Public Ledger for Auctions – Segurança de Sistemas e Dados

Maria Sousa Carreira up202408787
Matilde Isabel da Silva Simões up202108782
Ricardo Araújo Amorim up202107843
Faculdade de Ciências, Universidade do Porto

I. Introdução

Num problema descentralizado, a utilização de tecnologias seguras e fiáveis para gerir transações entre múltiplas entidades tornou-se essencial. A implementação que vai ser apresentada propõe uma plataforma distribuída para leilões, baseada em mecanismos de *blockchain* e redes *peer-to-peer* (P2P), com o objetivo de assegurar integridade, autenticidade e transparência ao longo de todo o ciclo de vida das transações.

Este relatório descreve a arquitetura do sistema, os principais componentes técnicos e as decisões tomadas em termos de segurança, consistência e robustez da rede.

II. Estrutura do Projeto

Este projeto foi desenvolvido em `Java` e segue a estrutura apresentada na Fig.1.

```
.
└── java
    └── main
        ├── auctions
        ├── blockchain
        │   ├── Blockchain.java
        │   ├── BlockHeader.java
        │   ├── Block.java
        │   ├── Chain.java
        │   ├── MerkleTree.java
        │   ├── PoW.java
        │   └── Transaction.java
        ├── kademlia
        │   ├── Bucket.java
        │   ├── Communication.java
        │   ├── MessageHandler.java
        │   ├── Node.java
        │   ├── PeerNode.java
        │   └── RoutingTable.java
        ├── main.java
        └── Utils.java
```

Figura 1. Estrutura do projeto.

Para começar, o diretório `blockchain/` inclui todas as classes que caracterizam a construção da *blockchain* e outras operações constituintes da etapa *distributed ledger* (Sec.IV). O diretório `kademlia/` corresponde à implementação do mecanismo P2P (Sec.V), incluindo todas as classes necessárias à construção de nós e à comunicação entre si. Para além disso, o ficheiro `main.java` implementa o menu visualizado por cada nó, incluindo a maioria da lógica do mecanismo de leilões (Sec.VI). Por fim, a classe `Utils` representam os métodos utilizados para executar algumas funções auxiliares.

III. Exemplo de Fluxo de Execução

Foi desenvolvido um vídeo demonstrativo que apresenta um fluxo de execução da solução implementada. *Link* do vídeo:

https://www.youtube.com/watch?v=VD3HqWd1Iuk

A solução foi executada em quatro **máquinas virtuais (VMs)** na Google Cloud Platform (GCP), distribuídas entre diferentes zonas de disponibilidade, com cada VM atuando como um nó independente da rede. Alternativamente, o sistema também pode ser executado **localmente**, bastando apenas alterar os endereços IP dos nós de bootstrap para apontarem para o IP local (`localhost`) ou para os IPs da rede interna local, permitindo a simulação completa em um único computador ou em rede privada.

IV. *Distributed Ledger*

O ***distributed ledger*** implementado neste projeto tem como objetivo garantir um **sistema descentralizado**, imutável e que permita verificar as transações dos leilões. A sua arquitetura combina técnicas de ***blockchain***, ***merkle tree***, **assinatura digital** e ***proof-of-work*** (PoW), garantindo segurança e resiliência do sistema.

A. *Estrutura dos Blocos*

Cada bloco da *blockchain* é constituído por duas partes principais: uma lista de **transações** e um **cabeçalho**, representado pela estrutura `BlockHeader`.

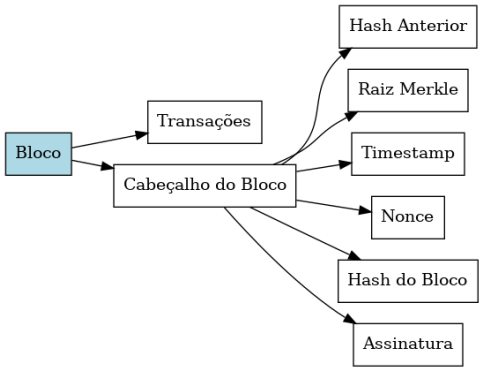

Figura 2. Estrutura dos Blocos.

As transações incluem informações como o **nó criador**, a **descrição** da operação, o **número do leilão** a que corresponde, o **tipo** de transação, o ***timestamp*** de criação e ainda a **assinatura digital** gerada com a chave privada do emissor.

