UNIVERSIDADE FEDERAL DE ITAJUBÁ - *CAMPUS* ITABIRA

FELIPE BALIEIRO CETRULO
THIAGO MACHADO DA SILVA

COMPONENTES ASSÍNCRONOS DE UM NODO DE CLIENTE COMPLETO DE BITCOIN: ROTEAMENTO DA REDE P2P, COMANDOS REMOTOS E ESTRUTURAÇÃO DA BLOCKCHAIN

ITABIRA
2018

FELIPE BALIEIRO CETRULO
THIAGO MACHADO DA SILVA

COMPONENTES ASSÍNCRONOS DE UM NODO DE CLIENTE COMPLETO DE BITCOIN: ROTEAMENTO DA REDE P2P, COMANDOS REMOTOS E ESTRUTURAÇÃO DA BLOCKCHAIN

Trabalho de Conclusão de Curso dos discentes Felipe Balieiro Cetrulo e Thiago Machado da Silva da Universidade Federal de Itajubá - *Campus* Itabira

Professor Orientador: Paulo José Lage Alvarenga

ITABIRA
2018

FELIPE BALIEIRO CETRULO
THIAGO MACHADO DA SILVA

COMPONENTES ASSÍNCRONOS DE UM NODO DE CLIENTE COMPLETO DE BITCOIN: ROTEAMENTO DA REDE P2P, COMANDOS REMOTOS E ESTRUTURAÇÃO DA BLOCKCHAIN

Este Trabalho de Pesquisa foi julgado, como requisito parcial, para aprovação na disciplina Trabalho Final de Graduação da Engenharia da Computação da Universidade Federal de Itajubá – *campus* Itabira.

Pontuação obtida: _____

Itabira, ____ de _____ de 2018.

Paulo José Lage Alvarenga
Professor Orientador

BANCA EXAMINADORA

_____
_____
_____

AGRADECIMENTOS

Aos professores e demais do corpo discente da faculdade pelo conhecimento proporcionado ao longo de vários anos. Conhecimento técnico, profissional e também pessoal, que auxiliou na forma como entendemos e enxergamos o mundo – uma via de mão dupla. Também ao nosso professor e orientador Paulo, pela paciência e orientação a nós e ao projeto, inclusive pelos conteúdos lecionados que se relacionam com o tema deste trabalho.

RESUMO

O *Bitcoin Core*, nodo mais popular de *Bitcoin*, está sendo modificado para que possua uma menor quantidade de bloqueios de execução concorrente. Decorrente deste tipo de modificação ser delicada e da necessidade da execução paralela, que permitiria maior uso da capacidade total dos computadores de múltiplos núcleos, é desejável que exista uma implementação alternativa de nodo que tenha garantias de segurança de memória, uma vez que a detecção e a correção posterior dos erros de execução paralela seriam custosas. Tais garantias são inerentes à linguagem *Rust*, que com conceitos incomuns, oferece ferramentas que, em tempo de compilação, previnem algumas classes de erros de memória, como *data-races* e *use-after-free*.

Foram compreendidos os conceitos gerais em torno do ecossistema do *Bitcoin*, da sua *Blockchain* e em torno do funcionamento de um nodo de uma rede *P2P*, para que componentes de um nodo pudessem ser planejados, implementados e utilizados de tal forma que consigam realizar conexões com outros nodos da rede e oferecer uma estrutura inicial de um nodo gerenciável, de modo a facilitar o desenvolvimento de componentes posteriores, que poderão possuir as mesmas características desejáveis de assincronia e segurança.

Para facilitar a execução assíncrona dos componentes, foram implementados *actors* do sistema *Actor Model*, alguns dos quais possuem um comportamento definido por máquinas de estados, em uma topologia de canais de comunicação entre os *actors*.

O programa foi compilado com sucesso e tal resultado indica fortes garantias de segurança, sendo isto devido, principalmente, às garantias inerentes ao compilador da linguagem de programação *Rust*. Também foram feitos testes que corroboram com a assincronicidade dos componentes internos e do comportamento do nodo em relação ao processamento de informação e sua comunicação com a rede.

Palavras-chave: *Actor Model*. *Bitcoin*. Nodo. *P2P*. *Rust*.

ABSTRACT

The Bitcoin Core, the most popular Bitcoin node, is being modified in order to have fewer thread blocks on concurrent excecution. Due to the delicated nature of this kind of modification, and also to the parallel execution requierement, which results in a higher usage of a multicore computer's total capacity, it is desired that there is an alternative node implementation that offers memory-safety guarantees when concurrent access are being made.

It was intended to understand the general concepts around the Bitcoin ecosystem, it's Blockchain and also around the operation of a P2P Node Network, so that components of a node could be planned, implemented and used in such that they would connect with other nodes from the network and also offer an initial structure of a manageable node, facilitating the development of later components that may have the same asynchrony and security requirements.

To facilitate the asynchronous execution of the components, actors from the Actor Model system were implemented, some of which had a behavior defined by state machines, also in a communication channel topology between themselves.

The software has been successfully compiled and therefore implies a strong security guarantee, mainly due to the inertors of the compiler of the Rust programming language. Tests were also carried out to corroborate to the internal components' asynchronicity and the node's behavior in relation to information processing and it's network communication.

Keywords: Actor Model. Bitcoin. Node. P2P. Rust.

LISTA DE ILUSTRAÇÕES

LISTA DE ABREVIATURAS E SIGLAS

Arc	*Atomic Reference Counter*
BTC	*Bitcoin*
CLI	*Command-Line Interface*
CPU	*Central Processing Unit*
IPC	*Inter Process Communication*
MPSC	*Multiple Producer Single Consumer*
Mutex	*Mutual Exclusion*
P2P	*Peer-to-Peer*
QRCode	*Quick Response Code*
Rc	*Reference Counter*
RIPEMD	*RACE Integrity Primitives Evaluation Message Digest*
RwLock	*Reader Writer Lock*
SHA	*Secure Hash Algorithm*
SPV	*Simple Payment Verification*
TCP	*Transmission Control Protocol*
UTXO	*Unspent Transaction Output*

SUMÁRIO

1 INTRODUÇÃO

Os incentivos para o uso de moedas e a possibilidade da especialização do trabalho, provocando o aumento da produtividade e da riqueza de um conjunto de pessoas, surgiram de forma interligada. A troca direta entre bens e serviços é ineficaz em comparação com a troca indireta, no qual faz uso de um objeto cuja aceitação e portabilidade são maiores. Existiram diversas tentativas de implementação de moedas com o passar do tempo.

> Ao longo da história, diferentes bens foram utilizados como meios de troca: tabaco, na Virgínia colonial; açúcar, nas Índias Ocidentais; sal, na Etiópia (na época, Abissínia); gado, na Grécia antiga; pregos, na Escócia; cobre, no Antigo Egito; além de grãos, rosários, chá, conchas e anzóis. (ROTHBARD, 2013, p. 16)

Na competição histórica dentre vários ensaios de sistemas monetários nas civilizações, o ouro e a prata se sobressaíram e, no entanto, ambos sofreram intervenções prejudiciais dos governos através de políticas relacionadas aos bancos centrais no século XX.

Em resposta a estas intervenções, os pensadores da Escola Austríaca de Economia teceram críticas aos bancos centrais pelo seu controle sobre as moedas, pois acreditavam que isso sempre resultaria em distorções indesejáveis na vida de um grande número de pessoas.O *Bitcoin* surgiu como uma opção monetária que independe da confiança de terceiros e que não é controlada por nenhuma autoridade central (NAKAMOTO, 2008). Indícios da motivação pessoal de Nakamoto podem ser vistas através de sua mensagem gravada na criação dos primeiros *bitcoins*, no bloco gênese: "*The Times 03/Jan/2009 Chancellor on brink of second bailout for banks*". Esta posição crítica perante a ajuda financeira governamental aos bancos, que no caso estão prestes a entrar em falência, condiz com as conclusões de economistas da Escola Austríaca de Economia:

> Isso significa, também, que o governo não pode nunca tentar escorar situações não-salutares das empresas; ele nunca deve financiar ou emprestar dinheiro para firmas com problemas. (ROTHBARD, 1969).

A crítica dos austríacos ao sistema bancário regulado por um banco central e o funcionamento do *Bitcoin* se encontram em outro aspecto, na ausência de uma autoridade central com o controle da emissão das unidades monetárias, uma vez que para os austríacos, esta centralização da emissão é utilizada fortemente no aumento ou sustento da dívida pública e esta é indesejada.

Para von Mises (1966), a existência de dívidas públicas e seu crescimento médio contínuo trará uma consequência indesejável para a economia:

> A história financeira do último século mostra um contínuo aumento do montante da dívida pública. Ninguém acredita que os Estados irão suportar eternamente a carga dos juros a pagar. É óbvio que, mais cedo ou mais tarde, todos estes débitos serão liquidados de alguma maneira, diferente daquela prevista no contrato. (VON MISES, 1966, p. 279).

Para Rothbard (2013), a autoridade monetária central, uma vez existindo, utiliza-se do mecanismo de emissão de títulos públicos para inflacionar a moeda e sustentar o sistema bancário regulado:

> Indubitavelmente, o ativo que o Banco Central mais compra são os títulos da dívida do governo. Agindo assim, o Banco Central garante que sempre haverá liquidez para este mercado. Mais ainda: o governo garante que sempre haverá um mercado para seus próprios títulos. O governo pode facilmente inflacionar a oferta monetária ao emitir novos títulos da dívida pública: com o Banco Central deixando claro que irá sempre comprar tais títulos, o sistema bancário irá criar dinheiro para adquirir estes títulos e em seguida irá revendê-los para o Banco Central. Muitas vezes, a função do Banco Central será justamente a de sustentar o preço dos títulos da dívida pública em um determinado nível, comprando maciçamente estes títulos em posse dos bancos. Isso irá gerar um aumento substancial das reservas bancárias. E caso os bancos decidam expandir o crédito tendo por base estas reservas, o resultado será uma hiperinflação. (ROTHBARD, 2013, p. 66).

Por outro lado, várias autoridades públicas ou com relações governamentais investiram e realizaram estudos das tecnologias relacionadas ao *Bitcoin*, principalmente ao que se refere à *Blockchain*. Como é dito por Peck (2017), muitos dos maiores bancos uniram forças para criar sistemas semelhantes ao *Bitcoin*. Um exemplo notável vem da Associação Russa de Criptomoeda e Blockchain, que segundo Partz (2018), pretende lançar uma criptomoeda estatal em 2019.

Isto demonstra um crescente interesse acadêmico, comercial, jornalístico, e estratégico pela tecnologia, tornando seu estudo e compreensão desejável. Pode-se abordar esta tecnologia de diversos ângulos, como seu impacto da aplicação em novas áreas comerciais ou em políticas públicas, seu contraste com outros tipos de moedas, seu funcionamento, dentre outros. Este trabalho pretende abordar um entendimento e implementação técnica do *Bitcoin*, a primeira criptomoeda global a atingir um acúmulo considerável de valor (centenas de bilhões de dólares), além de alterar a organização dos componentes internos do sistema de forma a oferecer vantagens em relação ao sistema implementado mais popular.

1.1 Nodo

O comportamento do sistema do *Bitcoin* se faz presente graças aos programas executados nos principais computadores que fazem parte da rede do *Bitcoin*, nos nodos. Um destes programas é chamado de *Bitcoin Core*, que existe em código-aberto e é implementado na linguagem de programação C++.

O *Bitcoin Core* é, segundo Antonopoulos (2014), um nodo de cliente completo de *Bitcoin*, que possui um módulo de meio de comunicação com a rede do *Bitcoin* que se encarrega de possibilitar a troca de informações entre computadores distintos da rede; um módulo que interage com uma cópia da *blockchain*, que contém o histórico das transações e das provas de posse sobre as carteiras; um módulo que implementa um aplicativo de uma carteira de *bitcoin*, que possibilita analisar o saldo do usuário através da leitura da *blockchain* e a estruturação de mensagens de transações; e um módulo da mineração, que faz o agrupamento, validação e verificação das transações em blocos e os conectam em uma cadeia que tende a crescer serialmente. Tais módulos representam as principais aplicações envolvidas na rede do *Bitcoin*. Antonopoulos (2014) ainda comenta que 90% dos nodos utilizados e expostos de forma pública, na rede principal, são diferentes versões do *Bitcoin Core*. Esta implementação, apesar de a separação modular dos componentes de um nodo de cliente completo e segundo Newbery (2017), não faz um bom proveito, em determinadas situações, do poder de processamento das máquinas pois a implementação de programas paralelos é uma tarefa delicada, uma vez que erros podem facilmente ocasionar em *crash* ou *memory corruption*.

Sendo assim, pode-se indagar: como obter um programa do módulo de roteamento que suporte chamadas de comandos remotos e a estruturação de uma *Blockchain* de um nodo de cliente completo de *Bitcoin* que seja assíncrono e cuja execução seja *memory-safe*?

Na busca de uma resposta a esta pergunta, foi encontrada uma ferramenta que traz garantias quanto à segurança de memória, a linguagem de programação *Rust*. Beingessner (2015) observa que a ergonomia de programas seguros com execução concorrente em C++ é menor do que em relação ao *Rust*. Portanto, também acredita-se que uma alternativa em *Rust* de algumas das principais funções do *Bitcoin Core* é benéfico para a saúde da rede, pela maior facilidade de revisão e aprimoramento de tal código, e da inserção dos executáveis compilados em *Rust* em outras linguagens de programação e dispositivos de *hardware*.

Ainda em relação à pergunta supracitada, uma das formas de uma organização com maior facilidade de previsão comportamental dos componentes de um sistema assíncrono e paralelo é especificada pelo *Actor Model*, um tipo de organização utilizada por diversos projetos como demonstrado por Lightbend (2018), *showcase* de várias empresas que obtiveram resultados positivos com o uso da ferramenta *akka*, feito na linguagem *Scala*. A linguagem *Rust* é apropriada para organizar o sistema segundo o mesmo modelo e, tendo em vista as vantagens de segurança e ergonomia que ela fornece e a importância do *Bitcoin* no cenário atual, utilizar *Rust* no desenvolvimento inicial de um nodo similar a um *Bitcoin Core* pode apresentar grandes benefícios para a tecnologia da rede da moeda virtual.

Com o uso do *Rust* em conjunto com a organização em *Actor Model*, foi demonstrado a implementação de um programa que, uma vez validada pelo compilador da linguagem *Rust*, responde a esta pergunta pois assume-se que tal implementação garante as características

desejadas de assincronia e segurança de memória.

