Projeto SSD

1st Rui Santos
DCC
FCUP
Porto, Portugal
up202109728@up.pt

2nd Ricardo Sá
DCC
FCUP
Porto, Portugal
up202408644@up.pt

***Abstract*—This report presents the design and implementation of a decentralized system that manages a public ledger for auctions. The system integrates a blockchain that suports Proof-of-Work consensus mechanisms, a secure peer-to-peer (P2P) network based on S/Kademlia, and a simple auction module for managing bids and sales. All communication was implemented using basic socket programming, ensuring full control over data flow and security. To validate the system's resilience, fault injection mechanisms were introduced. Key assumptions regarding the network and blockchain structure were made. Despite the challenges encountered, the system demonstrates the core functionalities of a secure, decentralized auction platform, paving the way for future improvements such as full Proof-of-Reputation integration and cloud-based testing.**

I. Introduction

This report focuses on our design and implementation of a distributed public ledger for an auction system. Unlike traditional blockchains such as Bitcoin or Ethereum, our system is developed with the primary goal of securely managing auctions in a fully distributed and non-permissioned environment. The entire project was developed in Java, relying solely on the language's standard libraries.

The work is structured into three major components: the secure distributed ledger, a peer-to-peer communication layer, and a simple auction mechanism. The blockchain supports Proof-of-Work (PoW) consensus algorithms (Proof-of-Reputation was not implemented, but we describe how that could be possibly done). The P2P layer is built "on top" of S/Kademlia, providing resilience against Sybil and Eclipse attacks. Finally, the auction system implements a single-attribute English auction model, with transactions being securely broadcast and permanently stored in the blockchain.

Throughout the development, particular attention was given to ensuring system security, scalability, and fault tolerance. A fault injection mechanism was also incorporated to demonstrate the system's resiliency under node failures.

II. General Architecture

The diagram in Fig. 5 represents the general architecture of a peer in our implementation of a distributed ledger system. On the left side, clients and a P2P network interact with the peer by sending various operations. Each incoming operation is handled by a dedicated handler thread, ensuring parallel and efficient processing of multiple simultaneous requests.

Inside the peer, the server module is responsible for managing critical components such as the blockchain, transaction pool, miner, and Kademlia-based storage. The blockchain and transaction pool store the ledger and pending transactions, while the miner performs Proof-of-Work computations to validate new blocks. Kademlia is used to organize and retrieve data efficiently in the distributed network.

Storage is a key element linking several components, including Kademlia and the routing table, which keeps track of the peer's known nodes. The server also manages a public key that ensures secure identification and cryptographic operations within the network. This architecture allows the peer to function independently and participate fully in the decentralized environment.

In the following sections we will be disusing in more detail the main components of the general architecture.

III. Blockchain

In our system, the blockchain was implemented as an immutable and ordered list of blocks, each cryptographically linked to its predecessor. This chain of blocks ensures the integrity, transparency, and security of the recorded auction transactions. Each block encapsulates vital information that enables the validation of the chain and supports consensus mechanisms Proof-of-Work (PoW).

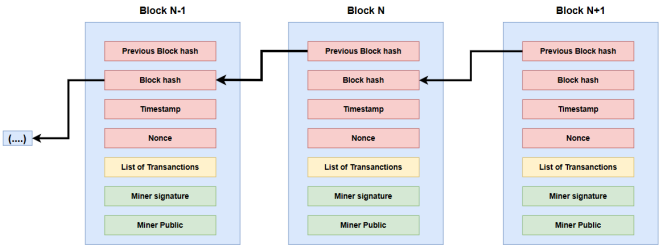

Fig. 1: Blockchain example & Block structure

As depicted in Fig. 1, each block contains the following fields:

- **Previous Block Hash**: A hash of the preceding block, ensuring the immutability and integrity of the chain.
- **Block Hash**: The unique hash of the current block (is composed by the *SHA256* of the concatenation of: previous block Hash, timestamp, nonce and transactions).

- **Timestamp**: A record of the exact moment when the block was created, providing a chronological order of transactions.
- **Nonce**: A number that miners adjust to satisfy the difficulty condition in Proof-of-Work.
- **List of Transactions**: A set of auction-related operations (such as bids or auction creations) signed with public key cryptography.
- **Miner Signature**: A digital signature generated by the miner to authenticate the block's creation.
- **Miner Public Key**: The miner's public key, which can be used to verify the miner's signature.

Blocks are linked through their hashes, forming a secure chain. Altering any single block would necessitate recomputing the hashes of all subsequent blocks, rendering such attacks computationally infeasible. Furthermore, as each block is signed with the miner's signature, signature verification would detect any tampering, causing blockchain validation to fail.

