é chamado de *Servlet* [32], correspondendo a uma junção de cliente e servidor.

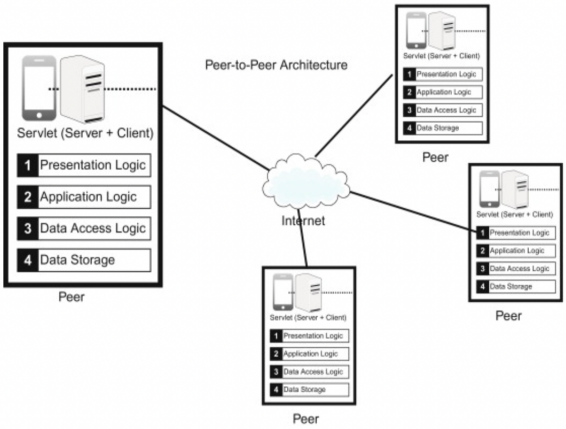
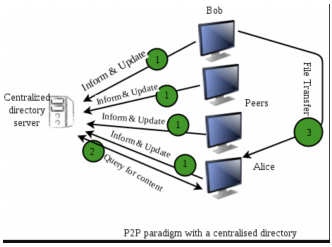


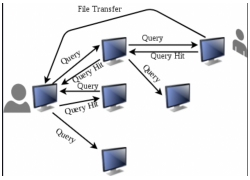
Figura 2.6: Visão de alto nível da arquitetura P2P [33]

A tecnologia P2P tem o interessante aspecto de não possuir uma entidade centralizadora, ou seja os dados e informações são distribuídos entre todos os nodos que sendo *servelets* tanto enviam quanto recebem dados. Isso faz com que o sistema seja muito escalável [34], isto é, quanto maior o número de participantes na rede, maior a capacidade do sistema, mais difícil é corromper os dados e mais rapidamente eles são transmitidos [35]. Essas características contrastam com outras formas de distribuição de conteúdo em rede na quais, quanto maior o número de clientes, mais degradado é o desempenho.

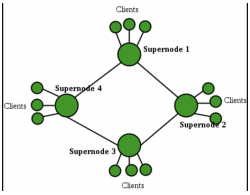
Hoje em dias existem distintas modalidades para a configuração de redes P2P levando em conta diferentes necessidades [36]. Algumas são mais centralizadas, como a de diretórios centralizados. Um exemplo desse tipo de estrutura é mostrado na Figura 2.7a. Tal estrutura parte da configuração de um nó central para registrar os *IPs* e quais arquivos existem nos demais nós participantes da rede. Temos também a estrutura de *Query flooding* neste modelo quando uma requisição (*query*) por dados é feita, esta *query* é propagada para todos os nós vizinhos ao nó que recebeu a requisição primariamente e assim sucessivamente até os dados serem fornecidos aos demais clientes (veja estrutura na Figura 2.7b). Finalmente temos o modelo de exploração de heterogeneidade, nos quais existem dois tipos de *peers*, os *child nodes* e os *super nodes*. Os super nodes comunicam-se entre si visando indexar a informação de todos os seus *child nodes*. Normalmente os super nodos requerem maior capacidade e qualidade de rede. Uma vez que a *query* é feita,



(a) Diretório Centralizado [36]



(b) *QueryFlooding* [36]



(c) *SuperNodes* [36]

Figura 2.7: Diferentes configurações de redes P2P

ela é propagada para o super nodo "pai"que então indica onde estão os dados requisitados, conforme mostrado na Figura 2.7c.

Blockchains possuem arquiteturas mais próximas as de redes P2P com *SuperNodes* [37], pois estas possibilitam um grau mais alto de descentralização uma vez que não existe dependência em um servidor centralizado. São as redes P2P que estão na base de qualquer *blockchain* e consequentemente diz-se que essa tecnologia é decentralizada, resistente a censura (nenhuma instituição tem controle sobre a rede, consegue alterá-la ou derrubá-la) e pública [38]. Esse modelo de rede, associado com os algoritmos de consenso (subseção 2.1.4) garantem a segurança das *blockchains* e as deixam praticamente impossíveis de sofrerem ataques maliciosos de qualquer gênero, garantindo assim a validade das transações, que no caso do *Bitcoin* já estão ocorrendo a 13 anos [39].

2.1.4 Algoritmos de Consenso

As *blockchains* não tem nenhuma autoridade central presente para validar e verificar as transações [40]. No entanto todas as transações são consideradas completamente seguras e verificadas a partir de algoritmos de consenso. Esses algoritmos garantem que todos os participantes (*peers*) da rede atingiram consenso sobre o estado dos dados presentes

na *blockchain*. De uma forma muito simplificada, pode-se enxergar uma *blockchain* como um banco de dados descentralizado de leitura pública, mas com regras muito restritivas para escrita, como uma planilha de *Excel* pública na qual qualquer um pode ver o registro das transações que ela indexa, existindo entretanto uma política complexa para escrita [41].

No caso da tecnologia base para este trabalho, essa regra de escrita é o protocolo de consenso, que consiste de algoritmos que os nodos validadores da rede precisam rodar para garantir a validade de cada uma das transações [42], isto é, que todos os *hashs* com transição de valores são autênticos e representam assinaturas dos endereços públicos(outros *hashs*) dos quais tratam e que esses endereços públicos realmente têm propriedade sobre os fundos/bens que estão sendo transacionados. Após todas as transações serem validadas, gera-se um *hash* dos *hashs*, a conhecida *merkle root*(2.1.2) de todo aqueles dados (ou bloco). Então essa *merkle root* é utilizada na parte final do protocolo de consenso, que corresponde a gerar a *hash* de bloco e adicioná-la como o endereço de próximo bloco. Também, vincula-se o endereço do bloco anterior, transmitindo esse endereço para todos os outros nodos validadores da rede e, caso seja aprovado, o consenso é estabelecido e a árvore de *Merkle* que representa os blocos da cadeia de blocos cresce.

Existem diferentes algoritmos de consenso possíveis para *blockchains*, sendo que todos têm como objetivo validar as transações e gerar um *hash* de bloco.

Esses algoritmos são intensamente estudados uma vez que a implementação deles impacta profundamente a *blockchain* tanto em quesitos de segurança, descentralização ou escalabilidade/performance de uma rede [43].

Se avaliarmos, no advento da tecnologia com o Bitcoin foi introduzido o algoritmo mais conhecido que é o de *Proof of Work* (PoW), este é o modelo no qual se mineram novos blocos [18].

Os algoritmos mais discutidos e utilizados atualmente são os de PoW e PoS (*Proof of Stake*) que serão os explicados nesta seção. A *blockchain* utilizada para implementação deste projeto utiliza o algoritmo de PoS, esses algoritmos trazem vantagens em custo da transação e velocidade. Conforme será apresentado essas vantagens são muito relacionadas com o mecanismo de consenso implementado e existem vários outros modelos de consenso possíveis [44].

2.1.4.1 *Proof of Work*

> “ O algoritmo de consenso *Proof of Work* é talvez a primeira instância do conceito de algoritmo de consenso sendo introduzido no mundo de blockchain. Em essência, é o algoritmo de consenso mais simples e é baseado no princípio de um problema de computação “difícil”. Basicamente, um problema difícil de resolver, mas fácil de verificar uma vez que uma solução é fornecida, é a base do algoritmo *Proof of Work*. Para que o algoritmo de consenso funcione, um problema matemático relacionado ao *hash* de criptomoeda do bloco que deve ser adicionado ao lado do blockchain. O único problema em encontrar uma solução eficiente é que a função *hash* usada para proteger o bloco é criptograficamente segura; portanto, apenas o modelo de força bruta pode ser usado para resolver o problema matemático. Uma vez que um certo nó ou líder afirma ter resolvido o problema, ele é verificado por todos os outros nós presentes na blockchain. Quando chegam a um consenso de que a solução deve ser aceita, diz-se que o bloco está “minado” e o líder é recompensado com uma certa quantidade de criptomoeda. Bitcoin, a primeira criptomoeda introduzida, usa *Hashcash*, um algoritmo de consenso baseado em Prova de Trabalho. Méritos: • Propriedades de ter problemas difíceis (solução fácil de verificar, mas difícil de conceber). Deméritos: • Quantidade extremamente alta de energia e poder de computação. ”[44]

Essa seção fala do primeiro algoritmo a ser usado para *blockchains* e que ainda é o utilizado pelas maiores redes: o *Bitcoin* e o *Ethereum*. Como é explicado nessa citação do *paper* ”Understanding and Analyzing Consensus Algorithms for Blockchain”[44], o modelo de prova de trabalho é a conhecida mineração de cripto. Nele, utiliza-se a capacidade computacional de um computador para resolver um problema que só é possível ser solucionado por força bruta (tentativa e erro). Esse problema consiste em pegar todas as transações processadas e aceitas pela rede em um intervalo chamado *blocktime* [45] (o tempo médio do Bitcoin é de 10 minutos, do Ethereum de 13 segundos [45]) e gerar um *hash* de todas elas com estruturas de *merkle trees*. Após esse processo, os mineradores precisam ir mudando um valor no cabeçalho do bloco chamado *nonce* [46] para gerar um *hash* de bloco que tenha um número pré-determinado de zeros menor que a dificuldade da rede. A

dificuldade da rede é um número ajustado a cada novo bloco gerado [47], sendo esse número transmitido a todos os participantes da rede. O primeiro participante a publicar um bloco válido é recompensado em *tokens* nativos da *blockchain* e esta é a prova de trabalho ou mineração de uma moeda. Tal fluxo é ilustrado na Figura 2.8.

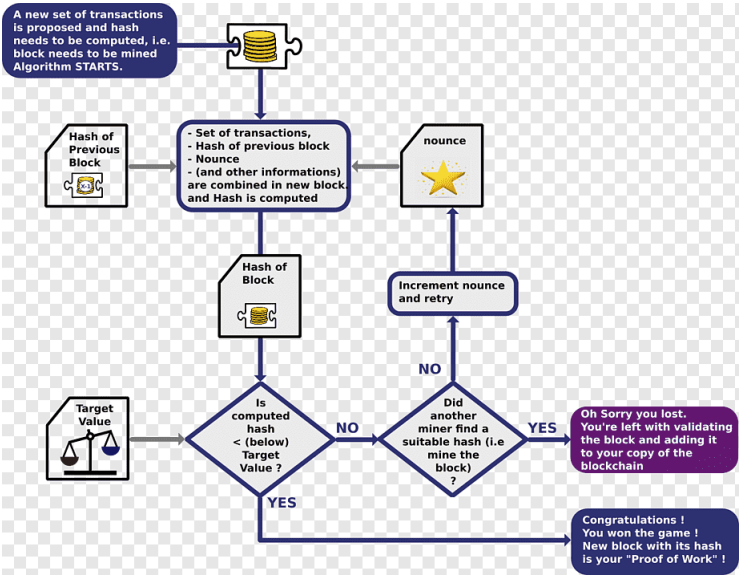


Figura 2.8: Consenso de prova de trabalho [48]

2.1.4.2 *Proof of Stake*

Proof of Stake [44] é um tipo diferente de mecanismo de consenso que as *blockchains* podem usar para concordar com um único registro verdadeiro do histórico de dados. Enquanto em *PoW* os mineradores gastam energia (eletricidade) para minerar blocos, em *PoS* validadores comprometem participação para atestar (ou 'validar') blocos.

Os validadores são os participantes da rede que executam nós (chamados nós validadores) para propor e atestar blocos em uma blockchain *PoS*. Eles fazem isso depositando criptomoedas (no caso do *Ethereum* 2.0, *ETH*) em um cofre na rede e se disponibilizam para serem selecionados aleatoriamente para propor um bloco. Outros validadores então "atestam"que viram o bloco. Quando um número suficiente de validações para o bloco for coletado, o bloco será adicionado à *blockchain*. Os validadores recebem recompensas por propor blocos com sucesso

(assim como fazem no *PoW*) e por validar os blocos que viram [49].

Os incentivos cripto econômicos para *PoS* são projetados para criar recompensas mais atraentes por comportamento adequado e penalidades mais severas por comportamento malicioso. O principal incentivo cripto-econômico se resume ao requisito de que os validadores depositem suas próprias criptomoedas — ou seja, seu dinheiro — na rede. Em vez de considerar o custo secundário de eletricidade para executar um nó *PoW*, os validadores em cadeias *PoS* são forçados a depositar diretamente uma quantia monetária significativa na rede [49].

Os validadores acumulam recompensas por fazer bloqueios (caso achem transações maliciosas) e atestados quando é sua vez de fazê-lo. Eles são penalizados por não cumprir suas responsabilidades quando for a sua vez de fazê-lo - ou seja, se estiverem offline. As penalidades por estar offline são relativamente leves e equivalem aproximadamente às recompensas esperadas ao longo do tempo. Portanto, se um validador estiver participando corretamente mais da metade das vezes, suas recompensas serão líquidas positivas.

Caso um validador tente atacar ou comprometer a *blockchain* tentando propor um novo conjunto de histórico de dados, no entanto, um mecanismo de penalidade diferente entra em ação: uma parte substancial do valor em *stake* será reduzido (possivelmente até o valor total em *stake*) e eles serão ejetados da rede. O resultado é um tremendo risco financeiro de um ataque fracassado por um validador. Para fazer uma analogia com o *PoW*, seria como se um minerador que falhou em um ataque a uma cadeia de *PoW* fosse forçado a queimar todo o seu equipamento de mineração em vez de apenas pagar o custo da eletricidade que gastou em um ataque fracassado. Além disso, essa arquitetura coloca a segurança da rede diretamente nas mãos daqueles que mantêm a rede e mantêm seu criptoativo nativo no próprio protocolo.

Proof of Stake aborda as três questões do trilema da *blockchain* – acessibilidade, centralização e especialmente escalabilidade:

Acessibilidade: As *blockchains Proof of Stake* não exigem que os validadores se preocupem com os custos iniciais de hardware ou prestem atenção às taxas de eletricidade da mesma forma que os mineradores nas cadeias *PoW* devem. Portanto, é uma barreira de entrada significativamente menor para um indivíduo executar um nó validador em uma cadeia *PoS* do que executar um nó de mineração em uma cadeia *PoW*. Há, no entanto, uma barreira notável à entrada acessível para *PoS*. Os validadores devem apostar uma quantidade mínima de criptografia para executar um nó validador completo. Para o *Ethereum* 2.0, por exemplo, esse valor é de 32 *ETH*. Para muitos, essa é uma quantia

significativa de dinheiro e um impedimento à participação ativa. Da mesma forma que as cadeias *PoW* têm grupos de mineração, no entanto, haverá grupos de *staking* que agregam os fundos de participantes incapazes ou não dispostos a depositar 32 *ETH*. Os grupos depositarão em seu nome e eles receberão recompensas como uma porcentagem de seu *stake*.

Centralização: Com barreiras reduzidas à entrada e a eliminação de preocupações com a minimização dos custos de eletricidade, as redes *PoS* são significativamente mais descentralizadas no nível do nó do que as redes *PoW*. A participação em uma cadeia *PoS* requer apenas fundos, uma conexão com a internet e tem baixo requisito de *hardware*, até mesmo uma *Raspberry Pi* é suficiente. Isso abre as portas de participação e geração de receita para um grupo muito maior de pessoas. Além disso, as economias de escala são muito menores em economia *PoS* do que *PoW*. Nos sistemas *PoW*, quanto mais poder de *hash* um minerador controlar, maior será a porcentagem de recompensas que ele poderá receber. No *PoS*, o % de retorno de um validador permanece constante se ele gerencia 1 ou 1.000 nós.

Escalabilidade: Prova de participação por si só não melhora a escalabilidade. No entanto, as arquiteturas *PoS* permitem a implementação de uma solução de escalabilidade conhecida como *sharding* sem reduzir a segurança. *Sharding* é um mecanismo de dimensionamento de banco de dados no qual um *blockchain* é particionado em várias cadeias de fragmentos, cada uma das quais é capaz de processar blocos. Isso alivia a *blockchain* de ter que processar cada bloco simultaneamente e, em vez disso, permite que vários blocos (e, em outras palavras, mais conjuntos de dados) sejam processados de uma só vez. Com o *Ethereum* 2.0, por exemplo, o *sharding* particionará o *blockchain* em 64 *shards* separados

O modelo baseado em *PoS* está se provando mais eficiente em diversos aspectos e vem sendo cada vez mais adotado [50]. Existem diversos diferentes algoritmos que usam *PoS*, está é somente uma definição de alto nível do funcionamento deste protocolo.

2.1.5 Estrutura dos Blocos

O bloco é o pedaço fundamental de dados de uma *blockchain*. A *chain* é ligada pelos blocos e eles armazenam todas as informações de transações e dados que ocorrem em uma rede. Cada bloco novo é gerado em intervalos médios de tempo. O tempo de processamento para um bloco é chamado *blocktime* e varia de rede para rede. Os blocos de maneira genérica são sempre divididos entre cabeçalho e corpo do bloco. No

cabeçalho temos informações como a *hash* do bloco anterior, a *merkle root* do bloco atual, o *nonce* (no caso de redes PoW) e a *timestamp* do momento de criação do bloco.

No corpo do bloco encontram-se informações de cada transação propagada na rede ao longo de momento de escrita do bloco com seus dados que sempre incluem os endereços, ou apenas um endereço, de quem esta recebendo a transação, mensagem ou dados e o endereço, ou endereços, de quem esta enviando a transação, as taxas sendo enviadas para o minerador (mineradores também recebem taxas direto dos usuários, pois os incentivos dados por validar o bloco não são eternos em caso de redes com limitação de oferta e para incentivá-los a minerar uma transação em caso de congestionamento de rede) e a assinatura (resultado da função de *hash* da chave privada de quem envia com os dados da transação).

Os modelos de transação variam de acordo com a lógica de funcionamento da verificação dos dados válidos a serem adicionados no banco de dados distribuído, o modelo para Bitcoin tem algumas peculiaridades que vão além do escopo necessário para este trabalho que utilizou a *sidechain* [51] *Polygon*, a qual apresenta o mesmo modelo de transações do *Ethereum*, ilustrado na Figura 2.9.

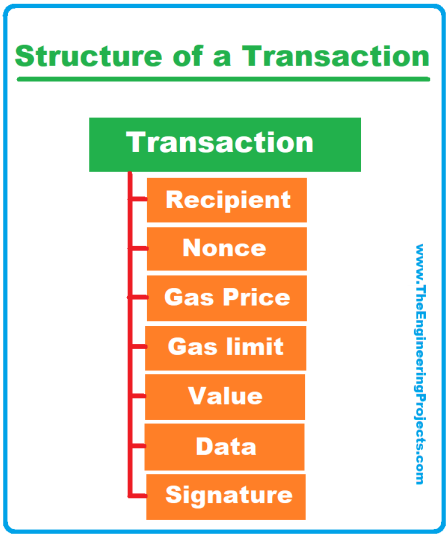


Figura 2.9: Estrutura transação do *Ethereum* [52]

2.1.6 Banco de Dados Distribuído

Como falado anteriormente, a *blockchain* se baseia em um banco de dados distribuído. No entanto, ao invés de precisar de uma contra parte centralizada confiável para validar as transações propagadas, temos os algoritmos de validação das assinaturas da transação e comparação com os registros de estado ou direito de gasto desde o começo da *blockchain* dos gastos de um endereço (os primeiro 20 bytes de uma chave pública) para confirmar cada transação propagada para os *fullnodes* (nodos que mantém todos os dados da blockchain e tem capacidade de rodar o algoritmo de consenso e emitir novos blocos), tudo isto dentro de uma rede P2P. Ainda, por volta do tempo médio de bloco, todas essas transações são indexadas e o novo bloco propagado passa por mais uma camada de verificação, a de consenso, que quando atingido pela rede dado que mais que 51% dela seja de *peers* honestos, é impossível de apresentar dados falseáveis ou mutáveis. Isto tudo forma o livro-razão distribuído com todos os registros de uma rede. No entanto, a arquitetura da informação desse livro segue dois moldes:

1. *UTXO*: *Unspent Transaction Output*;
2. *Account Model* - este modelo tem uma subsecção exclusiva.

UTXO

> " Em criptomoedas como Bitcoin, uma saída de transação não gasta (*UTXO*) é uma abstração de dinheiro eletrônico. Cada *UTXO* é análogo a uma moeda e possui uma certa quantidade de valor em sua respectiva moeda. Cada *UTXO* representa uma cadeia de propriedade implementada como uma cadeia de assinaturas digitais onde o proprietário assina uma mensagem (transação) transferindo a propriedade de seu *UTXO* para a chave pública do receptor."[53]

2.1.7 Contas

O estado global de uma *blockchain* baseada em contas é composto por contas que interagem umas com as outras por meio de uma estrutura de passagem de mensagens. A interação mais básica é a de enviar algum valor - como *tokens*, *matic*, *ether*, avalanche, dentre outros - a criptomoeda nativa da blockchain usada. Cada conta é identificada por um identificador hexadecimal de 20 bytes que é chamado de endereço (gerado a partir da chave pública da conta). O método mais aceito para

gerar o par de chave pública e privada é utilizando a *BIP39*(*Bitcoin Improvment proposal* [54]), que consiste de 2048 palavras aleatórias. Essas palavras passam por uma função de *hash* e uma de curvatura elíptica na qual é gerada a chave privada, esta deve ficar somente sobre controle do usuário que a possui pois ela permite assinar ordens e efetuar transferências na rede.

No caso de *blockchains* com modelo de contas, existem dois tipos de contas:

- *Externally Owned Accounts* - Uma conta controlada por uma chave privada e, se você for o proprietário da chave privada associada à conta, poderá enviar *tokens* e mensagens a partir dela.

- *Contract Owned Account* - uma conta que possui um código de contrato inteligente associado e sua chave privada não é de propriedade de ninguém.

As principais diferenças entre esses tipos de conta são descritos a seguir.

Externally Owned Accounts

1. Podem enviar transações (transferência de *tokens*, envio de mensagens assinadas ou interagir com algum contrato);

2. São controladas por chaves privadas;

3. Não tem código associado.

Contract Owned Accounts

1. Detém código associado;

2. A execução do código é acionada por transações (de usuários) ou mensagens (chamadas) recebidas de outros contratos;

3. Quando executado – executa operações de complexidade arbitrária (completude de *Turing*) – manipula seu próprio armazenamento persistente, ou seja, pode ter seu próprio estado permanente - pode chamar outros contratos.

2.1.8 *Explorers* e *RPCs*

Hoje em dia, existem diversas aplicações de base em cima das *blockchains* e, para possibilitar que usuários finais não tivessem que criar um nodo completo para a rede P2P de uma *blockchain* para conseguir interagir com ela, existem dois tipos de serviços que auxiliam muito desenvolvedores e usuários de aplicações em *blockchains*.

2.1.8.1 RPCs

Toda *blockchain* possui *APIs* expostas através de protocolos *HTTP* e *Websockets* que permitem que os indivíduos que rodam um nodo completo da rede consigam, através de requisições de escrita e leitura, enviar comandos ou buscar informações na *blockchain*. Esses comandos são enviados usualmente com um formato de dados chamado *JSON-RPC* e interagem diretamente com a *blockchain* via chamadas de *RPC*.

De forma similar ao nível de complexidade, em algumas *blockchains* requisições de *hardware* para ter um nodo completo local rodando também são altos. Foram criados diversos serviços de provedores de *RPC* que basicamente disponibilizam *URLs* prontas nas quais aplicações *web* conseguem montar *queries* e enviar os comandos necessários para interagir com algum protocolo em *blockchain*. Alguns bons exemplos desses provedores são:

- *Infura* (para *EVMs*);
- *Alchemy* (para *EVMs*);
- *Moralis* (para *EVMs*);
- *RPC* publico *polygon foundation*;
- *RPC* público Bitcoin.

Existem diversos serviços de *RPC*. No entanto, os melhores e mais bem documentados são relacionados aos compatíveis com *EVMs*, que serão melhores citados na próxima seção. Mas, o motivo disso é a possibilidade de criar *Smart Contracts*, o que aumenta muito a motivação e casos de uso para os *RPCs*.

2.1.9 Explorers

Os *explorers* são serviços indexadores da *blockchain*. Hoje em dia, cada rede já tem seu próprio *explorer*. Nos *explorers* é possível buscar por um

endereço, dependendo da *blockchain* endereço de contrato, transação, bloco e visualizar todas as informações relevantes sobre o item buscado. Eles facilitam a compreensão da atividade na rede e também o *tracking* das contas de usuários. Alguns exemplos de *explorer* são:

- *EtherScan (Ethereum)*
- *PolygonScan (Polygon)*
- *Bitcoin Explorer (Bitcoin)*

2.2 EVMS

Na seção 2.1 foi discorrido sobre os funcionamentos de forma geral de uma *blockchain* e seus componentes. Agora, entraremos mais a fundo na primeira tecnologia que possibilitou a *blockchain* ser tratada como um computador distribuído [55, 56], permitindo a execução de código pelos *peers* da rede. Este feito de forma simplificada foi somente a adição de uma máquina Virtual ao software de uma *blockchain* a *EVM (Ethereum Virtual Machine)* e a criação do modelo de contas que permite que estados sejam salvos dentro dos blocos [57]. Estas contribuições a estrutura de uma *blockchain* permitem que o estado de execução de uma função e sua saída sejam armazenados na rede bem como o *bytecode* de um código escrito. Isso tudo, por sua vez, torna possível existirem as *contract Owned Accounts* (endereços de *Smart-Contracts*). A definição dada para a *EVM* de acordo com a fundação *Ethereum* é:

> "A Máquina Virtual *Ethereum* é o computador virtual global cujo estado todos os participantes da rede *Ethereum* armazenam e concordam. Qualquer participante pode solicitar a execução de código arbitrário na *EVM*; a execução do código altera o estado do *EVM*"[58].

Avaliar *blockchains* como um *ledger* distribuído é muito útil para entender seus conceitos e também representa uma descrição fiel a *blockchains* de primeira geração como o Bitcoin [59]. No entanto, conforme as estruturas de *blockchain* foram evoluindo, as *blockchains* de segunda geração [59] passaram a permitir a construção de uma linguagem para desenvolvedores criarem seus *Smart Contracts* e garantir toda confiança de uma instituição financeira, no entanto sem uma parte centralizadora. Pensando em *chains* de segunda geração compatíveis com a *EVM*, a análise do banco de dados distribuído como um *ledger* distribuído fica

um pouco antiquada, uma vez que o que fica salvo no caso de *EVMS* são os estados de uma máquina de estados. Conforme a própria definição do *Ethereum*.

> "A analogia de um 'livro distribuído' é frequentemente usada para descrever *blockchains* como Bitcoin, que permitem uma moeda descentralizada usando ferramentas fundamentais de criptografia. Uma criptomoeda se comporta como uma moeda 'normal' por causa das regras que regem o que se pode e o que não se pode fazer para modificar o livro-razão. Por exemplo, um endereço Bitcoin não pode gastar mais Bitcoin do que recebeu anteriormente. Essas regras sustentam todas as transações no Bitcoin e muitas outras" *blockchains*.
>
> Embora o *Ethereum* tenha sua própria criptomoeda nativa (*Ether*) que segue quase exatamente as mesmas regras intuitivas, ele também permite uma função muito mais poderosa: contratos inteligentes. Para esse recurso mais complexo, é necessária uma analogia mais sofisticada. Em vez de um livro distribuído, o *Ethereum* é uma máquina de estado distribuído. O estado do *Ethereum* é uma grande estrutura de dados que contém não apenas todas as contas e saldos, mas um estado de máquina, que pode mudar de bloco para bloco de acordo com um conjunto predefinido de regras e que pode executar código de máquina arbitrário. As regras específicas de mudança de estado de bloco para bloco são definidas pelo *EVM*." [57]

Pode-se perceber que a *EVM* está no core do *Ethereum*. No entanto, um código de cliente do *Ethereum* tem toda a rede p2p, estruturas de bloco e algoritmo *PoW* de consenso também. O sucesso das *EVMs* foi tamanho que hoje tem-se dezenas de redes compatíveis com *EVM*, com diferentes algoritmos de consenso, estrutura de blocos e propriedades [58].

