CENTRO ESTADUAL DE EDUCAÇÃO TECNOLÓGICA PAULA SOUZA

Faculdade de Tecnologia Rubens Lara

Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas

HUGO BESSA SILVA DE OLIVEIRA

SISTEMA DE EDIÇÃO COLABORATIVA POR P2P UTILIZANDO PARADIGMA DE PROGRAMAÇÃO FUNCIONAL

Santos, SP

2016

HUGO BESSA SILVA DE OLIVEIRA

SISTEMA DE EDIÇÃO COLABORATIVA POR P2P UTILIZANDO PARADIGMA DE PROGRAMAÇÃO FUNCIONAL

Trabalho de Conclusão de Curso apresentado à Faculdade de Tecnologia Rubens Lara, como exigência para a obtenção do título de Tecnólogo em Análise e Desenvolvimento de Sistemas.

Orientador: Prof. Me. Alexandre Garcia de Oliveira

Santos, SP

2016

HUGO BESSA SILVA DE OLIVEIRA

SISTEMA DE EDIÇÃO COLABORATIVA POR P2P UTILIZANDO PARADIGMA DE PROGRAMAÇÃO FUNCIONAL

Trabalho de Conclusão de Curso apresentada à Faculdade de Tecnologia da Baixada Santista, como parte dos requisitos para a obtenção do Título de Tecnólogo em Análise e Desenvolvimento de Sistemas.

Aprovado em ____ de _____ de 20__

BANCA EXAMINADORA:

_____
Alexandre Garcia de Oliveira
Titulação:

_____
Nome do examinador:
Titulação:

_____
Nome do examinador:
Titulação:

Local: Faculdade de Tecnologia da Baixada Santista

Dedico esse trabalho à minha família e à minha noiva pelo grande apoio que me permitiu evoluir cada dia mais durante a minha vida. Vocês me fazem querer ir mais longe.

AGRADECIMENTOS

A Deus por minha saúde, família e amigos.

À minha noiva Jéssica Isabelle Ribeiro por todo apoio, incentivo e dedicação essenciais para a realização deste trabalho.

À minha família pelo incentivo constante aos estudos desde os meus primeiros anos de vida.

Ao Prof. Me. Alexandre Garcia de Oliveira, pela orientação que elevou a qualidade deste trabalho.

A todos os professores e pessoas que fizeram parte da minha formação, direta ou indiretamente.

“Coisas simples devem ser simples, coisas complexas devem ser possíveis.”

(Alan Kay, tradução nossa)

RESUMO

Com o crescente uso de dispositivos móveis para acessar a *web*, é importante que *softwares* adotem modelos de Forte Consistência Eventual para se manterem responsivos e consistentes — principalmente em redes com grande latência. Sistemas *peer-to-peer* permitem que clientes se comuniquem diretamente, potencialmente diminuindo a latência e removendo a necessidade de um servidor centralizado. Utilizando metodologias ágeis de desenvolvimento de software, validou-se a viabilidade implementação de um Sistema de Edição Colaborativa por *peer-to-peer* aplicando paradigma de programação funcional. Observou-se que a utilização do paradigma de programação funcional foi importante para garantir a qualidade do produto-final em um ambiente de desenvolvimento ágil e o modelo de Forte Consistência Eventual é adequado para um Sistema de Edição Colaborativa. Também constatou-se a possibilidade de implementar dado sistema utilizando tecnologias disponíveis em navegadores *web*.

Palavras-chaves: Sistema de Edição Colaborativa. Forte Consistência Eventual. Peer-to-peer. Programação Funcional.

ABSTRACT

Abstract…

Keywords: …

LISTA DE ABREVIATURAS E SIGLAS

LISTA DE FIGURAS

SUMÁRIO

1. INTRODUÇÃO

A difusão de dispositivos móveis leva, ano após ano, o acesso à internet para cada vez mais pessoas. "Em 2015 existem mais de 7 bilhões de assinaturas de celulares móveis em todo o mundo, partindo de menos de 1 bilhão em 2000." (SANOU; BRAHIMA, 2015, p. 1, tradução nossa).

Mais de 3 bilhões de pessoas tiveram acesso à rede mundial de computadores em 2015 (SANOU, BRAHIMA, 2015). *Softwares online*, como clientes de *e-mail*, sistemas de gerenciamento de conteúdo e sistemas de edição colaborativa de documentos podem ser utilizados por todos estes usuários se disponibilizados pela internet.

Algumas aplicações distribuídas utilizam o modelo de computação cliente-servidor, também conhecida como cliente-servidor de três níveis. Nesta arquitetura, todos os clientes, solicitantes de um recurso, se conectam e se comunicam através de um servidor, fornecedor de recursos.

Segundo Donald Bales (2003, p. 133, tradução nossa) "Uma arquitetura de três níveis adiciona ao modelo de dois níveis uma nova camada que isola o processamento de dados em um lugar centralizado […]". Este isolamento traz importantes vantagens para aplicações distribuídas. Entretanto, este modelo computacional apresenta problemas graves quando utilizado em redes de internet móveis.

Redes de internet móveis são geralmente mais lentas do que redes banda-larga cabeadas. Além disso, a cobertura geográfica e a qualidade da conexão variam drasticamente entre regiões — incluindo locais não muito distantes. Dados da OpenSignal (2016) mostram uma latência média de 172 milissegundos em redes 3G.

De acordo com Shapiro et al. (2011) modelos de Forte Consistência Eventual (FCE) garantem melhor disponibilidade e performance em sistemas distribuídos, principalmente em redes com grande latência. Isto é alcançado por atualizações serem realizadas em réplicas locais dos dados e eventualmente sincronizadas com outras réplicas dentro do sistema. Uma estrutura de dados compatível com as exigências de FCE é a *Conflict-free Replicated Data Type* (CRDT, em português Tipo de Dados Replicados Livre de Conflitos).

CRDTs garantem a consistência dos dados pois operações realizadas nesta estrutura são comutativas. CRDTs baseadas em operações transmitem seu estado propagando operações de atualização realizadas. Estas operações também devem ser comutativas. Para remover a necessidade de manter garantias de entrega única das mensagens de operação, as funções de operações de atualização também devem ser idempotentes e associativas.

Com as garantias de uma CRDT com operações associativas, comutativas e idempotentes, é possível construir um sistema em uma arquitetura de rede em que as réplicas se comunicam diretamente e propagam atualizações de forma distribuída e assíncrona. Esta arquitetura de rede se chama *peer-to-peer* (P2P), onde cada participante da rede atua tanto como cliente (requisitando dados) quanto como servidor (servindo dados) (SCHOLLMEIER; Rüdiger, 2002).

