UNIVERSIDADE REGIONAL INTEGRADA DO ALTO URUGUAI E DAS MISSÕES - CAMPUS DE ERECHIM
DEPARTAMENTO DE ENGENHARIAS E CIÊNCIA DA COMPUTAÇÃO
CURSO DE CIÊNCIA DA COMPUTAÇÃO

CRISTIAN JOSÉ AMBROSI

SMART SIGN: PLATAFORMA DE CONTRATOS DIGITAIS UTILIZANDO SMART CONTRACTS DA BLOCKCHAIN

ERECHIM - RS
2019

CRISTIAN JOSÉ AMBROSI

SMART SIGN:
PLATAFORMA DE CONTRATOS DIGITAIS UTILIZANDO SMART CONTRACTS DA BLOCKCHAIN

Trabalho de Conclusão de Curso apresentado como requisito parcial à obtenção do grau de Bacharel, Departamento de Engenharias e Ciência da Computação da Universidade Regional Integrada do Alto Uruguai e das Missões - Campus de Erechim.

Orientador: Gerson Groth

ERECHIM - RS
2019

AGRADECIMENTOS

Em primeiro lugar agradeço a minha mãe, Edite F. Duranti e a minha irmã, Cristina Ambrosi, pelo apoio recebido durante todo o curso e principalmente durante a realização deste trabalho, foram compreensivas com relação a momentos de estresse e disponibilidade de tempo. Agradeço a minha prima Clarice Ferrazzo e meu tio Dirceu J. Duranti por todo o apoio que precisei nesse periodo. Agradeço também a minha namorada pela paciência, compreensão e de manter-me motivado.

Agradeço a todos os professores pelo conhecimento transmitido e, de maneira especial, agradeço ao professor Gerson Groth pelo auxílio oferecido na realização deste trabalho. Agradeço aos meus colegas do curso, que tornaram esse período mais prazeroso, pelas trocas de risos, emoções, jogatinas de *Mortal Kombat* e desesperos diante as provas difíceis. No mais, meu muito obrigado.

Você tem que ser o que você realmente é. Pois se você não for quem você é, afinal quem é você?
(FUNNIE, Doug, 1991)

RESUMO

Contratos são instrumentos fundamentais para formalizar fechamentos de negócios ou prestações de serviços, proporcionando segurança e confiabilidade às partes interessadas. Entretanto, o baixo desempenho burocrático e eventualmente o alto custo, dificultam a sua realização, tornando o processo desgastante e exacerbado. Neste contexto, o trabalho em questão tem por objetivo o desenvolvimento de uma ferramenta que automatize a criação de contratos digitais na *blockchain Ethereum*, utilizando a tecnologia de *smart contracts* ou "contratos inteligentes". Para tal, foi desenvolvido um serviço na linguagem de programação *Solidity*, responsável pela programação dos *smart contracts*. Posteriormente, foi desenvolvida a ferramenta de comunicação com este serviço, utilizando a linguagem de programação *JavaScript* e a biblioteca *ReactJS*, possibilitando a criação de contratos digitais na *blockchain*. Ao final deste trabalho, foi possível apresentar uma ferramenta simples, capaz de automatizar a criação de contratos em uma rede descentralizada.

Palavras-chave: *Smart Contracts*. *Blockchain Ethereum*. *Solidity*. Contratos Digitais.

ABSTRACT

Contracts are the key tools for formalizing business closings or service delivery, providing security and reliability to stakeholders. However, the low bureaucratic performance and sometimes the high cost make it difficult to perform, making the process exhausting and exacerbated. In this context, the present work aims to develop a tool that automates the creation of digital contracts in blockchain Ethereum, using smart contract technology. To this end, a service was developed in the Solidity programming language, responsible for programming smart contracts. Later, the tool that communicates with this service was developed, using the JavaScript programming language and the ReactJS library, enabling the creation of digital contracts on the blockchain. At the end of this process, it was possible to present a simple tool capable of automating the creation of contracts in a decentralized network.

Keywords: Smart Contracts. Blockchain Ethereum. Solidity. Digital Contracts.

LISTA DE ILUSTRAÇÕES

LISTA DE QUADROS

LISTA DE ABREVIATURAS E SIGLAS

CCE	Criptografia de Curvas Elípticas
CSS	Cascading Style Sheets
DApps	Decentralized Application
DOM	Document Object Model
DPoS	Delegated Proof-of-Stake
ECMA	European Computer Manufacturers Association
EVM	Ethereum Virtual Machine
IDE	Integrated Development Environment
ISO/IEC	International Organization for Standardization / International Electrotechnical Commission
HTTP	HyperText Transfer Protocol
ML	Meta Language
MP	Medida Provisória
NPM	Node Package Manager
P2P	Peer-to-Peer
PBFT	Practical Byzantine Fault Tolerance
PDF	Portable Document Format
PHP	Personal Home Page
PoS	Proof-of-Stake
PoW	Proof-of-Work
SERPRO	Serviço Federal de Processamento de Dados
SPA	Single Page Application
UML	Unified Modeling Language

SUMÁRIO

1 INTRODUÇÃO

O surgimento da *blockchain* em 2008, inicialmente para ser um sistema capaz de registrar e armazenar a escrituração de transações realizadas com a moeda virtual *bitcoin*, acabou despertando a atenção do mercado financeiro para as inúmeras possibilidades que essa tecnologia poderia proporcionar, consequência de suas características de proteção criptográfica, base distribuída de dados e imutabilidade (ROCHA; PEREIRA; BRAGANÇA, 2018).

Assim, centenas de bifurcações da *blockchain* original foram criadas, com diferentes propósitos, destacando-se a *blockchain* da *Ethereum*, responsável por reacender a ideia de *smart contracts* ou "contratos inteligentes" de Nick Szabo, onde os contratos são capazes de se autoexecutarem tendo suas cláusulas escritas no código fonte. Os objetivos gerais dos *smart contracts* são satisfazer condições contratuais comuns, minimizar execuções maliciosas e a necessidade de autoridades intermediárias (ROCHA; PEREIRA; BRAGANÇA, 2018).

Nesse contexto, o presente trabalho propôs o desenvolvimento de uma ferramenta com o propósito de automatizar a criação de contratos digitais utilizando *smart contracts* da *Ethereum*, de forma simples e agilizando o processo de validação, assim garantindo sua confiabilidade através da criptografia e uma rede descentralizada.

Primeiramente, realizou-se o desenvolvimento de um serviço que possibilite a criação de *smart contracts*, utilizando a *Ethereum Virtual Machine* (EVM), que executa os contratos na *blockchain* da *Ethereum* e a linguagem de programação *Solidity* para a implementação dos *smart contracts*. Para a compilação, implantação e os testes dos *smart contracts* utilizou-se o *framework Truffle*.

Uma vez implementado e testado o serviço, o próximo passo consistiu no desenvolvimento da ferramenta de comunicação com este serviço, assim possibilitando a criação de contratos digitais. Para o desenvolvimento desta ferramenta utilizou-se a linguagem de programação *JavaScript*, a biblioteca *ReactJS*, juntamente com o *NodeJS*.

O trabalho também tem como propósito, o estudo de ferramentas disruptivas para serem aplicadas no desenvolvimento de aplicações *web*. Tendo em vista isso, no Capítulo dois são abordadas técnicas de desenvolvimento de aplicações descentralizadas, utilizando *blockchain*, bem como a apresentação e detalhamento das ferramentas para desenvolver o serviço de *smart contracts* para os contratos digitais.

No terceiro Capítulo são apresentadas as ferramentas utilizadas para o desenvolvimento da interface que comunica-se com o serviço da *blockchain*. Sendo estas a bibliteca *ReactJS*, a linguagem de programação *JavaScript*, o gerenciador de pacotes *NodeJS*, a linguagem de estilo CSS, o transpilador *Less* e o *framework Truffle*.

Após o estudo dos componentes utilizados para o desenvolvimento da aplicação, é apresentado no Capítulo quatro a análise de algumas aplicações já existentes no mercado que possuem objetivos semelhantes, bem como a proposta e objetivos da aplicação e os recursos utilizados para o desenvolvimento do mesmo.

A aplicação desenvolvida é descrita em seu estado final no quinto Capítulo, onde é apresentada a composição de suas telas, juntamente das funcionalidades de cada uma. Finalmente, no sexto Capítulo são apresentadas as conclusões finais da plataforma, e suas perspectivas futuras para o aprimoramento da mesma.

2 *BLOCKCHAIN*

A publicação do artigo Nakamoto (2008), através do pseudônimo de Satoshi Nakamoto (identidade ainda não confirmada), despertou grande interesse mundial. O artigo descrevia o funcionamento de uma moeda inteiramente digital, controlada por um protocolo e cuja a validação das transações financeiras era feita de maneira totalmente distribuída pelos nós de uma rede P2P, analisada na Subseção 2.1.3. Dessa maneira, a moeda circularia sem a necessidade de uma instituição reguladora ou uma autoridade central que validasse as transações.

O sistema foi colocado em operação em 2009 em código aberto desenvolvido de maneira colaborativa, dando início às primeiras transações (PIRES, 2016). Com a sua popularização, diversas criptomoedas começaram a surgir, como o *Litecoin*, *Dogecoin* e a *Ethereum*, denominadas de modo geral como *Altcoins*.

2.1 Como funciona a *Blockchain*

Segundo Crosby et al. (2016), pode-se pensar na *Blockchain* como sendo um livro público, onde todas as transações feitas estão escritas neste livro; quando uma nova transação ocorre ela é testada contra a própria *Blockchain* verificando e garantindo que a mesma transação não tenha sido realizada anteriormente. A SERPRO definiu a *Blockchain* da seguinte forma:

> *Blockchain* é uma tecnologia de validação inviolável que tem a descentralização como medida de segurança. Cria consenso e confiança na comunicação direta entre duas partes, sem o intermédio de terceiros. O protocolo é adequado para cenários que requerem privacidade e controle de identidade e permissões. (SERPRO, 2017)

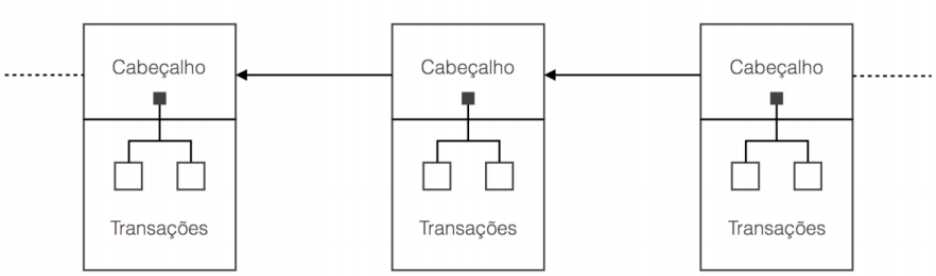
Narayanan et al. (2016), denomina a *Blockchain* como uma cadeia de registros imutáveis, públicos e distribuídos:

- **Cadeia:** Os registros estão cuidadosamente encadeados uns aos outros por meio de chaves públicas, entradas e saídas;
- **Imutável:** Uma vez que o registro é inserido na cadeia, ele não pode mais ser alterado;
- **Público:** A única condição necessária para que uma pessoa possa ter acesso aos registros do *blockchain* é que a mesma possua acesso à internet;
- **Distribuído:** A cadeia de registro não está armazenada em um único servidor central. Efetivamente, ela encontra-se replicada em milhões de máquinas distribuídas pelo mundo e nenhuma empresa ou indivíduo pode reivindicar a propriedade destes registros.

2.1.1 Blocos Encadeados

Segundo Narayanan et al. (2016), na *Blockchain* as transações são agrupadas em blocos, estando cada um conectado ao bloco anterior através de endereços *hash* formando uma cadeia. A Figura 1 apresenta uma visão do encadeamento de blocos.

Figura 1 – Cadeia de blocos da *Blockchain*



Fonte: Pires (2016)

Na área de transações estão todas as transações coletadas por aquele bloco. Na área de cabeçalho encontram-se o *hash* do cabeçalho do bloco anterior e a raiz da árvore de *Merkle*[1] das transações presentes no campo de transações. Comumente utilizando o algoritmo de *hash* SHA-256 analisado na Subsubseção 2.1.1.1, garante que modificações ou tentativas de fraudes sejam praticamente nulas, já que uma vez que um bloco é gerado fica impossível sua alteração (ULRICH, 2014).

Sabendo que o bloco está encadeado através de endereços *hash* a um bloco anterior a ele, uma fraude no mesmo pode ocasionar uma mudança de *hash*, possibilitando a detecção de tentativa de fraude pela rede (ULRICH, 2014).

2.1.1.1 *Hash* SHA-256

O SHA-256 é uma função de *hash* criptográfica unidirecional, ou seja, é possível usar uma função de *hash* para produzir uma saída ao receber um valor de entrada, porém é impossível usar a saída resultante para reconstruir sua entrada, sendo esse o principal recurso que o torna ideal para aplicação na rede *blockchain* (ASOLO, 2018).

2.1.2 Processo de Validação dos Blocos (Mineração)

Após a transação ser validada pelo processo de autenticação criptográfica, a mesma encontra-se pronta para ser coletada por um bloco e passar pelo processo de mineração. Um nó minerador da rede coleta as transações que deseja inserir na *blockchain* dentre todas as

[1] Árvore de *Merkle*, também conhecida como árvore de *Hash* Binária, é uma estrutura de dados usada para resumir e verificar com eficiência a integridade de grandes conjuntos de dados.

transações validadas que foram recebidas, calcula a raiz da árvore de *Merkle*, e passa a executar um algoritmo *proof of work* (prova de trabalho) (PIRES, 2016).

Uma prova de trabalho é um desafio criptográfico (algoritmo de consenso), utilizado para consentir que um nó realizou uma certa quantidade de trabalho, sendo um processo probabilístico e a probabilidade de sucesso dependente da dificuldade estabelecida (NAKAMOTO, 2008).