Com o uso do *Rust* em conjunto com a organização em *Actor Model*, foi demonstrado a implementação de um programa de nodo inicial de *Bitcoin* que, uma vez validada pelo compilador da linguagem *Rust*, responde a esta pergunta pois se assume que tal implementação garante as características desejadas de assincronia e segurança de memória. Além disto, é possível de se atingir validação pelo compilador do *Rust* em implementações de nodos de clientes incompletos, o que possibilita a inserção de implementações incrementais e que, mesmo não atendendo às funcionalidades totais do nodo de referência, garante as características desejadas de assincronia e segurança de memória para as funcionalidades até então implementadas.

Neste sentido, este trabalho, desenvolvido de forma incremental, tem por objetivos: obter um programa com alguns dos componentes[1] de um nodo de cliente completo de *Bitcoin* que seja assíncrono e cuja a execução seja *memory-safe*; compreender sobre a implementação do *Bitcoin*, e sobre seus componentes internos, sobre conceitos de rede P2P e sobre a linguagem *Rust*; utilizar o modelo matemático *Actor Model* para possibilitar a execução de código assíncrono e concorrente aplicado na linguagem *Rust*; realizar testes funcionais do nodo em comunicação com outros nodos da rede; realizar o *download* de informações externas pertinentes à *Blockchain*.

Os objetivos foram atingidos, implementando um programa conforme especificado, no qual as mensagens sobre gerenciamento de memória foram analisadas, o nodo executou suas funções e foi testado[2] por comandos remotos. Desse modo, no capítulo 2 serão abordadas as definições conceituais que envolvem *Bitcoin*, *Rust*, P2P e *Actor Model*. Em seguida, no capítulo 3, serão descritos como foram feitos o planejamento da arquitetura e a implementação do programa. Por fim, no capítulo 4, será apresentado uma discussão sobre os resultados obtidos da implementação do programa.

[1] Módulo de roteamento que suporte chamadas de comandos remotos e a estruturação de uma *Blockchain*.
[2] Teste de conexão com outro *peer* da rede P2P.

2 FUNDAMENTAÇÃO CONCEITUAL

2.1 Considerações Iniciais

Neste capítulo serão apresentando alguns conceitos necessários ao entendimento do trabalho desenvolvido. O primeiro destes conceitos será o surgimento e definição do *Bitcoin*, incluindo as características pertinentes ao seu entendimento, como a definição de transação, *Blockchain*, *mineração*, nodos e o protocolo da rede e também da sua relação com a criptografia.

O segundo conceito pertinente dissertado será a linguagem de programação utilizada na implementação do projeto, *Rust*. Serão apresentados os principais conceitos que destacam esta linguagem daquelas mais populares e conhecidas, como C++. São eles: *Ownership*, *Borrowing*, *Lifetime*, *Traits* – que podem ser inicialmente compreendidas como interfaces – e *Crates* – utilizadas como bibliotecas externas ao longo da implementação.

Outra noção que será explicada é o de rede P2P, onde será tratado a sua definição e suas principais características. O último conceito elucidado será o de *Actor Model*, onde será dada uma breve explicação do seu entendimento, a motivação de ter utilizado este modelo para trabalhar com paralelismo e também os requisitos necessários para se implementar um sistema utilizando este modelo.

2.2 *Bitcoin*

É considerado o início do *Bitcoin* a publicação do *whitepaper* técnico de Satoshi Nakamoto, onde foram dissertados algumas características comercialmente atrativas, o funcionamento de aspectos chaves da tecnologia e algumas análises estatísticas de risco sobre a organização dos computadores na rede.

A concepção do *Bitcoin* possibilitou a existência de moedas privadas que, até então, eram facilmente suspensas de circulação por autoridades legais. Tal foi o caso da *Liberty Dollar*, uma moeda privada que, segundo Feuer (2012), teve o seu criador formalmente acusado de conspiração e terrorismo pelo fato de competir com o banco central na criação e gerenciamento de uma moeda. Esta percepção demandou a busca por uma maior resistência pelas moedas descentralizadas, e isso implica a ausência de uma autoridade central, obrigando a rede a manter coerência sobre o estado dos saldos de todos usuários, considerados por cada nodo, de uma forma descentralizada.

O termo *Bitcoin* é recente e está em processo de compreensão. Segundo Ulrich (2014, p. 15):

> O Bitcoin é uma forma de dinheiro, assim como o real, o dólar ou o euro, com a diferença de ser puramente digital e não ser emitido por nenhum governo. [...] Com o Bitcoin você pode transferir fundos de A para B em qualquer parte do mundo sem jamais precisar confiar em um terceiro para essa simples tarefa.

De forma simplificada, o *Bitcoin* tem uma *blockchain* que representa um histórico imutável das transferências efetuadas no passado, e também uma acumulação de *Proof of Work* realizada na validação de cada bloco. Como uma pilha que pode apenas receber novos elementos, novas informações não alteram informações passadas. Cada participante tem a liberdade de alterar ou ignorar as informações da *blockchain* do seu próprio ponto de vista (em seu próprio disco-rígido), mas caso queira que outros nodos da rede não o ignore e façam determinadas alterações sobre suas próprias *blockchains* (em seus próprios disco-rígidos), deverá se comportar e se comunicar conforme as regras do sistema.

Antes de se definir o que é o *Bitcoin*, é recomendado seguir o padrão existente na literatura utilizada, onde o *Bitcoin* é dividido em duas perspectivas: seu funcionamento como uma tecnologia que oferece uma forma de pagamento formada por uma rede de computadores; e uma moeda digital. Tal diferenciação é dada na escrita da palavra *bitcoin*, onde o nome próprio *Bitcoin* representa a primeira perspectiva, e o nome em caixa baixa *bitcoin* representa a segunda perspectiva. Ainda como uma moeda digital, uma quantidade de *bitcoins* pode ter sua unidade representada por BTC.

Um aspecto para se compreender do *Bitcoin* é o seu contraste com outras formas de dinheiro, como foi feito por Bech e Garratt (2017), que delineou certas características que podem existir em comum entre diversas formas de dinheiro: ser de responsabilidade de alguém, ser *Peer-to-Peer* e ser eletrônico. As formas de dinheiro comparadas foram as *commodities* (como moedas de ouro), os papéis-moeda (como as notas de Real), os depósitos bancários (como Real em Conta-Corrente) e as criptomoedas (como o *Bitcoin*). O contraste pela primeira característica, de ser de responsabilidade de alguém, separa as *commodities* e as criptomoedas dos papéis-moeda e dos depósitos bancários, pois os últimos são de responsabilidade dos bancos centrais. O contraste pela segunda característica, de ser *Peer-to-Peer*, separa as *commodities*, as criptomoedas e os papéis-moeda dos depósitos bancários, pois apenas este último requer autorização de uma autoridade central na transferência de posse ou título de dinheiro. O contraste pela terceira característica, de ser eletrônico, separa as *commodities* e os papéis-moeda dos depósitos bancários e criptomoedas, pois apenas estes últimos existem eletronicamente.

Para Antonopoulos (2014), o *Bitcoin*, é um conjunto de conceitos e tecnologias que formam a base de um ecossistema de dinheiro digital. Sendo assim, para a compreensão da tecnologia e seu funcionamento, é necessário saber e compreender quais são os estes conceitos e tecnologias.

2.2.1 Transação

Um dos conceitos fundamentais para qualquer dinheiro e também para o *Bitcoin* é a ideia de transferência ou transação.

> Definimos uma moeda eletrônica como uma cadeia de assinaturas digitais. Cada proprietário transfere a moeda para o próximo, assinando digitalmente um hash das transações anteriores e a chave pública do próximo proprietário e as adicionando ao fim da moeda. Um beneficiário pode conferir as assinaturas para verificar a cadeia de propriedade. (NAKAMOTO, 2008, p. 2)

Como nesta definição inicial existem alguns termos que são técnicos, vale a pena entender a explicação a seguir:

> Em termos simples, uma transação informa para a rede que o dono de uma quantidade de bitcoins autorizou a transferência de alguns destes bitcoins para outro dono. O novo dono agora pode gastar esses bitcoins ao criar uma nova transação que autoriza a transferência para um outro dono, e assim por diante, em uma cadeia de posse de bitcoins. (ANTONOPOULOS, 2014, cap. Transações Bitcoin p. 4).

Após o entendimento da definição inicial de transação, é possível entender uma função peculiar e primordial para o *Bitcoin*: o encadeamento de transações. Cada transação define alguns dos critérios que serão avaliados sobre qualquer transação imediatamente posterior (que faz referência à anterior). Cada transação posterior tem a função de provar posse sobre os *bitcoins* que se pretende gastar.

Para Peck (2017), cada moeda gasta – atrelada a uma transação – está associada a uma chave pública. E a chave privada desta chave pública deve ser utilizada na assinatura digital de uma transação posterior. Esta última transação, também, deve associar uma nova – para ser utilizada posteriormente – chave pública.

Os termos utilizados por Peck (2017) e Nakamoto (2008), de assinatura digital e de chaves públicas e privadas, são termos utilizados na criptografia. Antonopoulos (2014) explica didaticamente que as chaves públicas e privadas são números inteiros com algumas relações matemáticas entre eles. Uma relação importante é que da chave privada, gera-se a chave pública, dada uma função de curva elíptica apropriada – curva que é a mesma utilizada por todos os usuários do *Bitcoin*. A chave pública é utilizada principalmente para se receber *bitcoins*. Já a chave privada, principalmente para gastar os *bitcoins* anteriormente recebidos – e associados à sua chave pública –, pois existe uma relação matemática entre ambas as chaves, no qual uma informação assinada digitalmente com uma chave privada pode ser verificada e validada pela chave pública.

2.2.2 *Blockchain* e Mineração

Segundo Chepurnoy (2016), as inovações utilizadas por Sakamoto foram as que resultaram em uma dinâmica de consenso emergente: da confiança da imutabilidade do histórico que aumenta exponencialmente sobre o período de existência de um dado em questão; das alterações atômicas do estado da rede por mudanças incrementais representadas por blocos; e a exclusão de encadeamentos alternativos de mudanças regido pela prova de trabalho total representada por cada encadeamento excludente entre si. Tais inovações envolvem os blocos – cada um constituído de um *header* e de um *payload* – e uma cadeia dos *headers* destes blocos. Cada bloco representa um incremento de informações na *blockchain*, de forma que novos nodos, aqueles fora de sincronia, podem atingir coerência com o restante da rede apenas pelos blocos posteriores ao último bloco comum.

Peck (2017) define a *Blockchain* como um livro-razão digital universalmente acessível, estruturado de tal forma que as alterações acontecem por adição de novas informações no final da cadeia de blocos, sendo cada adição um novo bloco, contendo informações como um conjunto de novas transações e uma referência para o bloco anterior.

Estruturalmente, cada bloco contém um *header* e um *payload*. Conforme foi postulado:

> Cada bloco contido na blockchain é identificado no cabeçalho do bloco por um hash, que é gerado utilizando-se o algoritmo criptográfico de hash SHA256. Cada bloco também contém uma referência ao bloco anterior, conhecido como o bloco pai, através do campo "hash do bloco anterior (previous block hash)" que existe no cabeçalho do bloco. Em outras palavras, cada bloco contém o hash de seu bloco pai no interior de seu próprio cabeçalho. A sequência de hashes ligando cada bloco ao seus pai cria uma corrente que pode ser seguida retrogradamente até o primeiro bloco que já foi criado, que é conhecido como o bloco gênese. (ANTONOPOULOS, 2014, cap. A Blockchain p. 1)

O *payload* de cada bloco adiciona transações à *blockchain*, que podem alterar os saldos dos usuários. É neste *payload* que a ordem das transações são definidas. Existindo a ordem de todas as transações, o gasto duplo – um caso problemático de quando participantes lidam com informações divergentes acerca dos seus saldos – é impossibilitado.

Um problema pertinente para um sistema que valida ou invalida transações sem uma autoridade central é evitar o gasto duplo, que, segundo Nakamoto (2008), ocorre quando não há um acordo sobre um histórico único entre todos os participantes e quando não há uma concordância em relação à ordem de todas as transações. Segundo ele, isto é resolvido quando cada transação é atrelada a um *timestamp* de um bloco. Nakamoto se inspirou em um sistema apresentado por Adam Back para implementar um servidor *timestamp* distribuído. Para isso, ele utilizou o conceito de *Proof-of-Work*, no qual aqueles que competem para adicionar novos blocos na *blockchain* conseguem provar que, na média, muitos ciclos de CPU foram gastos para esta tarefa.

É possível considerar o gasto duplo como um caso especial de *data-race*. Todo saldo que alguém tem em *bitcoin* depende do histórico em que este saldo se baseia, interpretado da *blockchain*. Todos os saldos remetem a um caminho de transferências dada por transações que existe até a criação de cada unidade de *bitcoin* (nas transações de *coinbase*), e também remetem, através do encadeamento dos *headers* dos blocos, ao bloco gênese, uma informação de estado inicial e fixa em todos os nodos de *Bitcoin*. Portanto o saldo não existe apenas pela última transação referente a este saldo na *blockchain*, mas a todo um encadeamento de transações. Imaginando que uma transação antiga desapareça, todas as transações que dependem daquela também desapareceriam devido à natureza do encadeamento de transferências do *Bitcoin*. Neste sentido, toda nova transferência é uma reafirmação da existência deste histórico de transações mais antigas no qual aquela nova transação se sustenta, e é uma afirmação de que este histórico persistirá (de que novas transações futuras poderão se basear nesta última). Esta organização estrutural delimita a possibilidade de *data-race* (e do gasto-duplo): quando alguma informação histórica é alterada.

A imutabilidade do histórico, que garante a verificação independente por cada nodo sobre cada transação e cada bloco passado – sendo isto uma função indispensável para um sistema descentralizado – é explanada por Chepurnoy (2016): O *header* de cada bloco tem uma referência válida para o *header* do bloco antecedente, uma referência válida às transações contidas no *payload* do bloco (chamado de *markle root*), e um número que é considerado, em conjunto com o próprio *header*, prova válida de dispêndio de esforço na criação do próprio bloco. Uma referência/prova perde a validade quando o que é referenciado/provado é alterado. Com este esquema, uma alteração em qualquer dado existente em um bloco antigo resultaria na invalidade da prova de esforço do bloco questão, e também na invalidade de toda a cadeia de *headers* posteriores àquele bloco. É então possível considerar dois estados distintos de cadeia de blocos: aquele inerte, antes da alteração sobre o dado antigo, e um alternativo, que inclui a alteração sobre o dado antigo. Este estado alternativo da cadeia de blocos não seria descartado pelos participantes da rede apenas se a somatória de dificuldade da prova de esforço contida nos blocos fosse maior do que no estado anterior (antes da tentativa de mudança sobre um dado antigo), o que só seria possível se o autor da alteração somasse à prova de esforço total do estado alternativo se re-criasse blocos posteriores à mudança, um feito mais custoso à medida da antiguidade do bloco alterado.