Para as *EVMs*, foram desenvolvidos diversos compiladores e *frameworks* com linguagens de alto nível para permitir a conversão dos protocolos e contratos escrito por desenvolvedores em *bytecode* interpretável pela *EVM* [60]. A *EVM* também possui um cliente *JSON-RPC* (2.1.8) disponível através de uma *API REST* que permite a publicação desses códigos na rede, o que significa que o *bytecode* é enviado para a *mempool* [61] ficando pendente. Após isso caso haja gás (veja 2.2.7) suficiente, ela é processada e validada pela rede. Então um endereço aleatório escolhido é transformado em uma ou algumas *Contract*

Owned Accounts. Para este trabalho o compilador utilizado foi o *solc* [62] e a linguagem foi *Solidity* [63].

2.2.1 Máquina de Estados

A *EVM* é a parte do *Ethereum* que lida com a implementação e execução de contratos inteligentes [57]. Transações simples de transferência de valor de uma *EOA* para outra não precisam envolvê-la, praticamente falando, mas todo o resto envolverá uma atualização de estado computada pela *EVM*. Em alto nível, a *EVM* executada na *blockchain Ethereum* pode ser pensada como um computador global descentralizado contendo milhões de objetos executáveis, cada um com seu próprio armazenamento de dados permanente [64].

A *EVM* é uma máquina de estado *Turing*-completa [13]. No entanto ela possui uma particularidade: todos os processos de execução são limitados a um número finito de etapas computacionais pela quantidade de gás (2.2.7) disponível para qualquer execução de contrato inteligente. Dessa forma o problema de *halting* [65] é "resolvido"(todas as execuções do programa serão interrompidas) e a situação em que a execução pode, acidentalmente ou maliciosamente, ser executada para sempre (um dos motivos pelo qual o *bitcoin* [66] e as *blockchains* de primeira geração não permitem diversas instruções a serem enviadas para a rede principalmente *loops*) é evitada para o *Ethereum*.

A *EVM* tem uma arquitetura baseada em *stack* (pilha), armazenando todos os valores na memória em uma pilha (ou seja não se usa registradores) [57]. A máquina virtual funciona com um tamanho de palavra de 256 bits (principalmente para facilitar operações de hashing nativo e curva elíptica) e possui vários componentes de dados endereçáveis [64]:

- Uma ROM de código de programa imutável, carregada com o *bytecode* do contrato inteligente a ser executado;
- Uma memória volátil, com cada local explicitamente inicializado em zero;
- Um armazenamento permanente que faz parte do estado *Ethereum*, também inicializado em zero.

A Arquitetura e Contexto de Execução da *Ethereum* Virtual Machine (EVM) é ilustrada na Figura 2.10 [64].

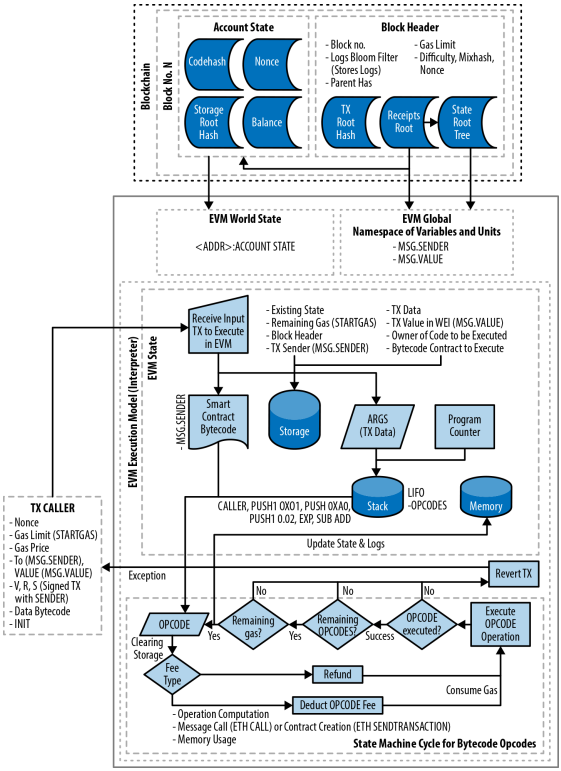


Figura 2.10: Arquitetura EVM e contexto de transação [64]

2.2.2 Função de Transição de Estados

Conforme explicitado anteriormente, o *Ethereum* é estruturada com uma máquina de estados. Assim, ela usa um modelo diferente do modelo de *UTXO* do bitcoin. Em redes compatíveis com *EVMs*, tem-se uma função de transição de estados para alterar o saldo de qualquer endereço participante da rede (seja uma *Externally Owned Account - EOA* ou *Contract Owned Account - COA*, com a diferença que transações entre duas EOA só precisam da função de transição de estados e não interagem diretamente com a *EVM*, portanto usam muito menos gás e geralmente são mais baratas). Essa função de transição tem uma lógica muito bem ilustrada no *WhitePaper* do *Ethereum* [67]:

> "A função de transição de estado Ethereum, APPLY(S,TX) - S' pode ser definida da seguinte forma:

1. Verifique se a transação está bem formada (ou seja, tem o número correto de valores), a assinatura é válida e o *nonce* corresponde ao *nonce* na conta do remetente. Caso contrário, retorne um erro.
2. Calcule a taxa de transação como $STARTGAS * GASPRICE$ e determine o endereço de envio da assinatura. Subtraia a taxa do saldo da conta do remetente e incremente o *nonce* do remetente. Se não houver saldo suficiente para gastar, retorne um erro.
3. Inicialize $GAS = STARTGAS$, e retire uma certa quantidade de gás por byte para pagar os bytes na transação.
4. Transfira o valor da transação da conta do remetente para a conta do destinatário. Se a conta de recebimento ainda não existir, crie-a. Se a conta de recebimento for um contrato, execute o código do contrato até a conclusão ou até que a execução fique sem gás.
5. Se a transferência de valor falhou porque o remetente não tinha dinheiro suficiente ou a execução do código ficou sem gás, reverta todas as alterações de estado, exceto o pagamento das taxas, e adicione as taxas à conta do minerador.
6. Caso contrário, devolva as taxas de todo o gás restante ao remetente e envie as taxas pagas pelo gás consumido ao minerador.

”

2.2.3 Instruções da *EVM*

Como foi apresentado na subseção 2.2.1, a *EVM* é uma máquina virtual com profundidade de 1024 itens na pilha. Cada item possui 256 bits por palavra, sendo escolhidos pela facilidade de uso com criptografia de 256 bits. Durante a execução, a *EVM* mantém uma matriz de bytes endereçada por palavra também chamada de memória transitória. Para os contratos, possuem uma matriz endereçável por palavra associada a conta em questão e fazendo parte do estado global que permite que as informações deles continuem após a execução em memória não volátil. O *bytecode* compilado do *smart contract* executa um certo número de *opcodes* na *EVM*, usando operações da pilha como *AND*, *XOR*, *SUB*, *ADD*, entre outras funções. Implementando também várias operações

específicas de *blockchain*, como *ADDRESS*, *BALANCE*, *BLOCKHASH* e funções de assinatura e *hashing* [64].

As instruções da *EVM* possuem muitas operações a mais das que foram citadas anteriormente,incluindo as ilustradas na lista abaixo [64]:

1. Operações aritméticas e lógicas bit a bit;

2. Consultas de contexto e execução;

3. Acesso a pilha,memória e armazenamento;

4. Controle de fluxos de operação;

5. Registros,chamadas e outros operadores.

Os *opcodes* disponíveis podem ser divididos nas seguintes categorias:

1. Instruções Aritméticas do *opcode*, Figura 2.11a;

2. Operações da pilha, Figura 2.11b;

3. Instruções para fluxo de controle, Figura 2.11c;

4. *Opcodes* para o sistema que executa o programa, Figura 2.11d;

5. *Opcodes* para comparações e lógica bit a bit, Figura 2.11e;

6. Opcodes que lidam com informações do ambiente de execução, Figura 2.11f;

7. Opcodes para acessar informações no bloco atual, Figura 2.11g;

```
ADD         //Add the top two stack items
MUL         //Multiply the top two stack items
SUB         //Subtract the top two stack items
DIV         //Integer division
SDIV        //Signed integer division
MOD         //Modulo (remainder) operation
SMOD        //Signed modulo operation
ADDMOD      //Addition modulo any number
MULMOD      //Multiplication modulo any number
EXP         //Exponential operation
SIGNEXTEND //Extend the length of a two's complement signed integer
SHA3        //Compute the Keccak-256 hash of a block of memory
```

(a) Opcodes 1 [64]

```
POP     //Remove the top item from the stack
MLOAD   //Load a word from memory
MSTORE  //Save a word to memory
MSTORE8 //Save a byte to memory
SLOAD   //Load a word from storage
SSTORE  //Save a word to storage
MSIZE   //Get the size of the active memory in bytes
PUSHx   //Place x byte item on the stack, where x can be any integer from
        // 1 to 32 (full word) inclusive
DUPx    //Duplicate the x-th stack item, where x can be any integer from
        // 1 to 16 inclusive
SWAPx   //Exchange 1st and (x+1)-th stack items, where x can be any
        // integer from 1 to 16 inclusive
```

(b) Opcodes 2 [64]

```
STOP      //Halt execution
JUMP      //Set the program counter to any value
JUMPI     //Conditionally alter the program counter
PC        //Get the value of the program counter (prior to the increment
          //corresponding to this instruction)
JUMPDEST  //Mark a valid destination for jumps
```

(c) Opcodes 3 [64]

```
LOGx            //Append a log record with x topics, where x is any integer
                //from 0 to 4 inclusive
CREATE          //Create a new account with associated code
CALL            //Message-call into another account, i.e. run another
                //account's code
CALLCODE        //Message-call into this account with another
                //account's code
RETURN          //Halt execution and return output data
DELEGATECALL    //Message-call into this account with an alternative
                //account's code, but persisting the current values for
                //sender and value
STATICCALL      //Static message-call into an account
REVERT          //Halt execution, reverting state changes but returning
                //data and remaining gas
INVALID         //The designated invalid instruction
SELFDESTRUCT    //Halt execution and register account for deletion
```

(d) Opcodes 4 [64]

```
LT      //Less-than comparison
GT      //Greater-than comparison
SLT     //Signed less-than comparison
SGT     //Signed greater-than comparison
EQ      //Equality comparison
ISZERO //Simple NOT operator
AND     //Bitwise AND operation
OR      //Bitwise OR operation
XOR     //Bitwise XOR operation
NOT     //Bitwise NOT operation
BYTE    //Retrieve a single byte from a full-width 256-bit word
```

(e) Opcodes 5 [64]

```
GAS             //Get the amount of available gas (after the reduction for
                //this instruction)
ADDRESS         //Get the address of the currently executing account
BALANCE         //Get the account balance of any given account
ORIGIN          //Get the address of the EOA that initiated this EVM
                //execution
CALLER          //Get the address of the caller immediately responsible
                //for this execution
CALLVALUE       //Get the ether amount deposited by the caller responsible
                //for this execution
CALLDATALOAD    //Get the input data sent by the caller responsible for
                //this execution
CALLDATASIZE    //Get the size of the input data
CALLDATACOPY    //Copy the input data to memory
CODESIZE        //Get the size of code running in the current environment
CODECOPY        //Copy the code running in the current environment to
                //memory
GASPRICE        //Get the gas price specified by the originating
                //transaction
EXTCODESIZE     //Get the size of any account's code
EXTCODECOPY     //Copy any account's code to memory
RETURNDATASIZE //Get the size of the output data from the previous call
                //in the current environment
RETURNDATACOPY //Copy data output from the previous call to memory
```

(f) Opcodes 6 [64]

```
BLOCKHASH   //Get the hash of one of the 256 most recently completed
            //blocks
COINBASE    //Get the block's beneficiary address for the block reward
TIMESTAMP   //Get the block's timestamp
NUMBER      //Get the block's number
DIFFICULTY //Get the block's difficulty
GASLIMIT    //Get the block's gas limit
```

(g) Opcodes 7 [64]

Figura 2.11: EVM Opcodes [64]

O diagrama da Figura 2.12, apresenta a arquitetura da *EVM*.

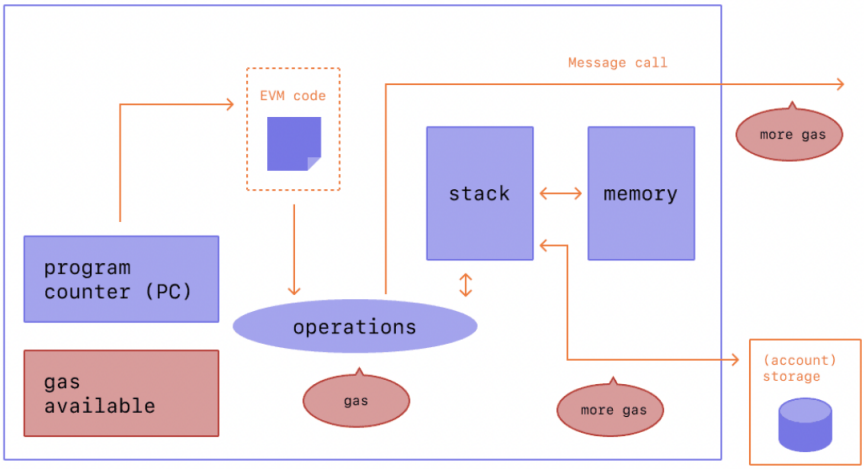


Figura 2.12: Diagrama adaptado das instruções da *EVM* [57]

2.2.4 Implementações de EVM

Todas as implementações da *EVM* devem seguir as especificações descritas no *Ethereum Yellow paper* [68]. [57] A *EVM* passou por inúmeras revisões e implementações em várias linguagens de programação ao longo dos 5 anos de história do *Ethereum*. Existem várias implementações da *EVM* independentes[57], alguns exemplos são apresentados na sequência:

- *Py-EVM - Python*;
- *evmone - C++*;
- *ethereumjs-vm - JavaScript*;
- *eEVM - C++*;
- *Hyperledger Burrow - Go*;
- *hevm - Haskell*;
- *GETH - Go*;

O modelo mais usado é o *GETH*, *go-ethereum*, distribuído pela fundação *Ethereum* como modelo oficial, mais otimizado e recente [69]. Este é o modelo base da maioria dos *forks* da *EVM* e em outras *blockchains* que implementam a lógica de máquina virtual do *Ethereum*.

2.2.5 Assinaturas Digitais

As assinaturas digitais em *EVMs* são feitas com o modelo de *Elliptic Curve Digital Signature Algorithm* (ECDSA) [70, 71]. Esse modelo é baseado em valores de chave públicas e privadas gerados por curva elíptica (gerados pela *bip39* [54]). Esse algoritmo simplesmente assina um conjunto de informações com a chave privada de um usuário e, dado que você tenha as mesmas informações, é possível gerar um *hash* e junto da assinatura efetuar algumas verificações matemáticas que somente no caso das informações que geraram o *hash* forem exatamente iguais as informações assinadas te retornam o valor da chave pública [72]. Mais adiante, pegando os 20 primeiros *bytes* da chave pública, é possível autenticar que uma *EOA* realmente assinou uma transação e a executar na *blockchain*. Resumindo, uma assinatura digital serve a três propósitos em implementações de *EVM*. Primeiro, a assinatura prova que o proprietário da chave privada, que é, por implicação, o proprietário de uma *EOA*, autorizou alguma transação para gastar seus fundos ou a execução de um contrato. Em segundo lugar, garante a legitimidade: a prova da autorização é inegável (se suas chave privadas foram vazadas, você compartilhou com alguém e quem as tem, tem direito sobre a conta). Em terceiro lugar, a assinatura prova que os dados da transação não foram e não podem ser modificados por ninguém após a transação ter sido assinada (imutabilidade da *blockchain*).

2.2.6 *ERCs* e *EIPs*

Como é possível compreender ao longo da seção de *EVMs*, uma máquina virtual combinado a *blockchain* permite que qualquer pessoa desenvolva códigos variados e os publique juntamente a uma *blockchain* que implemente alguma versão dessa máquina virtual. No entanto, como esses códigos implementados são usados por diversos protocolos na *blockchain*, precisamos de definições precisas de um padrão de *token* fungível, não fungível, modelos de assinatura e etc... A fundação *Ethereum* tem uma série de normas e padrões oficiais de forma a universalizar o acesso aos recursos da *blockchain*, também a motivação e definição de *OPCODES* próprios. Esses padrões são publicados na forma das *EIPs* e *ERCs* [73].

Da definição da fundação *Ethereum*:

- *EIPs*(*Ethereum Improvement Proposals*) - As Propostas de Melhoria *Ethereum* (*EIPs*) descrevem padrões para a plataforma *Ethereum*, incluindo especificações de protocolo principais, *APIs*

de cliente e padrões de contrato. As atualizações de rede são discutidas separadamente no repositório do *Ethereum Project Management*.

- *ERC* é um acrônimo que significa *Ethereum Request for Comments*. *ERCs* são padrões de nível de aplicativo para *Ethereum* e podem incluir padrões de *token*, registros de nomes, formatos de biblioteca/pacote e muito mais.

2.2.7 Gás

A gás conforme a definição do *Investopedia* é:

> "Gás refere-se à taxa, ou valor de preço, necessário para realizar com sucesso uma transação ou executar um contrato na plataforma *blockchain Ethereum*. Precificado em pequenas frações do *ether* de criptomoeda (*ETH*), comumente referido como *gwei* e às vezes também chamado de *nanoeth*, o gás é usado para alocar recursos da máquina virtual *Ethereum (EVM)* para que aplicativos descentralizados, como contratos inteligentes, possam se auto-executar de forma segura, mas descentralizada.
>
> O preço exato do gás é determinado pela oferta e demanda entre os mineradores da rede, que podem se recusar a processar uma transação se o preço do gás não atingir seu limite, e os usuários da rede que buscam poder de processamento."[74]

Desta definição formal, podemos compreender o que é a gás. O *gwei* comentado ao longo dessa definição é definido com dez elevado a nove *WEI*. O *WEI* por sua vez é a unidade base do *Ethereum* e do *token* nativo de toda *blockchain* compatível com *EVM* que é equivalente a dez elevado a menos dezoito *Ethers* ou do token nativo de uma *blockchain* compatível com *EVM*.

Por mais que a definição do *Investopedia* introduza o conceito de gás de uma forma menos densa, ela é incompleta em alguns pontos relevantes que podemos compreender através artigo de definição de gás da fundação *Ethereum*:

> "O cálculo da taxa total da transação funciona da seguinte forma: Unidades de gás (limite) * (Taxa base + Gorjeta)
>
> Digamos que a Jordânia tenha que pagar a Taylor 1 ETH. Na transação, o limite de gás é de 21.000 unidades e a taxa

básica é de 100 gwei. A Jordânia inclui uma gorjeta de 10 gwei.

Usando a fórmula acima, podemos calcular isso como 21.000 * (100 + 10) = 2.310.000 gwei ou 0,00231 ETH.

Quando a Jordânia enviar o dinheiro, 1,00231 ETH serão deduzidos da conta da Jordânia. Taylor receberá 1.0000 ETH. Minerador recebe a gorjeta de 0,00021 ETH. A taxa básica de 0,0021 ETH é queimada.

Além disso, Jordan também pode definir uma taxa máxima (maxFeePerGas) para a transação. A diferença entre a taxa máxima e a taxa real é reembolsada à Jordânia, ou seja, reembolso = taxa máxima - (taxa base + taxa de prioridade). A Jordânia pode definir um valor máximo a pagar pela execução da transação e não se preocupar em pagar "além"da taxa básica quando a transação for executada"[75].

O *gasPrice* é definido em *GWEI* de acordo com o nível de uso da rede e gargalo de processamento dos mineradores, para o caso de consenso PoW, ou validadores para o caso de consenso PoS. O *gasLimit* representa o máximo de gás que você deseja pagar (que é multiplicado pelo *gasPrice*) por uma transação. O gás calculado para o custo de uma transação ocorre conforme ela é executada. No entanto, existem mecanismos de estimativa que fazem previsões muito acuradas, os quais contabilizam todas as operações possíveis com pesos diferentes (de acordo com o poder computacional requisitado para executar a operação), desde necessidade de alocação de memória volátil, escritas na pilha da *EVM*, escritas em memória permanente (as mais caras computacionalmente), *loops* e etc.

2.2.7.1 EIP-712

Este tópico remete a uma *EIP* do *Ethereum*. Ele se aplica a todas as *EVMs*, muito importante para este trabalho e extensivamente usada nos códigos desenvolvidos para o protocolo desse trabalho. A *EIP-712* [76] apresenta a padronização para a assinatura de mensagens autorizando alguma ação a ser feita por um contrato pelo usuário e fica fora da *blockchain*, isto é, você usa o algoritmo de *ECDSA* para assinar um conjunto de informações e isso te retorna o *hash* da assinatura, que é armazenado e usado como parte de uma transação depois. Um bom exemplo é um evento de vendas, no qual a o interesse de se vender

um item mas não existem compradores no momento, então o evento de venda criado expressa o valor e o item sendo vendido. O usuário assina dando autorização para venda do item e posteriormente quando um comprador aparecer e tentar comprar o item, a autorização do vendedor já esta expressa e a transação pode ocorrer naquele exato momento. É importante reparar que até o momento do comprador aparecer, todo o resto do procedimento é efetuado *offChain* e essa assinatura tem que ser armazenada em um banco de dados seguro. O artigo da *EIP-712* apresenta a motivação para esse padrão:

> "Este *EIP* visa melhorar a usabilidade da assinatura de mensagens *offChain* para uso *onChain*. Estamos vendo uma crescente adoção da assinatura de mensagens *offChain*, pois economiza gás e reduz o número de transações no blockchain."[76]

2.3 Polygon

A Polygon [77] é uma *sidechain* [51] *layer 2* [78] para a rede do *Ethereum*. É importante entender que nem todos os *forks* de *EVM* são *layer 2*. Alguns são implementações totalmente novas que não dependem de uma rede primária para existir [79], mas basicamente isso quer dizer que a *Polygon* tenta resolver os problemas de escalabilidade do *Ethereum* [80] (*throughput* de transações e preço das taxas da rede) mas manter a segurança e descentralização do *Ethereum*. Isto é possível pois, como será explicado ao longo desta seção, a *Polygon* implementa uma nova versão de *EVM* com consenso de *PoS*. No entanto ela faz *checkpoints* de suas transações os anexando ao *Ethereum* para serem validados. Isso quer dizer que ela atua de forma auxiliar ao *Ethereum*, com muito mais velocidade e capacidade de processamento de transações por segundo. Também foi escolhida como a rede na qual este trabalho será publicado principalmente pelo preço menor de gás e por ser a *layer 2* com maior movimento de *NFTs* [81]. De acordo com a documentação oficial da *Polygon*

> "*Polygon* é uma solução de dimensionamento para *blockchains* públicos. Com base em uma implementação adaptada da estrutura *Plasma* (*Plasma MoreVP*) - com uma implementação baseada em contas (leia mais aqui), a *Polygon* suporta todas as ferramentas *Ethereum* existentes, além de transações mais rápidas e baratas."[82]

O modelo *Plasma MoreVP* [83] (*MoreVP* é um acrônimo para *More Viable Product* pois esta versão da plasma foi a última publicada após diferentes modelos propostos) é um sistema complexo criado pelos fundadores do *Ethereum* para propor à comunidade uma solução para escalar o *Ethereum*, que desde 2017 sofre com problemas de congestionamento de rede e continua crescendo em números de usuários. Os fundadores da *Polygon* são um exemplo de indivíduos do ecossistema que, após estudarem esta série de artigos, publicaram sua própria *sidechain* em cima do *Ethereum*.

Também é importante ressaltar que a *Polygon* se chamava *Matic* [77] até muito recentemente, mas, por questões de *branding*, foi alterado o nome da rede para *Polygon*. Apesar disso, o nome do *token* nativo da *Polygon* continua e continuará sendo *Matic*.

2.3.1 *Sidechains* e *Layer 2*

Sidechain é uma definição mais formal para *forks* da *EVM*, dado que algumas condições são respeitadas. De acordo com a definição da *Polygon*:

> "Pense em um *Sidechain* como um clone de um blockchain 'pai', suportando a transferência de ativos de e para a rede principal. É simplesmente uma alternativa à rede pai que cria uma nova blockchain com seu próprio mecanismo de criação de blocos (mecanismo de consenso). Conectar uma *sidechain* a uma rede pai envolve a configuração de um método de movimentação de ativos entre as rede"[82].

Os protocolos para enviar os ativos de uma rede para a outra são chamados de *bridges* [84]. Existem vários modelos de *bridges* e também uma dependência grande da implementação da *sidechain* para o formato de *bridge* que ela possui. Por mais que hoje em dia exista interoperabilidade entre as *sidechains*, elas devem ao menos possuir possibilidade de transferência com o *Ethereum* para poderem ser consideradas *sidechains*. No entanto toda *sidechain* é também por definição uma *blockchain* por si só, mesmo não possuindo *bridges*.

Já para se enquadrar na definição de *layer 2*, existem algumas considerações extras que devem ser observadas isto é a *sidechain* fazer *checkpoints* na rede pai.

Como melhor pontuado em um artigo escrito por *Sandeep Nailwal*, fundador da *Polygon*:

> "O *Polygon* funciona principalmente por meio de cadeias de *Commit*, que são redes de transações que operam adjacentes a uma blockchain principal – neste caso, *Ethereum*. As cadeias de confirmação agrupam lotes de transações e as confirmam em massa antes de retornar os dados à cadeia principal. Teoricamente, o *Polygon* eventualmente terá milhares de cadeias escalando juntas para aumentar a taxa de transferência, com o potencial de um dia gerar milhões de transações por segundo (TPS) quando conectado a uma cadeia principal como o *Ethereum*. Atualmente, o *Polygon* usa apenas a conectividade *Commit Chain* para melhorar os tempos de transação"[85].

O termo *commit* usado na citação acima vem da própria definição do termo, ou seja, de efetuar registro de transações e gerar um ponto salvo em um sistema. Nesse caso o *commit* contitui-se de *hashes* geradas desses blocos de transações através de *merkle roots* após algumas confirmações dos blocos na rede da *Polygon*. Assim, além de eficiência e escalabilidade as transações da *Polygon* garantem segurança passando por duas camadas de validação, uma no próprio consenso da *Polygon* e posteriormente o do *Ethereum*.

