Universidade de São Paulo
Instituto de Matemática e Estatística
Bachalerado em Ciência da Computação

Felipe Brigalante
Hugo Mitsumori
Mateus Rocha Mazzari

Paratii

São Paulo
Dezembro de 2017

Paratii

Monografia final da disciplina
MAC0499 – Trabalho de Formatura Supervisionado.

Supervisor: Prof. Dr. Alfredo Goldman
[Cosupervisor: Jorge Melegati]

São Paulo
Novembro de 2017

Resumo

Neste texto é descrito o desenvolvimento do projeto *open-source* Paratii em conjunto com outras equipes de diferentes países. O objetivo é desenvolver um sistema de distribuição de vídeo descentralizado que oferece aos criadores de conteúdo uma recompensa pelos vídeos disponibilizados em proporção à audiência gerada. Esse modelo é contrário à distribuição centralizada, mais utilizada nos dias atuais. Utiliza-se da arquitetura *peer-to-peer* para o armazenamento e transferência do conteúdo e a *blockchain* da Ethereum para o sistema de recompensa e troca de informações. A equipe ficou responsável pelo desenvolvimento do *front-end*, em Meteor.js, e optou pela utilização de métodos ágeis, buscando a escrita de código belo. Neste documento são apresentadas as tecnologias utilizadas, as funcionalidades implementadas e as dificuldades encontradas durante o desenvolvimento desse grande projeto. Graças à utilização de boas ferramentas e práticas de programação, foi possível contribuir para o desenvolvimento de um *front-end* funcional e evolutivo. O projeto continua em desenvolvimento pelas outras equipes responsáveis.

Palavras-chave: blockchain, desenvolvimento web, peer-to-peer, métodos ágeis.

Abstract

This text describes the development of the open-source project Paratii together with teams from different countries. The goal is to develop a decentralized video distribution system that offers content creators a reward for available videos in proportion to generated audience. This model is contrary to the centralized distribution, most used today. The peer-to-peer architecture is used for storing and transferring content and the Ethereum blockchain is used to reward and information exchange system. The team was assigned to the front-end development using Meteor.js, and choose to apply agile methods aiming for beautiful code. In this document, the employed technologies, functionalities implemented and the difficulties encountered during the development of this long project are presented. Thanks to the use of good tools and programming practices, it was possible to contribute to the development of a functional and scalable front-end. The project is still under development by other teams.

Keywords: blockchain, web development, peer-to-peer, agile methodology.

Sumário

Capítulo 1

Introdução

Nos últimos anos, houve um grande crescimento de distribuição de conteúdo e informação por vídeo pela Internet. Contudo, os produtores de vídeos possuem poucas alternativas para publicação e divulgação de seus trabalhos, acabando por publicá-los em grandes sites e redes sociais, como o Youtube e Facebook. Tais sites possuem políticas rigorosas e pouco transparentes em relação a como os vídeos são monetizados, em geral, pagando aos produtores um valor proporcional à quantidade de visualizações.

Por outro lado, se tem observado o desenvolvimento de tecnologias que permitem a criação de plataformas descentralizadas em que não há uma entidade capaz de controlá-las. Por exemplo, o protocolo BitTorrent[1], utilizado para transmissão de arquivos, e mais recentemente o Bitcoin[2], utilizado como moeda digital.

A proposta deste trabalho é o desenvolvimento de um sistema de distribuição de vídeo utilizando a plataforma *open source* distribuída Ethereum[3]. Essa plataforma surgiu bem recentemente e vem sendo muito valorizada por permitir a movimentação de contratos inteligentes, que são pequenos programas descritos por uma lógica pré-estabelecida. Esses programas ficam armazenados na *blockchain* (Subseção 3.2.2) e são executados e verificados por muitos computadores para garantir sua confiabilidade. Isso permite uma grande gama de possibilidades e novas formas de fornecer serviços.

A plataforma, denominada Paratii[4], permite transações utilizando uma criptomoeda (PTI) para movimentar valores monetários entre usuários, criadores de conteúdo e anunciantes de propagandas. Por não possuir uma infraestrutura dependente de equipamentos centralizados de uma empresa, mas do armazenamento dos usuários, os custos de hospedagem de vídeo são inferiores e os usuários têm participação ativa nos lucros e na valorização de conteúdo. Com isso, o objetivo é que o sistema seja auto gerenciável.

O desenvolvimento dessa plataforma é extenso e envolve inúmeros desafios. O foco do grupo foi a implementação do *front-end* da aplicação, isto é, o fluxo de navegação do usuário

[1] http://www.bittorrent.org/
[2] https://bitcoin.org/pt_BR/
[3] https://www.ethereum.org/
[4] http://paratii.video/

e o *player* de vídeo.

Além do contato com novas tecnologias, esse projeto proporcionou um contato com o mercado fora da universidade. O projeto é *open source*, patrocinado pela empresa brasileira BossaNova, e conta com a contribuição de programadores do mundo inteiro. Como o projeto envolve um grande número de pessoas, os princípios ágeis foram aplicados para facilitar a comunicação e organização entre os desenvolvedores e clientes e para garantir uma melhor qualidade de código.

O primeiro capítulo *Sobre o Paratii* introduz a ideia de negócio do Paratii. O capítulo *Projeto* descreve as tecnologias e ferramentas utilizadas. Em *Métodos Ágeis*, é explicado essa metodologia e como ela foi utilizada no trabalho. No capítulo *Desenvolvimento* é explicado detalhadamente a implementação do sistema. No capítulo *Funcionalidades* são apresentadas as funcionalidades implementadas pela equipe. A *Conclusão* apresenta um *feedback* dos colaboradores e uma reflexão do grupo.

Capítulo 2

Sobre o Paratii

2.1 Paratii

Como descrito no *whitepaper* (Caro, Gerbrandy, Sant'Ana, e Perez, 2017), Paratii é um sistema descentralizado de distribuição de vídeo que oferece aos criadores de conteúdo uma alternativa sem intermediação, justa e gratuita às plataformas centralizadas. Por meio desse serviço, os criadores de vídeo são recompensados pelos vídeos que produzem proporcionalmente à audiência gerada.

O proprietário do conteúdo pode definir o modelo de monetização (*pay-per-view*, *advertising*, etc), enquanto os usuários influenciam diretamente no valor do vídeo, que é medido a partir de um *token* criptográfico chamado PTI, definido pela venda de publicidade e de conteúdo pago. Dessa forma, como o valor do vídeo também é dependente da sua audiência, o produtor pode compartilhar parte do seu lucro com ela. Isso é possível a partir de um mecanismo chamado *mercado de atenção*.

O controle de qualidade é realizado pelos próprios usuários. Eles podem avaliar, categorizar e denunciar vídeos, sendo recompensados por isso. Para garantir que a qualidade não seja manipulável, há um mecanismo de reputação em que, quanto maior a reputação do usuário, maior o peso de suas ações. Um dos objetivos a longo prazo do sistema é encontrar o balanço ideal para que seja vantajoso para os usuários trabalharem em prol do bom funcionamento da plataforma, e pouco vantajoso tentar se aproveitar da lógica interna.

2.2 Visão Geral

O acesso à plataforma do Paratii é feito por meio de páginas HTML que incorporam o *player* referenciando algum vídeo de interesse. Além da reprodução do vídeo, na própria interface do *player*, o usuário é capaz de acessar sua *playlist* pessoal, assim como sua carteira digital, que guarda a criptomoeda utilizada nas transações internas. Sendo de uso livre, qualquer pessoa/organização que queira reproduzir algum vídeo em sua página poderá hospedá-lo na plataforma e incluir o *player* do Paratii no HTML.

Os vídeos são armazenados nas máquinas dos usuários e distribuídos seguindo o modelo P2P (*peer-to-peer*) e utilizando um cliente *torrent* interno ao *player*. Ao acessar um vídeo, o cliente *torrent* procura a forma mais eficiente de disponibilizar o conteúdo com base nos *peers* disponíveis que possuem o mesmo conteúdo.

O núcleo do sistema foi implementado utilizando a *blockchain* do Ethereum, uma plataforma que permite transações de *smart contracts*, códigos inteligentes que são armazenados na *blockchain* e podem ser transmitidos para outros usuários. Toda troca de informação dentro do sistema é feita utilizando diferentes tipos de *smart contracts*, incluindo transações financeiras e mecanismos de qualidade do serviço.

2.3 Peer-to-peer

Peer-to-peer (ponto-a-ponto, P2P) é uma arquitetura de redes de computadores que tem o objetivo de descentralizar o funcionamento da rede, de modo que cada computador da rede funciona como cliente e servidor ao mesmo tempo. No projeto, tal arquitetura é utilizada para descentralizar o armazenamento e a distribuição dos vídeos, permitindo que um usuário consiga, por meio do sistema, baixar um vídeo ou enviá-lo para outros usuários. Essa tecnologia funcionará em conjunto com uma camada de incentivo: quem disponibiliza o conteúdo receberá uma recompensa em PTI por isso.

2.4 Aspectos econômicos

O Ether(ETH) é a criptomoeda nativa da *blockchain* do Ethereum, sendo hoje uma das mais utilizadas no mercado e com popularidade crescente. O interessante, porém, é que os *smart contracts* permitem a definição de um *token* próprio que pode ser utilizado para monetização de serviços que funcionem dentro da *blockchain*. Foi criada então, a padronização ERC-20 8, que garante que os *tokens* funcionem da mesma forma dentro do sistema Ethereum e tenham suporte da maioria das *wallets* de ether.

O *token* Paratii(PTI) segue a ERC-20 e será a unidade de movimentação financeira no sistema de distribuição de vídeo. Basicamente toda transação terá seu valor em PTI, desde a hospedagem até o controle de qualidade.

Uma vez que um conteúdo é disponibilizado, o produtor pode escolher entre distribuição gratuita ou *pay-per-view*. Neste caso, o usuário libera a visualização do vídeo por meio de uma transação de PTI. No caso de vídeos gratuitos, o retorno financeiro vem do valor das propagandas, fazendo parte do *mercado de atenção*.

Apesar de o valor das propagandas não ser repassado integralmente para o produtor, o desconto será significantemente menor do que no caso de plataformas centralizadas (Youtube, por exemplo, retêm em média 45% do valor), que acabam gerando custo de hospedagem e gerenciamento do sistema. Além disso, parte do valor gerado será incorporado a um sistema

de bônus que pode ser repassado ao produtor dependendo da audiência alcançada, de modo que o retorno final pode superar o total do valor da propaganda.

Uma outra parte do valor gerado será repassado aos usuários da plataforma que irão contribuir para garantir a qualidade do sistema, não só gerando mais valor para o vídeo por meio de audiência, mas também por meio de avaliação de conteúdo, associação de temas em comum, e até mesmo detectando atitudes ilegais dentro do ambiente. Em resumo, os próprios usuários manterão a qualidade do sistema e serão recompensados por isso.

Capítulo 3

Projeto

3.1 Tecnologias

3.1.1 HTML, CSS e JavaScript

A plataforma do Paratii funcionará como um serviço no navegador. Portanto, faz uso das tecnologias básicas da Web: HTML[1] e CSS[2]. O HTML5 fornece os controles básicos da aplicação, incluindo a interface do *player* de vídeo, enquanto o CSS permite definições detalhadas da aparência e *layout* dos objetos. Para agilizar o desenvolvimento, foi utilizado o Less.js[3], um pré-processador de CSS que adiciona funcionalidades extras, como variáveis e funções.

Para implementar o aspecto dinâmico, assim como a manipulação do arcabouço utilizado (apresentado em mais detalhes na subseção seguinte), utilizamos a linguagem JavaScript[4], que tem como vantagem o suporte na maioria dos navegadores atuais. Uma das funcionalidades importantes é o uso de programação assíncrona, que permite atualizações dinâmicas na página sem necessidade de acessar o servidor a todo momento. Em especial, utilizamos a sintaxe e alguns recursos da especificação ES6[5].

É importante notar que os dados são transmitidos no formato JSON, que apresenta um fácil entendimento sobre as informações e é também o formato utilizado pelo MongoDB.

3.1.2 Meteor.js

Meteor.js[6] é uma plataforma de desenvolvimento JavaScript baseada em Node.js que permite a criação de aplicações completas e integradas no lado do servidor e do cliente. O arcabouço fornece várias bibliotecas e uma estrutura que torna fácil o desenvolvimento de

[1] https://www.w3.org/html/
[2] https://www.w3.org/Style/CSS/
[3] http://lesscss.org/
[4] *Speaking JavaScript*, Rauschmayer (2014)
[5] *Standard ECMA-262*, International (2015)
[6] *Discover Meteor*, Coleman e Greif (2015)

serviços web modernos e eficientes. É integrado, por padrão, ao banco de dados MongoDB e tem suporte a várias outras ferramentas que auxiliam o desenvolvimento.

Todo o projeto foi feito sobre essa plataforma pois fornece uma integração fácil com a API JavaScript do Ethereum (web3[7]). Além disso, foi necessário desenvolver um MVP (produto viável mínimo) em um tempo relativamente curto para ser apresentado, e o Meteor.js permite um desenvolvimento rápido e com curva de aprendizado acentuada.

3.1.3 Peer-to-peer

Para a implementação dessa arquitetura no projeto foram considerados três protocolos de compartilhamento de arquivos: WebTorrent[8], IPFS[9] e Swarm[10].

1. **WebTorrent**: um cliente de transmissão de *torrent* para navegadores. Prós: API pronta e de fácil implementação. Contras: é uma rede separada dos *torrents* comuns e não tem camada de incentivo;

2. **Swarm**: sistema de armazenamento de arquivos sobre *blockchain* sendo desenvolvida pela Ethereum. Prós: integrado à *blockchain*, em desenvolvimento pela própria Ethereum e inclui camada de incentivo. Contras: estágio inicial de desenvolvimento e ainda não é possível sua utilização num futuro próximo;

3. **IPFS**: protocolo *peer-to-peer* que tem como objetivo descentralizar e agilizar a internet. Prós: estado atual é funcional, tem intenção de adicionar camada de incentivo e sua utilização não é limitada ao armazenamento de arquivos. Contras: sua versão para navegador está mais atrasada que sua versão principal e só tem as operações básicas.

A primeira implementação de *peer-to-peer* foi feita utilizando o WebTorrent, pois este apresentava uma API mais amigável e é a opção mais estável/otimizada dentre as três. Com o desenvolver do projeto e a chegada de novos colaboradores, a equipe optou por migrar para o IPFS, que é o utilizado atualmente.

3.1.4 WebTorrent

Uma das maiores dificuldades relativas à distribuição de conteúdo são os custos em manter uma estrutura capaz de atender a acessos múltiplos e simultâneos a um mesmo recurso. O protocolo BitTorrent soluciona essa questão propondo que os recursos sejam fornecidos de forma distribuída. Ou seja, ao invés de haver uma única fonte de fornecimento (que eventualmente poderia ser sobrecarregada), todos que acessam os dados em questão

[7] https://github.com/ethereum/web3.js/
[8] https://webtorrent.io/
[9] https://ipfs.io/
[10] https://github.com/ethersphere/swarm

guardam cópias idênticas localmente e se tornam fornecedores do conteúdo. Dessa forma, a quantidade de pontos de fornecimento cresce proporcionalmente à quantidade de acessos.

Esse foi um dos princípios que norteou a concepção do projeto Paratii. Uma vez que vários usuários hospedam o mesmo conteúdo para acesso externo, não há necessidade de gastos com hospedagem em servidores que atendam a uma grande quantidade de acesso simultâneo. Os próprios usuários contribuem para a infraestrutura e são recompensados por fazerem parte do sistema.

No projeto foi utilizado, inicialmente, o WebTorrent, uma variante do BitTorrent desenvolvida para transmitir dados por meio de navegadores web e com uma API para JavaScript. Ela possui suporte para RTC (*Real-Time Communications*) e não necessita de plugins para ser utilizada pelo navegador.