A prova de trabalho não só tem uso na escolha dentre cadeias válidas e reforçar a imutabilidade de acordo com a antiguidade, mas também para, em certo grau, garantir a discretização temporal da criação de blocos válidos pela rede como um todo. Isso tem utilidade para o controle da quantidade de informações que são inseridas no sistema e da quantidade de unidades monetárias que são criadas. Além disto, tal discretização simplifica a distribuição da escolha daquela entidade que, em um dado momento, define quais informações serão adicionadas no sistema – entidade chamada de *leader* por Chepurnoy (2016) – uma vez que diminui as chances

da existência de duas destas entidades em simultâneo.

A criação de blocos é chamada de mineração e o entendimento mais difundido reflete na explicação dada por Antonopoulos (2014), no qual a mineração é um processo onde os nodos na rede competem entre si para criarem novos blocos ao utilizarem hardware especializado para resolver os algoritmos de *Proof-of-Work*, gravando a solução encontrada no bloco gerado e assim provando o esforço do próprio nodo. Como apenas o nodo ganhador pode adicionar o bloco na *blockchain* e o mesmo consegue provar o trabalho que realizou para fazer esta inserção, a mineração garante a segurança do sistema *bitcoin* e permite a existência de um consenso em toda a rede sem a necessidade de uma autoridade central, uma vez que este bloco recém-criado é transmitido e aceito para e pelos outros nodos da rede.

2.2.3 Protocolo e Nodos

Segundo Bitcoin.org (2018), a rede padrão do *Bitcoin* se comunica pelo protocolo de rede P2P do *Bitcoin*. Tal comunicação acontece entre nodos via TCP, e contém dados que representam troca de mensagens, estas, com *header* de tamanho fixo e possivelmente com *payload*. Existem algumas constantes nos *headers*: *command*, *magic bytes*, *payload len* e *payload checksum* que servem para especificar a rede a ser utilizada, como a *Mainet* e *Testnet*, especificar o tipo da mensagem, tamanho em *bytes* do *payload* e o seu *checksum*, respectivamente. A versão do protocolo e os *bitflags* de *services* são informados na mensagem de *Version* e a troca desta mensagem representa o *handshake* entre dois *peers* no protocolo. A versão do procolo é importante para que um *peer* informe aos outros a sua capacidade comunicativa. Os *services* informa aos outros os serviços que aquele *peer* está disposto a fazer.

Outro conceito essencial para ao entendimento de *bitcoin* são os nodos que fazem parte da rede P2P do *Bitcoin*. Existem diversas funcionalidades para um nodo e cada nodo pode exercer um combinado destas funcionalidades. Segundo Antonopoulos (2014), são quatro os principais tipos funções: carteira, mineração, *blockchain* completa e nodo de roteamento. Onde os nodos de Cliente SPV, Nodo de Mineração e o nodo de cliente completo tem a função de roteamento segundo o protocolo P2P da rede do *Bitcoin*. Com a exceção do nodo de Cliente SPV, todos os nodos também mantém uma cópia local da *blockchain*, que pode ser utilizada para a verificação *offline* de todas as transações. O Nodo de Mineração faz parte do grupo de mineradores. O Cliente SPV oferece uma carteira ao usuário, uma interface que gerencia as chaves privadas e facilita a criação das mensagens de envio de transações.

O Nodo de Referência implementa todas as quatro funções e é chamado de *Bitcoin Core*, implementado na linguagem de programação C++. Embora possam existir diversas implementações de nodos que preencham estas funções, é possível verificar, como em Yeow (2018), que a grande maioria dos nodos funcionais na rede P2P do *Bitcoin* são variações do *Bitcoin Core* – aqueles com o campo *User Agent* com o valor de "Satoshi".

Segundo Newbery (2017) este nodo não possui um desempenho satisfatório mesmo em um ambiente com muitos *cores* de processamento, pois mesmo que o programa seja executando com múltiplas *threads*, muitas funções utilizam do recurso de *lock* global, bloqueando outras *threads* até a liberação do recurso. Ele ainda complementa que seria necessário reduzir a quantidade de chamadas globais para poder executar tarefas de carteira, validação de blocos e enviar e construir blocos e transações em si.

Uma alternativa ao uso corrente de *lock* global está sendo desenvolvido e é possível visualizar o seu avanço em Yanofsky (2017) do *Bitcoin Core*. Este *pull request* faz parte de um projeto maior de adaptar o nodo de referência para trabalhar com o modelo IPC (*Inter Process Communication*). Esta mudança está em desenvolvimento desde março de 2017.

2.3 *Rust*

A linguagem de programação *Rust* foi desenvolvida pela fundação Mozilla, e segundo Katz (2014), muitas das características de *Rust* foram implementadas em outras linguagens, o que a torna interessante é ter estas características todas juntas e ainda possuir fortes garantias.

Graças ao sistema de tipagem por posse e *borrowing*, o compilador, em vez de sempre considerar que o código do programador nunca conterá algum comando que resulte em um comportamento indeterminado (que é inseguro para o sistema e para o usuário), pode realizar mais análises sobre o uso das variáveis para evitar alguns dos seguintes comportamentos: *use-after-free*, *data races*, índice-fora-da-borda e invalidação de iteração (BEINGESSNER, 2015). Por ser uma linguagem recente, ela procura sanar, de maneira planejada, problemas que até recentemente eram pertinentes ou custosas em execuções concorrentes.

Data races acontecem quando duas ou mais *threads* concorrentemente acessam um local na memória de forma dessincronizada e onde alguma é de escrita (BEINGESSNER, 2015). Situações onde ocorrem *data races* precisam ser tratadas com grande cautela, visto que ao se negligenciar esta ação pode gerar situações de incoerência de dados, ocasionando respostas indesejadas. Portanto pode-se mostrar muito desejável uma linguagem em que esta situação de *data races* é totalmente evitada. O *use-after-free* acontece quando uma memória é acessada por alguma referência após ser liberada do *stack* ou *heap* (BEINGESSNER, 2015). Esta complicação geralmente ocorre quando as variáveis de ponteiro não são devidamente controladas, pois pode ser arduoso fazê-lo e fácil de se cometer enganos durante a programação.

O índice-fora-da-borda acontece quando um índice que extrapola os limites do tamanho de um *array* é utilizado para acesso de um elemento daquela *array*. De maneira similar, a invalidação de iteração acontece quando uma coleção de dados é alterada enquanto é iterada, onde o iterador pode usar dados desatualizados ou inválidos (BEINGESSNER, 2015). Definir uma verificação em *runtime* que visa evitar tais equívocos e garantir comportamentos completamente

definidos exige uma maior atenção do programador, pois nem sempre os compiladores, como os de C++, irão alertá-lo no caso de um equívoco no uso desta verificação em *runtime*. No caso do *Rust*, o programador poderá deixar algumas preocupações recaírem sobre os testes realizados pelo compilador.

Dado estas características, é possível inferir que *Rust* é uma nova linguagem de programação que conseguiu incorporar conceitos de outras linguagens e ainda, por meio dos conceitos de tipagem por posse e *borrowing*, evitar comportamentos indesejados isto sem custo adicional em tempo de execução.

2.3.1 *Ownership*, *Borrowing* e *Lifetime*

Um dos pontos atrativos de *Rust* são suas garantias para programação paralela, que oferece uma solução alternativa para o problema do gerenciamento de memória. Este problema pode ser tratado de maneira superficial em linguagens de alto nível que possuem *garbage collector*, uma vez que não exigem gerenciamento manual de memória por parte do programador.

Em *Rust*, graças ao sistema de *ownership*, *borrowing* e *lifetime*, a memória e os recursos são gerenciados automaticamente, onde o escopo de inicialização de memória é o proprietário (*owns*) daquela memória e que ela, quando não mais apropriada por ninguém, é automaticamente desalocada (e os recursos são liberados). Cada memória pode ter apenas um único proprietário (*owner*), mas também pode ser emprestada por tempo determinado para outros escopos. Nesta ocasião o código, exceto quando ocorrer indicações explícitas, será compilado com sucesso se é garantido que ou há apenas um escopo com acesso de escrita, ou que há apenas escopos com acesso de leitura sobre uma mesma memória ao mesmo tempo. Sem tais análises sobre o acesso de memória dos escopos, as linguagens que possuem *garbage collector* geralmente não aliviam o gerenciamento manual de recursos como quando é necessário fechar um recurso que foi aberto, como em abertura de arquivos ou *locks* de acesso. (KATZ, 2014)

As garantias de segurança supracitadas referem-se aos escopos que recebem análises de *ownsership* e *borrowing* do compilador da linguagem *Rust*. Devido à necessidade de implementação de estruturas de baixo nível, na prática, é necessário declarar escopos que não participam destas análises e, sendo assim, não tem quaisquer garantias de segurança e são declaradas como *unsafe*. Em direção a esta realidade, a Agência Executiva do Conselho Europeu de Investigação suporta projetos de pesquisas relacionados às demonstrações de análises formais e da validação das alegações de segurança em torno da linguagem *Rust*, principalmente dos escopos *unsafe*, como é anunciado em Dreyer (2018).

Um grande passo em direção a este objetivo foi dado por Jung et al. (2018), no qual um *subset* da linguagem *Rust* foi formalmente verificado e que várias bibliotecas importantes da linguagem *Rust* que possuem escopos *unsafe* são seguradamente encapsuladas pelo tipo imposto

às suas implementações.

2.3.2 *Traits*

Outra importante ferramenta contida em *Rust* são os *Traits*, que segundo Rust Book (2018), são um tipo de abstração parecidos com interfaces de outras linguagens, e que permitem abstrair sobre os comportamentos de tipos que os possuem em comum. Ao se definir *Traits*, é possível especificar funções e outras notações exigidas na implementação do *Trait* por algum tipo de dado. Parâmetros genéricos podem então implementar *Traits* e serem compilados com otimizações específicas para tipos concretos.

Mesmo *Traits* sendo apenas uma espécie de abstração ele ainda facilita e agrega novas funcionalidades à linguagem. Conforme informado por Turon (2016), ele ainda possibilita a resolução de uma variedade de problemas além da abstração em si, como: implementação de novos *Traits* para tipos já existentes; exigência da importação (e possível implementação expecificada) do *Trait* para o escopo que pretende acessar os métodos definidos por ele; implementação condicional, quando a implementação também faz uso de parâmetros genéricos; *static/dynamic dispatch* quando usado em assinaturas de funções; e uso para marcação, como quando é especificado se uma estrutura ou primitiva garante a possibilidade de identidade para quaisquer valores, ou quando é especificado se uma estrutura pode ser enviada entre *threads*, dentre outros.

2.3.3 *Crates*

Em *Rust*, *Crates* são equivalentes às bibliotecas e pacotes de outras linguagens de programação. A especificação do uso de bibliotecas e suas versões são feitas no arquivo *cargo.toml*, que podem referenciar *crates* do *website crates.io* ou referências diretas para projetos *git*, como projetos do *Github*.

A *crate Futures* estabelece o funcionamento de *Futures* para *Rust* e segundo Lerche (2018b), "um future é um valor que representa a conclusão de uma tarefa assíncrona", sendo que esta ação pode depender de um evento externo a este escopo. Segundo Turon (2016), a motivação do *Future* é ter um retorno imediato da chamada de um processamento ou evento, e possibilitar aplicações lógicas combinatórias sobre estas variáveis "futurísticas", gerando novos *Futures*. Ainda ressalta que outras linguagens possuem este mesmo conceito, destacando que, em *Rust*, por serem feitas com base em *Traits*, estas combinações são compiladas e não adicionam custo em *runtime*, exceto por uma alocação por cada *Task*, que representa a ideia de um *Future* que, como um todo, pode ser executado no contexto de uma *thread* não necessariamente exclusiva.

A *crate Tokio*, segundo Lerche (2018a), é um sistema de *runtime* que facilita a coorde-

nação de *Tasks* e *Futures* para as *threads* estabelecidas no contexto do *Tokio*, que conta com a propagação da fila de eventos do sistema operacional.

O site oficial desta *crate*, Lerche et al. (2018), demonstra uma aplicação básica de chat utilizando vários conceitos de *Futures*. Nesta demonstração, existe uma lógica de *codec* para linhas seguidas de uma quebra de linha compatível com uma aplicação de *telnet*.

A *crate Struct-Opt*, segundo Pinot (2018), permite a análise e conversão de argumentos de linha de comando a partir da definição de estruturas de comandos. Desta forma, o código para as conversões, os comentários para a interface por linha de comando e as sugestões e indicações de erros têm a sua implementação simplificada e organizada.

A *crate State Machine Future*, segundo Fitzgerald (2018), permite a implementação de *Futures* para uma máquina de estados a partir da definição da máquina e suas transições, definidas em uma enumeração com marcações, no qual é gerado um *typestate* para cada estado, ou seja, os estados são estruturas da máquina e tais estado-estruturas possuem métodos cujo a assinatura é composta de uma enumeração das transições a partir daquele estado. Este arranjo estrutural permite algumas garantias em tempo de compilação, como a garantia de que não há uso de transições inválidas e que, de acordo com o fluxo de transições definido nas marcações, todos os estados são atingíveis partindo do estado inicial e que qualquer estado pode atingir o estado final. A estruturação de cada estado permite a existência de variáveis internas a cada estado, oferecendo um isolamento de dados coerente com a lógica de transição. Finalmente, a implementação de *Future* para a máquina facilita seu uso como um ator na configuração *Actor Model*, em combinação com *Tokio*.

2.4 *Peer-to-Peer*

O *Bitcoin* foi desenvolvido sobre uma rede P2P devido os requerimentos de dinâmica e comportamento. Para Buford, Yu e Lua (2008), uma rede sobreposta *peer-to-peer* (P2P) é um conjunto de dispositivos (*peers*) interligados, onde os *peers* possuem funções iguais na rede tanto para rotear mensagens quanto para compartilhar recursos. Eles complementam que existem alguns princípios inerentes ao paradigma P2P: *peers* devem ser capazes se organizarem sem um agente central; os dispositivos determinam quando saem ou entram na rede ou quais as informações que serão requeridas, ou seja o *peer* é independente; para que um *peer* faça parte da rede, é necessário que ele faça contribuições de recursos que servem para manter o funcionamento da rede e prover serviços aos outros *peers*; cada *peer* tem uma visão incompleta da rede e portanto depende de outros *peers* para poder repassar uma mensagem para a região correta na rede, assim ele deve conseguir funcionar normalmente diante da perda de ligação com outros dispositivos.

O sistema do *Bitcoin* satisfaz todos estes requisitos, portanto possui as qualificações

mínimas para desempenhar como uma rede P2P.

