Técnicas de redução de duplicados na criptomoeda Ethereum

Luís Aguiar, Paulo Silva, João Barreto, and Miguel Matos

INESC-ID, IST Lisboa luis.silva.aguiar@tecnico.ulisboa.pt
paulo.mendes.da.silva@tecnico.ulisboa.pt joao.barreto@tecnico.ulisboa.pt
miguel.marques.matos@tecnico.ulisboa.pt

Resumo As blockchains e as criptomoedas estão a ganhar cada vez mais mercado no mundo tecnológico. Devido ao seu potencial, novas criptomoedas foram criadas como o caso do Ethereum introduzindo novos casos de uso no mundo tecnológico. O Ethereum permite o desenvolvimento de aplicações descentralizadas na sua rede. Mas apesar de esta tecnologia ter um enorme portencial, existe algumas lacunas que precisam de ser combatidas. No caso particular da disseminação, o Ethereum apresenta grande número de duplicados na disseminação de blocos e de transacções que afetam o processamento de cada nó e congestionam a rede com informação redundante. Neste artigo apresentamos a nossa solução para mitigar este problema para os blocos. Os nossos resultados mostram que conseguimos reduzir o número de duplicados para 0 sem falhas e desconexões, onde a latência também é melhorada.

Keywords: Blockchain · Criptomoedas · Ethereum · Disseminação · Duplicados · Latencia · Blocos

1 Introdução

As blockchains são bastante usadas atualmente. A blockchain é uma lista de blocos que está em contínuo crescimento conforme vão sendo introduzidos novos dados, onde cada bloco contém a hash do bloco anterior [1].

Satoshi Nakamoto foi o responsável por criar a blockchain na Bitcoin, a criptomoeda mais valiosa em termos de mercado e mais conhecida atualmente [6]. A blockchain, no caso da Bitcoin e Ethereum, atua como um histórico público de transações, sendo estas guardadas em cada bloco da blockchain. Para um bloco ser válido é preciso gastar algum trabalho, no caso destas criptomoedas, poder computacional [3]. Os nós têm que encontrar um nonce, de forma a que a hash do bloco seja inferior a um determinado valor definido pela rede, a este processo dá-se o nome de Proof-of-Work (POW). Então para alterar o conteúdo de um bloco é preciso refazer o trabalho, uma vez que a hash do bloco vai ser alterada e é preciso recalcular o nonce, sendo por isso difícil alterar os seus dados. A blockchain é partilhada entre os vários nós da rede, não tendo um ponto central de vulnerabilidade nem de falha [2]. Mas para que um dado bloco seja introduzido é necessário que os nós concordem quando é que este deve ser

considerado válido ou não [4]. A este mecanismo de mútuo acordo, em Sistemas Distribuídos, chamamos de consenso e o protocolo de consenso usado na Bitcoin tem o nome de *Nakamoto Consensus* [5].

Mas a blockchain também tem os seus pontos fracos. Como é preciso gastar algum trabalho para que o bloco seja válido, o número de transacções processadas é bastante afetado: apenas algumas transacções são efetuadas por segundo. O Visa consegue suportar 24000 transações por segundo (tps), enquanto que a Bitcoin apenas 7 tps [10,8,3], ou seja, bastante menos. O Ethereum é outra criptomoeda bastante utilizada atualmente, estando em segundo lugar em termos de valor do mercado [6], conseguindo processar mais transações por segundo que a Bitcoin, mas mesmo assim bastante menos que o Visa (processa cerca de 15 tps) [11]. O foco do Ethereum é permitir o desenvolvimento de aplicações sobre a sua plataforma. A essas aplicações dá-se o nome de Decentralized Applications (DApps) [9,11]. Recentemente devido à falta da escalabilidade do Ethereum, o jogo CriptoKitties desenvolvido em cima da sua blockchain conseguiu quebrar a rede, tal a quantidade de utilizadores a jogarem o jogo [15].