3.1.5 IPFS

O IPFS (*InterPlanetary File System*) é um sistema de arquivos distribuídos *peer-to-peer* que contém nós que armazenam objetos no armazenamento local. Esses objetos representam arquivos e outras estruturas de dados e podem ser transferidos entre os nós (Benet, 2014).

O desenvolvimento desse protocolo vem da necessidade de agilizar e baratear o HTTP, que é dependente de um servidor centralizado.

O IPFS é dividido entre os seguintes sub-protocolos:

1. Identidades (*Identities*): um nó é uma estrutura que contém uma chave secreta e uma chave pública, o identificador do nó é um *hash* da chave pública;

2. Rede (*Network*): a comunicação pode ser feita em qualquer protocolo de transporte, com preferência ao WebRTC. As mensagens são verificadas por meio de um *checksum*;

3. Roteamento (*Rounting*): utilizando DSHT (*Distributed Sloppy Hash Table)* baseado no S/Kademlia é possível achar um nó que contenha o objeto desejado;

4. Troca (*Exchange*): a distribuição de dados ocorre na troca de blocos utilizando o protocolo BitSwap. Para que um nó contribua no compartilhamento dos blocos, há um sistema de crédito, que a partir de uma função probabilística, dificulta o roteamento quanto maior o débito;

5. Objetos (*Objects*): utilizando Merkle DAG como o Git, todo conteúdo tem um *multihash* único garantindo a detecção de dados corrompidos, adulterados ou duplicados;

6. Arquivos (*Files*): utilizando a mesma estrutura do Git com *blob*, *list*, *tree*, *commit* e *block*, permitindo o versionamento dos arquivos.

A criptomoeda Filecoin[11] adiciona uma camada de incentivo ao IPFS, recompensando quem hospeda arquivos na rede.

[11] https://filecoin.io/

3.2 Ethereum

3.2.1 Sobre

Ethereum é uma plataforma para construção de aplicações descentralizadas por meio da tecnologia *blockchain*. Tais aplicações são executadas sem qualquer possibilidade de censura, fraude ou interferência de terceiros.

O levantamento de recursos para o projeto iniciou-se em 2014 com a pré venda de Ether, a criptomoeda utilizada para executar transações dentro do *Ethereum*. Atualmente o projeto é mantido pela *Ethereum Foundation*, uma instituição suíça sem fins lucrativos.

3.2.2 Blockchain

A tecnologia *blockchain* surgiu como uma forma eficiente e confiável de registrar transações e mapear recursos de forma distribuída e aberta envolvendo um grande número de usuários sem a necessidade de gerenciamento de terceiros. Sua popularização se deu em conjunto com o surgimento das criptomoedas, principalmente a Bitcoin, apesar de seu uso não ser exclusivo para transações monetárias.

Essencialmente, a *blockchain* da Ethereum consiste em uma máquina de estados baseada em transações. No início a *blockchain* está vazia, sem que nenhuma transação tenha ocorrido, e à medida que as transações são aplicadas, o estado da *blockchain* se altera. Basicamente, o estado atual da *blockchain* é resultado da aplicação de todas as transações em ordem a partir do estado inicial vazio.

Essas transações são agrupadas em blocos, e cada bloco possui a referência para o bloco anterior, formando um encadeamento entre os blocos (daí o nome *Blockchain*).

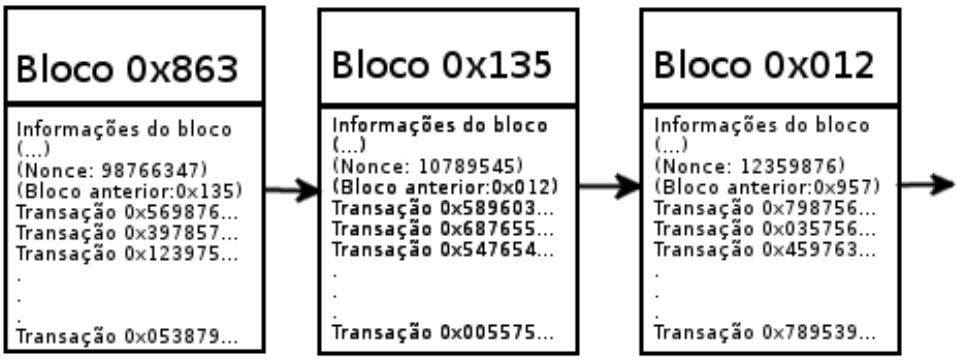

Figura 3.1: *Blocos encadeados*

Para que uma transação seja aplicada à *blockchain* é necessário que ela seja validada. Esse processo de validação é chamado de mineração, e ocorre quando um nó da rede gasta seus recursos computacionais para criar e validar blocos de transações. Alguns nós da rede, os mineradores, tentam ao mesmo tempo validar novos blocos, e quando algum deles consegue, os demais nós são informados que um novo bloco foi minerado e deve ser adicionado à

blockchain, fornecendo uma "prova"de que de fato esse novo bloco é válido. Essa "prova"é chamada de *proof of work* e será discutida em mais detalhes nas próximas subseções. Quando um bloco é minerado, o minerador responsável é automaticamente recompensado com Ether (ETH).

Potencialmente, vários blocos diferentes são minerados ao mesmo tempo por mineradores diferentes e isso causa uma série de problemas, pois alguns nós serão *blockchains* diferentes de outros e será impossível dizer qual deles está correto. Quando isso ocorre, dizemos que ocorreu um *"fork"* na *blockchain*.

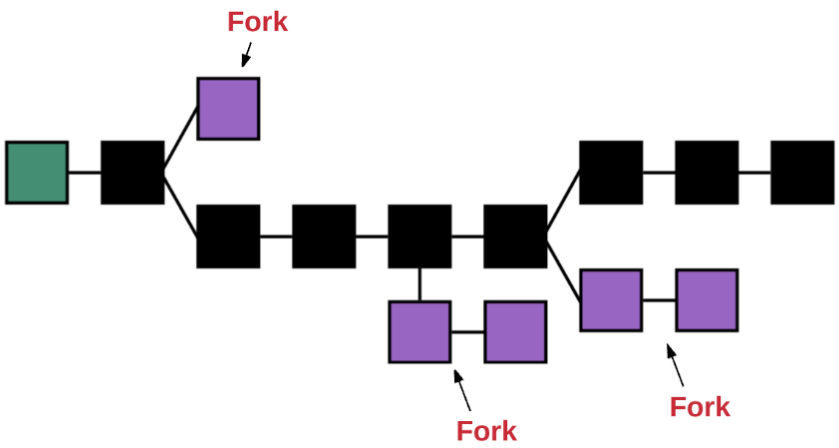

Figura 3.2: *Exemplo de forks na blockchain*
Kasireddy (2017)

Para resolver esse problema, o Ethereum possui um mecanismo chamado *GHOST (Greedy Heaviest Observed SubTree)*. Para determinar qual caminho de blocos deve ser utilizado pela *blockchain*, o *GHOST* simplesmente escolhe o caminho mais longo dentre todos os possíveis a partir do bloco inicial, pois quanto maior o caminho, maior o número de blocos nele e, consequentemente, maior foi o esforço computacional para minerá-lo. Desse modo, todos os nós podem concordar em qual *blockchain* utilizar. Note que não existe caminho mais correto ou menos correto, isso é apenas uma convenção para estabelecer qual *blockchain* deve ser utilizada em caso de conflitos.

3.2.3 Contas

O estado da *blockchain* é composto por objetos chamados de contas, as quais cada uma possui um estado individual e um identificador de 160-bits.

No Ethereum existem dois tipos de contas:

- *EOA (Externally Owned Accounts)*: São contas protegidas por chaves privadas e não tem nenhum tipo de código associada a elas;

- *Contract Accounts*: São contas controladas pelo código de seu *SmartContract* associado.

A diferença fundamental entre os dois tipo de conta é que as *EOA* podem enviar transações para outras contas criando e assinando-as digitalmente com sua chave privada. O único tipo de transação que pode ser enviado para uma *EOA* é a transferência de *Ether*. Por outro lado, transações enviadas para uma *Contract Account* irão executar o código associado à essa conta, permitindo que diversas ações sejam executadas: salvar alguma informação no estado dessa conta, realizar algum cálculo, entre outros. Já as *Contract Accounts* não podem enviar transações diretamente, mas podem enviar transações como consequência de uma outra transação recebida. Desse modo, todas as transições de estado da *blockchain* são geradas por transações criadas por *EOAs*.

O estado de uma conta possui os seguintes campos:

- *nonce*: Indica quantas transações foram criadas por essa conta no caso de *EOAs*, e no caso de *Contract Accounts* esse campo indica quantas criações de contratos foram feitas por essa conta;
- *balance*: A quantidade de Ether que a conta possui;
- *storageRoot*: O *hash* da raiz da *Merkle Patricia Tree* que armazena o conjunto de dados associado a essa conta. Inicialmente essa árvore está vazia;
- *codeHash*: Contém o *hash* do código associado a essa conta. Naturalmente esse campo só fica preenchido para *Contract Accounts*. Note que esse campo não pode alterado após sua criação.

3.2.4 Armazenamento de Dados

Como dito anteriormente, o estado da *blockchain* é o estado de todas as contas combinadas. Para armazenar as informações de todas as contas, o Ethereum utiliza uma estrutura de dados conhecida como *Merkle Patricia Tree*. Essa estrutura de dados é uma árvore binária na qual as informações ficam armazenadas apenas nos nós folha. Nos demais nós da árvore, ficam armazenados o *hash* de seus dois nós filhos. No caso do Ethereum, o algoritmo de *hash* utilizado é o KECCAK-256. Dessa forma a informação armazenada na árvore é quebrada em pequenos pedaços que são armazenados nos nós folha.

Nessa árvore, todos os dados armazenados devem ter uma chave que o identifique, e a busca por uma certa chave deve retornar o valor associado a ela. Assim, a *Merkle Patricia Tree* possui uma propriedade importante: para um dado nó na árvore e uma chave, deve ser possível descobrir em qual dos filhos desse nó potencialmente está armazenado o valor associado a essa chave, possibilitando consultas eficientes aos dados.

Dessa forma, cada bloco da *blockchain* possui uma árvore que mapeia uma conta a seu estado, e cada conta possui uma árvore que armazena seus dados como mostrado na Figura 3.3.

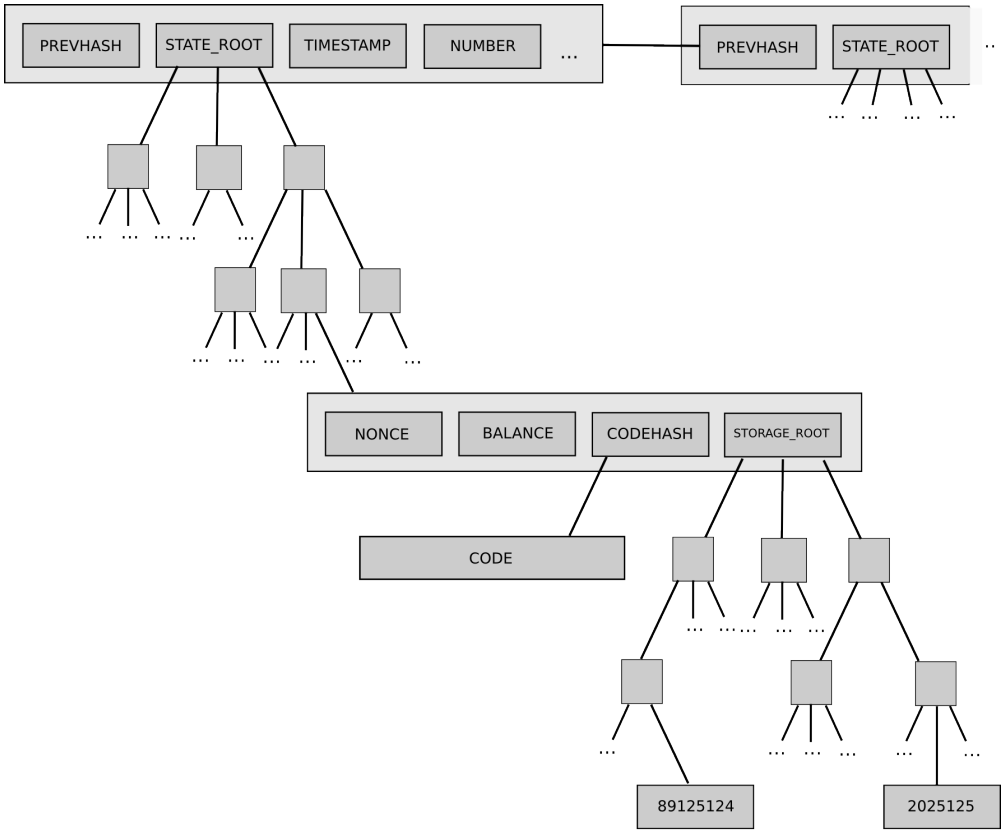

Figura 3.3: *Estrutura de dados da blockchain*
Buterin (2014a)

Além dessa arvore que armazena o estado das contas, cada bloco possui mais duas *Merkle Patricia Trees*. Uma para armazenar as transações que foram executadas(*Transaction Tree*) e outra para armazenar os "recibos"das transações(*Receipt Tree*). Isto é, quanto Ether foi transferido, qual o saldo final das duas contas e quanto Ether foi gasto para executar a transação.

3.2.5 Light Nodes e Full Nodes

A rede Ethereum consiste em vários nós que compartilham a mesma *blockchain* e propagam uns para os outros quando uma transação ocorre ou um bloco é minerado. Um nó é conhecido como *Full Node* caso ele possua toda a informação da *blockchain*. Todos os nós mineradores precisam ser *Full Nodes*, pois, para o processo de mineração esses dados são necessários. Porém, em muitos casos, os nós não precisam executar todas as transações, e apenas precisam executar consultas simples. Nesses casos, os nós conhecidos como *Light Nodes* armazenam apenas os cabeçalhos de cada bloco e alguns poucos nós da árvore em que a informação necessária está contida. Isso é possível devido a maneira como a informação

fica armazenada nas *Merkle Patricia Trees*, pois caso qualquer informação nas folhas seja alterada, algum de seus nós pais irá apresentar um *hash* inválido. Para fazer essa verificação o *Light Client* utiliza alguns poucos nós, como pode ser visto na Figura 3.4.

Em resumo, a grande vantagem de se usar *Merkle Patricia Trees* é que qualquer nó da rede pode validar um pequeno pedaço da *blockchain* sem precisar da *blockchain* completa, que pode eventualmente se tornar muito grande.

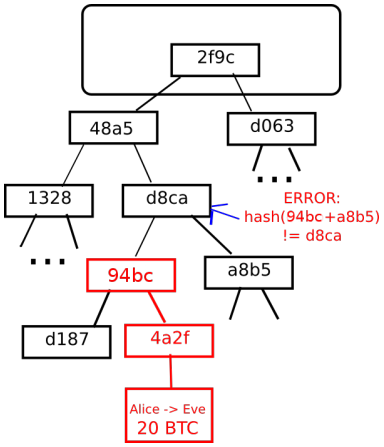

Figura 3.4: *Verificação de informação*
Buterin (2014b)

3.2.6 Transações

Como visto nas subseções anteriores, transações são mensagens assinadas criptograficamente e enviadas para a *blockchain*. Essencialmente, existem dois tipos de transações no Ethereum:

- Mensagens: basicamente o tipo de mensagem que foi explicado anteriormente, em que uma conta envia uma mensagem para outra;
- Criação de Contratos: esse tipo de transação é realizada para criar novos contratos.

Durante a execução de uma transação o esforço computacional envolvido é medido em *Gas*. Todas as transações possuem dois campos muito importantes: o *GasPrice* e o *GasLimit*. O *GasPrice* indica quanto em Ether o autor da transação está disposto a pagar por *Gas*, enquanto o *GasLimit* indica qual o máximo de *Gas* deve ser utilizado para essa transação. Desse modo, o máximo de Ether consumido para executar a transação será de $GasPrice \times GasLimit$. Caso a transação seja executada com sucesso, o *Gas* que não foi consumido é restituído para o autor da transação, e caso a transação falhe porque a quantidade de esforço computacional excedeu o *GasLimit*, todas as alterações feitas pela transação até o momento são revertidas e nenhum *Gas* é devolvido ao autor da transação. Afinal, houve esforço computacional, independentemente do resultado.