Um algoritmo de consenso é um processo de tomada de decisão para um grupo, onde cada indivíduo do grupo constrói e apoia a melhor decisão para todos do grupo (LAMOUNIER, 2018). Logo abaixo é citado alguns dos principais tipos de algoritmos de consenso:

- **Proof-of-Work (PoW):** Comumente utilizado pelas *blockchains* pela sua eficiência, tem a finalidade de demonstrar a um verificador que um determinado usuário passou por uma certa quantidade de trabalho computacional em um intervalo de tempo predeterminado. Porém, possui alto consumo de energia e necessita de grande poder computacional, o que resulta na concentração do poder de mineração entre aqueles que detêm controle de uma grande quantidade de *hardware* capaz de trabalhar em paralelo e o alto consumo de energia (JAKOBSSON; JUELS, 1999);
- **Practical Byzantine Fault Tolerance (PBFT):** Projetado para funcionar de maneira eficiente em sistemas assíncronos[2], tem como objetivo resolver os problemas de tolerância a falhas Bizantinas. Ou seja, alcançar um consenso na rede mesmo que alguns nós da rede falhem ou respondam com informações incorretas. Porém, tem como pontos negativos ataques *Sybil*, onde uma entidade invasora controla várias outras entidades (como nós de uma rede), e de escalabilidade; à medida que o número de nós na rede aumenta, o tempo necessário para responder à solicitação é aumentado consideravelmente (LIMA; GREVE, 2009);
- **Proof-of-Stake (PoS):** Possuindo um gasto de energia menor e não dependendo de tanto poder computacional em comparação ao PoW (KING, 2013), é um algoritmo de consenso em que o sistema faz uma escolha do nó que poderá criar um novo bloco por meio de um sorteio, cuja probabilidade de ser sorteado pode ser influenciada pela sua quantidade de moedas (ALIAGA et al., 2019);
- **Delegated Proof-of-Stake (DPoS):** Semelhante ao PoS, o DPoS possui *stakeholders* ou representantes, delegados em definir qual bloco será minerado, processo chamado de "votação de aprovação". Pode ser alterado o tamanho do bloco e quantidade de transações (ALIAGA; HENRIQUES, 2017).

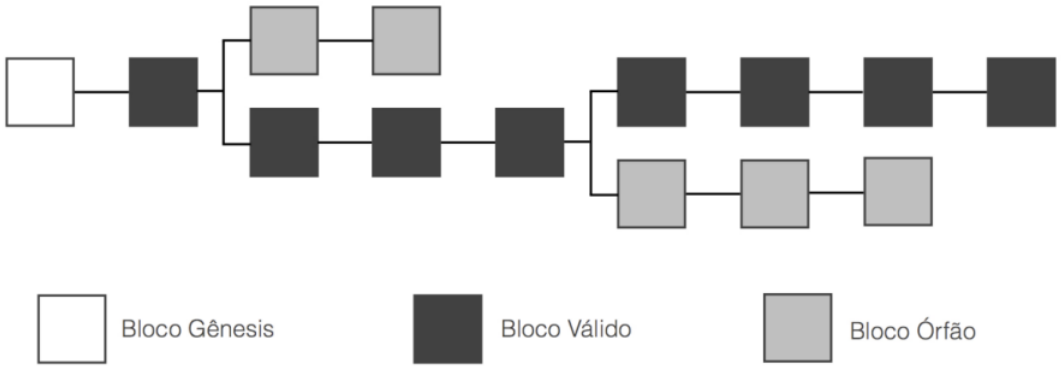
[2]Em um sistema assíncrono não existe sincronismo, sendo possível enviar diversas requisições em paralelo. A resposta retorna quando estiver pronta, não bloqueando o sistema, diferente de um sistema síncrono.

Assim que a prova de trabalho é dada como bem sucedida, o nó avisa toda a rede que resolveu o desafio e insere o bloco validado na *blockchain*. Cada um dos nós da rede recebe o novo bloco, então confirma-se de que se trata de um novo bloco, pois cada bloco possuiu sua numeração incremental, e assim adiciona o bloco à sua copia da *blockchain*. Esse processo repete-se por toda a rede, para que haja consenso entre os nós sobre o atual estado da *blockchain*. Feito isso, o minerador recebe o pagamento pelo seu trabalho no valor da unidade utilizada pela *blockchain* (PIRES, 2016).

Pode ocorrer a situação em que dois nós da rede realizem a prova de trabalho com pouca diferença de tempo, assim os dois propagarão blocos com a mesma numeração incremental. Essa situação é conhecida como “Corrida de Blocos”, onde o protocolo resolve utilizando o método da *Longest Chain Rule* ou "Maior Cadeia"(PIRES, 2016).

O método da Maior Cadeia consiste em identificar o maior segmento de blocos e torná-lo o segmento válido. Assim, cada nó adiciona blocos recém-criados a essa cadeia e a outra sequência de blocos seja abandonada, gerando os chamados "blocos órfãos". Deste modo, só existirá uma cadeia que liga o último bloco validado de volta ao primeiro bloco da cadeia (PIRES, 2016). Na Figura 2 é exemplificado o método da Maior Cadeia.

Figura 2 – Regra da Maior Cadeia

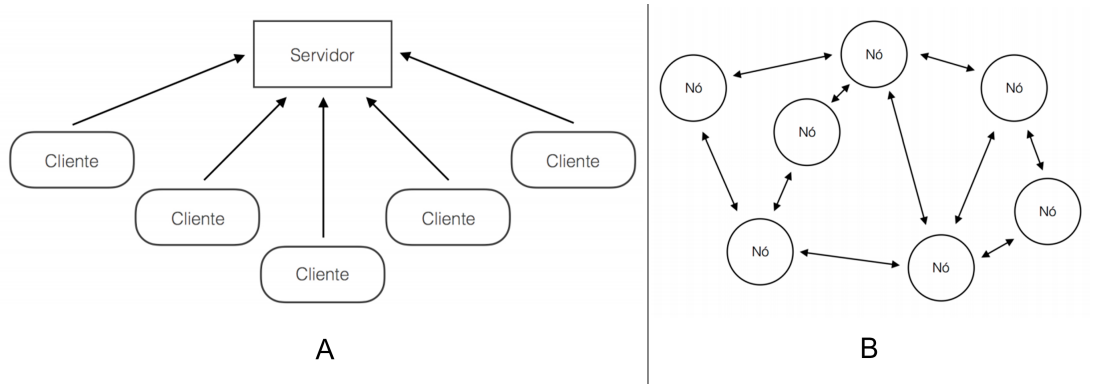


Fonte: Pires (2016)

2.1.3 Redes *Peer To Peer*

Uma rede *Peer To Peer* ou rede P2P, utiliza uma arquitetura descentralizada em que cada máquina, chamada de nó, executa ao mesmo tempo as funções de cliente e servidor (Figura 3, A). Este processo difere da arquitetura cliente-servidor em que uma máquina cliente apenas envia solicitações e aguarda pela resposta do servidor (Figura 3, B) (ROUSE, 2019). A Figura 3 exemplifica a estrutura dos dois modelos de arquitetura.

Figura 3 – Arquitetura Cliente-Servidor (A) e Arquitetura P2P (B)



Fonte: Pires (2016)

Segudo Rouse (2019), a rede P2P é caracterizada como um modelo transparente para o usuário final e de alta flexibilidade, normalmente cada nó da rede pode realizar *download* e *upload* ao mesmo tempo e novos nós podem entrar na rede, enquanto outros nós estão saindo. Diferente do modelo cliente-servidor, onde o desempenho é deteriorado à medida que o número de requisições dos clientes aumenta, no modelo P2P o desempenho geral da rede aumenta à medida que o número de nós cresce.

Na *blockchain* do *Bitcoin*, por exemplo, quando um novo nó é adicionado à rede, é realizado uma requisição pelo mesmo aos outros nós da rede, solicitando endereços de vários nós com os quais ele possa se conectar. Este processo se repete até que o nó adicionado esteja conectado a vários outros nós, de maneira razoavelmente aleatória. Uma vez conectado à rede e com os registros da *blockchain* atualizados, o nó está apto a publicar novos registros (PIRES, 2016).

Uma característica interessante de redes P2P é a capacidade de tolerância a erros. Quando um nó é desconectado da rede, a aplicação P2P pode continuar a operar utilizando outros nós que permanecem ativos. Na arquitetura cliente-servidor a conexão é interrompida se um servidor é desligado (ROUSE, 2019).

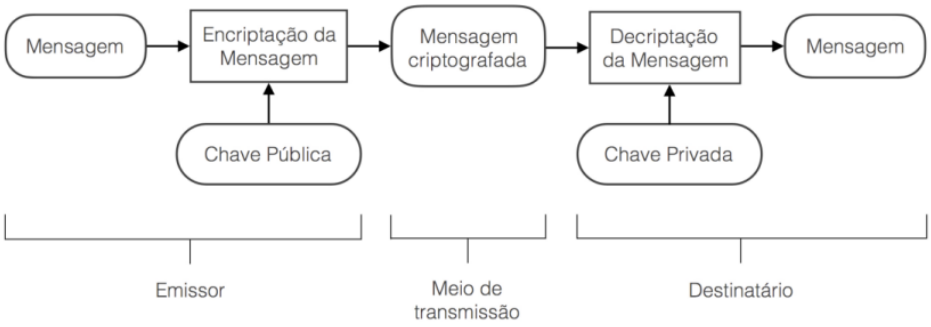
2.1.4 Criptografia Assimétrica

Segundo Salomaa (1996), a Criptografia Assimétrica ou Criptografia de Chave Pública, é o método criptográfico que faz uso de duas chaves diferentes, uma chave pública para criptografar e uma chave privada para descriptografar a mensagem. A Figura 4 ilustra o funcionamento de uma criptografia de chave pública.

> Em um sistema de criptografia assimétrica, a chave pública é divulgada abertamente para todos que queiram enviar uma mensagem ao dono da chave ou verificar a autenticidade de uma mensagem por ele enviada. A chave privada, por sua vez, é mantida em segredo e qualquer pessoa com acesso a ela será capaz de decodificar a mensagem criptografada ou atestar a autenticidade da mensagem por meio de uma assinatura digital. (SALOMAA, 1996)

A segurança de um sistema criptográfico assimétrico está na diferença entre o caminho de ida e o caminho de volta da operação matemática em que a criptografia está fundamentada. Quanto maior esta diferença, mais eficiente é o algoritmo (IMPAGLIAZZO; LUBY, 1989).

Figura 4 – Criptografia de Chave Pública



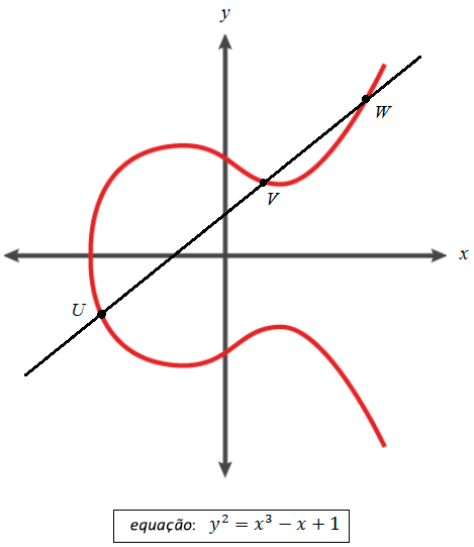
Fonte: Salomaa (1996)

Segundo Hankerson et al. (2004), atualmente a Criptografia de Curvas Elípticas (CCE) é considerada a mais eficiente, além de ser a mais utilizada para aplicações de blockchain. Sendo a mesma baseada nas propriedades matemáticas das curvas elípticas, descritas pela seguinte equação:

$$y^2 = x^3 + ax + b$$

Assim como exemplificado na Figura 5, as chaves públicas e privadas do algoritmo são calculadas a partir de repetidas operações de adição de coordenadas polares, obtidas pela interseção da curva elíptica com diversas retas tangentes (HANKERSON et al., 2004).

Figura 5 – Exemplo de um Ponto pertecente a uma Curva Elíptica



Fonte: Hankerson et al. (2004)

2.2 *Blockchain* da *Ethereum*

Fundada em 2014 por Vitalik Buterin, a *Ethereum* é um sistema de contratos inteligentes baseado em *blockchain*, composta por máquinas virtuais descentralizadas denominadas de *Ethereum Virtual Machines* (EVM), que executam os *smart contracts* (ETHEREUM, 2016).

Ethereum é uma plataforma aberta, que permite à qualquer pessoa construir e usar aplicativos descentralizados, sendo estes executados na tecnologia *blockchain*. Como o *Bitcoin*, ninguém controla ou possui *Ethereum*, mas ao contrário do protocolo *Bitcoin*, a *Ethereum* foi projetada para ser adaptável e flexível (ETHEREUM, 2016).

> *Ethereum* é uma *blockchain* programável. Ao invés de fornecer aos desenvolvedores um conjunto de operações predefinidas (por exemplo, transações de bitcoin), a *Ethereum* permite que os desenvolvedores criem suas próprias operações com qualquer complexidade. Desta forma, ela serve como uma plataforma para muitos tipos diferentes de aplicações *blockchain* descentralizadas, incluindo, mas não limitando-se a criptomoedas. (ETHEREUM, 2016)

Existem dois tipos de contas diferentes na *Ethereum*. Ambas identificadas por um endereço *Ethereum*, que são:

- **Externally Owned Accounts (EOA):** As EOAs ou "Contas de Propriedade Externa", são controladas pelos usuários, geralmente por meio de *software*, como um aplicativo de carteira externo à plataforma *Ethereum*, ou seja, são contas simples, sem nenhum código associado ou armazenamento de dados. Sendo controladas por transações criadas e assinadas criptograficamente com uma chave privada;
- **Contas de Contrato:** São controladas pelo código do programa (*smart contract*) executadas pela EVM, tendo código associado e armazenamento de dados. Não possuem chaves privadas e, portanto, são auto-executáveis de maneira predeterminada prescrita pelo código do *smart contract*.