Já o cabeçalho do bloco agrega toda a informação essencial que assegura a integridade e validade dentro da cadeia (`Chain`), nomeadamente, a ***hash*** do **bloco anterior** (que garante a ligação entre blocos), a ***merkle root*** [1] (as *hashes* das transações são agrupadas recursivamente até restar uma única raiz), um ***nonce*** (usado no processo de PoW), um ***timestamp*** de criação, a ***hash*** do próprio bloco (resultante da *hash* do cabeçalho) e a assinatura do cabeçalho (criada com a chave privada do *node* que minerou).

Esta separação entre as transações e esta informação permite validar de forma eficiente a autenticidade, ordem e integridade de cada bloco.

B. Proof-of-Work

Com a formação de um novo bloco, é realizado o mecanismo de PoW [1], implementado na função `PoW.miner()`. Este processo consiste em encontrar um valor do `nonce` tal que o `hash` **SHA-256** do cabeçalho do bloco **comece** com um **número específico de zeros**, definido pelo nível de dificuldade. No sistema implementado, a dificuldade é configurada para exigir um prefixo com **dois zeros**.

A função itera sucessivamente sobre valores aleatórios de `nonce`, gerando a `hash` do cabeçalho até encontrar um que satisfaça o critério de dificuldade. Esta operação é computacionalmente intensiva e não pode ser predita, o que garante a segurança do sistema ao dificultar a produção arbitrária de blocos válidos. Uma vez encontrado uma `hash` válida, este é atribuído ao cabeçalho do bloco e o bloco torna-se elegível para ser adicionado à cadeia.

Este processo assegura que o esforço computacional investido na criação de blocos **impede ataques** e incentiva a manutenção de uma **cadeia de blocos válida e verificável**.

C. Gestão de Múltiplas Cadeias

São suportadas **múltiplas *chains*** que, em conjunto, constituem a `Blockchain`. A criação da *blockchain* inicia-se com o ***genesis block*** [1], que é construído sem transações e sem `prevHash` e cuja `merkleRoot` é calculada com base numa lista de transações vazia. Este bloco é o primeiro da primeira cadeia e define a origem comum de todas as bifurcações subsequentes.

A *blockchain* é guardada em ficheiros `JSON`, utilizando a biblioteca `GSON`, permitindo **guardar** e **recuperar** o estado completo da *blockchain* entre execuções. Os ficheiros antigos são eliminados e substituídos pelos novos, assegurando a persistência consistente do estado da rede.

V. P2P SEGURO

A. Arquitetura do nó

A camada P2P implementada é baseada no protocolo ***S/Kademlia*** [2], uma versão segura do sistema de roteamento *DHT Kademlia*. Cada `Node` da rede é identificado por um `nodeId`, derivado da assinatura **SHA-256** da sua chave pública RSA. Essa identidade criptográfica garante unicidade e evita **ataques *Eclipse*** por geração arbitrária de IDs [2].

Os nós mantêm uma `RoutingTable` com **160 *buckets***, cada um (i) responsável por um intervalo de nós cujo ID varia de 2^i a 2^{i+1} [2]. Cada entrada armazena o quádruplo **<IP, porta UDP, ID, *timeAlive*>**, de um nó vizinho, onde `timeAlive` corresponde ao timestamp que indica o momento em que o nó entrou na rede. Esse quádruplo é atualizado cada vez que um outro nó desconhecido contacta o próprio. Caso um `Bucket` fique **cheio**, é **removido o nó mais recente**, prevenindo, mais uma vez, **ataques *Eclipse*** [2].

Para além disso, cada nó também mantém uma **cópia local da *blockchain***, um ***timestamp*** (`timeAlive`) que indica o tempo de vida total do nó e uma `transactionPool`, a **lista de transações** criadas pelo nó.

B. Comunicação Segura

Toda comunicação entre nós é encapsulada na classe `Communication`, que inclui: o **tipo** da mensagem, o nó **remetente** e o **destinatário**, a **informação** e a **assinatura digital** com algoritmo `SHA256withRSA` (das mensagens mais sensíveis como as do tipo `FIND_VALUE` e `STORE`).

As mensagens são assinadas com a chave privada do remetente e a verificação é feita com base na chave pública carregada de um **ficheiro local** mantido por cada nó. Este mecanismo garante **integridade**, **autenticidade** e **não-repúdio**.

C. Participação na Rede – `PeerNode.joinNetwork()`

O processo da primeira entrada na rede segue os seguintes passos (Fig. 3):

1) O nó gera e armazena uma solução válida para o **desafio** baseado em PoW, uma proposta para evitar **ataques *Sybil*** [2].
2) Envia uma mensagem `CHALLENGE` com a solução do desafio a um nó *bootstrap* (neste caso a qualquer nó da rede).
3) Assim que é aceite na rede, inicia a **descoberta de vizinhos** via `FIND_NODE`, recursivamente.
4) Cria uma **cópia da *blockchain*** do nó *bootstrap* com `FIND_BLOCKCHAIN`.