2.5 *Actor Model*

Devido a crescente demanda por computadores com multinúcleos, é desejável utilizar suas capacidades de concorrência. Conforme Sutter (2012) informa, em apenas seis anos, desde o marco onde cada casa possuía o seu computador, todos os dispositivos possuem processadores com vários núcleos e este fato não mudará devido ao desempenho que eles oferecem. Um paradigma que explora o paralelismo para estes novos dispositivos é o *Actor model*.

Neste paradigma só é possível realizar um processamento a partir da análise de uma comunicação. Para Agha (1985) este modelo consiste de um ator que executa uma computação quando recebe uma comunicação (contido em uma tarefa), a partir desta noção o sistema pode criar atores e outras tarefas e terminar elas quando não tiverem mais uso. Um programa que utiliza este conceito deve possuir: comportamento definido ao se receber uma tarefa; a opção de criar atores quando necessário; criar tarefas ao executar um comando de envio para outro ator; possuir recepcionistas capazes de receber comunicação externa e ter representação de atores externos que não são definidos no programa em si.

2.6 Considerações Finais

Os assuntos abordados são de áreas diversas, e cada um, em seu próprio domínio, tem estudos e desenvolvimentos correntes e inovadores. Este capítulo buscou introduzir o leitor nas áreas mencionadas e é recomendado um estudo mais minucioso nas obras supracitadas. Também é possível constatar que vários dos conteúdos empregados, por serem recentes e/ou estarem ainda em desenvolvimento, podem se tornar obsoletos em um futuro próximo.

3 PROJETO DESENVOLVIDO

3.1 Considerações Iniciais

Conforme mencionado na subseção 2.2.3 – Protocolo e Nodos – Antonopoulos considera que um nodo de *Bitcoin* possui quatro tipos de funcionalidades principais: o de carteira, o da mineração, de armazenar/acessar e processar a *blockchain* completa e de nodo de roteamento. Utilizando esta modularização, o programa foi planejado e o desenvolvimento dele foi iniciado, no qual a ordem de implementação dos módulos escolhida foi: primeiro o módulo de roteamento, então os procedimentos necessários para atender às chamadas de comandos remotos e por fim, os principais elementos necessários à aquisição dos dados da *Blockchain*.

O funcionamento e a lógica de parte destas funcionalidades para o programa implementado serão mais aprofundados em seções específicas. O código da implementação é *open-source* (CETRULO; SILVA, 2018).

3.2 Nodo de roteamento com *Actor Model*

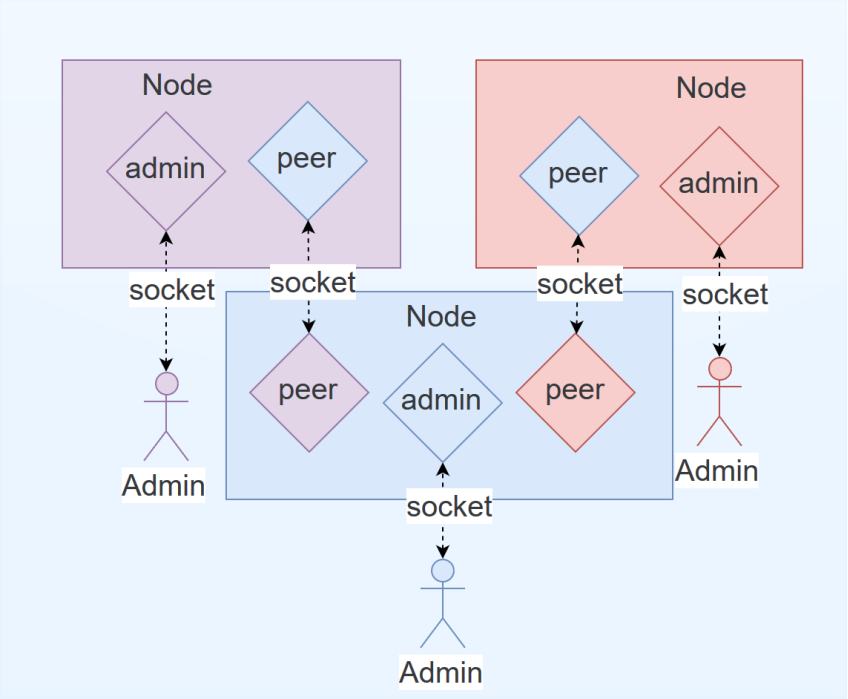
O programa foi desenvolvido sobre o protocolo P2P específico para o *Bitcoin*, conforme informado no subseção 2.2.3 – Protocolo e Nodos. Este protocolo é divido em duas partes: topologia do nodo em subseção 3.2.1 – Topologia do programa em *Actor Model* – e em subseção 3.2.2 – Comportamento do Programa – no qual a primeira refere-se apenas à topologia escolhida para tratar do assincronismo, e a segunda é referente às funcionalidades esperadas por um programa que interaja com a rede do *Bitcoin*.

3.2.1 Topologia do programa em *Actor Model*

O programa quando em execução é um potencial nodo – cliente/servidor – da rede *Peer-to-Peer* do *Bitcoin*. Como um *actor*, mantém três interfaces: escrita do *log* em um arquivo; escuta por novas conexões *TCP* por aplicações *telnet* para administradores do programa em uma porta; e escuta por novas conexões *TCP* por aplicações de outros nodos em outra porta. O seu comportamento, ao receber novas conexões administrativas ou de nodos, é o de criar *actors* de *admin* e de *peer*, respectivamente. Para evitar ambiguidade, é utilizado "nodo" para designar um programa executando em algum computador e "*peer*" para designar a representação interna (*actor*) de um programa executando em um computador externo.

3.2.1.1 *Actors* em uma rede P2P

Figura 1 – Usuários e *actors* em nodos diferentes.



Fonte: Os Autores (2018).

Na Figura 1 – Usuários e *actors* em nodos diferentes. – é mostrado uma topologia de conexão entre três nodos, e cada administrador, de uma cor, controla um nodo da mesma cor. Cada nodo mantém um *actor* do seu administrador, uma representação interna ao nodo. O nodo azul mantém uma conexão com os outros nodos, e representa cada um dos outros nodos através de *actors*, e os outros nodos criam tais representações de forma semelhante.

As informações salvas no arquivo de *log* são interferidas por *macros* na qual podem, através do compilador padrão, informar o horário, módulo, nível de depuração[1], arquivo e linha em que o comando se encontra no código.[2] Para isto as *macros* expandem em código que contém tais informações, efeito que contribui para a produtividade do desenvolvimento do programa.

3.2.1.2 Topologia Simplificada

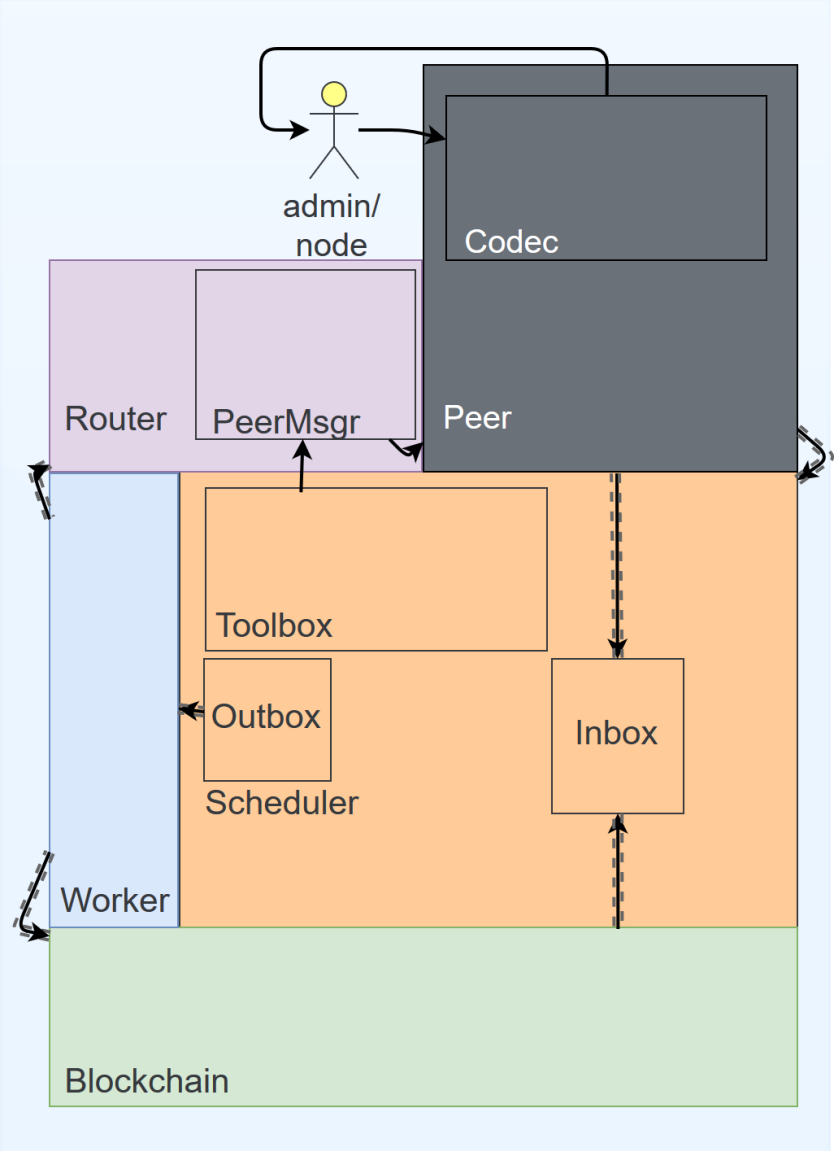
Para a análise da topologia de interação entre os *actors* internos ao sistema, convém uma representação gráfica delineando todos os tipos de *actors* e seus canais de comunicação.

[1] *Logs* podem ser completamente omitidos de forma seletiva de acordo com o seu nível de depuração.

[2] Exemplo:
`[17:40:32][btc][INFO] [/main.rs:101] New admin connection:  V4(127.0.0.1:60916).`

Os canais de comunicação entre *actors* distintos são representados por uma seta unidirecional, no qual o *actor* que inicia a mensagem é chamado de "ativo", e o que recebe a mensagem é chamado de "reativo". Se a seta do canal é tracejada, então o *actor* reativo irá gerar uma resposta às mensagens iniciadas pelo *actor* ativo através de um canal temporário de uso único. Os canais são representados por variáveis e são constituídos por partes transmissores e receptoras. Portanto os escopos (e atores) que podem transmitir e receber por um canal é determinado em tempo de compilação.

Figura 2 – Nodo simplificado utilizando *Actor Model*.



Fonte: Os Autores (2018).

A Figura 2 – Nodo simplificado utilizando *Actor Model*. – representa as principais interações entre os *actors* do sistema. Pela natureza isolada de *actors*, a interferência ou o compartilhamento de informação acontece exclusivamente por comunicação através de canais de mensagens.

Os *admins*/*peers* têm comunicações ativa e reativa, através de sockets, com os usuários ou o nodos externos. Tem também duas comunicações ativas com o *scheduler*, uma para enviar pedidos de processamento e outra para fins de cadastro internos do sistema. Finalmente, tem também uma comunicação reativa do *router*, caso algum *worker* queira enviar-lhe alguma mensagem. O *scheduler* tem comunicações ativas para os *workers* e o *router*, e comunicações reativas do *blockchain* e dos *admins*/*peers*. Os *workers* têm comunicações ativas com o *router* e *blockchain*, e comunicação reativa do *scheduler*. O *router* tem comunicações ativas com cada *admin*/*peer*, e comunicações reativas dos *workers* e do *scheduler*, este último para fins administrativos. O *blockchain* tem comunicação ativa com o *scheduler* e comunicações reativas dos *workers*.

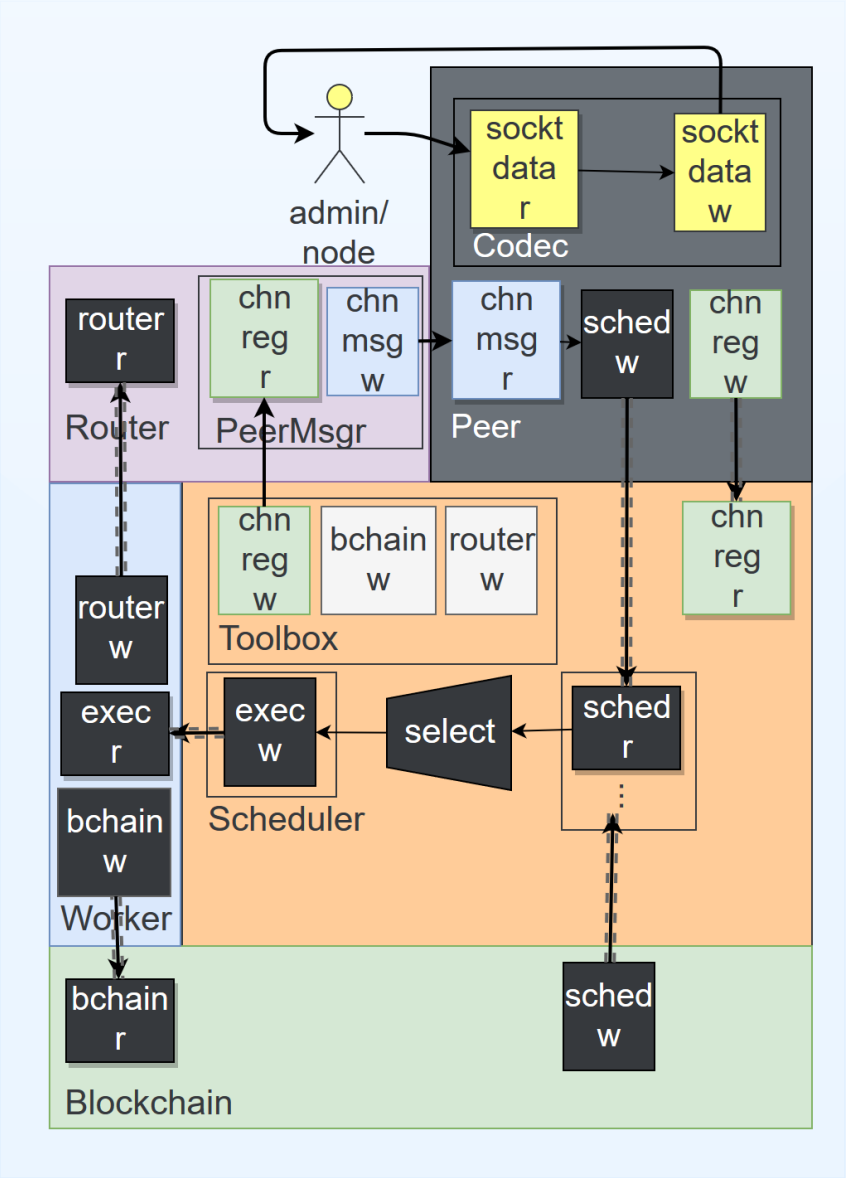
Com esta topologia de comunicação, distribuição de execução e acesso a dados, muitas execuções concorrentes são possíveis entre *workers* diferentes e o *blockchain*, desde que não precisem de um acesso exclusivo a uma mesma memória ou recurso.

