

# Public Ledger for Auctions

Segurança de Sistemas de Dados

Diogo Almeida, up201805400

Fábio Costa, up201604607

Sérgio Neto, up201407729

Maio  
2023

# 1 Arquitetura

O nosso sistema foi desenvolvido tendo em mente a arquitetura indicada em baixo com o objetivo de fornecer organização na nossa implementação. Ficamos assim com 3 interfaces diferentes que expõem as suas respectivas funcionalidades:

- AuctionApp vai expor os métodos responsáveis por manipulação e gerência de leilões e licitações;
- Blockchain vai expor os métodos responsáveis por manipulação e gerência da blockchain e de blocos na blockchain;
- Kademlia vai expor os métodos responsáveis pela gerência do estado da rede peer-to-peer e também pela propagação de informação.

Estas interfaces interagem umas com as outras de acordo com a imagem seguinte:

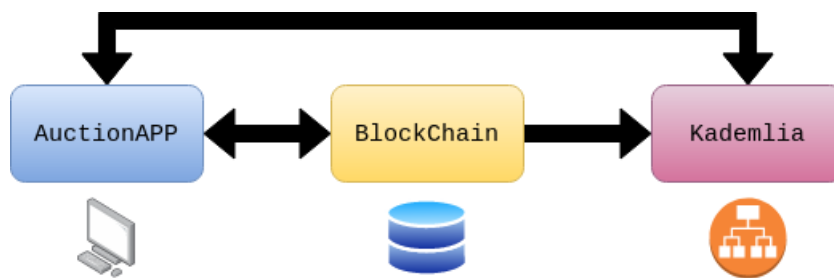


Figura 1: Arquitetura e interação do sistema

## 2 Desenho

O nosso sistema usa Proof of Work como mecanismo de consenso e todos os nós têm capacidade de validar transações e adicionar blocos à blockchain.

Todos os blocos e leilões são disseminados pela rede.

Divergências na blockchain vão ser resolvidas escolhendo a maior blockchain (ou a que cresce mais rápido) e os blocos da divergência resolvida são validados novamente, adicionados à blockchain maior e disseminados pela rede. Um nó subscrito a um determinado leilão vai notificar o user sempre que o estado desse leilão mudar, seja por uma nova licitação acontecer ou por o leilão terminar. Quando um leilão é iniciado ele tem uma duração fixa e estática que é definida por nós.

Quando o leilão chega ao fim, automaticamente é gerado um bloco com o estado final do leilão e este é disseminado na rede. Este comportamento é igual para todos os nós, ou seja, todos os nós que conhecerem o leilão vão

seguir esta diretiva.

De seguida vamos falar dos pormenores que achamos mais relevantes, mas tudo pode ser verificado no código enviado junto com este relatório ou neste repositório.

## 2.1 Segurança

Por desenho, a nossa implementação segue as seguintes regras para garantir algum grau autenticidade, integridade e dificultar que um elevado número de nós maliciosos aceda à rede:

- O processo de acesso à rede inicialmente exige que o nó mine um bloco, se o bloco for validado por um BootStrap, vai ser devolvido ao nó o seu **identificador único** na rede;
- Mensagens que circulam na rede, são acompanhadas por informação como:
  - Checksum - SHA2 do corpo da mensagem;
  - Assinatura - SHA2(PrivKey\_Remetente,Checksum);
  - PubKey - Chave pública do remetente.

de maneira a permitir o destinatário verificar a integridade da mensagem e a identidade do remetente;

- Todas as transações, leilões e licitações são adicionadas à blockchain.
- Blocos minerados são disseminados para toda a rede.

Com a implementação destas regras, consideramos que o nosso sistema é mais resistente a ataques Eclipse e Sybil.

## 2.2 AuctionAPP

Nesta estrutura é onde manipulamos/gerimos os leilões que vamos receber/criar e segue a seguinte estrutura:

- HashMap dos leilões inicializados;
- HashMap dos leilões recebidos pela rede;
- ArrayList de (identificador) leilões subscritos;

Para conseguir terminar os leilões de maneira automática, quando um leilão é adicionado, é lançada uma thread que acorda no tempo em que o leilão acabar e começa a lógica mencionada acima. Aqui expomos métodos como:

- Criar leilão;
- Listar todos os leilões;
- Licitar num leilão;
- etc..

## 2.3 BlockChain

Nesta estrutura é onde toda a lógica e gestão da blockchain se encontra. É composta por:

- Lista de Listas de Blocos, onde guardamos uma ou mais (no caso de divergências) blockchains;
- Lista de Blocos por confirmar, onde guardamos blocos que não conseguimos adicionar a nenhuma blockchain (o bloco identificado pela a sua hash anterior não existe).

Já os nossos blocos seguem a seguinte estrutura:

- Nonce, apenas um número;
- TimeStamp, gerado quando o método para criar o bloco é invocado;
- Hash, Cifra do conteúdo do bloco com  $N$  zeros no início;
- Hash do bloco anterior;
- Transaction, transação onde está toda a informação referente a um leilão/licitação.

