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. Use the provided skeleton code and unit tests.
2. **Efficient transactions and blocks verification:** Implement an efficient way to verify membership of certain transactions in a block using Merkle Trees. Use the provided skeleton code and unit tests.
3. **Command line client and benchmarks**

For each part of the assignment you should copy your implementation of the previous part. But **do not copy the tests**, they can differ from each part, copy only your implementation. If you prefer, you can create a new branch for each part of the assignment.

2 Part 1

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
}
```

3 Submission regulation

- Students create a folder <Student's ID> containing the contents following:
  - <Code> folder: contains the whole project.

  - Executable file (optinal).
  - <Report.pdf> file(at most 5 pages, single page, 12pt font) that includes:
    * Project structure.
    * Data structure.
    * Any other remarks about your design and implementation.
    * Table of complete features with self-grading.
    * All links and books related to your submission must be mentioned.
    * **DO NOT** insert you source code in the report.

- Compress the above folder into Student'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: 7 pts.
2. Report: 5 pts.
3. Total: 12 pts.

Good luck with your lab assignment!