Bitcoin-ish Miner (uma possível resolução)

Joana Parreira, *Student, FCT-UNL,* Jorge Pereira, *Student, FCT-UNL*

1 Introdução

O termo *blockchain* foi primeiramente cunhado em 1991, por Stuart Haber e W. Scott Stornetta [1]. A visão seria de usar esta tecnologia de forma a criar um *timestamp* digital, em que posteriormente não fosse possível ser alterado. Durante muitos anos este termo caiu no esquecimento ,mas em 2008 o termo *blockchain* ficou popular com a invenção da *Bitcoin* [2]. A *bitcoin* é um tecnologia de *ledger* distribuído que utiliza um protocolo *prove-of-work* como protocolo de *consensus* de forma a estabelecer uma ordem entre as transações.

O objectivo deste trabalho [3] é simular uma versão muito simplificada deste mecanismo de *consensus*: dado uma mensagem M e o número máximo N que o *nonce* pode tomar, calcular o *hash sha256*, resultante da concatenação entre M e o *nonce* n, entre 0 e N, que toma o maior valor.

2 Solução Implementada

A solução implementada é composta por dois grandes componentes: a) um cliente que é responsável por criar o *workload* das transacções que deverão ser minadas; b) um servidor que é responsável por calcular o valor do maior *hash* nas condições especificadas pelo cliente. Nesta secção iremos focar-nos no componente servidor, pois é nele que se encontra a maior complexidade do problema.

Sempre que um cliente pretende iniciar o cálculo de um *hash*, inicia uma conexão TCP para um servidor disponível, enviando a mensagem a ser usada e o número máximo para o *nonce*.

Por sua vez, o servidor começa por criar um *miner* e uma cache, de seguida cria uma *go routine* por cada conexão aceite e cujo pedido tenha sido avaliado como correto. Nessa *go routine*, o servidor invoca o *miner* para fazer a submissão do pedido com a mensagem e o número máximo para o *nonce*, e aguarda pela resposta, a qual é guardada na cache para possível consulta posterior. A resposta corresponde ao *hash* da mensagem e ao *nonce* que permitiu obter o *hash* de maior valor. O resultado obtido é enviado de volta para o cliente, que o imprime.

2.1 Miner

Foquemo-nos agora nos detalhes da implementação do *miner*.

O *miner* possui dois canais para a receção de pedidos por parte dos *workers*, um canal para pedidos mais pequenos (*inLight*), ou seja, com um intervalo mais pequeno de *nonces* a serem testados, e outro para pedidos maiores (*inHeavy*).

Existem dois tipos de *workers*, consoante a sua prioridade seja tratar de pedidos de um canal ou de outro. Cada *worker* executa em ciclo infinito, estando à escuta do canal que lhe ficou atribuído e, apenas no caso de esse canal estar vazio no momento, ele passa a ficar também à escuta no outro canal (Listagem 4). Assim que obtém um bloco, calcula os diferentes *hashes* com os vários *nonces* dentro do intervalo pretendido, guardando o *hash* com maior valor e o *nonce* correspondente, e retornando-os como *workResp*.

O *miner* recebe do servidor pedidos de trabalho com a mensagem a ser usada e o valor máximo de *nonce* a ser testado. Consoante a extensão do intervalo de *nonces* a serem testados, pode ser necessário dividir o trabalho em vários blocos, de acordo com o intervalo máximo definido para os mesmos. Assim, cada bloco tem no máximo *THRESHOLD_BLOCK_SIZE nonces* a serem testados. Os blocos são submetidos ao canal para pedidos mais pequenos caso o seu número total seja inferior a *THRESHOLD_LIGHT_HEAVY*, e ao canal para pedidos maiores caso contrário. Os blocos submetidos correspondem a um *workReq*, o qual possui a mensagem, o intervalo de *nonces* do respetivo bloco, e um canal usado para envio da resposta, o qual é

comum a todos os blocos do mesmo pedido. Esta submissão nos canais é realizada por uma *go routine*.