2.2.1 *Ethereum Virtual Machine*

Ethereum Virtual Machine (EVM) é a plataforma disponibilizada pela *Ethereum* para a execução das aplicações e *smart contracts*, garantindo que eles não tenham acesso aos estados uns dos outros e que a comunicação possa ser estabelecida sem interferências prejudiciais (GREVE et al., 2018).

> A EVM pode executar código de complexidade algorítmica arbitrária. Em termos de ciência da computação, a *blockchain Ethereum* é considerada *"Turing complete"*. Os desenvolvedores podem criar aplicativos que são executados na EVM usando linguagens de programação amigáveis, como por exemplo *JavaScript* e *Python*. (ETHEREUM, 2016)

A *Ethereum* é frequentemente chamada de "computador mundial". Assim como outras *blockchains*, ela possui o protocolo P2P. Cada um dos nós da rede executa a EVM e as mesmas instruções, mantendo "banco de dados" da *blockchain* atualizado por todos os nós conectados na rede (ETHEREUM, 2016).

Importante ressaltar que a paralelização massiva da computação em toda a rede *Ethereum* não foi feita para tornar sua execução mais eficiente. Esse processo, na verdade, torna os cálculos na *Ethereum* muito mais lentos e caros do que em um computador tradicional. Porém, a *Ethereum* executa a EVM para manter o consenso através da *blockchain* entre todos os nós. Este consenso descentralizado torna a *blockchain Ethereum* extremamente tolerante à falhas, garante que seu tempo de inatividade seja zero e torna os dados armazenados na *blockchain* sempre inalteráveis e resistentes à censura (ETHEREUM, 2016).

2.2.2 Custo e Recompensa da Rede pelo Trabalho

A *Ethereum* não é apenas uma criptomoeda, mas também uma plataforma programável, possibilitando a criação de aplicações que permitam tranferir e armazenar dados de maneira descentralizada. Porém, existe custo monetário para realizar estas operações, onde são pagas em *Ether*, que é calculado com base no custo computacional estimado em *Gas*.

2.2.2.1 *Ether*

Ether, ou "ETH", é a unidade monetária da *Ethereum*. O ETH é subdividido em unidades menores, até a menor unidade possível, chamada de *wei*. O valor do ETH é sempre representado internamente na EVM como um valor inteiro não assinado. Por exemplo, se for transacionado 1 ETH, a transação é codifica para 1000000000000000000 de *wei* como valor, pois 1 ETH é equivalente á 1 quintilhão de *wei* (1 x 10^{18} ou 1.000.000.000.000.000.000.000) (ANTONOPOULOS; WOOD, 2018).

O Quadro 1 mostra a representação de todas as denominações em *wei*. Suas unidades, seu nomes normais e seus nomes SI (*International System of Units*).

Quadro 1 – Denominações da moeda *Ether*.

Valor (em *wei*)	**Expoente**	**Nome comum**	**Nome SI**
1	1	*Wei*	*Wei*
1.000	10^3	*Babbage*	*Kilowei* ou *Femtoether*
1.000.000	10^6	*Lovelace*	*Megawei* ou *Picoether*
1.000.000.000	10^9	*Shannon*	*Gigawei* ou *Nanoether*
1.000.000.000.000	10^{12}	*Szabo*	*Microether* ou *Micro*
1.000.000.000.000.000	10^{15}	*Finney*	*Milliether* ou *Milli*
1.000.000.000.000.000.000	10^{18}	*Ether*	*Ether*
1.000.000.000.000.000.000.000	10^{21}	*Grand*	*Kiloether*
1.000.000.000.000.000.000.000.000	10^{24}		*Megaether*

Fonte: Antonopoulos e Wood (2018)

Segundo Antonopoulos e Wood (2018), quando é realizada uma transação, o remetente especifica em ETH o preço que está disposto a pagar por cada unidade de *Gas*, utilizando a Equação (1) logo abaixo. Isso permite que o mercado decida o preço do ETH em relação ao *Gas*.

$$taxaDeTransacao = totalDeGasUsado * precoGasPagoEth \quad (1)$$

Nesse mesmo âmbito, é importante frisar a diferença entre **Custo de *Gas***, que é o número de unidades de *Gas* necessárias para executar uma operação específica. E o **Preço de *Gas***, que é a quantidade de ETH paga por unidade de *Gas* ao enviar uma transação para a rede *Ethereum* (ANTONOPOULOS; WOOD, 2018).

2.2.2.2 *Gas de Bloco*

Gas de Bloco ou simplesmente *"Gas"*, é a unidade usada para calcular os recursos computacionais e de armazenamento necessários para executar ações na *blockchain* da *Ethereum* e recompensar os mineradores da rede pelo seu trabalho em ETH, levando em consideração todas as etapas computacionais realizadas pelas transações e a execução inteligente do código do contrato (ANTONOPOULOS; WOOD, 2018).

Além de ser uma unidade usada para calcular o valor gasto mensionado no parágrafo anterior, o *Gas* é usado para evitar *loops* infinitos ou outros desperdícios computacionais na rede. É indicado um custo de *Gas* limite utilizado em cada operação na EVM, especificado coletivamente pelos mineradores da rede. Se a operação ultrapassar o limite ela será interrompida e a transação será revertida. Mesmo que a operação não tenha sido bem-sucedida, será cobrado uma taxa ao remetente, uma vez que a rede mineradora realizou trabalho computacional até aquele momento (ANTONOPOULOS; WOOD, 2018).

> Embora o *Gas* tenha um preço, ele não pode ser "de propriedade" nem "gasto". O *Gas* existe apenas dentro do EVM, como uma contagem de quanto trabalho computacional está sendo executado. Ao remetente é cobrada uma taxa de transação em ETH, que é convertida em *Gas* para a contabilidade da EVM e, em seguida, volta ao ETH como uma taxa de transação paga aos mineiros. (ANTONOPOULOS; WOOD, 2018)

Segundo Antonopoulos e Wood (2018), todo código de operação executado tem um custo em *Gas*, portanto, o suprimento de *Gas* da EVM é reduzido à medida que o programa avança. Antes de cada operação, o EVM verifica se há *Gas* suficiente para pagar pela execução da operação, como resumido nos ítens logo abaixo:

- Se a EVM atingir o final da execução com êxito, o custo de *Gas* usado será pago à mineradora como uma taxa de transação, convertida em ETH com base no preço de *Gas* especificado na transação, como mostrado na Equação (2):

$$taxaDoMineiro = custoDeGas * precoDeGas \quad (2)$$

- O *Gas* restante da tranzação é reembolsado ao remetente, novamente convertido em *Ether* com base no preço de *Gas* especificado na transação, como mostrado na Equação (3) e Equação (4):

$$gasRestante = limiteDeGas - custoDeGas \tag{3}$$

$$etherReembolsado = gasRestante * precoDeGas \tag{4}$$

2.2.3 *Wallet* ou Carteira Digital

Segundo Antonopoulos e Wood (2018), uma Carteira Digital, é um *software* que serve como interface principal do usuário. Ela controla o acesso ao dinheiro de um usuário, gerenciando chaves e endereços, rastreando o saldo e criando e assinando transações. Além disso, algumas carteiras como a da *Ethereum* podem interagir com *smart contracts*, utilizando seu endereço de chave pública como assinatura para garantir a autenticidade do contrato.

> As carteiras da *Ethereum* contêm chaves, não ETH ou *tokens*. As carteiras são como chaveiros que contêm pares de chaves públicas e privadas. Os usuários assinam transações com as chaves privadas, provando assim que possuem o ETH, que é armazenado na *blockchain*. (ANTONOPOULOS; WOOD, 2018)

2.2.3.1 Tipos de Carteira Digital

Existem diferentes tipos de carteiras digitais disponíveis, todas com seus prós e contras. Abaixo estão listadas com uma breve explicação:

- **Carteira de Papel:** considerada a carteira mais segura, pois na prática é apenas um pedaço de papel com o código escrito. Tendo como maiores vantagens não estarem conectadas à internet e não precisarem serem armazenadas no computador, tornado a informação imune a ataques de *hackers* (ETHHUB, 2019);
- **Carteiras Móveis:** são carteiras com o cliente disponível para *smartphones*. O maior benefício é sua comodidade e facilidade de usar em deslocamentos,
- **Carteiras *Desktop*:** são executadas no computador do usuário, sendo possível baixar um cliente completo da *blockchain* ou uma versão mais leve. São carteiras seguras, desde que o computador não esteja comprometido, porém suscetíveis a falhas (ETHHUB, 2019);
- **Carteiras da *Web*:** utilizam armazenamento em nuvem sendo acessíveis de qualquer lugar. Geralmente são mais rápidas que as outras carteiras, porém suas chaves de endereço e criptografia ficam armazenadas *online* em servidores de terceiros, possibilitando a chance de sofrer ataques de *hackers* (ETHHUB, 2019);
- **Carteiras *Hardware*:** são parecidas com discos rígidos portáteis, geram as chaves pública e privada de maneira *off-line*. Como não são conectadas à internet, são completamente imunes a ataques de *hackers*, além de fornecerem *backup* e autenticação de dois fatores (ETHHUB, 2019).

2.2.4 Linguagem de Programação *Solidity*

Desenvolvida pela *Ethereum Foundation*, é a linguagem pelo qual são escritos os *smart contracts*. Ao invés dos contratos serem escritos diretamente no *bytecode* da EVM, são escritos em *Solidity* que é uma linguagem de alto nível (semelhante ao *JavaScript*), tornando o desenvolvimento mais simples e legível (GREGORY, 2019)

Segundo Grishchenko et al. (2018), *Solidity* é uma linguagem de programação chamada de "orientada a contratos", que usa o conceito de classe de linguagens orientadas a objetos para a representação de contratos. Na Figura 6 é possível observar um exemplo simples de *smart contract* escrito em *solidity* para salvar o nome de um usuário na *blockchain Ethereum*.

Figura 6 – Exemplo de *smart contract* escrito na linguagem *Solidity*

```
1  pragma solidity 0.5.11;
2
3  contract createUser {
4      Person[] public people;
5
6      uint256 public peopleCount;
7
8      struct Person {
9          string _firstName;
10         string _lastName;
11     }
12
13     function addPerson(string memory _firstName, string memory _lastName) public {
14         people.push(Person(_firstName, _lastName));
15         peopleCount += 1;
16     }
17 }
```

Fonte: Autor

Semelhante às classes na programação orientada a objetos, os contratos especificam campos e métodos para instâncias de contrato. Os campos podem ser vistos como armazenamento persistente de um contrato (instância) e os métodos do contrato podem, por padrão, serem chamados por qualquer transação interna ou externa (GRISHCHENKO et al., 2018).

2.3 *Smart Contracts*

Nicholas Szabo introduziu esse conceito em 1994 e definiu que um *smart contract* é como um protocolo de transação computadorizado que executa os termos de um contrato automaticamente (SZABO, 1997).

Segundo Roque (2003), um contrato é o acordo de duas ou mais pessoas para estabelecer, regular ou terminar um vínculo jurídico. Os mecanismos de governança de confiança na competência e contrato formal fornecem condições de facilitação única para o aprendizado interorganizacional. Deste modo, o *smart sontract* possui o mesmo tipo de acordo para cumprir ou não, mas elimina a necessidade de um terceiro para validar entre as partes acordadas.

> *Smart Contracts* são contratos digitais escritos em uma linguagem de programação executados em um computador. As regras de negócio são definidas no código. Assim, o *smart contract* é capaz de obter informações e processá-las de acordo com regras predefinidas. (CAMARGO, 2017)

Os *smart contracts* reduzem custos computacionais impostos por terceiros, pois utilizam protocolos e interfaces de usuário para facilitar todas as etapas do processo de contratação. As etapas contratuais de pesquisa, negociação, compromisso, desempenho e adjudicação constituem o domínio dos *smart sontracts*, isso fornece novas maneiras de formalizar e proteger as relações digitais que são muito mais funcionais do que a forma tradicional baseada em papéis (SZABO, 1997).

2.3.1 *Smart Contracts* da *Ethereum*

No contexto da *Ethereum* o termo "*smart contract*" é usado para referir-se a programas de computador imutáveis, que são executados deterministicamente no contexto de uma máquina virtual *Ethereum* como parte do protocolo de rede da mesma. Os *smart contracts* podem ser usados para criar uma ampla variedade de aplicativos descentralizados (DApps), que podem ser jogos, colecionáveis digitais, sistemas de votação online, produtos financeiros e muitos outros (ANTONOPOULOS; WOOD, 2018).

Os *smart contracts* são geralmente escritos em uma linguagem de programação de alto nível, como por exemplo a linguagem *Solidity*. Porém, para executar eles devem ser compilados no *bytecode* de baixo nível que é executado na EVM. Uma vez compilados, eles são implantados na plataforma *Ethereum*, onde cada contrato é identificado por um endereço *Ethereum*, derivado da transação de criação de contrato em função da conta de origem. O endereço *Ethereum* de um contrato pode ser usado em uma transação como destinatário, enviando fundos ao contrato ou chamando uma das funções do contrato (ANTONOPOULOS; WOOD, 2018).

