Constantino: Uma Arquitetura BFT Escalável e Eficiente para Blockchains

Ray Neiheiser[1], **Joni Fraga**[1], **Luciana Rech**[2]

[1] Departamento de Automação e Sistemas – Universidade Federal de Santa Catarina (UFSC)

[2]Departamento de Informática e Estatística – Universidade Federal de Santa Catarina (UFSC)

neiheiser.r@posgrad.ufsc.br, joni.fraga@ufsc.br, luciana.rech@ufsc.br

Abstract. *Due to the recent rise of interest in distributed ledger technology and blockchain applications the field of byzantine fault tolerance received additional attention. Traditional algorithms as PBFT and novel algorithms as Proof of Work (PoW), Proof of Stake (PoS) and hybrid algorithms have been developed to cope with the openness of these systems. Nonetheless, these systems either do not scale very well with the number of servers, or rely on an underlying crpytocurrency. Hierarchical structures as Steward or Fireplug which better cope with the increasing number of replicas do not cope with the adversarial model expected in the blockchain environment. We, propose a hierarchical architecture to cope with the adversarial model of the blockchain environment while scaling horizontally with the number of replicas.*

Resumo. *Recentemente tem havido um interesse crescente na tecnologia de ledger distribuídos e aplicações de blockchain. Algoritmos tradicionais como PBFT e novos algoritmos como Proof of Work (PoW), Proof of Stake (PoS) e algoritmos híbridos foram desenvolvidos para lidar com a abertura desses sistemas. No entanto, estas Soluções não apresentam boa escalabilidade com o alto número de servidores ou dependem de uma criptomoeda subjacente. Estruturas hierárquicas como Steward ou Fireplug, apresentam melhor escalabilidade com um possível número de réplicas crescente, porém não lidam com o modelo competitivo esperado no ambiente blockchain. Neste artigo é proposta uma arquitetura hierárquica para lidar com o modelo competitivo do ambiente blockchain que também escala (horizontalmente) com o número de réplicas.*

1. Introdução

Com a ascensão das ofertas iniciais da ICO (*Initial Coin Offerings*) em 2017, a arquitetura de *blockchain* recebeu um expressivo aumento no interesse industrial. Isto também motivou o interesse acadêmico a resolver questões em uma ampla gama de aplicações nos campos da economia, sociologia, biologia, etc [Zhao et al. 2016, Angraal et al. 2017].

Nas *blockchains* os blocos são enfileirados em uma lista encadeada e cada bloco subseqüente contém o *hash* do bloco anterior como referência, isto torna toda a cadeia imutável já que alterar um bloco em algum lugar da cadeia exigiria a atualização de cada bloco subjacente, o que torna a operação inviável. Um dos fatores chave das *blockchains* é a distribuição para alcançar a descentralização (evitando um terceira parte confiável). Portanto, eles geralmente são chamados de *ledgers* distribuídos [Nakamoto 2008]. As

blockchains são divididas em sistemas abertos, onde qualquer servidor pode entrar e sair livremente, e *blockchains* permissionadas, que são *blockchains* de consórcios onde a entrada é restrita pelo consórcio ou *blockchains* privados onde o *blockchain* inteiro é controlado por uma única entidade [Vukolić 2016]. A arquitetura *blockchain* garante que o sistema continue funcionando corretamente mesmo se uma determinada porcentagem dos servidores na rede estiver se comportando de forma arbitrária ou maliciosa.

No entanto, a maioria dos algoritmos propostos não lidam bem com a dimensão encontrada nestes *ledgers* distribuídos, onde mais de mil nós podem participar dos consensos. Por essa razão, o algoritmo *Proof of Work* (PoW) foi proposto para resolver estes consensos, visando diminuir a complexidade de mensagens. No entanto, PoW resulta em um alto custo computacional e um excessivo consumo de energia [Vukolić 2016]. Devido a isso, o *Proof of Stake* (PoS) foi proposto onde os nós de consenso devem colocar uma certa quantidade de moeda em jogo para participar do consenso que diminui o consumo de energia, mas também depende de uma criptomoeda subjacente.

Desde então, várias soluções híbridas foram propostas para melhorar o *throughput* e a latência desses algoritmos. No entanto, todos eles dependem do incentivo da criptomoeda subjacente que pode não existir em *blockchains* permissionados, como *Hyperledger* [Cachin 2016]. Os *Blockchains* permissionados geralmente executam uma versão adaptada do PBFT [Sousa et al. 2017], apresentando bom desempenho em relação a taxa de transferência e a latência, porém o algoritmo PBFT introduzido em [Castro and Liskov 1999] não escala muito bem em grandes sistemas pelo custo assintotico em termos de mensagens ($\mathcal{O}(n^2)$). Vários algoritmos já foram propostos para otimizar o citado problema de consenso, criando estruturas hierárquicas [Amir et al. 2010, Neiheiser et al. 2018].

Neste artigo é apresentada uma arquitetura hierárquica e algoritmos que tratam o modelo competitivo exigido de *ledgers* distribuídos e reduzem a complexidade da mensagem do consenso de $\mathcal{O}(n^2)$ para $\mathcal{O}(n)$. Com mudanças de visão estes custos caem de $\mathcal{O}(n^3)$ para $\mathcal{O}(n^2)$. As principais contribuições deste artigo são:

- Um modelo hierárquica que serve para reduzir os custos dos algoritmos;
- Uma adaptação do modelo hierárquico para o ambiente *blockchain*;
- Mudanças de visão com reduzida complexidade de mensagens.

O restante do artigo está organizado como se segue. Na seção 2 citamos os trabalhos relacionados. Depois, é apresentado o modelo do sistema no item 3. Na seção 4 são discritos aspectos da arquitectura e os algoritmos desenvolvidos. No item 5 é apresentado uma análise da proposta e considerações. A seção 6 apresenta as conclusões finais.

2. Trabalhos Relacionados

Diferentes frentes de trabalhos relacionados com aplicações *blockchain* tem sido desenvolvidas nos últimos anos. Nesta seção exploramos alguns destes trabalhos começando com o consenso bizantino e na sequência o *PoW*, *PoS* e trabalhos híbridos.