Os nós que processam as transações são os mineradores e, normalmente, quanto maior o *GasPrice* de uma transação, mais rapidamente essa transação será validada por um minerador, uma vez que os mineradores podem escolher quais transações eles querem processar. Além disso, é possível que os mineradores informem o mínimo de *Gas* a ser pago para que uma transação seja processada por eles.

Também há uma taxa associada ao consumo de disco. Essa taxa existe pois, caso uma transação adicione uma informação na *blockchain*, todos os *Full Nodes* terão que armazenar essa informação. Por outro lado, uma transação que possua uma instrução que remove um dado terá o custo dessa instrução completamente abatido, pois essa operação de certa forma é saudável para o sistema, tendo em vista que todos os *Full Nodes* terão espaço liberado.

Essas taxas são fundamentais para o Ethereum, pois qualquer operação dentro da *blockchain* é muito custosa, uma vez que deve ser replicada para todos os *Full Nodes* da rede. Atualmente, os contratos funcionam bem para tarefas simples. Por sua vez, tarefas complexas podem colocar toda a rede sobre estresse, e a adição dessas taxas previne que isso ocorra.

Além disso, a linguagem utilizada para a escrita dos contratos é Turing completa[12] e inevitavelmente está suscetível ao Problema da Parada[13]. A inexistência do *GasLimit* faria com que os mineradores potencialmente executassem alguma transação indefinidamente.

No caso de transações que são iniciadas por *Contract Accounts*(lembrando que essas transações só podem ser iniciadas como consequência de transações criadas por *EOAs*), o *GasLimit* não precisa ser especificado e será consumido o *Gas* da transação original que a iniciou. Sendo assim, é responsabilidade da conta que cria uma transação utilizar um valor para o *GasLimit* suficiente para executar todas as subtransações que possam ocorrer.

Além dos campos *GasPrice* e *GasLimit* discutidos acima, todas as transações possuem os seguintes campos:

- *nonce*: total de transações já criadas pelo autor dessa transação;
- *to*: endereço da conta que irá receber a transação. No caso criações de contrato esse campo deve ficar vazio;
- *value*: quantidade de Ether que será transferida na transação. Esse campo indica o saldo inicial da nova conta em transações de criação de contratos;
- *v,r,s*: campos relativos a assinatura da conta que criou a transação.

Além disso, transações de criação de contrato possuem o campo *init*, que contém o código necessário para criação do contrato, e as transações do tipo mensagem contêm um campo *data* utilizado para passar algum parâmetro ao chamar funções de uma *Contract Account*.

[12] https://pt.wikipedia.org/wiki/Turing_completude
[13] https://pt.wikipedia.org/wiki/Problema_da_parada

3.2.7 Contratos inteligentes

Como discutido na subseção anterior, os contratos são criados a partir de uma transação que especifica o código do contrato. Uma vez criado, o contrato não pode mais ser alterado e passa a executar o código exatamente como especificado em sua criação. O contrato é executado pelos mineradores por meio da EVM (*Ethreum Virtual Machine*), que é basicamente uma máquina de Turing, exceto pela necessidade intrínseca de *Gas* para executar as computações.

Em geral, os contratos são escritos em linguagens de alto nível, sendo *Solidity* a linguagem mais utilizada atualmente. Essas linguagens são compiladas para *bytecode*, que é interpretado pela *EVM*.

3.2.8 Mineração de Blocos

Para que um conjunto de transações em um bloco seja verificado e incluído na cadeia, é necessário um consenso[14] entre os participantes. O mecanismo de consenso mais comum é o *Proof of Work* (usado atualmente no Bitcoin e no Ethereum), em que o conteúdo do novo bloco deve passar por uma função de *hash* (no caso do Ethereum foi desenvolvido o *ethhash*), que pode ou não validar as transações. No cabeçalho do bloco, como podemos ver na Figura 3.1 é incluso um *nonce*, que aumenta a entropia da criptografia, dando origem a um número grande de possíveis valores de *hash*. Os mineradores, então, buscam, na força bruta, um *nonce* tal que o valor do *hash* gerado seja menor do que um outro valor definido previamente a cada bloco, chamado de *dificuldade*. Matematicamente é fácil verificar se o *hash* encontrado corresponde às informações do bloco. Porém, é difícil encontrar o inverso, ou seja, a função de *hash* correspondente que satisfaça a *dificuldade*. Para isso, é necessário que vários usuários trabalhem (por isso o nome *Proof of Work*) testando diferentes *nonce* até que um deles satisfaça a dificuldade. Esse processo de busca pelo *nonce* é conhecido como mineração de bloco, e geralmente o minerador que validar o bloco primeiro recebe uma recompensa em criptomoeda, que é automaticamente adicionada ao seu saldo. Em alguns casos, essa recompensa diminui com o tempo seguindo algum critério pré-estabelecido.

Outro aspecto interessante da mineração no Ethereum é a existência de *Ommers*. Devido a velocidade com que novos blocos são minerados (em torno de 15 segundos), é bastante comum que vários *forks* e seus mineradores não recebam nenhuma recompensa. Pensando nisso o Ethereum implementou um sistema de *Ommers*, no qual mesmo quando um bloco não vai para a *blockchain* principal, seu minerador recebe uma pequena recompensa por incluir esse bloco.

No momento da escrita desta monografia, o Ethereum utiliza o mecanismo *Proof of Work* para validação. Porém, o objetivo é que em um futuro breve, ocorra uma migração para um outro mecanismo de validação chamado de *Proof of Stake*. Nesse mecanismo, não existe

[14] *Understanding Consensus Models* Baliga (2017)

uma competição para ser o validador do bloco. Ao contrário, o validador é escolhido pela plataforma de maneira determinística. No mecanismo *Proof of Work*, uma grande quantidade de energia elétrica é desperdiçada pelas máquinas mineradoras. Ao migrar para o *Proof of Stake*, esse desperdício deixa de existir.

3.2.9 Ethereum versus Bitcoin

Apesar de ambos usarem a estrutura de *blockchain* e permitirem transações de criptomoedas, as implementações são bem diferentes. Além das diferenças paramétricas, como tempo de validação e tamanho de bloco, o Ethereum permite transações contendo trechos de código (*smart contracts*), enquanto as transações da Bitcoin são apenas de valores monetários. Esse diferencial do Ethereum abre um leque de possibilidades, como a definição de regras de negócio e a criação de criptomoedas próprias (*tokens* por qualquer usuário. Essa tecnologia proporcionou um grande crescimento para os chamados ĐApps [15](Decentralized Applications), aplicações decentralizadas de código aberto que funcionam sobre a *blockchain* e utilizam criptomoeda própria.

Outra diferença significativa é em relação ao método de mineração. Como descrito anteriormente, os mineradores gastam calculo computacional para achar um *nonce* que, após passar pela função de *hash* junto com o bloco, tenha um valor inteiro menor do que o valor da dificuldade. Pensando em otimizar a mineração de Bitcoins, foram desenvolvidas placas ASIC (*application-specific integrated circuit*) específicas para calcular a função de *hash* sobre os blocos, de modo que os usuários possuidores desses equipamentos possuem larga vantagem em relação aos que utilizam CPUs e GPUs convencionais. Para que haja uma competição mais igualitária, o Ethereum utiliza um algoritmo diferente, conhecido como *ethash*[16]. Esse algoritmo tem a propriedade de ser *memory hard*, implicando que existe uma limitação relativa ao uso da memória. Para minerar, faz parte do procedimento o armazenamento de um conjunto de dados relativamente grande (cerca de 2GB atualmente), impossível de ser armazenado por completo nos caches de processamento de placas ASIC. Por conta disso, é necessário fazer acesso à memória, que no geral é bem mais lento do que leitura de caches, e acaba por ser o fator limitante da velocidade de mineração. Isso faz com que o uso de GPUs seja a forma mais viável de minerar Ether atualmente.

3.2.10 *Ethereum e Paratii*

O Paratii está ligado profundamente ao *Ethereum* e a ideia inicial do projeto é construir uma aplicação completamente descentralizada para compartilhamento e visualização de vídeos através de diversos contratos inteligentes.

Os seguintes contratos precisam ser implementados para tornar o Paratii uma plataforma completamente funcional e descentralizada:

[15] https://www.stateofthedapps.com/
[16] https://github.com/ethereum/wiki/wiki/Ethash

- Um contrato especificando o *token* PTI (seguindo a ERC-20) e sua geração e transferência;
- Um contrato de conteúdo, que descreve as trocas de informações dos vídeos;
- Um contrato de emissão, que faz a distribuição de recompensas para os criadores de conteúdo levando em conta as visualizações e qualidade dos vídeos;
- Um contrato de reputação, que atualiza a reputação do usuário de acordo com suas contribuições e qualidade de seus vídeos;
- Um contrato de qualidade, que trata as avaliações feitas pelos usuários em relação aos vídeos, o chamado *Proof of Quality*;
- Um contrato para buscar garantir a autenticidade das visualizações dos vídeos.

3.3 Ferramentas

3.3.1 Git

Como o projeto está sendo feito por várias pessoas, é necessário utilizar algum sistema de controle de versão distribuído. Dentre as opções disponíveis, como Mercurial, SVN, CVS, a equipe optou por utilizar o Git. Como descrito no site do Git: "O Git é um sistema de controle de versão distribuído, gratuito e de código aberto projetado para lidar com tudo, desde projetos pequenos até muito grandes com velocidade e eficiência.". Tanto os membros do projeto no Brasil como os de fora têm bastante familiaridade com o Git e isso basicamente definiu a escolha por ele. É importante ressaltar também que o Git é o sistema de controle de versão mais utilizado atualmente e portanto, caso alguém se interesse em contribuir com o projeto é necessário saber usar o Git.

Foi utilizado o GitHub[17] para hospedar os repositórios do projeto, pois ele disponibiliza hospedagem gratuita para projetos de código aberto e oferece todo um sistema de *issues* e *tasks* que facilitam a documentação do projeto e a comunicação entre a equipe.

Ao iniciar o trabalho em uma issue do projeto, era criado um *branch* a partir do *branch dev*, e a cada trecho de código funcional e significativo, era feito um *commit* para salvar o estado do repositório. Ao fim do trabalho, era feito um *push* para gravar os *commits* no servidor remoto do Github. Caso a issue fosse finalizada, era aberto um *Pull request* para aplicar no *dev* as alterações feitas. Dessa forma, foi possível manter uma organização no fluxo de desenvolvimento.

[17] https://github.com

3.3.2 Gitter

Como as equipes de desenvolvimento são geograficamente distribuídas e com horários diversos, foi necessária a adoção de alguma ferramenta de comunicação.

Inicialmente, foi utilizado o Slack[18] para comunicação entre os membros da equipe. Ele permite a criação de salas de mensagens, utilização de *bots* e integração com vários serviços, entre eles o GitHub. Porém, era necessário um convite para a entrada de novos membros (o que foi se tornando frequente conforme o crescimento do projeto), então foi decidido a mudança para o Gitter[19].

O Gitter oferece algo parecido com o Slack: sala de mensagens de texto, integração com o GitHub e permite que qualquer um com o *link* da sala possa acessá-la, facilitando a entrada de novos membros.

3.3.3 Linter

Projetos grandes e com vários programadores podem sofrer inconsistência de estilo do código, como por exemplo, identação com *tabs* ou espaços, String com aspas simples ou duplas, variáveis com camelCase ou snake_case, entre outros. Além disso, como JavaScript é uma linguagem dinâmica, é preciso executar o programa para achar erros de sintaxe.

Para solucionar esses problemas, utiliza-se um *linter* e, no caso desse projeto, o ESLint[20], que tem a função de garantir que o estilo do código seja o mesmo em todo o projeto. O ESLint foi criado por Nicholas C. Zakas em Junho de 2013, e por meio de regras (*rules*) é possível definir padrões no estilo do código, e dessa forma o programador é avisado quando não está cumprindo uma dessas regras.

O ESLint também indica erros de execução do JavaScript sem precisar executar o programa. A utilização pode ser feita executando o comando **eslint** ou adicionando um plugin/extensão ao seu editor de texto. Junto com a integração contínua, o *linter* garante uma maior qualidade de código.

Posteriormente, o linter se tornou uma ferramenta obrigatória dentro do projeto. Assim que um commit é realizado, um script de pré commit hook executa o teste de linter e só permite a efetivação do commit se os testes passarem com sucesso. Caso contrário, exibe os erros que devem ser corrigidos para que o commit possa ser finalizado.

[18] https://slack.com
[19] https://gitter.im
[20] http://eslint.org/

```
1    3:10   error   'Session' is defined but never used   no-unused-vars
2    7:1    error   Expected space(s) after "if"          keyword-spacing
3    7:20   error   Missing space before opening brace    space-before-blocks
4   12:7    error   Expected space(s) after "if"          keyword-spacing
5   16:7    error   Expected space(s) after "if"          keyword-spacing
6   16:40   error   Missing space before opening brace    space-before-blocks
7   26:30   error   Missing trailing comma                comma-dangle
8   27:27   error   Requires a space after '{'           block-spacing
9   27:27   error   Missing space before opening brace    space-before-blocks
10  27:43   error   A space is required after ','        comma-spacing
11  27:47   error   Requires a space before '}'          block-spacing
12  27:47   error   Missing semicolon                     semi
13  27:48   error   Missing trailing comma                comma-dangle
14  32:6    error   Missing trailing comma                comma-dangle
15  38:24   error   A space is required after '{'        object-curly-spacing
16  38:25   error   Extra space after key '$or'          key-spacing
17  40:2    error   Unnecessary semicolon                 no-extra-semi
```

Listing 3.1: *Exemplo de saída do ESLint*

3.3.4 CircleCI

> "Integração Contínua é uma pratica de desenvolvimento de software onde os membros de um time integram seu trabalho frequentemente. Geralmente cada pessoa realiza a integração pelo menos diariamente – podendo haver múltiplas integrações por dia. Cada integração é verificada por um *build* automatizado (incluindo testes) para detectar erros de integração o mais rápido possível. Muitos times acham que essa abordagem leva a uma significante redução nos problemas de integração e permite que um time desenvolva software coeso mais rapidamente."
>
> Fowler (2006)

Para integração contínua junto ao GitHub, utiliza-se o CircleCI[21], que é executado toda vez que alguém dá um *push* no repositório. Sua função é executar todos os testes e o *linter*, garantindo que o código adicionado siga os padrões definidos. O resultado fica disponível no *commit* do GitHub e uma mensagem é enviada no Gitter. Caso o *push* seja na *branch master*, a versão é enviada automaticamente para o servidor de produção.

3.3.5 Floobits

O Floobits[22] é uma ferramenta para codificação colaborativa que permite que várias pessoas mexam ao mesmo tempo num projeto. Pode ser utilizado a partir de um *plugin* de

[21] https://circleci.com/
[22] https://floobits.com/

um editor de texto (Vim, Sublime, Atom, entre outros) ou do seu próprio editor online, permitindo o compartilhamento de tela e o acesso a um terminal. Dessa forma, junto com um aplicativo de comunicação em voz, é possível programar em par à distância sem muitos problemas.

Para utilizar essa ferramenta, foi criado um servidor remoto na plataforma para hospedar o código do projeto. Ao iniciar o editor de texto, basta um comando para associar o diretório local com o remoto e sincronizar os arquivos. Quando um outro integrante chega, é necessário apenas ter o cuidado de sincronizar as mudanças do servidor remoto no diretório local. A partir disso, toda alteração no código se reflete em tempo real no arquivo de todos os participantes.