O presente trabalho utiliza a CRDT Logoot para implementar um Sistema de Edição Colaborativa (SEC) distribuído com comunicação por uma rede *peer-to-peer*. A Logoot utiliza simples identificadores de posição para atingir comutatividade e associatividade em operações de inserção e remoção, que podem ser executadas em qualquer réplica participante do sistema sem alterar a ordem das linhas do documento (WEISS; URSO; MOLLI, 2008, p. 5).

Com o fim de facilitar a implementação da estrutura de dados Logoot, da interface de rede e do editor de texto necessários para o Sistema de Edição Colaborativa a ser desenvolvido como estudo de caso deste projeto, Elm foi a linguagem escolhida. Elm é uma linguagem de programação fortemente tipada que segue o paradigma funcional (CZAPLICKI, 2012), e o projeto tira proveito de suas estruturas de dados imutáveis e tipos.

1.1. OBJETIVO GERAL

Desenvolver um Sistema de Edição Colaborativa Logoot em um editor de texto com comunicação *peer-to-peer* (P2P) utilizando a linguagem de programação Elm.

1.1.1. OBJETIVOS ESPECÍFICOS

Desenvolver módulo Elm que implemente o sistema Logoot e permita todas as suas funcionalidades: adição, remoção e edição de conteúdo; Implementar módulo Elm que transforme a estrutura de dados Logoot em um editor de texto capaz de realizar operações de adição, remoção e edição de conteúdo no sistema; Implementar infraestrutura de rede pra suportar a Edição Colaborativa por P2P utilizando a tecnologia WebRTC; Demonstrar a utilização de uma linguagem de programação funcional para trabalhar com estruturas de dados complexas e compor pequenas partes do *software* em um sistema completo.

1.2. ORGANIZAÇÃO

A introdução do tema do trabalho está no primeiro capítulo. Nela temos a apresentação do tema do trabalho e seus objetivos.

No segundo capítulo a revisão bibliográfica é apresentada — um detalhamento de cada tecnologia utilizada para solucionar o problema apresentado. Estão neste capítulo assuntos como *Conflitc-free Replicated Data Types*, arquitetura de rede *peer-to-peer*, Logoot, paradigma de programação funcional, linguagem de programação Elm e testes de propriedade.

O terceiro capítulo aborda as metodologias utilizadas para a o desenvolvimento deste projeto.

No quarto capítulo o desenvolvimento do trabalho é destrinchado e apresentado, demonstrando as funcionalidades do Sistema de Edição Colaborativa implementado.

2. REVISÃO BIBLIOGRÁFICA

2.1. CONFLICT-FREE REPLICATED DATA TYPES

Em sistemas distribuídos é muito custoso manter uma ordem global de operações realizadas em uma estrutura de dados. Modelos de Forte Consistência Eventual (FCE) são mais adequados em sistemas distribuídos que têm como requisito a replicação e consistência de dados, principalmente quando diferentes clientes estão conectados por meio de redes instáveis com grande latência (SHAPIRO et al., 2011).

Como descrito por Shapiro et al. (2011), um modelo de Forte Consistência Eventual estabelece a convergência livre de conflitos de operações distribuídas em uma estrutura de dados. Dessa forma, réplicas não necessitam de sincronização e o sistema se mantêm consistente independente da ocorrência de inúmeras falhas. Na FCE, réplicas que receberam as mesmas atualizações eventualmente atingem o estado de convergência.

CRDTs são estrutura de dados que garantem a compatibilidade com o modelo de FCE. Nelas, operações podem ser realizadas localmente e eventualmente sincronizadas com outras réplicas, sempre mantendo a consistência entre as estruturas de dados distribuídas.

Segundo Sun et al. (1998 apud WEISS; URSO; MOLLI 2008, tradução nossa):

> Um sistema de edição colaborativa é considerado correto se este respeitar o critério CCI:
> - **Causality** (causalidade, em português): todas as operações ordenadas por uma relação de precedência, no sentido da relação *aconteceu antes* de Lamport [adicionar referência], são executadas na mesma ordem em todas as réplicas.
> - **Convergence** (convergência, em português): O sistema converge se todas as réplicas são idênticas quando o sistema está parado [...]
> - **Intention preservation** (preservação de intenção, em português): O efeito esperado de uma operação deve ser observado em todas as réplicas [...]

Existem dois tipos principais de CRDTs: baseadas em estado e baseadas em operações.

CRDTs baseadas em estado propagam seu estado completo entre réplicas, as quais utilizam uma função de combinação para aplicar alterações. A função de

combinação deve ser associativa, comutativa[1] e idempotente para assegurar as propriedades de convergência da FCE. Numa abordagem baseada em estado, clientes propagam diferentes estados de sua estrutura de dados, sem incluir informações sobre a evolução deste estado (AHMED-NACER et al., 2011).

Sendo *c* a função de combinação, s_x, s_y e s_z o estado de três réplicas *x*, *y* e *z*, então:

$$c(s_x, s_y) = c(s_y, s_x) \quad (1)$$

$$c(s_x, c(s_y, s_z)) = c(c(s_x, s_y), s_z) \quad (2)$$

$$c(s_x, s_y) = c(c(s_x, s_y), s_y) \quad (3)$$

A equação (1) representa a comutatividade, a (2) associatividade e a (3) idempotência.

As CRDTs baseadas em operações propagam operações como inserção e remoção entre réplicas, em vez de todo o estado. A convergência é garantida por assegurar que a função de aplicação de operação é uma função associativa, comutativa e idempotente.

Sendo *o* a função de aplicação de uma operação dado um estado, *s* o estado de uma réplica, *i* uma operação de inserção e *r* uma operação de remoção, então as propriedades de comutatividade, associatividade e idempotência são representadas pelas equações (4), (5) e (6) respectivamente: *o(o(s, i), r) = o(o(s, r), i)* (comutatividade e associatividade) e *o(s, i) = o(s, i)* (idempotência).

$$o(o(s, i), r) = o(o(s, r), i) \quad (4)$$

$$o(o(o(s, i), r), i) = o(o(o(s, r), i), i) \quad (5)$$

$$o(s, i) = o(o(s, i), i) \quad (6)$$

[1] De acordo com o critério de causalidade de sistemas de edição colaborativa, não é estritamente necessário que a função de combinação seja comutativa entre todas as versões do estado de uma réplica. Se o critério de causalidade for respeitado, a função de combinação só precisará ser comutativa entre versões que não possuem uma relação de precedência.