2.3.2 *Bridges*

A *Polygon* apresenta hoje dois modelos de *bridge compatível* com o *Ethereum* a *PoS Bridge* e a *Plasma Bridge*. Da definição da documentação da *Polygon*:

> "Uma ponte é basicamente um conjunto de contratos que ajudam na movimentação de ativos da cadeia raiz para a cadeia filha. Existem basicamente duas pontes para mover ativos entre *Ethereum* e *Polygon*. A primeira é a ponte Plasma e a segunda é chamada de *PoS Bridge* [82] ou *Proof of Stake bridge*. A ponte de plasma oferece uma maior garantia de segurança devido ao mecanismo de saída de plasma.
>
> No entanto, existem certas restrições no *token* filho e há um período de retirada de 7 dias associado a todas as saídas/retiradas do *Polygon* para o *Ethereum* na ponte *Plasma*.
>
> Isso é bastante doloroso para aqueles *DApps*/usuários que precisam de alguma flexibilidade e saques mais rápidos e estão satisfeitos com o nível de segurança fornecido pela

> ponte *Polygon Proof-of-Stake*, protegida por um conjunto robusto de validadores externos.
>
> A prova de ativos baseados em participação fornece segurança de *PoS* e saída mais rápida com um intervalo de ponto de verificação"[82].

2.4 Marketplaces de NFT

Marketplaces de *NFTs* são aplicações que necessitam de diversos componentes para funcionar adequadamente, sendo tais componentes ilustrados na Figura 2.13.

No entanto o componente central para um *marketplace* de *NFTs* é seu protocolo de *core* que são uma série de contratos publicados nas *blockchains*. Nesta seção, serão apresentados os principais componentes de um protocolo de *Marketplace de NFTs*. Conforme *Akash Takyar*, uma plataforma de *marketplace* de *NFTS* é definida com se segue:

> "Os *marketplace* de NFTs, como o nome indica, é uma plataforma descentralizada onde os usuários podem criar, comprar, vender e armazenar *tokens* não fungíveis. Uma plataforma de negociação de *NFTs* facilita a cunhagem e negociação de NFT em escala global, enquanto a *blockchain* como sua tecnologia subjacente garante transparência e registro imutável do processo de *tokenização* e negociação de ativos digitais.
>
> Os *marketplaces* de *NFT* estão ganhando força em setores como jogos, arte, redes sociais e música, capturando quase todos os mercados que lidam com ativos digitais. Com o aumento dos projetos no Metaverso, essas plataformas cresceram em destaque e relevância ainda mais.
>
> Constantemente os recursos dos *marketplaces* estão sendo atualizados e limitações como a falta de interoperabilidade dos *NFTs* estão sendo abordadas. *Bridges* entre *chains*, *NFTs* específicos de nicho, trocas de *NFTs* e compatibilidade com vários projetos do metaverso são alguns dos recursos avançados em *marketplaces* de *NFT* contemporâneos. "[86]

2.4.1 Aplicação de *Marketplace*

Conforme ilustrado na Figura 2.13, os principais componentes no desenvolvimento de uma aplicação de *marketplace* são:

- **Storefront**: O *front-end* da aplicação - mostra os *NFTs* da *blockchain* sendo usada, os donos dos *NFTs*, visualização das imagens e metadados, histórico de negociação e oferece mecanismos de compra e venda;
- **Filtros**: Os filtros para busca de *NFTs*, muito importantes em *marketplaces* com muitos *NFTs* - envolvem filtros por preço, moeda, categorias e etc...;
- **Mecanismos de Buscas**: Os mecanismos de *search bar* permitem busca por nome, coleção e tipo. Esses mecanismos são um desafio para serem otimizadas as *queries* para o banco de dados centralizado;
- **Criação de eventos de venda**: É uma tarefa do protocolo do *marketplace* e será discutido posteriormente;
- **Mecanismos de compra e *bids***: Também é tratado pelo protocolo;
- **Carteira Cripto**: A carteira cripto intermediará a comunicação segura entre um usuário e a aplicação através de uma *API*.

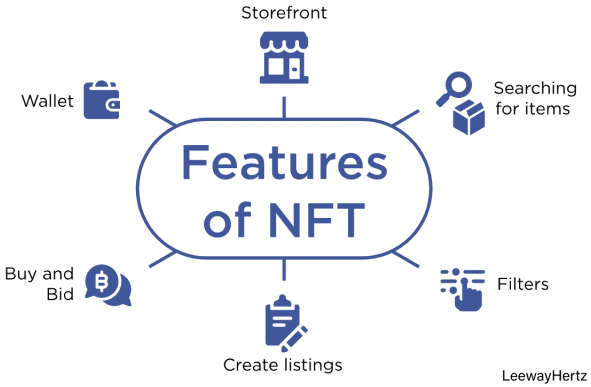


Figura 2.13: Leeway Hertz: diagrama de *marketplace* de *NFT*[86]

2.4.2 Protocolo de *Marketplace* de *NFT*

Protocolos de *marketplace* de *NFT* são *smart contracts* que implementam algumas interfaces e um padrão de lógica com regras bem definidas para possibilitar a troca de ativos registrados na *blockchain* de uma forma automática e segura. As duas principais referências de implementação para esses protocolos são a *Wyvern Exchange* [87] (usado pelo *OpenSea*) e o *Rarible Protocol* [88] (utilizado pela *Rarible*). Esses dois protocolos foram utilizados como base para o desenvolvimento deste trabalho.

Os ativos registrados em *blockchain* para o caso de *blockchains* compatíveis com *EVMs* são chamados, de forma geral, *tokens* [89] e apresentam uma série de interfaces normatizadas pela fundação *Ethereum* que devem ser implementadas para esses *tokens* serem considerados válidos. Existem três tipos de *tokens*:

1. *Tokens* fungíveis, normatizados pela *EIP-20*; [90]
2. *Tokens* não fungíveis, normatizados pela *EIP-721*; [91]
3. *Tokens* semi fungíveis, normatizados pela *EIP-1155*. [92]

Na sequência é apresentada a definição de cada modelo de *token* supracitados. Posteriormente são apresentadas as definições gerais de um protocolo de *marketplace*. Finalmente, é discutida a implementação dos protocolos de referência deste trabalho.

2.4.3 *Tokens*

As normas para *tokens* estão entre as mais importantes publicações de *EIPs* da fundação *Ethereum*, pois a maioria dos projetos desenvolvidos criam seus próprios *tokens* com as mais variadas finalidades, por exemplo:

1. *Tokens* fungíveis: indicam a valorização de mercado, os *shareholders* (poder de votação e decisão), dão liquidez, incentivos e etc;
2. *Tokens* não fungíveis: direito a propriedade de um item digital ou físico, como casas, obras de arte, copyright de músicas e etc;
3. *Tokens* semi fungíveis: propriedade sobre itens digitais ou físicos, como livros, colecionáveis, *skin* de jogos e etc.

Conforme apresentado na Seção 2.2 essas normas são validas e mandatórias para qualquer *blockchain* que implementa a máquina virtual do *Ethereum*.

2.4.3.1 *EIP-721*

A *EIP-721* é responsável por definir as interfaces mandatórias de funções e eventos para *tokens* não fungíveis em redes de *EVM*, bem como definir o que deve, não deve e é opcional de ser feito quanto a interfaces extras (chamadas de extensões). A *ERC-721* apresenta a implementação dessas interfaces da *EIP-721*. O melhor exemplo da *ERC-721* e suas extensões é desenvolvido pela *OpenZeppelin* [93]. A *OpenZeppelin* é uma biblioteca *Open Source* que traz implementações em *solidity* para as normas da fundação *Ethereum*. Inclusive sendo o exemplo de implementação indicado pela fundação *Ethereum* [94]. Também é importante definir que os *smart contracts* implementados com a *EIP-721* para *tokens* não fungíveis são normalmente chamados de coleções.

De acordo com a fundação *Ethereum*, a *EIP-721* é resumidos da seguinte forma:

> "O padrão a seguir permite a implementação de uma *API* padrão para *NFTs* em contratos inteligentes. Este padrão fornece funcionalidade básica para rastrear e transferir *NFTs*.
>
> Consideramos casos de uso de *NFTs* pertencentes e transacionados por indivíduos, bem como consignação a corretores/carteiras/leiloeiros terceirizados ("operadores"). *NFTs* podem representar propriedade sobre ativos digitais ou físicos..." [91]

Essa citação, retirada do *abstract* da *EIP-721*, além de apresentar uma boa definição do que é e para que serve o padrão 721, argumenta sobre o caso de uso desses *tokens* em serviços terceirizados que é principal relação entre *tokens* e o protocolo desenvolvido neste trabalho. A interface padrão do 721 é apresentada na listagem 2.1 e mostra todas as funções e eventos mandatórios definidos pela fundação *Ethereum*. Qualquer extensão do 721 necessita implementar essa interface também.

```
1  /// @title ERC-721 Non-Fungible Token Standard
2  /// @dev See https://eips.ethereum.org/EIPS/eip-721
3  ///  Note: the ERC-165 identifier for this interface is 0
       x80ac58cd.
4  interface ERC721 /* is ERC165 */ {
5      event Transfer(address indexed _from, address indexed
           _to, uint256 indexed _tokenId);
6      event ApprovalForAll(address indexed _owner, address
           indexed _operator, bool _approved);
7      function balanceOf(address _owner) external view returns
            (uint256);
8
9      function ownerOf(uint256 _tokenId) external view returns
            (address);
```

```
10
11      function safeTransferFrom(address _from, address _to,
            uint256 _tokenId, bytes data) external payable;
12
13      function safeTransferFrom(address _from, address _to,
            uint256 _tokenId) external payable;
14
15      function transferFrom(address _from, address _to,
            uint256 _tokenId) external payable;
16
17      function approve(address _approved, uint256 _tokenId)
            external payable;
18
19      function setApprovalForAll(address _operator, bool
            _approved) external;
20
21      function getApproved(uint256 _tokenId) external view
            returns (address);
22
23      function isApprovedForAll(address _owner, address
            _operator) external view returns (bool);
24  }
25
26  interface ERC165 {
27      function supportsInterface(bytes4 interfaceID) external
            view returns (bool);
28  }
```

Listagem 2.1: Interface mandatória do ERC721

O trecho de código apresentado na Listagem 2.1 é escrito em *solidity* e detalhado na sequência:

1. Essa interface só lista funções com o modificador *external*, esse modificador nativo do *solidity* indica que a função só pode ser acessada externamente ao contrato (funções de *API*);

2. Um contrato implementado com as regras da *EIP-721* é denominado *ERC721*;

3. Um *ERC721* publicado em alguma *EVM* armazena e permite meios de controles sobre todos os *tokens* individuais criados a partir de seu contrato. Observa-se no argumento *tokenId* que cada item criado tem um id único;

4. A listagem 2.1 também apresenta uma segunda interface a *ERC165*. Tal interface tem somente uma função (*supportsInterface*) que recebe como argumento de entrada um *interfaceId*, com o tipo *bytes4* nativo do *solidity*, e retorna um booleano indicando se a interface buscada é implementada ou não pelo contrato;

5. A função *safeTransferFrom* efetua a transferência de um *token* definido dentro do contrato de um *ERC721* para outro de forma segura e apresenta uma sobrecarga de operadores, que possibilita, que o usuário além de transferir um *token* envie uma mensagem;

6. A função *approve* e *setApprovalForAll* define meios para que um *EOA* de permissão a um endereço de operador transacionar todos os *tokens* que ele possui ou somente um *token* específico;

7. A função *ownerOf* permite verificar qual endereço é proprietário de um determinado *tokenId*.

No entanto a *EIP-721* não define como o usuário deve criar seus *tokens*. Por convenção o nome da função de criação de um *token* é *mint* e fica a critério do desenvolvedor como implementa-la.

A interface *ERC165*[95], mostrada na Listagem 2.1, recebe o parâmetro *interfaceID*. Esse parâmetro é utilizado para efetuar uma simples comparação que verifica os *IDs* das interfaces implementadas pelo contrato que utiliza a *ERC165*. Caso o identificador exista no contrato, a função *supportsInteface* retorna "*true*"e todas as funções externas implementadas por aquela interface podem ser usadas.

O cálculo para gerar um *interfaceID* é mostrado na Figura 2.14. Basicamente, é calculado um *hash* de 256 bits com *SHA-3* da assinatura das funções e na sequência é realizada uma operação *XOR* dos quatro primeiros *bytes* dos *hashes* [95].

Em relação as interfaces extras (extensões) do *ERC-721* [96], as principais são:

- *ERC721Burnable*: Essa extensão define boas práticas para permitir que um usuário destrua seu *token* seja por algum benefício garantido pelo criador do projeto ou para abdicar do direito a propriedade daquele item;

- *ERC721Enumerable*: Define uma indexação ordenada para os *NFTs* e também um *supply* máximo de *NFTs* diferentes por coleção;

- *ERC721Pausable*: É utilizado para impedir a transferência de *tokens* do contrato. Útil caso ocorra algum problema com o contrato mas muito criticado **ref**;

- *ERC721Metadata*: Essa extensão define funções para nome e simbolo de uma coleção e também uma interface para exposição dos metadados de um *token*;

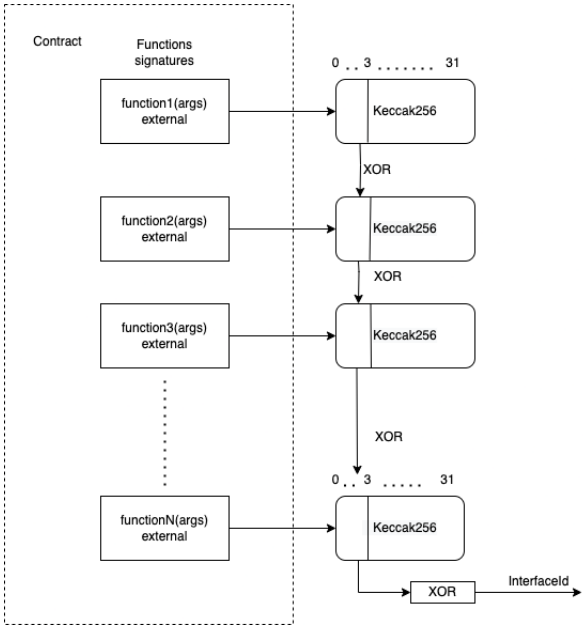


Figura 2.14: Diagrama do cálculo da *InterfaceID* Fonte: Autor

Dentre todas as extensões da *EIP-721*, a *ERC721Metadata* é a mais necessária para as aplicações de *marketplace* de*NFTs* pois através das funções mostradas na listagem 2.2 é possível nomear os itens que um *NFT* representa e também a coleção como um todo. Dentro da listagem 2.2, as funções mais importantes são mostrados em sequência:

- A função de *name* representa o nome dado a toda coleção, de forma as pessoas poderem reconhece-la;
- A função *tokenURI* envia o *URI*, no qual está determinado os metadados do *NFT*.

O URI que a função de *tokenURI* retorna é um arquivo *JSON*. O conteúdo é contido dentro do URI de forma a facilitar o acesso a eles por processos automatizados, tais quais o de uma aplicação de *marketplace*. Os campos base definidos pela *EIP*[91] são expostos na Listagem 2.3.

```
1 /// @title ERC-721 Non-Fungible Token Standard, optional
      metadata extension
2 /// @dev See https://eips.ethereum.org/EIPS/eip-721
3 ///  Note: the ERC-165 identifier for this interface is 0
      x5b5e139f.
```

```
interface ERC721Metadata /* is ERC721 */ {
    function name() external view returns (string _name);

    function symbol() external view returns (string _symbol)
        ;

    function tokenURI(uint256 _tokenId) external view
        returns (string);
}
```

Listagem 2.2: Interface MetadataERC721

```
{
    "title": "Asset Metadata",
    "type": "object",
    "properties": {
        "name": {
            "type": "string",
            "description": "Identifies the
            asset to which this NFT
            represents"
        },
        "description": {
            "type": "string",
            "description": "Describes the
            asset to which this NFT represents"
        },
        "image": {
            "type": "string",
            "description": "A URI pointing to a
               resource with mime
            type image/* representing the asset
               to which this NFT represents.
            Consider making any images
            at a width between 320 and
            1080 pixels and aspect
            ratio between 1.91:1 and 4:5
               inclusive." }}}
```

Listagem 2.3: JSON Metadata

Além dos campos indicados na Listagem 2.3, é normal existir um outro campo o de atributos, mesmo que não definido pela *EIP-721* foi padronizado pelas aplicações de *marketplace* *Rarible* e *OpenSea*. Boa

parte dos criadores seguem essa padronização do campo de atributos que consiste em um *array* de um objeto *JSON* com duas chaves/valores, sendo eles o *trait_type* e o *value*. Esses itens descrevem características do *NFT* que são responsabilidades do criador de cada item de *NFT* de serem gerados e formatados de acordo.

Existem complexidades a mais que envolvem a implementação real de um contrato para *NFTs* como o local em que são hospedados os arquivos de metadados e as imagens. A recomendação é usar o *IPFS* [97] para deixar esses dados imutáveis e descentralizados e, também, a estruturação das funções, principalmente a de *mint* e *burn* ou o valor do campo *nftID*. No entanto, cada um desses tópicos apresentam seus *trade-offs* e fogem do escopo deste trabalho.

2.4.3.2 *EIP-20*

O padrão da *EIP-20*[90] foi o primeiro a ser desenvolvido entre todos os outros e remete ao caso mais comum na *blockchain*, os *tokens* fungíveis, os quais são o caso mais similar a moeda corrente ou ações ordinárias em um mercado de valores. Existem diversos mecanismos que atuam atribuindo valor a esses *tokens* que não são diretamente relacionados a esse trabalho, mas são importantes para entender a legitimidade de um sistema *on chain*. As plataformas de corretora de cripto centralizadas possuem sistemas similares ao de uma corretora tradicional. No entanto, elas permitem a troca de criptoativos (sejam eles *tokens ERC-20* ou tokens nativos de alguma *blockchain*) por moedas fiduciárias. Alguns exemplos são a Binance e a FTX, as quais possuem pares de troca entre moedas fiduciárias e *stablecoins* [98]. As *stablecoins* são os criptoativos lastreados em alguma moeda fiduciária (como o dólar, real, euro e etc). Após converter essas moedas fiduciárias em *stablecoins*, é possível usar essas últimas para comprar diversos *tokens* fungíveis de projetos. Muitos desses *tokens* usam o padrão *ERC-20*, mas com um propósito bem diferente - atribuir valor a um projeto, indicar sua capitalização de mercado (mais similar com a compra de ações de alguma empresa), dentre outras. Os *tokens* fungíveis de projetos podem ser obtidos de forma descentralizada também, através das corretoras descentralizadas que são protocolos desenvolvidos da mesma forma que o *marketplace* desenvolvido neste projeto (utilizando alguma *blockchain* compatível com *EVM*, na maioria dos casos, normalmente o compilador *solc* e sendo publicado nas redes almejadas). No entanto, as corretoras descentralizadas possuem uma lógica bem diferente, utilizando os conceitos dos *AMM* (Automated Market Makers) [99], em sua grande maioria. Alguns exemplos de protocolos de corretora descentralizada

para *EVMs* são Uniswap, Balancer e Sushi. Não entraremos muito a fundo nesses tópicos já que ele somente é importante para entender as diferenças na forma de atribuição de valor entre um *token* fungível e um não fungível, sendo que no segundo caso o valor é atribuído de acordo com uma percepção social de qualidade e valor, uma vez que os itens são únicos. De acordo com a *EIP-20* que define a padronização do *ERC-20*:

> "Uma interface padrão permite que qualquer *token* no *Ethereum* seja reutilizado por outros aplicativos: de carteiras á *exchanges* descentralizadas"[90].

Da interface definida na *EIP-20*, ilustrada na listagem 2.4, para *tokens* fungíveis pode-se perceber que o *ERC-721* herda muitas das funcionalidades inicialmente pensadas para o padrão de *token* fungível, no entanto apresenta algumas diferenças cruciais devido a suas diferenças conceituais.

```
1  /**
2   * @title ERC20 interface
3   * @dev see https://github.com/ethereum/EIPs/issues/20
4   */
5  interface IERC20 {
6    function totalSupply() external view returns (uint256);
7
8    function balanceOf(address who) external view returns (
         uint256);
9
10   function allowance(address owner, address spender)
11     external view returns (uint256);
12
13   function transfer(address to, uint256 value) external
         returns (bool);
14
15   function approve(address spender, uint256 value)
16     external returns (bool);
17
18   function transferFrom(address from, address to, uint256
         value)
19     external returns (bool);
20
21   event Transfer(
22     address indexed from,
23     address indexed to,
24     uint256 value
25   );
26
27   event Approval(
28     address indexed owner,
```

```
29      address indexed spender,
30      uint256 value
31    );
32  }
```

Listagem 2.4: Interface do ERC20

Devido as diferenças entre um *token* fungível e não fungível, temos algumas diferenças nas funções e argumentos expostos pela interface. A lista abaixo comenta um pouco sobre cada função e sua motivação [90]:

- O *totalSupply* retorna o número de *tokens* em circulação do ativo;
- O *balanceOf*, como se repara não inclui um campo de *tokenId*, pois todos os *tokens* emitidos por um *ERC-20* são iguais, logo ele só retorna o saldo do usuário;
- A função de *approve* tem o mesmo objetivo da função de mesmo nome da *EIP-721* no entanto agora só se passa o endereço do operador (chamado de *spender*) e o montante que ele é permitido a gastar dos *tokens do usuário*;
- A função de *transferFrom* também tem o mesmo propósito que a definida pela *EIP-721*;
- A função *allowance* retorna o montante que um operador é permitido a gastar dos *tokens* de um endereço ainda.

A função de *mint* também fica aberta para o desenvolvedor. No entanto, para os *ERC-20*, salvo casos de *stablecoin*, é uma função relativamente mais simples e a vasta maioria dos projetos geram o *supply* máximo de um *token* fungível no momento de sua criação. No entanto, no caso do *ERC-20*, não existe uma extensão para definir o *supply* máximo, normalmente é usada uma constante definida no momento de criação do contrato.

Uma função opcional, que na prática é muito usada, é a função de leitura *decimals*, na qual o criador do *token* expõe quantas casas decimais o *token* fungível dele aceita. Este valor em *EVMs* pode variar de uma casa decimal a dezoito. A maioria dos projetos usa 18 por *default*, o mesmo que qualquer moeda nativa em *EVM*. No entanto, projetos de *stablecoins* às vezes representam menos casas decimais para retratar com mais fidelidade uma moeda fiduciária.

Repare que a interface *ERC-165* [95] não é definida no contrato base do *ERC-20*, mas praticamente todos os contratos de *token* fungível

recentes a implementam. Isso vem do fato que a *ERC-165* foi criada muito após a *ERC-20* ter sido aprovada. O padrão para interfaces foi criado quando os ecossistemas de *DeFi* começaram a eclodir e a interação entre contratos começou a se tornar uma grande necessidade.

2.4.3.3 *ERC-1155*

Por fim chegamos ao último e mais recente padrão de *token* definido para uso em ecossistemas de *EVM*. O *ERC-1155*[92] é um modelo de *token* semi-fungível. É muito usado para itens colecionáveis repetíveis como, por exemplo, cartas em um jogo no qual podemos ter vários itens iguais de um mesmo tipo. De acordo com a própria referência do *ERC-1155*:

> "Esse padrão descreve uma interface de contrato inteligente que pode representar qualquer número de tipos de *token* fungíveis e não fungíveis. Os padrões existentes, como o *ERC-20*, exigem a implantação de contratos separados por tipo de *token*. O ID do *token* do padrão *ERC-721* é um índice único não fungível e o grupo desses não fungíveis é implantado como um único contrato com configurações para toda a coleção. Em contraste, o *ERC-1155 Multi Token Standard* permite que cada ID de *token* represente um novo tipo de token configurável, que pode ter seus próprios metadados, fornecimento e outros atributos.
>
> O argumento _id contido no conjunto de argumentos de cada função indica um *token* específico ou tipo de *token* em uma transação. "[92]

Esse *token* apresenta uma série de funcionalidades um pouco mais complexas em relação aos outros, uma vez que ele é um híbrido que possibilita as funcionalidades de um *ERC-20* e de um *ERC-721* juntos. Sua interface base apresenta a necessidade de exposição das funcionalidades indicadas na listagem 2.5. A maior diferença nesse modelo são as funcionalidades de agrupamento possível trazidas pelas funções de *batch* que permitem operações com inúmeros diferentes *tokens* dentro do contrato de um *ERC-1155*.

```
1 /**
2     @title ERC-1155 Multi Token Standard
3     @dev See https://eips.ethereum.org/EIPS/eip-1155
4     Note: The ERC-165 identifier for this interface is 0
          xd9b67a26.
5  */
```

```
6  interface ERC1155 /* is ERC165 */ {
7
8      event TransferSingle(address indexed _operator, address
           indexed _from, address indexed _to, uint256 _id,
           uint256 _value);
9
10     event TransferBatch(address indexed _operator, address
           indexed _from, address indexed _to, uint256[] _ids,
           uint256[] _values);
11
12     event ApprovalForAll(address indexed _owner, address
           indexed _operator, bool _approved);
13
14     event URI(string _value, uint256 indexed _id);
15
16     function safeTransferFrom(address _from, address _to,
           uint256 _id, uint256 _value, bytes calldata _data)
           external;
17
18     function safeBatchTransferFrom(address _from, address
           _to, uint256[] calldata _ids, uint256[] calldata
           _values, bytes calldata _data) external;
19
20     function balanceOf(address _owner, uint256 _id) external
            view returns (uint256);
21
22     function balanceOfBatch(address[] calldata _owners,
           uint256[] calldata _ids) external view returns (
           uint256[] memory);
23
24     function setApprovalForAll(address _operator, bool
           _approved) external;
25
26     function isApprovedForAll(address _owner, address
           _operator) external view returns (bool);
27 }
```

Listagem 2.5: Interface do ERC1155

Contratos seguindo as normas da *EIP-1155*, apesar de não serem considerados *NFTs* pela própria definição do termo, têm seus contratos chamados de coleções e são muito utilizados como contratos para serem comercializados em *marketplaces* de *NFTs* principalmente nos *marketplaces* mais focados na venda de *skins* para jogos.

2.4.4 Definições Gerais

A Figura 2.15 mostra os blocos que são necessário para qualquer protocolo de *marketplace* de *NFTs*.

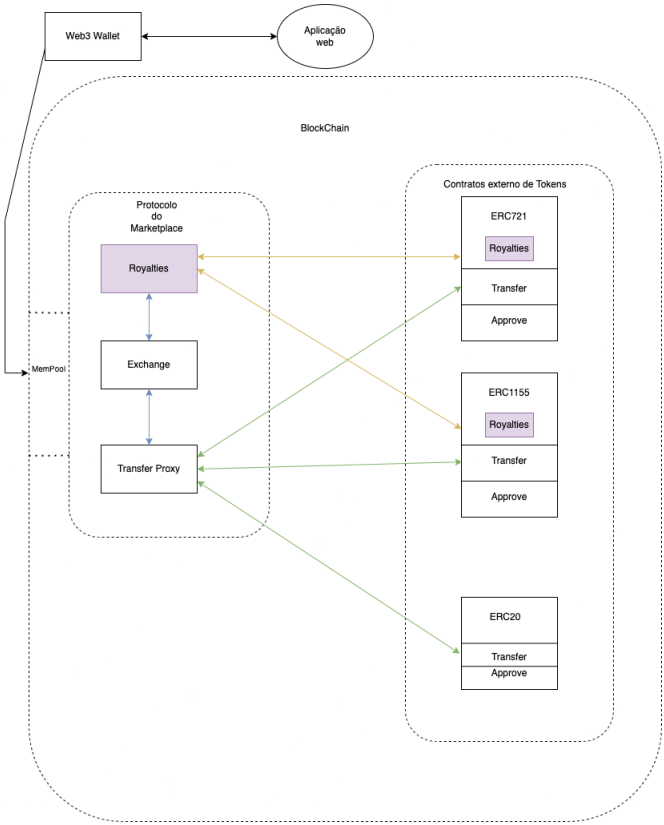


Figura 2.15: Diagrama de blocos de um protocolo de *marketplace*. Fonte: Autor

No diagrama da Figura 2.15, tem-se a Aplicação *web* que interage com alguma *Web3 wallet*, como a *Metamask*(2.5.4) como ponto de partida para interagir com o endereço dos contratos de um protocolo de *marketplace* na *blockchain*. Essa não é uma configuração mandatória. Qualquer serviço com acesso a um *RPC*(2.1.8) e *private keys* de alguma *EOA* ou outro contrato dentro da própria *blockchain* conseguiriam efetuar a chamada para os contratos do protocolo. No entanto, é a configuração mais usada para esse tipo de protocolo, pois as aplicações *web* construídas em cima dos protocolos facilitam muito o uso pela vasta maioria dos clientes de perfil não técnico.