Capítulo 4

Métodos Ágeis

4.1 O que são métodos ágeis

Métodos Ágeis é um modelo de gestão que inicialmente foi criado para o desenvolvimento de software, mas que nos dias de hoje pode ser utilizado em projetos de qualquer área. Ao contrário da forma tradicional, o modelo ágil é focado na imprevisibilidade dentro do projeto incentivando adaptação e inspeção frequentes. Seu objetivo é garantir uma entrega rápida e de qualidade com uma abordagem focada no negócio e nas necessidades do cliente.

Em 2000, a comunidade de *Extreme Programming*, XP, debateu sua relação com os, então chamados, métodos leves (Lightweight Methods). Tais métodos vieram em contradição aos métodos pesados, que tinham como característica o foco em documentação. Como consequência, percebeu-se a semelhança entre a XP e os métodos leves e decidiu-se montar uma reunião com pessoas interessadas nessas metodologias. Essa reunião em 2001, que contou com dezessete pessoas, resultou na criação do manifesto ágil Alliance (2001), um documento onde estaria contido a declaração das crenças e valores que as pessoas presente na reunião possuíam sobre desenvolvimento de software.

O manifesto descreve a valorização de:

- **Indivíduos e interação entre eles** mais que processos e ferramentas
- **Software em funcionamento** mais que documentação abrangente
- **Colaboração com o cliente** mais que negociação de contratos
- **Responder a mudanças** mais que seguir um plano

Existem diversos tipos de metodologias ágeis, como Scrum e XP. Porém, nesse projeto não foi utilizada nenhuma metodologia específica. As práticas foram decididas pelo grupo levando em consideração os valores e princípios descritos no manifesto.

4.2 Métodos ágeis no projeto

4.2.1 Princípios

"Pessoas relacionadas a negócios e desenvolvedores devem trabalhar em conjunto e diariamente, durante todo o curso do projeto."

Desde que o grupo aceitou fazer parte do projeto, o primeiro contato foi com pessoas relacionadas ao negócio, com um certo entendimento de criptomoedas e de *blockchain*, mas pouco conhecimento de programação. Desde então, tem sido mantido contato constante por meio do mensagens e reuniões semanais com essa equipe (e com as equipes de desenvolvimento) e eles têm ajudado a tomar várias decisões de projeto e também a integrar todas as equipes envolvidas na plataforma.

"Construir projetos ao redor de indivíduos motivados. Dando a eles o ambiente e suporte necessário, e confiar que farão seu trabalho."

Por mais que sejam vários times em diversos países trabalhando no desenvolvimento e com uma variedade no volume de contribuição, o ambiente de integração proporciona um contato eficiente de modo que todos conseguem fazer sua parte e se sentem motivados a melhorar o sistema constantemente.

"Processos ágeis promovem um ambiente sustentável. Os patrocinadores, desenvolvedores e usuários, devem ser capazes de manter indefinidamente, passos constantes."

Além da diferença geográfica e de especialidade, há também diferenças de experiência e disponibilidade de tempo. Existem times dedicados ao projeto, como também times envolvidos em outros projetos paralelos. De toda forma, é importante notar que têm sido possível manter um ritmo de desenvolvimento sem muita pressão e com implementações frequentes de novas funcionalidades.

"O Método mais eficiente e eficaz de transmitir informações para, e por dentro de um time de desenvolvimento, é através de uma conversa cara a cara."

Como existe uma impossibilidade de reuniões presenciais devido à distribuição geográfica, as reuniões semanais têm sido feitas por meio de video-chamadas (Skype, Hangout ou appear.in), além do contato diário na sala de chat aberto (Gitter). As reuniões são utilizadas para ter uma visão geral sobre o que foi feito até então e sobre as próximas prioridades, enquanto que no chat são compartilhados conteúdos pertinentes ao sistema e dúvidas rápidas.

"Software funcionando é a medida primária de progresso."

O objetivo de cada entrega é garantir que as novas funcionalidades funcionem corretamente sem que as funcionalidades antigas deixem de funcionar. Dessa forma, clientes e patrocinadores sempre tem acesso a um sistema funcionando e conseguem perceber o progresso do sistema. Desde que o sistema se tornou minimamente funcional, temos um ambiente

em produção correspondente à *branch master* do repositório, que é atualizado conforme as funcionalidades da *branch dev* são verificadas. Assim, é possível ter uma visão do *release* atual do sistema.

"Nossa maior prioridade é satisfazer o cliente, através da entrega adiantada e contínua de software de valor."

Nesse projeto, os clientes são basicamente os autores do *whitepaper* que deu início à ideia da plataforma, e eles fazem parte da equipe de desenvolvimento, estando em constante contato com as novas funcionalidades assim que são implementadas. Além do contato constante pelo chat e reuniões, usamos a especificação de *issues* do Git para ter um controle das funcionalidades a serem implementadas.

"Entregar software funcionando com frequência, na escala de semanas até meses, com preferência aos períodos mais curtos."

Por enquanto não houve necessidade de estabelecer prazos de entrega para cada funcionalidade. Há um objetivo de entregar um sistema funcional pronto para o lançamento até dezembro, e a cada reunião, todos tem uma boa visão sobre o que há atualmente e o que falta para chegar em tal ponto. Como não é um projeto comercial e todos estão motivados a desenvolver algo inovador, tem-se conseguido manter um ritmo bom e evolução constante.

"Aceitar mudanças de requisitos, mesmo no fim do desenvolvimento. Processos ágeis se adequam a mudanças, para que o cliente possa tirar vantagens competitivas."

Apesar de o *whitepaper* proporcionar uma visão geral de como o produto final deve se comportar, a maior parte dos detalhes, principalmente de *front-end* são especificadas ao longo do tempo. Não existem requisitos formais pré concebidos. Eventualmente novas *issues* são colocadas no Github e os desenvolvedores buscam implementar as melhores soluções tendo em mente a forma como outras plataformas similares do mercado se comportam.

"Em intervalos regulares, o time reflete em como ficar mais efetivo, então, se ajustam e otimizam seu comportamento de acordo."

Um dos maiores pontos de reflexão que o time tem discutido é como tornar o ambiente de desenvolvimento mais eficiente. A execução de um cliente local do sistema é relativamente custosa para a máquina e isso aliado ao consumo de recursos das outras ferramentas de desenvolvimento e comunicação muitas vezes tem deixado o ambiente lento. Por conta disso, constantemente os integrantes tem pesquisado melhores formas de aumentar a eficiência nesse sentido, desde o uso de serviços externo a aquisição de novos equipamentos.

"As melhores arquiteturas, requisitos e designs emergem de times auto-organizáveis."

Dada a responsabilidade pelo *front-end*, o grupo foi capaz de encontrar por conta própria as melhores estratégias para solucionar as questões de implementação sem depender muito

dos gerentes do projeto. Aos poucos foi conquistada uma confiança de que o grupo tem utilizado as melhores técnicas para tornar o código limpo e sustentável e que existe um bom senso de design para resultar em um produto elegante. Essa confiança tem dado liberdade para estratégias criativas de boa eficiência.

"Contínua atenção à excelência técnica e bom design, aumenta a agilidade."

Buscando aplicar os conhecimentos de várias disciplinas de desenvolvimento de sistema que os integrantes tiveram durante a graduação, houve um esforço grande em manter uma beleza de código utilizando princípios como DRY (Don't Repeat Yourself) e KISS (Keep It Simple Stupid). Dessa forma, foi possível ter um bom aproveitamento de código e muitas rotinas complexas puderam ser feitas de forma simples.

"Simplicidade: a arte de maximizar a quantidade de trabalho que não precisou ser feito."

Além de boas práticas de programação, foram utilizadas algumas ferramentas para automatizar tarefas recorrentes, de modo que a preocupação ficava centrada no código. Ao executar um comando de *commit* de um trecho novo de código, um *hook* executa a verificação de *linter* antes de permitir a finalização do *commit*. Assim, o time tem uma atenção constante para padronização de código. Após o *commit*, a ferramenta de integração continua (CircleCI) automaticamente roda uma bateria de testes para garantir o funcionamento da plataforma e integra ao ambiente de produção.

4.2.2 Práticas

Mesmo sem seguir uma metodologia específica, foram utilizadas algumas práticas de Extreme Programming.

A principal delas, utilizada somente pelo nosso time foi a de **programação em par**, as vezes presencial e as vezes remotamente pelo Floobits (Subseção 3.3.5), garantindo uma maior qualidade do código e evitando *bugs*, uma que o código é revisado por duas pessoas.

Outra prática é a **integração contínua**, em que novas funcionalidade são automaticamente integradas ao sistema em produção pelo CircleCI (Subseção 3.3.4). O *linter* (Subseção 3.3.3) garante a **padronização do código**.

Capítulo 5

Desenvolvimento

Esse capítulo descreve o andamento do projeto mês a mês, onde:

Progresso do grupo: descreve detalhadamente o que foi desenvolvido pela equipe (Felipe, Hugo e Mateus);

Projeto: descreve o que foi desenvolvido por outros colaboradores;

Dificuldades: descreve as dificuldades encontradas pelo grupo durante o mês;

Estatísticas: gráficos de dados retirados do repositório do projeto.

5.1 Maio

Progresso do grupo

No o inicio do projeto, o foco da equipe foi o estudo de tecnologias e ferramentas utilizadas, em particular, o Meteor. Para isso, foi desenvolvido num repositório próprio, um *player* de vídeo que continha uma barra de navegação lateral e um botão de *play* e *pause*. Esse código foi adicionado ao projeto e a partir desse ponto as implementações foram feitas no repositório principal.

Continuou-se mexendo na parte de *front-end* do projeto, em especial, a página do *player* de vídeo. Sempre utilizando em conjunto HTML, CSS e JavaScript, foram adicionadas várias funcionalidades ao projeto, entre elas:

- Botões do *player*: como o *player* deve ser totalmente personalizado, é utilizado somente o reprodutor de vídeo do HTML. Os botões tiveram que ser adicionados individualmente ao HTML e suas ações ao JS;
- Barras de progresso e volume: as barras foram adicionadas utilizando a *tag input*. Porém houve uma incompatibilidade entre navegadores a ser corrigida no futuro;

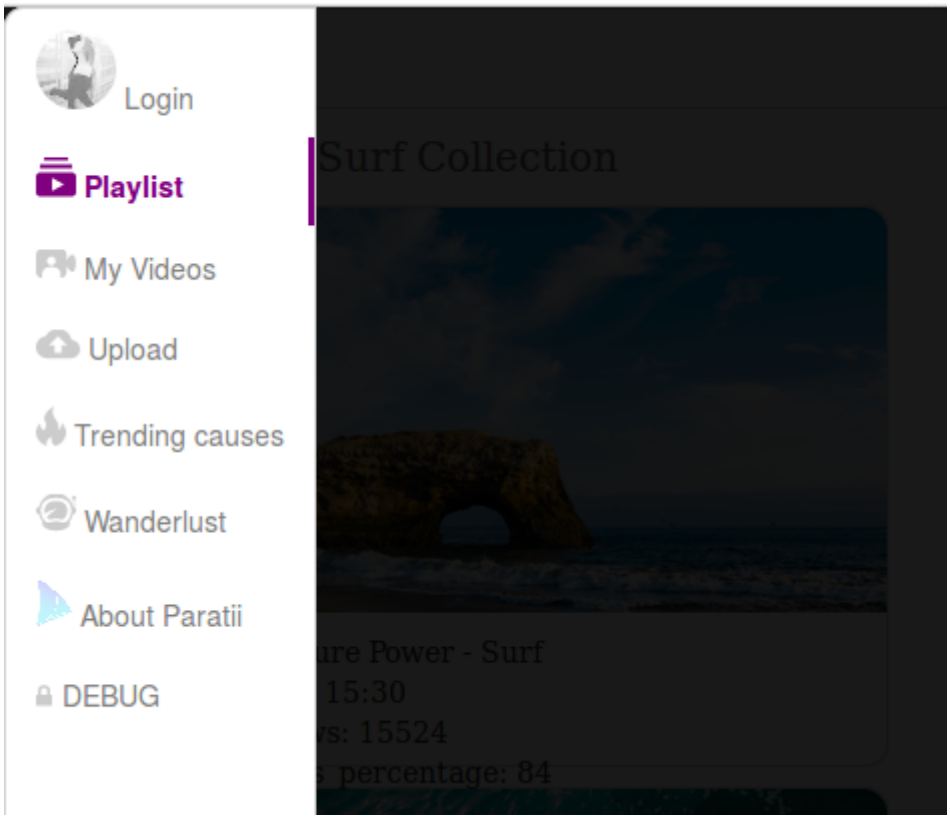

Figura 5.1: *Menu lateral maximizado*

- Comportamento da barra lateral: a barra de menu lateral tem três estados: maximizada, quando o mouse está sobre ela; fechada, quando o vídeo está sendo reproduzido; minimizada, quando apenas se mostra os ícones, caso o contrário;

- Lista de vídeos: uma grade de vídeos na página de *playlist* para acessá-los.

Para a realização dos teste de unidade, foi utilizado o Mocha[1] da biblioteca *praticalmeteor*. Os testes implementados foram os relacionados à página do *player*, em particular, os *helpers* do Meteor, que são funções que podem ser acessadas diretamente no HTML e utilizadas para acessar variáveis de forma reativa, como por exemplo, o tempo de vídeo atual, que durante a reprodução é alterado a cada segundo.

Em Meteor, exitem variáveis reativas (ReactiveVar) que quando são alteradas reativam as funções onde são acessadas. Dessa forma, junto com os *helpers* há uma comunicação assíncrona entre o cliente (HTML) e servidor (JS). Como várias variáveis reativas são utilizadas numa mesma página, usou-se um dicionário reativo para armazenar vários valores da mesma forma.

Projeto

No começo, houve um maior foco na parte de DevOps, adicionando a integração contínua com o CircleCI e logo em seguida os testes automatizados e o *linter*.