Escolher entre CRDTs baseadas em estado ou baseadas em operações exige um entendimento melhor sobre as vantagens e desvantagens de cada abordagem.

As CRDTs baseadas em estado possuem a vantagem de implementação de apenas uma função de união para convergir estados de duas réplicas. Como todo o estado é propagado, também não há a necessidade de manter um *buffer* com todas as atualizações realizadas no documento. Entretanto, em sistemas em tempo-real, a sobrecarga causada pela propagação de todo o estado a cada atualização pode deixar o sistema menos responsivo, já que a diferença entre os estados de cada réplica precisa ser calculado constantemente (AHMED-NACER et al., 2011).

Para minimizar esta sobrecarga, Almeida, Shoker e Baquero (2014) propõem a *Delta State-based CRDT* (δ-CRDT). Este tipo de dados permite que apenas diferenças (*deltas*, em inglês) sejam propagadas entre réplicas. As diferenças em δ-CRDTs são geradas pelos chamados *δ-mutators*, que retornam o resultado das mutações realizadas no estado.

CRDTs baseadas em operações não impõem grande sobrecarga na propagação de mensagens de atualização. Como CRDTs baseadas em operações armazenam cada operação durante a evolução de uma réplica, também não é necessário realizar operações de combinação em toda a CRDT — apenas aplicar operações ainda não incorporadas (AHMED-NACER et al., 2011).

2.2. LOGOOT

Logoot é uma estrutura de dados criada para ser utilizada em sistemas de edição colaborativa em redes distribuídas *peer-to-peer*. Ela é também uma CRDT, tornando possível que várias cópias de um mesmo documento sejam alteradas e sincronizadas sem conflito (WEISS; URSO; MOLLI, 2008, p. 5, tradução nossa).

Uma vantagem da Logoot quando comparada com outras estruturas de dados como Wooki (WEISS; URSO; MOLLI, 2007) é que ela não adiciona um grande peso em cima dos dados brutos de um documento devido à baixa quantidade de meta-dados necessários para mantê-la consistente entre sincronizações com réplicas distribuídas. Isso deve-se à utilização de identificadores de posição (pid) baseados em uma lista de inteiros que podem ser removidos da

CRDT sem afetar a ordem de outros identificadores remanescentes (WEISS; URSO; MOLLI, 2008, p. 5, tradução nossa).

A duas operações disponíveis em uma Logoot são *inserir* e *remover*. Para se adequar como uma CRDT, essas operações precisam ser idempotentes, comutativas e associativas. Uma implementação baseada no sistema proposto por Weiss, Urso e Molli (2008) respeita a regra de comutatividade apenas quando há a causalidade de operações de inserção e remoção, levando réplicas a divergirem dependendo da ordem em que recebem operações realizadas por outras réplicas quando a causalidade não pode ser garantida.

Dada as seguintes definições:

- *i = (pid, c)*: operação de inserção que insere um conteúdo *c* no identificador de posição *pid*;
- *r = (pid)*: operação de remoção que remove o identificador de posição *pid*;
- *a(op, s)*: função que aplica a operação *op* no estado *s* e retorna o novo estado.
- *si*: estado inicial de uma réplica.

$$a(\, i(pid_1, ""), a(r(pid_1), s_i)\,) \neq a(\, r(pid_1), a(i(pid_1, ""), s_i)\,) \quad (7)$$

Como ilustrado na equação (7), as operações *i* e *r* não são comutativas. Isso se deve à implementação originalmente proposta ignorar a remoção de um *pid* ainda não existente no estado de uma réplica, já que se espera que a causalidade entre estas duas operações. Na CRDT Logoot-Undo (WEISS; URSO; MOLLI, 2010) — uma evolução da Logoot que permite ações de *undo* (desfazer, em inglês) — os autores propõem a adição de relacionar cada *pid* com um *degree* (grau, em inglês). A adição do *degree* permite expressar na estrutura de dados quando um *pid* ainda não adicionado é removido. Dessa forma, quando uma réplica recebe uma operação remota de remoção de um *pid* e futuramente sua inserção, o efeito é o mesmo de quando ela receber a inserção primeiro.

Ao tornar as operações de inserção e remoção comutativas, é possível implementar um SEC sem garantir a causalidade destas operações — elas não são mais ordenadas por uma relação de precedência, já que podem ser executadas em qualquer ordem e produzir o mesmo resultado.

2.3. ARQUITETURA DE REDE *PEER-TO-PEER*

Definida por Kellerer (1998 apud SCHOLLMEIER 2002, p.1), uma arquitetura de rede é *peer-to-peer* (P2P, ponto-a-ponto, em português) se seus participantes agem tanto como clientes (requisitando recursos) quanto como servidores (fornecendo recursos), utilizando seus próprios recursos de hardware que são acessados diretamente por outros participantes da rede sem entidades intermediárias.

Este estilo de arquitetura ficou popularizada com o *software* de compartilhamento de arquivos de áudio Napster (SCHOLLMEIER, 2002, p. 1, tradução nossa).

"Eles [sistemas P2P] provêem escalabilidade de distribuição de conteúdo mais barata e resistente a tentativas de censura." (ANDROUTSELLIS-THEOTOKIS; SPINELLIS, 2004 apud WEISS; URSO; MOLLI, 2010, p.1, tradução nossa). No presente trabalho, esta arquitetura soluciona o problema da necessidade de um servidor centralizado para sincronizar réplicas, transformando cada usuário em uma réplica do Sistema de Edição Colaborativa.

Sistemas de conteúdo dinâmico, como sistemas de edição colaborativa ou de controle de versionamento, ainda não são muito bem suportados em redes P2P (WEISS; URSO; MOLLI, 2010, p. 1), muito devido à dificuldade de implementação de sistemas que mantenham a consistência de dados entre réplicas. A CRDT Logoot foi introduzida por Weiss, Urso e Molli (2008) justamente para solucionar este caso de uso.

2.4. PARADIGMA DE PROGRAMAÇÃO FUNCIONAL

Em linguagens de programação imperativas, programas são escritos para modificar o estado no seu contexto de execução. Logo, a execução de programas escritos com linguagens de programação imperativas é não determinístico. Um outro paradigma de programação propõem uma abordagem diferente: a aplicação de funções que transformam dados de forma determinística.