O número de transações processadas por segundo também é afetado devido a parte do trabalho da rede poder a vir a ser desperdiçado [5]. Por exemplo, um nó gera um bloco *b* e propaga-o para a rede. Entretanto outro nó que ainda não recebeu o bloco *b* gera um bloco concorrente *b'* disseminando-o também para a rede. A rede vai ver então dois blocos concorrentes como verdadeiros: a este processo damos o nome de *fork*. Tanto a Bitcoin como o Ethereum resolvem este problema escolhendo a sequência de blocos que tiver mais trabalho. Ou seja, mais tarde, quando outro bloco for gerado vai ter como antecessor um dos blocos concorrentes e essa sequência vai ter mais trabalho que a outra, logo vai ser escolhida como a verdadeira e a outra sequência vai ser descartada, havendo então trabalho desperdiçado. Mesmo que um bloco concorrente não seja gerado, o tempo desperdiçado a tentá-lo gerar por não saber da existência do outro bloco, já incorre de trabalho desperdiçado. A disseminação tem um papel fundamental na ocorrência de trabalho desperdiçado. Se a disseminação for muito lenta, o tempo gasto a tentar gerar um bloco concorrente é maior, podendo em certos casos gerar *forks* [5].

Outra coisa a ter em conta, no algoritmo de disseminação, para além da rapidez é o número de informação duplicada que cada nó recebe, desperdiçando a computação de cada nó e também congestionando a rede com informação redundante [7]. Uma abordagem simplificada em que todos os nós enviam informação para todos os nós que eles conhecem, apesar de a informação chegar mais rápido, o número de duplicados vai ser bastante grande. Então é necessário reduzir o número de duplicados que afeta a largura de banda e o processamento de cada nó, sem comprometer a rapidez da disseminação.

O objetivo deste artigo é melhorar o algoritmo de disseminação da blockchain, onde nos focámos no protocolo do Ethereum. Analisámos o código do Geth, o cliente de Ethereum mais usado na versão 1.8.17-stable para observar como é que a disseminação acontece no Ethereum e tentar melhorá-la. Focámo-nos mais na parte dos blocos, onde conseguimos reduzir o número de duplicados sem

qualquer falha e desconexão para zero e melhorámos a latência de disseminação, em alguns casos significativemente.

Na Secção 2 introduzimos alguns conceitos básicos e o estado de arte para melhor compreender a nossa solução, na Secção 3 explicamos a nossa abordagem de forma a reduzir o número de blocos duplicados, na Seccção 4 mostramos os resultados obtidos do Ethereum com a nossa solução e como ele se encontra atualmente, onde o número de duplicados é bastante reduzido e finalmente na Secção 6 concluimos o nosso artigo.

2 Trabalho Relacionado

2.1 Redes Sobrepostas

Cada nó mantem uma lista de nós com quem ele comunica. A essa lista damos o nome de *vista* e a todos os nós contidos nessa lista damos o nome de *vizinho*. O conjunto de todas as vistas de todos os nós damos o nome de *rede sobreposta* (em inglês *overlay network*). A rede sobreposta é uma rede de computador construída em cima de outra em particular a rede física.

2.2 Disseminação

Existem três principais abordagens para disseminar os dados: *inundação* (em inglês *flooding*), *árvore* e *epidémico* também conhecido como *gossip* [7,14,16].

- Inundação: Mais simples, onde a mensagem é enviada para todos os vizinhos o que resulta em muitos duplicados na rede.

- Árvore: É criada uma estrutura em árvore sem qualquer ciclo para disseminar os dados. A raiz da árvore envia a mensagem para os seus filhos e os filhos da raiz para os seus filhos e assim sucessivamente. Esta abordagem leva a não haver duplicados, mas na presença de falhas e desconexões esta estrutura precisa de ser recalculada e só depois de estar recalculada é que se pode enviar a mensagem, para garantir que todos os nós da rede a recebem.

- Epidémico: A mensagem é enviada para um número aleatório de nós escolhidos da vista. O tamanho dos nós escolhidos dá-se o nome de *fanout*. Esta técnica é mais robusta, porque esta escolha aleatória permite a criação de múltiplos caminhos entre dois nós, continuando a mensagem a ser entregue a todos os nós com elevada probabilidade na presença de falhas e desconexões. Cria-se uma *overlay* não estruturada. Existem duas abordagens de como a mensagem pode ser transmitida: o nó que envia toma a iniciativa e envia a mensagem mal a recebe (abordagem *push*) ou então os nós podem perguntar periodicamente por uma nova mensagem a determinados vizinhos (abordagem *pull*). Estas abordagens podem ser combinadas com a abordagem *eager* e *lazy*. Na *eager* a mensagem é enviada para todos os nós, enquanto que na *lazy* primeiro é enviado um anúncio e só depois a mensagem (caso o vizinho

não a tenha). As abordagens *eager* e *lazy* representam um compromisso entre latência e largura de banda. Se a mensagem é sempre enviada para os vizinhos sem anunciá-la primeiro, a latência é mínima (o nó apenas envia a mensagem), mas a largura de banda vai ser maior, porque o número de informação duplicada vai ser grande (o nó vai enviar a mensagem mesmo se o recetor já a tenha recebido por outro nó). No caso do *lazy*, o recetor recebe primeiro um anúncio e só depois a mensagem se não a tiver. Mais mensagens vão ser trocadas na rede (o anúncio e a mensagem em si), mas o número de duplicados é zero.