The use of a blockchain in this project ensures the decentralization, transparency, and immutability of auction transactions. By recording all operations in a distributed ledger, we guarantee that no single entity can tamper with bids or auction results, providing trust among participants without requiring a central authority.

A. Proof-of-Work

Proof-of-Work (PoW) is the original consensus mechanism introduced by Bitcoin [2]. It requires participants (miners) to solve complex cryptographic puzzles in order to validate new blocks. This process demands significant computational power, making it highly secure against attacks, as altering the ledger would require an impractical amount of resources. In our system, PoW is implemented to ensure that auction transactions are added to the blockchain in a decentralized and tamper-resistant manner. For our project we only managed to implement this consensus mechanism.

This was the only consensus mechanism we used in our implementation.

B. Proof-of-Reputation

Proof-of-Reputation (PoR) is another consensus mechanism that is designed to leverage the historical behavior and trustworthiness of participants in order to secure a distributed network. In contrast to traditional models like Proof-of-Work (PoW), Proof-of-Reputation evaluates a node's past actions, interactions, and contributions to determine its influence and authority in the system.

To integrate Proof-of-Reputation into our existing PoW-based system, we would have to assign each node a dynamic reputation score that would be derived from factors like: successful block validations, uptime and correct message handling. This score should be present in every block header or SecureMessage. During consensus, reputation would be combined with PoW, for example by reducing the effective mining difficulty or prioritizing proposals from higher-reputation nodes. After each block is accepted, a nodes reputations would be adjusted upward for honest behavior and downward for misbehavior. This hybrid approach would preserve PoW's security guarantees while providing greater influence to well-behaved peers.

IV. Kademlia & S/Kademlia

To start, we implemented the basic Kademlia protocol as originally described in [1]. Building on top of this foundation, we adapted our implementation to mitigate Sybil and Eclipse attacks, thus transitioning to S/Kademlia. The key modification required for this transition was the creation of a new **SecureMessage** class (illustrated in Fig. 2). This class ensures the authenticity and integrity of messages exchanged between nodes by incorporating cryptographic signatures. We integrated SecureMessage into all Kademlia operations, as well as other critical interactions where message security was deemed necessary. This enhancement reinforces trust among peers and provides the necessary guarantees to support a secure and resilient peer-to-peer network.

SecureMessage

```
private final String command;

private final Object payload;

private final PublicKey senderPublickKey;

private final byte[] signature;

private byte[] signPayload(PrivateKey privateKey){}

public boolean verifySignature() {}
```

Fig. 2: SecureMessage.java contents (without constructor)

A. Node structure

As illustrated in Fig. 3, a Kademlia/S-Kademlia node is composed of several key components:

- **Node ID**: A unique identifier for the node, used for routing and storage operations.In our case we generate a binary Id from the *SHA256* hash of the public key. Also in our implementation we have a constant named *NUMBER_OF_BITS_NODE_ID* that defines the length of the node Id (in a realist scenario this value would be at least 160).
- **IP Address**: The network address where the node can be reached.
- **Port**: The communication port used by the node.
- **Public key**: The node's public key, used for secure communication and identity verification.
- **Routing table**: Contains a list of buckets, with each bucket holding nodes that are organized according to the XOR distance metric, as described in [1].
- **Storage**: Acts as a persistent storage system for blocks. It is intended to permanently store mined blocks from the

blockchain, serving as a backup to allow the blockchain state to be reloaded if necessary. In our implementation, storage is simplified by using an in-memory map to store blocks. However, in a more realistic scenario, blocks should be saved to a more permanent storage medium (e.g., file system or database).

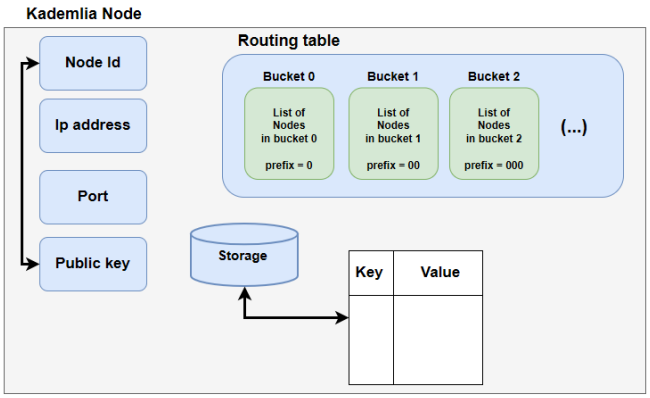

Fig. 3: Kademlia node

B. Operations

Regarding Kademlia and S/Kademlia-related functionalities, the main Remote Procedure Calls (RPCs) defined in our implementation are the following:

- **JOIN_NETWORK**: Operation allowing a node to join the distributed network by contacting an existing bootstrap node.
- **PING**: Basic check to verify if a node is active and reachable.
- **FIND_NODE**: Search operation to locate the K closest nodes to a given target ID, based on the XOR distance metric.
- **FIND_VALUE**: Retrieves the value associated with a specific key if available. otherwise, behaves similarly to *FIND_NODE*.
- **STORE**: Operation to store a given block in the appropriate node(s) according to the DHT structure.
- **GET_ROUTING_TABLE**: Allows a node to retrieve the current state of its routing table or the routing table of a peer.
- **GET_BLOCKCHAIN**: Retrieves the local copy of the blockchain maintained by a node.
- **GET_STORAGE**: Retrieves the set of stored blocks managed by the node's storage subsystem.