"[...] *linguagens funcionais* são linguagens declarativas em que o modelo básico de computação é a função" (HUDAK, 1989, p. 361, tradução nossa). O estado de um programa funcional é aplicado como argumento de funções. Linguagens de programação modernas possuem um conjunto de utilitários para facilitar a escrita de programas funcionais.

Uma propriedade muito importante das funções em linguagens funcionais, e outras linguagens de estilo puro e declarativo, é a transparência referencial. "O termo *transparência referencial* é frequentemente utilizado para descrever este estilo de programação [puro e declarativo] em que 'iguais podem ser trocados por iguais'." (HUDAK, 1989, p. 362, tradução nossa). Isso significa que a aplicação de uma função pode ser substituída pelo seu resultado sem alterar o comportamento do programa.

A transparência referencial é a essência de muitas das vantagens desta classe de linguagem de programação. A garantia de uma função sempre retornar o mesmo resultado quando aplicada com os mesmos argumentos, não dependendo do seu contexto de execução, é a chave para tornar programas funcionais altamente performáticos com o uso de computação paralela, por exemplo.

Tratar funções como valores de primeira-classe é bem comum entre linguagens funcionais. Com funções sendo tratadas como outros valores (como inteiros e *strings*), é possível criar funções que abstraem a aplicação de outras funções ou que retornam funções para serem utilizadas posteriormente. Este tipo de função que retorna e/ou recebe outras função se chama Função de Ordem Superior (*Higher order function*, em inglês). Abstrair e construir programas a partir deste simples tipo "[...] aumenta a modularidade, servindo como um mecanismo de juntar fragmentos de um programa." (HUGHES, 1984 apud HUDAK, 1989, p. 383, tradução nossa).

Funções na linguagem de programação Elm (CZAPLICKI, 2012), por exemplo, utilizam o método de *currying*. Estas funções são sequências de funções que aplicam o valor de cada argumento retornando uma nova função para o argumento sucessor, até a exaustão da lista de argumentos. O *currying* permite a escrita concisa de programas que passam valores em uma cadeia de aplicação de funções, já que é relativamente fácil criar abstrações de outras funções.

2.5. LINGUAGEM DE PROGRAMAÇÃO ELM

Para a implementação deste trabalho, foi escolhida a utilização da linguagem de programação Elm. Elm é uma linguagem de programação funcional criada especificamente como uma ferramenta para a implementação de *websites* e *webapps*. (CZAPLICKI, 2012).

Uma das principais características do Elm é que ele é fortemente tipado. Isto possibilita que o compilador garanta que todos os dados que estão fluindo entre as funções de um programa respeitam os tipos especificados ou inferidos. O compilador do Elm é capaz de inferir o tipo de funções e gera mensagens de erro altamente informativas quando o usuário tenta compilar uma aplicação com problemas de tipos.

A linguagem Elm foi inicialmente criada como uma proposta de uma linguagem que seguisse a ideia de Concurrent Functional Reactive Programming (Concurrent FRP) (CZAPLICKI, 2012). Sua implementação baseia-se no estudo detalhado de diversas formas de implementar uma linguagem de programação que segue os conceitos de FRP. Entretanto, a linguagem abandonou boa parte de seus conceitos iniciais em sua versão 0.17, adotando um sistema mais simples chamado de *Command and Subscriptions* (CZAPLICKI, 2016).

> Programação reativa functional é uma abordagem declarativa para design de GUI. O termo declarative faz a distinção entre o "o quê" e o "como" da programação. Uma linguagem declarativa permite que você diga *o quê* é exibido, sem ter que especificar exatamente *como* o computador deve fazer isso. Com a programação reativa funcional, muitos dos detalhes irrelevantes são deixados para o compilador, livrando o programa para pensar em um nível muito mais alto do que a maioria dos frameworks de GUI disponíveis. (CZAPLICKI, 2012, p. 1)

O conjunto das principais funcionalidades da linguagem Elm deram espaço para a criação da *Elm Architecture* (Arquitetura Elm, em português). A *Elm Architecture* tem como princípio três principais partes claramente separadas: Model, Update e View (THE ELM ARCHITECTURE, 201-).

O Model (modelo, em português) é o estado de uma aplicação. Em uma aplicação que segue a *Elm Architecture*, todo o estado está centralizado no Model. Já a Update (atualizar, em português) é uma função que transforma o Model durante a execução do programa conforme mensagens de alteração são enviadas pela View. Por último, a View é uma função que recebe um Model e retorna uma representação

declarativa da interface de usuário a ser exibida, que futuramente envia mensagens de alteração, fechando o ciclo de dados do sistema.

Esta simples e poderosa arquitetura pode ser utilizada para implementar pequenos protótipos e têm se provado muito valiosa para também implementar aplicações maiores como no caso da NoRedInk e CircuitHub (THE ELM ARCHITECTURE, 201-). Em conjunto com outras funcionalidades como tipos algébricos, estruturas de dados imutáveis e funções puras, o Elm é uma solução completa para implementação de aplicativos web interativos e confiáveis.

Como o compilador do Elm gera código JavaScript, executado por todos os navegadores web modernos, esta linguagem é totalmente adequada para a criação de aplicações web.

2.6. TESTES DE PROPRIEDADE

Testes são uma importante parte do processo de desenvolvimento de software. Para garantir a qualidade da implementação de aplicativos, é importante averiguar a exatidão da implementação de acordo com a especificação — evitando *bugs* e problemas na interação do usuário com a aplicação.

> Testar é de longe a abordagem mais utilizada para garantir a qualidade de software. Testar também é muito trabalhoso, representando até 50% do custo de desenvolvimento de software. [...] O custo de testar motiva esforços para automatizá-lo, inteiramente ou parcialmente. (CLAESSEN; HUGHES, 2000, p. 1, tradução nossa)

Claessen e Hughes (2000) propõem uma maneira mais eficiente de testar programas Haskell, utilizando ideias como geração aleatória de testes e especificações executáveis, chamada *QuickCheck*. Funções puras não geram efeitos colaterais em outras partes do sistema, tornado-as mais fácil de testar já que é possível analisar o resultado da execução destas funções apenas ao analisar o valor retornado. Linguagens funcionais como Haskell e Elm permitem apenas a criação de funções puras, fator que as torna especialmente adequadas para testes automatizados, segundo Claessen e Hughes (2000).