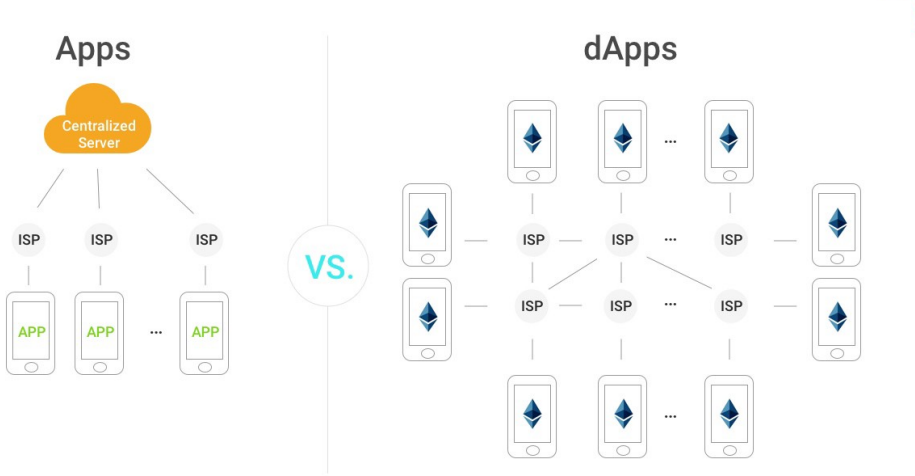
Todos os *smart contracts* na *Ethereum* são executados, devido a uma transação iniciada a partir de um EOA. Um contrato pode chamar outro contrato que pode chamar outro contrato, e assim por diante, mas o primeiro contrato dessa cadeia de execução sempre será chamado por uma transação de um EOA (ANTONOPOULOS; WOOD, 2018).

É importante ressaltar que o código de um contrato não pode ser alterado. No entanto, um contrato pode ser excluído, removendo seu código e estado do endereço de armazenamento da *blockchain*, deixando uma conta em branco. Todas as transações enviadas para o endereço dessa conta após a exclusão do contrato resultam em nenhuma execução do código. Esta exclusão não remove o histórico de transações do contrato, uma vez que a própria *blockchain* é imutável (ANTONOPOULOS; WOOD, 2018).

2.4 Aplicações Descentralizadas

Aplicações descentralizadas, ou DApps, são aplicativos distribuídos por vários servidores, ao invés de ficarem em um único servidor centralizado. Ajudam a gerenciar grandes volumes de dados e a demanda de tráfego, evitando o tempo de inatividade do aplicativo em projetos críticos que não podem ser perdidos. Nesse mecanismo, os dados de um aplicativo têm várias cópias em vários nós da rede (CHANDRAYAN, 2018). Na Figura 7 é exemplificada a diferença entre aplicações centralizadas e descentralizadas.

Figura 7 – Visão da diferença entre Aplicações Centralizadas e Descentralizadas



Fonte: Rupareliya (2018)

> Aplicações Descentralizadas são aplicativos de internet distribuídos, que rodam em rede *P2P* descentralizada e cuja base de código é aberta publicamente. O *DApp* não é de propriedade de nenhum nó único na rede. Pares na rede podem ser qualquer computador que esteja conectado à internet. (CHANDRAYAN, 2018)

Segundo Antonopoulos e Wood (2018), existem algumas vantagens em desenvolver um DApp que uma arquitetura centralizada não pode fornercer, que são:

- **Resiliência:** Pelo fato da lógica de negócio ser controlada por *smart contracts*, o *back-end* do DApp é totalmente distribuído e gerenciado em uma plataforma *blockchain*. E diferentemente de um *software* implantando em um servidor centralizado, um DApp jamais terá tempo de inatividade e continuará disponível desde que a plataforma *blockchain* continue funcionando;
- **Transparência:** Pela própria natureza da *blockchain*, é permitido que todos inspecionem o código de um DApp e tenha-se a certeza sobre sua função. Qualquer nova alteração no seu código será armazenada para sempre em um novo bloco na *blockchain*, juntamente com as informações do autor da modificação;

- **Resistência à censura:** O usuário sempre poderá interagir com um DApp sem interferência de qualquer controle centralizado, desde que possua acesso a um nó da rede *Ethereum*.

2.4.1 Aplicações Descentralizadas da *Ethereum*

Em uma aplicação descentralizada, os *smart contracts* são usados para armazenar a lógica de negócio (código de programa) e o estado relacionado do aplicativo. Podemos pensar em um *smart contract* substituindo um componente do lado do servidor (ANTONOPOULOS; WOOD, 2018).

Os *smart contracts* da *Ethereum* permitem construirmos arquiteturas nas quais uma rede de *smart contracts* chame e passe dados entre si, lendo e escrevendo suas próprias variáveis de estado, com sua complexidade restrita apenas pelo limite de bloqueio de *Gas* do bloco. Depois de implantar o *smart contract*, sua lógica de negócio pode ser usada por muitos outros desenvolvedores no futuro (ANTONOPOULOS; WOOD, 2018).

2.4.2 Validade Jurídica de Contratos Digitais

Segundo Cimatti (2018), a edição da MP 2.200-2/01, entre outras providências, visou a garantir a autenticidade, integridade e a validade jurídica de documentos em forma eletrônica. Sendo assim, contratos digitais tem plena validade jurídica desde que possam cumprir as funções de qualquer contrato escrito, que são:

- Declarar as vontade das partes em realizar o negócio;
- Exprimir o exato conteúdo do negócio;
- Servir como meio probatório.

Para isso, a plataforma deve certificar-se que a tecnologia utilizada para prestar o serviço é capaz de garantir a autenticidade e integridade do contrato digital, ou seja, é necessário que os contratos digitais possibilitem a inequívoca identificação das partes signatárias e a inviolabilidade do seu conteúdo (CIMATTI, 2018).

3 FERRAMENTAS DE DESENVOLVIMENTO

Neste Capítulo são descritas as ferramentas utilizadas no desenvolvimento do presente trabalho.

3.1 Biblioteca *ReactJS*

ReactJS é uma biblioteca *JavaScript* para criação de interfaces desenvolvida pelo Facebook em 2011, que oferece uma nova abordagem para a construção de interfaces de usuário baseada em componentes. É possível utilizar no navegador, no servidor e em dispositivos móveis (ANTONIO, 2015).

Sua abordagem é a renderização reativa da interface do usuário. Essa abordagem separa o estado (todos os dados internos que definem o aplicativo em um determinado ponto no tempo) da interface que é apresentada ao usuário. Com o *ReactJS*, é possível declarar como o estado é representado com os elementos visuais do DOM, a partir de então, o DOM é atualizado automaticamente para refletir as mudanças de estado (ANTONIO, 2015).

O "cerne" do *ReactJS* é tornar possível a redução da complexidade da criação e manutenção de interfaces. Isso envolve o conceito de dividir a interface do usuário em componentes, blocos de construção autocontidos específicos, que são fáceis de reutilizar, estender e manter (ASHWINI, 2017).

Porém, se a intenção é desenvolver um projeto grande, como por exemplo um SPA, somente com *ReactJS* não será possível. Para isso, o *ReactJS* mantém um ecossistema muito rico de bibliotecas e ferramentas para auxiliar no desenvolvimento de projetos (ANTONIO, 2015).

3.1.1 Programação Reativa

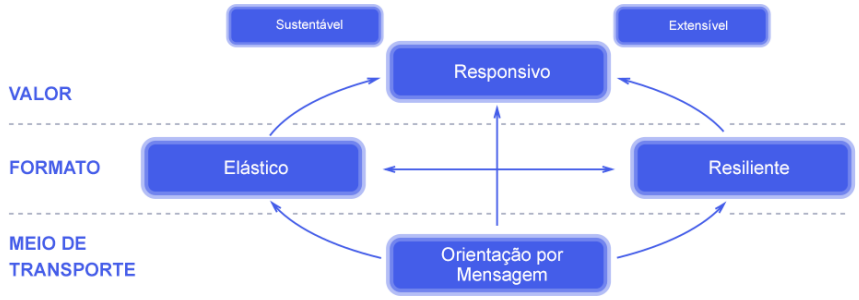
O principal motivo para a adoção desse paradigma de programação é o contínuo crescimento da internet nos mais diversos aspectos, como por exemplo, maior quantidade de usuários, maior quantidade de *requests* a servidores, maior volume de dados e maior exigência de performance e tempo de resposta (JONAS et al., 2014).

Aplicações construídas com programação reativa são mais flexíveis, mais fáceis de serem acopladas e escalonadas. Isso torna o seu desenvolvimento mais fácil e passível de mudanças. São significativamente mais tolerantes à falhas. As aplicações reativas são altamente responsivas, oferecendo aos usuários um *feedback* interativo e eficaz (JONAS et al., 2014).

Segundo JONAS et al. (2014), a programação reativa possui quatro pilares (Figura 8), que são:

- **Elástico (*Elastic*):** Reagem a demanda de carga, podendo aumentar ou diminuir os recursos alocados para atender estas demandas. Também podem fazer o uso de múltiplos *cores* e de múltiplos servidores;
- **Resiliente (*Resilient*):** Reagem e se recuperam de falhas de *software*, *hardware* e de conectividade, mantendo-se responsivo. Estas falhas estão contidas em cada componente, isolando os componentes uns dos outros;
- **Orientação por Mensagem (*Message Driven*):** Em vez de compor aplicações por múltiplas *threads* síncronas, sistemas são compostos de gerenciadores de eventos assíncronos e não bloqueantes;
- **Responsivo (*Responsive*):** Oferece interações ricas ao usuário. Com a interface comportando-se de maneira oportuna para o mesmo.

Figura 8 – Ciclo dos Quatro Pilares da Programação Reativa



Fonte: Jonas et al. (2014)

3.1.2 Diferença entre *ReactJS* e *React Native*

Segundo Ashwini (2017), *ReactJS* é uma biblioteca *JavaScript* responsável por criar uma hierarquia de componentes da interface do usuário, ou seja, é responsável somente pela renderização dos componentes da interface do usuário utilizando da melhor forma o DOM. Fornece suporte para *front-end* e servidor.

React Native é uma estrutura para criar aplicativos nativos usando *JavaScript*, ou seja, não é preciso desenvolver o mesmo aplicativo em diferentes linguagens de programação pra diferentes plataformas, pois ele permite a reutilização da camada lógica para todas as plataformas. Possui módulos e componentes nativos que melhoram seu desempenho, além de usar a biblioteca *ReactJS* para a renderização da interface (ASHWINI, 2017).

3.2 *JavaScript* ou *ECMAScript*

JavaScript como é popularmente conhecida, é uma linguagem de programção criada em 1995 pelo programador Brendan Eich, porém em 1995 Brendan e os principais desenvolvedores da linguagem associaram-se a fundação ECMA. Como o nome *JavaScript* já havia sido patenteado pela então na época Sun Microsystems, optou-se por definir o nome oficial da linguagem de *ECMAScript* no ano de 1997 (MALAVASI, 2017).

Segundo Malavasi (2017), como o nome ficou popular por toda a comunidade de desenvolvedores, a linguagem é chamada de *JavaScript* até os dias de hoje, ficando *ECMAScript* para referenciar a versão da mesma. *ECMAScript* passou por diversas versões, as principais são listadas abaixo:

- ***ECMAScript* 1:** representa a primeira versão da linguagem lançada em 1997, com os padrões e normativas definidas pela ECMA;
- ***ECMAScript* 2:** criada em 1997 para se adequar à ISO/IEC 16262;
- ***ECMAScript* 3:** criada em 1999, possui melhorias como o laço de repetição *do-while*, tratamento de exceções, dentro outros recursos;
- ***ECMAScript* 4:** lançada em 2008, teve seu principal desenvolvimento baseado em ML[1];
- ***ECMAScript* 5:** concluída em 2012, foi desenvolvido recursos importantíssimos como o suporte a JSON, métodos mais avançados de manipulação de *arrays*, *getters* e *setters*, dentre outros;
- ***ECMAScript* 6:** lançada em 2015, possui recursos avançados à linguagem como *reflection*, *collections*, *binary data*, *arrow functions*, dentre outros;
- ***ECMAScript* 7:** lançada em 2016, inclui recursos como bloco de escopo de variáveis e funções, *await* e *async* (palavras-chave para programação assíncrona);
- ***ECMAScript* 8:** lançada em 2017, possui como principal recurso novo a integração sintática com promessas assíncronas;
- ***ECMAScript* 9:** lançada em 2018, inclui principalmente a nova propriedade assíncrona *Promise.prototype.finally* entre outras melhorias na mesma;
- ***ECMAScript* 10:** lançada em 2019, inclui novos recursos como *Array.prototype.flat*, *Array.prototype.flatMap* e melhoria nas funções *Array.sort* e *Object.fromEntries*.

JavaScript é uma linguagem de programação interpretada. Originalmente implementada como parte dos navegadores *web* para que *scripts* pudessem ser executados do lado do cliente e interagissem com o usuário sem a necessidade de passar pelo servidor. Assim controlando o navegador, realizando comunicação assíncrona e alterando o conteúdo do documento exibido (MOZILLA DEVELOPER NETWORK, 2019). Na Figura 9 é possível ver um exemplo de código escrito em *JavaScript* (*ECMAScript* 8).

[1] ML é uma linguagem de programação funcional muito utilizada em ambientes de pesquisa acadêmica.

Figura 9 – Exemplo de código *JavaScript* e sua respectiva saída

```javascript
const toys = {
    'Goku': '20',
    'Buzz Lightyear': '15',
    'Mimikyu': '5',
    'WarGreymon': '0'
}

Object.entries(toys).map(([name, count]) => {
    console.log(`${name.padEnd(20, ' -')} Count: ${count.padStart(3, '0')}`)
});

// Prints ...
// Goku - - - - - - - - Count: 020
// Buzz Lightyear - - - Count: 015
// Mimikyu - - - - - -  Count: 005
// WarGreymon - - - - - Count: 000
```

Fonte: Autor

Atualmente, é a principal linguagem para programação *client-side* em navegadores *web*. Começa também a ser bastante utilizada do lado do servidor através de ambientes como o *NodeJS*. Foi concebida para ser uma linguagem *script* com orientação a objetos baseada em protótipos, tipagem fraca e dinâmica, funções de primeira classe e suporte à programação funcional (MOZILLA DEVELOPER NETWORK, 2019).