O diagrama da Figura 2.15 mostra 3 blocos centrais que são ne-

cessários para o desenvolvimento de um protocolo de *marketplace*: o de *Exchange*, *TransferProxy* e *Royalties*.

O bloco de *Exchange* é o ponto central deste tipo de protocolo, sendo responsável por entender as ordens de compra ou venda, quais itens estão sendo trocados, quem deve ser pago, etc. É o bloco mais complexo do protocolo e será explicado em detalhes no desenvolvimento. É importante entender que, como em qualquer *Exchange*, em um protocolo de *marketplace* também são necessárias duas partes, a do criador da ordem (*maker*) querendo comprar ou vender algum item e a contraparte tomadora (*taker*) que aceita a ordem criada.

O bloco de *TranferProxy* interage com o bloco de *Exchange* e é responsável por transferir os *tokens* para seus novos donos, efetuando a troca.

O bloco de *Royalties* é responsável por identificar se existem *royalties* vinculados a um *token* ou coleção de *ERC-721* ou *ERC-1155*. Esse bloco está grifado em roxo, bem como as interfaces de *royalties* nos *tokens*, pois, como é mostrado na subseção de *tokens*, não existe uma extensão padrão em nenhum dos *tokens* para uma interface de *royalties*. Logo, a implementação dele varia muito e nem todos os protocolos implementam uma interface de *royalties*. Esses fatos serão vistos nas próximas subseções que abordam os casos específicos dos protocolos de base para este trabalho.

A linha tracejada indicando a *memPool* se refere ao fato que, conforme mostrado nas seções de *blockchain* (Seção 2.1) e *EVMs* (Seção 2.2), toda transação na *blockchain* precisa ser minerada (PoW) ou validada (PoS) para ser adicionada na rede e, no caso de uma *blockchain* com *EVM*, é preciso gastar o processamento de um minerador ou validador para ser executada na máquina virtual. Assim, toda transação enviada vai para uma fila, chamada de *memory pool*, e os *full-nodes* da rede escolhem qual transação pegar de acordo com as taxas sendo pagas por elas (no caso de *EVMs*, a quantidade de gás enviado). Somente após um *fullnode* executar a transação, adicioná-la na *blockchain* e o bloco ser confirmado por 51% da rede, as transferências feitas pelo protocolo de *marketplace* são consideradas completas.

Quanto ao bloco de *Exchange*, existe um ponto muito importante para seu desenvolvimento, que é definir se a *Exchange* ira possuir um *orderbook onChain* ou um *orderbook offChain*. Essa definição muda muito a estrutura implementada dentro do bloco. Os dois casos e suas diferenças são explicados nos tópicos em sequência:

- *Exchange* com *orderbook onchain*: Toda as ordens criadas ficam armazenadas no próprio contrato da *Exchange*, a informação fica

escrita na própria *blockchain*. Essa implementação é muito mais custosa pois armazenar informação na *blockchain* é caro;

- *Exchange* com *orderbook offchain*: As ordens são armazenadas em algum banco de dados controlado pela aplicação escrita em cima do protocolo do *marketplace*, utilizam a *EIP-712*, discutida na seção 2.2, para efetuar assinaturas de uma forma segura e prática. Esse é considerado o mecanismo mais eficiente, por ser barato (exonera a *blockchain* de ter de indexar informações extras sobre a ordem) e seguro.

Tanto os contratos da *Wyvern Exchange*, quanto os da *Rarible-protocol* utilizam modelos de *orderbook offchain* em seus contratos de *Exchange*. A Figura 2.16 ilustra algumas especificações necessárias no diagrama de blocos genérico da Figura 2.15 quando se tem um *orderbook offchain*.

A Figura 2.16 mostra que os blocos necessários para o desenvolvimento do protocolo são os mesmos considerando um *orderbook offchain* e que toda a parte dentro da *blockchain* continua exatamente igual. No entanto a diferença principal é vista no nível da aplicação em que fica explicitado que o *maker* e o *taker* (dois clientes da aplicação) tem papéis distintos e bem definidos para o processo de troca possibilitado pelo protocolo. A Figura 2.16 também deixa clara a necessidade de um banco de dados para armazenar as assinaturas e ordens criadas pelo *maker* para que o *taker* possa interagir com o protocolo.

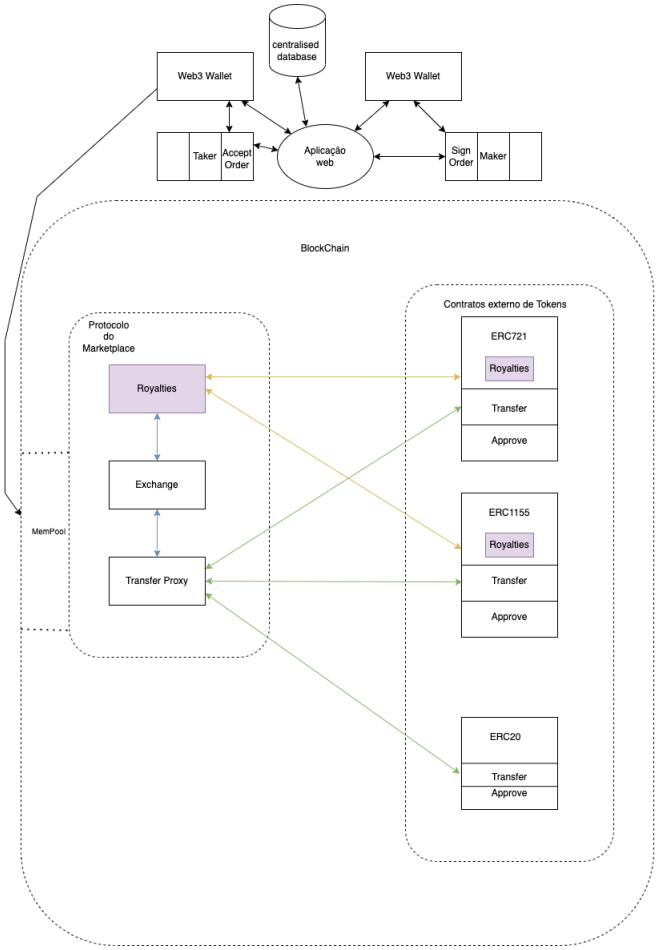


Figura 2.16: Diagrama de blocos *offchain orderbook*. Fonte: Autor

2.4.5 Proxies

Os contratos de *proxy* de transferência, ilustrados no diagrama da Figura 2.15 são responsáveis por transferir os *tokens* entre os usuários interagindo com o *marketplace*. Para seu funcionamento basta alguns contratos simples que implementam as interfaces de *Transfer* dos tipos de *tokens* comercializados em um *marketplace*. Esses contratos são publicados isoladamente e é necessário que o dono de um determinado *token* use a função de *approve* no contrato do *token* que ele possui

indicando como operador o contrato de *proxy* implementado pelo *marketplace* com o qual ele está interagindo.

2.4.6 Ordens

Todo *marketplace* de *NFT* precisa implementar uma estrutura de dados para definir seu modelo de ordens. No próximo capítulo, é explicada a estrutura de ordens do protocolo desenvolvido neste trabalho, que é baseada nas estruturas de ordens implementadas pelo protocolo *Rarible* que foi escolhido por conta de estar utilizando uma versão mais recente do *solc* que permite o uso de *arrays*, o que não era possível na versão do *solc* utilizada pelos contratos da *Wyvern*. A documentação da *Wyvern Exchange* traz uma boa definição para a necessidade de uma estrutura de ordem, os componentes necessários e também como eles são aplicados em um protocolo de *marketplace* de forma geral:

> "*Wyvern* é um protocolo de troca descentralizada de primeira ordem. Protocolos existentes comparáveis, como *Etherdelta*, 0x e *Dexy*, são de ordem zero: cada ordem especifica uma negociação desejada de dois ativos discretos (geralmente dois *tokens* em uma proporção específica e um valor máximo). Em vez disso, as ordens *Wyvern* especificam predicados sobre transições de estado: uma ordem é uma função que mapeia uma chamada feita pelo *maker*, uma chamada feita pela contraparte e metadados da ordem para um booleano (se a ordem é elegível para *trade* ou não). Esses predicados são arbitrários - qualquer ativo ou qualquer combinação de ativos representável no *Ethereum* pode ser trocado com uma ordem *Wyvern* - e, de fato, *Wyvern* pode instanciar todos os protocolos mencionados acima."[100]

Wyvern foi o primeiro protocolo a permitir que uma ordem definisse a comercialização de um ativo por qualquer outro desde que o *maker* crie e assine uma ordem. O protocolo da *Rarible* também tem as mesmas capacidades.

2.4.7 Royalties

Os *Royalties* definem uma taxa a ser paga para o criador de um determinado item ou coleção não fungível ou semi-fungível. O *Royalty* é pago para o criador original de um *asset* todas as vezes que ele é vendido. A venda recursiva trará as taxas de royalties ao criador no caso de revenda

de sua criação ou inovação. Existem diversas formas de efetuar o pagamento de *royalties* para um criador e, como visto nas discussões sobre *EIPs* de *tokens*, não existe uma extensão oficial para a implementação dos *Royalties* em conjunto com um *token*. Recentemente foi publicada uma *EIP* para definir uma interface padrão para *royalties*. No entanto essa EIP ainda não é utilizada na prática pela maioria dos protocolos de *marketplace* e será discutida no próximo capítulo. O bloco para uma interface de *royalties*, ilustrado na Figura 2.15, foi a inovação que os protocolos da *Rarible* trouxeram para o desenvolvimento de protocolos de *marketplace*, sendo o primeiro protocolo a desenvolver essas interfaces muito importantes. Os contratos da *Wyvern Exchange* não apresentam o bloco de *royalties*. Logo seu funcionamento é exatamente como o ilustrado na Figura 2.15, porém sem o bloco de Royalties no protocolo.

2.5 Ferramentas

2.5.1 Solidity

É uma linguagem de programação orientada a objetos de alto nível para a implementação de *smart contracts* [101]. Trata-se de uma linguagem *curly-bracket* que separa trechos de códigos por colchete, possuindo influência nas linguagens C++, *Python*, *JavaScript*. A *Solidity* é executada pela *Ethereum Virtual Machine* (*EVM*). Da própria documentação da linguagem:

> "*Solidity is statically typed, supports inheritance, libraries and complex user-defined types among other features.*"[101]

A linguagem *solidity* permite operações de *loops*, condicionais e etc, tais como qualquer outra linguagem de alto nível. Os códigos criados com *solidity* precisam ser compilados para gerar um *bytecode* interpretável pela maquina virtual do *Ethereum*. O compilador mais estável e popular sendo utilizado hoje em dia é o *solc*.

Existem algumas bibliotecas com *templates* de contratos e interfaces, implementando a maioria das *ERCs* e *EIPs* definidas de forma eficiente e segura. A *biblioteca* auxiliar mais utilizada para *solidity* atualmente é a *OpenZeppelin* [93]. Para utilizar uma biblioteca com *solidity* basta instalar ela no seu ambiente de desenvolvimento e dar um *import*.

Com esta linguagem podemos criar contratos para várias finalidades diferentes dentro da *blockchain* alguns exemplos são: votação, financi-

amento coletivo, carteiras-cofre (precisam de duas ou mais assinaturas para fazer uma transação), protocolos de leilões, corretoras e *market-places* sem necessidade de intermediários.

Quando se dá um *deploy* em contratos com o *Solidity* em alguma rede *EVM*, é recomendado utilizar a última versão atualizada dessa linguagem por conta dos *upgrades* na *EVM*, bem como novos recursos e correções de *bugs* são introduzidos regularmente [102], além de se atualizarem juntamente com modificações recorrentes das redes do *Ethereum* ou compatíveis como as *Layers 2* [78], *SideChains* [51] e outras *blockchains* compatíveis com *EVMS*

O site oficial apresenta uma documentação bem detalhada sobre a linguagem *solidity* e do *OpenZeppelin* de como implementar os padrões definidos pela fundação *Ethereum* [101].

O *Solidity* é uma lingaguem nova, surgiu em 2014 com uma proposta de *Gavin Wood*, com *Christian Reirwiessner* e *Alex Beregszaszi* liderando a equipe de desenvolvimento do *Ethereum*. [103]

Outras *blockchains*, além da *Ethereum*, passaram a utilizar a *Solidity* por conta da facilidade de desenvolvimento como por exemplo a *Monax* e a *Hyperledger Burrow* [104]. Além disso, essa linguagem gera certas controvérsias relacionando a sua usabilidade, a qual permite criar aplicativos descentralizados facilmente sendo a segurança destes aplicativos bastante questionável, ainda mais quando se leva em conta o montante armazenado nesses protocolos atualmente. Para um exemplo prático, no primeiro semestre de 2021 atingiu um total de 89,54 biliões de dólares, um aumento de mais de 7 vezes em relação ao mesmo semestre de 2020 [105].

Claro que os lideres desta lista são empresas já estabelecidas com avaliações das melhores firmas de auditorias que existem. No entanto, o que preocupa é a facilidade de publicação de um protocolo desses que permite a distribuição global de produtos financeiros com facilidade. Por este motivo um ditado muito disseminado no ambiente de "cripto"é *do your own research* (faça sua própria pesquisa) [106], uma vez que dado esta facilidade que permite grandes avanços e modernização dos instrumentos e sistemas financeiros de forma dinâmica e eficiente, o sistema também acaba infelizmente sendo contaminado por esquemas de pirâmides, golpes e fraudes dos mais diversos tipos [107].

2.5.2 Hardhat

Existem diversos *frameworks* para desenvolvimentos de *smart contracts* [108]. O escolhido para este trabalho foi o *Hardhat* devido a diversas

vantagens perceptíveis desse *framework* para desenvolver *smart contracts*.

Hardhat é um ambiente de desenvolvimento para compilar, implantar, testar e depurar sistemas que rodam com *EVMS*. Ele serve para auxiliar os desenvolvedores no gerenciamento e automatização das tarefas, adicionando facilmente novas funcionalidades no processo da construção dos *smart contracts* e de aplicativos descentralizados.

O ambiente do *Hardhat* vem embutido com a *Hardhat Network* [109] que é basicamente uma rede local com o sistema do *Ethereum* no qual é possível de forma muito dinâmica e rápida testar a execução de códigos compilados com *Solc*, o compilador principal de *solidity*. Após instalado em seu sistema operacional, ele utiliza o *Hardhat Runner*[110] que é um cliente de linha de comando que permite executar diversas tarefas essenciais ao desenvolvimento de protocolos para aplicações descentralizadas de forma ágil. Por exemplo, o comando *compile* rapidamente compila e gera os executáveis compatíveis com *EVMS*.

Em conjunto com a *Hardhat network*, que é possível de ser executada em terminal com o comando *npx harhdat node*, caso você opte por utilizar *npx* [111], é possível também simular a execução dos códigos *solidity* em uma rede emulando seu comportamento em redes reais.

Por fim, muitas finalidades do *Hardhat* vem de *plug-ins* e o desenvolvedor tem a opção de escolher quais usar.

2.5.3 Web3

Web 3 [112] é uma *API* em *javascript* desenvolvida para interagir com a *API* de mesmo nome exposta pelos nodos de uma *EVM* que aceita os comandos em *JSON-RPC* [113]. Ela tem o propósito de que os chamados *Dapps* [114], que são basicamente aplicações web descentralizadas, consigam de uma forma rápida e eficiente se conectar a um provedor de *Ethereum* [115] e interagir com informações *onChain*, incluindo a chamada de execução de rotinas e padrões em *smart contracts*.

A biblioteca do web3.js é uma coleção de módulos que contém funcionalidades para o *Ethereum ecosystem*. Como se pode ver a seguir:

- *eth*: para os *blockchains* compatíveis com *EVMS* e por consequência para os *smart contracts*.
- *shh*: para os protocolos *whisper*, servindo para comunicar e transmitir o p2p.
- *bzz*: é para o protocolo *swarm*, o armazenamento descentralizado de arquivos.

- *utils*: contém funções úteis auxiliares para desenvolvedores de *Dapps*.

2.5.4 Metamask

A *Metamask* é o exemplo mais popular [116] de *wallet* não custodial [117] de extensão de navegador de cripto para *tokens* de *blockchains* compatíveis com *EVM*. Existem diversas implementações e modelos de *wallet* para as mais variadas *blockchains*, como a *Pali Wallet* que é desenvolvida para as *blockchain* da *Syscoin*[118] e esta passando por uma expansão para suportar também *blockchains* compatíveis com *EVM*. No entanto, devido à quantidade de *chains* compatíveis com *EVM* e do fato de todas elas seguirem o mesmo padrão de interfaces *web3* expostas através de *APIs REST* e *Websockets*, uma *wallet* desenvolvida para uma *EVM* pode facilmente ser expandida para dar suporte a todas. Isto foi muito bem executado pelo time da *Metamask*. Algumas imagens da *aplicação* são ilustradas nas Figuras 2.17. Todas as carteiras de cripto utilizam a criptografia adequada de acordo com o modelo de *chain* que desejam suportar No caso da *Metamask* são os modelos explicitados nas seção 2.2, a *kecakk256*.

O componente principal de qualquer *wallet* cripto é expor métodos de assinatura digital, *broadcasting* de transações para a rede com a qual elas integram, indexar os dados públicos (proveniente por *queries* a *API web3* usando-se os endereços públicos de uma *wallet*) e também proteger a chave privada nunca permitindo a exposição da mesma a nenhuma aplicação terceira, sem confirmação reforçada do dono da *wallet*.

A *metamask* é um exemplo de carteira de cripto não custodial, isto é a chave privada do usuário e *seedPhrase* só ficam armazenadas em seu dispositivo eletrônico, nunca sendo enviadas para algum servidor ou aplicação [117], este é o padrão da industria em cripto que deu origem ao ditado "não são suas chaves, não são suas moedas (*not your keys, not your coins*)"[119]. No entanto, esse é um ditado polêmico e existem contrapontos como o do fato de guardar seu próprio dinheiro ser muito arriscado para pessoas em geral [120].

O blog *Herong Yang* [121], que apresenta uma série de livros gratuitos sobre tópicos gerais em computação, traz alguns pontos chaves interessantes sobre a *Metamask*. O primeiro tópico é referente ao fato que antes da popularização das soluções de *wallets* de cripto e serviços de *RPC* (*APIs* e *Websockets* expondo objetos *web3*, no caso do *Ethereum*), para conseguir interagir com uma *blockchain*, era necessário

rodar um nodo completo da rede em sua própria máquina.

Os tópicos trazidos por *Herong Yang* quanto aos recursos disponibilizados pela ferramenta são [122]:

- *MetaMask* é uma ponte que permite que você visite a web distribuída de amanhã em seu navegador hoje. Ele permite que você execute *dApps Ethereum* diretamente no seu navegador sem executar um nó *Ethereum* completo.
- O *MetaMask* inclui um cofre de identidade seguro, fornecendo uma interface de usuário para gerenciar suas identidades em diferentes sites e assinar transações de blockchain
- Você pode instalar o complemento *MetaMask* no *Chrome*, *Firefox*, Opera e no novo navegador Brave. Se você é um desenvolvedor, pode começar a desenvolver com o *MetaMask* hoje mesmo.
- A missão da *MetaMask* é tornar o *Ethereum* o mais fácil de usar para o maior número possível de pessoas.

E continuando, para começar a utilizar esta ferramenta os seguintes tópicos são uma boa base na concepção do autor do livro [122]:

- *MetaMask* é uma carteira da *Ethereum* baseada em navegador fornecida por *metamask.io*;
- Você pode instalar a extensão da *MetaMask* no *Google Chrome* com muita facilidade;
- Sua carteira *MetaMask* é protegida apenas por uma senha;
- *MetaMask* permite que você crie 1 conta *Ether* para todas as redes *Ethereum*;
- Você pode obter alguns *Ethers* gratuitos na rede de teste *Ropsten Ethereum* em *faucet.metamask.io* ou em *faucet.dimensions.network*;
- Você pode conectar o *MetaMask* ao nó *"geth"* local com o *URL RPC*: *http://localhost:8545*;
- Você pode receber e enviar *Ethers* com a *MetaMask* uma vez conectado à rede;
- *MetaMask* mantém um arquivo de *logs* de estado para registrar o histórico de cada transação de saída;

- Você pode exportar a chave privada da sua conta *MetaMask* e importar para outras carteiras *Ether*.

Um dos pontos centrais, entretanto, que faz a *Metamask* ser tão utilizada é a exposição, no ambiente do navegador onde a extensão esta instalada, de um objeto *web3* injetado e muito bem documentado com a qual *Dapps* (aplicativos que usam protocolos em *blockchain*) conseguem interagir facilmente enviando as transações e requisição de assinatura para que os usuários possam interagir com os protocolos usados por esse tipo de aplicação.

O objeto *web3* injetado pela *metamask* possui diversos métodos. Aqui comentaremos somente sobre dois deles muito relevantes para este trabalho. Os demais métodos podem ser conferidos na documentação oficial [123]. São eles:

1. **eth_sendTransaction**: As transações são uma ação formal em um blockchain. Eles são sempre iniciados no *MetaMask* com uma chamada ao método *eth_sendTransaction*. Eles podem envolver um simples envio de *ether*, podem resultar no envio de *tokens*, na criação de um novo contrato inteligente ou na mudança de estado no blockchain de várias maneiras. Eles são sempre iniciados por uma assinatura de uma conta externa ou um simples par de chaves.

2. **signTypedData_v4**: O método *signTypedData_v4* atualmente representa a versão mais recente da especificação *EIP-712*, com suporte adicional para *arrays* e com uma correção de última hora para a forma como os *structs* são codificados.

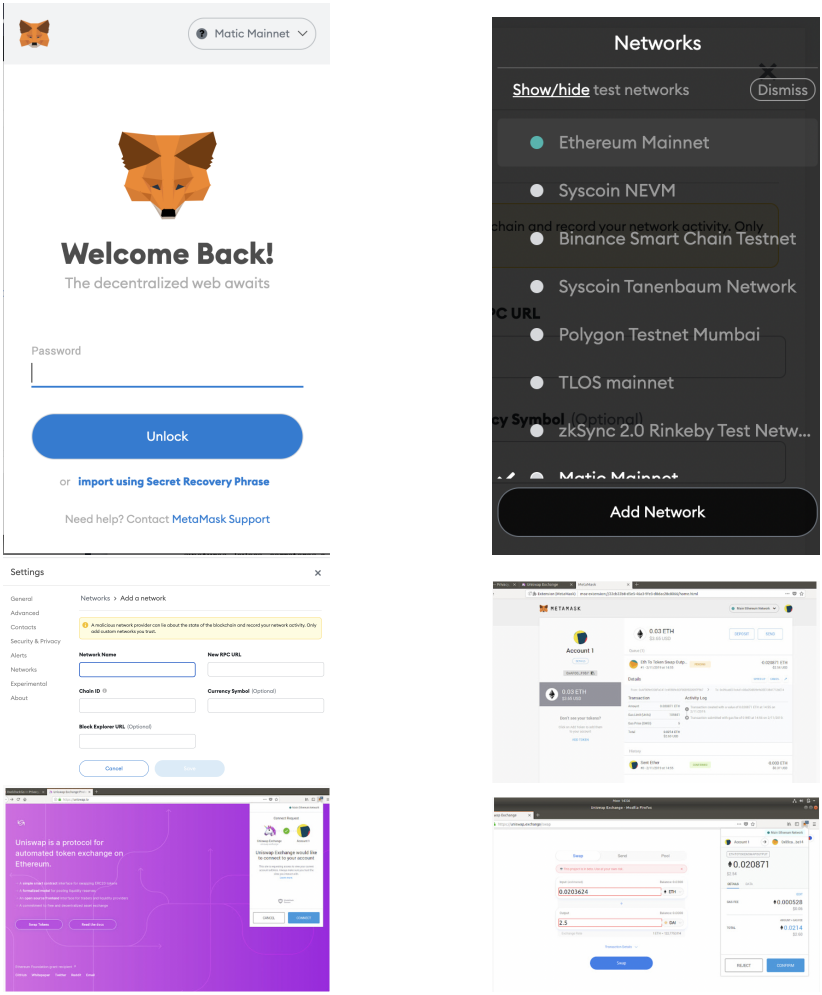


Figura 2.17: Telas da *Metamask*. Fonte: Autor

CAPÍTULO 3

Desenvolvimento

Neste capítulo, é apresentado o desenvolvimento deste trabalho. Inicialmente uma visão geral do projeto é apresentada detalhando os conceitos de *marketplace* de *NFTs* e *blockchain* utilizados. Na sequência, são detalhados os modelos de *token*, *proxies* de *tokens*, estrutura de ordens e *royalties* utilizados pelo *marketplace*. Finalmente é descrito o processo completo para se efetuar uma troca no *marketplace* e o funcionamento do bloco central de *exchange*.

3.1 Visão Geral

O protocolo de *marketplace* desenvolvido neste trabalho possibilita vendas, compras e ofertas de *nfts* por qualquer outro *token* fungível de forma automática. Tal desenvolvimento é baseado nos protocolos de referência (*Wyvern Exchange* e *Rarible Protocol*) que foram apresentados no Capítulo 2. O protocolo desenvolvido neste trabalho é denominado

de *Luxy*.

O *Luxy* foi desenvolvido utilizando a linguagem *solidity* e as bibliotecas da *OpenZeppelin* que são as mesmas bibliotecas utilizads pelos protocolos de referência. Outra ferramenta chave para o desenvolvimento do protocolo é o *Hardhat*, também comentado no capítulo 2, utilizado para compilar os códigos, gerar e otimizar os *bytecodes* de saída do processo de compilação, efetuar os testes de lógica e unitários (que foram escritos em *javascript*). O *Hardhat* também foi utilizado para publicar os contratos para teste na rede Mumbai e para utilização em produção na rede *Polygon*.

Os códigos finais do *Luxy* estão *open-source* no *Github*. Os testes relevantes ao escopo deste trabalho estão na *branch main*. As outras *branchs* do *git* são dedicadas ao desenvolvimento de melhorias e adaptações futuras do protocolo que serão discutidas na conclusão.

Pelo *GitHub* é possível clonar o *Luxy* e efetuar os testes de lógica e negócios para comprovar o seu devido funcionamento. Na pasta *Audit* do repositório no *GitHub* citado é apresentado o resultado final da auditoria de segurança do protocolo realizado pela Cyrex, uma respeitada firma de segurança de *software* e *ethical hacking*.