Depois do *PBFT* [Castro and Liskov 1999] vieram várias otimizações deste algoritmo. Uma destas soluções é apresentada por Amir em [Amir et al. 2010], onde os servidores são agrupados em grupos (*clusteres*) hierárquicos. Nesta situação, os membros de grupo local devem assinar cada ação encaminhada por seus líderes ao grupo global usando

criptografia de limiar. Esta abordagem melhora o desempenho significativamente. Porém, as assinaturas em cada passo no sistema tem um custo computacional muito alto, especialmente com o aumento do tamanho dos grupos locais. O *Fireplug* [Neiheiser et al. 2018] reduz este custo computacional obrigando os líderes a provar que estão corretos para os membros dos seus grupo locais, porém não trata com o modelo competitivo esperado no *blockchain*.

Honeybadger [Miller et al. 2016] trabalha com Consenso Bizantino Assíncrono sem depender de questões temporais. Isso é feito reduzindo o problema para *Asynchronous Common Subset Agrement* e *Binary Byzantine Agreement* combinados com o uso de mensagens em lote (*batch*) para diminuir a complexidade das mesmas de $\mathcal{O}(n^3)$ para $\mathcal{O}(n^2)$. Ainda não tendo restrições temporais, o que é muito útil neste ambiente altamente competitivo dos *blockchains*, a ocupação da largura de banda é relativamente alta especialmente devido a criação de lotes de mensagens. Outra solução chamada Gosig foi proposta em [Li et al. 2018] onde a eleição do líder é feita através da execução de funções verificáveis aleatórias que diminuem os riscos de ataques aos líderes eleitos, uma vez que são conhecidos apenas no momento do consenso. Além disso, eles otimizam a comunicação, confiando na propagação de mensagens no modo *gossip* entre as réplicas e aplicando multi-assinaturas (vetor de assinaturas). Enquanto todas estas propostas apresentam melhorias significativas em comparação aos modelos tradicionais, os custos de comunicação das mesmas são ainda muito altos, dificultando a escalabilidade destas propostas para um número elevado de réplicas.

Devido a dificuldade de escalabilidade, muitos *blockchains* dependem de abordagens baseadas na Teoria dos Jogos, onde os participantes honestos na rede são recompensados. No *Proof of Work* (PoW), proposto pela primeira vez por Satoshi Nakamoto [Nakamoto 2008], os participantes do consenso (chamados mineiros) tem que resolver um enigma computacional que transforma um voto por processo para um voto por CPU. O primeiro a resolver o enigma é então recompensado com o incentivo (valor em cryptomoeda). No *Proof of Stake* (PoS) o mineiro tem que colocar uma parte em cryptomoeda em jogo para participar do consenso, se não seguir o protocolo o mineiro pode perder parte de sua aposta (*stake*) [King and Nadal 2012].

Outros modelos existem, como o *Delegated Proof of Stake* (DPoS) onde as partes interessadas elegem representantes em um grupo limitado de nós (testemunhas) para executar o consenso do sistema. Isso reduz o número de réplicas que participam do consenso e melhora significativamente o desempenho [Schuh and Larimer 2017]. Embora, os três modelos citados acima (*PoW*, *PoS* e *DPoS)* sejam bem dimensionados quanto ao número de réplicas, estes exigem o custoso algoritmo de *PoW* como base, o que reduz significativamente o desempenho geral.

Além disso existem modelos híbridos que usam uma versão otimizada do PBFT e integram a produção de blocos, melhorando desta forma o sistema. Um exemplo é o *Tendermint* [Kwon 2014] que apresenta um forte controle para ambientes competivos e maliciosos. O Ouroboros [Kiayias et al. 2017] é um protocolo popular, sendo um híbrido do *PoS* que apresenta uma grande vantagem em termos de eficiência sobre a maioria dos algoritmos de *PoS*. Também interessante é o *Casper the Friendly Finality Gadget* [Buterin and Griffith 2017], que cria uma rede de *PoS* sobre um *blockchain* de *PoW* fornecendo a abstração *finality* que protege o *blockchain* contra reversão de blocos passa-

Tabela 1. Tabela de Comparação

	Honeybadger	Gosig	Casper	HotStuff	OurosBoros	Tendermint	Steward	Constantine
Latência	Baixa	Baixa	Média	Baixa	Baixa	Baixa	Baixa	Muito baixa
Throughput	Alto	Alto	Médio	Muito alto	Alto	Alto	Alto	Muito alto
Criptomoeda	Não	Não	Sim	Sim	Sim	Não	Não	Não
Bandwidth	$\mathcal{O}(n^3)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$

dos através de *forks*. A abordagem *Hot Stuff* [Abraham et al. 2018] simplifica o algoritmo *PBFT*, baseando-se em um agente central que serve como um intermediário na comunicação entre as réplicas. Além disso, com o uso da criptografia de limiar, a complexidade em mensagens é reduzida para $\mathcal{O}(n)$. O líder centralizado pode ser eleito regularmente usando *PoW* ou *PoS* para garantir a segurança do sistema. No entanto, todas essas soluções dependem de uma criptomoeda subjacente para incentivar o comportamento correto e/ou requer um consenso custoso em termos de mensagens.

Diferente da maioria das abordagem de *blockchain* citadas acima, o protocolo proposto neste artigo, segue um modelo hierárquico e não requer uma criptomoeda subjacente, uma vez que é assumido um modelo de sistema permissionado. Além disso, diferentemente de Steward [Amir et al. 2010], não há a necessidade dos grupos locais assinarem todas as ações de seus líderes nos passos do protocolo.

3. Modelo do Sistema

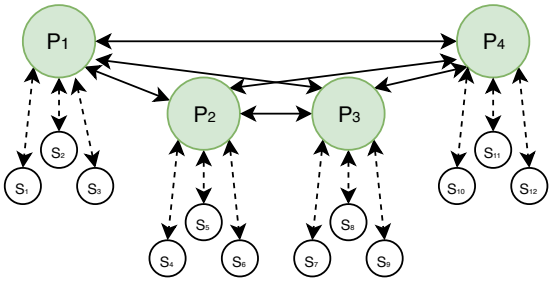

Figura 1. Sistema hierarquico

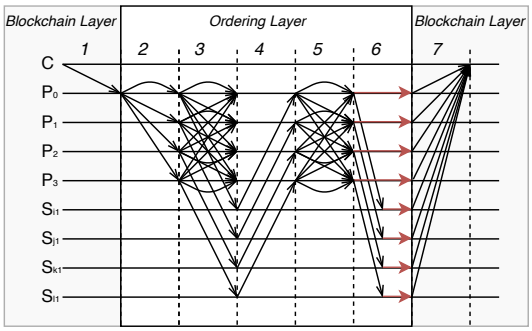

Figura 2. Protocolo hierarquico

O sistema é composto de processos clientes e de um conjunto de n processos servidores (n réplicas). Os clientes são capazes de se conectar a qualquer servidor através da rede. É assumido uma distribuição geográfica dos n servidores em que o cliente geralmente se conecta a mais próxima réplica. O modelo de sistema é assíncrono relaxado com períodos de sincronismo (modelo semi-síncrono) o que garante a terminação das aplicações (evitando a barreira da condição FLP [Fischer et al. 1982])

Servidores e clientes possuem certificado de suas chaves públicas emitidos por uma mesma entidade de certificação confiável. Esta situação garante que clientes e servidores possam se autenticar mutuamente quando do estabelecimento de canais de comunicação seguros. As mensagens no sistema são assinadas. Uma vez estabelecidos estes canais seguros, as mensagens que devem trafegar nos mesmos são cifradas com criptografia simétrica.