[1] https://github.com/practicalmeteor/meteor-mocha

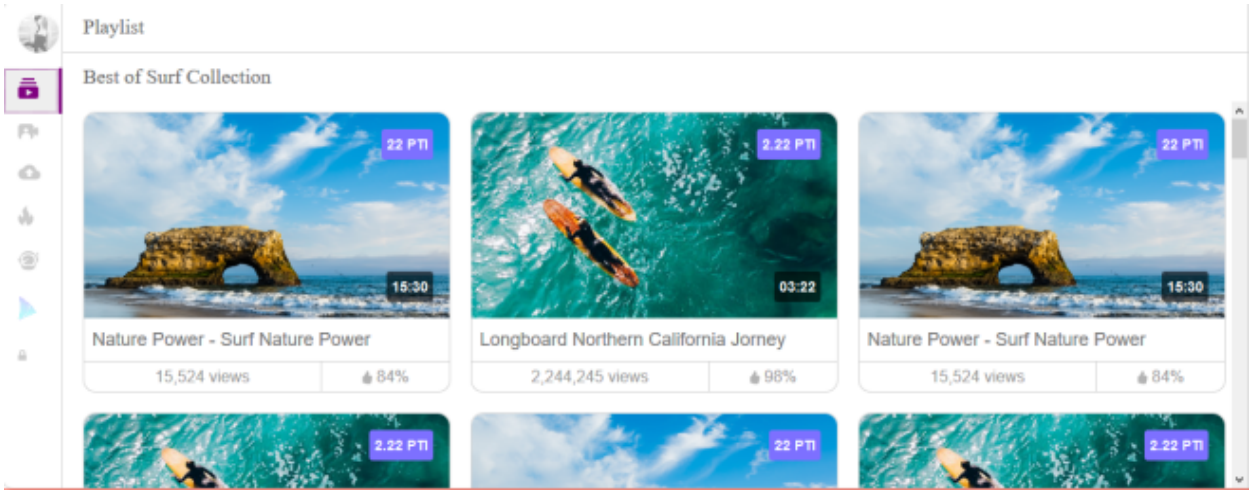

Figura 5.2: *Playlist com menu lateral minimizado*

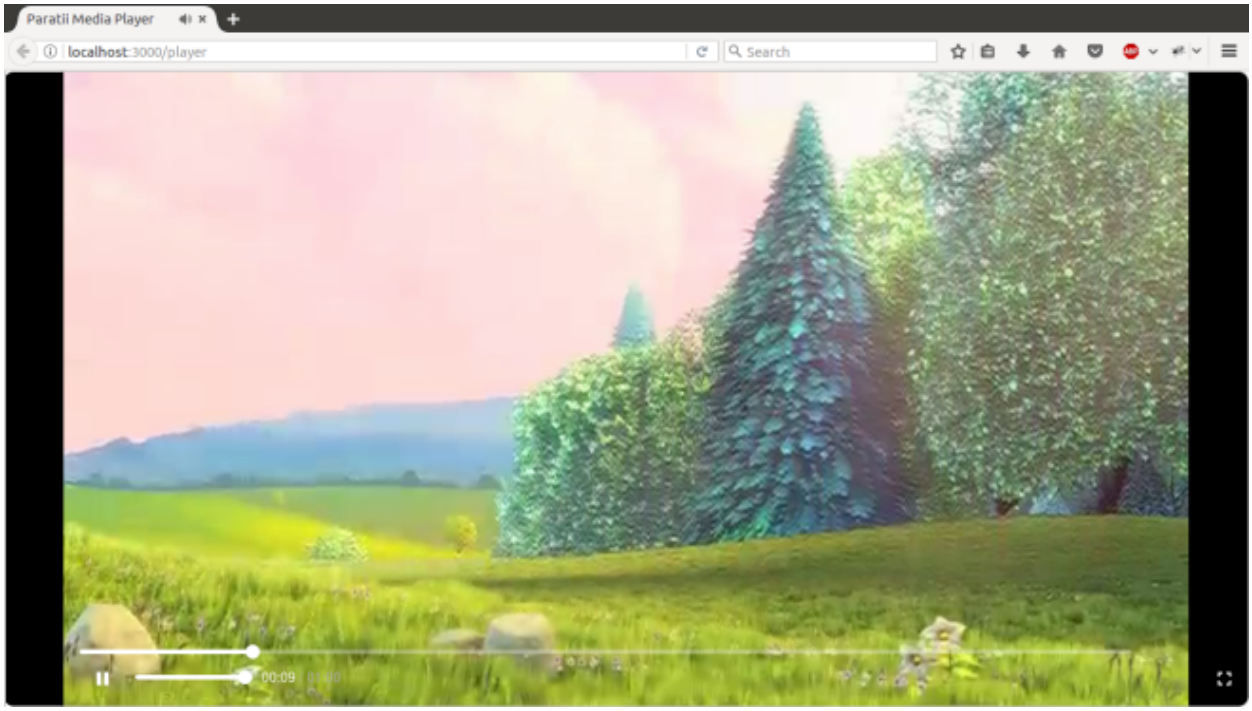

Figura 5.3: *Player com menu lateral fechado*

Após a adição do código do *player*, iniciou-se a implementação das contas de usuário, como registro e login. Também foram adicionados alguns vídeos para testes.

Dificuldades

A utilização de JavaScript como a linguagem principal mostrou-se uma dificuldade inicial do projeto pois não houve muito contato com ela durante a graduação, assim como programação assíncrona. Somando-se à utilização de um arcabouço completamente desconhecido, o resultado foi um início lento com grande necessidade de refatoração de código.

Outra linguagem pouco vista na graduação é o CSS. Pequenos ajustes na aparência da páginas demandavam bastante tempo e seu resultado era apenas satisfatório. O Paulo auxiliou algumas vezes, mas a equipe não contava com um designer dedicado.

Um problema relacionado a manipulação de imagens vetoriais (.svg) realçou a falta de conhecimento dessas duas linguagens, resultando em mais tempo de dedicação do que o necessário para essa tarefa.

A maior dificuldade enfrentada foi na parte dos testes. O Meteor não é muito agradável com testes de unidade pois ele funciona em cima de "instâncias de modelo"(de *Template Instance*), a qual dá acesso às funções JavaScript. Porém, não se tem acesso a todas as funções e variáveis, tornando impossível testar todas as situações. A dificuldade de lidar com as instâncias não foi somente do grupo, se estendendo ao projeto como um todo. Após algumas tentativas foi possível testar uma parte limitada do projeto, e por isso foram discutidas soluções para esse problema no mês seguinte.

Além dos pontos apresentados, pode-se dizer que a execução do cliente sobre o Meteor consome uma grande quantidade de memória, causando um certo incômodo durante o desenvolvimento. Diversas vezes foi necessário fechar o ambiente para liberar recursos, e algumas vezes, causou até um travamento total na máquina. Ao longo do tempo, cada um foi encontrando formas diferentes de amenizar o problema e tornar a rotina de programação mais fluída e com menos interrupções desnecessárias.

Estatísticas

Figura 5.4: *Estatísticas de Maio*

5.2 Junho

Progresso do grupo

No início do mês houve a primeira reunião com vídeo-chamada com toda a equipe de desenvolvimento via Skype. Nessa reunião ficou decidido que os próximo passos seriam a continuação do desenvolvimento do *player* e o início da implementação da carteira (*wallet*). Sobre os testes, como a maior parte do desenvolvimento está no *front-end*, decidiu-se focar em testes de aceitação.

Os testes de unidade tem como objetivo testar uma parte mínima do sistema de forma isolada. Porém, em páginas HTML com JavaScript, tudo está interligado. Assim, utilizam-se testes de aceitação, que além de testar as funcionalidades como unidade, testam a integração

entre elas e a visualização do cliente. Para isso adicionamos a biblioteca Chimp[2] ao projeto.

O Chimp é uma biblioteca que une vária ferramentas para a realização dos mais variados testes. Por exemplo, os testes podem ser escritos em Mocha, Cucumber ou Jasmine e utiliza WebdriveIO ou Selenium para testes no navegador. Além disso, Chimp tem algumas facilidades para testes em Meteor, permitindo executar comandos direto no servidor.

Após a reunião, foram necessários alguns dias para estudar o Chimp e logo em seguida foram implementados os testes de aceitação da página do *player*.

```
1 it('play the video', function () {
2     browser.url('http://localhost:3000/player/12345');
3     browser.waitForExist('#video-player');
4     browser.click('#play-pause-button');
5     assert.isTrue(browser.getAttribute('#nav', 'class').includes('closed')
          );
6     assert.isTrue(browser.getAttribute('.player-controls', 'class').
          includes('pause'));
7     assert.isTrue(browser.getAttribute('.player-overlay', 'class').
          includes('pause'));
8 });
```

Listing 5.1: *Exemplo de teste no Chimp*

Em seguida, foi necessário alterar a forma como a barra de progresso estava implementada, pois no modo em que estava, não era possível editar sua aparência. Utilizando somente *tags* <*div*> e classes CSS foi implementada uma barra de progresso que mostra tanto o momento atual do vídeo quanto o que já foi carregado.

Figura 5.5: *Barra de progresso com divs*

A mesma coisa foi feita com a barra de volume. Um funcionamento adicional do volume é que a barra fica escondida até o usuário passar o mouse em cima do ícone. Dessa forma a barra é expandida e fica visível até o usuário mover o mouse para fora da região.

Até esse momento, os vídeos do sistemas estavam armazenados na máquina local ou em uma máquina específica compartilhada por uma URL. Como o objetivo do sistema é ter uma rede *peer-to-peer* de compartilhamento dos vídeos, decidiu-se implementar a biblioteca do WebTorrent.

A implementação apresentou pouca dificuldade. A API é bem simples e poucas modificações foram necessárias. Os maiores problemas foram:

- A geração do HTML pela API entrava em conflito com os *helpers* do Meteor. Assim, a implementação inicial estava um pouco improvisada. Após um estudo mais

[2] https://chimp.readme.io/

aprofundado da API, o código ficou mais simples e com resultados melhores do que anteriormente;

- A barra de progresso precisou ser alterada para receber as informações da API e não do próprio HTML. Foi a maior mudança de código necessária para a implementação do WebTorrent;
- Como o WebTorrent utiliza WebRTC, alguns navegadores podem não dar suporte para essa tecnologia, assim como versões mais antigas de navegadores populares. O caso mais especial é o navegador Brave. Ele tem sua própria implementação do WebTorrent que sobrescreve a implementação do sistema.

No final do mês, o foco foi polir o *front-end*, em especial a página da carteira, que estava sendo desenvolvida pela equipe. As principais mudanças foram desativar alguns links da barra lateral que ainda não estavam implementados e melhorar os painéis da página da carteira e da página do usuário.

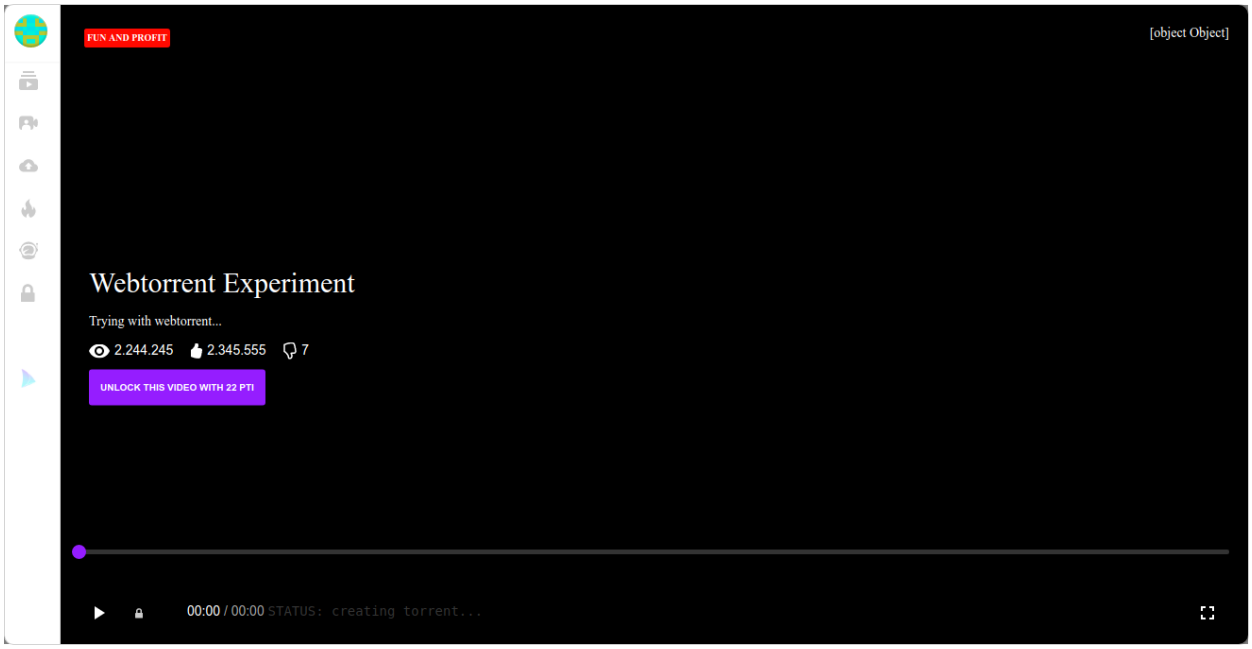

Figura 5.6: *Player com webtorrent*

Projeto

O Chimp foi adicionado ao projeto e à integração contínua. Dessa forma, todo *commit* enviado ao repositório é testado e passa por uma aprovação.

Nesse mês, houve a troca do canal de comunicação do Slack pelo Gitter. O principal motivo dessa troca é que no Slack é necessário aprovar a entrada de novos membros, dificultando a entrada de novas pessoas que queiram colaborar com o projeto de alguma forma.

O foco principal do desenvolvimento foi a criação das páginas da carteira e do usuário. Cada usuário tem uma carteira associada a endereço único que representa o usuário dentro da *blockchain*. Nesse estágio, ainda não havia uma *blockchain* implementada. Portanto, as operações da carteira eram feitas no banco de dados Mongo.

Dificuldades

Grande parte do tempo gasto desse mês foi relacionado aos testes. Mesmo utilizando o Chimp, o Meteor não é bem estruturado para testes, como são feitos em linguagens utilizadas durante o curso (Ruby on Rails, por exemplo).

Na integração com o CircleCI, como o Chimp abre uma janela do navegador, foram necessárias algumas configurações para a adição do navegador ao ambiente. Outro problema é que como são testes de aceitação feito num navegador, os teste são extremamente lentos e *timeouts* são frequentes principalmente na integração contínua.

Estatísticas

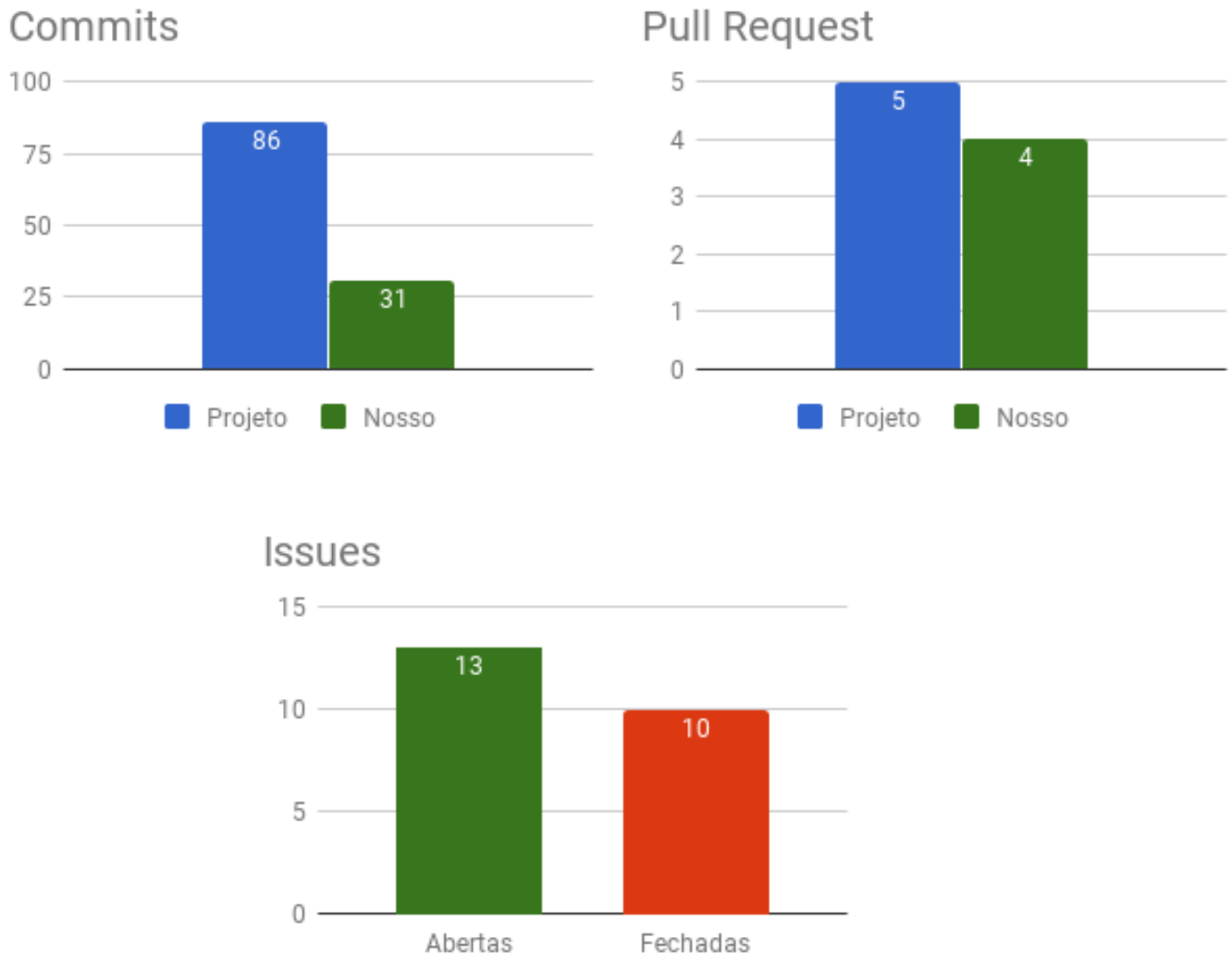

Figura 5.7: *Estatísticas de Junho*

5.3 Julho

Progresso do grupo

A parte do *player* desenvolvida foram os estados do botão de volume. Além do controle de volume em barra, há um botão mudo. Esse botão tem dois estados:

- Quando o vídeo está com som, o botão tem a aparência de um alto falante e ao ser clicado desativa o som o vídeo, altera a posição da barra de som para zero;
- Quando o vídeo está mudo, o botão tem a mesma aparência que anteriormente só que com um barra na diagonal, ao ser clicado o vídeo volta ao volume que está anteriormente assim como o controle do volume.

