Simplified Blockchain Implementation and Verification

1 Introduction

The main objective of this lab assignment is to build a simplified blockchain. A blockchain is basically a distributed database of records. What makes it unique is that it uses cryptographic hash functions to create a tamper-proof mechanism of committed transactions through distributed consensus. Most blockchains are permissionless, which means that they allow public membership of nodes, often implemented on top of a peer-to-peer network, allowing a public distributed database, i.e. everyone who uses the database has a full or partial copy of it. A new record can be added only after the consensus between the other keepers of the database. Also, it's blockchain that made crypto-currencies and smart contracts possible.

This lab consists of three parts. Each part will be explained in more detail in their own sections.

1. **The chain of blocks:** Implement a chain of blocks as an ordered, back-linked list data structure.
2. **Efficient transactions and blocks verification:** Implement an efficient way to verify membership of certain transactions in a block using Merkle Trees.
3. **Command line client and benchmarks**

2 Explanation

2.1 Blocks In blockchain, it's the blocks that store valuable information. For example, Bitcoin blocks store transactions, the essence of any crypto-currency. Besides this, a block contains some technical information, like its version, current timestamp, and the hash of the previous block. In this assignment, we will not implement the block as it's described in current deployed blockchains or Bitcoin specifications, instead, we'll use a simplified version of it, which contains only significant information for learning purposes. Our block definition is defined in the `block.go` file and has the following structure:

```
type Block struct {
    Timestamp int64
    Transactions []*Transaction
    PrevBlockHash []byte
    Hash []byte
}

type Transaction struct {
    Data []byte
}
```

`Timestamp` is the current timestamp (when the block is created), `Transactions` is the actual valuable information containing in the block, `PrevBlockHash` stores the hash of the previous block, and `Hash` is the hash of the block. In Bitcoin specification `Timestamp`, `PrevBlockHash`, and `Hash` are block headers, which form a separate data structure, and `Transactions` is a separate data structure (for now, our transaction is only a two-dimensional slice of bytes containing the data to be stored). You can read more about how transactions are implemented here.

Each block is linked to the previous one using a hash function. The way hashes are calculated is a very important feature of blockchain, and it's this feature that makes blockchain secure. The thing is that calculating a hash is a computationally difficult operation, and it takes some time even on fast computers. This is an intentional architectural design of blockchain systems, which makes adding new blocks difficult, thus preventing their modification after they're added. We'll discuss and implement this mechanism in the Lab 3.

For now, you will just take block fields (i.e. headers), concatenate them, and calculate a SHA-256 hash on the concatenated combination. To do that, use the `SetHash` function. Feed the `PrevBlockHash`, `Transactions`, and `Timestamp` into the hash in this order.

To compute the SHA-256 checksum of the data you can use the `Sum256` function from the go crypto package.

We also want all transactions in a block to be uniquely identified by a single hash. To achieve this, you will get hashes of each transaction, concatenate them, and get a hash of the concatenated combination. This hashing mechanism of providing unique representation of data will be given by the `HashTransactions` function, that will take all transactions of a block and return the hash of it to be included in the block `Hash`.

2.2 Blockchain

Now let's implement a blockchain. In its essence, a blockchain is just a database with a certain structure: it's an ordered, back-linked list. Which means that blocks are stored in the insertion order and that each block is linked to the previous one. This structure allows to quickly get the latest block in a chain and to get a block by its hash.

In Golang, this structure can be implemented by using an array and a map: the array would keep ordered hashes (arrays are ordered in Go), and the map would keep hash to block pairs (maps are unordered). But for now, in your blockchain prototype, you just need to use an array as shown below.

```
type Blockchain struct {
    blocks []*Block
}
```

2.3 Merkle Trees

Merkle Trees, conceptualized by Ralph Merkle, are a critical component in the field of computer science and have found significant application in blockchain technology. These tree structures offer a secure and efficient mechanism for summarizing and validating the integrity of large data sets. In blockchain, they are particularly pivotal for aggregating and verifying transactions, ensuring the reliability and robustness of the blockchain's data integrity.

2.4 Structure of a Merkle Tree

The architecture of a Merkle Tree is an interesting combination of simplicity and cryptographic security. It can be understood in three primary components:

- **Leaf Nodes:** Each leaf node in a Merkle Tree signifies an individual data block, which can be a single transaction, a document, or any piece of data. These nodes are the fundamental building blocks of the tree and are formed by hashing the data they represent using a cryptographic hash function, like SHA-256, ensuring that the data's integrity is cryptographically secure.

- **Non-Leaf Nodes:** The non-leaf, or intermediate nodes, are formed by hashing the concatenation of their child nodes. This process is recursively applied, starting from the leaf nodes and moving upwards, thereby linking the nodes cryptographically. This chaining effect ensures that any alteration in a single piece of data alters the hash of the parent node, making unauthorized changes detectable.
- **Root Node:** The top of the tree, called the root node, is the result of this step-by-step hashing process. The hash stored in this node, known as the Merkle Root, is a single hash that effectively represents the entire dataset beneath it. The integrity of the Merkle Root is is extremely important as it reflects the integrity of the entire tree.

2.5 Merkle Trees in Blockchain In the context of blockchain, Merkle Trees play two vital roles that are crucial for the technology's efficiency and security:

1. **Block Integrity:** Within a blockchain, each block typically contains a multitude of transactions. To ensure the integrity of these transactions, a Merkle Tree is constructed with each transaction forming a leaf node. The Merkle Root of this tree is then stored in the block's header. This structure not only secures the transactions but also links each block to its predecessor, creating an unbreakable chain of blocks, hence the term 'blockchain'.
2. **Efficient Verification:** One of the standout features of using Merkle Trees in blockchain is the efficiency it brings to data verification. Instead of needing to review an entire block or the entire blockchain, a user or a node can simply check the integrity of a specific transaction against the Merkle Root in the block header. This process, known as Merkle Proof, significantly reduces the amount of data required for verification, enabling faster and more efficient validation processes, which is particularly advantageous in large-scale, distributed blockchain networks.

2.6 Developing a Command Line Client

1. **Designing the Interface:** Outline the commands and functionalities your client will support.
2. **Parsing Commands:** Implement command parsing logic.
3. **Interacting with Blockchain:** Ensure your client can interact with the blockchain backend.

3 Submission regulation

- Students create a folder <Group's ID> containing the contents following:
  - <Code> folder: contains the whole project.
  - Executable file (optinal).
  - <Report.pdf> file(at most 15 pages, single page, 12pt font) that includes:
    - Information about team members.

  * Project structure.
  * Data structure.
  * Any other remarks about your design and implementation.
  * All links and books related to your submission must be mentioned.
  * **DO NOT** insert your source code in the report.
  * Description of how your system operates: Provide a detailed explanation of the functioning and flow of your system, highlighting key components and interactions.
  * Challenges encountered during implementation: Discuss any technical, design-related, or logistical hurdles you faced and how you overcame them.
  * Lessons learned: Share insights and knowledge gained throughout the project and explain how it contributed to your personal growth.
  * The report can be written in Vietnamese or English.

- Compress the above folder into Group's ID.zip for submission.
- Submission with wrong regulation will result in a "0" (zero).
- Plagiarism and Cheating will result in a "0" (zero) for the entire course and will be subject to appropriate referral to the Management Board for further action.

4 Grading (may be changed later)

1. Coding: 5 pts.
2. Report: 7 pts.
3. Total: 12 pts.

Good luck with your lab assignment!