Resultados encontrados pelos autores do *QuickCheck* em seus experimentos demonstraram que ao utilizá-lo para testar diferentes problemas algumas classes de erros são encontradas mais facilmente (CLAESSEN; HUGHES, 2000, p. 6-9):

a) Especificações incompletas ou muito abrangentes;

b) Implementações com lógica incorreta;

c) Implementações que não levam em conta diferentes tamanhos para os dados de entrada, incluindo dados como listas vazias.

Com o *QuickCheck*, escreve-se especificações formais que podem ser automaticamente verificadas — que determinam se o programa está funcionando de acordo com o critério especificado. Também pode ser necessário prover geradores de dados caso os incluídos na ferramenta não forem suficiente. Isso substitui a necessidade de implementar casos de teste manualmente.

A ferramenta elm-test[2] implementa testes automatizados utilizando as mesmas ideias propostas por Claessen e Hughes (2000), permitindo a utilização desta forma de testes para verificar programas escritos com a linguagem de programação Elm.

[2] Disponível em https://github.com/elm-community/elm-test. Acessado em 9 de novembro de 2016.

3. METODOLOGIA

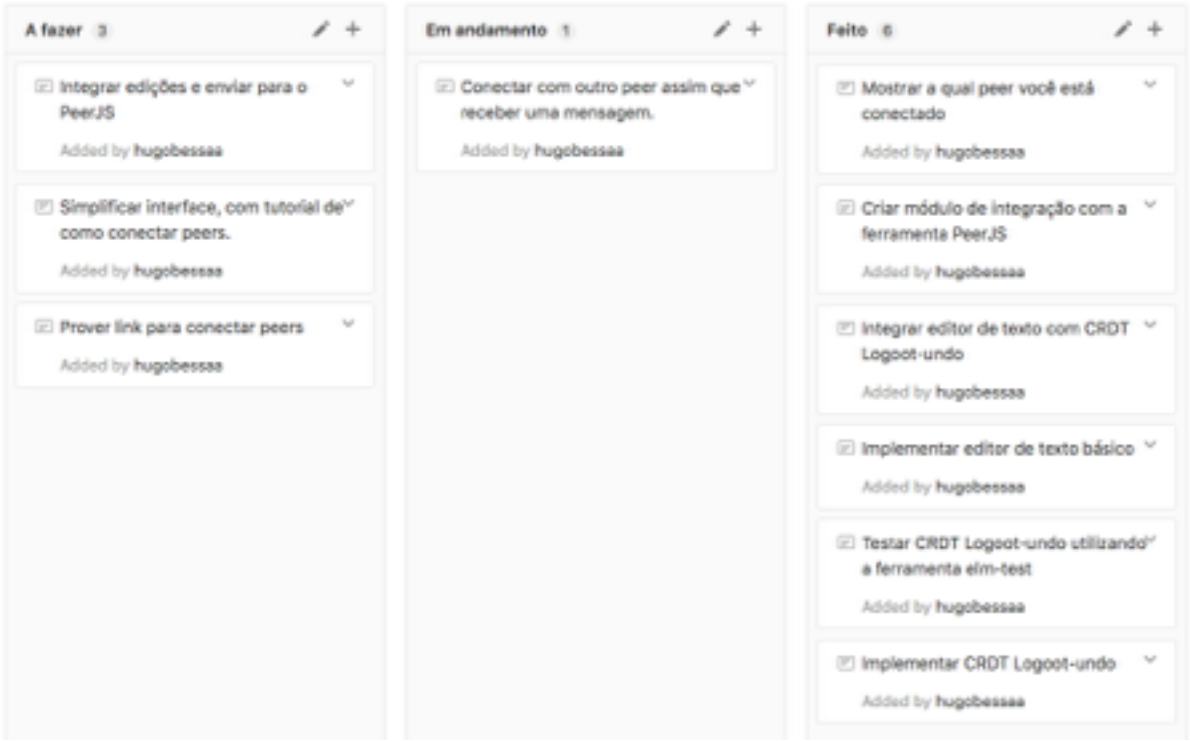
O presente trabalho foi desenvolvido utilizando a metodologia ágil de desenvolvimento de *software*. Segundo Beck et al. (2001), o desenvolvimento ágil valoriza "indivíduos e interações sobre processos e ferramentas, *software* funcional sobre documentação abrangente, colaboração de clientes sobre negociação de contratos, responder a mudanças sobre seguir um plano". Estes valores realçam a importância de *software* desenvolvido, testado e executado acima de outros objetivos como a presença de uma documentação extensiva — esta que frequentemente não é atualizada pelos participantes do projeto.

Projetos que adotam metodologias de desenvolvimento ágil também possuem documentação, embora possivelmente muito menos do que projetos que usam metodologias mais tradicionais baseadas em grandes quantidades de documentos e planejamento.

Para ajudar na organização do processo de implementação do presente trabalho, utilizamos a ferramenta Kanban. "Kanban foi uma ferramenta originalmente utilizada na Toyota para balancear demanda e capacidade através da cadeia de valor." (SKARIN, 2015, p. 4, tradução nossa [adicionar referência]). Ainda segundo Skarin (2015), Kanban é muito útil para permitir a visualização dos processos de trabalho, gerenciar o fluxo de tarefas e limitar o *Work-in-progress* (em português, trabalho em andamento). No presente trabalho utilizamos o Kanban para gerenciar as tarefas a fazer, em andamento e concluídas, permitindo a visualização do andamento do projeto. O software escolhido para auxiliar na utilização do Kanban foi o recém-lançado GitHub Projects[3]. Nele é possível criar um quadro Kanban com colunas e adicionar cartões representando tarefas a serem concluídas. Alguns exemplos de cartões de tarefas são "Implementar editor de texto básico", "Criar módulo de integração com a ferramenta *PeerJS*" e "Prover *link* para conectar *peers*". A Figura 1 mostra uma captura de tela do quadro Kanban durante o desenvolvimento do projeto.

[3] Lançado no dia 14 de setembro de 2016. Notas de lançamento disponíveis em <https://github.com/blog/2256-a-whole-new-github-universe-announcing-new-tools-forums-and-features>. Acesso em 12 de novembro de 2016.

Figura 1 — Quadro Kanban durante o processo de desenvolvimento



Fonte: Elaborado pelo autor

A utilização da ferramenta no presente trabalho foi simples em comparação com implementação em times com projetos e equipes maiores, que acompanham diversas métricas e o utilizam para realmente melhorar processos durante longos períodos de tempo. Apesar disso, o Kanban foi útil na medida em que ele permitiu que o processo de desenvolvimento fosse suficientemente organizado durante a implementação do SEC.