Neste trabalho, foi considerado um ambiente de *blockchain* dinâmico onde os

servidores podem entrar e sair. Nesta proposta consideramos *permissioned blockchains*, o que significa que servidores têm que enviar requisições para participar ou sair do sistema.

O ambiente destas aplicações de *blockchains* deve conviver com a malícia dos participantes. Neste sentido, temos que assumir hipóteses de faltas bizantinas. O sistema deve funcionar desde que não seja ultrapassado o limite f para processos maliciosos. Portanto, o número de servidores no sistema deve ser dado por n onde $n \geq 3f+1$. Processos maliciosos ou bizantinos possuem um comportamento arbitrário, ou seja, podem atrasar ou corromper informações e atentar contra a autenticidade de sujeitos e informações. Nos canais de comunicação que envolvem a Internet, podem haver perdas e assumimos um comportamento *fair lossy* [Reynal 2005] [1]. As comunicações dentro de uma organização que participa de um consórcio em *blockchain* permissionado, tratamos como canais confiáveis [Reynal 2005]. Ou seja, por estas últimas se darem provavelmente dentro de *clusters* locais, consideramos os canais como sem perdas.

3.1. Organização do Sistema

O Sistema depende de uma estrutura hierárquica exibida na Figura 1. Os processos servidores são subdivididos em processos líderes e seguidores. Com isto, a estrutura hierárquica de servidores define k réplicas líderes formando o grupo de mais alto nível ou global da hierarquia (grupo de líderes). Cada um destes líderes representa no nível da liderança o seu grupo local. Os servidores de cada grupo local são considerados geograficamente próximos uns dos outros, o que resulta em uma latência mais baixa dentro destes grupos. São k processos servidores líderes e $n-k$ seguidores. O tamanho k do grupo global é restrito pela relação $k^2 \leq n$ garantindo com isto o custo linear dos consensos.

Na figura 1, são apresentados quatro processos líderes ($k=4$), formando o grupo global. Os doze processos seguidores $(S_1, S_2, ..., S_{12})$ presentes na figura vão compor com os seus líderes quatro grupos locais $((P_1, S_1, S_2, S_3)$, (P_2, S_4, S_5, S_6), (P_3, S_7, S_8, S_9) e $(P_4, S_{10}, S_{11}, S_{12}))$. Os processos seguidores recebem e enviam mensagens somente para seus líderes.

O sistema é feito para uma composição variável considerando a natureza de *blockchains*. Para que funcione diante de um ambiente malicioso, é necessário que os parâmetros da proposta sejam definidos para uma composição mínima. Esta composição mínima de n servidores é dada pela relação $n = 3f + 1$. É necessário também que esta composição mínima de servidores esteja distribuída de forma igualitária entre os servidores líderes, formando os grupos locais. Sem estas condições citadas, a composição mínima não funcionária. Outro aspecto a se considerar é que o sistema deve fornecer uma estrutura que o permita evoluir em uma situação de larga escala. O número de servidores que vão ser agregados durante seu ciclo de vida deve encontrar facilidades para tal. Para isto temos que trabalhar com um número máximo de k líderes. Como k é limitado por $k^2 \leq n$ então é assumido a situação de k máximo onde $k^2 = n$. Por outro lado, se a composição mínima deve funcionar então, temos que assumir $k^2 = 3f + 1$. Com isto teremos f dado pela expressão $f = \lceil \frac{(k^2-1)}{3} \rceil$. Se considerarmos o exemplo da figura 1 onde $k = 4$ ($n = 16$), o limite de faltas do sistema será dado por $f = 5$.

[1] Um canal ligando processos p_i a p_j possui características *fair lossy* quando não cria ou duplica mensagens, mas pode perder as mesmas. Porém, se p_i envia um número infinito de mensagens para p_j, então este recebe um número infinito de mensagens de p_i

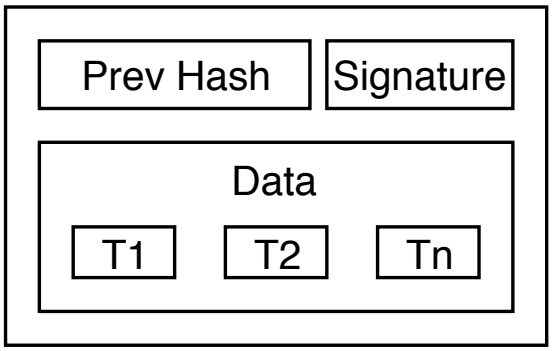

Figura 3. Modelo de Bloco

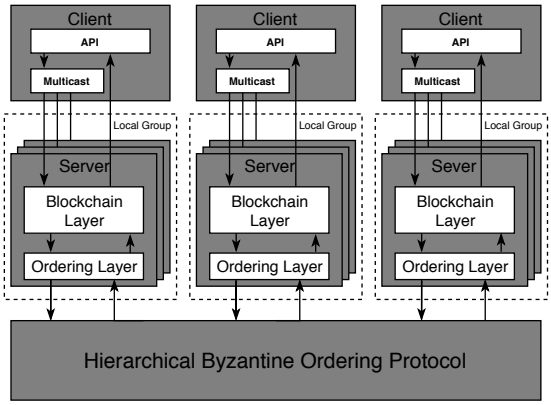

Figura 4. Arquitectura do Sistema

Note que o valor de f é sempre maior que o valor de k. A composição mínima e outras maiores devem continuar funcionando mesmo com o comprometimento de uma minoria de servidores líderes. É necessário sempre que $k/2+1$ líderes sejam corretos para o sistema funcionar de forma adequada. No exemplo da figura 1 precisamos de $k \geq 3$ líderes corretos (mesmo com $f = 5$). Se os líderes selecionados em uma visão do sistema forem faltosos em sua totalidade ou maioria, nenhum bloco de transações será ordenado pela camada *Ordering* e o protocolo de ordenação em sua visão vai sendo reconfigurado até atingir uma visão onde os líderes corretos formam a maioria do grupo de líderes e um bloco possa ser decidido (a visão deve ter $k/2 + 1$ líderes corretos).

