UNIVERSIDADE FEDERAL DE ITAJUBÁ - *CAMPUS* ITABIRA

FELIPE BALIEIRO CETRULO
THIAGO MACHADO DA SILVA

NODO DE BITCOIN: CLIENTE COMPLETO EM RUST

ITABIRA
2018

FELIPE BALIEIRO CETRULO
THIAGO MACHADO DA SILVA

NODO DE BITCOIN: CLIENTE COMPLETO EM RUST

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

NODO DE BITCOIN: CLIENTE COMPLETO EM RUST

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. E especialmente ao professor 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 modificados para que possua uma menor quantidade de bloqueios de execução concorrente. Devido este tipo de modificação ser delicado e devido a necessidade da execução paralela, que resultaria em um maior uso da capacidade total das computadores de múltiplos núcleos, é desejável que exista uma implementação alternativa do nodo que tenha garantias de segurança de memória. 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*. Pretende-se compreender 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 o nodo possa ser planejado, implementado e utilizado, realizando o *download* da *Blockchain* e fazendo parte da rede da criptomoeda.

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

ABSTRACT

The Bitcoin Core, the most popular Bitcoin node, is being changed in order to have fewer concurrent excecution thread blocks. Due this type of modification being delicated, and due to the parallel execution needed, which results in a higher usage from 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. Such guarantees are inherent to the Rust language, which with uncommon concepts, offers tools that, in compile time, prevent some memory class errors, such as data-races and use-after-free. It is intended to comprehend the overall concepts related to the Bitcoin ecosystem, it's Blockchain and a P2P node's function in order to plan, implement and utilize a new node, whereas it will be able to download the Blockchain and be part of the crypto's network.

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

LISTA DE ILUSTRAÇÕES

LISTA DE ABREVIATURAS E SIGLAS

Arc	*Atomic Reference Counter*
BTC	*Bitcoin*
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*

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, com o uso de um objeto cuja aceitação e portabilidade fossem 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 das grande maioria das pessoas.

1.1 Encontro com a Escola Austríaca de Economia e a presente situação econômica

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 refe à *Blockchain*. Como é dito por Peck (2017), muitos dos maiores bancos uniram forças para criar novos 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 bilhares de dólares), além de alterar a organização dos componentes internos do sistema de forma a oferecer vantagens em relação ao sistema implementado mais popular.

1.2 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 da separação modular dos componentes de um nodo 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 nodo completo de *Bitcoin* que seja assíncrono e cuja execução seja *memory-safe*?

Uma ferramenta que traz garantias quanto à segurança de memória é 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*.

Uma das formas de organização 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 de um nodo similar a um *Bitcoin Core* pode apresentar grandes benefícios para

a tecnologia da rede da moeda virtual.

Com base neste conceito e também nesta linguagem de programação, busca-se uma resposta à pergunta supracitada, e desta forma, a hipótese contida neste trabalho pode ser descrita como: uma implementação prática de um nodo de *Bitcoin* utilizando *Actor Model* e a linguagem de programação *Rust* possui as características desejadas de assincronia e segurança de memória.

Neste sentido, este trabalho tem por objetivos: obter um nodo completo de *Bitcoin* que seja assíncrono e cuja 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* e verificação de informações externas pertinentes à *Blockchain*; e realizar testes de *stress* individualmente ao Nodo.

Para que os objetivos sejam atingidos, será implementado um nodo conforme especificado, no qual as mensagens sobre gerenciamento de memória serão analisadas, o nodo executará suas funções e será estressado[1] por comandos e *actors* que dividirão os mesmos recursos da máquina. 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 nodo completo. Por fim, no capítulo 4, será apresentado uma conclusão sobre os resultados obtidos da implementação do nodo.

[1] Teste de *stress*.

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* e *Lifetime*, as *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 uma moeda privada 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 na ausência de uma autoridade central, obrigando a rede a manter coerência sobre o estado dos saldos de todos usuários, considerados por cada usuário, 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". Ele ainda acrescenta que "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 seguir 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.