2.2.1 Sistemas de Disseminação

O Brisa [16] é um sistema de disseminação híbrido que só funciona para ambientes onde não existem nós bizantinos que combina a robustez e a escalabilidade do epidémico com a eficiência das estruturas de disseminação (árvores e grafos acíclicos dirigidos (DAGs)). O Brisa é desenvolvido de forma a que na presença de falhas e desconexões este sistema é rapidamente reparado.

Inicialmente todos os nós enviam a mensagem para a sua vista (aplicam o *flooding*), depois cada nó seleciona um deles como o seu pai (por exemplo aquele que lhe enviou primeiro a mensagem) e envia uma mensagem de desativação para todos os outros nós, isto é, diz aos outros nós para lhe deixarem de enviar futuras mensagens. Então futuras mensagens vão ser enviadas apenas pelo seu pai. Sem falhas e sem desconexões, o Brisa é altamente eficiente e não tem qualquer duplicado. Mas quando o pai se desconecta ou falha, o filho envia uma mensagem de ativação para todos os seus vizinhos, para estes lhe passarem a enviar futuras mensagens. Depois o primeiro nó que lhe enviar a próxima mensagem é escolhido como o próximo pai. Neste caso, já vai haver duplicados.

Outra coisa a ter em conta é que pode haver muitos geradores de mensagens na rede, logo isso leva a ter várias estruturas de disseminação. Cada estrutura de disseminação do Brisa está identificada por um ID único. Quando um nó gera uma mensagem ele vai escolher como estrutura aquela onde ele está mais próximo da raiz. A mensagem vai ser então enviada para o seu pai e filhos e assim sucessivamente até esta chegar à raiz. Depois de chegar à raiz funciona normalmente, onde a mensagem vai ser enviada para os seus filhos que ainda não receberam a mensagem. De notar que, apesar de não ser tão eficiente como ter uma estrutura de disseminação para cada gerador, os nós apenas têm que guardar poucas estruturas. Se um sistema tivesse N nós e todos pudessem ser geradores então cada nó teria que guardar também N estruturas.

Para além do Brisa existe o Thicket [17], algoritmo que serve para criar várias árvores onde cada nó é poucas vezes nó interior e mais vezes nó folha. Assim, na presença de falhas é possível que a mensagem continue a chegar a todos os nós. Tal como o Brisa, este algoritmo só funciona para modelos *crash*.

2.3 Disseminação do bloco no Ethereum

A disseminação do bloco está representado na Figura 1. O nó gera o bloco (encontra o nonce de forma a que a hash do bloco seja inferior a um determinado valor definido pela rede) (passo 1) e propaga-o para o máximo entre 4 e a raiz quadrada dos seus vizinhos (NewBlocksMsg) (passo 2). Note que o nó só envia o bloco para nós que ele sabe que não o têm. Se o número de nós que não tem o bloco é inferior a 4 então ele envia o bloco apenas para esses. Os restantes vizinhos, se existirem, o nó vai enviar a hash do bloco e o número (NewBlockHashesMsg) (passo 3). Depois cada um desses nós vai seguir os seguintes passos:

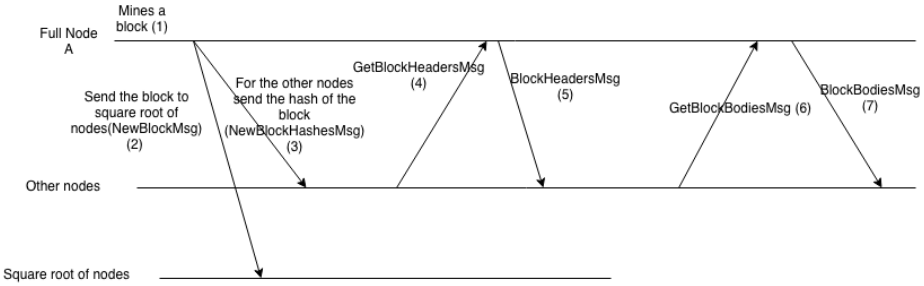

Figura 1. Dissemination of a block in the Ethereum Network.