Em conjunto com a ferramenta Kanban, a metodologia Lean também foi aplicada. "Lean é uma prática de produção que considera qualquer consumo de recursos para qualquer objetivo sem ser o de criação de valor para o consumidor final um desperdício, e então um alvo para eliminação." (SKARIN, 2016, p. 7, tradução nossa). Logo, o foco em cada parte do processo de desenvolvimento deve estar em entregar valor. Seguindo essa metodologia, diminuímos a quantidade de trabalho focado no planejamento do projeto, apontando o esforço no projeto na pesquisa e implementação do SEC.

O processo de pesquisa envolveu o estudo detalhado de como implementar um SEC, quais tecnologias estão disponíveis para estabelecer uma comunicação P2P em um navegador *web*.

Para auxiliar o processo de desenvolvimento, também foram levantados requisitos funcionais pertinentes para a implementação do sistema. Requisitos funcionais são identificados durante a análise de requisitos de um projeto e são "uma tarefa necessária, ação ou atividade que deve ser cumprida" (DEPARTMENT OF DEFENSE, 2001, p. 36, tradução nossa).

RF001 - Inserir carácter de texto em editor
RF002 - Remover carácter de texto em editor
RF003 - Enviar alterações locais para *peer* conectado
RF004 - Aplicar alterações recebidas de *peer* conectado
RF005 - Prover *link* para compartilhar documento com outro *peer*
RF006 - Conectar com *peer* ao acessar *link* de compartilhamento
RF007 - Mostrar informação sobre a conexão entre *peers*

4. DESENVOLVIMENTO

O desenvolvimento do projeto iniciou com a implementação da CRDT Logoot (WEISS; URSO; MOLLI, 2008), em um módulo Elm chamado Logoot. A versão 0.17 do Elm foi utilizada. A primeira parte da implementação consistiu em estabelecer os tipos básicos que seriam utilizados. Os tipos estão ilustrados na Figura 2:

Figura 2 — Tipos implementados para a CRDT Logoot-undo

```
type Logoot a = Logoot a
        { cemetery : Dict Pid Int
        , content :
            { first : ( Pid, a )
            , intermediate : Dict Pid a
            , last : ( Pid, a )
            }
        , sorted : List ( Pid, a )
        }
type alias Pid = ( Positions, Clock )
type alias Positions = List Position
type alias Position = ( Line, Site )
type alias Line = Int
type alias Site = Int
type alias Clock = Int
```

Fonte: Elaborado pelo autor

Estes tipos utilizam os tipos disponíveis na biblioteca básica do Elm para expressar a estrutura da Logoot. O tipo `Logoot a` é o principal. Ele representa um Logoot-undo com valores de um tipo a qualquer e é um *record* de *cemetery*, *content* e *sorted* (respectivamente registro, cemitério, conteúdo e ordenado, em português). Estes são detalhes de implementação da CRDT Logoot, utilizados para garantir todas as propriedades esperadas.

Uma CRDT Logoot sempre possui ao menos duas entradas, a primeira (*first*, em inglês) e a última (*last*, em inglês). Estas entradas são essenciais pois as operações de inserção na Logoot-undo sempre geram um identificador único que deve estar entre outras duas entradas segundo a função de ordenação. Para representar esta estrutura, o *content* também é um *record* e contém os dados inseridos na estrutura de dados em *intermediate* (intermediário, em português), junto às entradas *first* e *last*.

Todas as operações realizadas no tipo `Logoot a` são comutativas. Para assegurar esta propriedade matemática, o *cemetery* deste tipo é implementado

como o *cemetery* proposto por Weiss, Urso, Molli (2010). As operações de inserção e remoção também seguem a lógica proposta por Weiss, Urso, Molli (2010). A funcionalidade de operações *undo* e *redo* (respectivamente desfazer e refazer, em português) foi omitida na implementação da CRDT Logoot no presente trabalho.

Para evitar que os detalhes de implementação da estrutura de dados estivessem abertos para outras partes do código que utilizam este tipo, apenas o tipo `Logoot a` é exportado, não seu construtor. Logo, a única maneira de criar e alterar um `Logoot a` é através de funções exportadas pelo módulo, permitindo que os detalhes de implementação sejam alterados sem a necessidade de atualizar outros módulos que utilizam este tipo.

Uma parte importante do funcionamento da Logoot é o `Pid`. Este tipo representa uma tupla de posições (`Positions`) e um número crescente (`Clock`). Positions é uma lista com elementos que representam uma posição (`Position`). Position é uma tupla de um identificador de linha (`Line`) e um identificador do cliente que criou esta linha (`Site`). Na Figura 3 podemos observar diversos valores do tipo `Pid`, organizados em ordem crescente segundo a lógica de ordenação proposta por Weiss, Urso, Molli (2008).

Figura 3 — Valores do tipo Pid representados em ordem crescente

```
( [ ( 0, 0 ) ], 0 )
( [ ( 1, 0 ) ], 1 )
( [ ( 1, 0 ) ], 2 )
( [ ( 2, 3 ), (1, 0) ], 10 )
( [ ( 2, 3 ), (1, 3) ], 3 )
( [ ( 2, 3 ), (5, 3), (1, 0) ], 1 )
```

Fonte: Elaborado pelo autor

O desenvolvimento do SEC se deu logo após a implementação do módulo Logoot. Toda a lógica necessária para exibir a interface de usuário e reagir às interações dos usuários está dentro do módulo Elm Editor.

O módulo Editor segue a Elm Architecture (THE ELM ARCHITECTURE, 201-). Esta arquitetura permite a divisão lógica de diferentes partes do sistema que exercem diferentes responsabilidades. A interface de usuário se torna uma função do estado e o estado uma função do estado anterior — ou inicial — e uma acão do sistema.

Inicialmente o estado (Model) da aplicação é um *record* com a representação vazia de um `Logoot String`, um valor do tipo `String` vazio representando o texto do documento, um *site* do tipo `Site`, um *clock* do tipo `Clock`, um *peer* do tipo `Site` e um *location* (localização, em português) do tipo `String` representando a URL (Uniform Resource Locator. Localizador padrão de recursos, em português) da página onde o SEC é executado. Estes dados iniciais são retornados pela função `init`. A declaração do tipo do Model é demonstrado na Figura 4.