Figura 5.8: *Estados de volume*

Além disso, com a implementação da carteira de usuário (Veja a próxima subseção) uma grande quantidade de funções assíncronas foi adicionada ao projeto e muitas delas sem tratamento adequado.

Num primeiro momento foram adicionados retornos de chamada (callbacks) em alguns trechos de código e posteriormente houve a tentativa de adicionar Promessas(Promises) no projeto através da biblioteca Bluebird[3]. Contudo, alguns problemas aconteceram ao tentar utilizar o Bluebird em conjunto com as bibliotecas que já existiam no projeto, principalmente com a de manipulação da carteira, e a equipe optou por manter somente os retornos de chamada para o tratamento de funções assíncronas.

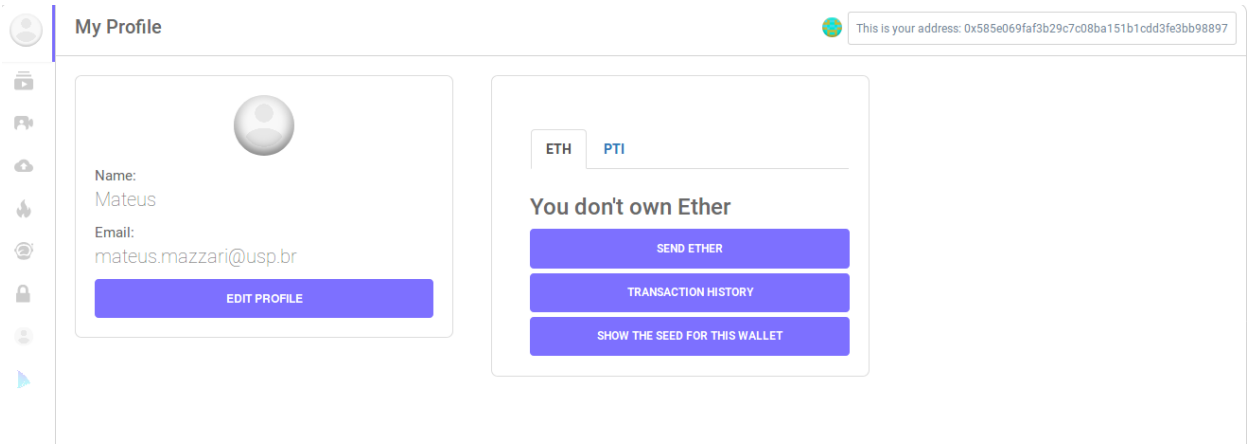

Figura 5.9: *Página da carteira/perfil do usuário*

Projeto

O foco central do projeto continuou na implementação da carteira do usuário. Para armazenar informações importantes relacionadas a carteira, foi utilizada uma *KeyStore* da biblioteca ETH-Lightwallet[4].

A *KeyStore* fica disponível somente para o usuário logado(?) e no cache do navegador, então outros navegadores e outros usuário não tem acesso a essa *KeyStore*. Quando um usuário entrar na sua conta em outro navegador, não haverá nenhuma carteira ligada a

[3] http://bluebirdjs.com
[4] https://github.com/ConsenSys/eth-lightwallet

conta. Para recupera-la é necessário inserir doze palavras disponibilizadas durante a criação da carteira, chamadas de *seed*.

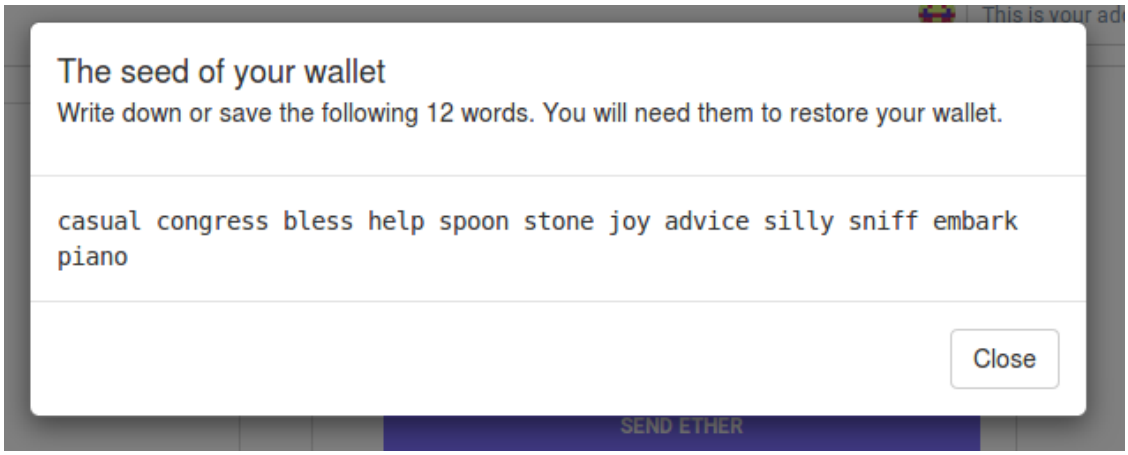

Figura 5.10: *As 12 palavras para recuperar a conta*

Também, iniciou-se a integração com a *blockchain*. Na carteira é possível enviar ether (ETH) e paratii (PTI) para um endereço escrito pelo usuário. Na parte de ETH, a biblioteca web3 já tem as principais funcionalidades, como retornar o balanço atual da carteira e enviar ETH para outras pessoas. Para o PTI, foi necessário criar os próprios contratos com a linguagem Solidity, e a integração deles com o sistema foi feito com a mesma biblioteca web3, pois o PTI é uma ERC-20.

```
1 contract ParatiiToken is StandardToken {
2     string public name = "Paratii";
3     string public symbol = "PTI";
4     uint public decimals = 18;
5     uint public INITIAL_SUPPLY = 21000000 * (10**decimals);
6
7     function ParatiiToken() {
8       totalSupply = INITIAL_SUPPLY;
9       balances[msg.sender] = INITIAL_SUPPLY;
10    }
11 }
```

Listing 5.2: *Exemplo de código do Smart Contract*

Dificuldade

Como dito anteriormente a maior dificuldade se deu ao integrar o Bluebird com as bibliotecas que já estavam no projeto e usavam retornos de chamada por padrão, dificultando um pouco o encadeamento de chamadas assíncronas e o tratamento de erros.

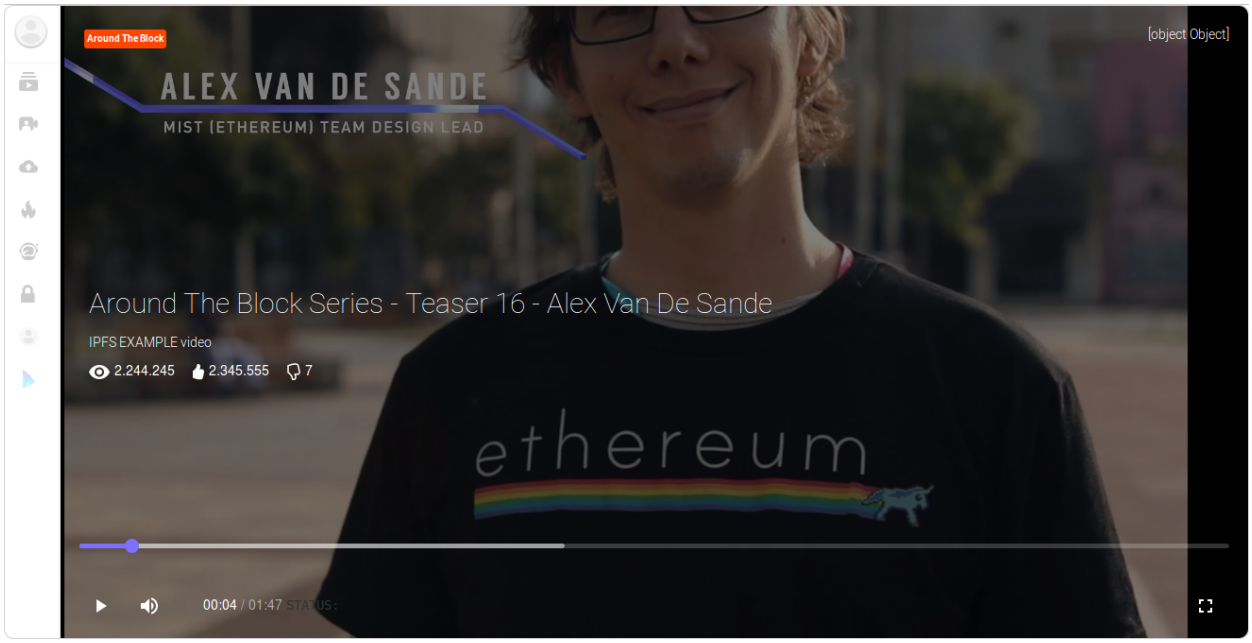

Figura 5.11: *Player em julho*

Estatísticas

Figura 5.12: *Estatísticas de Julho*

5.4 Agosto

Progresso do grupo

O foco foi a correção de dois *bugs* relacionados à *seed* da conta, que são as doze palavras utilizadas para recuperação. O primeiro era durante a recuperação da carteira. A inserção de uma *seed* inválida não devolvia nenhum *feedback* para o usuário. Como o *callback* da função retornava um erro nesse caso, adicionou-se um tratamento para esse erro apresentando na tela do usuário uma mensagem explicando que a *seed* era inválida.

O segundo ocorria na hora de criar uma conta nova. Caso utilizasse um *email* já cadastrado, a conta não era criada, mas um novo endereço e *seed* eram ligados a esse *email*. Para a solução desse *bug*, utilizou-se a prática do TDD (*Test Driven Development*).

Primeiro, garantiu-se que não teriam duas contas com o mesmo *email* e isso é indicado ao usuário durante o registro. O próximo teste garante que ao falhar na criação da conta com um *email* em uso, o endereço da carteira não seja sobrescrito.

```
1 it('try to register a new account with an used email', function () {
2     server.execute(createUser);
3     browser.url('http://localhost:3000/profile');
4     browser.waitForExist('#at-signUp');
5     browser.$('#at-signUp').click();
6     browser.waitForExist('[name="at-field-name"]');
7     browser
8       .setValue('[name="at-field-name"]', 'Guildenstern')
9       .setValue('[name="at-field-email"]', 'guildenstern@rosencrantz.com')
10      .setValue('[name="at-field-password"]', 'password')
11      .setValue('[name="at-field-password_again"]', 'password');
12    browser.$('#at-btn').click();
13    browser.waitForVisible('.at-error', 2000);
14    const error = browser.getText('.at-error');
15    assert.isNotNull(error, 'should exist a error message');
16    assert.equal(error, 'Email already exists.');
17 });
```

Listing 5.3: *Código de um dos testes*

A única coisa que não testada é o desempenho, que é um dos causadores do *bug*, pois a criação da carteira inicia-se antes da confirmação do cadastro. O outro problema é que a criação da *keystore* utiliza a senha do usuário e ela só é acessível antes da confirmação do cadastro.

A solução foi dividir o processo em duas partes: criação e armazenamento da *keystore*. A criação da *keystore* continuou antes da confirmação do cadastro para otimizar o desempenho e para ter acesso à senha do usuário. A mudança ocorreu no armazenamento.

O armazenamento da *keystore* só ocorre depois da confirmação do cadastro, ou seja, o *email* só é ligado a essa nova *keystore* caso o cadastro ocorra com sucesso.

No final do mês, adicionou-se uma indicação se o vídeo foi comprado na página de *playlists* com a utilização de um *check icon* no vídeo.

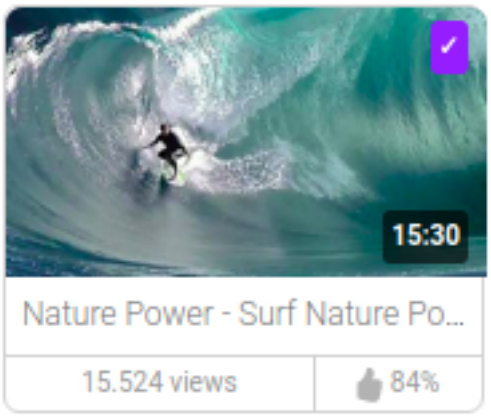

Figura 5.13: *Vídeo com o check de comprado*

Projeto

Uma funcionalidade adicionada nesse mês foi uma página que lista as transações do usuário. Tanto essa página como alguns *bugs* encontrados mostraram que a integração com a *blockchain* não estava completamente correta. Boa parte da equipe se dedicou a melhorar essa integração e seus testes.

A grande novidade desse mês foi a adição do Yahya à equipe. Ele ficou responsável por adicionar o IPFS ao sistema. Essa integração só é possível porque ele colabora tanto na comunidade do Paratii como na do IPFS. Nesse mês, somente adicionou a capacidade de assistir ao vídeo, sem a capacidade de *upload*.

Dificuldades

A maior dificuldade foi entender o funcionamento e o código do cadastro do usuário, pois essa parte foi feita por outras pessoas. O melhor jeito encontrado foi tentar cobrir a maior parte dos casos com testes para que qualquer alteração no código não interferisse no funcionamento esperado.

Os testes com as funcionalidades básicas já estavam implementados, então foram adicionados os testes para os problemas em questão. Esse foi o principal motivo de utilizar-se TDD nessa tarefa, pois não se sabia a total dimensão que as alterações podiam causar. Essa tarefa mostrou a importância dos testes automatizados em projetos colaborativos.