> 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 pretende-se 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*, e a chave privada, para gastar os *bitcoins* recebidos para a 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 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 do dado em questão; das alterações do estado da rede por mudanças incrementais representadas por estruturas atômicas; e a exclusão do encadeamentos alternativos de mudanças regido pela prova de trabalho total representada por cada encadeamento excludentes entre si. Tais inovações envolvem os blocos, cada um constituído de um *header* e de de *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 possivelmente 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 à 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 setido, 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 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 a o 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 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 local 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 em sites online,

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 servir montar blocos e transações si.

Uma alternativa ao uso corrente de *lock* global está sendo desenvolvido e é possível visualizar o seu avanço através de Yanofsky (2017) do *Bitcoin Core*. Este *pull* 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 destes comportamentos, tais como: *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 onde alguma é de escrita e alguma é dessincronizada (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 muito pequeno ou grande, em relação ao tamanho de um *array*, é acessado. 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 a verificação em *runtime* que visa evitar tais equívocos e garantir comportamentos completamente definidos pressupõe uma maior atenção por parte do programador, pois nem sempre os compiladores de C++ mais comuns irão avisá-lo de tais descuidos. 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 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 ocorrer indicações explícitas, será compilado com sucesso apenas se é garantido que há apenas um ou nenhum escopo com acesso de escrita sobre uma mesma memória ao mesmo tempo, em contraposição à linguagens que possuem *garbage collector*, elas geralmente não aliviam o gerenciamento manual de recursos como quando é necessário fechar um recurso que foi aberto. (KATZ, 2014)

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

funcionalidade 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 especificação) 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 parâmetros, e também para retorno de métodos ou funções (este último em experimentação); e uso para marcação, como: especificar se uma estrutura ou primitiva garante a possibilidade de identidade para quaisquer valores; se pode ser enviada entre *threads*, dentre outros.

2.3.3 *Crates*

Em *Rust*, *Crates* são equivalentes às bibliotecas e pacotes de outras linguagem 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, porém 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 coordenação de *Tasks* e *Futures* para as *threads* estabelecidas no contexto do *Tokio*, que conta com a propagação da fila de eventos do sistema operacional.

Um 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 tem 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; é necessário que para um *peer* fazer parte da rede ele deve fazer 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 desejado utilizar suas funcionalidades. 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 irá mudar devido o desempenho que eles oferecem. Um paradigma que 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 novos 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 novos atores quando necessário; criar novas tarefas ao executar um comando de envio para outro ator; possuir recepcionista 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 do nodo foi iniciado, no qual a ordem de implementação dos módulos escolhida foi: primeiro o nodo de roteamento, então a *blockchain* completa e por último a carteira. O funcionamento e a lógica de cada uma destas funcionalidades para o nodo implementado será mais aprofundado nos respectivos capítulos.

3.2 Nodo de roteamento com *Actor Model*

O nodo deve ser 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 *Actor Model* e Protocolo do nodo, no qual a primeira refere-se apenas à topologia escolhida para tratar com assincronismo, e a segunda é referente às funcionalidades esperadas por um nodo da rede do *Bitcoin*.

