Faculdade de Ciências da Universidade do Porto

Public Ledger for Auctions

Project Final Report

Diogo Silva, up202105327
Leandro Costa, up202408816

Master in Segurança Informática

Professors: Prof. Rolando da Silva Martins

May 24, 2025

1 System Architecture

Distributed Ledger

The project implements a robust distributed ledger system that serves as the backbone of the entire application. At its core, the system relies on a **Proof of Work (PoW)** consensus mechanism, wherein each block must satisfy a predefined difficulty level to be considered valid. This mechanism ensures integrity and resistance to tampering.

Blockchain Implementation: Implemented using a custom `Block` structure, which includes the following essential components:

- **Transactions:** Stored as a vector of bytes, providing a simple but effective way to store transaction data.
- **Timestamp:** Managed using the `chrono` library to provide precise time records.
- **Previous Block Hash:** Establishes a cryptographic link to the preceding block, ensuring immutability.
- **Nonce:** A variable used in PoW computation to meet the target difficulty.
- **Block Hash:** Computed using SHA-512, combining all block components for integrity verification.

The blockchain implementation uses a structured block design consisting of a header and body. The block header contains essential metadata about the block and the block body contains the actual transaction data.

Listing 1: Block Header Struct

```
#[derive(Debug, Clone, Serialize, Deserialize)]
pub struct BlockHeader {
    prev_hash: Vec<u8>,    // Hash of the previous block
    nonce: u64,           // Number for Proof of Work
    difficulty: u64,      // Required number of leading zeros
    timestamp: u64,       // Block creation time
}
```

Listing 2: Block Body Struct

```
#[derive(Debug, Clone, Serialize, Deserialize)]
pub struct BlockBody {
    transactions: Vec<u8>,  // Serialized transaction data
}
```

Storage Mechanism: The system utilizes the **Kademlia Distributed Hash Table (DHT)**. Upon creation, each block is serialized and hashed, and a unique storage key is derived by truncating the final 20 bytes of the hash. This key is then used to store the block in the DHT, ensuring decentralized and collision-resistant placement within the network.

During initialization, the **genesis block**, the first block of the blockchain, is created using the `create_genesis_block` function. This block defines the initial state of the distributed ledger.

Blocks are stored locally in a thread-safe manner using mutex locks. This design ensures safe concurrent access to the blockchain, particularly when multiple operations, such as reading or appending blocks, occur simultaneously.

Proof-of-Work: The block's security is maintained through a sophisticated hashing mechanism using SHA-512, where the hash is computed by combining multiple components. The get_hash() method demonstrates this multi-component approach:

Listing 3: Get Hash from block method

```
    pub fn get_hash(&self) -> Vec<u8> {
        let mut hash = digest::Context::new(&digest::SHA512);
        hash.update(&self.header.get_parent_hash());
        hash.update(self.header.get_timestamp().to_string().as_bytes());
        hash.update(self.body.get_transactions());
        hash.update(self.header.get_nonce().to_string().as_bytes());
        hash.update(self.header.get_difficulty().to_string().as_bytes());
        let digest_result = hash.finish();

        let mut hash_vec = Vec::new();
        for byte in digest_result.as_ref() {
            hash_vec.push(*byte);
        }
        hash_vec
    }
```

The implementation includes a proof-of-work consensus mechanism through its mining process. The `mine()` method systematically attempts different nonce values until it finds a valid hash:

Listing 4: Mine block method

```
    pub fn mine(&mut self) -> u64 {
        loop {
            if self.is_valid() {
                println!(
                    "Block mined with nonce: {}, hash: {}",
                    self.header.get_nonce(),
                    hex::encode(self.get_hash())
                );
                return self.header.get_nonce();
            }

            self.header.set_nonce(self.header.get_nonce() + 1);
        }
    }
```

And a block is considered valid when its hash meets the difficulty requirements, as implemented in the `is_valid()` method:

Listing 5: Method to check block's validity

```
    pub fn is_valid(&self) -> bool {
        let hash = self.get_hash();
        let target = "0".repeat(self.header.get_difficulty() as usize);
        let hash_str = hex::encode(hash);
        hash_str.starts_with(&target)
    }
```

This method systematically attempts different nonce values until it finds one that produces a hash meeting the difficulty requirements, specifically, a hash that begins with a number of zeros equal to the difficulty level. The difficulty parameter can be adjusted to control the computational effort required for mining, allowing the system to maintain a consistent block creation rate regardless of network computing power.

For data persistence and network communication, the block implements serialization and deserialization capabilities using the serde framework, as evidenced by the ´`serialized()` and `deserialized()` methods.

Secure P2P Communication

The project implements a secure P2P network using:

SKademlia DHT: Each node is uniquely identified by its ID, which is derived from its public key. This cryptographic identity ensures that nodes cannot easily spoof their identity. The implementation uses the ring library for cryptographic operations, ensuring secure key generation and verification. The project implements SKademlia to provide a secure and efficient distributed storage and routing system. The routing table is a fundamental component of SKademlia, implemented as follows:

Listing 6: RoutingTable Struct

```
pub(crate) struct RoutingTable{
    curr_node: node::Node,
    k_bucket_map: HashMap<u8, k_bucket::K_Bucket>,
    local_storage: HashMap<[u8; 20], Vec<u8>>,
}
```

Nodes: Implemented as the fundamental units of the network, where each node maintains a routing table of known peers. Being the nodes identified by their ID (derived from their public key). The core node structure is defined by their `ID`, a 160-bit identifier, an `IP` and a `port`.

The node ID serves as the primary identifier in the network, derived from the node's public key. This ID is crucial for determining node distance in the network, organizing the routing table, and maintaining the network topology. The project implements the XOR-based distance metric, which is fundamental to Kademlia's operation:

Listing 7: Function to calculate distance between nodes

```
pub fn distance(node_id1: &[u8; 20], node_id2: &[u8; 20]) -> [u8; 20] {
    let mut dist = [0u8; 20];
    for i in 0..20 {
        dist[i] = node_id1[i] ^ node_id2[i];
    }
    dist
}
```

This distance calculation forms the basis for organizing nodes in the routing table and optimizing network queries. The implementation ensures efficient node discovery and routing by maintaining a structured view of the network topology.

Bucket Implementation: The routing table is organized into buckets, each containing nodes that share a specific prefix length in their ID space. The bucket structure is defined as:

Listing 8: Bucket Structure

```
pub(crate) struct K_Bucket{
    k: usize,
    nodes: VecDeque<Node>,
}
```

Security Features: The protocol also prevents **eclipse attacks** by requiring the nodes to maintain multiple disjoint paths to other nodes, through the previous implementation. To resist Sybil attacks, the idea is a static cryptographic puzzle mechanism that requires nodes to generate valid IDs through proof-of-work. This would utilize the `ring` library for cryptographic operations, specifically Ed25519 key pairs and SHA-256 hashing.

The proof-of-work mechanism utilizes `ring::digest::Context` to compute block hashes that must meet a specific difficulty target, defined by the number of leading zeros. This is illustrated in the block validation process:

Listing 9: Block Validation

```
pub fn is_valid(&self) -> bool {
    let hash = self.get_hash();
    let target = "0".repeat(self.header.get_difficulty() as usize);
    let hash_str = hex::encode(hash);
    hash_str.starts_with(&target)
}
```

Communication Protocol: Utilizes gRPC for efficient and reliable communication between nodes, ensuring effective data exchange across the network. It employs the tonic and tokio libraries for asynchronous operations, enabling the system to handle multiple concurrent requests without blocking. Each node in the network operates a continuous server thread that listens for incoming requests, while new requests are processed in separate threads to maintain both responsiveness and scalability.