Estatísticas

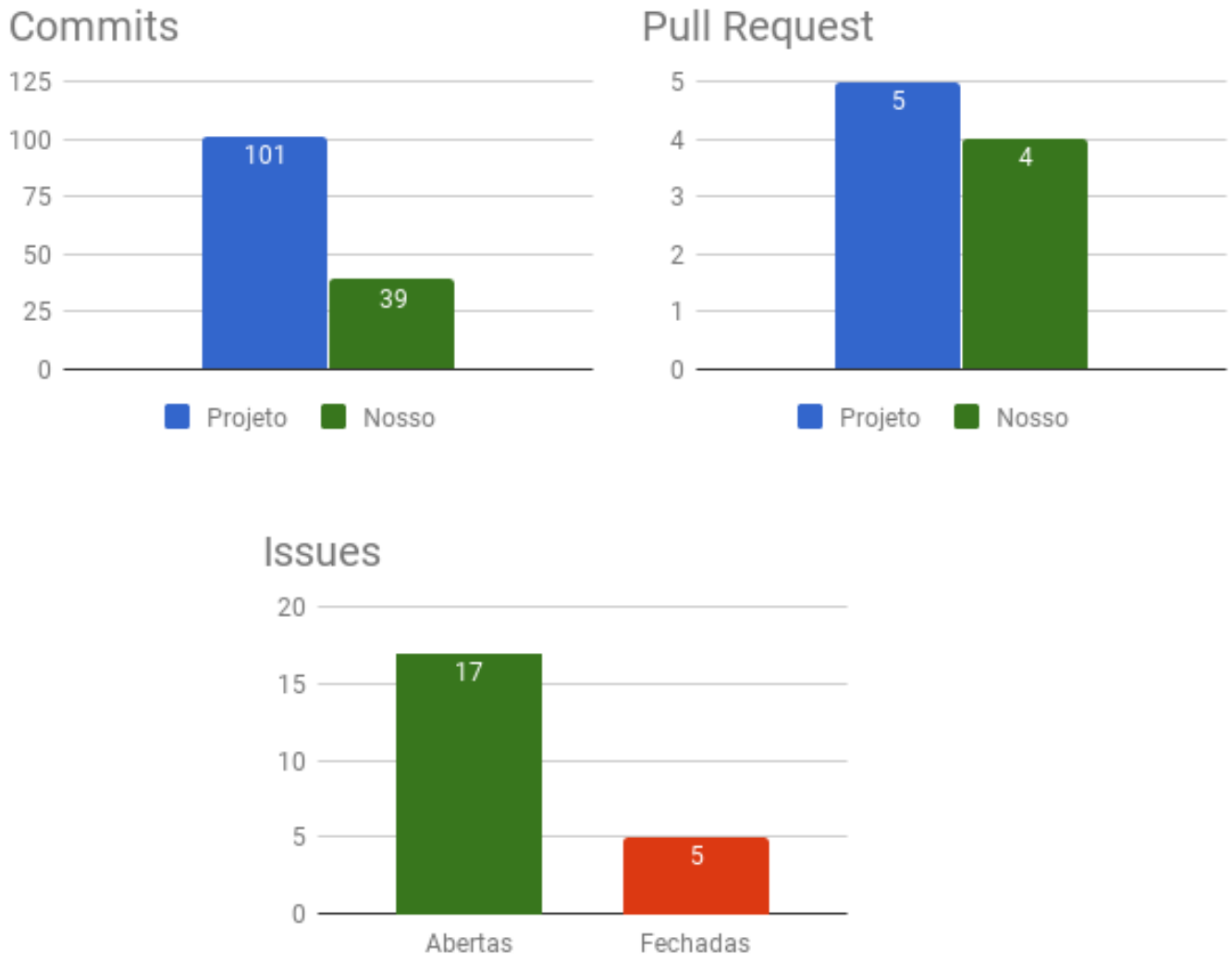

Figura 5.14: *Estatísticas de Agosto*

5.5 Setembro

Progresso do grupo

Continuando a tarefa da indicação se o vídeo foi comprado, foram implementados três testes:

1. Não deve mostrar nenhum preço quando o vídeo é gratuito;
2. Deve mostrar o preço do vídeo numa *tag* quando o vídeo é pago;
3. Deve mostrar um *check* quando o vídeo já foi comprado.

Com esses testes e a refatoração para se adequar ao novo linter, essa tarefa foi finalizada e um *pull request* realizado.

Após a implementação do IPFS, surgiram alguns *bugs* e aproveitou-se para consertar alguns comportamentos indesejados, como:

- Vídeos do IPFS iniciando automaticamente;
- Botão de *pause* não aparecendo;
- WebTorrent não funcionando;
- A barra lateral deixou de aparecer quando o vídeo está pausado;
- Quando o vídeo fica *fullscreen*, ocorre uma animação indesejada.

Outra funcionalidade adicionada pelo grupo foi a adição de dois botões para percorrer a *playlist* no *player*. Um botão passa para o próximo vídeo, enquanto o outro muda para o vídeo anterior da *playlist*. Além disso, caso o botão de vídeo anterior seja acionado no meio da reprodução, ao invés de mudar de vídeo, o atual é reiniciado.

Para isso, foi necessário informar qual a *playlist*, pois um mesmo vídeo pode pertencer a várias *playlists*. O modelo adotado foi passar a *id* da *playlist* na URL como um *Query Parameter* que são os dados após o **?** na URL e não são obrigatórios.

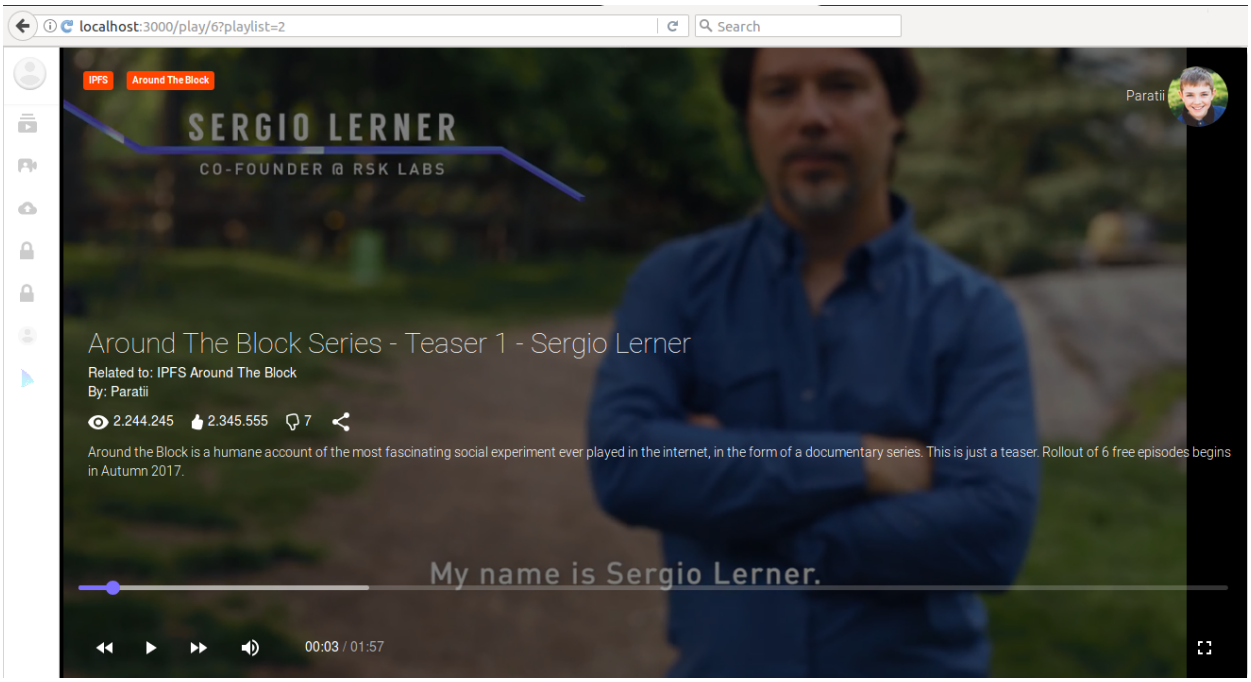

Figura 5.15: *Presença dos dois botões quando a URL informa a playlist*

Projeto

O projeto teve três focos durante esse mês:

1. Implementação do IPFS: focando na otimização, pois a inicialização do vídeo é muito demorada;

2. *Embedded Player*: o *player* acoplável é um dos focos do sistema. Para a implementação dele seriam alguns ajustes na aparência e no comportamento do sistema inteiro. Algumas dessas mudanças são necessárias para cumprir requisitos de sites para permitir que o *player* seja acoplado;

3. Transição do armazenamento para contratos: todo armazenamento das informações estava no MongoDB, que é um banco de dados centralizado. Iniciou-se a implementação de *smart contracts* para que esses dados fiquem armazenados na Blockchain.

O Pedro iniciou uma total remodelagem do design do sistema, repensando páginas e fluxos.

Dificuldades

Esse mês não apresentou grandes dificuldades técnicas. O desconhecimento de algumas funcionalidades do Meteor foi o que mais atrapalhou, principalmente na implementação do ícone de compra de vídeo.

Nessa fase, foi necessário utilizar boa parte do tempo disponível para a escrita desta monografia, diminuindo a contribuição do grupo para o projeto.

Estatísticas

Figura 5.16: *Estatísticas de Setembro*

5.6 Outubro

Progresso do grupo

Com maior dedicação na monografia, o foco nesse mês foi em alguns pequenos polimentos. O primeiro deles foi na implementação do *player* acoplável. O vídeo sempre iniciava automaticamente, mesmo para vídeos não comprados, e a barra lateral não estava sumindo quando o vídeo era executado.

Primeiro, o tratamento da opção *autoplay* na URL foi corrigido. Isso estava fazendo com que todos os vídeos começassem automaticamente. Em seguida, foi necessário bloquear o vídeo até a resposta da chamada assíncrona que verifica se o vídeo é gratuito ou está desbloqueado. O problema da barra lateral era uma sobrescrita do estado dela no *player*

acoplável.

A segunda correção foi na página de perfil/carteira do usuário que informava incorretamente que o usuário estava sem fundos enquanto a Blockchain era inicializada. Após consulta com o Jelle, os estados e suas respectivas mensagens são:

- Não conectado: *"Not connected to the blockchain"*;
- Conectado:
  - Calculando montante: *"Connecting to blockchain"*;
  - Carteira vazia: *"You don't own Ether"*;
  - Carteira com X ethers: *"You own X Ether"*.

Projeto

As maiores mudanças foram na aparência do sistema. Paulo foi responsável por criar um CSS próprio para a plataforma, deixando, assim, de utilizar o Bootstrap. Com isso, a aparência da plataforma recebeu um toque único e personalizado.

As otimizações do IPFS continuaram sendo feitas, focando principalmente na inicialização do vídeo e na duplicação de dados. Em relação ao *player* acoplável, dedicou-se a preparar o *player* para ser aceito em vários sites. Notou-se que uma refatoração deveria ser feita para tratar eventos JavaScript padrões.

Uma nova funcionalidade inicialmente implementada foi o tratamento de usuário não registrado, chamado de "usuário anônimo". Dessa forma, ao entrar no sistema, um endereço da *blockchain* é criado e as interações com o sistema são feitas por meio desse endereço. Quando o usuário criar uma conta, esse endereço é ligado a essa nova conta. Com isso, o usuário não precisa se cadastrar para utilizar grande parte das funcionalidades.

Dificuldades

Como as tarefas realizadas foram razoavelmente simples e como o grupo já está acostumado com o sistema, não houve nenhum dificuldade na parte técnica. O mais difícil foi conciliar o foco na escrita desse documento com as tarefas de desenvolvimento e reuniões.

Estatísticas

Figura 5.17: *Estatísticas de Outubro*

Capítulo 6

Funcionalidades

As funcionalidades descritas a seguir são as implementadas até o dia 01/11/2017, funcionalidades podem ter sido adicionadas ou removidas após essa data. A versão mais nova pode ser visualizada no site do Paratii[1]

6.1 Player

O *player* tem as funções básicas implementadas pelo grupo, como *play*, *pause*, barra de volume, botão mudo, barra de progresso, botão de tela cheia, botão de próximo vídeo e de vídeo anterior. Nessa versão, o IPFS é utilizado para armazenar os vídeos e informa a velocidade de download em vermelho.

Figura 6.1: *Vídeo pausado no player*

Enquanto o vídeo está sendo reproduzido, a barra lateral de navegação é recolhida, as informações do vídeo desaparecem e, caso não tenha nenhum movimento no *mouse*, os controles do *player* também desaparecem.

[1] http://player.paratii.video

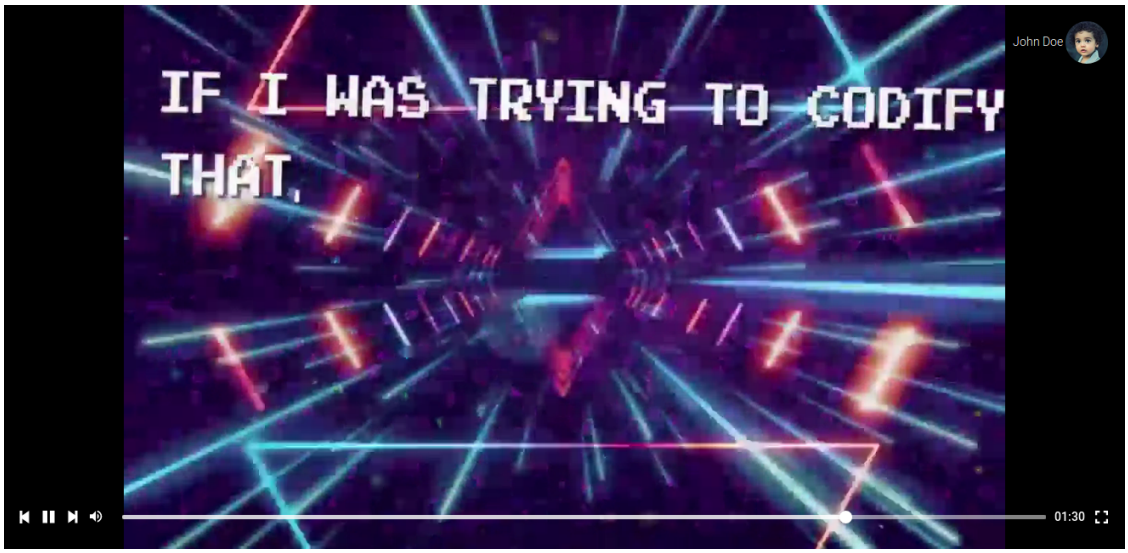

Figura 6.2: *Vídeo sendo reproduzido no player*

Também é possível comprar um vídeo dependendo da forma de monetização que o produtor escolher. Para realizar a compra é necessário possuir PTI e ETH. O PTI representa o valor de compra do vídeo, enquanto o ETH é necessário para realizar a transação.

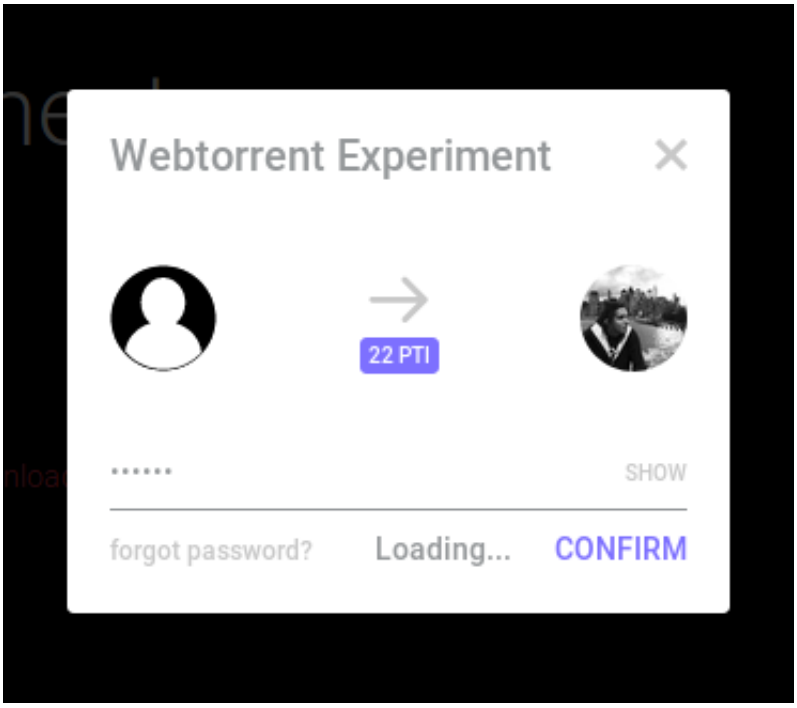

Figura 6.3: *Compra de vídeo*

6.2 Profile

O fluxo de criação de conta está completo. Após a criação, o usuário recebe as 12 palavras (*seed*) para recuperação de conta, recebendo o devido aviso de segurança.