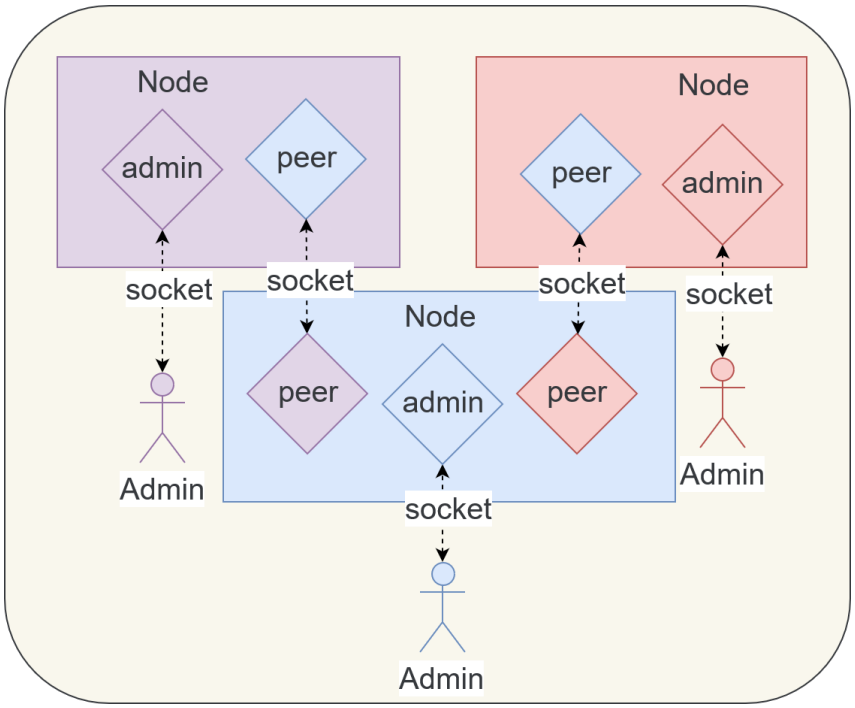
3.2.1 Topologia do nodo em *Actor Model*

O programa quando em execução é um potencial nodo 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 nodo 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 novos *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 nodo externo.

3.2.1.1 *Actors* em uma rede P2P

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.

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



Fonte: Autoria própria.

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

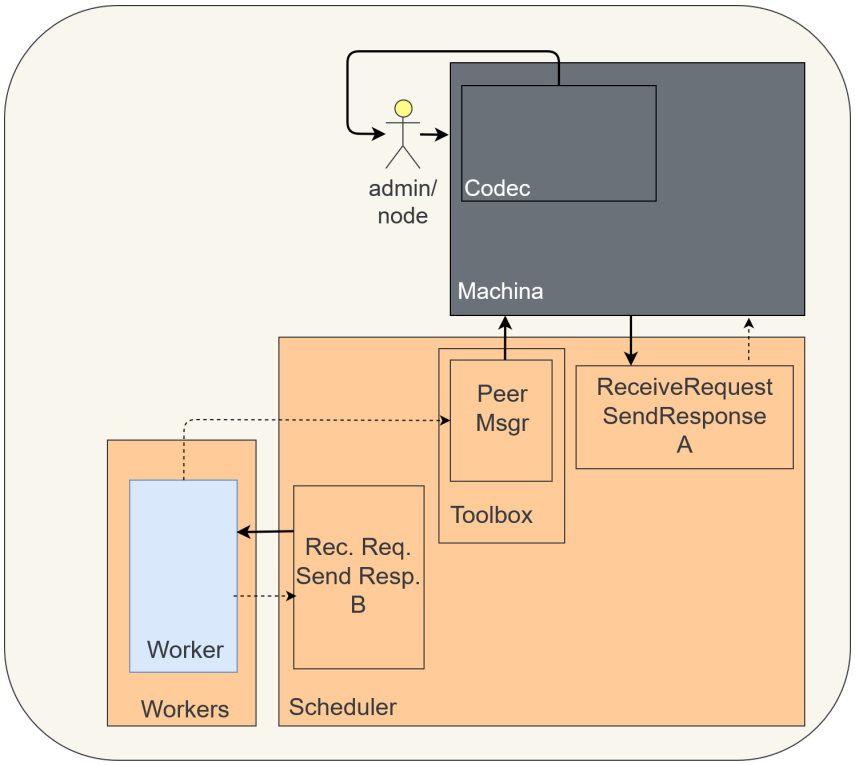
Para a análise da topologia de interação entre os *actors* internos ao sistema, convém uma representação gráfica delineando todos os tipos de *actors* e seus canais de comunicação.