3.3 *NodeJS*

Criado em 2009 pelo programador Ryan Dahl, o *NodeJS* ou simplesmente *Node*, combina o interpretador de *JavaScript* V8 da Google e um laço de eventos para rodar *JavaScript* no servidor. Aproveitando o poder e a simplicidade da linguaguem, tornou tarefas difíceis de escrever, como por exemplo, aplicações assíncronas, em tarefas fáceis (NODE BR, 2016b).

O *Node* é um servidor, que diferentemente de servidores como *Apache* e *Toomcat* que são *ready-to-install*, ele possui o conceito de módulos que podem ser adicionados no seu núcleo, ajudando a resolver os problemas de escopo e diminuindo a complexidade das aplicações. Outra grande vantagem é que a maioria dos seus métodos são todos assíncronos, assim nenhum programa é bloqueado (NODE BR, 2016b).

3.3.1 Motor *JavaScript* V8

O V8 como é comumente chamado, é um interpretador de *JavaScript* escrito na linguagem C++ estremante eficiente e rápido, criado pela Google e que vinha sendo usado somente em seu navegador. Foi liberado para ser incorporado em qualquer projeto de código livre (NODE BR, 2016b).

3.3.2 *Node Package Manager*

O *Node Package Manager*, ou "NPM", é um utilitário de linha de comando que interage com o seu respositório online, que conta com milhares de publicações de código livre (pacotes). A partir dele é possível baixar e instalar automaticamente pacotes, para serem utilizados em projetos que utilizam *Node* (NODE BR, 2016a).

3.4 CSS

O CSS ou *Cascading Style Sheets* (Folha de Estilo em Cascada), é uma linguagem de estilo utilizada para definir a aparência em páginas da *web* que adotam para o seu desenvolvimento linguagens de marcação (como XML, HTML e XHTML). O CSS define como serão exibidos os elementos contidos no código de uma página como mostrado na Figura 10, sua maior vantagem é efetuar a separação entre o formato e o conteúdo de um documento (MATERA, 2012).

Figura 10 – Exemplo de código CSS

```
1   html, body {
2       background: #343b47;
3       font: 400 16px 'Roboto Slab', serif;
4       color: #d4d4d4;
5       line-height: 1.4;
6       letter-spacing: 1.2px;
7       -moz-osx-font-smoothing: grayscale;
8       -webkit-text-stroke: 0;
9       height: 100%;
10  }
11
12  div > p {
13      font-size: 18px;
14      color: #000;
15  }
16
17  .container {
18      max-width: 1200px;
19      width: 100%;
20      margin: 0 auto;
21  }
22
23  nav h1 {
24      font-size: 30px;
25      position: relative;
26      top: 7px;
27      margin-top: 0;
28  }
```

Fonte: Autor

Segundo o artigo da Matera (2012), entre os motivos que adotaram o CSS como a principal linguagem de estilo, foi a internet tonando-se cada vez mais rápida, sendo possível adotar *layouts* mais complexos e modernos. E o crescimento no uso de aplicações *mobile*, em que as páginas precisam ser leves e o conteúdo ser apresentado de forma correta em diferentes dispositivos, o que não seria possível utilizando-se da *tag* de HTML *table*.

3.5 *Less*

Segundo Sellier (2017), *Less* é um transpilador[2] *open-source* de CSS, criado pelo programador Alexis Sellier em 2009. Com ele é possível criar varáveis, aninhamento e escopo de código e funções. Os grande diferenciais do *Less* para outros transpiladores CSS, é a compilação em tempo real pelo navegador por meio do *LessJS* e que pode ser executado tanto em *client-side* quanto em *server-side*. Na Figura 11, há um exemplo de código *Less* transpilado para CSS.

Figura 11 – Exemplo de código *Less* (A) transpilado para CSS (B)

```less
// Suport Success
@c-sucess: #739E41;

// Suport Success Light
@c-error: #FF6347;

.container {

    .toastblock {
        align-items: center;
        display: flex;
        justify-content: space-between;

        .-toastmsg {
            font-size: 16px;
            line-height: 1.4;
            margin-right: 10px;
            color: @c-sucess;
        }
    }

    .-texred {
        position: relative;

        .-toastmsg {
            color: @c-error;
        }

        button {
            color: @c-error;
        }
    }
}
```

A

```css
.container .toastblock {
  align-items: center;
  display: flex;
  justify-content: space-between;
}
.container .toastblock .-toastmsg {
  font-size: 16px;
  line-height: 1.4;
  margin-right: 10px;
  color: #739E41;
}
.container .-texred {
  position: relative;
}
.container .-texred .-toastmsg {
  color: #FF6347;
}
.container .-texred button {
  color: #FF6347;
}
```

B

Fonte: Autor

[2]Processo semelhante a compilação, a diferença é que o alvo do compilador é um código de mais baixo nível, enquanto que o transpilador tem como alvo um código fonte de uma linguagem diferente, ou a mesma escrita de outra forma.

3.6 *Bootstrap*

Originalmente desenvolvido na empresa Twitter em 2011 pelos programadores Mark Otto e Jacó Thornton, pensado como um instrumento para facilitar a consistência de ferramentas internas. O *Bootstrap* é um *framework front-end open-source*, voltado principalmente para o desenvolvimento de projetos *mobile*. Todos os componentes oferecidos são responsivos, ou seja, eles redimensionam-se, diminuem ou ficam maiores dependendo do tamanho da resolução de tela do dispositivo em que a página é renderizada (BOOTSTRAP, 2018).

Uma das principais características do *Bootstrap* é seu sistema de grade, que pode escalar até 12 colunas de acordo com o tamanho da tela. Atualmente ele se encontra na versão 4, o que trouxe muitas melhorias em relação a versão 3, como a simplificação dos componentes de navegação (BOOTSTRAP, 2018).

3.7 *Framework Truffle*

Criado pelo programador Tim Coulter, o *Truffle* é um ambiente de desenvolvimento e teste em *Solidity*, que conta com um *pipeline* de ativos para a *Blockchain Ethereum*, com o objetivo de facilitar o desenvolvimento na EVM. Com ele é possível criar e testar DApps, compilar e implantar *smart contracts* e gerenciar a rede (TRUFFLE SUITE, 2017).

O *Truffle* possui outros dois componentes principais que auxiliam no desenvolvimento dos DApps, que são:

- ***Ganache*:** é uma *blockchain* pessoal que permite ao desenvolvedor criar *smart contracts*, DApps e testá-los com uma ferramenta de linha de comando disponível para Windows, Mac e Linux (GOLDEN, 2017);
- ***Drizzle*:** é uma biblioteca de desenvolvimento *front-end* personalizada, baseada na biblioteca *JavaScript Redux*, capaz de sincronizar automaticamente dados de contrato, dados de transação e outros dados (GOLDEN, 2017).

4 DESENVOLVIMENTO DO SISTEMA

Neste Capítulo é apresentada uma análise de aplicações existentes na área de contratos e assinaturas digitais, bem como a proposta do presente trabalho e os recursos e diagramas utilizados no desenvolvimento.

4.1 Análise de Aplicações Existentes

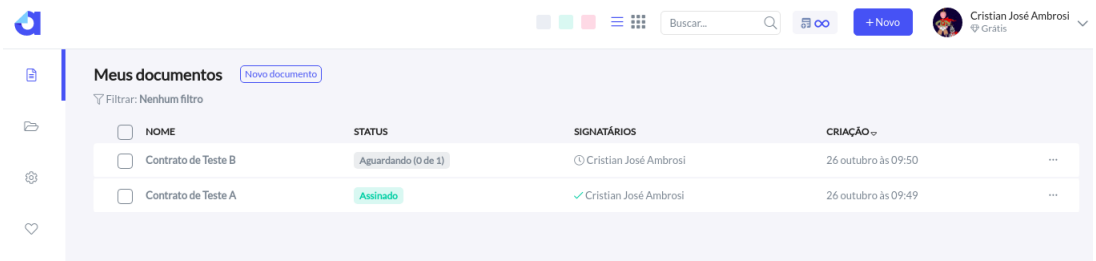
Existem algumas aplicações *web* no mercado atualmente, que disponibilizam o serviço de validação de contratos por meio de assinaturas digitais. São aplicações eficazes ao que propõe-se, mas todas possuem, por assim dizer, a limitação de serem aplicações centralizadas. Outra desvantagem, é o preço alto para utilizar seus planos corporativos ou que disponibilizam maior quantia de recursos ao usuário. A seguir são apresentados algumas aplicações que atuam no mercado de validação de contratos.

4.1.1 Autentique

Autentique[1] é uma plataforma *web* que dispõem o serviço de assinatura digital para documentos contratuais. Nele, é possível criar ou fazer *upload* de um documento e adicionar as pessoas que irão assinar o contrato. O contrato ficará no aguardo até todos assinarem, assim que todos assinarem é enviado um alerta que o contrato está pronto.

Na Figura 12, é exibido o *dashboard* da plataforma com dois contratos, um eperando a assinatura do signatário e o outro devidamente assinado e pronto.

Figura 12 – Interface da plataforma Autentique



Fonte: Autor

O Autentique disponibiliza dois planos de assinaturas, grátis e corporativo (mensal). Algumas de suas principais funcionalidades são:

- Criação de modelos de contratos;
- *Upload* de contratos;

[1] https://www.autentique.com.br

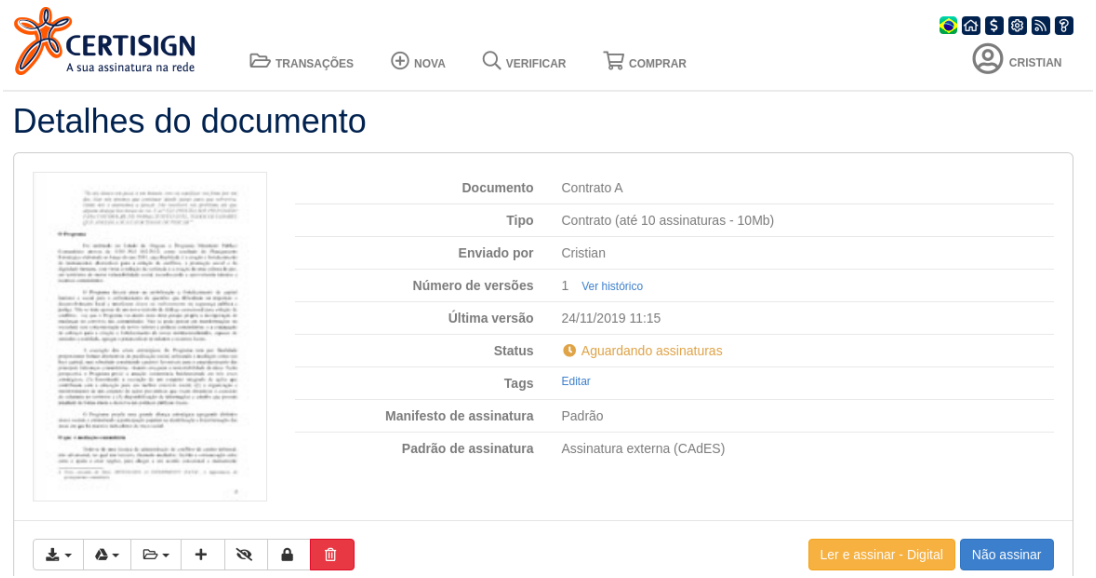
- Indicar signatários e o seu papel, por meio do cadastro de e-Mail do mesmo;
- Selecionar o local das assinaturas no documento.

4.1.2 CERTISIGN - Portal de Assinaturas

Especialista em identificação digital, a CERTISIGN[2] disponibiliza o serviço de assinatura digital para qualquer tipo de documento. Armazena de forma encriptada os documentos por até 5 anos.

A CERTISIGN disponibiliza quatro planos anuais de assinatura do seu serviço, individual, grupo de trabalho, pequena empresa e empresa digital. Na Figura 13, é exibido o *dashboard* da plataforma com informações de um contrato, aguardando a assinatura do signatário.

Figura 13 – Interface da plataforma CERTISIGN



Fonte: Autor

Uma das principais funcionalidades da plataforma é o Verificador de Assinaturas, onde é possível validar a integridade da assinatura e do documento. Também é possível enviar por e-Mail o contrato, para que os signatários adicionados possam assinar.

4.2 Proposta do Novo Sistema

Após a análise das plataformas *web* existentes no mercado, chegou-se a conclusão da necessidade do desenvolvimento de um novo *software*, a fim de apresentar um sistema igualmente seguro e de fácil acesso, com preços acessíveis e sem burocracias.

Semelhante as plataformas cidatadas a cima, o presente trabalho pretende desenvolver funcionalidades como, cadastro de um documento PDF, vinculção de signatários bem como a

[2] https://www.certisign.com.br

assinatura no contrato. O diferencial é que se trata de uma aplicação descentralizada, ficando a cargo da rede da *blockchain Ethereum* tornar o contrato um documento válido juridicamente.

4.2.1 Metas e Objetivos do Sistema

O desenvolvimento de uma plataforma *web*, onde o usuário possa cadastrar de forma simples contratos, que serão validados pela rede descentralizada da *blockchain Ethereum*. Também, dando a opção de vincular outros usuários para assinar o contrato, usando a assinatura digital da *Ethereum* (chave pública da *wallet*), que garante a autenticidade da mesma.

4.3 Recursos Utilizados para o Desenvolvimento

Tendo em vista o desenvolvimento da aplicação, foram utilizadas ferramentas e tecnologias que suprissem as necessidades da mesma. Ferramentas que oferecem soluções robustas para trabalhar nos diversos segmentos do sistema desenvolvido.