É possível interagir com o protocolo desenvolvido na rede de teste (Mumbai) e *mainnet* da *Polygon*, utilizando a *metamask* ou qualquer outra *wallet web3* suportada pelo *explorer* oficial da *Polygon*. Os links apresentados na sequência mostram a implementação dos protocolos nas redes *mainnet* e Mumbai da *Polygon*:

- Link implementação *Polygon Mainnet*.
- Link implementação *Polygon Testnet (Mumbai)*.

A Figura 3.1 apresenta o diagrama de blocos do protocolo desenvolvido neste trabalho. A partir de tal diagrama, observa-se que o protocolo *Luxy* implementa todos os blocos definidos na seção 2.4. Observa-se ainda na Figura 3.1 que *LuxyCore* é o nome dado ao bloco de *Exchange*, e que que *LuxyRoyalties* é o nome dado ao bloco de *Royalties* (Figura 3.1). Também é possível extrair da Figura 3.1 que o protocolo desenvolvido implementa um modelo de *orderbook offchain* (veja Seção 2.4).

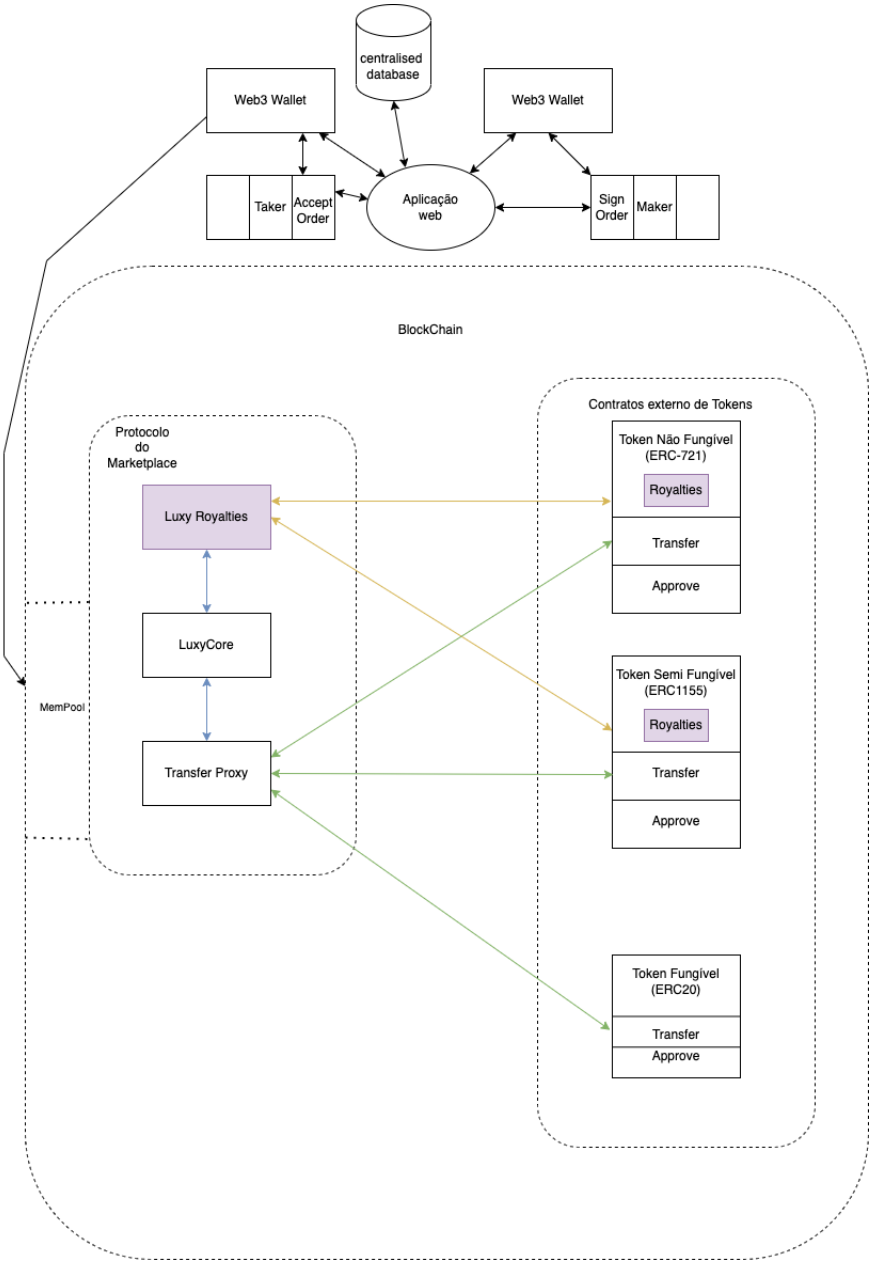


Figura 3.1: Diagrama de blocos do Luxy. Fonte: Autor

Os principais diferenciais do protocolo implementado são customizações e atualizações:

- As customizações foram feitas nos blocos de *Royalties* e *Exchange* (Figura 2.16) para permitir funcionalidades extras e ampliar o suporte a mais interfaces de Royalties, as quais serão comentadas nas Secções 3.4 e 3.5.

- As atualizações foram feitas em relação aos protocolos de referência. Por exemplo a atualização das *EIPs* e *ERCs* para usar a versão mais atual da biblioteca da *OpenZeppelin*, que traz formas mais eficientes de implementação das *ERCs*. Essas atualizações trazem *breaking changes* e espaço para otimizações de acordo com as melhores práticas de desenvolvimento em *solidity*.

O diagrama de blocos 3.1 parte da premissa que uma venda é sempre composta somente de duas partes, o *maker* e o *taker*. O *maker* é o criador da ordem (seja uma oferta de venda criada pelo dono do *NFT* ou uma oferta de compra criada por quem almeja ter o *NFT*). O *taker* é a pessoa que consome uma ordem criada e executa o negócio na *blockchain*.

Como já mencionado, o protocolo implementado neste trabalho utiliza um *orderbook offchain*. Para criar a ordem o *maker* necessita executar a assinatura digital (que é baseada na *EIP-712*, seção 2.2), aprovar o contrato de *proxy* do *marketplace* (bloco de *transferProxy*) e gastar os fundos ou *NFT* que serão usados quando uma contra-parte (o *taker*) aceitar sua oferta .

O bloco de *Exchange*, ou no caso do *Luxy* denominado *LuxyCore*, é composto de várias etapas: análise das ordens e validação das assinaturas (OrderValidator), *matching* de itens (AssetMatcher) e processamento dos valores das transferências (LuxyTransferManager). Por apresentar essas diferentes etapas o bloco do *LuxyCore* é detalhado na Figura 3.2. Observa-se, nessa Figura, duas funções que permitem a comunicação dos participantes de uma venda com o *LuxyCore*: *matchOrders* e a *cancel*. A *matchOrders* é a função que utiliza a maior parte dos códigos implementados pelo protocolo, passando por todos os blocos do diagrama da Figura 3.2. Portanto, tal função é usada de base para o detalhamento apresentado na sessão 3.5, em que considera-se um *maker* criando uma ordem, depois um *taker* aceitando a ordem previamente criada e enviando a transação para a *blockchain*. A função *cancel* permite cancelar para sempre o *hash* e assinatura de uma ordem. Essa função é necessária para impedir que algum *taker* desatualizado

processe uma ordem criada por algum *maker* que já não tem mais interesse no negócio. O protocolo identifica que a ordem enviada está cancelada na etapa de validação do *Asset Matcher* (detalhado na seção 3.5).

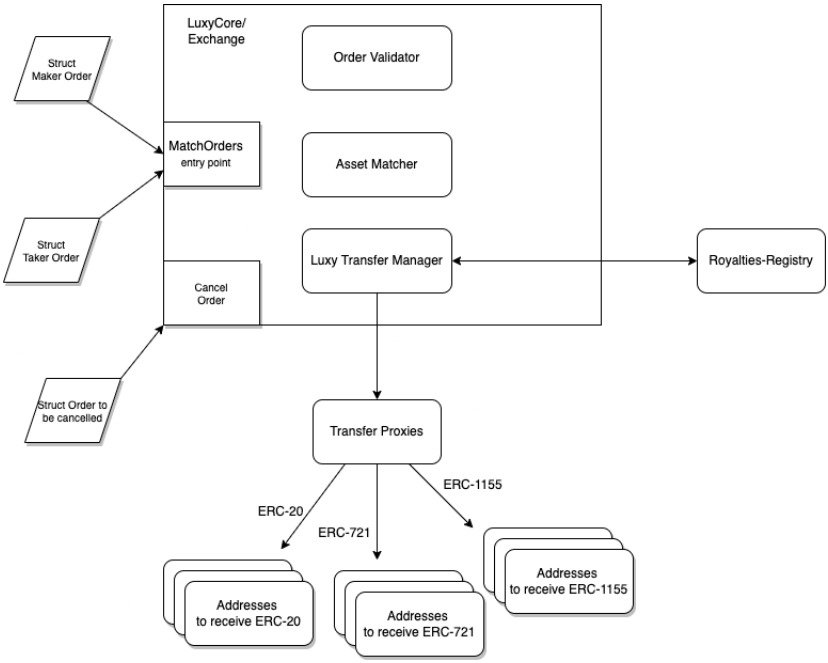


Figura 3.2: Detalhamento do bloco *Luxy Core*. Fonte: Autor

3.2 Tokens

Os *tokens* aceitos pelo *marketplace Luxy* são aqueles padronizados para *blockchains* que implementam *EVMs*, conforme ilustrado pelo diagrama da Figura 3.1. Nesta seção, pretende-se detalhar como os contratos de *proxy* foram implementados para o protocolo do *luxy*.

Além disso é apresentado de forma breve os dois modelos de *tokens* (*ERC-721* e *ERC-1155*) desenvolvidos. Esses *tokens* não fazem parte do protocolo, pois do ponto de vista do protocolo são como qualquer outro *token* externo sendo negociado. Para que os usuários possam criar seus próprios contratos de *tokens* com base nas implementações

do *Luxy*, foi criado um contrato de *Factory*. Os contratos de *factory* e *tokens* serão apresentados de maneira sucinta pois eles não tem relação direta com o desenvolvimento do protocolo.

3.2.1 Transfer Proxies

Os contratos de *proxy* do *Luxy* são divididos em dois contratos diferentes, o *ERC20TransferProxy* (responsável somente pelos ERC-20) e o *TransferProxy* (responsável pelos contratos de *ERC-721* e *ERC-1155*). Esses contratos poderiam ter sido feitos em um único contrato, mas foram separados por conta de eficiência, para tornar mais prático o processo de alterá-los caso surja um padrão melhor para lidar com a transferência de um dos três modelos de *tokens*(as atualizações para os padrões *ERC-721* e *ERC-1155* "andam mais juntas") e para redução dos gastos com gás.

Existem funções auxiliares implementadas pelos contratos do *LuxyCore* que permitem alterar os contratos de *proxy* utilizados pelo bloco de *Exchange* ou adicionar um contrato de *proxy* novo caso um novo padrão seja definido. Voltando ao diagrama 3.1, a conexão entre os blocos do *LuxyCore* e *TransferProxy* é uma simplificação que facilita o entendimento e traz a mesma funcionalidade do que ocorre de fato. A simplificação por um bloco de *Transfer Proxy* trata de múltiplos contratos de *TransferProxy* (para cada padrão de *token* aceito) que podem ser adicionados ou trocados e se conectam com o bloco do *LuxyCore*, através do endereço do contrato de *proxy*. Também é necessário adicionar o tipo do *token* novo na *LibAsset* para que o *LuxyCore* saiba identificar corretamente o tipo de *token* sendo comercializado e usar o *proxy* apropriado. Caso seja usado o *proxy* de um *token* errado a operação falha, como ocorre essa falha fica mais claro na seção 3.3 juntamente da explicação sobre os componentes da estrutura que compõe uma ordem, em especial sobre a *LibAsset* que armazena o endereço e tipo de padrão do *token* para o *marketplace* comercializar.

Os contratos de *proxy* basicamente implementam as funções de *safeTransferFrom* de cada uma das interfaces dos *tokens*(*ERC-1155*,*ERC-721* e *ERC-20*), expostas na Seção 2.4, e são baseados nos contratos dos *tokens* da biblioteca *OpenZeppelin* em uma versão mais recente que as utilizadas pelo protocolo da *Rarible*. O objetivo desse contrato é ser o operador dos *tokens* que o protocolo irá comercializar, conforme ilustrado na listagens das interfaces 2.1, 2.2, 2.4 nas funções de *transfers* e de *approve*. Dessa forma, caso o endereço do contrato de *proxy* tenha sido previamente aprovado a transferir os *tokens* de um determi-

nado usuário através da funções de *Approve*, o protocolo do *marketplace* consegue efetuar essas transferências de forma automática através dos contratos de *proxy*.

A estrutura proposta sozinha, não é considerado segura, pois um usuário pode ter aprovado o contrato de *proxy* a transferir seus fundos de forma acidental ou ele pode esquecer de remover a aprovação caso não se interesse mais no negócio. Devido a essa preocupação com segurança, os contratos da *Wyvern exchange* foram os primeiros a começar aplicar um modelo de assinatura digital *offChain* baseada na *EIP-712*, esse modelo implementa uma camada extra de segurança validando que o dono do *token* assinou uma mensagem permitindo que a transação ocorra. O modelo de assinatura digital *offchain* também tem vantagens de performance em um protocolo pois elimina a necessidade de armazenar a estrutura da ordem na *blockchain*, sendo necessário apenas armazenar sua *hash*. Também, foi adicionado um modelo que permite (e é recomendado ao usuário criador da ordem) definir um tempo de expiração para cada ordem. O protocolo do *Luxy* também aplica os mesmos padrões de segurança, no entanto com a versão mais atualizada da *EIP-712*. Essas validações serão melhor aprofundadas nas seções de Ordens e Validações.

Os contratos de *proxy* do *Luxy* cobrem todos os tipos de *tokens* existentes em *blockchains* compatíveis com *EVMs* com exceção de um, o *token* nativo da *blockchain*. Este *token* em especial não pode ser utilizado a partir de um *proxy*, pois ele é definido nativamente em cada *blockchain* e tem muitas considerações especiais, como ser a base para o pagamento de todas as transações aos mineradores ou validadores da rede (gás fees), conforme discutido na seção 2.2, e também ser o *token* de recompensa por validar ou minerar um bloco na *blockchain*. Essas definições vêm da implementação do *core* da *blockchain* e não é possível definir uma interface nos moldes de *ERC-20* para esses *tokens* que são fungíveis. Logo, não existe nenhum contrato de *proxy* que possa ser usado para transferir esses *tokens* em nome do usuário. No entanto existem funções nativas de todas as *blockchains* com compatibilidade para *EVMs* para transferir esses *tokens* desde que o endereço dono dos *tokens* nativos seja quem está chamando a transferência. Portanto no caso específico de um *taker* aceitando uma oferta que pede por *tokens* nativos, a operação é possível. No caso de *bids* o problema é contornado utilizando, no caso da *Polygon*, o contrato de *wrapper* de *Matic* (Seção 2.3) que é um contrato especial disponibilizado pela fundação *Polygon*. Esse contrato implementa toda a interface do *ERC-20* e, a partir de depósitos de *Matic*, cria *tokens* representativos de seu *token* nativo

que podem ser utilizados pelos contratos de *proxies* dos protocolos de *marketplace*.

3.2.2 Luxy721 e Luxy1155

Os contratos de *ERC-721* e *ERC-1155* do *Luxy* são de uso público. Eles existem para que usuários que não possuem sua própria coleção e não tem interesse em ter uma própria possam facilmente criar *NFTs* nessa coleção, que é aberta para qualquer *EOA* poder criá-los. Existe também a possibilidade de criar sua própria coleção nos formatos do *luxy*, utilizando contratos de *Factory*. Basicamente são contratos que permitem a criação de outros contratos, nesse caso das implementações da *ERC-721* e *ERC-1155* do *Luxy*. O funcionamento dos contratos de *Factory* não é relevante para este trabalho e portanto não foi detalhado mas sua definição segue a *EIP-2470*[124] e *EIP-1014*[125]. Neste caso basta saber que ele permite facilmente a criação de outras coleções nos formatos do *Luxy* para qualquer usuário do *marketplace* com a configuração de alguns parâmetros, como *supply* máximo, quem pode criar itens na coleção e etc.

Os contratos de coleção do *Luxy* são implementados utilizando as extensões de metadados e *enumerable*, comentadas na seção 2.4. Portanto funcionam de forma padrão, no entanto foram implementados algumas *features* extras:

- Apresentam inicializadores que permitem os usuários usar o modelo de metadados padrão ou fazer eles de forma que podem ser alterados depois, caso os metadados possam ser alterados posteriormente existem variáveis que indicam isso para os potenciais compradores também fiquem cientes disso. As coleções abertas de *ERC-721* e *ERC-1155* do *Luxy* não permite modificações posteriores dos metadados.

- Modelos de Royalties flexíveis que serão detalhados na seção 3.4, que são importante para o bloco *royalties-registry* representado na Figura 3.2.

- A função de *mint* (criar itens) pode ser pública ou privada para um endereço ou mais, definidos pelo criador da coleção.

3.2.3 Luxy ERC-20

Existe um *token* *ERC-20* para retratar o projeto *Luxy* tanto na *mainnet* quanto na *testnet* da *polygon*. Ele é tratado diferente dos outros *ERC-*

20 negociados pelo *marketplace* apresentando descontos nas taxas de negociação do protocolo para incentivar seu uso dentro do *marketplace*. É possível negociar o *token* do *Luxy* em algumas corretoras descentralizadas como a *SushiSwap*, mencionada na seção 2.4, com pares com *USDC* (uma *stablecoin* lastreada em dólar) e *Matic* a moeda nativa da *Polygon*.

3.3 Orders

Nessa seção é tratada primeiramente a estrutura de informações com a qual é montada uma ordem dentro do *marketplace Luxy* (mesma estrutura utilizada pelo protocolo da *Rarible*), depois trata-se dos mecanismos de assinatura digital implementados em acordo com a *EIP-712* (mesmos mecanismos inicialmente desenvolvidos pela *Wyvern Exchange*) e por fim dos mecanismos que validam essas informações de uma forma a garantir que todo usuário interagindo com o *marketplace* possa aproveitar ao máximo de um sistema de trocas *on chain*, sem necessitar de intermediários ou se preocupar com possibilidades de falhas no protocolo ou comportamento inesperado.

3.3.1 Estrutura das Ordens

Tanto as ordens do *maker* quanto as do *taker* seguem o mesmo padrão de formatação, com a diferença de que para o *maker* se requer assinatura sempre. Já o *taker* pode ter necessidade de assinar a transação caso um terceiro esteja chamando a função de *matchOrders* para efetuar a troca para as duas partes que estão fechando negócio. No entanto, na maioria dos casos a verificação de assinatura do *taker* não é necessária, pois o caso mais comum é o consumidor da ordem chamar o contrato diretamente, o que deixa fácil identificar que quem está enviando os dados da ordem de *taker* é quem esta chamando o contrato e possui os *tokens* necessários para a troca.

O diagrama da Figura 3.3 apresenta as estruturas (*structs*) que definem uma ordem. Elas apresentam sempre os campos de *maker* e *taker*, sendo efetuada uma validação de referência cruzada nas duas ordens no contrato de *AssetMatcher* (diagrama 3.2), indicando que o endereço de *maker* de uma é o *taker* da outra. Isso é feito pois cada parte está criando sua própria ordem e portanto é o *maker* da sua própria ordem. No entanto, a primeira ordem criada tem a caracterização como *maker* perante o *Asset Matcher* pois, como discutido no parágrafo anterior, a

parte criadora nunca inicializa a transação chamando o contrato, sempre necessita de uma assinatura válida e define o prazo de expiração da oferta.

O diagrama da Figura 3.3 mostra que é possível para o *taker* definir um prazo de expiração para sua oferta também, mas isso não faria muito sentido visto que a parte consumidora é quem deseja aceitar uma ordem que já existe.

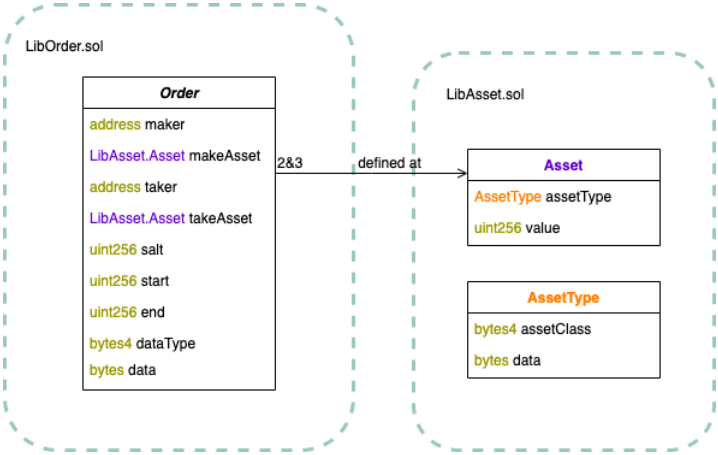


Figura 3.3: Diagrama do modelo de dados de uma Ordem. Fonte: Autor

Detalhando o diagrama da Figura 3.3, são apresentadas três *structs*:

- A *Order*, definida na biblioteca auxiliar do protocolo *LibOrder*;
- A *Asset*, que define o tipo e sua quantidade de *token* sendo oferecido e pedido em troca;
- A *AssetType*, que define os tipos de *tokens* que são aceitos pelo *marketplace* e o contrato aonde ele esta definido.

Tanto a *Asset*, quanto a *AssetType* são definidas na biblioteca auxiliar *LibAsset*.

Toda ordem montada deve ter um *maker* e um *taker*, no entanto em ordens criadoras normalmente o endereço do *taker* é colocado como zero de forma a permitir a qualquer participante da *blockchain* a aceitar a oferta. O campo do *taker* só é populado por um endereço específico em dois casos. Primeiro quando o criador da ordem de venda de um *NFT*

ou de uma oferta por um *NFT* (*bid*) quer efetuar uma venda casada, ou seja ele já sabe a *EOA* que almeja para o item que está vendendo ou querendo adquirir, de forma a bloquear outros participantes a comprarem ou aceitarem a oferta por aquele item. O segundo caso é quando já existe uma ordem previamente criada. Nesse caso, o endereço do *taker* é populado com o endereço do *maker* de uma ordem previamente assinada para a validação das ordens confirmar que de fato o *maker* da ordem já assinada corresponde ao *taker* da ordem que está disparando a transação na *blockchain*. Esse é o tipo de ordem considerada como caracterização de *taker* e na maioria dos casos não requer assinatura.

O formato descrito é utilizado pois poupa a criação de duas *structs* para o modelo de *Ordem* (uma exclusiva para o *maker* e outra para o *taker*), assim diminuindo o tamanho de espaço em *bytes* necessários que sejam alocados pela pilha da *EVM* para executar a função de transferência de *assets*. Por consequência, o custo para um cliente pagar em taxas de gás ao interagir com os contratos do protocolo é também reduzido.

Essas considerações de custos estão presentes em todo o desenvolvimento do protocolo. Outro exemplo é a estrutura de *Asset*, a qual possui alguns motivos para apresentar este formato. O primeiro é que muitos contratos *ERC-20* mais antigos não implementam o padrão *ERC-165* e portanto não tem exposição de seu *interfaceId* definido. Logo, a lógica para saber qual função dos *proxies* de transferência chamar e a quantidade a transferir parte da avaliação da *struct* de *Asset* e da *AssetType*. O segundo é que dessa forma também se padroniza a *struct* de ordem de uma forma que a mesma estrutura é possível de ser usada para efetuar transferências para qualquer *standard* de *token*.

Para que o *Luxy Transfer Manager* identifique qual *proxy* deve ser utilizado em uma transferência, é utilizada a *struct AssetType* e, conforme mencionado na Seção 3.2, é possível adicionar novos *proxies* de transferências para serem utilizados pelo *marketplace*. No entanto, além de adicionar um novo *TransferProxy*, deve-se também adicionar um novo *AssetType*.

A *struct AssetType* define informações vitais do contrato para o qual a ordem está sendo criada, de forma a orientar o protocolo como lidar com a ordem (caso essa informação seja colocada de forma equívoca, o contrato reverte e as transações nunca são realizadas). A variável *assetClass*(representada na Figura 3.3) é definida pelos quatro primeiros *bytes* da saída de uma função de *hashing keccak256* dos tipos de *tokens* aceitos pelo *marketplace* atualmente, os quais são:

- *ETH*;

- *ERC20*;
- *ERC721*;
- *ERC1155*.

O primeiro item da lista, *ETH*, é usado para representar qualquer *token* nativo de uma *blockchain* compatível com *EVM*. Como exemplo, os contratos para este projeto estão publicados na *Polygon*, sendo o *token* nativo o *Matic* que é o *token* nativo aceito neste caso (e único existente pois cada *blockchain* possui somente um *token* nativo, conforme explicado na seção 2.1). Os demais itens da lista, *ERC20*, *ERC721* e *ERC1155* representam os padrões de *tokens* existentes no ambiente de *EVMs*2 que são aceitos pelo *marketplace Luxy*.

A informação a ser passada no campo *data* varia de acordo com o modelo de *token* sendo usado na ordem, codificando sempre o endereço e quando existente(*ERC-721* e *ERC-1155*) o *tokenId* do item. Caso seja um *token* nativo é passado o parâmetro zero como endereço.

O campo de *value* tem o propósito de passar o montante de um determinado *token* a ser transacionado. No caso do *ERC-721*, deve ser sempre enviado o valor 1.

A *salt* é utilizada para determinar se a ordem é do tipo *maker* ou *taker*. Caso seja passado o parâmetro 0 para a *salt*, sabe-se que tratase de uma ordem *taker* e não há necessidade de conferir a assinatura desta ordem, mas é obrigatório que a pessoa chamando o contrato seja quem está enviando a ordem. No caso de ordens criadoras (*makers*), o propósito da *salt* é garantir que não seja possível enviar ordens repetidas, logo a *salt* cumpre a definição de sua definição:

> "Em criptografia, um salt é um dado aleatório que é usado como uma entrada adicional para uma função unidirecional que faz hash de dados, uma senha ou frase secreta."[126]

Ou seja, ela evita que duas ordens *makers* repetidas sejam enviadas para a *blockchain*. É responsabilidade do criador da ordem ou da aplicação que intermedia a criação de gerar *salts* diferentes e usar a mesma salt para gerar o *hash* de uma ordem que o criador deseje cancelar.

Os campos de *start* e *end* têm o propósito de dar prazo de expiração para algumas ofertas. O valor inteiro a ser passado é em *timestamp* em *UTC* e *UNIX TIME*.

Os campos de *dataType* e *data* servem para operações mais complexas, como quando o *maker* não deseja receber o pagamento diretamente

no endereço dele, mas quer distribuir o pagamento para algum outro endereço (no caso do pagamento ser pago para outra pessoa, a função de *transferPayout*, representada no diagrama da Figura 3.8, formata essas transações de acordo).