Caso não seja a primeira entrada do nó na rede, existe a **opção de reentrar** com a solução do desafio armazenada e, se as tiver, carregar uma *routing table* local e, também, uma cópia da *blockchain* local.

Após este processo, cada nó executa uma **verificação da consistência** da sua *blockchain* com os restantes nós da rede:

1) O nó seleciona até k vizinhos da sua *routing table*.
2) Envia uma mensagem do tipo `RECENT_BLOCKS_REQUEST`, solicitando uma lista de blocos recentes de cada k-nó vizinho.
3) O nó calcula a frequência de cada *hash* entre os blocos recebidos, identificando a maioria e determinando quais os blocos com maior probabilidade de serem **legítimos**.

Se o nó conseguir ligar a sua *blockchain* aos blocos recebidos, considera-se a **cadeia consistente**. Caso contrário, ele tenta **reconstruir a ligação** entre a cadeia local e a da maioria, seguindo a cadeia de `prevHash` dos blocos recebidos até uma **profundidade limite**. Se for possível encontrar essa ligação, os blocos são integrados na cadeia local, se a ligação não for encontrada, o nó solicita e substitui toda a *blockchain* local com a da maioria, utilizando a mensagem `FIND_BLOCKCHAIN`. Este método garante que a ***blockchain* local seja legítima com grande probabilidade**.

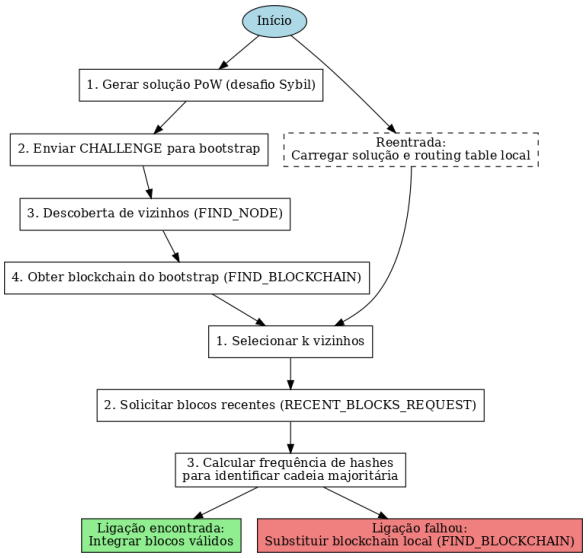

Figura 3. Representação resumida do *join network*.

D. Propagação de Blocos

A camada P2P suporta a **propagação de blocos** e a **recuperação de blocos órfãos** com `STORE` e `FIND_VALUE`, respetivamente. Assim que um nó minera um bloco, ele deve propagá-lo com `STORE` para a rede, enviando-o a todos os nós da sua *routing table*. Se um nó receber um bloco orfão, ou seja, que não se ligue diretamente à sua *blockchain*, deve enviar um `FIND_VALUE` recursivo de modo a encontrar os seu predecessores. No entanto, este processo **não dura mais do que 3 iterações**, o que evita *loops* infinitos em busca de blocos maliciosos e grandes **desperdícios de recursos computacionais**.

Para além disso, foi implementada uma função `Blockchain.trimFork()` que é responsável pelo **tratamento de *forks***. Um *fork* acontece quando dois nós enviam uma mensagem `STORE` com a mesma `prevHash`. Neste momento, o nó que recebeu estes blocos cria uma nova `Chain` para o bloco que recebeu em último lugar, na **mesma altura** (`blockchainHeight`) que o outro. A partir daí, cada nó obedece à **regra** de continuar a *chain* cujo primeiro bloco tem um ***timestamp* mais recente**. Após cada `STORE` recebido é feita a verificação de cada *fork*, assim que uma dessas *chains* prevalece durante **mais que dois blocos adicionais**, o *fork* é decidido e as *chains* não escolhidas são **eliminadas**.

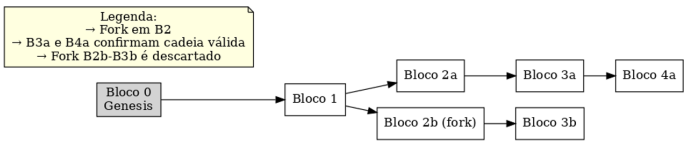

Figura 4. Exemplo de um *fork*.