All the above RPC operations use **SecureMessage** mechanism to ensure confidentiality, integrity, and authenticity. Furthermore, during communication, each node validates the authenticity of other nodes by verifying that the Node ID is correctly derived from its public key, thereby further mitigating Sybil attacks.

C. Resistance against Sybil and Eclipse attacks

S/Kademlia [3] extends the original Kademlia protocol [1] by introducing several security improvements, most notably to mitigate Sybil and Eclipse attacks.

In our implementation, each node's ID is generated as the hash of its public key, i.e., **nodeId = H(publicKey)**. This relationship is strictly enforced and validated through the `checkNodeId()` function.

Because it is computationally infeasible to generate multiple public keys that produce arbitrary node IDs, an adversary is "unable" to create a large number of valid identities. This mechanism effectively mitigates Sybil attacks by ensuring that node identities are cryptographically bound to their keys and are not freely chosen.

Additionally, since node IDs are deterministically derived from public keys and cannot be freely selected, an attacker cannot strategically place malicious nodes near a target node in the ID space. This prevents the adversary from isolating a node and manipulating its network view, thereby mitigating Eclipse attacks.

V. Peer-to-peer (P2P)

For the testing of our implementation, we simulated a network that resembles the one depicted in Fig. 4. Initially, the P2P network is formed by three peers, with an additional peer joining the network later on. Each peer is designed to handle multiple clients concurrently, utilizing threads and proper synchronization mechanisms to ensure that all operations are processed correctly and consistently.

To maintain full control over communication between peers, we chose to rely solely on low-level socket programming, without using any external libraries beyond the standard libraries.

All tests were conducted locally, as we were unable to obtain the necessary credits to deploy the system on a cloud platform. However, in theory, the application could be deployed across multiple virtual machines (one for each peer) with minimal adjustments, and the system should operate normally under those conditions.

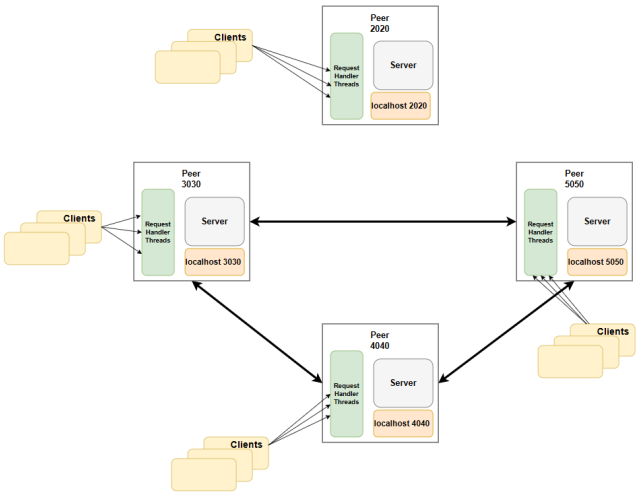

Fig. 4: P2P network used for testing

A. Fault injection mechanism

For the fault injection mechanism, we developed two simple scenarios to demonstrate the system's resilience:

- **Invalid Block Injection:** We attempted to inject an invalid block into a peer's blockchain. The peer correctly identified the block as invalid, rejected it, and returned an appropriate error message.
- **Invalid Node Injection:** We attempted to insert an invalid node into a peer's routing table. The peer validated the node's identity and, upon detecting the invalidity, rejected the node and returned an error message.

These tests showcase the system's ability to detect and mitigate faulty or malicious actions, maintaining the integrity of both the blockchain and the peer-to-peer network.

VI. Auction

For the auction component of the project, we defined three basic transaction types: `START_AUCTION`, `CLOSE_AUCTION`, and `PLACE_BID`.

In addition, we implemented the logic required to support a simple auction system, including functionalities to retrieve available auctions, list the bids placed on a specific auction, and determine and print the winner once an auction is closed.

Transactions are considered "permanent" and "valid" only when they are recorded in the blockchain. The only exception is the `CLOSE_AUCTION` transaction, which is considered valid immediately upon creation, as it can only be issued by the original creator of the respective auction.