- ***Visual Studio Code*:** é um editor de código *open-source* leve e altamente extensível, desenvolvido pela Microsoft. Possui integrações com sistemas de versionamento de código e ferramentas de *debug*, ao mesmo tempo que milhares de extensões desenvolvidas pela comunidade são disponibilizadas gratuitamente através de um *marketplace* mantido pelos colaboradores do mesmo. O *Visual Studio Code* possui ampla capacidade para desenvolvimento em diversas linguagens de programação, como *Java*, *Python*, PHP, *Javascript*, entre outras;
- ***Git*:** é um sistema *open-source* de controle de versão desenvolvido por Linus Torvalds. Com ele é possível criar todo histórico de alterações no código de um projeto e facilmente voltar para um ponto específico do código, para saber como estava naquele momento. Também ajuda a controlar o fluxo de novas funcionalidades entre muitos desenvolvedores no mesmo projeto;
- ***GitHub*:** é um serviço online de hospedagem de repositórios *Git*, possui diversas integrações com serviços que auxiliam no *deploy* da aplicação através de integração contínua;
- ***GulpJS*:** é uma ferramenta para automação de tarefas desenvolvida em *JavaScript*. Tarefas como minificar, otimizar e compilar arquivos;
- ***Remix*:** é uma IDE *open-source* da *Ethereum*, que ajuda a escrever *smart contracts* em *Solidity* diretamente no navegador. Com ela é possível testar, depurar e implantar *smart contracts* na *blockchain*.

4.4 Diagramas UML

Nesta Seção, é ressaltado como a interface da aplicação foi estruturada através do diagrama de Atividades da modelagem UML, utilizando a ferramenta *Astah Community* para a elaboração do mesmo.

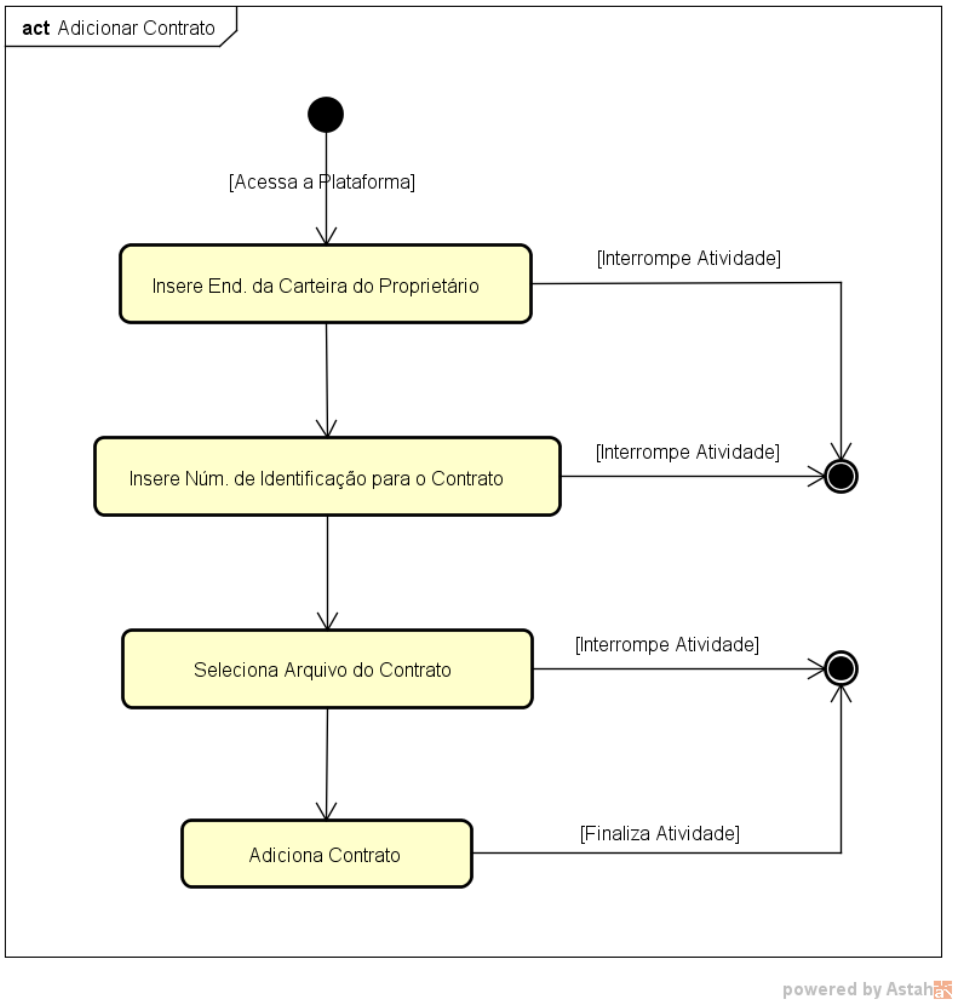
4.4.1 Diagrama de Atividades

Segundo Guedes (2011), o Diagrama de atividades é focado em descrever os passos a serem percorridos para a conclusão de uma atividade específica. Sendo considerado o diagrama com maior ênfase ao nível de algoritmo da UML e possívelmente um dos mais detalhistas.

4.4.1.1 Adicionar Contrato

Na Figura 14, é apresentado o diagrama de atividades para adicionar um contrato na plataforma.

Figura 14 – Diagrama de Atividades - Adicionar Contrato

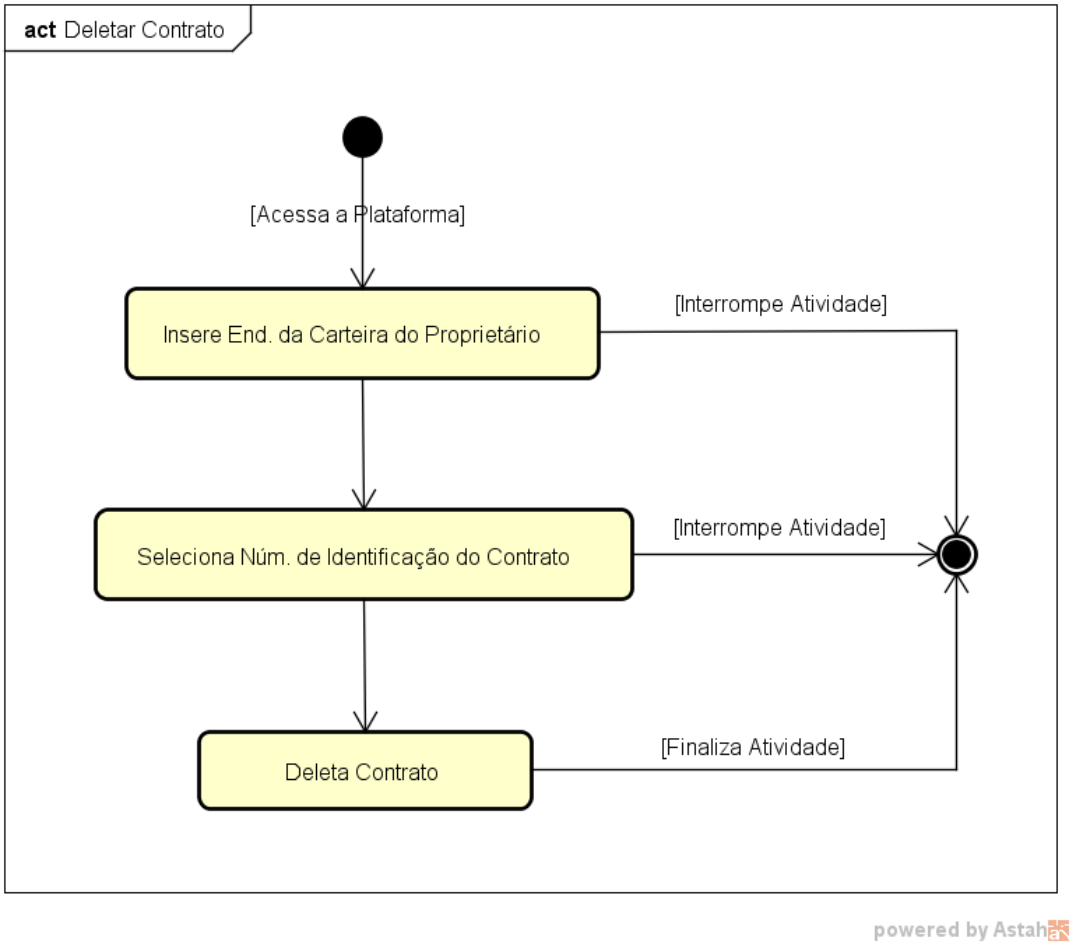


Fonte: Autor

4.4.1.2 Deletar Contrato

A Figura 15, apresenta o diagrama de atividades para deletar um contrato cadastrado na plataforma.

Figura 15 – Diagrama de Atividades - Deletar Contrato

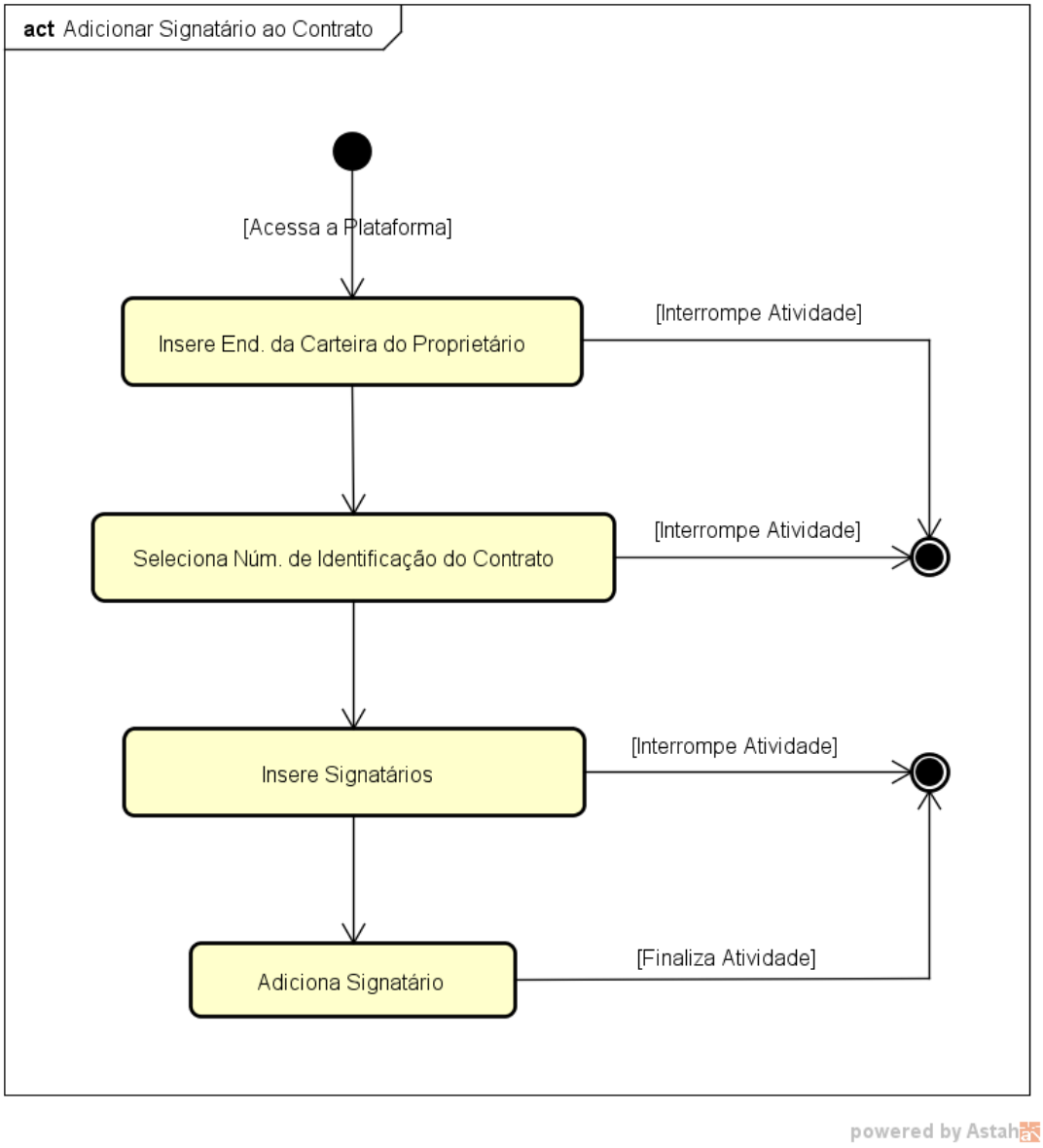


Fonte: Autor

4.4.1.3 Adicionar Signatário

Na Figura 16, é apresentado o diagrama de atividades para adicionar até dois signatários a um contrato cadastrado.