3.2. Extratificação nos Servidores

O sistema é composto por dois subsistemas, a "camada *Blockchain*"e a "camada *Ordering*", como mostrado na Figura 4. O sistema é construído de uma maneira modular que permite integrá-lo facilmente como um protocolo de consenso em qualquer *blockchain* como, por exemplo: *Hyperledger Fabric* [Cachin 2016]. O cliente envia por meio de uma API suas mensagens para o grupo local mais próximo. A camada *Blockchain* recebe as solicitações do cliente e as agrupa em blocos que serão enviados à camada *Ordering*. Esta, por sua vez, é responsável pela ordenação dos blocos de entrada e os retorna para a camada *Blockchain* em ordem total. A camada *Blockchain* verifica o bloco e, se for válido, o anexa de forma ordenada ao seu *blockchain*.

O ambiente de *blockchain* assumido tem os blocos produzidos em uma determinada periodicidade e com tamanho fixo similar à maioria dos *blockchains* como o *Bitcoin* [Nakamoto 2008]. Em nossa proposta, como nos *blockchains* existentes, os blocos podem ser entregues vazios para garantir a periodicidade dos mesmos [2]. A figura 3 mostra um exemplo de modelo de bloco. Cada bloco consiste em um campo de dados que contém as transações de clientes, incluindo seus certificados e assinaturas, um campo de assinatura que contém a assinatura do criador do bloco e o *hash* do bloco anterior no sistema.

4. Dinâmica dos *Blockchains*

O grupo global (o grupo de líderes) possui um coordenador que numa instância de consenso impõe o seu bloco como o próximo a ser ordenado. A cada execução de uma

[2]No *Bitcoin*, de 2009 a 2015, a produção de blocos vazios ficava em torno de 20% [Redman 2018]. Já no *Ethereum*, atualmente, entre 2 a 3% dos blocos produzidos são vazios [Svanevik 2018].

instância deste consenso, um bloco tem sua ordem definida. Este consenso que define a ordem dos blocos no sistema é executado por um protocolo que possui passos semelhantes ao PBFT. O protocolo proposto neste texto define a troca de coordenador a cada instância concluída de consenso (a cada bloco ordenado). Esta função de coordenador, portanto, circula entre os líderes do grupo global. Esta alternância de coordenadores aumenta o desempenho do protocolo, além de diminuir o impacto de um coordenador comprometido.

Um cliente envia sua transação para um grupo local de servidores mais próximo ou conhecido (*multicast* na Figura 4). O servidor líder de grupo local ao receber uma transação de cliente é responsável por incluir a mesma em um bloco corrente e depois quando for coordenador, deve propor este bloco para a camada *Ordering*.

Na camada *Blockchain* são previstos dois intervalos de tempo: o $time_{toS}$ e o $time_{toR}$. Como citado na seção anterior, o modelo de *blockchain* é definido por uma periodicidade na proposição de blocos. Com isto, foi definido para uma instância de consenso que se inicia, um atraso $time_{toS}$ que deve ser respeitado pelo coordenador para envio da proposição de um bloco. Este tempo $time_{toS}$ tem seu início a partir da última inserção na cadeia de blocos e só quando se completa, o coordenador propõe o seu bloco para ordenação. O outro tempo ($time_{toR}$) é um *timeout* que estipula o prazo máximo que um servidor na camada *Blockchain* deve esperar por um bloco ordenado pela camada *Ordering*. Na falta de um bloco no fim deste prazo, existirá uma suspeita sobre a camada abaixo e uma solicitação de *view change* é acionada na *Blockchain Layer* (ver seção 4.2). É assumido no presente projeto este *timeout* como sendo $time_{toR} \geq 2 * time_{toS}$.

4.1. Descrição dos Algoritmos

4.1.1. Ações na Camada *Blockchain*

As ações executadas por um servidor na camada *Blockchain* começam com um cliente C_j enviando um grupo local de servidores uma mensagem de requisição a que deve conter a transação T desejada pelo cliente, uma estampilha (t) com o tempo de envio, o identificador do cliente e a assinatura desta mensagem (α_j) usada em verificações de autenticidade.

A recepção desta mensagem com a transação do cliente C_j por um servidor S_i tem como objetivo incluir a mesma em um bloco a ser destinado à *blockchain*. Se a transação já foi recebida, ou está incluída na cadeia de blocos (bloco já foi ordenado) ou se faz presente em bloco pendente (bloco já completo, mas não ordenado) ou ainda em bloco corrente (bloco não completo em seu tamanho máximo). Se o bloco com a transação já foi ordenado, fica evidente que o cliente não recebeu a resposta e a mesma é então novamente enviada ao C_j. Se o servidor S_i recebe uma mensagem com transação enviada pela primeira vez e for um líder de grupo local então esta transação e suas informações são adicionadas em seu bloco corrente (B_c).

Quando ocorre o retorno da camada *Ordering* no servidor S_i, uma decisão foi concluída para a instância m de consenso. Se a decisão ocorreu de maneira adequada o bloco resultante B_d, proposto pelo coordenador, é incluído na blockchain BC_i na posição do consenso m.

Se o resultado é o valor *default* ($\perp$), isto significa que a instância m terminou, mas não houve consenso por um bloco e deve ser executada a reconfiguração (*viewChange*) do grupo local da réplica na camada abaixo. Este valor *default* significa que o consenso

da camada de baixo terminou com dois ou mais blocos diferentes e, por consequência, o coordenador é faltoso.

Os servidores também mantêm um *timeout* para a decisão de uma instância de consenso m. O valor deste *timeout* é dado por $time_{toR}$ (seção anterior). No esgotamento deste *timeout* uma reconfiguração dos grupos locais (*view change*) do sistema é necessária.

Como consequência do bom andamento de um consenso m, um servidor correto S_i deve responder ao cliente, enviando uma mensagem de resposta (*reply*) com o resultado R da transação T executada no *blockchain*. Este *reply* contém informações que visam a autenticidade como o carimbo de tempo t da requisição do cliente, o identificador do bloco na *blockchain* (BC_i) e o identificador e a assinatura do servidor S_i.