1) Se a hash é desconhecida na sua blockchain, ele vai pedir pelo *block header* (GetBlockHeaderMsg) (passo 4). Se o nó recebe a hash por mais que um peer então ele pede a um desses nós de forma aleatória; 2) Depois de receber o *header* (passo 5) o nó faz algumas verificações como verificar se estava à espera daquele *header* e se, entretanto, não recebeu o bloco de outro nó. De notar que o nó pode receber entretanto o bloco completo de outro peer e se isso acontecer então ele já tem a informação toda necessária e não precisa de pedir pelo resto do bloco. Pode acontecer também que o corpo do bloco é vazio, isto é não tem qualquer tipo de transações. Se isto acontecer, então o nó não pede o corpo e adiciona-o logo à sua blockchain; 3) Em seguida, se o corpo do bloco não for vazio e se, entretanto, não recebeu o bloco completo de outro nó, então ele vai pedir pelo corpo do bloco (GetBlockBodiesMsg) (passo 6); 4) Quando o corpo é recebido (BlockBodiesMsg) (passo 7), ele verifica se o *header* corresponde ao corpo que recebeu e se já tem o bloco na chain. Se não, então verifica o proof-of-work do bloco usando o *header* e se for válido, o nó vai propagá-lo imediatamente para o máximo entre 4 e a raiz quadrada dos seus vizinhos que ele sabe que ainda não têm o bloco. Note que este processo pode levar a duplicados, porque o vizinho pode já ter recebido o bloco de outro nó, mas torna a disseminação mais rápida. Em seguida vai então validar o bloco, verificando se todas as transações estão corretas e se estiver tudo em ordem, então o nó vai anunciar o bloco para os restantes vizinhos (envia a hash e o número).

O passo 4 é o único passo feito pelos nós que receberam logo o bloco completo, através da mensagem NewBlocksMsg.

3 Abordagem

Nesta secção, descrevemos a nossa abordagem, que tem como objetivo reduzir o número de blocos duplicados na rede. Dado os blocos ocuparem uma parte substancial da largura de banda, neste trabalho focamo-nos exclusivamente neles. No Ethereum, o número de blocos duplicados continua a ser bastante grande, apesar de ser aplicada uma combinação das abordagens *eager* e *lazy*. Sempre que um nó recebe um novo bloco válido, vai propagá-lo para o máximo entre 4 e a raiz quadrada dos seus vizinhos, sem perguntar a estes se conhecem ou não o bloco, originando duplicados.

Para tentar reduzir estes duplicados e melhorar a latência da rede, a nossa solução passa por construir uma árvore de disseminação, sendo esta muito parecida com a do Brisa, mas adaptada para um ambiente bizantino. A ideia base do trabalho assenta na observação que apenas algumas mining pools (utilizadores que atuam em conjunto para gerar um bloco válido) é que são responsáveis pela geração de blocos: "Ethermine", "SparkPool", "Nanopool","F2_Pool2" e "MiningPoolHub_1" [19]. Isto abre a oportunidade para explorar uma abordagem baseada em árvores, visto que cada nó tem que manter poucas. Para identificar cada árvore, usa-se como identificador o endereço do nó que gerou o bloco que se encontra no *coinbase address*. Quando um nó recebe um bloco olha para o endereço que está no *coinbase address* e usa-o para identificar a árvore que vai usar.

A árvore é construída de forma idêntica à do Brisa: inicialmente, todos os nós enviam o bloco para todos os seus vizinhos e escolhem como pai o que lhe enviou primeiro o bloco, enviando uma mensagem de desativação para todos os outros nós. Esta mensagem de desativação vai informar os outros vizinhos para passarem a anunciar o bloco, em vez de o enviarem diretamente (começam a aplicar uma *lazy* approach). Estes nós têm que continuar a anunciar o bloco, pois o pai pode ser malicioso e decidir não enviar a mensagem: o ambiente é bizantino. A árvore está então construída, onde todos os *links* para onde o bloco é enviado diretamente constituem a árvore.

Depois de estar construída a árvore, quando um segundo bloco é gerado por um determinado nó, os nós vão usar o *coinbase address* para identificar a árvore a usar na disseminação e enviam o bloco diretamente para todos os vizinhos que não lhe enviaram nenhuma mensagem de desativação para esse *coinbase address*. Para os restantes ele anuncia o bloco. Se não houver nenhum nó bizantino, nenhuma falha e desconexão e nenhum problema na rede, o número de duplicados vai ser 0, uma vez que todos estão a seguir o protocolo e a árvore não precisa de ser reconstruída.