Enquanto ocorre esta subdivisão do trabalho e sua submissão por partes aos canais, o *miner* vai estar à escuta no canal de resposta correspondente ao pedido em causa. As respostas obtidas por esse canal correspondem aos diferentes blocos, pelo que, adicionalmente, é necessário comparar de entre as respostas intermédias qual a que tem um *hash* maior, correspondendo assim à resposta final.

2.2 Cache

Adicionalmente, implementámos também um sistema de cache, o qual permite obter resultados de forma mais rápida para pedidos repetidos, ou seja, com a mesma mensagem e o mesmo número de *nonce* máximo.

A cache corresponde a um mapa, cuja chave é uma string que representa o pedido (com a sua mensagem e o *nonce*) e cujo valor é uma estrutura *entry*, composta pela resposta computada pelo servidor e pela sua validade na cache. A cache tem um valor específico associado à validade e cada novo elemento terá o mesmo tempo de validade, o qual se inicia quando adicionado. No caso da validade ser 0, as entradas não perdem a validade.

Adicionalmente, a cache tem dois canais, um para receber novas entradas a serem adicionadas (*inchan*), recebendo a chave e o valor resposta, e outro para receber os pedidos de consulta à cache (*outchan*), recebendo a chave do pedido em questão e o canal que deve ser usado para enviar a resposta.

Aquando da criação da cache é criada uma *go routine* para lidar com os pedidos que chegam à cache (Listagem 1). Nesta, é ativado um *timeout* a cada 100 milissegundos, ativando a remoção de todas as entradas cuja validade tenha expirado. Além disso, esta *go routine* lê de ambos os canais da cache. Ao ler do *inchan*, guarda no mapa da cache a nova entrada. Quando lê do *outchan*, procura pela entrada com a chave pretendida e, caso não exista, retorna *nil* no canal preparado para a resposta. Caso exista, retorna o valor resultado guardado.

Listing 1
Código da cache que lida com os pedidos de criação de novas entradas ou de consulta à cache

```
1  wait := time.Millisecond * 100
2  timeout := time.After(wait)
3  for {
4    select {
5    case <-timeout:
6      c.evictEntries()
7      timeout = time.After(wait)
8    case el := <-c.inchan:
9      c.entries[el.key] = entry{
10       obj: el.e,
11       exp: returnExp(c),
12     }
13   case el := <-c.outchan:
14     i, ok := c.entries[el.reqKey]
15     if ok {
16       el.respchan <- &i.obj
17     } else {
18       el.respchan <- nil
19     }
20   }
21 }
```

3 CORRECTNESS

Cada pedido recebido pelo servidor é verificado antes de ser submetido aos *workers*. Esta verificação é feita: a) ao comparar o tipo da mensagem recebida, a qual deve ser de tipo *Request*; b) ao comparar os limites superior e inferior de *nonce* fornecidos, garantindo que o limite superior é sempre maior ou igual ao inferior; c) ao averiguar que a mensagem não é uma string vazia.

Foram também criados testes para o *miner*, para verificar se a resposta obtida é a correta. Para tal, foi criado um teste que gera mensagens e *nonces* máximos aleatórios. Existe também uma versão mais leve, com menor número de testes e com valores menores de *nonce*, para uma execução mais rápida. Cada teste realiza o pedido ao *miner* e, de seguida, realiza a mesma computação de forma sequencial, comparando os resultados obtidos pelos dois métodos.

4 WORKER CREATION

Os *workers* são criados depois do servidor começar a executar, antes de receber qualquer pedido de conexão. O servidor cria um *miner* e é nessa altura que são criados os dois canais dos quais os *workers* vão ler e posteriormente são criados os *workers*, cada um executando numa *go routine*.

São criados dois tipos de *workers*, um cuja leitura principal é feita do canal *inLight*, para pedidos mais leves; e outro que lê principalmente do canal *inHeavy*, para pedidos mais pesados. O número de *workers* criados de cada tipo é fornecido como parâmetro no construtor do *miner*.