O cliente deve aguardar $f + 1$ respostas válidas e coincidentes em termos de resultado antes de aceitar o mesmo. Se não recebe $f + 1$ *replies* coincidentes em prazo definido previamente (*timeout* do cliente), então o cliente, usando o *multicast* de sua *API*, deve enviar a mesma requisição de transação para outro grupo local do sistema. Como este reenvio mantém o carimbo de tempo original, nenhum servidor correto aceita a mesma transação mais de uma vez e muito menos anexa a mesma ao seu *blockchain*.

Se S_i for o coordenador do próximo consenso que deve ordenar um bloco, este servidor então prepara uma temporização para suas ações no consenso $m + 1$. Com isto, é estabelecido o atraso de $time_toS$ para a sua proposição de bloco. No esgotamento deste prazo $time_toS$, o coordenador deve propor um bloco para ordenação. Um servidor líder sempre saberá quando é próximo coordenador do grupo global (ordenação dos líderes baseadas na ordem crescente de seus identificadores). Este coordenador começa a sua inspeção a procura de um bloco pela lista de blocos pendentes. Se houver blocos pendentes, então o bloco a ser proposto será o primeiro desta lista. Se não houver pendentes então o bloco corrente é proposto (B_c) mesmo que vazio ou não completo em seu tamanho fixo. Com isto, o servidor deve calcular um resumo (aplicando uma função *Hash*) no último bloco ordenado (pelo consenso m) na sua cadeia de blocos. Este resumo calculado é incluído no campo $B.hash$ do bloco a ser proposto no consenso $m + 1$.

4.1.2. Ações na Camada *Ordering*

O algoritmo 1 descreve o protocolo de consenso usado para definir a ordem dos blocos que são registrados na *blockchain* do sistema. Este algoritmo hierárquico da camada *Ordering* possui três etapas: *Pre-Prepare*, *Prepare* e *Commit*, que estão representadas por cinco passos na Figura 2.

Como a cada instância de consenso assume um novo coordenador para propor seu bloco, cada líder espera até que chegue sua vez como coordenador em uma visão global (vG_i). Neste algoritmo, a réplica S_i é a coordenadora e recebeu um bloco B da camada *Blockchain* (linha 4). Com isto, S_i propõe o bloco recebido através de uma mensagem *pre-prepare* para todas as réplicas na liderança (linhas 6 a 7). Esta mensagem é representada pela tupla $<$ PRE-PREPARE, $B, vG_i, i >_{\alpha_i}$ com B como o bloco proposto e onde vG_i é a visão da liderança que contém os $id's$ dos líderes atuais do sistema (grupo global). O identificador i do coordenador e a sua assinatura α_i também são representados na tupla.

Na linha 9, o servidor S_i recebe uma mensagem válida de *pre-prepare*. Mensagens inválidas são descartadas. As mensagens são consideradas válidas se as verificações de assinaturas estão corretas e se ainda não foram recebidas nesta *view*. Estas verificações ocorrem em todos os passos e são omitidas no código para simplificar o mesmo. Em seguida, é verificado se o emissor é o coordenador da visão global corrente e também se a visão enviada (vG_i) coincide com a visão atual do receptor (linha 10). Após estas verificações se confirmarem, o líder S_i salva o bloco proposto em seguida e constrói uma mensagem *prepare* (linha 12). A tupla $<$ PREPARE, $B, vG_i, i, \alpha_c >_{\alpha_i}$ representa esta mensagem e possui entre seus campos, o bloco proposto B, a assinatura α_c do coordenador sobre a mensagem *pre-prepare* e acompanham ainda, o identificador i e a assinatura α_i do emissor (o líder S_i) .

A mensagem *prepare* é enviada aos servidores do grupo de líderes (linha 13) e ao seu grupo local através de canais confiáveis. Este mensagem permite que líderes que tenham perdido o *pre-prepare* possam se recompor no protocolo. Um líder S_i, ao receber um *prepare* válido, verifica se já recebeu mensagem de *pre-prepare* no consenso atual (linha 23). Se não recebeu, coloca o mesmo bloco em $buf_m(B)$.

Algorithm 1 Camada *Ordering*

```
1:  vG_i ← ∅                                          ▷ The global view
2:  vL_i ← ∅                                          ▷ The local view
3:  buf_m ← ∅                          ▷ Buffer for consensus instance m
4:  on Ordering.propose(B)                            ▷ At the coordinator
5:      buf_m(B) ← buf_m(B) ∪ {B}
6:      msg_i ←< PRE-PREPARE, B, vG_i, i >_{α_i}
7:      send msg_i to leadership
8:  end
9:  on receive(pre-prepare_j)
10:     if vG_i = vG_c ∧ vG_i.coord = c then
11:         buf_m(B) ← buf_m(B) ∪ {B}
12:         msg_i ←< PREPARE, B, vG_i, i, α_c >_{α_i}
13:         send msg_i to leadership ∧ group[i]
14:     end if
15: end
16: on receive(prepare_j)
17:     buf_m(P) ← buf_m(P) ∪ {prepare_j}
18:     if S_i is not leader then
19:         buf_m(B) ← buf_m(B) ∪ B
20:         msg_i ←< LCOM, α_c, vG_i, i, αB_i >_{αC_i}
21:         send msg_i to leader
22:     else
23:         if buf_m(B) is empty then
24:             buf_m(B) ← buf_m(B) ∪ {B}
25:         end if
26:         if (|buf_m(P)| ≥ ⌊k/2⌋ + 1) ∧ timeout then
27:             α_cm ← combineSig(buf_m(LCOM).αC_j)
28:             α_bk ← combineSig(buf_m(LCOM).αB_j)
29:             msg_i ←< COMMIT, α_c, vG_i, i, α_cm, α_bk >
30:             send msg_i to leadership
31:         end if
32:     end if
33: end
34: on receive(lcommit_j)
35:     buf_m(LCOM) ← buf_m(LCOM) ∪ {lcommit_j}
36: end
37: on receive(commit_j)
38:     if verify(α_bk) ∧ verify(α_cm) then
39:         buf_m(C) ← commit_j
40:         if |buf_m(C)| ≥ ⌊k/2⌋ + 1 then
41:             if |buf_m(B)| ≠ 1 then
42:                 decision_m.B ←⊥
43:             else
44:                 decision_m.B ← buf_m(B)
45:             end if
46:             decision_m.sigs ← buf_m(C).α_bk
47:             send decision_m to group[i]
48:             return decision_m.B
49:         end if
50:     end if
51: end
52: on receive(decision_m)                  ▷ Reception of decision_m
53:     if verify(decision_m.sigs) then
54:         return decision_m.B
55:     end if
56: end
```