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

Os *admins*/*peers* não tem nenhuma interferência que não aquelas de seus canais de comunicação e de *sockets* que veem de usuários/nodos externos. O *scheduler* e os *workers*

[1] *Logs* podem ser completamente omitidos de forma seletiva de acordo com o seu nível de depuração.
[2] *E.g. [17:40:32][btc][INFO] [/main.rs:101] New admin connection: V4(127.0.0.1:60916).*

Figura 2 – Nodo simplificado utilizando *Actor Model*.



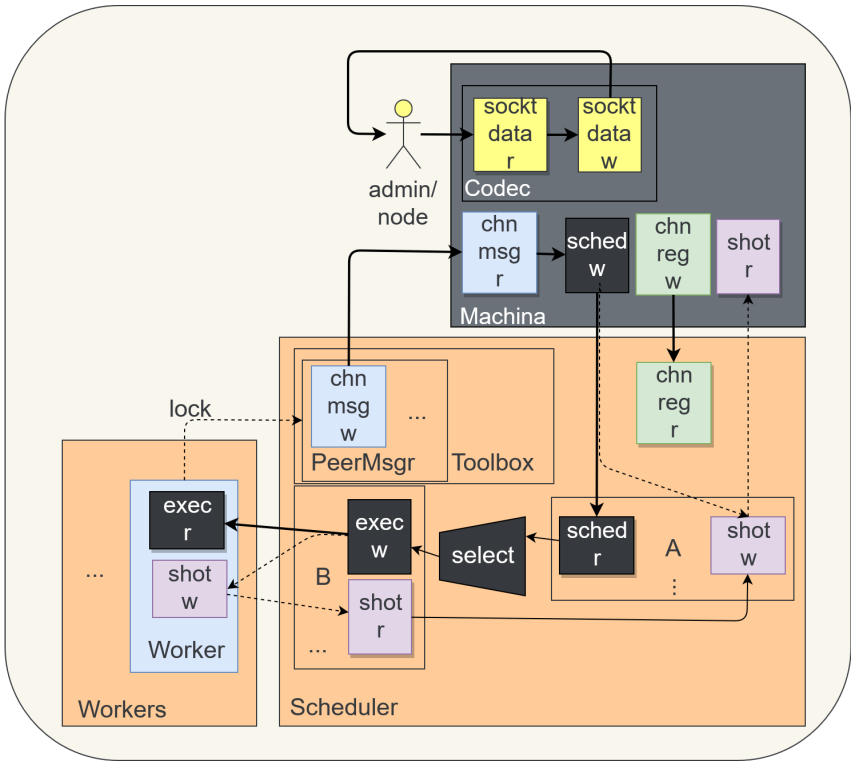
Fonte: Autoria própria.

também interagem com outros *actors* através de canais de comunicação. No entanto todos os *workers* tem acesso interno ao *toolbox*, que é armazenado no *scheduler* e é utilizado de maneira similar a uma variável global que fornece acessos e informações pertinentes sobre o nodo. O *toolbox* é composto por partes menores que, cada uma, tem seu próprio acesso mutualmente exclusivo, delimitado por um *mutex*, prevenindo *data-races* em um nível de acesso controlado. Com esta topologia de comunicação, distribuição de execução e acesso a dados, muitas execuções concorrentes serão possíveis entre *workers* diferentes, desde que não precisem de um acesso exclusivo a uma mesma variável.[3]

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.

[3] *RwLock* é uma delimitação alternativa ao *mutex* que permite múltiplos acessos concorrente exclusivamente de leitura.

Figura 3 – Nodo utilizando *Actor Model*.



Fonte: Autoria própria.

3.2.1.3 Topologia Completa

A Figura 3–Nodo utilizando *Actor Model*–representa as principais interações entre os *actors* do sistema. Pela natureza de isolamento dos *actors*, o compartilhamento de informação acontece principalmente por comunicação através de canais unidirecionais de mensagens.

3.2.1.3.1 *Admins* e *Peers*

Logo após a inicialização, não há nenhum *worker*, nenhum *admin* e nenhum *peer*, há apenas o *scheduler* e o *actor* do nodo. 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 mensagens enviadas por um *worker*, *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*. O terceiro canal tem uma função administrativa perante o *scheduler*, utilizado somente em momentos de cadastramento ou descadastramento de

actors[4].