The system automatically verifies the validity of each transaction. If a transaction is found to be invalid — either at the moment of creation or later — it is either rejected or removed from the transaction pool, ensuring the consistency and reliability of the auction mechanism.

A. "Publisher/Subscriber" system

In this setup, nodes that create auctions or place bids act as "publishers," while other nodes interested in receiving auction and bid updates act as "subscribers." Although the system does not implement a fully asynchronous event-driven Pub/Sub model, it offers basic subscription functionality by allowing peers to request and retrieve updated auction-related data on demand.

VII. General Assumptions

Throughout the development process, we made several assumptions regarding the system and its behavior. These assumptions include:

- All peers/nodes in the network act as miners.
- All peers/nodes in the network also act as bootstrap nodes.
- The k-closest value used in Kademlia operations is assumed to be sufficiently large to return all nodes in the network.
- Each bucket in the routing table can hold at least n nodes, where n represents the total number of nodes in the network.
- The `STORE` RPC in Kademlia is implemented using a broadcast-like pattern to ensure wider propagation of data.
- The blockchain is represented as a simple list of blocks (instead of a directed acyclic graph (DAG) or a tree structure).
  - To address this limitation and support forks, the blockchain could be modeled as a tree where each block can have multiple children. In case of a fork, nodes would follow the chain with the highest cumulative work or the highest reputation (depending on the consensus mechanism).
- Each peer in the network has access to the correct public key of all other peers and securely stores its own private key.
- We assume that there are no instances of fork or message delays during normal operation.

VIII. Conclusion

In this project, we developed a distributed system capable of managing a public ledger for auctions, integrating key components such as a blockchain, a secure P2P communication layer based on S/Kademlia, and a simple auction mechanism. Throughout the development process, we encountered several challenges, most of which we were able to successfully overcome.

Future work could involve refining and optimizing the communication system to make the interactions between peers clearer and more efficient. Additionally, implementing the Proof-of-Reputation consensus mechanism would enhance security and robustness by incorporating trust dynamics directly into the consensus process. Moreover, a more complete Publisher/Subscriber (Pub/Sub) system could be designed to fully support real-time auction events. Finally, deploying and testing the system on a cloud platform would allow us to evaluate its behavior in a more realistic, large-scale environment, uncovering possible challenges not observable in a local setup.

References

[1] Petar Maymounkov and David Mazieres. Kademlia: A peer-to-peer information system based on the xor metric. In International Workshop on Peer-to-Peer Systems, pages 53–65. Springer, 2002

[2] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008

[3] Ingmar Baumgart and Sebastian Mies. S/kademlia: A practicable approach towards secure key- based routing. In Parallel and Distributed Systems, 2007 International Conference on, pages 1–8. IEEE, 2007

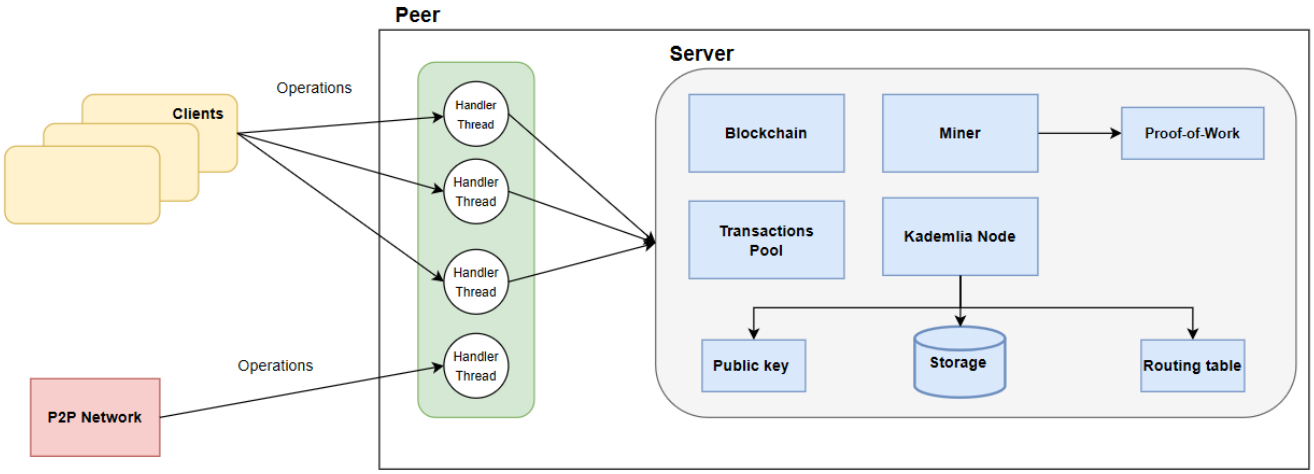

Fig. 5: General Architecture