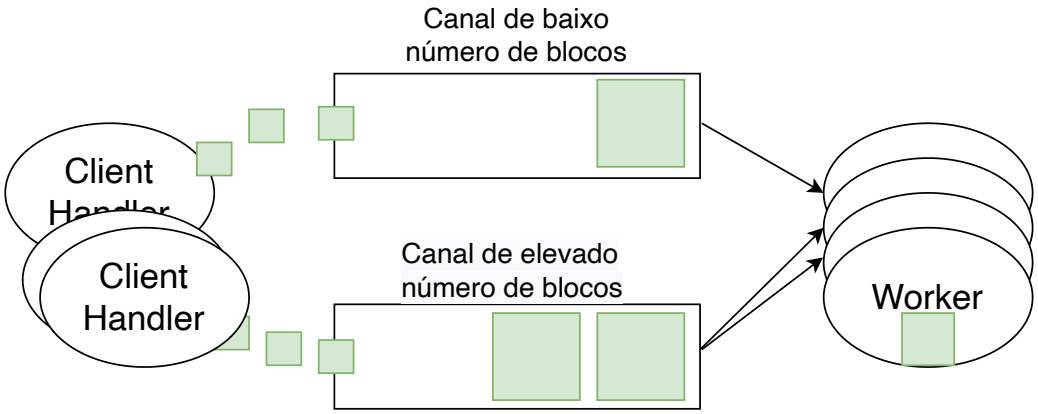

Fig. 1. Arquitectura da solução implementada

Listing 2
Código responsável por agregar as diferentes repostas dos *workers*

```
1 ...
2 max := <-respCh
3 var i uint64
4 for i = 0; i < blocks; i++ {
5   max = max.max(<-respCh)
6 }
7 ...
```

5 WORKER MANAGEMENT

A nossa implementação oferece uma abstracção semelhante a uma *pool* de *workers*, assim o trabalho é submetido numa *pool* em que diferentes *workers* estarão à escuta de trabalho. É de notar que existem dois canais onde o trabalho será submetido e, portanto, é como se existissem duas *pools* diferentes, mas algo flexíveis, pois cada *worker* tem uma *pool* preferencial definida por defeito, e apenas caso o canal respetivo esteja vazio, esse *worker* passa a estar disponível para executar trabalho do outro canal.

De forma a paralelizar o trabalho pelos diferentes *workers*, este será divido em blocos de tamanho menor. Estes blocos devem ter um tamanho algo significativo de forma a que os custos de *management* não sejam superiores aos custos de executar tal trabalho.

O pedido de trabalho recebido pelo *worker* possui um canal por onde deve ser enviada a sua resposta por parte do *worker*. Do outro lado desse canal encontra-se a *go routine* que lançou o trabalho, a aguardar pelos *hashes* dos diferentes blocos para os agregar e calcular o maior *hash* de entre eles (listagem 2).

Listing 3
Código responsável por limitar o número de *requests* feitas por um cliente

```
1
2 a := rate.NewLimiter(rate.Limit(RATE),BURST)
3 for {
4     if err := a.Wait(ctxt); err != nil {
5       fmt.Printf("%v\n", err)
6       return err
7     }
8
9     conn, err := ln.Accept()
10 ...
11 }
```

6 CLIENT RATE MANAGEMENT

A nossa implementação oferece a possibilidade de limitar o número de *requests* que o servidor serve. Para conseguir essa limitação, utilizamos o módulo presente na biblioteca de experiência do *golang* (*golang.org/x/time*). Dentro do *package time* existe uma implementação de um *rate limiter*, que permite limitar o número de operações do cliente aceites pelo servidor. Assim, a cada T segundos são libertados B *tokens*, em que cada *token* permite uma operação de um cliente. Este *rate limiter* é criado antes do servidor aceitar qualquer conexão, e é responsável por um potencial tempo de espera antes de aceitar cada conexão por parte de um cliente (listagem 3).

É preciso ter em atenção que este *rate limiter* oferece limite global para as operações de todos os clientes. Para oferecer limitação por cliente seria necessário criar uma abstracção de sessão, de forma a que cada cliente tivesse um *rate limiter* associado à sua sessão.