Figura 16 – Diagrama de Atividades - Adicionar Signatário

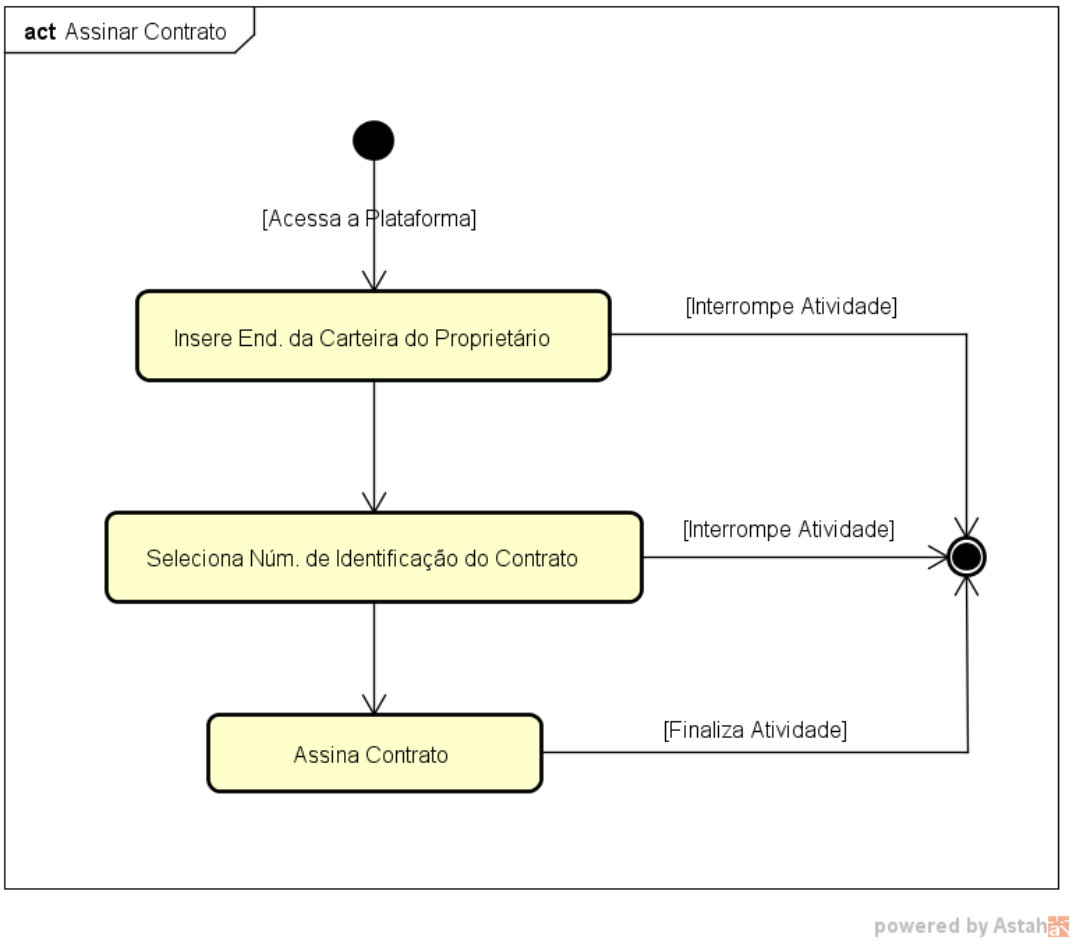


Fonte: Autor

4.4.1.4 Assinar Contrato

A Figura 17, apresentada o diagrama de atividades para assinar um contrato.

Figura 17 – Diagrama de Atividades - Assinar Contrato

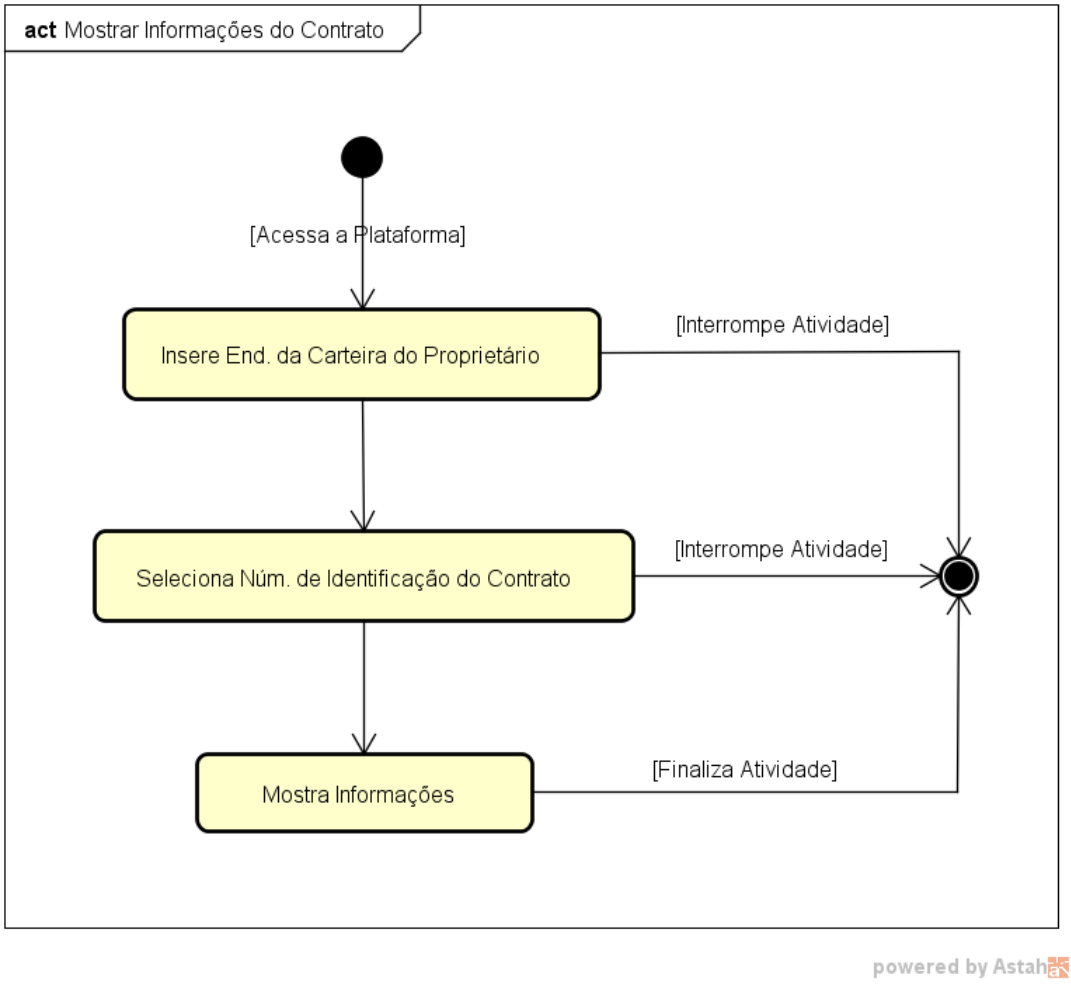


Fonte: Autor

4.4.1.5 Mostrar Informações do Contrato

Na Figura 18, é apresentado o diagrama de atividades para mostrar as informações de um contrato cadastrado na *blockchain*.

Figura 18 – Diagrama de Atividades - Mostrar Informações do Contrato



Fonte: Autor

4.5 Serviço de *Smart Contracts* da *Ethereum*

Para o desenvolvimento deste serviço, foi necessário a instalação da EVM no ambiente local. Tendo a EVM instalada localmente, não foi necessário esperar o tempo de fechamento de bloco da *blockchain* para testar as implantações.

Utilizando o *framework Truffle*, foi possível abstrair algumas camadas de conexão entre a lógica dos *smart contracts* e a EVM, onde os contratos são implantados. A seguir serão apresentados peguenos fragmentos de código, responsáveis pela lógica e implantação dos contratos.

4.5.1 Função de Adicionar Contrato

A função é responsável por adicionar o contrato na *blockchain Ethereum*, como apresentado na Figura 19. Ela espera um número de identificação e o *hash* gerado a partir do arquivo submetido.

Figura 19 – Fragmento de Código - Adicionar Contrato

```
32  function addContract(uint _contractNumber, bytes32 _hashFile) onlyOwner {
33      require(_contractNumber > 0);
34      require(_hashFile > 0);
35
36      require(contracts[_contractNumber].contractState == ContractState.None);
37
38      ContractData memory contractData;
39      contractData.hashFile = _hashFile;
40      contractData.contractState = ContractState.Created;
41
42      contracts[_contractNumber] = contractData;
43      contractNumbers.push(_contractNumber);
44  }
```

Fonte: Autor

4.5.2 Função de Adicionar Signatários

Como apresentado na Figura 20, a função é responsável por vincular signatários a um contrato já cadastrado. Ela espera o número de identificação do contrato e o endereço público da carteira digital do usuário.

Figura 20 – Fragmento de Código - Adicionar Signatários

```
46  function addSignees(uint _contractNumber, address[] _accountAddresses) onlyOwner {
47      require(_contractNumber > 0);
48      require(_accountAddresses.length > 0);
49      require(contracts[_contractNumber].contractState == ContractState.Created);
50      require(contracts[_contractNumber].signeesIndex.length == 0);
51
52      for (uint i = 0; i < _accountAddresses.length; i++) {
53          contracts[_contractNumber].signees[_accountAddresses[i]] = SignState.Unsigned;
54          contracts[_contractNumber].signeesIndex.push(_accountAddresses[i]);
55      }
56
57      contracts[_contractNumber].contractState = ContractState.Activated;
58  }
```

Fonte: Autor

4.5.3 Função de Assinar Contrato

Como apresentado na Figura 21, a função é responsável pela assinatura do signatário vinculado ao contrato, onde somente um usuário vinculado é possível assinar o mesmo. Ela espera o número de identificação do contrato.

Figura 21 – Fragmento de Código - Assinar Contrato

```
109     function signContract(uint _contractNumber) private {
110         require(msg.sender ≠ owner);
111         require(_contractNumber > 0);
112         require(contracts[_contractNumber].contractState == ContractState.Activated);
113 
114         if (contracts[_contractNumber].signees[msg.sender] == SignState.Unsigned) {
115             contracts[_contractNumber].signees[msg.sender] = SignState.Signed;
116             setContractCompletedIfAllSignaturesAreSet(_contractNumber);
117         }
118     }
```

Fonte: Autor

4.5.4 Função de Deletar Contrato

A função da Figura 22 deleta os contratos cadastrados, desde que nenhum signatário tenha sido vinculado ao contrato. Ela espera o número de identificação do contrato.

Figura 22 – Fragmento de Código - Deletar Contrato

```
83      function deleteContract(uint _contractNumber) private onlyOwner {
84          require(_contractNumber > 0);
85          if (contracts[_contractNumber].contractState == ContractState.None ||
86              contracts[_contractNumber].contractState == ContractState.Created) {
87              delete contracts[_contractNumber];
88              updateContractNumbers(_contractNumber);
89          }
90      }
```

Fonte: Autor

4.5.5 Função para Setar Contrato como Finalizado

A função é responsável por setar os contratos de uma determinada carteira como finalizada, assim como mostra a Figura 23, desde que todos os signatários vinculados tenham assinado com seu endereço de chave pública.

Figura 23 – Fragmento de Código - Seta Contrato como Finalizado

```
92  function setContractCompletedIfAllSignaturesAreSet(uint _contractNumber) private {
93      uint signedCounter = 0;
94      uint numberOfSignees = contracts[_contractNumber].signeesIndex.length;
95
96      for (uint i = 0; i < numberOfSignees; i++){
97          address signeeAddress = contracts[_contractNumber].signeesIndex[i];
98
99          if (contracts[_contractNumber].signees[signeeAddress] == SignState.Signed) {
100             signedCounter++;
101         }
102     }
103
104     if (numberOfSignees == signedCounter) {
105         contracts[_contractNumber].contractState = ContractState.Completed;
106     }
107 }
```

Fonte: Autor

5 DESCRIÇÃO DO SISTEMA

Neste Capítulo, são apresentadas as funcionalidades do sistema desenvolvido, juntamente com imagens de suas interfaces.

No decorrer do desenvolvimento o nome escolhido para a plataforma foi *Smart Sign*, que traduzido para o português significa “Assinatura Inteligente”, remetendo a sua principal funcionalidade: criar contratos e validá-los de forma segura e automática por meio de *smart contracts* da *blockchain*.

5.1 Tela para Adicionar Contrato

A primeira tela do *dashboard* da plataforma é a de cadastro do contrato, conforme apresenta a Figura 24. O usuário tem acesso ao formulário, onde poderá informar um número identificador para o contrato, informar o endereço público da carteira digital que prentende vinculá-lo e enviar um documento em formato de PDF, que será calculado o *hash* do mesmo para ser salvo na *blockchain*.

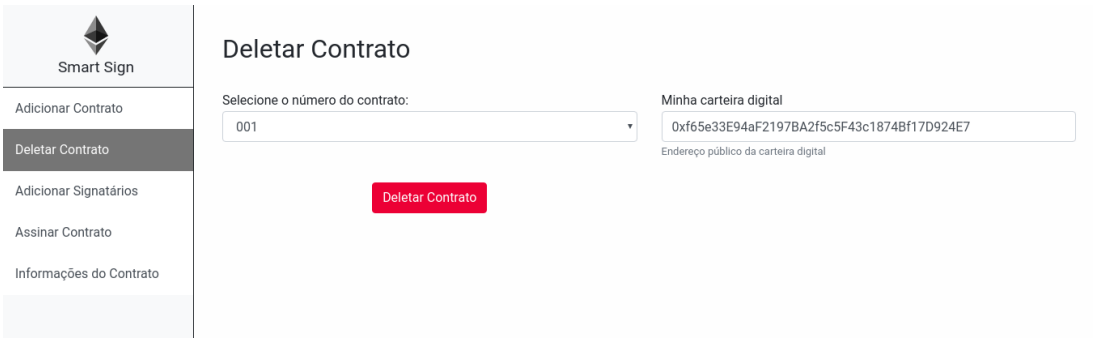
Figura 24 – Smart Sign - Tela de Cadastro do Contrato

Fonte: Autor

5.2 Tela para Deletar Contrato

Como apresentado na Figura 25, na tela para deletar contratos, é preciso informar o endereço público da carteira digital, assim será listado o número identificador do contrado vinculado a esta carteira. Um contrato só será excluído se nenhum endereço de signatário foi atribuído a ele.

Figura 25 – Smart Sign - Tela para Deletar Contratos



Fonte: Autor

5.3 Tela para Adicionar Signatário ao Contrato

Nesta tela, é possível vincular dois signatários ao contrato através do endereço público de carteira digital dos mesmos, como apresentado na Figura 26. Para issso é preciso informar o endereço público da carteira digital do proprietário e escolher o contrato que deseja vinculá-los.