Um pormenor importante a referir é que a estrutura Transaction inclui uma Hash do seu corpo, uma Assinatura, que é a Hash assinada com a chave privada de quem lançou a transação, e a chave pública (para validar a assinatura) de maneira a garantir autenticidade.

### 2.3.1 Blocos por Confirmar

Sempre que recebemos um bloco novo, nós verificamos se a sua hash está contida em algum dos blocos na lista de blocos por confirmar e se existir tentamos resolver os blocos. Esta maneira não garante a resolução do conflito, por isso, nós pensamos em fazer o seguinte:

Quando um bloco ficar por confirmar, vamos fazer um pedido (iterativo) à rede, parecido com o findNode usado no Kademlia, que eventualmente retorna o bloco que queremos. Este pedido seria feito de forma assíncrona. No entanto acabamos por não ter tempo de integrar este mecanismo (apesar de termos a função findBlock implementada) e os blocos simplesmente são ignorados.

### 2.3.2 Divergências

A nossa implementação de blockchain suporta divergências e resolve as mesmas. Quando existem blockchains divergentes a nossa tentativa de as resolver segue a seguinte lógica:

$$\begin{aligned} BiggestSize &= Maxsize(bchain1, bchain2, bchain3, \dots) \\ Ahead &= Constant \end{aligned}$$

Se  $BiggestSize - Size(blockChainXpto) > Ahead$  retiramos a lista  $blockChainXpto$  da nossa lista de blockchains e voltamos a minar os blocos dessa lista que não existem na maior blockchain. Estes blocos vão ser sempre adicionados à maior blockchain.

### 2.3.3 Fork

Um fork acontece quando um novo bloco, para adicionar à blockchain, não aponta para o ultimo bloco adicionado à blockchain, mas sim para outro bloco que foi inserido antes, isto é, quando um bloco novo aponta para um bloco que já tem predecessores acontece um fork.

No entanto há outra condição que verificamos para fazer fork da blockchain. Se o index onde acontece o conflito for  $x$ , só fazemos o fork se a seguinte condição for verificada:

$$BiggestSize - x > Ahead$$

A figura abaixo descreve o mecanismo que implementamos para lidar com esta situação:

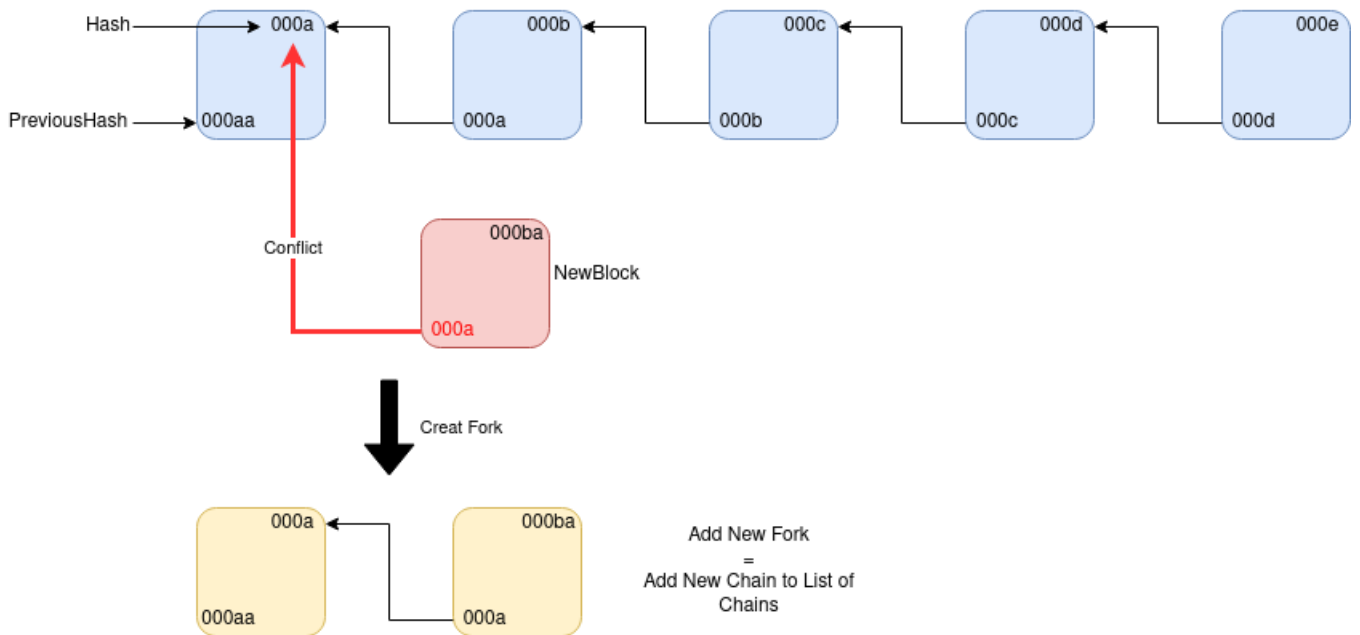


Figura 2: Criação de novos Forks/blockchain divergente

Na nossa implementação, ao acontecer um fork, criamos uma nova lista de blocos que vai ter os blocos até ao ponto de conflito e adicionamos o novo bloco, depois disto adicionamos esta nova lista à nossa lista de listas.