Listing 4
Código responsável por escolher o canal de onde um worker retira trabalho

```
1  select {
2    case <-ctxt.Done():
3      return
4    case req := <-inDefault:
5        computeHigherHash(req,id)
6    default:{
7        select {
8        case req := <-inOther:
9          computeHigherHash(req,id)
10       case req := <-inDefault:
11         computeHigherHash(req,id)
12       }
13     }
14   }
15 }
```

7 Fairness and responsiveness

A nossa implementação tem em consideração as propriedades de *fairness* e *responsiveness* de forma a diminuir o tempo de resposta média a cada cliente.

Para conseguir diminuir o tempo médio de espera de cada cliente a nossa solução implementa três estratégias: a) as *requests* de diferentes tamanhos serão divididas por dois canais; b) ambos os canais são limitados; c) cache para atender pedidos repetidos de forma imediata.

Na primeira estratégia, com a criação de dois canais, as *requests* são divididas por ambos, como se pode ver na figura 1. *Requests* divididas em mais que um determinado número de blocos serão consideradas como pesadas e inseridas no canal destinado para tal, caso contrário serão inseridas no canal de baixo número de blocos. Para garantir que ambos os canais são servidos com *fairness*, *workers* estão distribuídos pelos dois canais dando prioridade a um deles, ou seja, se um dos *workers* estiver atribuído para servir o canal de baixo número de blocos, este irá servir todas as *requests* desse canal antes de ir obter trabalho no canal de alto número de blocos. Na listagem 4 é possível ver o pedaço de código responsável por tal escolha do canal.

Na segunda estratégia, ao limitar o tamanho dos canais será criada uma escolha *fair* de quem insere por parte dos canais do *go*. Quer isto dizer que clientes com trabalhos com números de blocos muito diferentes entre si, apesar de poderem ser considerados como sendo da mesma categoria, leves ou pesados, conseguirão inserir trabalho nesse canal com a mesma probabilidade, com alguma intercalação entre si, pelo que trabalhos com menor número de blocos conseguem terminar sem necessitarem aguardar que o outro trabalho de maiores dimensões termine primeiro.

Por último, adicionamos cache para que *requests* previamente feitas sejam atendidas de forma imediata pela cache, assim é possível garantir que *requests* repetidas tenham latência quase idêntica ao tempo de RTT.

8 Trabalho Futuro

A nossa solução tira partido da cache para responder mais rapidamente a *requests* previamente efectuadas. Este sistema de cache poderia ser melhorado, pois apenas permite responder a *requests* totalmente idênticas. Assim, para ter uma solução mais completa, a cache deveria poder responder a pedidos para a mesma mensagem com *nonce* superior, apenas calculando o restante entre o *nonce* máximo e a resposta presente na cache.

Outra melhoria futura seria dar a possibilidade de calcular dinamicamente qual o tamanho máximo de um bloco de trabalho e qual o número de blocos em que uma *request* deveria ser considerada pesada, isto tudo de forma a reduzir o tempo de resposta do servidor.

9 Conclusão

Para este trabalho foi necessário criar um programa simplificado do processo de *mining* da *bitcoin*. Foi criado um cliente que efetua os pedidos de *hash* de uma determinada mensagem, e um servidor que calcula o maior *hash* para cada pedido. Dada a diferença na quantidade de trabalho de diferentes pedidos, é imperativo garantir alguma *fairness*, isto é, não faz sentido pedidos muito pequenos ficarem a aguardar que o servidor sirva pedidos que chegaram antes, mas cujo tamanho é muito superior. Através de algumas estratégias, como o uso de cache e o uso de canais limitados e especializados, foi possível contornar este problema. O sistema de *client rate management* também evita que o servidor receba muitos pedidos simultâneos, conseguindo assim tempo extra para responder aos pedidos pendentes.

References

[1] Haber, S., and Stornetta, W. S. How to Time-Stamp a Digital Document. *J. Cryptol. 3*, 2 (jan 1991), 99–111.
[2] Nakamoto, S. Bitcoin: A peer-to-peer electronic cash system, 2009.
[3] Tonhinho, B. Bitcoin-ish miner, 2020.