Cada pedido realizado por um *admin/peer* implica em uma resposta, mesmo que ela não seja significativa para o *admin/peer*. Ao elaborar o pedido, o *actor* cria um canal *oneshot*[5], armazena a capacidade de receber o valor que será transmitido pelo canal (nomeado *shot r* e em lilás) e envia a capacidade de transmitir o valor junto ao próprio pedido (nomeado *shot w*, em lilás, e armazenado no *scheduler*, bloco *A*).

3.2.1.3.2 *Scheduler*

O *scheduler* é o *actor* de interface entre os *admin/peer* e os *workers*. Ele recebe vários pedidos de vários *admins/peers* 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 dos *shot r* (em lilás, no bloco *B*) e encaminha as respostas de volta para os *admins/peers* através do *shot w* (em lilás, no bloco *A*). De maneira similar ao *admin/peer*, o *scheduler* cria um canal *oneshot* para receber uma resposta de um *worker*, e encaminha esta nova capacidade de escrita para o *worker*[6].

No *scheduler*, o bloco *A* representa uma listagem de vários canais referentes a cada *admin/peer*: um *sched r* para cada *peer/admin* e um *shot w* para cada resposta esperada por cada *admin/peer*[7]. O bloco *B* representa uma listagem de vários canais referentes a cada *worker*: um *exec w* para cada *worker* e um *shot r* para cada pedido em execução, ou em fila de execução, por cada *worker*.[8]

Scheduler armazena uma alternativa às variáveis globais: o *toolbox*, variável que é compartilhada[9] entre *workers*, fornece um acesso indireto a estruturas de acesso exclusivo e

[4] É com o uso deste canal que um *admin* pode iniciar uma nova conexão com outro nodo, ao realizar um pedido a um *worker* de início de uma conexão TCP que, em sucesso, cria um novo *peer*. Como o *worker* deverá cadastrar este *peer* no *scheduler*, o *admin* envia junto ao próprio pedido uma cópia da capacidade de transmissão 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 *scheduler*. Dado esta dinâmica possibilitada pelos pedidos, a topologia do nodo é controladamente variável, pois existem possibilidades de interações temporárias/momentâneas.

[5] De uso único.

[6] Reaproveitando a memória alocada do pedido realizada pelo *admin/node*.

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

[8] Um *worker* pode internamente conter uma fila de pedidos, mas executa um 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–porém, neste caso, a implementação do *scheduler* precisaria ser generalizada em alguns pontos.

[9] Através de um *Arc* (*Atomic Reference Counter*), um acesso indireto que pode ser compartilhado entre *threads* diferentes. Esta caracterização da variável importa pois a desalocação de recursos no *Rust* é determinística, e o recurso referente ao *toolbox* é desalocado quando nenhum *worker* e nem o *scheduler* estiverem referenciando o

referentes ao nodo como um todo, como explanado em subseção 3.2.1.2–Topologia Simplificada. O *toolbox* conterá várias variáveis que poderão ser acessadas pelos *workers*, mas por enquanto apenas a *PeerMsgr*, ou *Peer Messenger*, foi criada, pois foi considerada indispensável para a topologia considerada. O *Peer Messenger* tem uma parte transmissora de vários canais, um para cada *admin/peer*.[10]

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 *Worker*

O *worker* é um *actor* que interage com o *scheduler* e *admins/peers* (estes últimos indiretamente). 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). Cada pedido é tratado isoladamente–o *worker* não tem alteração interna de estado como efeito de um pedido–, sendo desestruturado e então interpretado. Para a posterioridade, é esperado que novos tipos de *actors* existirão, e que poderão interagir com os canais do *toolbox*. Neste caso, os *workers* poderão precisar de uma implementação por máquina de estado para que possam interagir com novos *actors* de forma assíncrona.