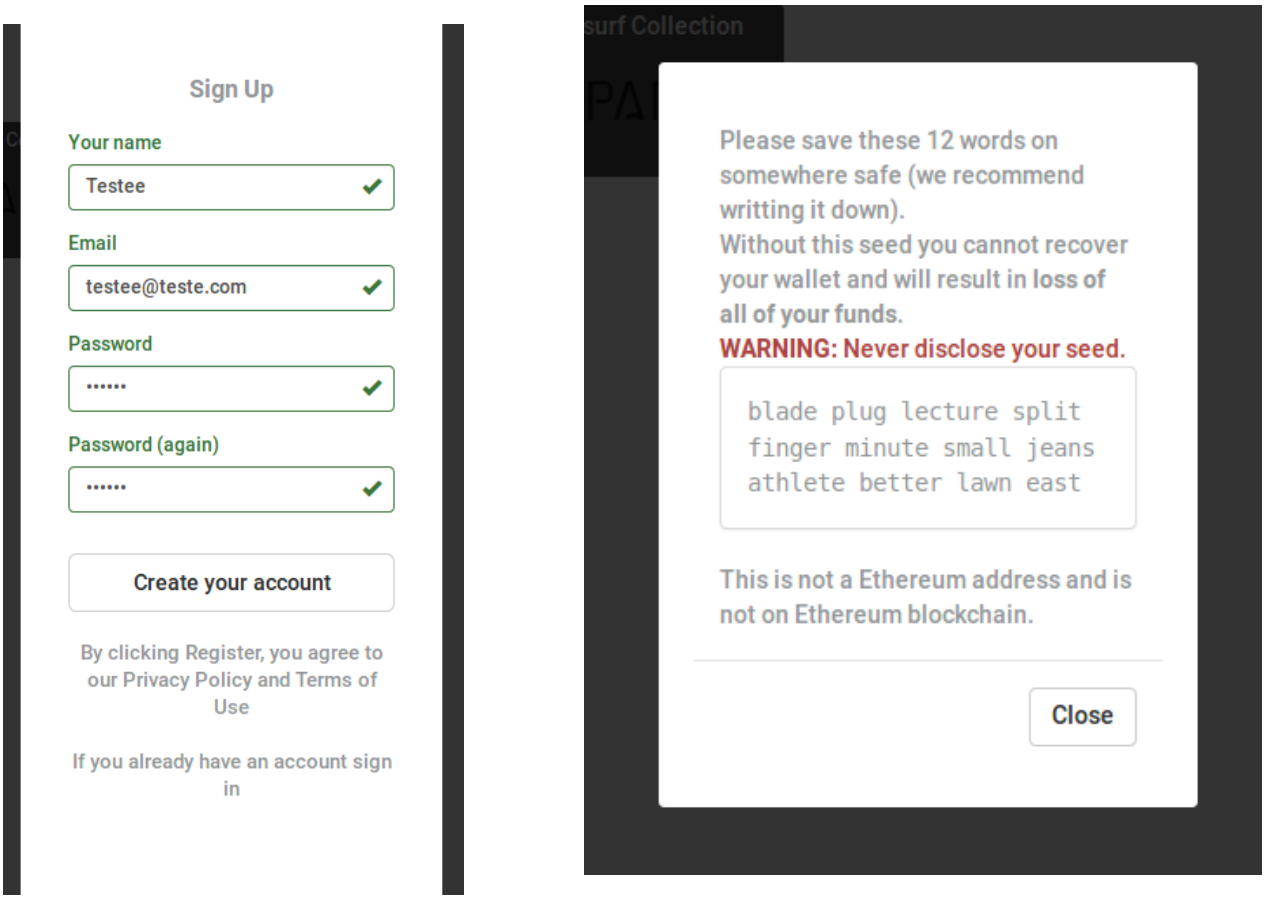

Figura 6.4: *Fluxo de criação de conta*

A página de perfil/carteira pode ser acessada a partir do primeiro ícone do menu lateral. Nessa página o usuário pode visualizar o seu endereço da *blockchain*, editar sua imagem de perfil, transferir ETH e PTI, verificar o histórico de transações e acessar as palavras pra recuperação de conta (*seed*). Quase todas essas funcionalidades pedem uma nova verificação de senha.

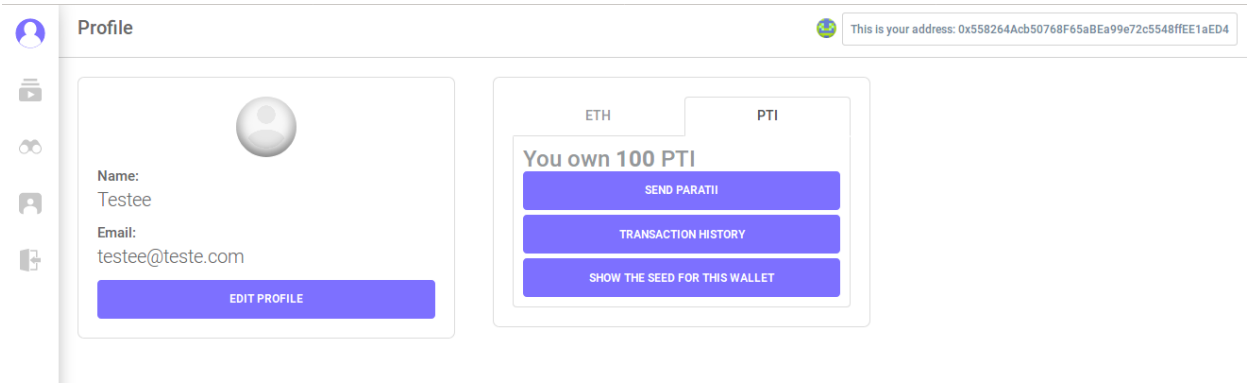

Figura 6.5: *Carteira*

O último ícone do menu lateral permite que o usuário encerre a sessão (*logout*).

6.3 Playlist

A partir do segundo ícone do menu, pode-se acessar a lista de *playlists* que aparece em grade informando o nome e quantidade de vídeos. Ao selecionar uma das opções, é apresentada uma página parecida com a grade dos vídeos da *playlist*, informando os nomes e o preço, quando houver.

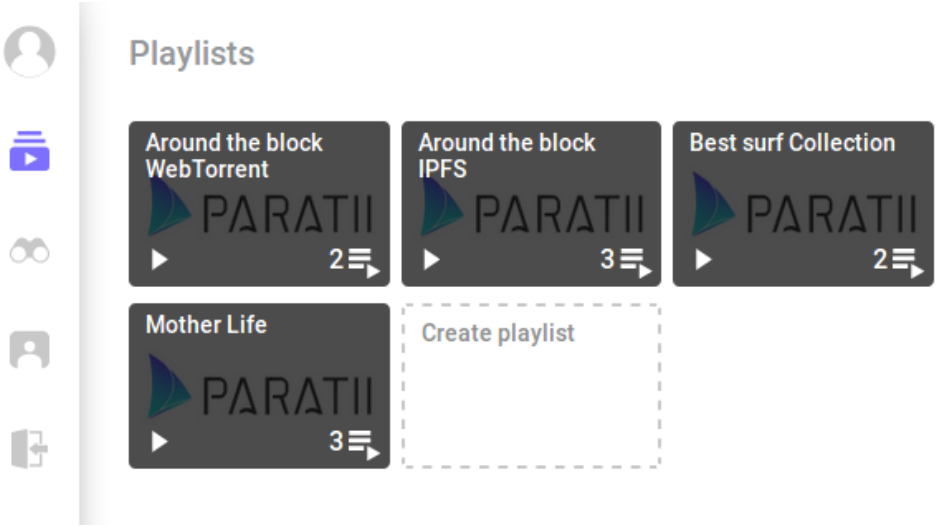

Figura 6.6: *Lista de playlists*

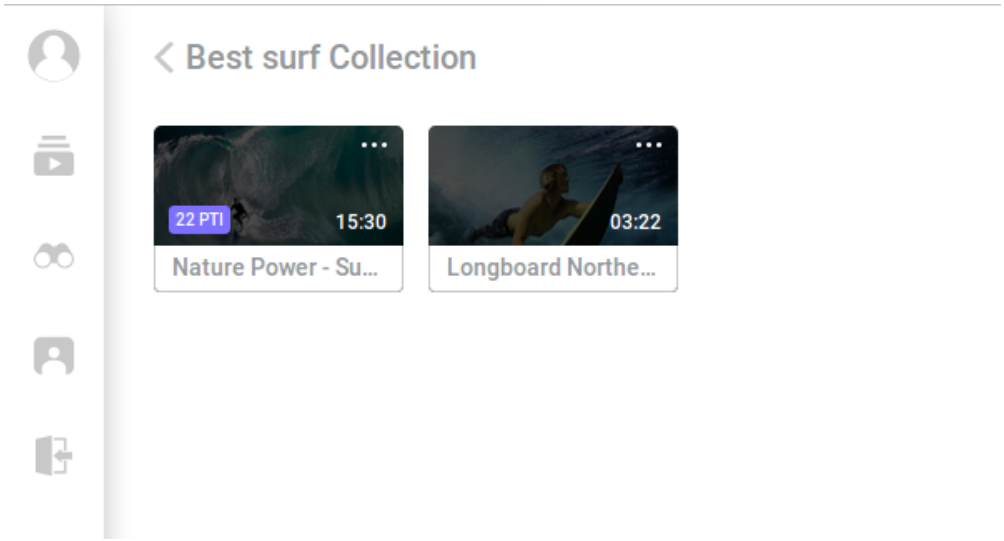

Figura 6.7: *Primeira: lista de playlists; Segunda: lista de vídeos de uma playlist*

Capítulo 7

Conclusão

7.1 Feedback

Abaixo o *feedback* (em inglês) de Felipe Sant'Ana, um dos co-fundadores do Paratii, o mais próximo de um cliente desse projeto:

"*The dedication of the team was nice and positively surprising many times (it's not everyday you get into github and there's contributions sent over the weekend from young boys you didn't know until months before). Guys were 100% available whenever we needed them, and I'm sure they have dedicated themselves beyond the hours in many occasions.*

They bought more than expected to the project. I thought the scope of collaboration would mostly include an analysis of the work done, or helping organize the project workflows, but the group was present in many of the calls and contributed to the codebase with bug fixes, code cleaning and some feature implementations with varying intensity. Mateus was rarely absent and has done some meaningful contributions to meteor development and user experience in general - I have openly offered "research grants"along the journey and would happily hire him by the way.

Maybe the team could be a bit more proactive in the sense of recognising the strengths of each and splitting up tasks as for delivering more holistic/broad contributions, i.e. work as a semi-autonomous cell."

Para uma visão mais técnica, Jelle Gerbrandy, co-fundador e líder tecnológico deu o seguinte *feedback*:

"*The team was always available if university obligations did not get into the way, so that was very satisfactory. The team actually contributed quite a bit: 161 commits and almost 4000 lines of code.*

There were no real big flaws in the team, but, given their limited experience, they needed close management, something that we have probably not provided enough. The team knowledge is satisfactory and brought quite a bit to development; as you can see the statistics above - they have helped to move the project forward considerably in its initial stages."

7.2 Conclusão

Foi muito proveitoso trabalhar em um projeto grande e de aplicação real. Apesar de a graduação oferecer várias disciplinas relacionadas ao desenvolvimento de sistemas, foi necessário muito auto-aprendizado e trabalho em equipe para implementar de forma fluída as funcionalidades delegadas ao time. Por outro lado, o conhecimento base que os integrantes adquiriram no curso permitiu que as novas tecnologias fossem utilizadas de forma disciplinada e consciente.

Um dos maiores aprendizados foi a integração conjunta de vários times de desenvolvimento. Graças à experiência dos outros desenvolvedores e da aplicação de boas práticas de projeto, foi possível organizar tudo de modo que as tarefas ficassem bem divididas e com pouca dependência externa, permitindo a integração de times com ritmos e estilos diferentes. O uso de ferramentas inteligentes também foi fundamental para ter um ambiente de desenvolvimento otimizado e integrado.

As implementações eram mais complexas do que o grupo estava habituado em EPs convencionais e a organização dentro da equipe foi essencial para conseguir desenvolver implementações de qualidade. O código das funcionalidades passava por constante evolução e refatoração para atender a novos requisitos que foram surgindo, o que demandou bastante trabalho e estudo. A aplicação dos métodos ágeis pela equipe possibilitou a solução de vários desafios de forma organizada. A programação pareada, em especial, se mostrou muito eficiente na produção de código eficiente e legível.

Nos momentos em que o rendimento caia, principalmente em época de provas e trabalhos da graduação, as reuniões foram vitais para ter uma visão geral do que estava sendo desenvolvido nas outras áreas do projeto e gerar motivação para tentar avançar, mesmo que um pouco, no desenvolvimento do *front-end*. Apesar de nem sempre haver uma conciliação de horários, o grupo procurou manter sempre algum integrante presente nessas chamadas, pois era o ponto de partida para tomadas de decisões importantes e ideias de novas funcionalidades.

A equipe gostou bastante do que o projeto se propõe a fazer e logo de início, todos quiseram fazer parte disso. Conforme o projeto foi crescendo, foi se tornando cada vez mais gratificante ver algo desenvolvido na graduação se tornando um serviço real e com grande potencial de inovação no mercado. O Paratii tomou uma proporção bem grande, com investidores reais e várias pessoas interessadas em contribuir. Levando em conta os *feedbacks* dos outros membros da equipe, ficou claro a capacidade dos alunos do Bacharelado em Ciência da Computação de se adaptar e aprender novas ferramentas que surgem a todo momento e transformar ideias em produtos concretos.

7.3 Disciplinas relevantes

MAC0242 - Laboratório de programação II: primeiro desenvolvimento de um projeto de médio prazo feito em Ruby on Rails, além do primeiro contato com métodos ágeis. A implementação de um sistema de porte maior do que o costume foi o ponto principal da disciplina;

MAC0439 - Laboratório de Bancos de Dados: contato com MongoDB e outros bancos de dados NoSQL, facilitando a utilização do mesmo nesse projeto;

MAC0472 - Laboratório de Métodos Ágeis: desenvolvimento de um projeto grande em grupo utilizando práticas de *eXtreme Programming*, o Paratii foi um dos possíveis projetos desse ano e foi por meio dessa disciplina o primeiro contato.

Referências Bibliográficas

Alliance(2001) The Agile Alliance. Manifesto for agile software development. http://agilemanifesto.org/, 2001. Citado na pág. 21

Baliga(2017) Dr. Arati Baliga. Understanding blockchain consensus models. Relatório técnico, Persistent Systems Ltd. Último acesso em 25/10/2017. Citado na pág. 15

Benet(2014) Juan Benet. Ipfs - content addressed, versioned, p2p file system. https://github.com/ipfs/papers/raw/master/ipfs-cap2pfs/ipfs-p2p-file-system.pdf, 2014. Último acesso em 26/11/2017. Citado na pág. 8

Buterin(2014a) Vitalik Buterin. Ethereum development tutorial. https://github.com/ethereum/wiki/wiki/Ethereum-Development-Tutorial, Julho 2014a. Último acesso em 26/11/2017. Citado na pág. 12

Buterin(2014b) Vitalik Buterin. Ethereum whitepaper. https://github.com/ethereum/wiki/wiki/White-Paper, Setembro 2014b. Último acesso em 26/11/2017. Citado na pág. 13

Caro *et al.***(2017)** Diego Di Caro, Jelle Gerbrandy, Felipe Sant'Ana e Paulo Perez. Paratii whitepaper. http://paratii.wpengine.com/wp-content/uploads/2017/03/paratii-whitepaper-16-mar.pdf, Março 2017. Último acesso em 02/08/2017. Citado na pág. 3

Coleman e Greif(2015) Tom Coleman e Sacha Greif. *Discover Meteor*. Versão em Português gratuita. Citado na pág. 6

Fabian Vogelsteller(2015) Vitalik Buterin Fabian Vogelsteller. Erc-20 token standard. https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20-token-standard.md, 2015. Último acesso em 13/09/2017. Citado na pág. 4

Fowler(2006) Martin Fowler. Continuous integration. https://martinfowler.com/articles/continuousIntegration.html, Maio 2006. Último acesso em 11/08/2017. Citado na pág. 19

Git() Git. Git site. https://git-scm.com/. Último acesso em 09/08/2017. Citado na pág. 17

International(2015) Ecma International. Standard ecma-262. https://www.ecma-international.org/ecma-262/6.0/, 2015. Citado na pág. 6

Kasireddy(2017) Preethi Kasireddy. How does ethereum work, anyway? https://medium.com/@preethikasireddy/how-does-ethereum-work-anyway-22d1df506369, Setembro 2017. Último acesso em 25/11/2017. Citado na pág. 10

Rauschmayer(2014) Dr. Axel Rauschmayer. *Speaking JavaScript*. Versão online gratuita. Citado na pág. 6