VI. Mecanismo de leilões

O sistema implementa um **mecanismo de leilão** do tipo ***english auction***. Este modelo é assíncrono, operando sobre a **rede P2P descentralizada**, onde nós –vendedores e compradores – interagem através de transações guardadas na *blockchain*.

A. Modelo de Leilão e Tipos de Transação

Os leilões são estruturados em **quatro fases** principais, representadas por transações distintas, todas **assinadas** com a chave privada do nó criador da transação. Os tipos de transação definidos no sistema são:

- `CREATE_AUCTION`: Criada por um vendedor para definir um novo leilão, identificando o item e gerando um identificador único.
- `START_AUCTION`: Indica o início oficial da aceitação de licitações para o leilão criado.
- `BID`: Enviada por um comprador, contendo o valor da sua proposta.
- `CLOSE_AUCTION`: Criada pelo vendedor, indicando o encerramento do leilão.

B. Armazenamento na Blockchain e Propagação

As transações são agrupadas num ***transaction pool*** local. Quando esse conjunto atinge um **tamanho** ou **tempo limite** (`TRANS_POOL_LIMIT_LENGTH` e `TRANS_POOL_LIMIT_TIME`), é criado um **novo bloco** com as transações pendentes. Esse bloco é minerado e posteriormente propagado pela rede através de mensagens `STORE`, garantindo a integridade com a *merkle root* e assinaturas criptográficas validadas.

Ao ser recebido, **cada bloco é verificado** (assinatura, *merkle root* e encadeamento via `prevHash`). Se o bloco for válido, é armazenado na cadeia principal; caso contrário, pode ser tratado como órfão até que os blocos anteriores sejam recuperados via `FIND_VALUE`.

C. Determinar o Vencedor do Leilão

Após o envio da transação do tipo `CLOSE_AUCTION`, o sistema inicia uma **contagem de blocos** até à **confirmação efetiva do encerramento**, utilizando o parâmetro `BLOCK_CHAIN_LIMIT`. A determinação do vencedor não é feita imediatamente, mas sim apenas quando a *blockchain* atinge uma altura correspondente a **dois blocos** após o bloco que contém a transação de fecho do leilão. Esta abordagem garante que todas as propostas válidas (`BID`) propagadas pouco antes ou em simultâneo com o encerramento tenham tempo suficiente para serem incluídas na cadeia principal.

O processo de seleção percorre todas as transações do tipo `BID` associadas ao identificador `auctionNumber` do leilão encerrado, considerando apenas aquelas com `timestamp` **anterior ou igual ao bloco de fecho**. A proposta com o valor mais elevado é então considerada vencedora.

Este mecanismo garante justiça, autenticidade e transparência no processo de leilão, mitigando situações de perda de propostas devido à possível latência da propagação.

VII. Mecanismo de *Fault Injection*

Na generalidade, foram considerados dois tipos de injeções maliciosas: quando um **nó se desconecta inesperadamente** e quando um nó, potencialmente malicioso, envia para a rede um **bloco manipulado**. A solução para o primeiro caso já foi abordada anteriormente no **processo de participação na rede**

(Sec.V-C). Esta solução permite a qualquer nó voltar muito facilmente à rede, caso fique indisponível.

Para testar a robustez e a resiliência da rede no caso de um nó injetar um bloco malicioso, foi implementado o `HACKER MENU`. Com esta funcionalidade, um nó pode **manipular qualquer valor de um bloco**, incluindo a sua assinatura, a sua *merkle root*, transações e a sua *hash* ou *previous hash*, e propagá-lo para a rede com uma **mensagem igualmente manipulada**. Neste caso, implementaram-se vários mecanismos para combater este problema: **verificação da assinatura** de mensagens do tipo `STORE` e do *block header*, **cálculo da *merkle root***, **assinatura de todas as transações** e o **tratamento correto de blocos orfãos** (Sec.V-D).

VIII. *Proof-of-Reputation*

É possível **evoluir o modelo atual** para um mais robusto e **meritocrático**, inspirado na abordagem de ***Proof-of-Reputation*** [3]. Este mecanismo atribui o poder de decisão aos nós com **melhor histórico de comportamento** e contribuição, em vez de depender unicamente de poder computacional ou ordem de chegada das mensagens.

A. Motivação e Vantagens

Ao contrário dos mecanismos como PoW, a reputação não pode ser adquirida subitamente. Em vez disso, é **acumulada ao longo do tempo** com base na participação regular e honesta no sistema, **penalizando comportamentos maliciosos**. Este modelo reduz significativamente o risco de ataques como ***flash attacks*** e favorece a **estabilidade** e **previsibilidade** a longo prazo [3].