3.2.2 Protocolo do Nodo

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.

---

toolbox. Como a contagem automática de referência (*Rc*, *Reference Counter*) precisa ser sincronizada entre *threads* diferentes, tal operação, neste caso, requer o custo adicional de ser atômica.

[10] *E.g.* quando um *admin* requisita a desconexão e remoção estável de um *peer*. Nesta caso, um *worker* recebe e executa este pedido, e retorna uma resposta ao *admin*. Antes de responder, o *worker* acessa o *toolbox* e adquire exclusividade temporária de acesso sobre o *Peer Messenger*–esperando de forma síncrona caso tal estrutura esteja em uso (*locked*). Ao ter acesso exclusivo sobre o *Peer Messenger*, o *worker* envia uma mensagem de comando de alta prioridade para o *peer* a ser removido–de que o *peer* deve se preparar para se remover–, e então desaloca a parte transmissora deste próprio canal com o *peer*, e também descadastra este *peer* do *Peer Messenger*. O *peer*, ao entrar em um estado neutro (ver o estado *standby* em subseção 3.2.2.2–*Peer Actor*) inicia alguns procedimentos internos a ele–como continuar esperando respostas, mesmo que insignificantes, já existentes a retornarem a ele–para que ele possa ser removido com segurança: ele próprio pedir seu descadastramento junto ao *scheduler* e então entrar em um estado terminal, com a desalocação e desligamento de todos os demais recursos, como canais de *socket*.

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 lê o *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 à cada uma das variáveis do *header* esta dentro do esperado[11] 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*[12] 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[13], e o do *toolbox*,

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

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

[13] 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*[14].

No estado de *standby*, após ser recebido qualquer notificação, são descartados todas as respostadas provenientes do *scheduler* e é verificado o conteúdo do *toolbox* 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 irá definir como deverá ser a resposta e o que deverá ser esperado. Para os procedimentos mais complexos é iniciado um estado que irá iniciar uma nova máquina de estados interna para tratá-los, facilitando assim o entendimento. Todos este procedimento é cíclico, terminando apenas quando esta 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[15], e o do *socket*.

Quando chega alguma informação no *socket* do *admin* gerenciado pelo *codec*, a máquina é notificada e começa o procedimento. O primeiro passo é comparar a *string* obtida com o argumento pré-definido construído em *struct-opt* 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.

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.

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

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

3.3 *Blockchain* completo

Este módulo consistirá em um programa de interface que fornecerá informações sobre a *blockchain* escrita em disco. A escrita se dará pela interface do nodo de roteamento da rede, onde será pedido e recebido, analisado e validado e então escrito em disco informações dos blocos e das transações da *blockchain*. Será necessário conhecer a estruturação dos blocos e seus *headers*, das transações e das informações nelas contidas, como *scripts* simples e lineares que serão executados por empilhamento, e das assinaturas digitais.

3.4 Carteira de Bitcoin

Este módulo consistirá em um programa seguro de interface que fornece informações, como histórico de transação e contabilização do saldo, e opções de ações, como transferência de *bitcoins* e a representação dos endereços públicos em código QR, relacionadas às chaves privadas fornecidas ou criadas no programa pelo usuário. Todas as informações ou ações referentes às chaves privadas serão encriptadas por segurança.

3.5 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. Pela ordem escolhida da implementação, alguns módulos foram estudados e prototipados–como as conexões de uma rede P2P–, e outros ainda não foram explorados–como a execução do *script* das transações do *Bitcoin*. Dado este fato, uma fluxo do funcionamento de um nodo completo não foi demonstrado.

REFERÊNCIAS

AGHA, G. A. **Actors**: A model of concurrent computation in distributed systems. 1. ed. [S.l.]: MIT Press, 1985. Citado na página 23.

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 na página 19.

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

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 3 vezes nas páginas 17, 18 e 19.

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 22.

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 22.

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 22.

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. [S.l.]: Mises Brasil, 2013. 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 21.

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 23.

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 na página 11.

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 na página 20.