Para cada *coinbase address* existe apenas uma árvore. Se para um determinado *coinbase address*, um bloco ainda desconhecido é recebido primeiro por um nó que não é pai (e.g., problema na rede, falha ou desconexão por parte do pai

ou o pai é bizantino), este passa a ser pai e o anterior deixa de o ser, sendo enviada uma mensagem de ativação para o novo pai (i.e., avisa o nó para enviar o próximo bloco diretamente) e uma mensagem de desativação para o antigo.

Caso o pai falhe ou se desconecte, o novo pai pode ser escolhido de duas formas: 1) o nó envia uma mensagem de ativação para todos os seus vizinhos, isto é, avisa a todos os seus vizinhos para lhe passarem a enviar blocos futuros e depois escolhe como pai o que lhe enviar primeiro o bloco, levando a muitos duplicados; 2) o nó escolhe como pai o vizinho que lhe tem anunciado mais vezes primeiro o bloco para aquele *coinbase address*, levando a 0 duplicados.

Na presença de nós bizantinos, estes podem atacar o sistema de quatro maneiras: 1) não enviando o bloco recebido para os seus vizinhos, afetando apenas o tempo da recepção do bloco, caso este seja pai. Os seus vizinhos continuam a receber o bloco por outros nós e o que lhe enviar primeiro passa a ser pai e o nó bizantino deixa de o ser (é enviada uma mensagem de ativação para o novo pai e uma mensagem de desativação para o antigo); 2) não seguindo o protocolo e enviando os blocos para todos os seus vizinhos, afetando o número de duplicados no sistema. Exatamente o mesmo problema pode acontecer no Ethereum Normal, pelo que não vai ser abordado no resto do artigo; 3) enviando um bloco com *header* e corpo inválido, ataque que também não é abordado na avaliação, porque os efeitos causados no Ethereum Normal seriam os mesmos que na nossa solução (congestionamento na rede); 4) enviando blocos com headers válidos e corpo inválido, caso abordado na avaliação.

De forma a aplicar a nossa abordagem, tivemos que perceber todo o código do Geth e adaptá-lo de forma a funcionar na nossa rede local, onde foram alteradas cerca de mil linhas de código. Em seguida, mostramos os resultados obtidos aplicando a nossa abordagem.

4 Avaliação

Nesta secção, avaliamos a nossa abordagem num ambiente realista, onde medimos o número de blocos duplicados na rede, primeiro por bloco e depois o número de duplicados que cada nó recebeu. Para além disso medimos a latência que cada bloco demora a chegar a todos os nós.

Desenvolvemos uma blockchain privada na nossa rede local usando o Geth na versão 1.8.17-stable, que é a implementação do cliente do Ethereum mais usada. Corremos 45 nós na mesma máquina que tem as seguintes especificações: uma Intel(R) Xeon(R) Gold 6138 CPU correndo a 2.00GHz com 20 cores, cada uma com 2 threads e duas 32GiB DIMM DDR4 Synchronous 2666 MHz de RAM. Apenas três desses nós são geradores de blocos, simulando o ambiente real: apenas uma baixa percentagem de nós é responsável por gerar blocos.

De forma a serem gerados blocos na nossa blockchain privada mudámos o POW do Ethereum, onde cada bloco é gerado aproximadamente de 20 em 20 segundos (tempo médio de geração de blocos na rede do Ethereum), caso contrário iria demorar bastante mais tempo para um bloco ser gerado. Nesta versão alterada é usado um temporizador, onde o tempo de geração de cada bloco segue

uma distribuição de Poisson. No caso das transacções, são injetadas 4 por segundo, ainda assim bastante menos em comparação às que o Ethereum recebe (cerca de 12 a 15 transacções por segundo [19]), sendo depois estas propagadas na rede seguindo o protocolo do Ethereum. Injetamos apenas 4 por segundo para não sobrecarregar muito a nossa máquina e assim ter mais nós a correr e mesmo injetando mais transacções por segundo, os resultados iriam ser praticamente os mesmos.

Corremos testes de uma hora, onde ignorámos os 10 minutos iniciais e finais, para o Ethereum Normal e para a nossa implementação, onde tivemos que introduzir novas mensagens na rede de forma a ser construída a árvore de disseminação, como explicado anteriormente.