Um seguidor S_i ao receber um *prepare* válido vindo de seu líder, coloca o mesmo em seu *buffer* (linha 17) e salva o bloco vindo do coordenador (linha 19). Na sequência, este seguidor monta uma mensagem de *commit* local (a tupla $<$ LCOM, $\alpha_c, vG_i, i, \alpha B_i >_{\alpha C_i}$) onde α_c é a assinatura do coordenador sobre campos do *pre-prepare*, i e vG_i são o identificador e a visão do seguidor S_i e αB_i é a assinatura deste seguidor sobre o bloco (B). Além disso, inclui uma assinatura αC_i, feita sobre campos desta mensagem ($LCOM$). Esta mensagem é por fim enviada por S_i ao líder de

seu grupo local (linha 21). Os líderes, ao receberem mensagens válidas de *commit* local, colocam as mesmas no $buf_m(LCOM)$ (linhas 34-35).

Para o *commit* de grupo, o líder S_i cria nas linhas 27-28 do algoritmo as multi-assinaturas α_{cm} (sobre campos do *commit*) e α_{bk} (sobre o bloco), usando suas assinaturas e as de seus seguidores (αC_i's e αB_i's respectivamente)[3]. Na sequência, a mensagem de *commit* que deve representar o grupo no passo 5 é montada, conforme a tupla $<$ COMMIT, $\alpha_c, vG_i, i, \alpha_{cm}, \alpha_{bk} >$, onde α_c representa a assinatura do coordenador sobre o *pre-prepare*, vG_i é a visão atual, i é o id da réplica líder e α_{cm} e α_{bk} são as multi-assinaturas citadas acima. Esta mensagem é enviada para o grupo de líderes (linha 30). As multi-assinaturas levam no *commit* o reconhecimento da validade dos passos anteriores do grupo local.

As réplica líderes S_i ao receberem mensagens de *commit*, depois da verificação das assinaturas, salvam em *buffers* ($buf_m(C)$) estas mensagens (linhas 37-39). Para que se chegue a decisão de consenso é necessário a recepção de um número maior ou igual a $\lfloor \frac{k}{2} \rfloor + 1$ mensagens de *commit* ($|buf_m(C)| \geq \lfloor \frac{k}{2} \rfloor + 1$, linha 40), contendo então $\lfloor \frac{n}{2} \rfloor + 1$ ou mais assinaturas de réplicas o que garante que uma maioria das réplicas corretas participaram da decisão. Atingido este valor de mensagens *commit*, a decisão começa com a verificação se ocorreram mais de um bloco proposto em $buf_m(B)$ ($|buf_m(B)| \neq 1$). Se foram propostos mais de um bloco neste consenso então o coordenador é malicioso e o valor default ($\perp$) é a decisão do consenso. Se o *buffer* possuir só um bloco, este será a decisão correta do consenso. Ambos os resultados de consenso são atribuidos à estrutura $decision$ no seu campo B (linhas 42-44).

Na linha 46, usando cada mensagem de *commit* recebida em $buf_m(C)$, as réplicas vão formar uma lista de multi-assinaturas sobre o bloco proposto (α_{bk}'s). Esta lista vai ser anexada ao campo sig da estrutura $decision_m$. Esta estrutura, é então enviada por cada líder ao seu respectivo grupo local (linha 47). Na sequência o líder S_i retorna $decision_m.B$ à camada *Blockchain* na linha 48, concluindo o consenso.

Na linha 52, os seguidores recebem a mensagem com a estrutura $decision_m$, verificam a lista de multi-assinaturas (linha 53) e, se válidas, retorna $decision_m.B$ à camada *Blockchain* (linha 54 do algoritmo).

4.2. *View Changes*

Em nosso sistema, existem dois níveis de visões: A visão global que contém o *id* da visão, o do coordenador e os *id*'s de todos os líderes (acompanhados por suas multi-assinaturas). Uma visão local deve conter o *id* da visão local, os *id*'s dos seguidores, o líder atual e uma multi-assinatura sobre esta visão.

Se um servidor não conclui uma instância de consenso de forma adequada, uma reconfiguração será acionada, provocando uma troca de visão local na camada *Ordering*. O algoritmo 2 descreve estas trocas de visão local. Neste algoritmo, a réplica S_i incrementa o *id* da visão local, troca o líder de forma determinista e então envia uma mensagem *elect* para o líder da nova visão. Esta mensagem contém a *view* local e uma assinatura local α_i sobre esta visão (linhas 3-5). Depois, aguardará a mensagem de confirmação do líder recém eleito contendo a nova visão assinada pelo grupo. Se S_i receber esta

[3] α_{cm} e α_{bk} são montadas a partir dos seguidores que respondem no prazo *timeout* (linha 26).

confirmação dentro de um limite de tempo e com a multi-assinatura válida, a nova visão é aceita (linhas 7-9). Se a réplica não recebeu uma confirmação válida, acionará uma nova reconfiguração que incrementará novamente o *id* da visão e trocará de líder mais uma vez, recomeçando este protocolo novamente (linha 13). Neste passo é importante que o líder antigo não participe na nova eleição pois é considerado suspeito.

O Algoritmo 3 mostra o código que uma réplica executa ao receber as mensagens *elect*. Ao receber esta mensagem, a réplica S_i adiciona a mesma ao seu *buffer* (linha 5). Recebendo uma maioria de mensagens *elect* das réplicas locais ($l/2 + 1$), S_i combina as assinaturas locais α_j formando uma multi-assinatura da nova visão de grupo local (linhas 6 e 7). Este líder também verifica a assinatura de grupo resultante (linha 8). Se correta, o novo líder atualiza sua visão local e envia mensagens de confirmação com a visão e a assinatura de grupo para todas as réplicas do grupo local e também para a liderança. Os membros do grupo local, conforme explicado anteriormente, verificam a assinatura e se correto, aceitam a nova visão. As réplicas do grupo de líderes também verificam a assinatura e se correto, incluem o líder recém eleito formando uma nova liderança (vG).