Figura 4 — Declaração do tipo Model

```
type alias Model =
    { text : String
    , logoot : Logoot String
    , site : Site
    , clock : Clock
    , peer : Site
    , location : String
    }
```

Fonte: Elaborado pelo autor

A função `init` (de *initialize*, inicializar, em português) é uma função que recebe um `peer` e um `location` e retorna uma tupla de `Model` representando o estado inicial do programa e um gerador aleatório de Int, utilizado para definir o valor de `site` no `Model`. Estes dois valores, *peer* e `location`, vêm da lógica de inicialização do SEC escrito em Elm através do JavaScript. Esta lógica também garante que o SEC será renderizado em toda a página no navegador *web*. O valor de `location` é diretamente baseado na URL da página em que o SEC é executado e o valor `peer` é baseado na *hash* da URL. Como um exemplo, quando o SEC é executado em uma página na URL `http://locahost:8000/index.html#1234`, `location` tem o valor de `http://localhost:8000/index.html` e `peer` tem o valor de `1234`.

Quando o usuário acessa a página da web onde o SEC é executado, a primeira tela apresenta o editor de texto com uma mensagem simples incentivando o usuário a escrever algo. Uma seção com instruções para compartilhar o documento escrito com um outro usuário e um link para o repositório do projeto no site GitHub.com também são apresentados nesta tela, como mostra a Figura 5.

Figura 5 — Primeira tela exibida para o usuário

Fonte: Elaborado pelo autor

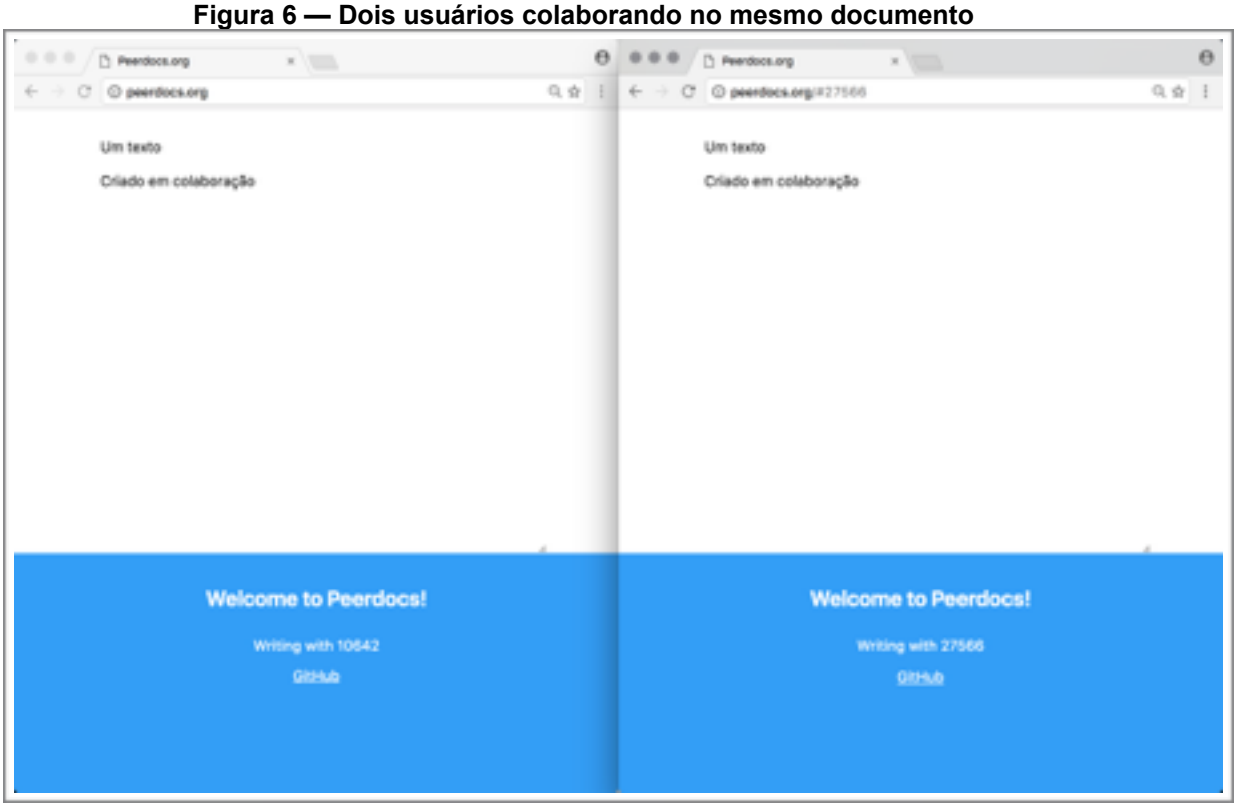
Ao acessar o link destacado na seção de instruções — http://peerdocs.org/#27566 no caso ilustrado pela Figura 5 — e digitar algo no campo de texto, o usuário é conectado ao outro usuário de `site` identificado na *hash* da URL. Essa conexão se dá através do serviço *PeerJS*, utilizando o `site` e uma chave chave privada necessária para autorização e criando uma conexão com outro usuário.

Quando um usuário digita ou remove uma parte do conteúdo no campo de texto, uma ação é enviada. O programa então computa um novo `Model` de acordo com o que foi alterado. A definição da posição, da operação e de quais carácteres foram alterados é feita através da geração de uma lista de carácteres que foram adicionados, removidos ou não modificados. Após gerar esta lista de diferenças, o programa executa as operações necessárias na CRDT Logoot e armazena todas as operações de inserção e remoção que devem ser enviadas para o usuário conectado. Com todas essas informações computadas, uma tupla de novo `Model` e

operações a serem enviadas é retornado. O Elm se encarrega, então, de atualizar a interface baseado no novo `Model` e a enviar as operações através do *PeerJS*.

Ao receber mensagens de operações de outro usuário conectado, o programa executa cada operação na `logoot` presente em seu `Model`. A tela exibida quando dois usuários estão conectados é representada pela Figura 6.

Figura 6 — Dois usuários colaborando no mesmo documento



Fonte: Elaborado pelo autor

Quando dois usuários já começaram a colaboração em um documento, a seção de instruções mostra apenas uma mensagem ccom o `site` do usuário ao qual o SEC está conectado.