Primeiro testámos sem quaisquer nós bizantinos e depois com nós bizantinos, havendo no decorrer das experiências algumas desconexões. Usámos a ferramenta NEED [20], para introduzir latência na rede e definir a largura de banda e assim simular o ambiente do Ethereum da forma mais realista possível. Separámos os nós em vários grupos (4 grupos para a experiência sem qualquer nó bizantino e 6 grupos para a experiência com nós bizantinos) e definimos a latência entre eles e a largura de banda. Entre o *grupo1* e o *grupo2* a latência é de 50ms e o upload e download de 100Mbps, entre o *grupo1* e o *grupo3* a latência é de 100ms e o upload e download de 50MBs, entre o *grupo1* e *grupo4* a latência é de 1000ms e o upload e download de 20MBs, entre o *grupo1* e o *grupo5* a latência é de 60ms e o upload e download de 30Mbps e entre o *grupo1* e *grupo6* a latência é de 80ms e o upload e download de 10Mbps. Na experiência sem nós bizantinos, o *grupo1*, *grupo2* e *grupo3* têm 10 nós cada um e o *grupo4* 15 nós, enquanto que na experiência com nós bizantinos o *grupo1*, *grupo2*, *grupo3* têm 10 nós e o *grupo4* tem 6, o *grupo5* tem 5 e o *grupo6* tem 4. Estes dois ultimos grupos correspondem a nós bizantinos (*grupo5*, *grupo6*).

Os resultados dos duplicados da experiência sem quaisquer nós bizantinos encontram-se representados na Figura 2 e na Figura 3, onde se pode verificar que os duplicados são reduzidos para 0. Na Figura 2 está representada uma Função de distribuição acumulada (em inglês usa-se a abreviação *CDF*) por número de duplicados por bloco, onde se pode verificar que 100% dos blocos tiveram 0 duplicados. Na Figura 3 está representada uma *CDF* do número de blocos duplicados que cada nó na rede recebeu e como se pode observar na nossa solução nenhum nó recebeu qualquer duplicado em contraste com o Ethereum Normal, onde houve nós a receber 400 duplicados.

Avaliámos também os sistemas com 20% de nós bizantinos, onde numa rede de 45 nós, 9 deles são bizantinos (4 não enviam sempre os blocos e 5 enviam blocos com *header válido* e corpo inválido). Por cada bloco criado são gerados 5 blocos inválidos (1 por cada nó bizantino). Como se pode ver na Figura 4 e na Figura 5, os resultados são praticamente idênticos aos resultados sem qualquer nó bizantino, onde os duplicados são aproximadamente 0. Isto acontece, porque é improvável que os nós bizantinos contribuam para a reconstrução da árvore, uma vez que raramente são pais (se enviarem sempre blocos inválidos nunca vão ser pais e se só enviarem algumas vezes os blocos precisam de ser mais rápidos

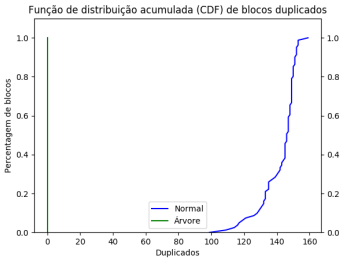

Figura 2. Quantidade de blocos duplicados na rede por bloco.

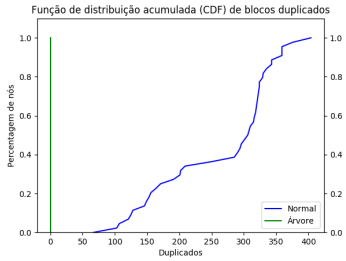

Figura 3. Quantidade de blocos duplicados que cada nó recebeu.

que todos os outros nós, de forma a que esse bloco chegue mais rápido do que o mesmo enviado por outro nó). Como existiram algumas desconexões, a árvore de disseminação foi reconstruída algumas vezes, sendo esta a principal causa dos duplicados apresentados na Figura 4. Podem acontecer fundamentalmente duas situações que conduzem a duplicados: 1) O pai desconecta-se e leva a que o nó tenha que escolher outro vizinho como pai: ou é enviada uma mensagem de ativação para todos os seus vizinhos (podendo haver vizinhos - 1 duplicados na receção do próximo bloco), ou então caso exista, é escolhido o vizinho que lhe enviou o anúncio mais vezes em primeiro lugar (levando a 0 duplicados na próxima recepção do bloco). 2) Outras vezes o nó recebe primeiro o bloco de um nó que não era o seu pai. Isto pode acontecer, se a árvore for reconstruída num ramo acima e então o caminho mais rápido pode ser alterado. O nó muda então de pai enviando uma mensagem de ativação para este e uma de desativação para o antigo. Mas existe uma janela em que o antigo pai ainda não recebeu a mensagem de desativação, e envia mesmo assim o bloco completo para o nó: levando então a um duplicado.