3.3.2 Assinatura Digital

A assinatura digital é feita conforme a *EIP-712*, o que significa que ela é executada de uma forma totalmente *off chain*. Isso quer dizer que não envolve escrita na *blockchain* e, portanto, ela tem a vantagem de não ter custo nenhum, mas a desvantagem de qualquer aplicação de *marketplace* ter que armazenar essa assinatura em um banco de dados centralizado e seguro. Esse processo, ilustrado pelo diagrama da Figura 3.4, é o ponto inicial para todos os processos de troca efetuados pelo protocolo do *marketplace*. Tal diagrama usa como auxiliar a lógica implementada da aplicação do *luxy*. O link para acessar e testar o fluxo em uma aplicação de produção é provido no final da seção de *Resultados*. A implementação dos controladores de *web3*, uso da *wallet web3*, servidor de *backend* e banco de dados são apresentadas superficialmente nas seções 2.4 e 2.5. Suas implementações são pontos auxiliares para a validação deste trabalho, no entanto fogem do escopo do trabalho. Por mais que a *aplicação luxy* tem um caráter complementar a todo protocolo desenvolvido aqui e será utilizada como exemplo principal para demonstrar as funcionalidades do protocolo, é possível interagir com o protocolo deste trabalho com *scripts* com interação direta com o contrato ou qualquer outro *front-end* visto que os contratos publicados neste trabalho estão públicos nas redes da *Polygon* e *Mumbai*.

Dito isso, o diagrama da Figura 3.4 ilustra o processo de gerar uma ordem com assinatura e salvar seus dados para uso posterior (no caso, conforme é feito pela aplicação *web Luxy*, que tem o mesmo nome que o protocolo). Este caso específico pode ser facilmente generalizado para se adequar ao uso de qualquer protocolo de *marketplace* com *orderbook offchain*.

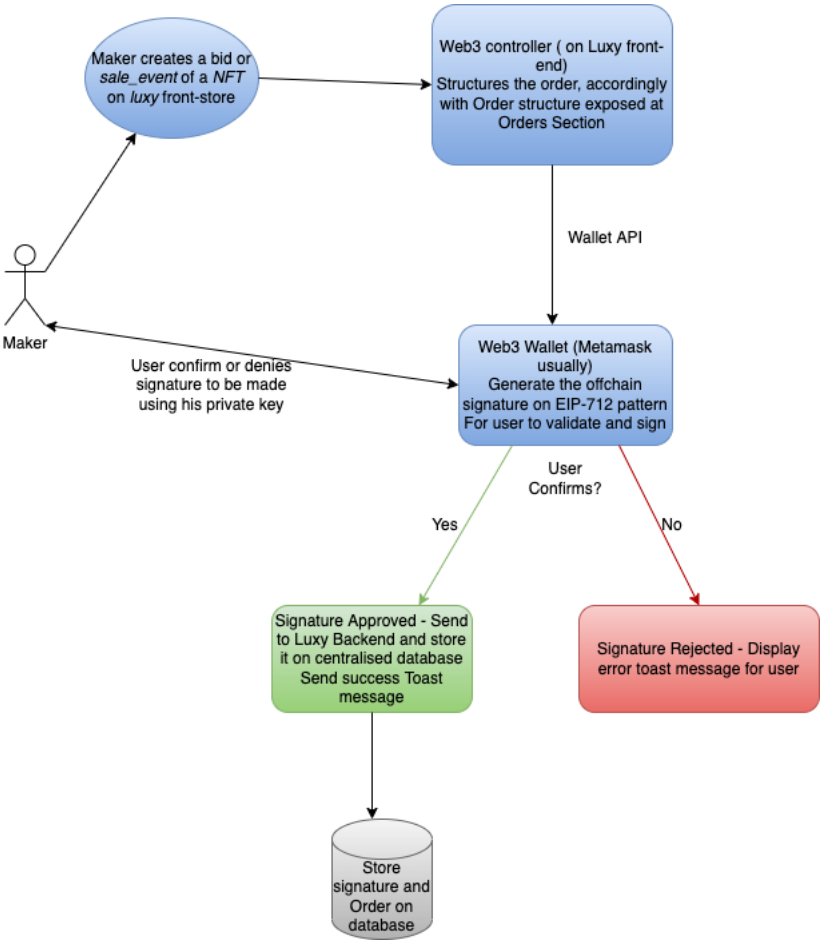
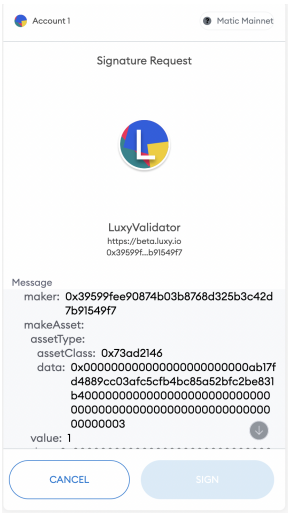
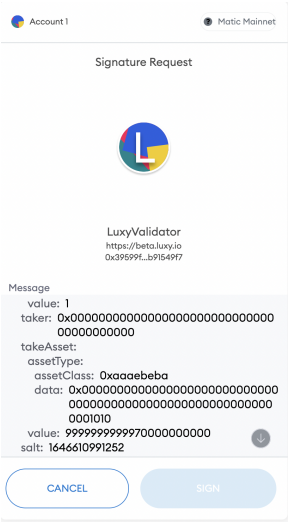


Figura 3.4: Diagrama de criação de uma *maker* order. Fonte: Autor

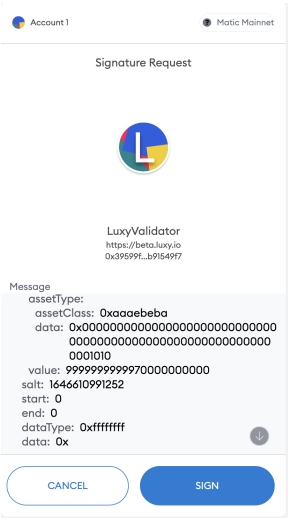
A Figura 3.5 mostra o modelo de assinatura *signedTypedData_v4* (veja a Seção 2.5), que é a ultima atualização da *EIP-712* e o mecanismo de assinatura digital atualmente implementado pelo protocolo *luxy*.



(a) Primeira Parte signedTypedData



(b) Segunda Parte signedTypedData



(c) Terceira Parte signedTypedData

Figura 3.5: Demo *signedTypedData* sendo usada em uma *wallet* web3. Fonte: Autor

As imagens apresentadas na Figura 3.5 ilustram um caso real da criação de ordem do tipo *maker* com a aplicação *luxy*. É um exemplo prático, que pode ser testado em tempo real diretamente no *website* da aplicação do *luxy*. Ela também valida que toda a estruturação de mensagem a ser assinada nos formatos *EIP-712* está funcionando adequadamente.

Podemos ainda extrair das imagens da Figura 3.5, pelas informações no topo de cada imagem, que o nome da mensagem a ser assinada é *LuxyValidator*, o *website* requisitante é do *beta* oficial da aplicação e o endereço do contrato corresponde ao o endereço do protocolo na *mainnet* da *Polygon*. Ainda, é possível ver todo o conteúdo da mensagens, cada uma das sub-Figuras da Figura 3.5 representa uma parte da mensagem a ser assinada para autorizar o protocolo do *markteplace* a transacionar um ativo para um usuário. Se comparado com o diagrama da estrutura das ordens exposto na Figura 3.3 a mensagem assinada apresenta os mesmos campos.

Uma constante crítica a essa versão da *EIP-712* é que ela ainda não tem uma padronização para reconhecer tipos (*int*, *string* e etc) e permitir que a mensagem seja mostrada com seu conteúdo ao invés de *hashes* em hexadecimal, como no caso atual da *signedTypedData_v4* exposta pela *API* da *Metamask*, que é o que esta sendo usado neste exemplo. Isso seria muito interessante para um público menos técnico conseguir entender melhor o que está assinando, pois a compreensão desses *hashes* exige um grau de conhecimento mais profundo da estrutura do protocolo do *luxy*, de funções nativas de alguma *API* de *web3* e da *EVM*, para serem convertidas de volta nos textos e valores que de fato o cliente tem intenção de assinar.

3.3.3 Validação das assinaturas e ordens

A implementação da validação das assinaturas usa *ECDSA* conforme os protocolos base para este projeto. A *ECDSA* é a função de assinatura exposta por qualquer implementação de uma *EVM*. As funções de validação de assinatura estão implementadas na LibSignature.sol e sua aplicação esta implementada no arquivo OrderValidator.sol.

A validação de ordens ocorre quando um *taker* chama a função de *matchOrders*, conforme ilustrado pelo diagrama da Figura 3.6. Esse diagrama ilustra a continuação assíncrona do diagrama da Figura 3.4. Nele podemos observar que o *taker* encontra algum item pelo qual se interessa no *marketplace* ou se interessa por uma *bid* feita em um de seus itens, que é mostrado de forma fácil na aplicação uma vez que

todas as ofertas e vendas criadas estão salvas no banco de dados da aplicação e são expostos visualmente para os usuários no *front-end*.

Então uma *order* é criada e dessa vez ao invés de necessitar de uma assinatura, o *taker* chama diretamente o contrato do protocolo do *luxy* e inicializa a transação na rede da *Polygon*, o que acontece a partir do momento que a função é transmitida para a *blockchain*, o que será explicado na seção *LuxyCore* juntamente do diagrama da Figura 3.8. No entanto é importante fazer uma ressalva aqui: o contrato também permite que o *taker* crie uma assinatura e a função de *matchOrder* seja chamada por um terceiro operador, que por consequência arcaria com os custos de chamar a operação na *blockchain*, o que é quase zero para o caso da interação deste protocolo na *blockchain* da *Polygon*. Alguns testes efetuados são mostrados na Seção 4.

Este caso foi implementado para eventos especiais nos quais algum agente decida pagar pela transação de algum usuário e não é implementado atualmente pela aplicação *luxy*, a qual usa o protocolo somente como uma camada de segurança para permitir interação de usuários sem confiança em uma autoridade central (os contratos são públicos, *openSource* e estão em uma rede decentralizada e que não pode ser desligada ou alterada conforme explicado na seção 2.1) e uns nos outros.

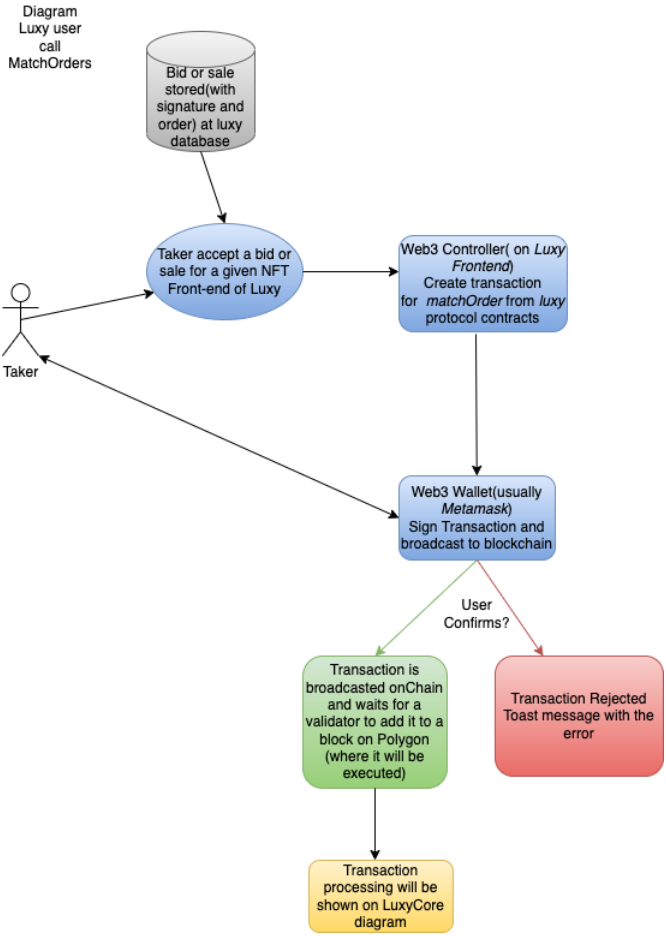


Figura 3.6: Diagrama do *taker* aceitando uma ordem previamente criada. Fonte: Autor

3.4 Royalties

Os *royalties* são uma parte fundamental na criação de *NFTs*. Eles são uma forma de garantir que o criador de um determinado *NFT* ou de uma coleção ganhe uma porcentagem sobre os direitos de revenda de seu conteúdo, como comentado na seção 2.4.

Tal como as interfaces de *royalties* de *marketplace*, os modelos de

royalties on chain ainda são variados e, por mais que recentemente tenha sido criado uma *EIP*[127] para padronizar uma interface para *royalties on chain*, ela ainda é limitada e não abrange eficientemente todos os aspectos necessários por demanda dos criadores para os *royalties*. No entanto, por ser o padrão normatizado da fundação *Ethereum*, esta EIP recente foi implementada pelo protocolo do *Luxy*.

Esta seção é dividida visando mostrar cada uma das três interfaces de *royalties* do *Luxy*. Posteriormente, é brevemente discorrido sobre o contrato de *Royalties-Registry* que interpreta os *royalties* nos contratos de *nft* sendo negociados, utilizando da *EIP-165* para validar se alguma das interfaces implementadas no *Royalties-Registry* é aceita; também é responsável por definir e alterar os *royalties* sobre as coleções.

3.4.1 Interface 1 e 2 - *RaribleV2 LuxyV1*

As interfaces do *RaribleV2* (única aceita pelo protocolo da *Rarible*) e *LuxyV1* serão explicitadas juntas pois a interface do *Luxy* é derivada da *RaribleV2*.

A estrutura de dados delas é a mesma, mudando principalmente a assinatura e portanto o *interfaceId* pela *EIP-165*. A assinatura foi alterada para permitir o uso desta interface nos contratos de coleções implementados pelo *Luxy*. Também algumas regras aplicadas pelo protocolo da *Rarible* foram alteradas.

A implementação dos *royalties* pelos contratos do *Luxy* não apresenta limite para o número de endereços, isto é, todos os contribuidores podem receber uma parcela dos *royalties* criados diretamente em sua carteira, permitindo royalties de até 68% sobre o item do *NFT*.

Basicamente, existe uma *struct* chamada *LibPart*, explicitada na Listagem 3.1. Tal *struct* define os royalties a serem pagos para um endereço tendo dois campos: o endereço e o valor, que é um percentual. Esses valores são definidos usando *Basis Point*[128], que é uma forma de definir um valor percentual com *unsigned integers*, pois *solidity* não aceita valores em ponto flutuante (2.5).

Essa *struct* é usada como *array* nas funções de *set royalties* permitindo o que é chamado de *royalties* colaborativos, ou seja, os *royalties* podem ser divididos em quantos endereços forem necessários para o *Luxy* e limitados a cinco na implementação da *Rarible*.

Deve-se lembrar que esses *royalties* são incorporados nos contratos dos próprios *NFTs* devendo ser definidos inicialmente na função de *mint* de um novo item. Portanto, são individuais e específicos. Também é possível aos endereços recebedores de *royalties* transferi-los para outra

carteira caso necessário.

```
1  //SPDX-License-Identifier: MIT
2  pragma solidity ^0.8.0;
3
4  library LibPart {
5      bytes32 public constant TYPE_HASH =
6          keccak256("Part(address account,uint96 value)");
7
8      struct Part {
9          address payable account;
10         uint96 value;
11     }
12 }
```

Listagem 3.1: LibPart

3.4.2 Interface 3 - *EIP-2981*

Este é o novo padrão para *royalties* definido. Ele apresenta algumas limitações em sua implementação, como o retorno da função de *royalties* sobre um item não apresentar o valor de *royalties* a ser pago em percentual (o que o protocolo do *Luxy* espera receber) e somente a quantia de *tokens* a ser paga em *royalties* no decorrer de uma venda. Também limitações que só permitem um endereço a receber os *royalties*. Da definição da EIP:

> “ Esse padrão permite que contratos, como *NFTs* que suportam interfaces *ERC-721* e *ERC-1155*, sinalizem um valor de royalties a ser pago ao criador ou detentor dos direitos da NFT toda vez que a NFT for vendida ou revendida. Isso é destinado a mercados NFT que desejam apoiar o financiamento contínuo de artistas e outros criadores de NFT. O pagamento de royalties deve ser voluntário, pois mecanismos de transferência como *transferFrom()* incluem transferências NFT entre carteiras, e executá-las nem sempre implica que uma venda ocorreu. *Marketplaces* e indivíduos implementam esse padrão recuperando as informações de pagamento de royalties com *royaltyInfo()*, que especifica quanto pagar para qual endereço por um determinado preço de venda. O mecanismo exato de pagamento e notificação ao destinatário será definido em futuros *EIPs*. Este *ERC* deve ser considerado um alicerce mínimo e eficiente em termos de *gas* para inovação adicional nos pagamentos de royalties de NFT.”[127]

No entanto ela tem a vantagem de estar começando a ser utilizada por diversos contratos novos e ter baixo custo em gás.

A interface do *Royalties-Registry* do *Luxy* busca por esse padrão e efetua a leitura de seus valores caso ele exista. Esta *EIP* ainda está em progresso e provavelmente será atualizada para permitir mais funcionalidades. Sua implementação base, referenciada na Listagem 3.2, mostra que a lógica para cálculo de porcentagem é deixada por conta do desenvolvedor, o que tem seus pontos positivos por não exigir que todo protocolo de *marketplace* passe a utilizar *Basis Point* como padrão para calcular porcentagens.

```
1   function royaltyInfo(
2         uint256 _tokenId,
3         uint256 _salePrice
4     ) external view returns (
5         address receiver,
6         uint256 royaltyAmount
7     );
8 }
```

Listagem 3.2: EIP2981

Isso também é facilmente contornado por *marketplaces* que utilizam o valor em percentual. Por sua lógica, basta enviar um *salePrice* como 10000 (equivalente a 100% em *Basis Point*) e utilizar o campo de *royaltyAmount* como o percentual a ser pago.

3.4.3 *Royalties Registry*

O contrato de registro dos *royalties* é chamado após as validações e antes das transferências começarem para o protocolo ter todos endereços aos quais deve enviar os fundos, seguindo o fluxograma indicado na Figura 3.7. Ele é estruturado verificando se existe algum *royalty* configurado sobre a coleção (o que é configurável pelo dono do contrato do *token* sendo comercializado no *marketplace* e exige interação direta com o contrato de *Royalties-Registry*). Caso haja definição de *royalty*, os endereços e os percentuais são adicionados também no mesmo *array* sem fazer distinção entre *royalties* sobre a coleção ou sobre o item. Posteriormente é feita uma busca se os *Tokens ERC-721* ou *ERC-1155* sendo negociados implementam alguma das três interfaces descritas acima. Caso existam implementações, os valores são lidos e salvos em um *array* com a lista de endereços a serem pagos e os percentuais das vendas aplicados.

No entanto existem duas limitações, conforme a Figura 3.7 indica, de 30% em *royalties* de coleção e 68% em *royalties* de *NFTs*. Caso

alguma das duas condições ultrapasse esses limites, a operação é rejeitada pelo protocolo. Também, para adicionar *royalties* sobre a coleção, o endereço do *owner* da coleção de *NFTs* ou do contrato de *royalties* deve adicionar os valores. Verificações na própria interface de *royalties* do *luxy* já existem. No entanto, o *Royalties-Registry* efetua essa redundância para o caso de alguma outra função utilizando a mesma assinatura da interface do *Luxy* mudar esses valores percentuais.

Para otimizar as operações e evitar que o contrato execute operações repetitivas de leitura nos contratos externos de *NFTs*, o contrato tem uma cache que armazena os valores lidos de *royalties*, usando esses valores em uma próxima interação. É responsabilidade do *owner* da coleção do *NFT* limpar a cache do contrato de registro de *royalties* caso deseje alterá-los em seus contratos. Essa implementação visa deixar o contrato mais rápido e menos custoso em itens muito comercializados no protocolo. Tal como os contratos de *Transfer Proxy*, o contrato de *Royalties-Registry* é usado como referência pelo contrato *LuxyCore*, mais especificamente no bloco *Luxy Transfer Manager*. Portanto, ele pode facilmente ser atualizado caso a *EIP-2981* evolua ou alguma outra regra precise ser aplicada.

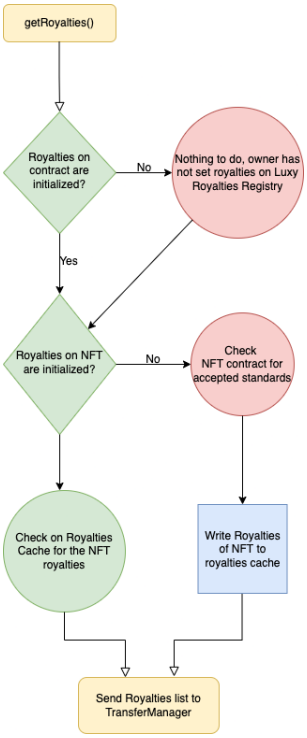


Figura 3.7: Fluxograma busca de Royalties. Fonte: Autor

3.5 *Luxy Core*

Esta é a seção central do desenvolvimento, onde o núcleo (*core*) do protocolo do *marketplace Luxy* é descrito, o qual corresponde ao contrato que expõe as funções de *cancel* e *matchOrders* no diagrama da Figura 3.2. Esse contrato se relaciona com todos os blocos, que no caso são outros contratos, representados no diagrama da Figura 3.2.

O fluxograma da Figura 3.8 ilustra em detalhes todos os processos do protocolo ocorrendo a partir do momento que um *taker* inicializa o processo *onChain* ilustrado no diagrama da Figura 3.6.

Como pode-se perceber, a cadeia lógica para efetuar a transferência, ilustrada no fluxograma 3.8 é bem extensa. A explanação sobre seus componentes centrais sera distribuída ao longo desta seção.

A Figura 3.8 começa do ponto em que a transação de troca entre os *assets* está na *memPool*. A seta verde indica o caso no qual se

tem gás suficiente adicionado e um validador (a *Polygon* usa *PoS*, 2.3) pega a transação para ser processada e o caso amarelo é o caso em que foi adicionado uma taxa muito baixa pelo *taker* e a transação fica pendente até que a rede tenha menos transações sendo processadas e algum validador julgue interessante processar a transação.

Neste fluxograma, os componentes em azul claro tem uma relação de herança com o contrato central *LuxyCore* da aplicação. Os componentes em prata e roxo são contratos periféricos utilizados pela aplicação, de forma a ser possível atualizar suas interfaces conforme modelos novos de *royalties* ou *transferProxy* ganhem dominância no mercado.

O componente em formato de nuvem remete à interação com contratos externos de *tokens* desenvolvido por outras pessoas. Os componentes em vermelho remetem a rejeições na execução do protocolo, disparadas por conta de algum dos contratos (caso a operação reverta, toda execução é desfeita e a transação não é adicionada na *blockchain* e por isso podemos transferir um dos *assets* antes do outro sem nos preocuparmos em validar se o usuário de fato possui o *token* ou a quantidade indicada dele).

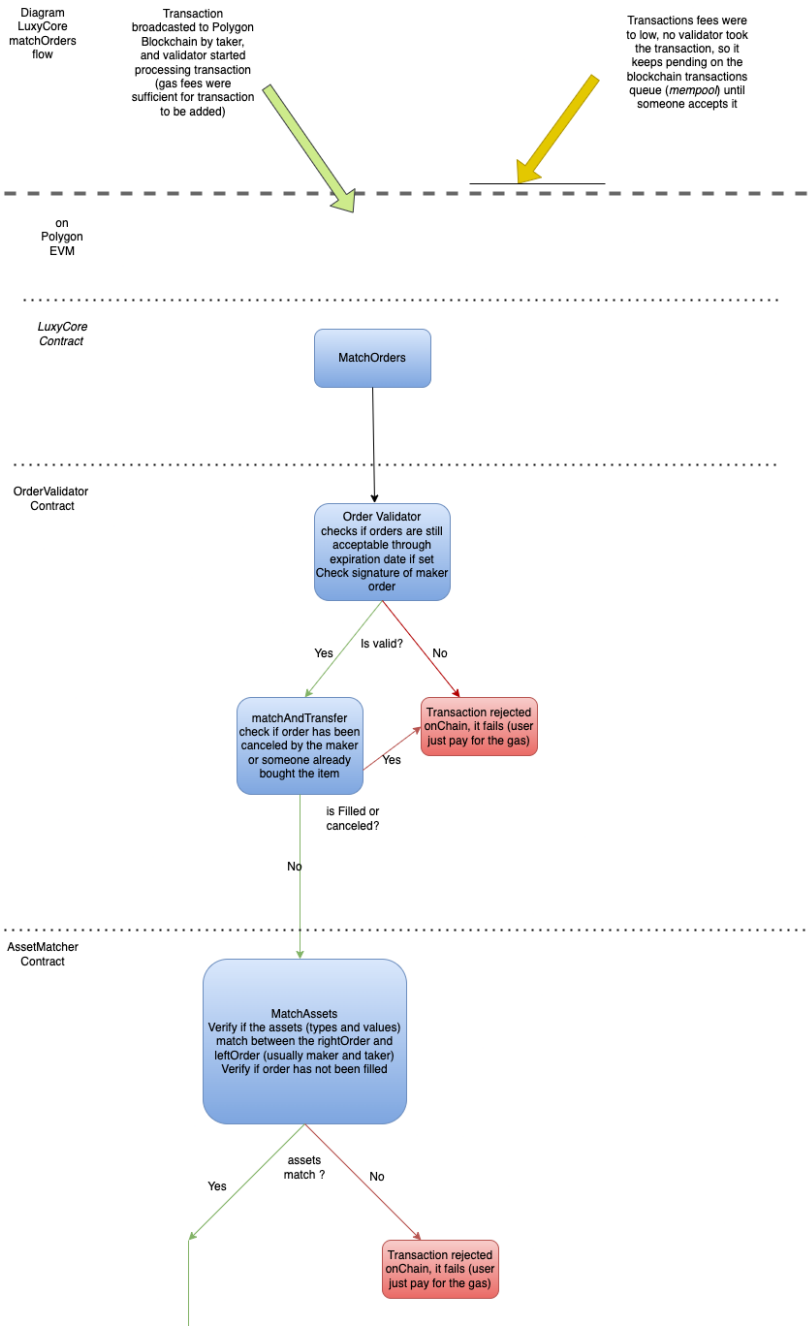


Figura 3.8: Diagrama do core do protocolo 1. Fonte: Autor

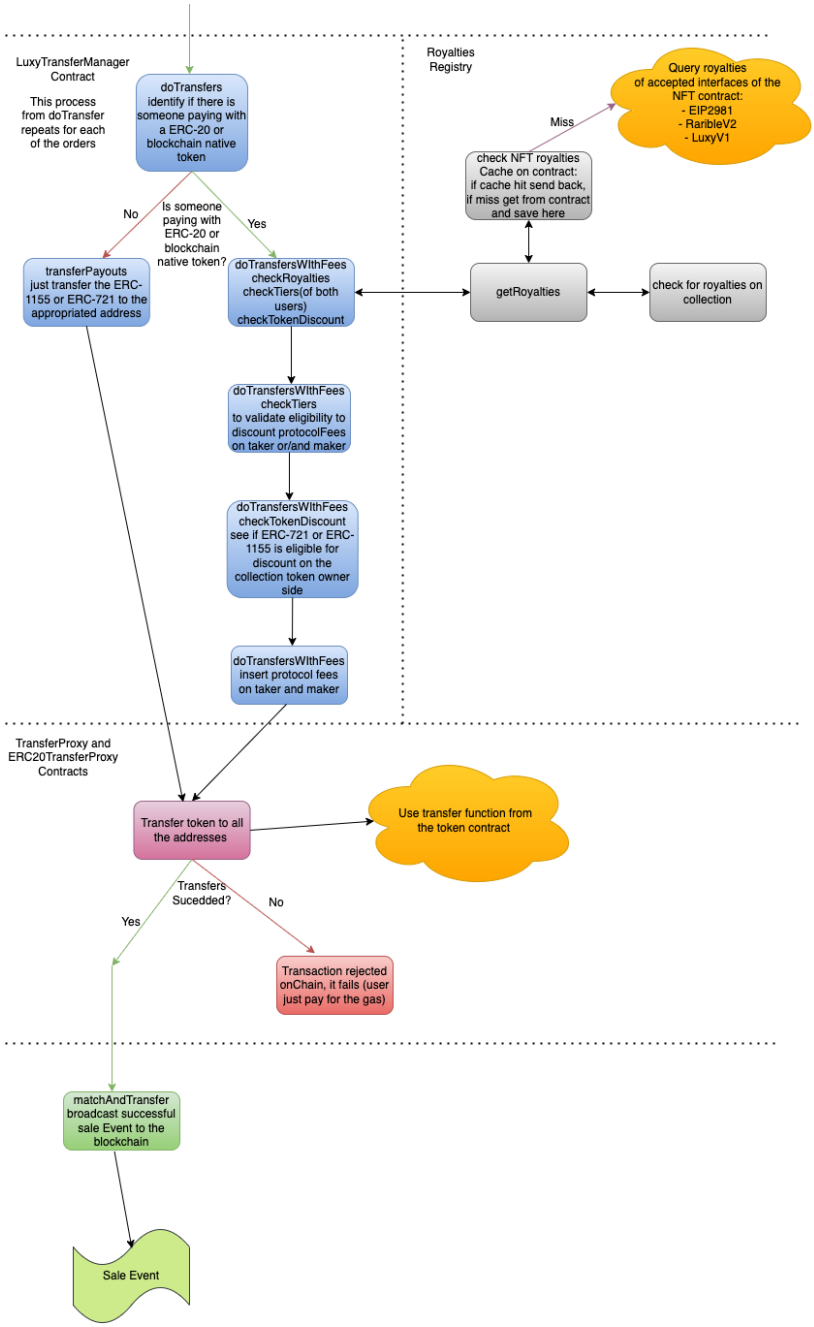


Figura 3.9: Diagrama do core do protocolo 2. Fonte: Autor

O último componente em verde claro é o evento de venda disparado ao final da execução para a informação ser indexada pela aplicação do *luxy* e o bancos de dados centralizado atualizado.