The communication protocol is defined in the **communication.proto** file, which outlines the gRPC service and message structures. The Kademlia service includes several RPC methods:

- Ping: Used to check if a node is alive.
- Store: Allows a node to store a key-value pair in the network.

- FindNode: Retrieves the k-closest nodes to a given key.
- FindValue: Retrieves a value associated with a key, or the k-closest nodes if the value is not found.

These methods are implemented in the server code, where each node handles incoming requests and performs the necessary operations. The use of gRPC provides a strong foundation for the communication protocol, offering features like bidirectional streaming, strong typing, and efficient binary serialization, which are essential for the distributed nature of the system.

Auction Mechanism

Auction Structure: Demonstrates the practical application of the distributed ledger. Every auction is uniquely identified and stored on the blockchain using a specific key format: **auction:x**, where x represents an incrementing ID. This ID is managed by keeping track of the latest auction and incrementing it for new auctions. The auction system uses `serde` and `serde_json` for serialization and deserialization, which is crucial for storing auctions in the distributed ledger:

Listing 10: Serialized /Deserialized functions

```
    pub fn serialized(&self) -> String {
        serde_json::to_string(self).unwrap()
    }
    pub fn deserialized(serialized: &str) -> Self {
        serde_json::from_str(serialized).unwrap()
    }
```

Storage and Validation: The auction system implements a robust storage and validation mechanism. Auctions are stored directly in the blockchain, ensuring immutability and transparency. The storage system uses a hierarchical key structure:

- Auction storage: auction:x where x is the auction ID
- Bid storage: auction:id:bid:x for individual bids

Validation is a critical component of the system, implementing multiple layers of verification:

- Timestamp verification ensures bids are made within the auction duration
- Price verification validates bid amounts against auction rules
- Hash verification of the auction/bid object prevents tampering

Transaction Flow: The auction system implements a clear and secure transaction flow. For auction creation, the process follows a specific sequence:

1. The auction object is serialized
2. The serialized data is hashed
3. The resulting block is stored in the blockchain

Bid submission follows a similar but more complex process:

1. Bids are validated against auction rules
2. Valid bids are stored with cryptographic signatures
3. Signatures contain both the auction ID and hash for verification

Signature Implementation: Critical security feature that ensures the integrity and authenticity of auction data. The implementation uses cryptographic signatures to verify the identity of participants and prevent tampering with auction data.

Listing 11: Auction Signature Structure

```
pub(crate) struct AuctionSignature {
    pub auction_id: String,
    pub auction_hash: Vec<u8>,
}
```

Signatures have in their structure the acution ID and the auction hash They are stored in the blockchain to ensure their immutability.

Publisher/Subscriber Pattern: Used for real-time auction updates. This pattern serves several critical functions: Enables real-time tracking of latest bids, maintains consistent auction state across the network and provides immediate updates to all participating nodes.

The publisher/subscriber implementation uses the tokio runtime for handling async operations, ensuring efficient communication between nodes. This pattern is particularly important for maintaining auction fairness and transparency, as all participants receive updates simultaneously.

2 Technical Implementation

Key Libraries: The project leverages several key libraries to implement its functionality. The cryptographic operations are handled by the `ring` library, while random number generation is managed by `rand`. The async processing is implemented using `tonic` and `futures`, with `tokio` providing the runtime and gRPC functionality.

Architecture Components: The user interface is built using `eframe` and `egui`, which provide a modern and responsive GUI. The implementation includes features like network visualization, auction management, transaction history, node information display. The architecture is designed for scalability and maintainability, ensuring a clear separation of concerns between various components. The GUI thread runs continuously, managing user interactions and displaying information, while the server thread handles network operations and maintains the Kademlia network.

Storage is implemented with multiple layers:

- Local storage with mutex protection for thread safety
- Distributed storage through the Kademlia DHT for network-wide data
- Blockchain for an immutable transaction history