Algorithm 2 View Changes [1]

```
1:  vL_i ← nil                                          ▷ The Local View
2:  on reconfigurationRequest()
3:      vL_i.id++                                       ▷ Increment view id
4:      vL_i.rotateLeader()
5:      send < ELECT, vL_i, α_i > to vL_i.leader
6:  end
7:  on receive(confirm_j)
8:      if verify(confirm_j.msig) then
9:          vL_i ← confirm_j.vL                         ▷ Update View
10:     end if
11: end
12: on confirmationTimeout()
13:     reconfigurationRequest()
14: end
```

Algorithm 3 View Changes [2]

```
1:  vL_i ← nil                                          ▷ The Local View
2:  buf ← ∅                                             ▷ The message buffer
3:  c ← nil                                             ▷ Last Consensus
4:  on receive(elect_j)
5:      buf ← buf ∪ {elect_j}
6:      if buf.size ≥ l/2 + 1 then                      ▷ If received majority
7:          msig ← combine(buf)
8:          if verify(msig) then
9:              vL_i ← elect_j.vL
10:             send < CONFIRM, msig, vL_i > to group
11:             send < NOTIFY, msig, vL_i > to leaders
12:         end if
13:     end if
14: end
```

4.3. Análise de Complexidade em Mensagens

Nesta subseção, calculamos a complexidade em mensagens da nossa proposta para mostrar a escalabilidade da mesma. Como mencionado anteriormente, assumimos que o tamanho do grupo global é restrito a k que é limitado pela relação $k^2 \leq n$. Esta é uma restrição básica para se conseguir um custo linear da abordagem proposta, evitando o custo quadrático de protocolos semelhantes ao PBFT.

Na primeira etapa do protocolo, o cliente transmite suas transações para um grupo local. Assumindo l servidores no grupo, isto resulta em uma complexidade de mensagem de $\mathcal{O}(l)$ onde $l < n$. O mesmo ocorre no passo 7 da figura 2, em que o cliente é notificado com as mensagens de *reply*. No passo 2 há uma complexidade de mensagem de $\mathcal{O}(k)$ em que k descreve o número de líderes na liderança, já que o coordenador encaminha uma mensagen de *pre-prepare* com a proposta de bloco somente aos líderes.

No passo 3, os líderes enviam uma mensagen de *prepare* na liderança e também para seus seguidores. Isso resulta em k^2 mensagens na liderança e l mensagens enviadas aos grupos locais. Onde l é o número total de seguidores no sistema e é dado pela relação $l = n - k$. O custo assintótico em termos de mensagens neste passo é dado por $\mathcal{O}(n)$.

Na etapa de *commits*, (passos 4 e 5) o custo relacionado aos *commits* parciais é

$\mathcal{O}(n)$. Já os *commits* com as multi-assinaturas envolve somente a liderança e tem como custo $\mathcal{O}(k^2)$. Que também resulta em uma complexidade de $\mathcal{O}(n)$ para estes dois passos. O passo 6 corresponde ao envio da estrutura $decision_m$ aos seguidores o que resulta em $\mathcal{O}(n)$. Com isto, o custo total assintótico da proposta é linear ($\mathcal{O}(n)$) e portanto escala de forma adequada para valores de n.

Considerando que, no pior caso, f servidores falhos assumem em sequência o papel de coordenador, o custo das $f + 1$ reconfigurações necessárias é dado por $\mathcal{O}(f * n)$ considerando um custo de $\mathcal{O}(n)$ para cada reconfiguração o que implica no final um custo de $\mathcal{O}(n^2)$. O PBFT considerando f reconfigurações resulta em um custo de $\mathcal{O}(n^3)$. Portanto, a proposta desenvolvida apresenta vantagens em termos de mensagens.

5. Considerações Sobre a Proposta

O presente trabalho tem como objetivo o desempenho no alcance da consistência dos *blockchains*. Alguns trabalhos partiram para o uso do PBFT (replicação ativa) com este objetivo. Mas estes ou somente ordenavam transações ou então com um uso mais efetivo (se aplicados em blocos), ficavam limitados pelos custos assintóticos quadráticos do PBFT ($\mathcal{O}(n^2)$). Neste trabalho foi proposto um algoritmo de consenso com custo linear. Mas isto leva a um relaxamento da propriedade de acordo deste consenso. Ou seja, réplicas corretas que não concluem o consenso de forma adequada[4]. Réplicas corretas que não concluem o consenso são recuperadas pela camada *Blockchain* através de temporizações e reconfigurações.

Neste algoritmo, por envolver assinaturas em qualquer troca de mensagens, o único comportamento que envolve valor é o do coordenador, que pode mandar blocos diferentes em uma instância de consenso para seus pares, resultando no envio de um valor *default* para a camada *Blockchain*[5]. As outras réplicas se tentarem mudar os valores dos blocos, terão suas mensagens descartadas. O único efeito que podem conseguir é o atraso ou o não envio de mensagens. As réplicas corretas no sistema, em maioria, devem mascarar estes efeitos faltosos.

A situação mais crítica é o prazo *timeout* do consenso se esgotar sem decisão. Neste caso, isto significa que tivemos menos que $k/2+1$ grupos corretos entregando suas mensagens de *commit*. Na camada *Blockchain*, as réplicas que tiveram os *timeouts* esgotados, enviam pedidos de reconfiguração de seus grupos locais, envolvendo a troca de seus líderes e provocando uma nova *leadership* no sistema. Se o coordenador for malicioso (mais de um bloco no sistema) estas reconfigurações retiram o mesmo do *leadership*. E outra réplica, na sequência, assume a coordenação, propondo seu bloco para o consenso m. Estas reconfigurações ocorrem até que o consenso termine de maneira adequada.

Quando temos $k/2 + 1$ ou mais líderes corretos, as réplicas que tiveram como caminho a maioria dos líderes corretos decidiram no consenso m. Os líderes não corretos deixam seus seguidores corretos sem consenso. E, neste caso, Somente reconfigurações

[4]A propriedade de acordo para consenso determina que todas as réplicas corretas decidam pelo mesmo valor. O acordo na nossa proposta exige que a maioria das réplicas corretas decidem pelo mesmo valor.

[5]Um coordenador nunca pode fabricar ou mudar transações em um bloco porque estas estão acompanhadas pelas assinaturas dos clientes. O coordenador porém pode enviar blocos diferentes aos diversos líderes do sistema no sentido de invalidar o consenso.

não resolvem a inconsistência dos seguidores corretos. A medida que devolve a consistência é a solicitação de estado (do *blockchain*) a várias réplicas do sistema.