O *scheduler*, em particular, é criado na inicialização do programa e termina apenas na terminação geral do programa, e uma de suas funções é coordenar os pedidos de execuções – assíncronos e concorrentes – que serão enviados pelos demais *actors*, que podem ter especificações de prioridade e sempre esperam por alguma resposta. Outra importante tarefa do *scheduler* é executar a política de criação, remoção e utilização de *workers*, *actors* que apenas executam o que lhes é pedido e produzem uma resposta de forma assíncrona. De forma semelhante, o *router* e o *blockchain* são criados na inicialização do programa e não têm previsão para término senão na terminação do programa. O primeiro tem por principal função fornecer uma comunicação com qualquer *admin*/*peer* conectado ao nodo. Já o segundo, tem por principal função o gerenciamento e interface para os dados dos blocos, transações e suas cadeias.

3.2.1.3 Topologia Completa

A Figura 3 – Nodo utilizando *Actor Model*. – representa as principais interações entre os *actors* do sistema de forma mais detalhada. Destaca componentes envolvidos nas comunicações ativas (componentes "w" – *writers* – ou "tx" – transmissores) e nas comunicações reativas (componentes "r" – *readers* – ou "rx" – receptores). As setas internas a um mesmo *actor* representam meramente fluxo de dados que existem entre a separação lógica dos componentes, ou seja, não representam envio de mensagens através de canais.

Figura 3 – Nodo utilizando *Actor Model*.



Fonte: Os Autores (2018).

3.2.1.3.1 *Admins* e *Peers*

O nodo não demanda que, antes ou após a inicialização, existam algum *worker*, nenhum *admin* e nenhum *peer*, mas apenas o *scheduler*, o *router*, o *blockchain* e o *actor* do nodo[3]. O *actor* do nodo recebe as novas conexões TCP em portas específicas, e inicia a configuração de um novo *actor*. Seja um administrador ou um outro nodo, este *admin/peer actor*, de início, tem três canais que existirão ao longo da existência do próprio *admin/peer actor*: *chn msg r* (azul), *sched w* (preto) e *chn reg w* (verde). O primeiro canal possibilita que o *admin/peer actor* receba

[3] Este ator não é representado nas figuras mas pode receber novas conexões TCP e tem um canal de comunicação com o *router*. Sua única função é realizar o cadastro de conexões que são iniciadas por outros nodos. Futuramente, existe a intenção de formalizar este ator em um arquivo separado no código do programa.

mensagens enviadas por um *worker*, por meio do *router*, *e.g.* como um efeito de um pedido realizado por um *admin*. O segundo canal permite que o *peer* envie pedidos de execução que serão realizados pelos *workers*.

Já o último canal tem uma função administrativa perante o *scheduler*, utilizado somente em momentos de cadastramento ou descadastramento de *actors*. É com o uso deste canal que um *admin* pode solicitar uma nova conexão com outro nodo, ao realizar um pedido a um *worker* que, em sucesso, cria uma conexão TCP e um novo *peer*. Como o *worker* deverá cadastrar este *peer* no *router*, o *admin* envia junto ao próprio pedido uma cópia da capacidade de transmissão[4] do canal *mpsc* (*multiple producer, single consumer*) de des/cadastro. O *worker*, nesta situação, não guarda esta capacidade de des/cadastro consigo, mas a transfere para o novo *peer* que está sendo criado. Também é com o uso deste canal que o *admin/peer* podem se descadastrar junto ao *router*.[5] Dado esta dinâmica possibilitada pelos pedidos, a topologia do nodo é controladamente variável, pois existem possibilidades de interações temporárias/momentâneas.

Cada pedido realizado por um *admin/peer* implica uma resposta, mesmo que ela não seja significativa para o *admin/peer*. Ao elaborar o pedido, o *actor* cria um canal *oneshot*[6], armazena a capacidade de receber o valor que será transmitido pelo canal (canal não demonstrado na figura) e envia a capacidade de transmitir o valor junto ao próprio pedido (não demonstrado na figura mas armazenado no *scheduler*, no *Inbox*).

3.2.1.3.2 *Scheduler*

O *scheduler* é o *actor* de interface entre os *admin/peer*, o *blockchain* e os *workers*, e também um *actor* que, no des/cadastramento de novos *admins/peers*, realiza comunicações administrativas com o *router*. Ele recebe vários pedidos de vários *admins/peers* e do *blockchain* através do *sched r* (em preto), lida com a carga de trabalho dos *workers* e também com a sua criação e destruição (mostrado como o componente *select*, em preto), encaminha os pedidos aos *workers* através dos *exec w* (em preto), recebe a resposta dos *workers* através de canais *oneshots* (armazenados no *Outbox*, porém não mostrados na figura) e encaminha as respostas de volta para os *admins/peers* ou *blockchain* através de canais *oneshots* (armazenados no *Inbox*, mas não representados na figura).

No *scheduler*, o *Inbox* representa uma listagem de vários canais referentes a cada *admin/peer* ou *blockchain*: um *sched r* para cada *peer/admin* e um *oneshot* (não representado na

[4] Portanto, em execução, a topologia pode diferir, de maneira controlada, daquela mostrada nas figuras de topologia.

[5] Devido ao fato de que não há interação com *actors* senão por mensagens, eles devem se encarregar de seu próprio descadastro, liberação de recursos e sua própria destruição de acordo com sua reação a mensagens específicas.

[6] De uso único.

figura) para cada resposta esperada por cada *admin/peer* ou *blockchain*[7]. O *Outbox* representa uma listagem de vários canais referentes a cada *worker*: um *exec w* para cada *worker* e um *oneshot* (não representado na figura) para cada pedido em execução, ou em fila de execução, por cada *worker*.[8]

O *scheduler* também armazena dois transmissores administrativos que são utilizados por *worker* e *admin/peer* para o *router* e o *blockchain*, que são clonados na criação de novos *actors*.

Por fim, o *scheduler* não interpreta nenhum pedido ou resposta. Isto simplifica a elaboração, execução concorrente e reutilização de pedidos por diferentes *actors*. Pelo fato do *scheduler* não precisar fazer esta interpretação, o *workflow* dos pedidos e respostas é simplificado, e o comportamento dos *actors* se torna mais previsível.

3.2.1.3.3 *Router*

O *router* tem a função de receber mensagens dos *workers* (através do *router r*, em preto) e encaminhar pedidos ou mensagens para *admins/peers* específicos, uma vez que possui a estrutura chamada de *Peer Messenger* que pode transmitir para cada *admin/peer*, inclusive em *broadcast* (através dos *chn msg w*). O *router* também tem um canal receptor para fins administrativos (através do *chn msg w*, em azul), sendo este utilizado para realizar o des/cadastro dos atores do *admins/peers*.

O *Peer Messenger* tem uma parte transmissora de vários canais, um para cada *admin/peer*. Por exemplo, quando um *admin* requisita a desconexão e remoção de um *peer*, um *worker* recebe e executa este pedido do *admin*, e retorna uma resposta a ele. Antes de responder, o *worker* realiza um pedido para o *router* para que o mesmo encaminhe o pedido de remoção de alta prioridade para o *peer* especificado. O *peer*, ao receber o pedido, se prepara para se remover: entra em um estado neutro (ver o estado *standby* em subseção 3.2.2.2 – *Peer Actor*) e inicia alguns procedimentos internos a ele, como esperar a finalização das mensagens que já estavam em vigor. Ao entrar em um estado seguro para sua destruição, inicia um desencadeamento administrativo com o *scheduler* para o descadastro de canais referentes a este *peer*. O *scheduler* propaga tal intenção para o *router*, que também realiza alguns descadastros. Finalmente, o *peer* tem seus recursos desalocados e liberados, como seu uso de *sockets*.

[7] Um *admin/peer* ou *blockchain* pode realizar vários pedidos e esperar por todos eles ao mesmo tempo, mantendo vários *oneshots* consigo.

[8] Um *worker* contêm uma fila de pedidos, mas executa um pedido por vez. Este arranjo permite que exista uma reserva de *workers* com uma reserva de *threads* com uma reserva de *cores* de processamento para execuções prioritárias, caso seja de interesse do gerenciador do nodo.

3.2.1.3.4 *Worker*

O *worker* é um *actor* que interage com o *scheduler*, *blockchain* e *router*, sendo este último utilizado para interagir indiretamente com os *admins/peers*. Ele mantém uma lista de pedidos a serem executados e executa o que for de maior prioridade. Após cada execução, o *worker* adquire pedidos novos enviados a ele pelo *exec r* (em preto), os ordenando na lista de pedidos. Depois da execução, prepara a resposta e a envia através do *shot w* (em lilás), no qual cada pedido é tratado isoladamente – o *worker* não tem alteração interna de estado como efeito de um pedido –, sendo desestruturado e então interpretado.

Ao longo de um procedimento de uma tarefa, o *worker* pode interagir com o *router* a fim de aquisição de informações sobre a lista de *admins/peers* que estão conectados, bem como realizar mensagens para serem propagadas para qualquer *admin/peer*. Além disto, pode também interagir com o *blockchain*, enviando informações que são consideradas imprevistas ou administrativas.

Futuramente, espera-se que novos tipos de *actors* existirão, e que poderão interagir com os canais do *router*. Neste caso, os *workers* poderão precisar de uma implementação por máquina de estado para que possam interagir com novos *actors* de forma assíncrona.

3.2.1.3.5 *Blockchain*

O *blockchain* é um *actor* que realiza procedimentos exclusivamente referente aos dados da *blockchain* do *Bitcoin*. Comporta-se através de uma máquina de estados e tem canais semelhantes aos *admins/peers*, no qual pode receber mensagens informativas ou administrativas (através do *bchain r*, em preto) ou indiretamente enviar pedidos aos *workers* (através do *sched w*, em preto).

Este *actor* tem seu comportamento definido por uma máquina de estados, porém sua implementação está em esboço.

3.2.2 Comportamento do Programa

O protocolo na implementação do nodo é dividido em três partes: *codec*, *peer actor* – representação interna de nodos externos – e *admin actor* que é a representação do usuário administrador conectado ao nodo, e por meio deste serão executados os comandos referentes à carteira.

3.2.2.1 *Codec*

O *codec* é um componente que gerencia o *socket* e realiza a abstração de dados, sendo assim, é preciso existir um tipo de *codec* para cada tipo de *actor* diferente que interage com um *socket*, um para o *admin* e outro para o *peer*. O *codec* do *admin* não possui funções de abstração de dados, pois apenas lida com a interface *telnet*, sendo assim apenas realiza a leitura do *socket* e gera um *string* até ler uma quebra de linha e quando necessário envia no *socket* uma *string*, utilizado para *feedback*.

O *codec* do *peer* trabalha com um *socket* direto com outro nodo onde as mensagem estão em bytes, e ele realiza o processo de abstração desta mensagem em estruturas que possuem significado (mais alto nível). A codificação ocorre apenas na construção de mensagem feita pelo *actor* do *worker* e a decodificação é feita ao receber uma mensagem no *socket*, ocorre quando o nodo referente àquele *peer* manda uma mensagem para o nodo.

O processo de decodificação começa quando chega uma quantidade de bytes no *socket* do *peer*, então é verificado se existem pelo menos 24 bytes de informação (tamanho do *header*) e caso existam é verificado se os bytes referentes a cada uma das variáveis do *header* está dentro do esperado[9] e com estas variáveis é construído o *header*. Com o valor da variável *payload size* será lido esta quantidade exata de bytes do *socket*, caso existam, e se os *checksums*[10] forem iguais é criado a abstração do *payload* e assim acaba o processo, pois já existe as abstrações necessárias para se construir a mensagem abstraída.

O processo de codificação é iniciado quando chega no canal do *worker* uma tarefa de envio de mensagem para algum nodo, é criado a abstração do *payload*, feita de acordo com o comando especificado, e gerado os bytes deste *payload* assim é calculado o seu *checksum* e seu tamanho e gerado a abstração do *header* a partir destas informações, agora é necessário apenas gerar o hexadecimal do *header*, a mensagem serializada é a concatenação do hexadecimal do *header* e do *payload*.

3.2.2.2 *Peer Actor*

Sempre quando é iniciado uma nova conexão com algum nodo é criado um *peer*, e após isto é iniciado a máquina de estados deste *actor*. Nesta máquina o seu estado inicial, chamado de *standby*, fica à espera de alguma notificação do *codec*. Assim que é recebido esta notificação são tratados os três canais que ele possui: o do *socket*, o do *scheduler*, canal que possui o *feedback* do *worker* em relação ao pedido realizado e pode ter a sua mensagem ignorada[11], e o do *router*,

[9] As variáveis *command* e *magic bytes* possuem um conjunto definido de valores.

[10] *Payload checksum* informado no *header* e *checksum* do *payload* lido.

[11] A mensagem pode ser ignorada pois este retorno não é a resposta de nenhuma ação e sim das comunicações, portanto não há necessidade de ser enviada para o outro nodo ou informado no servidor.

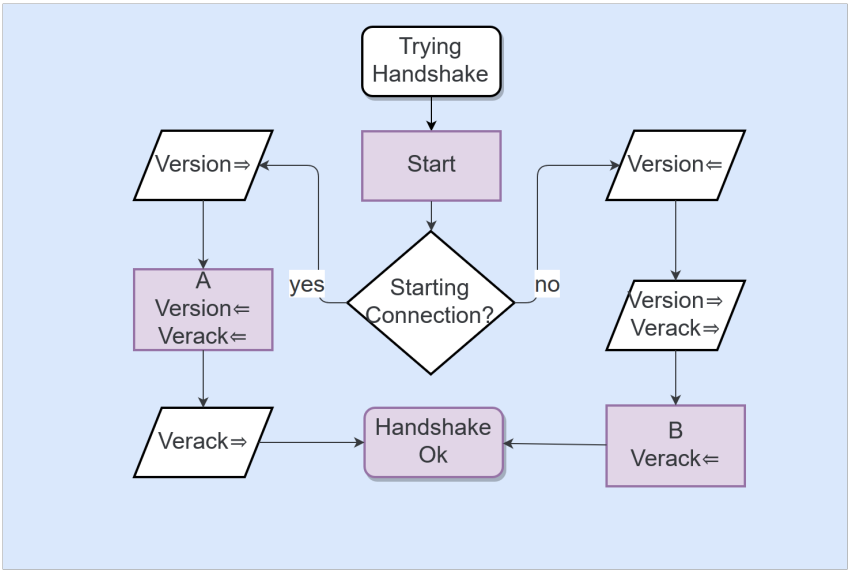
canal referente às tarefas que o *peer* deve realizar, enviadas diretamente pelo *worker*[12].