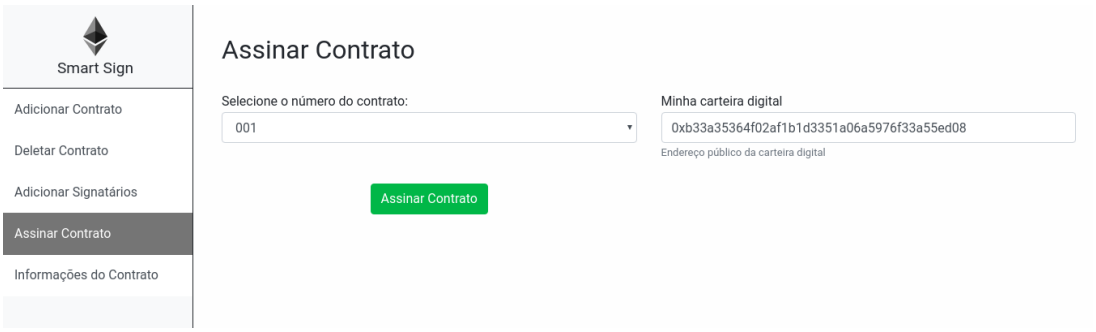
Figura 26 – Smart Sign - Tela para Adicionar Signatário ao Contrato

Fonte: Autor

5.4 Tela para Assinar Contrato

Na tela de assinatura do contrato, é preciso informar o endereço público da carteira digital do signatário e escolher o contrato que deseja assinar, como apresentado na Figura 27. Só será possível assinar um contrato em que o signatário tenha sido vinculado, do contrário ele receberá um erro ao tentar assinar um contrato não vinculado.

Figura 27 – Smart Sign - Tela para Assinar Contrato



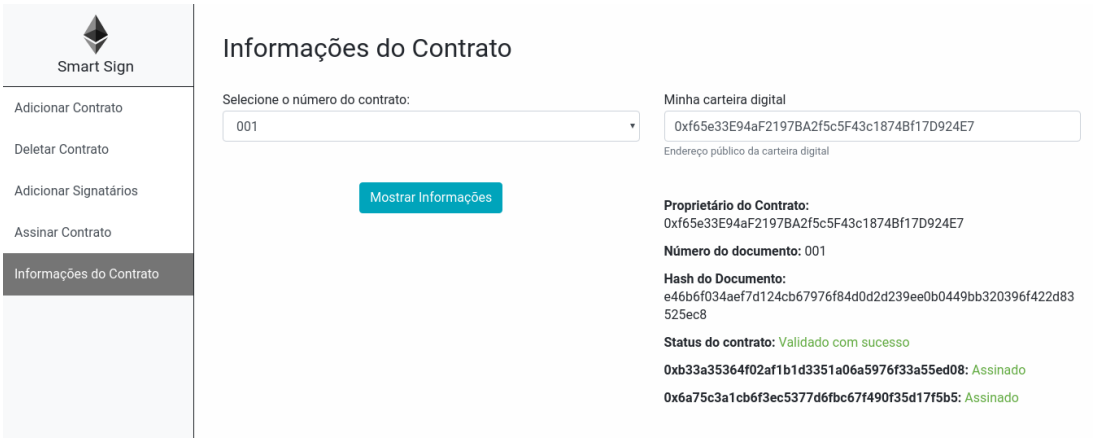
Fonte: Autor

5.5 Tela para Mostrar Informações do Contrato Cadastrado

Como apresentada na Figura 28, na última tela é possível obter informações de um contrato cadastrado. Para isso, é preciso informar o endereço público da carteira digital do proprietário ou do signatário e selecionar o contrato através do número identificador. A informações que podem ser visualizadas são:

- Proprietário do contrato;
- Número do documento;
- Hash do Documento;
- Status do contrato, que pode ser "Validado com sucesso" ou "Aguardando validação";
- Endereço público dos signatários com seu status, que pode ser "Assinado" ou "Aguardando assinatura".

Figura 28 – Smart Sign - Tela para Mostrar Informações do Contrato Cadastrado



Fonte: Autor

6 CONCLUSÃO E TRABALHOS FUTUROS

O estudo dos conceitos de desenvolvimento do serviço de *smart contracts* numa rede *blockchain*, juntamente das ferramentas utilizadas na elaboração do trabalho e todo o processo de desenvolvimento envolvido, permitiu a exploração real das etapas da construção de uma plataforma descentralizada. Este processo possibilitou adquirir uma visão ampla da área de desenvolvimento de *software* e suas particularidades.

Na elaboração deste trabalho, foi possível conhecer diversas características e restrições das tecnologias utilizadas no desenvolvimento do serviço de *smart contracts*. Com a EVM em conjunto ao *framework Truffle*, foi possível prover um ambiente para o desenvolvimento e implantação de contratos.

Durante a elaboração deste trabalho, o tópico que mais necessitou atenção foi no desenvolvimento e implantação de novos signatários nos contratos. Foi preciso encontrar uma maneira de após o contrato ser criado, os novos usuários com suas chaves públicas preveamente escolhidas, assinarem o contrato e assim implantá-lo na *blockchain* da *Ethereum*.

Outro ponto importante abordado durante a elaboração do trabalho, foi o desenvolvimento da interface do usuário, no qual o usuário interage com o serviço de *smart contracts* para a criação de contratos digitais. Utilizando a biblioteca *ReactJS*, em conjunto com a linguagem de programação *JavaScript* e o gerenciador de pacatos *NodeJS*, foi possível a contrução da interface de maneira simples e rápida, além torná-la de fácil manutenibilidade.

De acordo com a análise realizada durante o estudo de conceitos e outros serviços similares, observou-se a oportunidade de desenvolver uma plataforma que mudasse a maneira cotidiana de criar e validar documentos jurídicos, tanto físicos quanto digitais, utilizando uma tecnologia ainda pouco explorada nesse mercado. Assim, conseguiu-se diminuir os processos burocráticos, que são os grandes problemas enfrentados ao criar um documento juridíco, sem perder a validade jurídica.

Para trabalhos futuros, no que diz respeito ao serviço de *smart contracts*, pretende-se aprimorar o contrato possibilitando a adição de campos personalizados, implementar um algoritmo para calcular a quantidade gasta em ETH pelas transações realizadas e tornar dinâmica a quantidade de signatários vinculados ao contrato.

Já na plataforma desenvolvida em *ReactJS*, pretende-se desenvolver o módulo de cadastro de usuários por meio da chave pública de suas repectivas carteiras digitais, um módulo capaz de avisar ao signatário previamente cadastrado, de que o mesmo possui um contrato à espera de sua assinatura. Outra opção de melhoria, é disponibilizar em um *marketplace* exclusivo para aplicações descentralizadas, tornando assim a aplicação mais próxima do conceito de DApp.

REFERÊNCIAS

ALIAGA, Y.; HENRIQUES, M. **Uma comparação de mecanismos de consenso em blockchains**. 2017.

ALIAGA, Y. et al. **Proof-of-Stake baseado em Tempo Discreto**. 2019.

ANTONIO, C. S. **Pro React: Build complex front-end applications in a composable way with React**. 2015.

ANTONOPOULOS, A. M.; WOOD, G. **Mastering Ethereum**. [S.l.]: O'Reilly, 2018. v. 1.

ASHWINI, A. **What is the Difference between ReactJS and React Native?** 2017. Disponível em: <https://www.cognitiveclouds.com/insights/what-is-the-difference-between-react-js-and-react-native>. Acesso em: 16 de setembro de 2019.

ASOLO, B. **What Is SHA-256 And How Is It Related to Bitcoin?** 2018. Disponível em: <https://www.mycryptopedia.com/sha-256-related-bitcoin>. Acesso em: 09 de dezembro de 2019.

BOOTSTRAP. **Bootstrap - The most popular HTML, CSS, and JS library in the world**. 2018. Disponível em: <https://getbootstrap.com/docs/4.3/getting-started/introduction>. Acesso em: 20 de novembro de 2019.

CAMARGO, R. F. de. **Bitcoins, Blockchain e Smart Contracts: por que a área financeira precisa saber isso?** 2017. Disponível em: <https://www.treasy.com.br/blog/bitcoins-blockchain-smart-contracts>. Acesso em: 02 de novembro de 2019.

CHANDRAYAN, P. **Understanding DApp's: Decentralized Applications**. 2018. Disponível em: <https://medium.com/swlh/understanding-dapps-decentralized-applications-8f3668ebdc9a>. Acesso em: 20 de maio de 2019.

CIMATTI, R. **Os contratos digitais têm validade jurídica?** 2018. Disponível em: <https://www.migalhas.com.br/dePeso/16,MI281445,31047-Os+contratos+digitais+tem+validade+juridica>. Acesso em: 25 de novembro de 2019.

CROSBY, M. et al. **BlockChain Technology: Beyond Bitcoin**. 2016.

ETHEREUM. **Ethereum Homestead**. 2016. Disponível em: <http://www.ethdocs.org/en/latest/index.html>. Acesso em: 20 de maio de 2019.

ETHHUB. **Welcome to EthHub**. 2019. Disponível em: <https://docs.ethhub.io>. Acesso em: 11 de agosto de 2019.

GOLDEN. **Truffle Framework**. 2017. Disponível em: <https://golden.com/wiki/Truffle_Framework>. Acesso em: 17 de novembro de 2019.

GREGORY. **Solidity for Beginners · Smart Contract Development Crash Course**. 2019. Disponível em: <https://www.dappuniversity.com/articles/solidity-tutorial>. Acesso em: 23 de outubro de 2019.

GREVE, F. et al. **Blockchain e a Revolução do Consenso sob Demanda**. 2018.

GRISHCHENKO, I. et al. **A Semantic Framework for the Security Analysis of Ethereum Smart Contracts**. 2018.

GUEDES, G. T. A. **UML 2 - Uma Abordagem Prática**. [S.l.]: Novatec, 2011.

HANKERSON, D. et al. **Guide to Elliptic Curve Cryptography (Springer Professional Computing)**. [S.l.]: Springer, 2004.

IMPAGLIAZZO, R.; LUBY, M. **One-way Functions are Essential for Complexity Based Cryptography**. 1989.

JAKOBSSON, M.; JUELS, A. **Proofs of Work and Bread Pudding Protocols(Extended Abstract)**. 1999.

JONAS, B. et al. **The Reactive Manifesto**. 2014. Disponível em: <https://www.reactivemanifesto.org>. Acesso em: 21 de maio de 2019.

KING, S. **Primecoin: Cryptocurrency with Prime Number Proof-of-Work**. 2013.

LAMOUNIER, L. **Algoritmos de Consenso: A Raiz Que Sustenta a Tecnologia Blockchain**. 2018. Disponível em: <https://101blockchains.com/pt/algoritmos-de-consenso>. Acesso em: 26 de outubro de 2019.

LIMA, M. S. d.; GREVE, F. G. P. **Detectando Falhas Bizantinas em Sistemas Distribuídos Dinâmicos**. 2009.

MALAVASI, A. **Afinal, Javascript e ECMAScript são a mesma coisa?** 2017. Disponível em: <https://medium.com/trainingcenter/afinal-javascript-e-ecmascript-s~ao-a-mesma-coisa-498374abbc47>. Acesso em: 14 de setembro de 2019.

MATERA, S. **O que é CSS e qual sua importância?** 2012. Disponível em: <http://www.matera.com/blog/post/o-que-e-css-e-qual-sua-importancia>. Acesso em: 03 de setembro de 2019.

MOZILLA DEVELOPER NETWORK. **JavaScript**. 2019. Disponível em: <https://developer.mozilla.org/pt-BR/docs/Web/JavaScript>. Acesso em: 14 de setembro de 2019.

NAKAMOTO, S. **Bitcoin: A Peer-to-Peer Electronic Cash System**. 2008.

NARAYANAN, A. et al. **Bitcoin and Cryptocurrency Technologies: A Comprehensive Introduction**. [S.l.]: Princeton University Press, 2016.

NODE BR. **O que é a NPM do Node.JS**. 2016. Disponível em: <http://nodebr.com/o-que-e-a-npm-do-nodejs>. Acesso em: 10 de novembro de 2019.

NODE BR. **O que é Node.js?** 2016. Disponível em: <http://nodebr.com/o-que-e-node-js>. Acesso em: 10 de novembro de 2019.

PIRES, T. P. **Tecnologia Blockchain e suas Aplicações para provimento de Transparência em Transações Eletrônicas**. 2016.

ROCHA, R. V. F.; PEREIRA, D. O.; BRAGANÇA, S. H. F. J. **Blockchain e Smart Contracts: Como a tecnologia está mudando a Intermediação e o Direito Empresarial**. 2018.

ROQUE, S. J. **Direito contratual civil-mercantil**. [S.l.]: Ícone, 2003. v. 1.

ROUSE, M. **What is peer-to-peer (P2P)?** 2019. Disponível em: <https://searchnetworking.techtarget.com/definition/peer-to-peer>. Acesso em: 10 de setembro de 2019.

RUPARELIYA, P. **What Are Decentralized Applications (dApps)? — Explained With Examples**. 2018. Disponível em: <https://hackernoon.com/what-are-decentralized-applications-dapps-explained-with-examples-7ff8f2c4a460>. Acesso em: 20 de maio de 2019.

SALOMAA, A. **Public-Key Cryptography**. [S.l.]: Springer-Verlag Berlin Heidelberg, 1996.

SELLIER, A. **Less**. 2017. Disponível em: <http://lesscss.org>. Acesso em: 23 de novembro de 2019.

SERPRO. **Serpro lança plataforma Blockchain**. 2017. Disponível em: <http://www.serpro.gov.br/menu/noticias/noticias-2017/serpro-lanca-plataforma-blockchain-2>. Acesso em: 09 de setembro de 2019.

SZABO, N. **Smart Contracts: Formalizing and Securing Relationships on Public Networks**. 1997. Disponível em: <https://firstmonday.org/ojs/index.php/fm/article/view/548/469>. Acesso em: 02 de novembro de 2019.

TRUFFLE SUITE. **Sweet Tools For Smart Contracts**. 2017. Disponível em: <https://www.trufflesuite.com/docs>. Acesso em: 17 de novembro de 2019.

ULRICH, F. **Bitcoin: A Moeda na Era Digital**. [S.l.]: Instituto Ludwig Von Mises Brasil, 2014.