Porém, pode ocorrer algumas réplicas corretas isoladas em grupos com maioria de faltosas. Neste caso, o grupo é faltoso (maioria de faltosos) e não interessa o número de reconfigurações, o comportamento do grupo sempre será o mesmo. É necessária a intervenção de administradores para a renovação do grupo. Estas réplicas corretas muito dificilmente recuperam estado correto pela dependência de assinaturas de grupo.

Mas o que pode ser garantido é que em qualquer situação onde houver a decisão por um Bloco, as transações do mesmo estão registradas no *blockchain* e os clientes terão sempre a recepção da resposta correspondente.

Os valores de tempos utilizados em simulações preliminares no nosso sistema foram $time_{toS} = 1$ segundo (tempo de envio para o coordenador) e o $time_{toR} = 3$ segundos (o *timeout* dos consensos). Se considerarmos os tempos de outras abordagens que variam de 10 minutos do Bitcoin [Nakamoto 2008] a 0.5 segundos do EOS [Grigg 2017], consideramos as nossas escolhas adequadas garantindo na média um bom desempenho. Está faltando para a arquitetura proposta uma simulação que inclua o efeito das réplicas maliciosas. Nesta situação, o efeito das reconfigurações poderá ser medido, permitindo então a comparação com as atuais propostas de *blockchain* .

Os grupos no sistema proposto foram feitos para começar com uma configuração mínima, porém devem evoluir. Foi considerado que *datacenters* podem lançar novos servidores (ou diminuir) conforme a demanda de transações. Com isto, as composições de grupo podem evoluir, serem diferentes entre si, e possuirem membros em números maiores que os da composição mínima. Porém, os parâmetros definidos (f e k) devem permanecer os mesmos da composição mínima.

Outra evolução possível seriam outros *datacenters* entrarem no sistema (outras empresas) e neste caso deve haver modificação do número de líderes no *leadership*. Ou seja, k e f devem sofrer modificações adequadas; considerado que cada *datacenter* deve ter a sua representação neste grupo. As saídas destes *datacenters* também são previstas.

As propriedades de *safety* e *liveness* podem ser facilmente verificadas para este algoritmo de consenso quando mantido o limite de faltas f. O *safety* é sempre garantido devido à necessidade de obter uma maioria de assinaturas das réplicas do sistema para um bloco persistir no *ledger* distribuído. Quanto a terminação (*liveness*), esta pode ser facilmente verificada no algoritmo 1, na situação de $\lfloor \frac{k}{2} \rfloor + 1$ líderes corretos participando do consenso (que pode ser conseguida através de reconfigurações), garantindo desta forma, a maioria de réplicas corretas e a consequente terminação do algoritmo.

6. Conclusões

A principal contribuição deste artigo foi apresentar um algoritmo e uma arquitetura que reduzem o custo de mensagens de consenso em *blockchains* para $\mathcal{O}(n)$. Diferentemente dos trabalhos relacionados, falhas maliciosas podem ocorrer em componentes, em qualquer nível da hierarquia.

Ainda que com suas propriedades relaxadas, sempre é garantido que todas as réplicas corretas irão acrescentar apenas blocos corretos no seu *blockchain* e jamais duas réplicas corretas poderão decidir por blocos diferentes para o mesmo consenso.

Referências

Abraham, I., Gueta, G., and Malkhi, D. (2018). Hot-stuff the linear, optimal-resilience, one-message bft devil. *arXiv preprint arXiv:1803.05069*.

Amir, Y., Danilov, C., Dolev, D., Kirsch, J., Lane, J., Nita-Rotaru, C., Olsen, J., and Zage, D. (2010). Steward: Scaling byzantine fault-tolerant replication to wide area networks. *TDSC*.

Angraal, S., Krumholz, H. M., and Schulz, W. L. (2017). Blockchain technology: applications in health care. *Circulation: Cardiovascular Quality and Outcomes*.

Buterin, V. and Griffith, V. (2017). Casper the friendly finality gadget. *CoRR*.

Cachin, C. (2016). Architecture of the hyperledger blockchain fabric. In *Workshop on Distributed Cryptocurrencies and Consensus Ledgers*, volume 310.

Castro, M. and Liskov, B. (1999). Practical byzantine fault tolerance. OSDI '99.

Fischer, M. J., Lynch, N. A., and Paterson, M. S. (1982). Impossibility of distributed consensus with one fault process. Technical report, YALE UNIV NEW HAVEN CT.

Grigg, I. (2017). Eos: An introduction. *http://org/papers/EOS_An_Introduction.pdf*.

Kiayias, A., Russell, A., David, B., and Oliynykov, R. (2017). Ouroboros: A provably secure proof-of-stake blockchain protocol. In *CRYPTO*.

King, S. and Nadal, S. (2012). Ppcoin: Peer-to-peer crypto-currency with proof-of-stake. *self-published paper, August*, 19.

Kwon, J. (2014). Tendermint: Consensus without mining. *Draft v. 0.6, fall*.

Li, P., Wang, G., Chen, X., and Xu, W. (2018). Gosig: Scalable byzantine consensus on adversarial wide area network for blockchains. *CoRR*, abs/1802.01315.

Miller, A., Xia, Y., Croman, K., Shi, E., and Song, D. (2016). The honey badger of bft protocols. CCS '16. ACM.

Nakamoto, S. (2008). Bitcoin: A peer-to-peer electronic cash system.

Neiheiser, R., Presser, D., Rech, L., Bravo, M., Rodrigues, L., and Correia, M. (2018). Fireplug: Flexible and robust n-version geo-replication of graph databases. In *ICOIN*.

Redman, J. (2018). The reason why bitcoin miners dedicate time to mining empty blocks.

Reynal, M. (2005). A short introduction to failure detectors for asynchronous distributed systems. *SIGACT News*, 36:53–70.

Schuh, F. and Larimer, D. (2017). Bitshares 2.0: General overview.

Sousa, J., Bessani, A., and Vukolic, M. (2017). A byzantine fault-tolerant ordering service for the hyperledger fabric blockchain platform. *CoRR*, abs/1709.06921.

Svanevik, A. (2018). Why all these empty ethereum blocks?

Vukolić, M. (2016). The quest for scalable blockchain fabric: Proof-of-work vs. bft replication. In *Open Problems in Network Security*.

Zhao, J. L., Fan, S., and Yan, J. (2016). Overview of business innovations and research opportunities in blockchain and introduction to the special issue. *Financial Innovation*.