No estado de *standby*, após ser recebido qualquer notificação, são descartados todas as respostadas provenientes do *scheduler* e é verificado o conteúdo enviado pelo *router* e são executadas todas as tarefas existentes nele. A maioria das tarefas serão com relação ao envio de mensagem para o nodo relativo ao *peer*, no entanto mensagens diferentes precisam ser tratadas de forma diferentes, de acordo com o protocolo do *Bitcoin*, por exemplo, caso seja recebido uma tarefa de *ping* é necessário que seja esperado uma mensagem de resposta de *pong* do nodo.

Como existem diversos tipos de comandos e de protocolos, que podem ser demasiadamente complexos, é trocado de estado e este novo estado definirá como deverá ser a resposta e o que deverá ser esperado. Para os procedimentos mais complexos é iniciado um estado que iniciará uma nova máquina de estados interna para tratá-los, facilitando assim o entendimento.

Uma máquina de estados interna define o comportamento necessário para se realizar o *handshake*, segundo os parâmetros estabelecidos pelo protocolo da rede de *Bitcoin*, visto que este é um comportamento mais complexo que demanda de três estados bem distintos e com tomadas de decisões. Na Figura 4 – Maquina de estados do *Handshake* – é mostrada esta representação, sempre que é iniciada a conexão com um novo nodo é dado início nesta máquina, sendo possível existir duas situações: quando o nodo em si iniciou a conexão ou quando um nodo externo iniciou a conexão TCP – situações representadas pelos diferentes caminhos após a pergunta "*Starting Connection?*".

Figura 4 – Maquina de estados do *Handshake*



Fonte: Os Autores (2018).

Caso o nodo inicie o processo, é feito um pedido de criação da mensagem de "*Version*"

[12] Um exemplo de tarefa é a *rawmsg*, onde deve ser enviado a mensagem hexadecimal recebida neste canal para o *socket*.

para um *worker* e assim que ele retorna tal mensagem ela é enviada para o respectivo nodo e então é trocado de estado. No próximo estado o *peer* fica esperando até receber ambas as mensagens de reconhecimento da mensagem de "*Version*" – chamada de "*Verack*" – como da mensagem de "*Version*" do nodo ao qual está se conectando, caso ambas as mensagens forem recebidas e estejam válidas é pedido para o *worker* criar a mensagem de "*Verack*" do nodo externo assim que o *worker* retorna esta mensagem ela é enviada para o nodo e finalizado a máquina de estados.

Caso a conexão seja iniciada pelo próprio nodo, ou seja, quando é recebida uma mensagem válida de "*Version*" é feito um pedido ao *worker* do "*Verack*" desta mensagem de "*Version*" e também da mensagem de "*Version*" do nodo em si, assim que o *worker* retorna ambas estas mensagens elas são enviadas para o nodo que iniciou o processo e é trocado o estado desta máquina interna. Neste novo estado é esperado uma mensagem de "*Verack*", relacionado a mensagem de "*Version*" enviada previamente, quando chega esta mensagem e ela é válida o processo de *handshake* é finalizado com sucesso.

Com exceção do *handshake*, que é realizado apenas uma vez no momento de inicialização da máquina de estados do *peer*, todos os procedimentos são cíclicos, terminando apenas quando a conexão *peer*-nodo acaba.

3.2.2.3 *Admin Actor*

De maneira similar ao *actor* do *peer*, existe uma máquina de estados para controlá-lo com a diferença de que a sua conexão não é com um nodo de *Bitcoin* mas sim com uma interface *telnet*. Assim, este também permanece no estado inicial esperando alguma notificação do *codec*. Este *actor*, no entanto, possui apenas dois canais de comunicação: o do *scheduler*, sendo que o *feedback* do *scheduler* neste caso é importante[13], e o do *socket*.

Quando chega alguma informação no *socket* do *admin* gerenciado pelo *codec*, a máquina é notificada e inicializa o procedimento. O primeiro passo é a comparação interna, derivada pela *crate struct-opt*, da *string* obtida com o argumento enviado pelo administrador e, caso sejam iguais, é criado a estrutura da requisição e é modificado de estado. Caso contrário, é apresentado o *help* do comando que falhou de volta para o administrador.

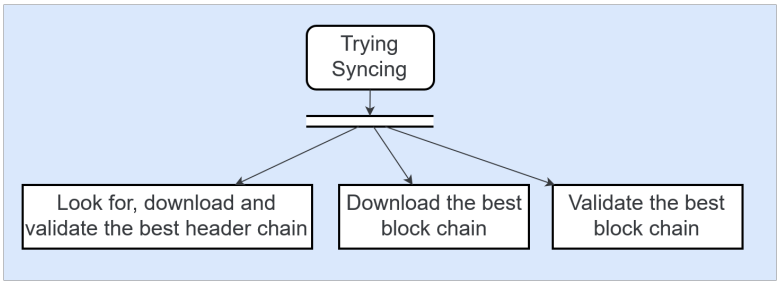
Na maioria dos casos o pedido do *admin* possui um *workflow* simples, onde é enviado uma requisição ao *scheduler* e ele espera o *feedback* para enviar ao *codec*. Após isto, ele retorna ao estado inicial e, para estes casos, é definido um estado chamado de *simpleWait* que define este comportamento. Para o restante dos casos, assim como na máquina do *peer*, é criado um estado para cada tipo de pedido e, caso exista um muito complexo, é utilizada uma máquina de estados interna.

[13] Informar os resultados do comando executado pelo *worker* na interface *telnet*.

3.3 *Blockchain* completo

Este módulo consiste em um *actor* que se comporta de acordo com uma máquina de estado e realiza funções essenciais que envolvem os dados de uma *blockchain* de *Bitcoin*. A definição do comportamento e da máquina estão em fase de prototipação. Inicialmente, as funções que foram necessárias de se implementar foram representadas por máquinas internas (estados da máquina externa) e estão listadas na Figura 5 – Máquina de estados *Syncing* – processo de sincronização dos dados referentes à *blockchain* do *Bitcoin*.

Figura 5 – Máquina de estados *Syncing*



Fonte: Os Autores (2018).

O processo de sincronização é parte essencial de uma criptomoeda, para que o nodo possa verificar quaisquer recebimentos e realizar envios com base nestes recebimentos. Tal processo envolve o *download*, armazenamento, verificação, validação e gerenciamento dos blocos e demais informações. Com a existência dos requisitos de assincronia e concorrência, o comportamento esperado deste ator se mostrou mais complexo do que o inicialmente esperado.

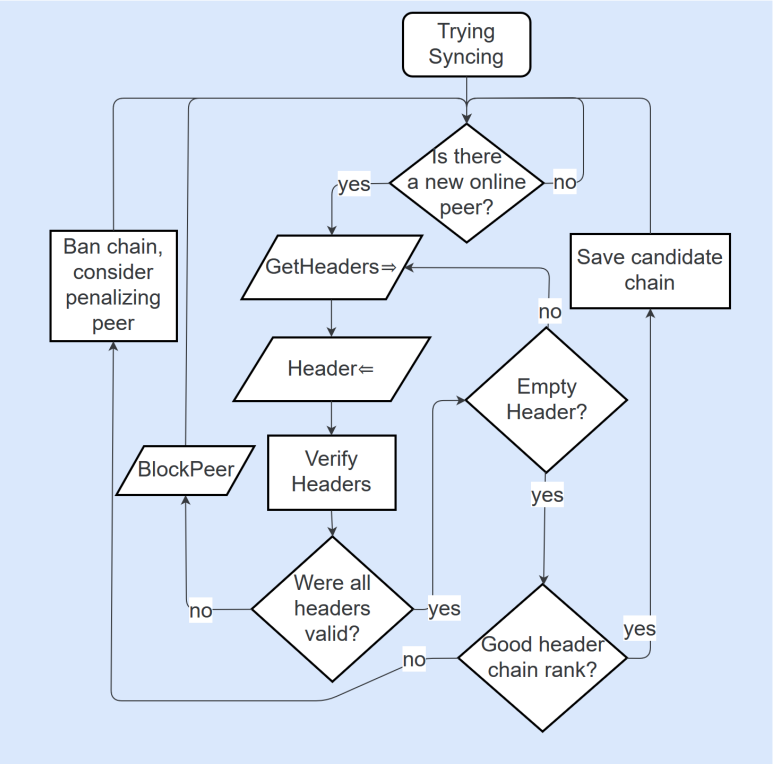
Devido à necessidade dos nodos de escolherem apenas a *blockchain* que tiver a maior somatória de *Proof-of-Work*, o nodo deve lidar com a competição entre várias *blockchains*, que são estruturas que podem ter pontos de intersecção uma com as outras, tornando-as complexas. Portanto o nodo desenvolvido irá sempre desconfiar da *blockchain* que segue, mesmo que de fato for a seguida pelos demais nodos, de um modo geral. Assim, o nodo deverá sempre buscar por encadeamentos alternativos e verificar se algum é potencialmente superior à cadeia preferida. Neste aspecto, tal função é executada indefinidamente[14], pois mesmo que todos os *peers* da rede já tivessem sido consultados, não há garantias de que algum *peer* terá uma *blockchain* com maior *Proof-of-Work* no futuro. Portanto, foram esboçadas as principais funções relacionadas ao processo de sincronização e também foi destacado que tais funções podem, dependendo da situação em que o nodo se encontra, ser executadas de maneira paralela.

A Figura 5 – Máquina de estados *Syncing* – mostra as três funções principais do processo de sincronização: (a) *Look for, download and validate the best header chain*; (b) *Download the*

[14] Mesmo sendo uma função complexa, a sua execução, em vários momentos distintos, não seria prejudicial ao desempenho do nodo, visto que o processo de sincronização para nodos que possuem a mesma *blockchain* consiste apenas no pedido de uma mensagem de "*Header*"e sua verificação.

best block chain e (c) *Validate the best block chain*. A separação de dois tipos de *downloads* e um tipo de verificação e validação é possível graças à própria estrutura da *blockchain* do *Bitcoin*, no qual os *headers* são até 4 ordens de magnitude menor do que os blocos e podem indicar informações valiosas para tomadas de decisão.

Figura 6 – Máquina de estados *Syncing Headers*



Fonte: Os Autores (2018).

Um esboço da função (a) é demonstrado na Figura 6 – Máquina de estados *Syncing Headers* – no qual o nodo lida com as possibilidades de estar desconectado, de ser vítima de nodos que enviam dados inválidos ou indesejáveis[15]. A finalidade deste procedimento é obter um conjunto de *header chains* com um potencial de *Proof-of-Work* maior do que a melhor *header chain* já verificada (neste caso, com os blocos também verificados e validados).

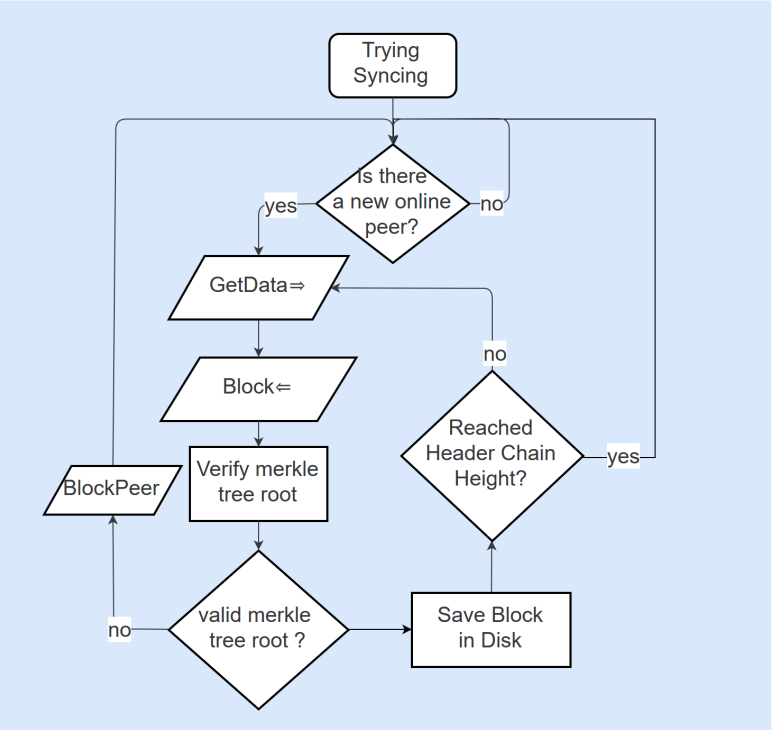
Obtendo-se a *header chain* com o maior potencial, é desejável iniciar o *download* dos blocos referentes à esta cadeia de *peers* específicos e que corroboram com esta cadeia – processo (b). Porém, também pode ser desejável que o processo (a) recomece o seu próprio ciclo.

O processo (b), com um comportamento funcional que pode ser visto na Figura 7 – Máquina de estados *Syncing Blocks* (*download*) – possui alguns requisitos funcionais, como: muitos *downloads* em paralelo dos blocos com um ou mais *peers* – sempre da *header chain* com maior potencial; penalização de *peers* com comportamentos indesejados; a escrita em disco

[15] Portanto, o nodo deve buscar economizar recursos de processamento e disco e banda de rede e então deve penalizar a conexão com tais *peers*.

dos blocos salvos; e a possibilidade da liberação de recursos em disco em caso de detecção de invalidade. Tais funcionalidades visam realizar o *download* e armazenamento de todas as informações indicadas pela melhor *header chain* em conhecimento do nodo – *download* dos blocos completos.