O objetivo desta parte é esclarecer a parte central do protocolo, que é a função que o *taker* executa no diagrama 3.6. Para inicializar a transação usa-se a *matchOrders*, ilustrada na Figura 3.10, que é uma função de escrita na *blockchain* e conta com 5 argumentos de entrada.

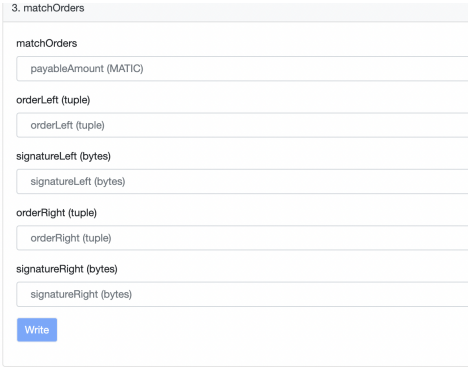


Figura 3.10: Chamada do *matchOrders* no explorer da *Polygon*. Fonte: Autor

Os argumentos de entrada da *matchOrders*, Figura 3.10, são as ordens, assinaturas (ilustradas nos diagramas 3.4, 3.6) e *payableAmount*, o qual só é utilizado em transações que envolvem *tokens* nativos pela peculiaridade desse *token* conforme explicado na seção 3.3.

3.5.1 *Asset Matcher*

Começando pelo *AssetMatcher*, podemos reparar pelo diagrama 3.8 que ele ocorre após as validações centrais da ordem serem efetuadas, ou seja, já é sabido que as assinaturas estão corretas, a ordem não esta expirada e, caso sejam ordens com *taker* ou *maker* casados, os endereços públicos das *EOA* batem.

O *assetMatcher* então conclui as próximas etapas de verificação, isto é, se as ordens já não foram executadas ou canceladas pelo *maker*, se os *assets* são tipos de *tokens* aceitos pelo protocolo (como falado na seção 3.3. novos assets podem ser adicionados com seus respectivos *proxies* de transferência) e também se os *assets* "batem", ou seja, as

ordens da esquerda e da direita querem trocar os *assets* e quantidades que cada lado possui.

Aqui é um bom ponto de ressaltar que o protocolo do *luxy*, como dos *marketplaces* de base citados na fundamentação teórica, permite a troca de quaisquer tipos de *tokens* desde que eles sejam aceitos pelo protocolo. No entanto este *marketplace* é considerado exclusivo para *NFTs*, pois como citado na seção 2.4, existem jeitos muito mais eficientes para a troca entre *tokens ERC-20*. As corretoras decentralizadas são um bom exemplo.

3.5.2 *Luxy Transfer Manager*

Caso não ocorram reversões nas funções do contrato de *AssetMatcher*, o próximo passo é dado pelo gerente de transferências. O *Luxy Transfer Manager*, conforme ilustrado na Figura 2.13 implementa toda lógica de transferência, desde a avaliação do montante total a ser transferido, descontando os royalties desse montante e as taxas do protocolo, que são definidas em uma função auxiliar implementada pelo gerente de transferência e exportada pelo contrato do *LuxyCore* que herda suas propriedades (a titulo de curiosidade a taxa base do *marketplace* atualmente é de 2%).

O diagrama da Figura 2.13 mostra que, de acordo com o tipo de *token*, temos duas funções diferentes que são chamadas a *transferPayouts* e *doTransfersWithFees*, e também que todo esse processo é efetuado em cada uma das ordens.

Caso o *token* seja um *ERC-721* ou um *ERC-1155*, não existem taxas do protocolo ou *royalties* a serem adicionados, pois a troca é entre dois itens colecionáveis ou de propriedade. Mas no caso mais comum de todo *marketplace*, o interesse está em vender um item ou objeto por um *token* fungível com representação de valor claro. No caso de um *token* fungível sendo negociado, verifica-se se quem paga a *fee* é o *maker* ou o *taker*, de acordo com uma hierarquia de prioridade. Existem três tipos de cenários possíveis para a execução das funções:

- FeeSide = None. O protocolo não recebe taxas e é chamado a função de *transferPayouts* duas vezes.

- FeeSide = *Maker*. O *maker* passa pela função *doTransferWithFees* e o *taker* pela função *transferPayouts*.

- FeeSide = *Taker*. O *taker* passa pela função *doTransferWithFees* e o *maker* pela função *transferPayouts*

A sequência lógica de execução para *transferPayout* ou *doTransferWithFees* é ilustrada no diagrama da Figura 3.8.

Essa etapa finaliza somente com a chamada do *transferProxy* em cada uma de suas execuções, existe uma última chance de reversão ali caso os donos do *assets* tenham removido o *approve* do contrato de *transferProxy* para operar aqueles itens ou não os possuam mais.

No caso do *token* nativo da *blockchain*, existe uma função nativa da *EVM* chamada que pode ser visualizada como uma função de *proxy* de transferência também.

Por fim, encerra-se o protocolo emitindo o evento de venda para toda *blockchain*. Esse evento de venda é emitido assim que a primeira confirmação (o bloco no qual a transação é iniciada é adicionado na *blockchain*) é obtida. Todos os participantes da rede inscritos no evento emitido pelo contrato através da conexão com *Websocket* de *web3* provida por algum *RPC* operacional na *blockchain* na qual o contrato foi publicado recebem a informação e conseguem atualizar suas aplicações ou serviços. No caso deste trabalho, é a rede da *Polygon* que devido ao seu *blocktime* baixo e consenso de *PoS*, tem uma confirmação que ocorre em cerca de 1 a 5 segundos após a transação ser enviada na *blockchain*.

CAPÍTULO 4

Resultados

Os resultados foram obtidos de forma final através dos contratos referentes a este protocolo publicados na *mainnet* da *Polygon* e implementados em uma aplicação de *front-end*, conforme sera discutido na seção 4.3. No entanto ao longo do desenvolvimento os resultados foram obtidos através de testes com o objetivo primário de validar que a lógica de negócio foi implementada corretamente e secundário de otimizar os consumos de gás as operações realizadas pelo contrato. Pode-se verificar esses testes no *Github* na pasta de *test*. Na figura 4.2, observa-se o resultado esperado após se rodar todos os *scripts* de testes. Para executar os testes, basta executar os comandos das Listagens 4.1 e 4.2.

Listagem 4.1: Comandos testes 1

```
npx hardhat node
```

Listagem 4.2: Comandos testes 2

```
npx hardhat test
```

4.1 Validação dos Contratos

Para testar todo o protocolo, foi necessária a implementação de algumas aplicações simples para validar a *EIP-712*, ilustrada na Figura 4.1, com uma carteira *web3* para confirmar que a decodificação da assinatura resultava no mesmo endereço púbico que o populado pelo campo de *maker* da *struct* de ordens.

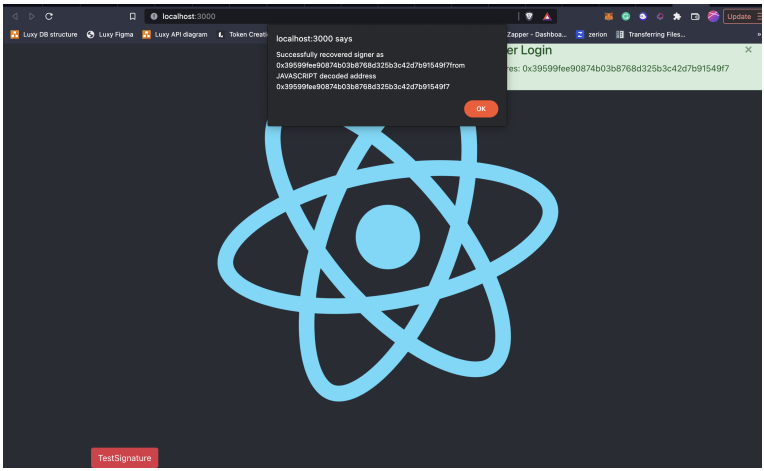


Figura 4.1: Verificação inicial da implementação da *EIP-712*. Fonte: Autor

O caso geral de testes implementados é explicado em 4.1.1 e também a Figura 4.2 apresenta o resultado esperado ao se rodar todos os testes. O foco nos próximos parágrafos será nos testes finais que validam principalmente a lógica de cancelamento e da função *matchOrders* que como mencionado ao longo do capítulo 3, utilizam quase toda lógica de negócios do protocolo.

Os teste foram todos executados com a rede local disponibilizada pelo *Hardhat*, a qual emula a *blockchain* do *Ethereum* e por tabela todas as *blockchains* compatíveis com *EVMs*. É possível escolher a versão da *EVM* utilizada (no caso foi a última que é a que a *polygon* utiliza, sendo uma *sidechain* eles sempre atualizam sua versão da *EVM* junto do *Ethereum*), o preço da gás a ser considerado, entre outros parâmetros através de um arquivo de configurações (similar a um *makefile* só que em *JSON*).

Toda vez que se inicializa um servidor *JSON-RPC* com *Hardhat*

automaticamente são geradas dez contas de teste com dez mil *ethers* cada. O comando da Listagem 4.1 demonstra como inicializar um servidor *JSON-RPC* em cima da rede do *Hardhat*, o que expõe a *API* e *websocket* de *web3*, entre outras interfaces padrão da implementação da *EVM*, para uso do desenvolvedor. Através da comunicação com esse servidor é que os testes são executados.

Após a validação da *EIP-712*, todas as funções escritas na aplicação mostrada na Figura 4.1, foram adaptados para *scripts*, disponíveis no *Github* na pasta de testes. Esses *scripts* são:

1. *assets.js*: Simula a conversão para *bytes4* da *assetClass*, explicada na seção 3.3, e da codificação do campo de data que contém o endereço e *tokenId* do contrato, somente endereço caso seja *ERC-20*.

2. *Order.js*: Simula a criação das ordens utilizadas pelo protocolo desenvolvido como um *array* de *JSON* da forma que o protocolo espera recebe-las e também efetua a assinatura das ordens.

3. *EIP712.js*: Simula gerar assinaturas nos padrões da *EIP-712*. Esses *scripts* são chamado pelo *script* de ordens.

4. *sign.js*: Simula utilizar os mecanismos de *ECDSA* para testar como funcionam assinaturas dentro de uma *EVM*, de uma forma mais básica que a *EIP-712* mas utilizando todos os mesmo conceitos de base.

Para os testes do arquivo *ExchangeLuxy.test.js*, basta os três primeiros *scripts* mencionados na enumeração 4.1. Esse arquivo apresenta diversos testes para casos diferentes de ordens interagindo com o *matchOrders*. São verificados detalhes como:

- Todos os cenários em que a transação deve falhar, se a reversão esta acontecendo de fato e pelo motivo esperado (ex: uma ordem é cancelada e alguém tenta usa-la);

- Comercialização de todas as combinações possíveis entre os três tipos de *tokens* aceitos;

- Verificar se as taxas do protocolo estão sendo deduzidas de forma apropriada;

- Verificar se os descontos nas taxas do protocolo estão funcionando como deveriam;

- Se a interface de royalties está funcionando apropriadamente, se os *royalties* são pagos aos endereços devidos e nas quantidades devidas ou se todas as interfaces estão sendo reconhecidas;
- Verificar se o sistema de *tiers* para *holders* do *token* fungível do *Luxy* esta funcionando corretamente;
- Verificar se modificações de administrador nas taxas do protocolo ou algum outro parâmetro de controle são propagas corretamente.

Esses são alguns dos testes principais feitos nesse arquivo de teste que resume os pontos centrais da operação do protocolo.

4.1.1 Testes de lógica

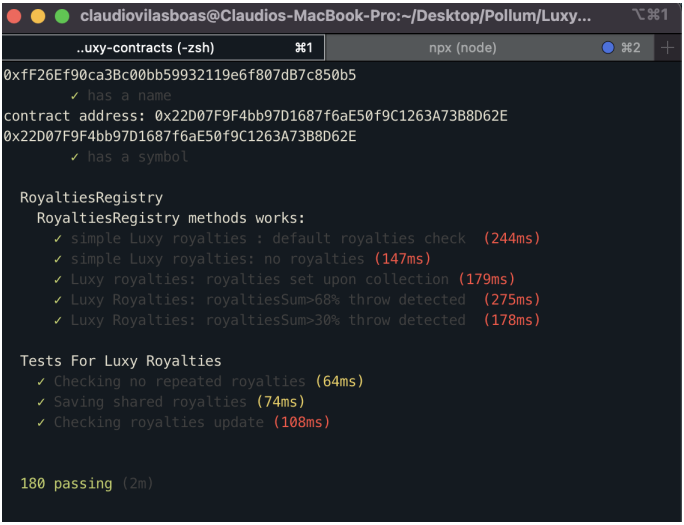


Figura 4.2: Chai e Hardhat testes no protocolo de *marketplace*. Fonte: Autor

A Figura 4.2 mostra que os 180 testes implementados passaram. A Figura também ilustra os últimos métodos a serem testados os do *Royalties-Registry* e dos *Royalties* do *Luxy* (somente por que os testes são executados em ordem alfabética, os últimos a serem escritos foram os da *ExchangeLuxy*). Cada item em verde indica um teste sendo efetuado. No caso dos *royalties*, são testados diversos itens explicados na seção de 3.4, como se o contrato de *RoyaltiesRegistry* reverte e emite a

mensagem apropriada caso os *royalties* sobre a coleção excedam 30%, valor máximo definido para eles, ou os *royalties* sobre o *NFT* excedam 68%.

Também são identificados corretamente os *royalties* de contratos de *tokens ERC-721* ou *ERC-1155* implementando as interfaces de *royalties* aceitas pelo *marketplace*. No caso das interfaces de *royalties* do *luxy*, ilustradas na Figura 4.2, identifica-se se *royalties* diferentes para endereços repetidos são bloqueados ou os *royalties* são salvos propriamente na memória do contrato, ou se o endereço de algum beneficiário foi alterado.

Apesar de ter sido dado um peso maior aos testes da *ExchangeLuxy*, todos os testes de componentes modulares são muito importantes. Por exemplo, o teste de validação das assinaturas (um dos primeiros a ser escritos e sem os quais o trabalho definitivamente não teria avançado) ou de adição de um tipo de *token* novo e seu respectivo *TransferProxy*, o que com certeza vai ser necessário ao longo dos próximos anos. Também os testes de todas as funções restritas, por exemplo indicar que o endereço para qual são encaminhadas as taxas do *marketplace* só possam ser chamadas pelo endereço do *owner* dos contratos. Uma vez que as consequências de qualquer abertura para *hack* de um protocolo desses pode ser motivo de encerramento do projeto e especialmente por este motivo, conforme mencionado na Seção 3.1, mesmo com todos os testes implementados foi efetuado uma auditoria de segurança sobre todo o código desenvolvido neste trabalho para corrigir qualquer possível falha.

Diversos testes foram efetuados, algumas melhorias foram implementadas. O documento que descreve os testes e resultados é publico e está disponível no repositório do projeto no *GitHub*. Houveram algumas rodadas de auditorias e correções, mas cotando a própria empresa de segurança, a conclusão final sobre as auditorias efetuadas sobre os contratos desenvolvidos neste trabalho foi:

> "A Cyrex determinou que a maturidade geral de segurança dos contratos inteligentes é grande e atenderá ao apetite de risco de qualquer usuário final. Nenhuma vulnerabilidade foi descoberta durante a auditoria. Várias recomendações foram implementadas de forma correta, mas mais importante, os contratos inteligentes foram testados e validados minuciosamente pelos especialistas em segurança de contratos inteligentes da Cyrex."

4.2 Performance

Os testes de gás foram efetuados em três diferentes redes (*Ethereum* com consenso *PoW*, *Polygon* com consenso *PoS* e a *Avalanche*, escolhida de forma aleatória e também por apresentar um algoritmo de consenso diferente dos abordados por este trabalho. Esses testes foram efetuados aproveitando os 180 testes já escritos para validar a lógica do contrato, utilizando um pacote chamado *hardhat-gas-reporter* que efetua essas medições para contabilizar o custo das funções implementadas por um contrato.

Considerando o objetivo primário, que é publicar os contratos na *Polygon*, o consumo em gás das chamadas do contrato é considerado adequado. Pode-se adicionalmente verificar pelo *gas tracker* do *explorer* da *Polygon* os custos das funções do protocolo.

As tabelas nas Figuras 4.3, 4.4 e 4.5 apresentam, na primeira coluna a versão do compilador *solc* utilizada. Tal coluna é subdividida entre os contratos e métodos utilizados nos testes.

Na segunda coluna observasse que primeiro que os otimizadores foram utilizados, o preço de gás em *GWEI* (obtido em tempo real direto da *blockchain alvo* por uma *API*) e em sequência os mínimos, máximos gastos em gás.

Na terceira coluna tem-se o número de iterações da simulação, no caso desses testes foram 1000 vezes, depois se tem as médias de gasto em gás no campo *AVG*.

Na quarta coluna se tem o limite de gás possível por bloco da *blockchain* alvo. Na sequência, observa-se em vermelho o preço da cotação do *token* nativo da blockchain de teste em relação ao dólar do momento de execução dos testes. Finalmente, o número de chamadas feitas por cada iteração dos testes aos métodos e o preço médio em dólar pelo custo de chamada a cada método dos contratos.

Deve-se lembrar que estes testes são simulações e podem apresentar pequenas divergências aos custos atuais na rede alvo e também que o *gasPrice* altera de acordo com o uso da rede.

Avaliando a tabela da performance do *Luxy* na rede do *Ethereum* (Figura 4.3) observa-se que o preço médio de custo de gás para a chamada da função do *matchOrders* foi de 18 dólares. Para comparar com o custo de gás do *OpenSea* (líder de mercado) foi realizada uma média do preço de execução das 20 ultimas transações da função *AtomicMatch* (equivalente ao *matchOrders*) do contrato *Wyvern Exchange*. O custo médio da função *AtomicMatch* foi de 22 dólares para as mesmas condições (*gasPrice* e preço do *Ethereum* em relação ao dólar) realiza-

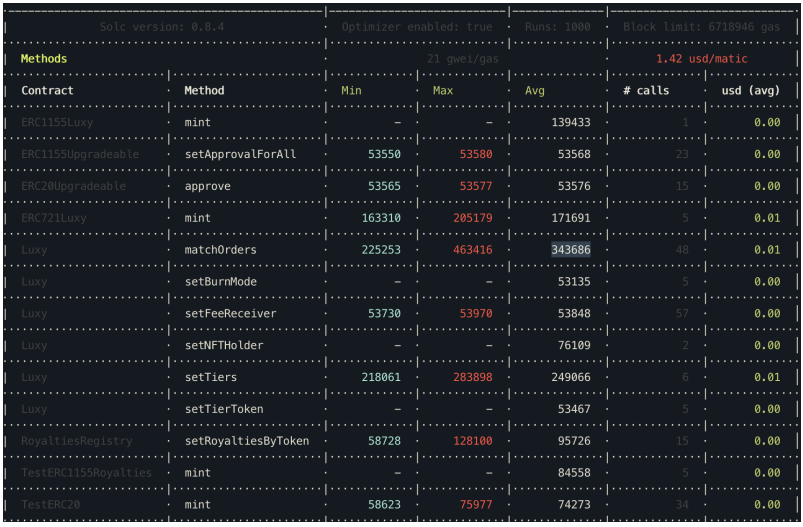
das no teste com o *Luxy*. Observa-se que a função de *matchOrder* tem uma implementação mais otimizada que a do protocolo do *OpenSea*.

Outra informação relevante extraída dos testes de performance é que de fato *EVMs* com algoritmo de consenso *PoS* apresentam preços consideravelmente abaixo aos seus predecessores com *PoW*. Tal informação pode ser observada ao se comparar os preços médios da tabela da Figura 4.3, representando a *blockchain* do *Ethereum* que apresenta consenso *PoW*, e da tabela da Figura 4.4 que mostra os preços médios para a *blockchain* da *Polygon* com algoritmo de consenso *PoS*.

A comparação de algoritmo de consenso com a *Avalanche* foge do escopo deste trabalho, uma vez que esta rede usa um algoritmo de consenso totalmente diferente dos abordados ao longo deste trabalho. Seu consumo só foi apresentado (Figura 4.5) para expor a existência de novos algoritmos de consenso que estão sendo desenvolvidos.

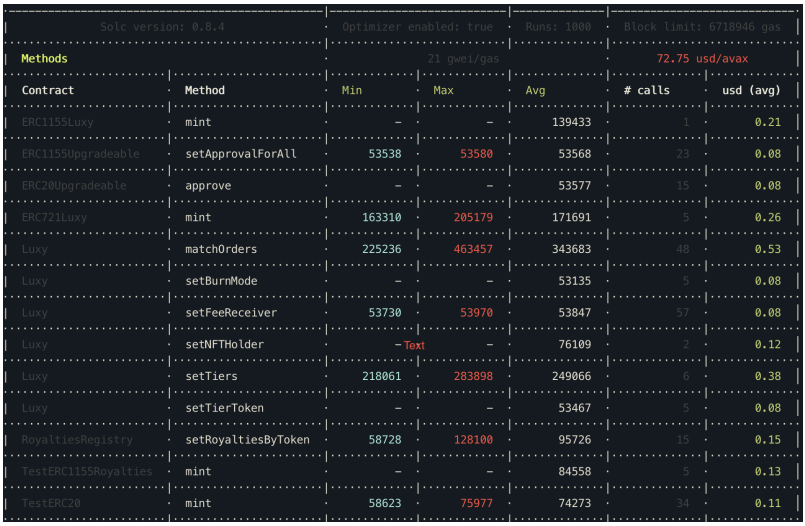
Solc version: 0.8.4	Optimizer enabled: true	Runs: 1000	Block limit: 6718946 gas	
Methods	21 gwei/gas		2495.90 usd/eth	
Contract	**Method**	**Min**	**Max**	**Avg**	**# calls**	**usd (avg)**
ERC1155Luxy	mint	–	–	139433	1	7.31
ERC1155Upgradeable	setApprovalForAll	53546	53580	53568	23	2.81
ERC20Upgradeable	approve	53565	53577	53576	15	2.81
ERC721Luxy	mint	163310	205179	171691	5	9.00
Luxy	matchOrders	225224	463457	343687	48	18.01
Luxy	setBurnMode	–	–	53135	5	2.79
Luxy	setFeeReceiver	53730	53970	53847	57	2.82
Luxy	setNFTHolder	–	–	76109	2	3.99
Luxy	setTiers	218061	283898	249066	6	13.05
Luxy	setTierToken	53455	53467	53465	5	2.80
RoyaltiesRegistry	setRoyaltiesByToken	58728	128100	95726	15	5.02
TestERC1155Royalties	mint	–	–	84558	5	4.43
TestERC20	mint	58623	75977	74273	34	3.89

Figura 4.3: Consumo médio em taxas da *blockchain* do *luxy* no *Ethereum*. Fonte: Autor



Solc version: 0.8.4	Optimizer enabled: true	Runs: 1000	Block limit: 6718946 gas	
Methods	21 gwei/gas		1.42 usd/matic	
Contract	**Method**	**Min**	**Max**	**Avg**	**# calls**	**usd (avg)**
ERC1155Luxy	mint	–	–	139433	1	0.00
ERC1155Upgradeable	setApprovalForAll	53550	53580	53568	23	0.00
ERC20Upgradeable	approve	53565	53577	53576	15	0.00
ERC721Luxy	mint	163310	205179	171691	5	0.01
Luxy	matchOrders	225253	463416	343686	48	0.01
Luxy	setBurnMode	–	–	53135	5	0.00
Luxy	setFeeReceiver	53730	53970	53848	57	0.00
Luxy	setNFTHolder	–	–	76109	2	0.00
Luxy	setTiers	218061	283898	249066	6	0.01
Luxy	setTierToken	–	–	53467	5	0.00
RoyaltiesRegistry	setRoyaltiesByToken	58728	128100	95726	15	0.00
TestERC1155Royalties	mint	–	–	84558	5	0.00
TestERC20	mint	58623	75977	74273	34	0.00

Figura 4.4: Consumo médio em taxas da *blockchain* do *luxy* na *Polygon*. Fonte: Autor



Solc version: 0.8.4	Optimizer enabled: true	Runs: 1000	Block limit: 6718946 gas
Methods	21 gwei/gas		72.75 usd/avax

Contract	Method	Min	Max	Avg	# calls	usd (avg)
ERC1155Luxy	mint	–	–	139433	1	0.21
ERC1155Upgradeable	setApprovalForAll	53538	53580	53568	23	0.08
ERC20Upgradeable	approve	–	–	53577	15	0.08
ERC721Luxy	mint	163310	205179	171691	5	0.26
Luxy	matchOrders	225236	463457	343683	48	0.53
Luxy	setBurnMode	–	–	53135	5	0.08
Luxy	setFeeReceiver	53730	53970	53847	57	0.08
Luxy	setNFTHolder	–	Text	76109	2	0.12
Luxy	setTiers	218061	283898	249066	6	0.38
Luxy	setTierToken	–	–	53467	5	0.08
RoyaltiesRegistry	setRoyaltiesByToken	58728	128100	95726	15	0.15
TestERC1155Royalties	mint	–	–	84558	5	0.13
TestERC20	mint	58623	75977	74273	34	0.11

Figura 4.5: Consumo em taxas da *blockchain* do *luxy* na *Avalanche*. Fonte: Autor

4.3 Aplicação

A plataforma que utiliza os contratos do protocolo do *marketplace luxy* ainda(Abril de 2022) em desenvolvimento. No entanto, é possível acessar a versão beta da aplicação. Para adquirir acesso e poder testar praticamente todos tópicos abordados neste trabalho basta mandar um *email* para: claudio@luxy.io .