## 2.4 Kademlia

Como na implementação original do Kademlia[1] a nossa interface expõe os métodos Ping, Store, Find\_Node e Find\_Value. Para além destes criamos mais dois métodos:

- challenge - Inicia o processo de desafio para um nó que se tenta juntar à rede;
- join - Inicia o processo de adesão à rede.

Para facilitar o trabalho de networking, fizemos uso da ferramenta gRPC, e os nossos rpcs são os seguintes:

```
service Kademlia {  
  rpc ping(NodeInfo) returns (NodeInfo);  
  rpc store (StoreRequest) returns (StoreResponse);  
  rpc findNode (FindNodeRequest) returns (stream FindNodeResponse);  
  rpc findValue (FindValueRequest) returns (stream FindValueResponse);  
  rpc challenge (ChallengeRequest) returns (ChallengeResponse);  
  rpc join (JoinRequest) returns (JoinResponse);  
}
```

Figura 3: Remote Procedure Calls do Kademlia

Além disto temos um mecanismo de disseminação. Ao receber um store quer seja de um bloco ou um leilão, nós verificamos se o respetivo objeto já existe localmente, se existir o nó não faz nada, mas se não existir, trata de fazer a adição do respetivo objeto e depois faz um storeResquest garantindo assim que a informação eventualmente chega a toda a rede.

## 3 Trabalho Futuro

Existem várias alterações que podem ser feitas à nossa implementação, principalmente aquelas que visam melhorar a eficiência de espaço como por exemplo:

- Alterar nossa blockchain para fazer uso de Merkle trees, o uso de espaço da blockchain melhora para complexidade  $O(n)[2][3][4]$ ;
- Em vez de usar uma Lista de Listas para guardar as várias blockchains que podem surgir de conflitos, podemos usar uma Árvore, isto iria reduzir imenso a utilização de memória da aplicação;

Melhorar o mecanismo de resolução de um fork, neste momento todos os blocos que têm de ser validados novamente, são validados no próprio nó onde o conflito se tenta resolver, isto faz com que o nó fique demasiado tempo apenas a validar/minerar.

Ao receber blocos cujo o bloco anterior não existe, devemos fazer um pedido assíncrono find desse bloco anterior de maneira a eventualmente receber todos os blocos que são necessários para resolver este conflito.

### 3.1 Proof-of-stake

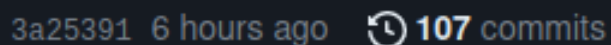
Infelizmente não conseguimos implementar o mecanismo de consenso baseado em *Delegated Proof of Stake*, conceptualmente a ideia não é difícil de perceber e apesar de conceptualmente entendermos o mecanismo, praticamente, não conseguimos formalizar uma implementação concreta. Quanto a este tema concordamos que permanecem várias duvidas mas seria algo interessante de adicionar ao sistema que temos neste momento.

## 4 Conclusão

Em conclusão, o projeto foi desenvolvido com 4 questões centrais/mente. A descentralização e a confiança dos dados entre nós. A segurança criptográfica, que para além de ser uma extensão do problema anterior, também permite garantir a autenticidade de cada transação. A escalabilidade e eficiência foi algo ponderado, mas não foi conseguido.

Optamos por realizar este trabalho com Java, dada a fluência que temos com a língua de programação e mesmo assim devido a problemas de sincronização e conversão achamos que conseguimos extrair ainda mais conhecimento sobre esta linguagem.

Apesar de a implementação não ser a mais eficiente, sentimos que o trabalho foi bem conseguido e que atingimos os objetivos a que nos propusemos no início do semestre. Mesmo depois de várias alturas de frustração em que, por exemplo, passamos (7) horas a tentar resolver um dead lock para no final tudo se resolver com uma linha de código, podemos concluir, como mostra o nº de commits no nosso git4



que foi um bom trabalho de se fazer. **Obrigado pelo desafio! :)**

## Referências

- [1] P. Maymounkov e D. Mazières, «Kademlia: A Peer-to-Peer Information System Based on the XOR Metric,» en, em *Peer-to-Peer Systems*, P. Druschel, F. Kaashoek e A. Rowstron, eds., sér. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2002, pp. 53–65, ISBN: 978-3-540-45748-0. DOI: 10.1007/3-540-45748-8\_5.
- [2] *Merkle Tree [Explained]*, en, jun. de 2021. URL: <https://iq.opengenus.org/merkle-tree/> (acedido em 21/05/2023).
- [3] *Merkle Tree in Blockchain*, en. URL: <https://www.topcoder.com/thrive/articles/merkle-tree-in-blockchain> (acedido em 21/05/2023).
- [4] *Merkle Tree — Brilliant Math & Science Wiki*, en-us. URL: <https://brilliant.org/wiki/merkle-tree/> (acedido em 21/05/2023).