Figura 7 – Máquina de estados *Syncing Blocks* (*download*)



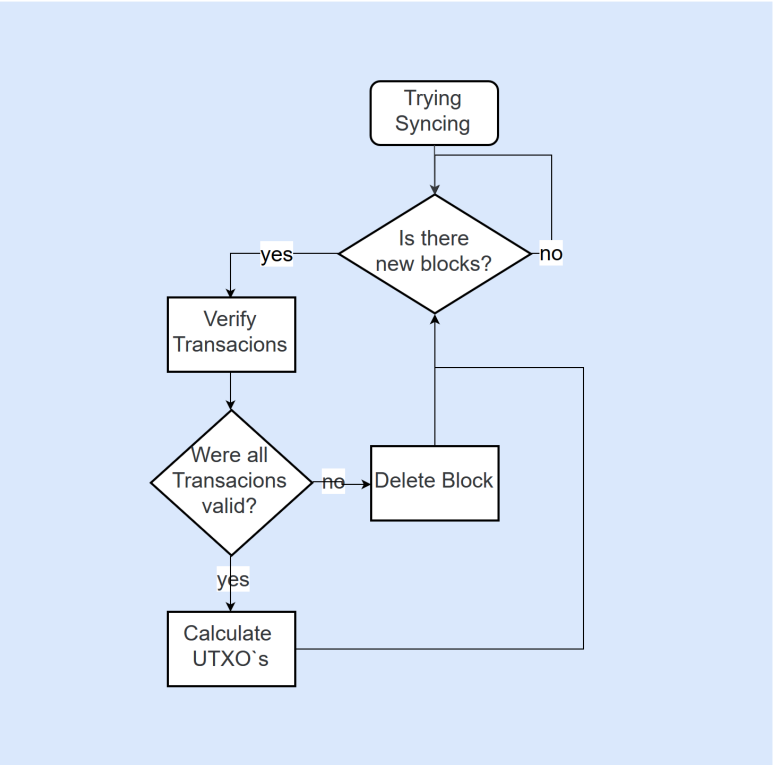
Fonte: Os Autores (2018).

O processo (c) pode ser visualizado na Figura 8 – Máquina de estados *Syncing Blocks* (validação) – e também teve alguns requisitos inicias, como: realizar verificações em paralelo, quando possível, na modalidade *master-slave* das transações de um bloco individualmente; lidar com re-verificações caso sejam necessárias, quando um mesmo bloco possuir duas ou mais transações digitalmente encadeadas; fazer uso da biblioteca padrão de consenso[16] para o corpo e *script* das transações; realizar validações parciais de blocos posteriores ao último bloco validado de uma cadeia totalmente validada, para potencialmente aproveitar recursos disponíveis; atualizar o conjunto *UTXO* a cada bloco validado; dentre outros.

Existem outras funções que não estão relacionadas com a sincronização do nodo (em primeira pessoa), mas sim como uma interface de serviço para exploradores e administradores de *blockchain*, carteiras e outros *peers* da rede. Tais procedimentos não foram elaborados nem implementados por este trabalho.

[16] Escrita em *C++* e de difícil e improdutivo reprodução em outras linguagens, do *Bitcoin Core*, é um componente destacável deste programa e suporta as verificações do histórico do *Bitcoin* e abrange muitos casos de validade ou invalidade desta criptomoeda.

Figura 8 – Máquina de estados *Syncing Blocks* (validação)



Fonte: Os Autores (2018).

3.4 Considerações Finais

O planejamento e implementação de um programa crítico são beneficiados pela modularização de seus componentes, que é a direção desejada por este trabalho.

A escolha das dependências das bibliotecas foram feitas de acordo com as necessidades específicas de cada componente. Por exemplo, não foi esperado nenhum benefício ao se atrelar, para o *scheduler*, uma definição comportamental baseado em máquinas de estados, diferentemente dos *admins/peers*.

Ferramentas e ambientes como o editor de texto *VSCode* com *plugins* relacionados à linguagem *Rust*, *git* e *Github* contribuíram com a produtividade e organização da implementação, que teve seu início como estudos básicos da linguagem de programação *Rust* – tarefa que exigiu um esforço considerável apesar da experiência com linguagens como *C++* e *Java*.

As partes já planejadas e implementadas são essenciais para um nodo completo de *Bitcoin*, e sua estruturação presume a inserção de novas funções e casos de usos, facilitando o desenvolvimento futuro do programa.

4 CONCLUSÃO

4.1 Interfaces e Comandos Remotos

O nodo em execução apresentou quatro tipos de interfaces que puderam ser utilizadas para a visualização de informação, e três para serem utilizadas para a entrada de informação. A primeira interface, mostrada na Figura 9a – Nodo: Compilação e Inicialização – é a comum de um programa *CLI* onde, por um ambiente de linha de comando, o usuário pode visualizar as saídas que o nodo imprime. Nesta interface, a única entrada aceita é o da escolha de parâmetros, ao se iniciar o nodo. Uma vez iniciado, o usuário da máquina pode terminar o processo forçadamente de acordo com a configuração de seu ambiente (`Ctrl+C`, por padrão no console do *Windows*). A segunda interface é o arquivo salvo como uma impressão idêntica àquelas informações que são impressas na primeira interface, no qual um usuário da máquina pode ler as informações do arquivo. A terceira interface, mostrado na Figura 9b – *Admin*: Conexão e Uso – é a interface remota de comunicação, uma conexão *telnet* via *TCP* com os administradores, os quais podem se relacionar com o nodo de maneira similar àquela dos programas *CLI*, com inserção de comandos, subcomandos e parâmetros e, possivelmente, leitura das respostas textuais do nodo. A quarta interface é uma conexão segundo o protocolo *P2P* do *Bitcoin*, via *TCP*, no qual um nodo externo pode conectar-se a ela e realizar determinas interações com o nodo, segundo o que é previsto na máquina de estado da sua representação interna de um *peer*[1].

É por meio da primeira, segunda e da terceira interfaces – textos no terminal do nodo, os arquivos de *log* e mais textos no terminal *telnet* respectivamente – que os resultados foram observados. A produção de resultados deriva da interação intencional administrativa, por comandos do *admin*.

As interfaces, quando mostradas em figuras neste trabalho, tiveram o formato de seu conteúdo textual adaptado para o ambiente deste documento, e seu propósito: comandos administrativos são mostrados com um cifrão (`$`) à sua esquerda para destaque; espaços de alinhamentos podem ter sido substituídos por acentos de til (~) para melhor visualização; e estruturas com vários níveis de estruturas internas podem conter novas linhas e espaçamentos extras, para facilitar a visualização. Não houve nenhuma alteração do conteúdo textual que não seja estes supracitados.

[1] Esta interface não foi mostrada por ser uma representação serializada do protocolo de comunicação da rede *P2P* do *Bitcoin*.

Figura 9 – Inicialização e conexão de *admin*

(a) Nodo: Compilação e Inicialização

```
$ cargo run
~~~ Finished dev [unoptimized + debuginfo] target(s) in 0.17 secs
~~~ Running target\debug\btc.exe\
[02:49:03][btc][INFO] [\main.rs:273] server_peer running on V4(127.0.0.1:8333)
[02:49:03][btc][INFO] [\main.rs:274] server_admin running on V4(127.0.0.1:8081)
```

(b) *Admin*: Conexão e Uso

```
$ telnet localhost 8081
WELCOME
$ help
Command could not be executed
Cause:  HelpDisplayed
Message:
~0.1.0
Felipe Cetrulo <fecetrulo@hotmail.com>:Thiago Machado da Silva <thi@unifei.edu.br>
Adiministrative commands
USAGE:
~~~~ <SUBCOMMAND>
FLAGS:
~~~ -h, -help ~~~~~ Prints help information
~~~ -V, -version ~~ Prints version information
SUBCOMMANDS:
~~~ bchain ~~ Blockchain-related tasks
~~~ debug ~~~ Utilities-related tasks
~~~ exit ~~~~ Exits the CLI and disconnects the admin peer
~~~ help ~~~~ Prints this message or the help of the given subcommand(s)
~~~ node ~~~~ Node-related tasks
~~~ peer ~~~~ Peer-related tasks
~~~ util ~~~~ Utilities-related tasks
~~~ wallet ~~ Wallet-related tasks
Aditional Info:
None
```

Fonte: Os Autores (2018).

4.2 Nodo de Roteamento

A funcionalidade do *codec* foi observada através da interfaces com o *admin* como mostrado em Figura 10a – Mensagem Serializada em um Comando e Estruturada em um *Log* – onde, via comandos de *admin*, foi dado um comando de `debug msg -hex`[2] e então observados os dados em um formato estrutural e, se possível, validado. Pôde-se também visualizar erros caso ocorresse algum problema na desserialização, incluindo a cadeia de dependências de todas as estruturas atreladas àquele erro – como mostrado na Figura 10b – Mensagem Serializada Errada. Uma das maneiras de se testar alguns tipos de des/serialização é através do comando `cargo test`, comando específico de um ambiente padrão do *Rust*, para que as funções atribuídas com a marcação *test* sejam executadas e suas falhas, caso ocorram, sejam relatadas. As mensagens serializadas de teste contidas no projeto foram obtidas da documentação em Bitcoin.org (2018).

A funcionalidade de roteamento foi observada através de impressão textual em arquivos de *log*, os quais foram produzidos indiretamente de um comando de um *admin*, como `peer add`

[2] Comando utilizado para realizar a deserialização de uma mensagem em bytes, neste caso era uma hexadecimal de uma mensagem de "*Ping*".

Figura 10 – Inicialização e Conexão de *Admin*

(a) Mensagem Serializada em um Comando e Estruturada em um *Log*

```
$ debug msg -hex F9BEB4D970696E6700000000000000000800000088EA81760094102111e2af4d
[04:30:28][btc::actor::worker][DEBUG] [\actor\worker.rs:230] New Debug message received Message:
- Message Header:  Message Header:
- Message Network Identification:  Main <D9B4BEF9>
- Message Command OP_CODE: <ping\0\0\0\0\0\0\0\0>
| - 112, 105, 110, 103, 0, 0, 0, 0,
| - 0, 0, 0, 0,
| |
- Payload Length:  8
- Payload Checksum:  1988225672
- Message Payload:
| Ping:
| - Nonce:  5597941425041871872
```

(b) Mensagem Serializada Errada

```
$ debug msg -hex F9BEB4D970696E6700000000000000000800000088EA81760094102111e2af4a
[04:30:34][btc::actor::worker][WARN] [\actor\worker.rs:254] Something wrong on msg to all peerErr(Error(
~~~ Msg(
~~~~~~~ "[\\codec\\msgs\\msg\\mod.rs:178] Error at payload checksum (expected:  1988225672, found:
1551673675)"
~~~ ),
~~~ State {
~~~~~~~ next_error:  None,
~~~~~~~ backtrace:  None
~~~ }
))
```

Fonte: Os Autores (2018).

`-addr=189.38.88.105:8333`[3], um procedimento que, como um todo, imprime informações ao longo de sua existência no arquivo de *log* – como mostrando em Figura 11a – Iniciando Conexões com *Peers*. De início, o *worker* inicia uma conexão *TCP* e então, sendo este um comportamento esperado, o *peer* transmite os dados de "*Version*" contendo algumas informações que auxiliarão as comunicações futuras, como a especificação da versão do protocolo que o nodo, em primeira pessoa, se baseia. Durante cada recebimento de uma mensagem via *socket*, é impresso informações relativas a ela, como se o número de *bytes* necessários para o conhecimento do *header* já estão disponíveis em um *buffer* específico, e se a mensagem recebida é válida segundo algumas verificações iniciais. Portanto, pela observação de informações referentes à transmissão e recebimento de mensagens e valores válidos que foram produzidos pelo nodo externo, infere-se de que a funcionalidade de roteamento funcionou como o esperado.

Após verificada a capacidade de conexão do nodo desenvolvido com outros nodos da rede, foi testado a sua capacidade de se conectar simultaneamente a múltiplos nodos. para isto, o *admin* iniciou os comandos para a conexão com vários nodos externos e então entrou com o comando `peer list`[4] – como mostrado em Figura 11b – Listagem de *Peers* – observando assim, em sua interface, uma lista coerente com as conexões que o nodo efetuou.

[3] Comando referente adição de um nodo, onde o terceiro parâmetro referente-se ao endereço de IP de um nodo externo. Este nodo, utilizado neste caso, foi o nodo brasileiro com o melhor *ranking* geral no site Yeow (2018).

[4] Comando que retorna a quantidade de *peers* e *admins* conectados e suas informações básicas.

Figura 11 – Roteamento com *Peers*

(a) Iniciando Conexões com *Peers*

```
$ peer add -addr=189.38.88.105:8333
Ok(WorkerResponseContent(
~~ PeerAdd(Some(V4(189.38.88.105:8333))),
~~ AddrReqId(1,1))
)
$ peer add -addr=167.99.107.210:8333 Ok(WorkerResponseContent(
~~ PeerAdd(Some(V4(167.99.107.210:8333))),
~~ AddrReqId(1,2))
)
```

(b) Listagem de *Peers*

```
$ peer list
Ok(WorkerResponseContent(
~~~ ListPeers({
~~~~~~ 1:  V4(127.0.0.1:60895),
~~~~~~ 3:  V4(167.99.107.210:8333),
~~~~~~ 2:  V4(189.38.88.105:8333)
~~~ }),
~~~ AddrReqId(1, 3)
))
```

Fonte: Os Autores (2018).

4.3 Estruturação da *Blockchain*

Os procedimentos do *download* dos dados da *blockchain* giram em torno tanto do *codec* e roteamento quanto do *actor blockchain*. Este, possuindo uma máquina própria e os canais necessários para conseguir se comunicar com os *workers* e portanto realizar pedidos para envios de mensagens para outros nodos através de seus *peers*, pode criar expectativas de respostas específicas para *peers* específicos, podendo assim, tanto realizar *upload/download* de informações quanto avaliar a conexão e comportamento dos outros *peers* a fim de indicar para o nodo como um todo que pode ser desejável iniciar conexões com novos nodos e terminar conexões com nodos específicos.

É importante ressaltar que, apesar da existência de funcionalidades do entorno deste procedimento específico existirem, o próprio *actor* permaneceu em faze de planejamento comportamental e não teve tal implementação concluída. As funcionalidades que estão implementadas são as elaborações, des/serializações e comunicações das mensagens de *GetHeaders* e *GetData* e das respostas esperadas "*Headers*" – como mostrado em Figura 12 – Mensagem *Headers* Desserializada de um *Peer* – e "*Block*", respectivamente, que são utilizadas para requisitar e receber, respectivamente, informações do encadeamento de cabeçalhos e dos blocos da *blockchain*.