Todas as funcionalidades do protocolo discorridas neste trabalho já estão implementadas, testadas, auditadas e possuem integração com o *front-end* da aplicação.

A implementação do protocolo e também da integração do *front-end* foram enormes desafios ao longo do desenvolvimento da plataforma. No entanto, para uma aplicação de *marketplace* bem sucedida, existem diversos outros pontos que estão em desenvolvimento como um serviço de indexação de *blockchain* eficiente, otimizações das *queries* de um banco de dados centralizado, escalabilidade à infraestrutura para comportar usuários, manutenção dos servidores, desenvolvimento da *UI* e etc.

Os resultados numéricos de pessoas utilizando o protocolo *Luxy* ainda são muito baixos, visto que a aplicação está em fase beta e fechada à comunidade em geral (somente foram liberado o acesso a algu-

mas poucas centenas de pessoas para testarem o produto). A aplicação do *marketplace* será lançada somente no final do mês de abril de 2022. No entanto todas as rodadas de negociação do *token* que representa a plataforma foram muito bem vistas por investidores e pela comunidade de *NFTs* da *Polygon*, conforme representado pela tabela da Figura 4.6.

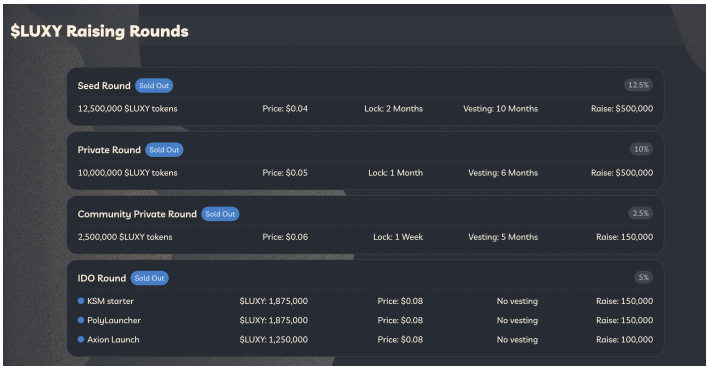


Figura 4.6: Rodadas de negociação do *token* representante do projeto. Fonte: Autor

Por fim o autor deixa aqui o convite para quem interessar possa de conferir o *roadmap* da aplicação em *luxy.io* ou acessar as partes de finanças do projeto que envolvem o *token ERC-20* desenvolvido para representar o projeto, também de se aventurar nas documentações documentações do projeto e entender melhor as propriedades do *token* e utilidade desse *token*.

CAPÍTULO 5

Considerações Finais

Neste trabalho foi desenvolvido um protocolo para *marketplace* de *NFTs* com *orderbook offchain* em redes descentralizadas para *blockchains* compatíveis com *EVMs*. O protocolo proposto foi desenvolvido em *solidity* com o auxilio do *framework hardhat*.

Inicialmente, foram avaliados protocolos de *marketplace* referências no mercado, também foram analisados padrões e *OP_CODES* das *EVMs* pertinentes ao desenvolvimento de um protocolo de *marketplace* e definido a *blockchain* da *Polygon* para publicar o projeto oficialmente. Na sequência, foi definida a estrutura do protocolo desenvolvido para este trabalho, as alterações necessárias e otimizações a serem feitas. Posteriormente, foi implementado o protocolo em *solidity*. Finalmente, o protocolo implementado foi testado, auditado e publicado nas redes *Mumbai* e *Polygon*.

O protocolo implementado obteve um consumo de gás adequado em diversas redes testadas, por exemplo, na rede alvo *Polygon* o consumo de gás é irrisório possibilitando uma atualização de implementação con-

siderando um *orderbook onchain*. Foram desenvolvidos e executados testes lógicos que validam diversos cenários diferentes para garantir que a lógica de negócio implementada esta correta e sem *bugs*.

O protocolo desenvolvido ao longo desse trabalho obteve resultados adequados e está sendo usado por uma aplicação real com lançamento previsto para o final do mês de Abril de 2022.

Ao finalizar esse trabalho percebeu-se que algumas melhorias e expansões são possíveis considerando o estado atual do protocolo *Luxy*:

- Adaptar o *orderbook* para uma estrutura de dados dentro da *blockchain*, um *orderbook onchain*.
- Criar versões dos contratos de *Royalties Registry* que permitam que criadores de coleções de *NFT* possam facilmente adicionar seus modelos de *royalties* ao protocolo.
- Incrementar o protocolo para permitir a negociação de mais de um *NFT* por ordem. Além disso, publicar o contrato em outras redes compatíveis com *EVMs*, permitindo o acesso do *Luxy* a mais pessoas.

Bibliografia

[1] (). "The Exponential Growth of NFTs," endereço: https://linkedin.com/pulse/exponential-growth-nfts-digitaltradingcard?trk=organization-update-content_share-article.

[2] M. Fox. (). "The NFT market is now worth more than $7 billion," endereço: https://markets.businessinsider.com/news/currencies/nft-market-worth-7-billion-legal-issues-could-hinder-growth-2021-11.

[3] S. Haigney. (). "Next Stop: The Metaverse," endereço: https://www.sothebys.com/en/articles/next-stop-the-metaverse.

[4] C. Mozée. (). "Christie's sold $150 million of NFTs in 2021," endereço: https://markets.businessinsider.com/news/currencies/christies-nft-sales-total-beeple-auction-crypto-cryptopunks-2021-12.

[5] R. Setti. (). "Brasileiro transforma obras de arte em NFTs em feira que acontece em Nova York," endereço: https://blogs.oglobo.globo.com/capital/post/brasileiro-transforma-obras-de-arte-em-nfts-em-feira-que-acontece-em-nova-york.html.

[6] (). "The Largest NFT marketplace," endereço: https://opensea.io/.

[7] D. Analytics. (). "Dashboard daily sale volume Opensea," endereço: https://dune.xyz/queries/11385/22601.

[8] S. Kearns. (). "OpenSea Sets New Record," endereço: https://hypebeast.com/2022/2/opensea-new-record-nft-sales-january-2022.

[9] Fantástico. (). “Obra de arte digital é vendida por quase US$ 70 milhões,” endereço: https://g1.globo.com/fantastico/noticia/2021/03/21/obra-de-arte-digital-e-vendida-por-quase-us-70-milhoes-entenda-o-que-e-o-nft.ghtml.

[10] (). “Non-Fungible Token (NFT) Definition,” endereço: https://www.investopedia.com/non-fungible-tokens-nft-5115211.

[11] (). “Smart contracts defined,” endereço: https://www.ibm.com/se-en/topics/smart-contracts.

[12] (). “Solana Docs,” endereço: https://docs.solana.com/developing/on-chain-programs/overview.

[13] (). “Turing Complete,” endereço: https://academy.binance.com/en/glossary/turing-complete.

[14] G. Suyambu, M. Anand e M. Janakirani, “Blockchain – A Most Disruptive Technology On The Spotlight Of World Engineering Education Paradigm,” *Procedia Computer Science*, v. 172, pp. 1–7, jan. de 2020. DOI: 10.1016/j.procs.2020.05.023.

[15] S. Kemp. (). “DIGITAL 2021: GLOBAL OVERVIEW REPORT,” endereço: https://datareportal.com/reports/digital-2021-global-overview-report#:~:text=Online%5C%20time%5C%20jumps&text=The%5C%20latest%5C%20findings%5C%20from%5C%20GWI,year%5C%20increase%5C%20of%5C%204%5C%20percent..

[16] M. Kilzi. (). “The Anatomy Of Personal Data Sovereignty,” endereço: https://www.forbes.com/sites/forbesbusinesscouncil/2021/05/04/the-anatomy-of-personal-data-sovereignty/?sh=25d267f961e1.

[17] F. M. Benčić e I. Podnar Žarko, “Distributed Ledger Technology: Blockchain Compared to Directed Acyclic Graph,” em *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, 2018, pp. 1569–1570. DOI: 10.1109/ICDCS.2018.00171.

[18] S. Nakamoto. (). “Bitcoin: A Peer-to-Peer Electronic Cash System,” endereço: https://bitcoin.org/bitcoin.pdf.

[19] S. Ghimire e H. Selvaraj, “A Survey on Bitcoin Cryptocurrency and its Mining,” dez. de 2018, pp. 1–6. DOI: 10.1109/ICSENG.2018.8638208.

[20] W. Macharia, “Cryptographic Hash Functions,” mai. de 2021.

[21] (). “Importance of Hashing,” endereço: https://www.geeksforgeeks.org/importance-of-hashing/#:~:text=Hashing%5C%20gives%5C%20a%5C%20more%5C%20secure,doesn't%5C%20define%5C%20the%5C%20speed..

[22] R. Kutcha. (). “The hash – a computer file’s digital fingerprint,” endereço: https://newtech.law/en/the-hash-a-computer-files-digital-fingerprint/.

[23] (). “What is hashing ?” Endereço: https://dcxlearn.com/blockchain/what-is-hashing/.

[24] (). “SHA256 HASH CORE,” endereço: https://opencores.org/projects/sha256_hash_core.

[25] A. Tsankov. (). “ECIP-1049: Why Ethereum Classic should Adopt Keccak256 for its Proof of Work Algorithm.,” endereço: https://antsankov.medium.com/ecip-1049-why-ethereum-classic-should-adopt-keccak256-for-its-proof-of-work-algorithm-e45aee32d8a9.

[26] (). “SHA,” endereço: https://www.pcmag.com/encyclopedia/term/sha#:~:text=(Secure%5C%20Hash%5C%20Algorithm)%5C%20A%5C%20family,Standards%5C%20%5C%26%5C%20Technology%5C%20(NIST)..

[27] (). “Guia Sobre Merkle Trees e Merkle Roots,” endereço: https://academy.binance.com/pt/articles/merkle-trees-and-merkle-roots-explained.

[28] (). “Merkle tree,” endereço: https://en.wikipedia.org/wiki/Merkle_tree.

[29] (). “Signed system volume security in macOS,” endereço: https://support.apple.com/sr-rs/guide/security/secd698747c9/web.

[30] F. R. Ortega. (). “EXT in Detail High-Performance Database Research Center,” endereço: https://slideplayer.com/slide/12631185/. Hash trees slides.

[31] (). “Peer-to-peer,” endereço: https://en.wikipedia.org/wiki/Peer-to-peer.

[32] A. Tiwana, “Chapter 5 - Platform Architecture,” em *Platform Ecosystems*, A. Tiwana, ed., Boston: Morgan Kaufmann, 2014, pp. 73–116, ISBN: 978-0-12-408066-9. DOI: https://doi.org/10.1016/B978-0-12-408066-9.00005-9. endereço: https://www.sciencedirect.com/science/article/pii/B9780124080669000059.

[33] ——, “Chapter 5 - Platform Architecture,” em *Platform Ecosystems*, A. Tiwana, ed., Boston: Morgan Kaufmann, 2014, pp. 73–116, ISBN: 978-0-12-408066-9. DOI: https://doi.org/10.1016/B978-0-12-408066-9.00005-9. endereço: https://www.sciencedirect.com/science/article/pii/B9780124080669000059.

[34] F. Baccelli, F. Mathieu, I. Norros e R. Varloot, “Can P2P Networks be Super-Scalable?” *Proceedings - IEEE INFOCOM*, abr. de 2013. DOI: 10.1109/INFCOM.2013.6566973.

[35] I. H. Witten, M. Gori e T. Numerico, “CHAPTER 7 - THE DRAGONS EVOLVE The new theater: the audience watches the web evolve.,” em *Web Dragons*, sér. The Morgan Kaufmann Series in Multimedia Information and Systems, I. H. Witten, M. Gori e T. Numerico, ed., San Francisco: Morgan Kaufmann, 2007, pp. 211–242, ISBN: 978-0-12-370609-6. DOI: https://doi.org/10.1016/B978-012370609-6/50010-2. endereço: https://www.sciencedirect.com/science/article/pii/B9780123706096500102.

[36] (). “P2P(Peer To Peer) File Sharing,” endereço: https://www.geeksforgeeks.org/p2ppeer-to-peer-file-sharing/.

[37] (). “Blockchain: What are Nodes and SuperNodes?” Endereço: https://www.thestartupfounder.com/blockchain-what-are-nodes-and-supernodes/.

[38] (). “Blockchain & Role of P2P Network,” endereço: https://www.blockchain-council.org/blockchain/blockchain-role-of-p2p-network/.

[39] (). “13 years ago, Satoshi Nakamoto sent Hal Finney 10 bitcoins in Bitcoin’s first transaction ever,” endereço: https://coingeek.com/13-years-ago-satoshi-nakamoto-sent-hal-finney-10-bitcoins-in-bitcoins-first-transaction-ever/#:~:text=On%5C%20January%5C%2012%5C%2C%5C%202009%5C%2C%5C%20exactly,Hal%5C%20Finney%5C%2C%5C%20a%5C%20revered%5C%20cryptographer..

[40] M. Iansiti e K. R. Lakhani. (). “The Truth About Blockchain,” endereço: https://hbr.org/2017/01/the-truth-about-blockchain.

[41] (). “What is Blockchain Technology and How Does It Work?” Endereço: https://www.simplilearn.com/tutorials/blockchain-tutorial/blockchain-technology.

[42] (). "What Is a Blockchain Consensus Algorithm?" Endereço: https://academy.binance.com/en/articles/what-is-a-blockchain-consensus-algorithm.

[43] (). "The Blockchain Trilemma: Fast, Secure, and Scalable Networks," endereço: https://www.gemini.com/cryptopedia/blockchain-trilemma-decentralization-scalability-definition.

[44] M. R. Nipun Bansal Mrinal Singhal e L. Arora, "Understanding and Analyzing Consensus Algorithms for Blockchain," *Nat. Volatiles & Essent. Oils*, 2021. DOI: https://www.nveo.org/index.php/journal/article/view/662.

[45] J. Frankenfield. (). "Block Time," endereço: https://www.investopedia.com/terms/b/block-time-cryptocurrency.asp#:~:text=Block%5C%20time%5C%20is%5C%20the%5C%20measure,it%5C%20was%5C%20introduced%5C%20in%5C%202009..

[46] ——, (). "Nonce," endereço: https://www.investopedia.com/terms/n/nonce.asp#:~:text=Nonce%5C%20in%5C%20Cryptocurrency%5C%3F-,A%5C%20nonce%5C%20is%5C%20an%5C%20abbreviation%5C%20for%5C%20%5C%22number%5C%20only%5C%20used%5C%20once,in%5C%20order%5C%20to%5C%20receive%5C%20cryptocurrency..

[47] ——, (). "Cryptocurrency Difficulty," endereço: https://www.investopedia.com/terms/d/difficulty-cryptocurrencies.asp#:~:text=A%5C%20high%5C%20cryptocurrency%5C%20difficulty%5C%20means,the%5C%20network's%5C%20hash%5C%20power%5C%20changes..

[48] J. Kehrli. (). "Blockchain explained," endereço: https://www.niceideas.ch/blockchain_explained.pdf.

[49] E. Foundation. (). "PROOF-OF-STAKE," endereço: https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/.

[50] L. Conway. (). "Proof-of-Work vs. Proof-of-Stake: Which Is Better?" Endereço: https://blockworks.co/proof-of-work-vs-proof-of-stake-whats-the-difference/#:~:text=Proof%5C%2Dof%5C%2Dstake%5C%20validators%5C%20only,energy%5C%20costs%5C%20that%5C%20can%5C%20fluctuate..

[51] (). “SIDECHAINS,” endereço: https://ethereum.org/en/developers/docs/scaling/sidechains/#:~:text=A%5C%20sidechain%5C%20is%5C%20a%5C%20separate,by%5C%20a%5C%20two%5C%2Dway%5C%20bridge..

[52] R. Davis. (). “ETHEREUM TRANSACTIONS,” endereço: https://www.theengineeringprojects.com/2021/06/ethereum-transactions.html.

[53] (). “Unspent Transaction Output,” endereço: https://en.wikipedia.org/wiki/Unspent_transaction_output.

[54] A. V. Marek Palatinus Pavol Rusnak e S. Bowe. (). “BIP39,” endereço: https://github.com/bitcoin/bips/blob/master/bip-0039.mediawiki.

[55] (). “Ethereum Virtual Machine (EVM),” endereço: https://coinmarketcap.com/alexandria/glossary/ethereum-virtual-machine-evm.

[56] (). “Ethereum,” endereço: https://en.wikipedia.org/wiki/Ethereum.

[57] (). “ETHEREUM VIRTUAL MACHINE (EVM),” endereço: https://ethereum.org/en/developers/docs/evm/.

[58] (). “INTRO TO ETHEREUM,” endereço: https://ethereum.org/en/developers/docs/intro-to-ethereum/.

[59] (). “THE BLOCKCHAIN GENERATIONS,” endereço: https://www.ledger.com/academy/blockchain/web-3-the-three-blockchain-generations#:~:text=Key%5C%20Takeaways%5C%3A,the%5C%20hands%5C%20of%5C%20the%5C%20people..

[60] (). “INTRODUCTION TO THE ETHEREUM STACK,” endereço: https://ethereum.org/en/developers/docs/ethereum-stack/.

[61] S. Sen. (). “How to access Ethereum Mempool,” endereço: https://www.quicknode.com/guides/defi/how-to-access-ethereum-mempool.

[62] T. T. (). “Ethereum EVM illustrated,” endereço: https://takenobu-hs.github.io/downloads/ethereum_evm_illustrated.pdf. (slide 109).

[63] (). “SMART CONTRACT LANGUAGES,” endereço: https://ethereum.org/en/developers/docs/smart-contracts/languages/.

[64] (). “The Ethereum Virtual Machine,” endereço: https://cypherpunks-core.github.io/ethereumbook/13evm.html#evm_architecture. chapter 13.

[65] E. Kyi e T. Liu. (). “Ethereum and the Halting Problem,” endereço: https://jcrouser.github.io/CSC250/projects/etherium.html.

[66] N. Venkitaramanan. (). “Why is Bitcoin not Turing complete?” Endereço: https://www.quora.com/Why-is-Bitcoin-not-Turing-complete.

[67] V. Buterin. (). “Ethereum Whitepaper,” endereço: https://ethereum.org/en/whitepaper/.

[68] D. G. WOOD. (). “ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER BERLIN VERSION,” endereço: https://ethereum.github.io/yellowpaper/paper.pdf.

[69] (), endereço: https://geth.ethereum.org/.

[70] (). “Elliptic Curve Digital Signature Algorithm,” endereço: https://en.wikipedia.org/wiki/Elliptic_Curve_Digital_Signature_Algorithm.

[71] (). “Cryptography,” endereço: https://cypherpunks-core.github.io/ethereumbook/04keys-addresses.html. chapter 4.

[72] M. Zuidhoorn. (). “The Magic of Digital Signatures on Ethereum,” endereço: https://medium.com/mycrypto/the-magic-of-digital-signatures-on-ethereum-98fe184dc9c7.

[73] (). “Ethereum Improvement Proposals,” endereço: https://eips.ethereum.org/#:~:text=EIPs,the%5C%20Ethereum%5C%20Project%5C%20Management%5C%20repository..

[74] (). “Gas (Ethereum),” endereço: https://www.investopedia.com/terms/g/gas-ethereum.asp.

[75] (). “GAS AND FEES,” endereço: https://ethereum.org/en/developers/docs/gas/.

[76] L. L. Remco Bloemen e J. Evans. (). “EIP-712: Ethereum typed structured data hashing and signing,” endereço: https://eips.ethereum.org/EIPS/eip-712.

[77] (). “What is Polygon (MATIC)?” Endereço: https://www.coinbase.com/pt/learn/crypto-basics/what-is-polygon.

[78] (). “Layer-1 and Layer-2 Blockchain Scaling Solutions,” endereço: https://www.gemini.com/cryptopedia/blockchain-layer-2-network-layer-1-network.

[79] J. Sidhu. (). “Network-Enhanced Virtual Machine (NEVM),” endereço: https://syscoin.org/news/nevm-and-related-changes-to-syscoin.

[80] (). “Ethereum’s scaling issues strike again as TIME Magazine’s NFTs sell for 30 times their price,” endereço: https://www.businessinsider.in/cryptocurrency/news/chinas-crypto-crackdown-is-shaking-markets-but-its-been-trying-to-rein-in-the-sector-for-years-heres-a-timeline-of-beijings-regulatory-sweep-/articleshow/86491203.cms.

[81] (). “NFTs Overview,” endereço: https://dappradar.com/nft.

[82] (). “New to Polygon?” Endereço: https://docs.polygon.technology/docs/home/new-to-polygon/.

[83] (). “Plasma (Blockchain),” endereço: https://pt.wikipedia.org/wiki/Plasma_(Blockchain).

[84] (). “Blockchain bridges development,” endereço: https://boostylabs.com/blockchain/bridges#:~:text=Blockchain%5C%20bridges%5C%20are%5C%20software%5C%20used,them%5C%20in%5C%20the%5C%20required%5C%20one..

[85] S. Nailwal. (). “Polygon (MATIC): The Swiss Army Knife of Ethereum Scaling,” endereço: https://www.gemini.com/cryptopedia/polygon-crypto-matic-network-dapps-erc20-token.

[86] A. Takyar. (). “Step-By-Step Guide on how to develop an NFT marketplace plataform,” endereço: https://www.leewayhertz.com/develop-nft-marketplace-platform/.

[87] (). “Project Wyvern,” endereço: https://github.com/ProjectWyvern.

[88] (). “Rarible Protocol,” endereço: https://github.com/rarible.

[89] (). “Digital Assets: Cryptocurrencies vs. Tokens,” endereço: https://www.gemini.com/cryptopedia/cryptocurrencies-vs-tokens-difference.

[90] F. Vogelsteller e V. Buterin. (). “EIP-20: Token Standard,” endereço: https://eips.ethereum.org/EIPS/eip-20.

[91] J. E. William Entriken Dieter Shirley e N. Sachs. (). "EIP-721: Non-Fungible Token Standard," endereço: https://eips.ethereum.org/EIPS/eip-721.

[92] (). "EIP-1155: Multi Token Standard," endereço: https://eips.ethereum.org/EIPS/eip-1155.

[93] (). "The standard for secure blockchain applications," endereço: https://openzeppelin.com/.

[94] (). "ERC-721 NON-FUNGIBLE TOKEN STANDARD," endereço: https://ethereum.org/en/developers/docs/standards/tokens/erc-721/.

[95] (). "EIP-165: Standard Interface Detection," endereço: https://eips.ethereum.org/EIPS/eip-165.

[96] (). "ERC-721," endereço: https://docs.openzeppelin.com/contracts/4.x/api/token/erc721.

[97] (). "What is IPFS?" Endereço: https://docs.ipfs.io/concepts/what-is-ipfs/.

[98] (). "Stablecoins," endereço: https://www.investopedia.com/terms/s/stablecoin.asp.

[99] F. Martinelli. (). "What Is an Automated Market Maker (AMM)?" Endereço: https://medium.com/balancer-protocol/what-is-an-automated-market-maker-amm-588954fc5ff7.

[100] (). "Protocol Components," endereço: https://wyvernprotocol.com/docs/protocol-components#:~:text=Wyvern%5C%20is%5C%20a%5C%20first%5C%2Dorder,ratio%5C%20and%5C%20a%5C%20maximum%5C%20amount)..

[101] (). "Solidity," endereço: https://docs.soliditylang.org/en/v0.8.13/.

[102] E. Pillmore. (). "The EVM Is Fundamentally Unsafe," endereço: https://medium.com/hackernoon/the-evm-is-fundamentally-unsafe-d69f2e3b908b.

[103] (). "Solidity," endereço: https://en.wikipedia.org/wiki/Solidity.

[104] S. Vale. (). "Solidity: a linguagem de programação para criar os smart contracts na Ethereum," endereço: https://www.voitto.com.br/blog/artigo/linguagem-de-programacao-solidity.

[105] (). "Defi Pulse," endereço: https://www.defipulse.com/.

[106] (). “Do Your Own Research (DYOR),” endereço: https://academy.binance.com/en/glossary/do-your-own-research.

[107] U. Securities e E. Comission. (). “Ponzi schemes Using virtual Currencies,” endereço: https://www.sec.gov/files/ia_virtualcurrencies.pdf.

[108] (). “DAPP DEVELOPMENT FRAMEWORKS,” endereço: https://ethereum.org/en/developers/docs/frameworks/.

[109] (). “Hardhat Network,” endereço: https://hardhat.org/hardhat-network/.

[110] (). “Overview,” endereço: https://hardhat.org/getting-started/.

[111] (). “npx,” endereço: https://www.npmjs.com/package/npx.

[112] (). “web3.js - Ethereum JavaScript API,” endereço: https://web3js.readthedocs.io/en/v1.7.1/.

[113] (). “JSON-RPC API,” endereço: https://ethereum.org/en/developers/docs/apis/json-rpc/.

[114] (). “INTRODUCTION TO DAPPS,” endereço: https://ethereum.org/en/developers/docs/dapps/.

[115] (). “How to Connect to Ethereum Nodes,” endereço: https://moralis.io/how-to-connect-to-ethereum-nodes/.

[116] L. Campbell. (). “Top DeFi Wallets,” endereço: https://defirate.com/wallet/.

[117] O. Renwick. (). “What’s In A Self-Custody (Non-Custodial) Wallet, Anyway?” Endereço: https://consensys.net/blog/metamask/whats-in-a-self-custody-non-custodial-wallet-anyway/.

[118] (). “Syscoin,” endereço: https://syscoin.org/.

[119] (). “Not Your Keys, Not Your Coins. It’s that simple.,” endereço: https://www.ledger.com/academy/not-your-keys-not-your-coins-why-it-matters.

[120] D. G. Birch. (). “Not Your Keys, Not Your Coins? Whatever,” endereço: https://www.forbes.com/sites/davidbirch/2021/10/15/not-your-keys-not-your-coins-whatever/?sh=4fff3b014c1f.

[121] H. Yang. (). “Herong Yang Tutorial Books,” endereço: https://www.herongyang.com/.

[122] ——, (). "MetaMask - Browser Based Ethereum Wallet," endereço: https://www.herongyang.com/Ethereum/MetaMask-Browser-Based-Ethereum-Wallet.html.

[123] (). "Introduction," endereço: https://docs.metamask.io/guide/.

[124] R. G. Schmidt. (). "EIP-2470: Singleton Factory," endereço: https://eips.ethereum.org/EIPS/eip-2470.

[125] V. Buterin. (). "EIP-1014: Skinny CREATE2," endereço: https://eips.ethereum.org/EIPS/eip-1014.

[126] (). "Salt Cryptography," endereço: https://en.wikipedia.org/wiki/Salt_(cryptography).

[127] B. M. Zack Burks James Morgan e J. Seibel. (). "EIP2981: NFT Royalty Standard," endereço: https://eips.ethereum.org/EIPS/eip-2981.

[128] J. Fernando. (). "Basis Points (BPS)," endereço: https://www.investopedia.com/terms/b/basispoint.asp.