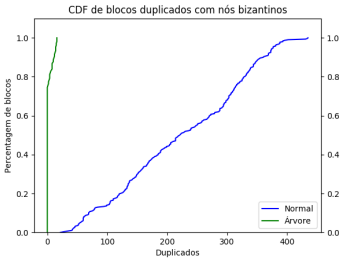

Figura 4. Quantidade de blocos duplicados na rede por bloco

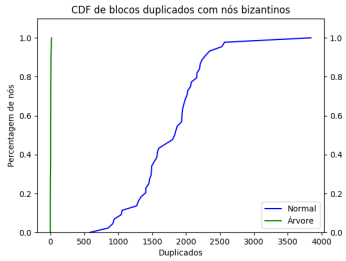

Figura 5. Quantidade de blocos duplicados que cada nó recebeu

Outra métrica que utilizámos foi medir o número de blocos com *header* válido e corpo inválido que cada nó recebeu, quando o bloco é criado. Por cada bloco

inválido analisamos quantas vezes os nós da rede o receberam e calculámos a média. Os resultados estão representados na Figura 6. O Ethereum Normal apresenta um valor superior em relação à nossa abordagem, porque os nós verificam apenas o *header* do bloco e em seguida enviam o bloco para o máximo entre 4 e a raiz dos seus vizinhos. Isto apesar de tornar a disseminação relativamente mais rápida, a quantidade de informação duplicada e inválida vai ser maior. Na nossa abordagem, os nós apenas enviam os blocos para os seus filhos e anunciam para os restantes vizinhos, quando verificam o bloco completo. Logo o número de blocos inválidos vai ser igual ao número de nós a quem o nó bizantino enviou o bloco.

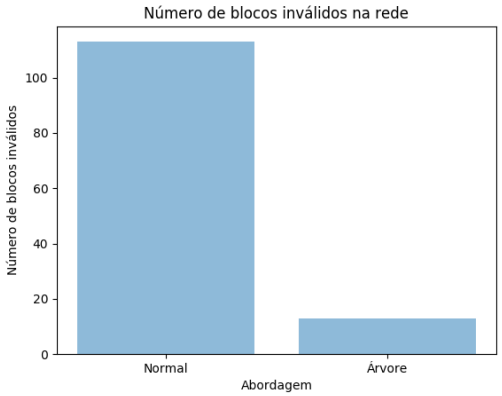

Figura 6. Número de blocos inválidos na rede.

No caso da latência, esta também é melhorada com a nossa abordagem (Figura 7 e Figura 8), porque a árvore é construída com base na rapidez da recepção do primeiro bloco, onde o pai se não for bizantino envia o bloco para o seu filho. Mas na presença de falhas e desconexões, a árvore tem que ser recalculada, acabando por haver melhores caminhos. Na presença de falhas, pode acontecer que um bloco indireto transmitido por um dos vizinhos que não é pai chegue mais rápido que o bloco direto transmitido pelo seu pai. Isto significa que havia um caminho rápido, pois o nó teve tempo de trocar várias mensagens até receber o bloco indiretamente (ver Figura 1), acabando pela latência da disseminação ser alterada. No caso do Ethereum Normal, os vizinhos são sempre escolhidos de forma aleatória em vez da rapidez, levando então a uma latência maior. A Figura 7 e a Figura 8 representam uma CDF do tempo que cada bloco demorou a chegar a todos os nós.

Como se pode observar na Figura 8, a latência é bastante grande no Ethereum Normal com nós bizantinos, havendo muitos blocos a demorarem vários minutos a chegarem a todos os nós. Isto acontece, porque no Ethereum Normal são en-

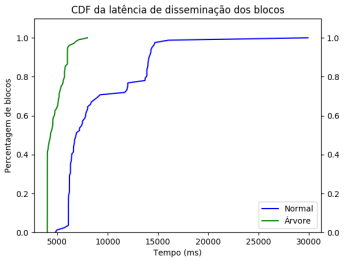

Figura 7. Tempo que cada bloco demorou a chegar a todos os nós, sem nós bizantinos.

Figura 8. Tempo que cada bloco demorou a chegar a todos os nós, com nós bizantinos.

viados bastantes blocos inválidos, acabando por congestionar o processamento dos blocos em cada nó.