B. Modificações Necessárias no Sistema Atual

Para suportar *Proof-of-Reputation* no nosso sistema, seria necessário introduzir diversas alterações:

- **Tipos Fixos de Nós:** Atualmente, qualquer nó pode atuar como *bootstrap*, *miner* ou *normal*. Neste modelo, deveríamos definir um tipo único de nó com responsabilidades uniformes, de forma a permitir uma avaliação comparável do desempenho ao longo do tempo.
- **Sistema de Recompensas:** A atribuição de reputação deveria estar associada a ações verificáveis, como o envio válido de blocos, sendo acumulada de forma transparente ao longo do tempo. O artigo [3] identifica como limitação o facto de novos participantes entrarem com reputação nula, mas desaconselha a atribuição inicial de reputação como solução, uma vez que isso poderia facilitar ataques. Em vez disso, propõe limitar a taxa de crescimento da reputação (*voting power*) ao longo do tempo, de forma a impedir ganhos rápidos de influência por parte de novos nós, mesmo que possuam elevado poder computacional. No nosso sistema, uma abordagem alternativa — mas que exigiria mecanismos de controlo adicionais — seria introduzir recompensas iniciais cuidadosamente geridas para os primeiros participantes legítimos. Em ambos os casos, seria importante estabelecer um intervalo mínimo entre recompensas sucessivas, evitando a acumulação rápida e artificial de reputação.
- **Alterações às Classes** `Node`, `Blockchain` e `PeerNode`:
  - Na classe `Node`, seria necessário incluir um atributo e mecanismos de atualização da reputação associada ao nó, bem como métodos para aceder à mesma.
  - A `Blockchain` teria de permitir a validação de reputações com base no histórico da cadeia.
  - A classe `PeerNode` teria de ser adaptada para, ao tomar decisões de consenso (como aceitar blocos ou transações), ponderar a reputação dos emissores, atribuindo maior confiança àqueles com maior reputação.
- **Grupo de Consenso:** Apenas os nós com maior reputação formariam o grupo de consenso ativo, tendo maior peso nas decisões. Este grupo seria atualizado dinamicamente com base na reputação acumulada até ao momento [3].

C. Integração com Sistemas BFT

Além disso, como discutido no artigo [4], seria possível integrar este modelo com **mecanismos de consenso** tolerantes a falhas bizantinas (*Byzantine Fault Tolerance* - BFT), utilizando reputação como critério de seleção e ponderação nos algoritmos de consenso. Isto permitiria obter uma forte consistência e maior resiliência a ataques.

Em resumo, a transição para um modelo de PoR implicaria um maior controlo sobre os papéis dos nós, uma gestão mais sofisticada do histórico de interações e um sistema de incentivos claros. No entanto, esta mudança traria benefícios significativos em termos de segurança e robustez, adaptando o sistema para aplicações reais e sustentáveis em contextos distribuídos.

IX. Conclusão

O sistema desenvolvido demonstra como é possível criar uma plataforma de leilões descentralizada, segura e verificável, utilizando conceitos de *blockchain*, PoW e redes P2P com verificação criptográfica.

Importa salientar que todos os requisitos propostos para o projeto foram integralmente implementados e o mecanismo de PoR foi explicada como poderia ser implementada, a partir do sistema atual que foi desenvolvido.

A arquitetura desenvolvida apresenta um elevado grau de flexibilidade e escalabilidade, permitindo que o sistema seja facilmente adaptado a outras aplicações distribuídas. Com os fundamentos bem definidos, o projeto pode evoluir para incorporar novos mecanismos de confiança, privacidade e tolerância a falhas, respondendo às exigências de cenários ainda mais complexos.

Referências

[1] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *SSRN Electronic Journal*, 8 2008.

[2] Ingmar Baumgart and Sebastian Mies. S/kademlia: A practicable approach towards secure key-based routing. *Proceedings of the International Conference on Parallel and Distributed Systems - ICPADS*, 2, 2007.

[3] Jiangshan Yu, David Kozhaya, Jeremie Decouchant, and Paulo Esteves-Verissimo. Repucoin: Your reputation is your power. *IEEE Transactions on Computers*, 68:1225–1237, 8 2019.

[4] Nuno Neto, Rolando Martins, and Luís Veiga. Atlas, a modular and efficient open-source bft framework. *Journal of Systems and Software*, 222:112317, 4 2025.