5. CONSIDERAÇÕES FINAIS

Implementar um aplicativo que funcione bem em redes lentas e/ou com grande latência é um desafio, especialmente quando é custoso garantir a ordem total das mensagens que trafegam pela rede. Com a ajuda das propriedades matemáticas asseguradas pela CRDT Logoot, foi possível criar um Sistema de Edição Colaborativa que funciona em uma rede *peer-to-peer*.

Para a implementação da CRDT Logoot, utilizar a linguagem de programação Elm possibilitou a garantia de que o programa não tem problemas de tipos e não falhará durante a sua execução. Apenas a análise estática dos tipos não é suficiente para garantir que a implementação está seguindo as propriedades esperadas. Testar o programa utilizando a ferramenta elm-test foi essencial para assegurar que a implementação era realmente comutativa, associativa e idempotente e também produzia a ordem esperada para cada elemento inserido. Durante o desenvolvimento da Logoot, os testes de propriedades mostraram problemas em casos em que a entrada de alguma função não era exatamente a esperada, mesmo que o tipo fosse válido.

A interface de rede do presente projeto foi implementada em JavaScript. No momento da implementação deste trabalho, não existia nenhuma biblioteca ou módulo para criar canais *peer-to-peer* utilizando apenas código Elm. Infelizmente JavaScript não é uma linguagem pura, funcional e fortemente tipada, o que deixa parte do projeto sem as garantias dadas pelo Elm. Apesar disso, o uso de *commands* e *subscriptions* para integrar código Elm com código JavaScript funcionou sem grandes problemas e tornou possível a implementação de uma forma declarativa de conectar com a API do serviço *PeerJS*.

Com o aprendizado adquirido durante a construção deste projeto será possível melhorar o Sistema de Edição Colaborativa por P2P e espera-se adicionar diversas novas funcionalidades em trabalhos futuros. Entre elas: adição de suporte a sincronização de mensagens enviadas enquanto desconectado da rede, colaboração com múltiplos usuários, persistência do documento no dispositivo do usuário e melhorias de performance para documentos grandes.

6. REFERÊNCIAS BIBLIOGRÁFICAS

ICT Data and Statistics Division. **ICT Facts & Figures 2015**. Maio de 2015. Disponível em: <http://www.itu.int/en/ITU-D/Statistics/Documents/facts/ICTFactsFigures2015.pdf>. Acesso em: 10 de abril de 2016.

BALES, Donald. **JDBC Pocket Reference**. julho de 2003. Disponível em: <https://web.archive.org/web/20061212141647/http://java.sun.com/developer/Books/jdbc/ch07.pdf>. Acesso em: 29 de abril de 2016.

OpenSignal. **O Estado das Redes Móveis: Brasil (fevereiro 2016)**. fevereiro de 2016. Disponível em: <http://opensignal.com/reports/auto/data-2016-02-brazil/report_pt.pdf>. Acesso em: 16 de maio de 2016.

SHAPIRO, Marc; PREGUIÇA, Nuno; BAQUERO, Carlos; ZAWIRSKI, Marek. **Conflict-free Replicated Data Types**. 19 de julho de 2011. Disponível em: <https://hal.inria.fr/inria-00609399v1/document>. Acesso em: 10 de abril de 2016.

AHMED-NACER, Mehdi, IGNAT, Claudia-Lavinia, OSTER, Gérald, ROH, Hyun-Gul, URSO, Pascal. **Evaluating CRDTs for Real-time Document Editing**. Disponível em: <https://hal.archives-ouvertes.fr/inria-00629503/document>. Acesso em: 21 de setembro de 2016.

SCHOLLMEIER, Rüdiger. **A Definition of Peer-to-Peer Networking for the Classification of Peer-to-Peer Architectures and Applications**. 2002. Disponível em <http://www.computer.org/csdl/proceedings/p2p/2001/1503/00/15030101.pdf>. Acesso em: 10 de abril de 2016.

LETIA, Mihai; PREGUIÇA, Nuno; SHAPIRO, Marc. CRDTs: **Consistency without concurrency control**. junho de 2009. Disponível em: <https://hal.inria.fr/inria-00397981/file/RR-6956.pdf>. Acesso em: 1 de outubro de 2016.

WEISS, Stéphane; URSO, Pascal; MOLLI, Pascal. **Logoot: a P2P collaborative editing system**. 10 de dezembro de 2008. Disponível em: <https://hal.inria.fr/inria-00336191/PDF/main.pdf>. Acesso em: 11 de abril de 2016.

WEISS, Stéphane; URSO, Pascal; MOLLI, Pascal. **Wooki: a P2P Wiki-based Collaborative Writing Tool**. 25 de junho de 2007. Disponível em: <https://hal.inria.fr/inria-00156190v2/document>. Acesso em: 13 de junho de 2016.

WEISS, Stéphane; URSO, Pascal; MOLLI, Pascal. **Logoot-undo: Distributed Collaborative Editing System on P2P networks**. 2010. Disponível em: <https://pdfs.semanticscholar.org/75e4/5cd9cae6d0da1faeae11732e39a4c1c7a17b.pdf>. Acesso em: 9 de novembro de 2016.

HUDAK, Paul. **Conception, Evolution, and Application of Functional Programming Languages**. setembro de 1989. Disponível em: <https://cse.sc.edu/~mgv/csce330f15/haskell/p359-hudak.pdf>. Acesso em: 9 de novembro de 2016.

CZAPLICKI, Evan. Elm: **Concurrent FRP for Functional GUIs**. 30 de março de 2012. Disponível em: <http://elm-lang.org/papers/concurrent-frp.pdf>. Acesso em: 9 de novembro de 2016.

CZAPLICKI, Evan. **A Farewell to FRP**. 10 de março de 2016. Disponível em: <http://elm-lang.org/blog/farewell-to-frp>. Acesso em: 9 de novembro de 2016.

THE ELM ARCHITECTURE. 201-. Disponível em: <http://guide.elm-lang.org/architecture/>. Acesso em: 9 de novembro de 2016.

CLAESSEN, Koen; HUGHES, John. **QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs**. 2010. Disponível em: <http://www.cs.tufts.edu/~nr/cs257/archive/john-hughes/quick.pdf>. Acesso em: 9 de novembro de 2016.

BECK, Kent, et al. **Manifesto for Agile Software Development**. Disponível em: <http://agilemanifesto.org/iso/en/>. Acesso em: 14 de junho de 2016.

SKARIN, Mattias. **Real-World Kanban**: Do Less, Accomplish More with Lean Thinking. The Pragmatic Programmers, 2015. 138 p.