4.4 *Memory-safety* e Assincronismo

O código implementado do nodo foi compilado sem erros pelo compilador da linguagem *Rust*. Devido ao fato do código não declarar escopos *unsafe* de forma explícita, pode-se

Figura 12 – Mensagem *Headers* Desserializada de um *Peer*

```
[01:42:53][btc::codec::msgs][DEBUG] [\codec\msgs\mod.rs:113] Finished building msg recieved:
Message:
- Message Header:  Message Header:
- Message Network Identification:  Main <D9B4BEF9>
- Message Command OP_CODE: <headers\0\0\0\0\0>
| - 104, 101, ~97, 100, ~101, 114, 115, ~ 0,
| - ~ 0, ~ 0, ~ 0, ~ 0,
| |
- Payload Length:  162003
- Payload Checksum:  1999330214
- Message Payload:
| Headers {
~~~ count:  U16(2000),
~~~ headers:  [
~~~~~~~ BlockHeaders {
~~~~~~~~~~ version:  1,
~~~~~~~~~~ prev_block:  [111, 226, 140, 10, 182, 241, 179, 114, 193, 166, 162, 70, 174, 99, 247, 79, 147, 30,
131, 101, 225, 90, 8, 156, 104, 214, 25, 0, 0, 0, 0, 0],
~~~~~~~~~~ markle_root:  [152, 32, 81, 253, 30, 75, 167, 68, 187, 190, 104, 14, 31, 238, 20, 103, 123, 161,
163, 195, 84, 11, 247, 177, 205, 182, 6, 232, 87, 35, 62, 14],
~~~~~~~~~~ timestamp:  1231469665,
~~~~~~~~~~ bits:  486604799,
~~~~~~~~~~ nonce:  2573394689
~~~~~~~ },
~~~~~~~ BlockHeaders {
~~~~~~~~~~ version:  256,
~~~~~~~~~~ prev_block:  [0, 72, 96, 235, 24, 191, 27, 22, 32, 227, 126, 148, 144, 252, 138, 66, 117, 20, 65,
111, 215, 81, 89, 171, 134, 104, 142, 154, 131, 0, 0, 0],
~~~~~~~~~~ markle_root:  [0, 213, 253, 204, 84, 30, 37, 222, 28, 122, 90, 221, 237, 242, 72, 88, 184, 187,
102, 92, 159, 54, 239, 116, 78, 228, 44, 49, 96, 34, 201, 15],
~~~~~~~~~~ timestamp:  1723642011,
~~~~~~~~~~ bits:  16777033,
~~~~~~~~~~ nonce:  3184658461
~~~~~~ },
~~~~~ ...
~~~ ]
}
```

Fonte: Os Autores (2018).

deduzir que tal programa trouxe fortes garantias de ser *memory-safe*. Dos escopos *unsafe* das bibliotecas utilizadas como dependências, segundo as pesquisas referenciadas em seção 2.3 – *Rust* – metodologias de verificação estão atualmente sendo elaboradas e executadas, de forma que existe a possibilidade da garantia de segurança de memória por construção ser fortalecida de acordo com o resultado destas pesquisas.

Durante a implementação, foi despendido de forma recorrente uma grande quantidade de esforço para resolver problemas de *lifetimes* dos escopos com referência e *ownsership* sobre memórias, e como resultado deste processo, que se mostrou através de impedimentos de compilações, foi obtido as garantias por construção supracitadas.

A funcionalidade de assincronismo foi observada através do comando de *admin* `debug wait 10`[5] seguido do comando `debug wait 1`, como observado na Figura 13 – Teste simples de assincronia. – e sendo impresso na interface do *admin* respostas em ordem contrária aos comandos enviados, que puderam ser observadas no campo `AddrReqId`[6]. Portanto, verificou-se

[5] Quando este comando é recebido por um *worker* o mesmo espera o tempo (em segundos) representado pelo terceiro parâmetro da entrada.

[6] Variável de uma estrutura-tupla, que possui os valores de identificadores do *actor* e do número do pedido daquele *actor* feito ao *worker*.

um caso em que o processo desempenhou uma tarefa de forma assíncrona, onde o *admin* não precisou bloquear uma *thread* enquanto esperava pelas respostas de pedidos arbitrários que ainda seriam executados.

Figura 13 – Teste simples de assincronia.

```
$ debug wait -delay=10
$ debug wait -delay=1
[04:16:26][btc::actor::admin::machina][INFO] [\actor\admin\machina\mod.rs:207] started wait cmd
[04:16:26][btc::actor::worker][DEBUG] [\actor\worker.rs:97] Request received:  Wait
[04:16:30][btc::actor::admin::machina][INFO] [\actor\admin\machina\mod.rs:207] started wait cmd
[04:16:30][btc::actor::worker][DEBUG] [\actor\worker.rs:97] Request received:  Wait
[04:16:31][btc::actor::worker][DEBUG] [\actor\worker.rs:282] response sent.
[04:16:31][btc::actor::admin][INFO] [\actor\admin\mod.rs:58] Oneshot response arrived, and got ignored:
Ok(
~~~ Ready(
~~~~~~~ Ok(
~~~~~~~~~~ WorkerResponseContent(
~~~~~~~~~~~~~~~ Empty,
~~~~~~~~~~~~~~~ AddrReqId(
~~~~~~~~~~~~~~~~~~~ 1,
~~~~~~~~~~~~~~~~~~~ 3
~~~~~~~~~~~~~~~ )
~~~~~~~~~~ )
~~~~~~~ )
~~~ )
)
[04:16:36][btc::actor::worker][DEBUG] [\actor\worker.rs:282] response sent.
[04:16:36][btc::actor::admin][INFO] [\actor\admin\mod.rs:58] Oneshot response arrived, and got ignored:
Ok(
~~~ Ready(
~~~~~~~ Ok(
~~~~~~~~~~ WorkerResponseContent(
~~~~~~~~~~~~~~~ Empty,
~~~~~~~~~~~~~~~ AddrReqId(
~~~~~~~~~~~~~~~~~~~ 1,
~~~~~~~~~~~~~~~~~~~ 2
~~~~~~~~~~~~~~~ )
~~~~~~~~~~ )
~~~~~~~ )
~~~ )
)
```

Fonte: Os Autores (2018).

4.5 Desfecho e Trabalhos Futuros

O desenvolvimento deste estudo permitiu mostrar a idealização e a construção de partes essenciais de um nodo alternativo da rede *P2P* do *Bitcoin*, sendo que este desenvolvimento ofertou a possibilidade que pode resultar em um nodo completo com vantagens sobre o nodo mais utilizado atualmente, principalmente pela sua execução assíncrona e paralela devido à topologia principal que utiliza *Actor Model* em sua construção e as garantias de segurança que são inerentes à linguagem de programação *Rust*. Os resultados obtidos foram satisfatórios e ocorreram da forma como foi esperada, corroborando com as premissas do desenvolvimento. Existem vantagens secundárias deste projeto como a possibilidade de desenvolvimento incremental cujo cada resultado pode sempre ter sua memória – no âmbito de *data-race* e *dangling pointers* – ter fortes garantias em tempo de compilação, a possibilidade de um aumento da diversidade de nodos na rede e o uso de ferramentas que estão em constante evolução.

O trabalho foi desenvolvido incrementalmente, sendo compilado com sucesso durante fases intermediárias, com o objetivo de facilitar e auxiliar futuros desenvolvimentos. Portanto, este possui a intenção futura de disponibilizar um nodo completo assíncrono e *memory-safe* para uso gratuito e irrestrito. Para isto, será necessário continuar o desenvolvimento de funções ainda primordiais para o funcionamento de um nodo completo, como: a validação total das transações através de bibliotecas como indicado em Wiki (2017) e dos blocos; o gerenciamento da *blockchain* como planejado no seção 3.3 – *Blockchain* completo; a *API* específica para carteiras e exploradores de *blockchain*; mineração opcional de blocos; e testes exaustivos de segurança do nodo e de suas funcionalidades.

REFERÊNCIAS

AGHA, G. A. **Actors**: A model of concurrent computation in distributed systems. 1. ed. [S.l.]: MIT Press, 1985. Citado na página 24.

ANTONOPOULOS, A. M. **Mastering Bitcoin**. 1. ed. [S.l.]: O'Reilly Media, 2014. Citado 5 vezes nas páginas 12, 15, 16, 17 e 19.

BECH, M.; GARRATT, R. Central bank cryptocurrencies. 2017. Disponível em: <https://www.bis.org/publ/qtrpdf/r_qt1709f.htm>. Acesso em: 19 abr. 2018. Citado na página 15.

BEINGESSNER, A. **You can't spell trust without Rust**. 2. ed. [s.n.], 2015. Disponível em: <https://github.com/Gankro/thesis/blob/master/thesis.pdf>. Acesso em: 19 abr. 2018. Citado 2 vezes nas páginas 12 e 20.

BITCOIN.ORG. **Bitcoin Developer Reference**. 2018. Disponível em: <https://bitcoin.org/en/developer-reference>. Acesso em: 19 abr. 2018. Citado 2 vezes nas páginas 19 e 41.

BUFORD, J. F.; YU, H.; LUA, E. K. **P2P Networking and Applications**. 1. ed. [S.l.]: Morgan Kaufmann Publishers, 2008. Citado na página 23.

CETRULO, F. B.; SILVA, T. M. da. **RustBTC**. 2018. Disponível em: <https://github.com/swfsql/rustbtc>. Acesso em: 22 mai. 2018. Citado na página 25.

CHEPURNOY, A. The blockchain and the scorex tutorial. 2016. Disponível em: <https://github.com/ScorexFoundation/ScorexTutorial/blob/master/scorex.pdf>. Acesso em: 19 abr. 2018. Citado 2 vezes nas páginas 17 e 18.

DREYER, D. **European Research Council Project "RustBelt"**. 2018. Disponível em: <https://plv.mpi-sws.org/rustbelt/>. Acesso em: 22 mai. 2018. Citado na página 21.

FEUER, A. **Prison May Be the Next Stop on a Gold Currency Journey**. 2012. Disponível em: <https://www.nytimes.com/2012/10/25/us/liberty-dollar-creator-awaits-his-fate-behind-bars.html>. Acesso em: 27 maio 2018. Citado na página 14.

FITZGERALD, N. **Crate state machine future**. 2018. Disponível em: <https://docs.rs/state_machine_future/0.1.6/state_machine_future/>. Acesso em: 19 abr. 2018. Citado na página 23.

JUNG, R. et al. Rustbelt: Securing the foundations of the rust programming language. 2018. Disponível em: <www.mpi-sws.org/~dreyer/papers/rustbelt/paper.pdf>. Acesso em: 23 mai. 2018. Citado na página 21.

KATZ, Y. **Rust Means Never Having to Close a Socket**. 2014. Disponível em: <http://blog.skylight.io/rust-means-never-having-to-close-a-socket/>. Acesso em: 19 abr. 2018. Citado 2 vezes nas páginas 20 e 21.

LERCHE, C. **Crate tokio**. 2018. Disponível em: <https://docs.rs/tokio/0.1.5/tokio/>. Acesso em: 19 abr. 2018. Citado na página 22.

LERCHE, C. **Tokio**. 2018. Disponível em: <https://tokio.rs/>. Acesso em: 19 abr. 2018. Citado na página 22.

LERCHE, C. et al. **Example: A Chat Server**. 2018. Disponível em: <https://tokio.rs/docs/getting-started/chat/>. Acesso em: 19 abr. 2018. Citado na página 23.

LIGHTBEND. **Lightbend Case Studies**: akka. 2018. Disponível em: <https://www.lightbend.com/case-studies#tag=akka>. Acesso em: 19 abr. 2018. Citado na página 12.

NAKAMOTO, S. Bitcoin: A peer-to-peer electronic cash system. 2008. Disponível em: <https://bitcoin.org/bitcoin.pdf>. Acesso em: 19 abr. 2018. Citado 3 vezes nas páginas 10, 16 e 17.

NEWBERY, J. **What did Bitcoin Core contributors ever do for us?** 2017. Disponível em: <https://medium.com/@jfnewbery/what-did-bitcoin-core-contributors-ever-do-for-us-39fc2fedb5ef>. Acesso em: 19 abr. 2018. Citado 2 vezes nas páginas 12 e 20.

PARTZ, H. **Is CryptoRuble Back?**: Launch set for mid-2019, says russian blockchain association. 2018. Disponível em: <https://cointelegraph.com/news/is-cryptoruble-back-launch-set-for-mid-2019-says-russian-blockchain-association>. Acesso em: 20 abr. 2018. Citado na página 11.

PECK, M. E. **Blockchains**: How they work and why they'll change the world. 2017. Disponível em: <https://spectrum.ieee.org/computing/networks/blockchains-how-they-work-and-why-theyll-change-the-world>. Acesso em: 19 abr. 2018. Citado 3 vezes nas páginas 11, 16 e 17.

PINOT, G. **TeXitoi/structopt**: Parse command line argument by defining a struct. 2018. Disponível em: <https://github.com/TeXitoi/structopt>. Acesso em: 19 abr. 2018. Citado na página 23.

ROTHBARD, M. N. **Depressões Econômicas**: A causa e a cura. 1969. Disponível em: <https://www.mises.org.br/Article.aspx?id=228>. Acesso em: 19 abr. 2018. Citado na página 10.

ROTHBARD, M. N. **O Que o Governo Fez Com o Nosso Dinheiro?** 1. ed. Mises Brasil, 2013. Disponível em: <http://rothbardbrasil.com/wp-content/uploads/arquivos/nossodinheiro.pdf>. Acesso em: 19 abr. 2018. Citado 2 vezes nas páginas 10 e 11.

RUST BOOK. **The Rust Programming Language**. 2018. Disponível em: <https://github.com/rust-lang/book>. Acesso em: 19 abr. 2018. Citado na página 22.

SUTTER, H. **A Heterogeneous Supercomputer in Every Pocket**. 2012. Disponível em: <https://herbsutter.com/welcome-to-the-jungle/>. Acesso em: 19 abr. 2018. Citado na página 24.

TURON, A. **Zero-cost futures in Rust**. 2016. Disponível em: <https://aturon.github.io/blog/2016/08/11/futures/>. Acesso em: 19 abr. 2018. Citado na página 22.

ULRICH, F. **Bitcoin**: A moeda na era digital. 1. ed. LVM EDITORA, 2014. Disponível em: <http://rothbardbrasil.com/bitcoin-a-moeda-na-era-digital/>. Acesso em: 20 abr. 2018. Citado na página 14.

VON MISES, L. **Ação Humana**: Um tratado de economia. 3.1. ed. 1966. Disponível em: <http://rothbardbrasil.com/acao-humana-um-tratado-de-economia-42/>. Acesso em: 19 abr. 2018. Citado 2 vezes nas páginas 10 e 11.

WIKI, B. **Libbitcoin Consensus**. 2017. Disponível em: <https://en.bitcoin.it/wiki/Libbitcoin_Consensus>. Acesso em: 27 maio 2018. Citado na página 46.

YANOFSKY, R. **[experimental] Multiprocess bitcoin by ryanofsky**: Pull request #10102 - bitcoin/bitcoin. 2017. Disponível em: <https://github.com/bitcoin/bitcoin/pull/10102>. Acesso em: 19 abr. 2018. Citado na página 20.

YEOW, A. **Network Snapshot**. 2018. Disponível em: <https://bitnodes.earn.com/nodes/>. Acesso em: 19 abr. 2018. Citado 2 vezes nas páginas 19 e 42.