5 Agradecimentos

Este trabalho é parcialmente financiado pelo Fundo Europeu de Desenvolvimento Regional (FEDER) através do Programa Operacional Regional de Lisboa e por Fundos Nacionais através da FCT - Fundação para a Ciência e a Tecnologia no âmbito do projeto Lisboa-01-0145-FEDER-031456 (Angainor) e UID/CEC/50021/2019.

6 Conclusão

A solução em árvore é uma boa solução para reduzir o número de duplicados, onde estes são reduzidos para praticamente 0 sem qualquer nó bizantino, falhas e ou desconexões e sem problemas na rede. Mesmo na presença de falhas e desconexões e com nós bizantinos, o número de duplicados é bastante melhorado, onde no pior dos casos de cada uma das abordagens por bloco o valor é reduzido de 400 para cerca de 20 duplicados. Os blocos inválidos na rede também são bastante reduzidos, isto porque na nossa abordagem o nó verifica primeiro o bloco completo e só depois é que o envia e a latência também é melhorada. Como contrapartida, se houverem muitos nós a gerarem blocos, muitos nós têm que manter muitas árvores de disseminação.

De trabalho futuro, vamos tentar reduzir o número de transacções duplicadas, aplicando uma política similar ao dos blocos do Ethereum Normal e tentar avaliar os resultados em sistemas maiores, com mais nós na rede e avaliar o nosso sistema usando a cloud com máquinas bastante separadas geograficamente.

Referências

1. Weber, Ingo and Gramoli, Vincent and Ponomarev, Alex and Staples, Mark and Holz, Ralph and Tran, An Binh and Rimba, Paul: On availability for blockchain-based systems (2017)
2. Cachin, Christian and Vukolić, Marko: Blockchains consensus protocols in the wild, Journal arXiv preprint arXiv:1707.01873 (2017).
3. Nakamoto, Satoshi: Bitcoin: A peer-to-peer electronic cash system (2008)
4. Decker, Christian and Wattenhofer, Roger: Information propagation in the bitcoin network, Peer-to-Peer Computing (P2P), IEEE Thirteenth International Conference on (1–10) (2013)
5. Natoli, Christopher and Gramoli, Vincent: The blockchain anomaly, Journal arXiv preprint arXiv:1605.05438 (2016)
6. CoinMarketCap : Cryptocurrency Market Capitalizations, https://coinmarketcap.com/ (2019)
7. Miguel Matos: Epidemic Algorithms for Large Scale Data Dissemination, PhD thesis, Universidade do Minho (2013)
8. Bach, LM and Mihaljevic, B and Zagar, M: Comparative analysis of blockchain consensus algorithms, 2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO) (2018)
9. Ethereum Foundation: Ethereum White Paper, https://github.com/ethereum/wiki/wiki/White-Paper
10. Sompolinsky, Yonatan and Zohar, Aviv: Secure high-rate transaction processing in bitcoin, International Conference on Financial Cryptography and Data Security on (507–527) (2015)
11. Wood, Gavin : Ethereum yellow paper, https://ethereum.github.io/yellowpaper/paper.pdf"
12. Eugster, P Th and Guerraoui, Rachid and Handurukande, Sidath B and Kouznetsov, Petr and Kermarrec, A-M: Lightweight probabilistic broadcast, Journal ACM Transactions on Computer Systems (TOCS) (341–374) (2013)
13. Kim, Seoung Kyun and Ma, Zane and Murali, Siddharth and Mason, Joshua and Miller, Andrew and Bailey, Michael: Measuring Ethereum Network Peers, Proceedings of the Internet Measurement Conference 2018 (91–104) (2018)
14. Eugster, Patrick T and Guerraoui, Rachid and Kermarrec, A-M and Massoulié, Laurent: Epidemic information dissemination in distributed systems (60–67) (2004)
15. Kharif, Olga:Cryptokitties mania overwhelms ethereum network's processing
16. Matos, Miguel and Schiavoni, Valerio and Felber, Pascal and Oliveira, Rui and Riviere, Etienne: Lightweight, efficient, robust epidemic dissemination Journal Journal of parallel and distributed computing (987–999) (2013)
17. Rodrigues, Luís and Leitão, João and Ferreira, Mário: Thicket: Construção e Manutenção de Múltiplas Árvores numa Rede entre Pares
18. Bitcoin Wiki: Protocol rules, https://en.bitcoin.it/wiki/Protocol_rules
19. EtherScan: Ethereum (ETH) Blockchain explorer, https://etherscan.io/
20. Neves, João: Container Network Topology Modelling, Master Thesis, IST (2018)