L-Università ta' Malta

DLT5400 - DLT Implementation and Internals

Assignment P1 - SBB Blockchain Implementation

Balaganapathy Vanagiri Selvakumar

Centre for Distributed Ledger Technologies

July 4, 2023

1 I - Average Block Time

In this section, we will see how we decided on our mining difficulty, i.e. the number of consecutive zeros on our block hash. It should be noted that this instance was run on a *MacBook Pro* with *Apple Silicon M1 Max* with *32G of RAM.*

Difficulty	Encoding	TXN Model 3	AVG Time / Block
5 Zero's	JSON	Account	1.38
5 Zero's	JSON	UTXO	1.85
5 Zero's	BYTE	Account	-
5 Zero's	BYTE	UTXO	-
6 Zero's	JSON	Account	18.5
6 Zero's	JSON	UTXO	21.4
6 Zero's	BYTE	Account	-
6 Zero's	BYTE	UTXO	-

Table 1: Block Time Calculation

The decision has been made to use 5 consecutive zeros as our difficulty level, as it would be easier to demo and the chosen value is near than the higher difficulty times The timings are obtained with inline statements and logs and averaged finally.

2 II - Protocols

The protocol is an essential part of communication between the peers. We have used a structured protocol with a JSON Payload object.

2.1 Protocol Structure

```
<STX><LEN><PAYLOAD><ETX>
```

Where the *<PAYLOAD>* can be either a JSON object or a Byte Encoding. The implementation only contains the JSON part, the Byte encoding has not been implemented.

2.2 Protocols Used

- ***GET_COUNT*** - This protocol message is used to get the number of blocks from the peers.

```
JSON IMPLEMENTATION -->
<PAYLOAD>
    <OPID> - 'a'
    <OBJ> - {NONE}

BYTE ENCODING IMPLEMENTATION -->
<PAYLOAD>
```

```
    <OPID> - 'a'
FINAL PROTOCOL - <STX><LEN><OPID><ETX>
```

- ***COUNT*** - This protocol message contains the number of blocks that the peer has.

```
JSON IMPLEMENTATION -->
<PAYLOAD>
    <OPID> - 'c'
    <OBJ> - {"blocks": "n"} //WHERE n IS NUMBER OF BLOCKS

BYTE ENCODING IMPLEMENTATION -->
<PAYLOAD>
    <OPID> - 'a'
    <COUNT> - n //WHERE n IS NUMBER OF BLOCKS
FINAL PROTOCOL - <STX><LEN><OPID><COUNT><ETX>
```

- ***GET_BLOCK_HASHES*** - This protocol message is used to get the hashes of the blocks that the peer has mined.

```
JSON IMPLEMENTATION -->
<PAYLOAD>
    <OPID> - 'b'
    <OBJ> - {NONE}

BYTE ENCODING IMPLEMENTATION -->
<PAYLOAD>
    <OPID> - 'b'
FINAL PROTOCOL - <STX><LEN><OPID><ETX>
```

- ***BLOCK_HASHES*** - This protocol message contains the hashes of all the blocks that the peer has.

```
JSON IMPLEMENTATION -->
<PAYLOAD>
    <OPID> - 'h'
    <OBJ> - {"blockHashes":[HASH_1, HASH_2, .., HASH_n]}

BYTE ENCODING IMPLEMENTATION -->
<PAYLOAD>
    <OPID> - 'a'
    <COUNT> - n //WHERE n IS NUMBER OF BLOCKS
    (<HASH>) - HASH_1 // n NUMBER OF HASHES PASSED
```

```
FINAL PROTOCOL - <STX><LEN><OPID><COUNT><HASH><HASH><ETX>
```

- ***REQ_BLOCK*** - This protocol message requests the peer to send the block of a particular hash.

```
JSON IMPLEMENTATION -->
<PAYLOAD>
    <OPID> - 'r'
    <OBJ> - {"hash": HASH}

BYTE ENCODING IMPLEMENTATION -->
<PAYLOAD>
    <OPID> - 'r'
    <HASH> - HASH
FINAL PROTOCOL - <STX><LEN><OPID><HASH><ETX>
```

- ***BLOCK*** - This protocol message contains the block that was requested by the peer.

```
JSON IMPLEMENTATION -->
<PAYLOAD>
    <OPID> - 'x'
    <OBJ> - {"block": blockObj}
    //WHERE blockObj is
    {
    'block': {'nonce': , prevHash': '', 'Transactions': [{},
    {'TXN_Hash': '', 'From': '0', 'To': '', 'Sign': None}],
    'blockHash': ''}
    }
    //EMPTY VALUES ARE FOR STRUCTURAL UNDERSTANDING ONLY
BYTE ENCODING IMPLEMENTATION -->
<PAYLOAD>
    <OPID> - 'x'
    <NONCE><PREV_HASH>(<TRANSACTION>)<BLOCK_HASH> - RESPECTIVE VALUES
FINAL PROTOCOL -
    <STX><LEN><OPID><NONCE><PREV_HASH>(<TRANSACTION>)<BLOCK_HASH><ETX>
```

- ***NEW_BLOCK*** - This protocol message contains the new block object that has been mined.

```
JSON IMPLEMENTATION -->
<PAYLOAD>
    <OPID> - 'z'
```

```
        <OBJ> - {"block": blockObj}
        //WHERE blockObj is
        {
        'block': {'nonce': , prevHash': '', 'Transactions': [{},
        {'TXN_Hash': '', 'From': '0', 'To': '', 'Sign': None}],
        'blockHash': ''}
        }
        //EMPTY VALUES ARE FOR STRUCTURAL UNDERSTANDING ONLY
    BYTE ENCODING IMPLEMENTATION -->
    <PAYLOAD>
        <OPID> - 'z'
        <NONCE><PREV_HASH>(<TRANSACTION>)<BLOCK_HASH> - RESPECTIVE VALUES
    FINAL PROTOCOL -
        <STX><LEN><OPID><NONCE><PREV_HASH>(<TRANSACTION>)<BLOCK_HASH><ETX>
```

THE BELOW IS ONLY FOR PROOF OF TURN(PoT) MINER

- ***GET_PUBLIC_KEY*** - This protocol is to send the Public key to the peers to calculate the next turn.

```
    JSON IMPLEMENTATION -->
    <PAYLOAD>
        <OPID> - 'g'
        <OBJ> - {NONE}
    BYTE ENCODING IMPLEMENTATION -->
    <PAYLOAD>
        <OPID> - 'g'
    FINAL PROTOCOL -
        <STX><LEN><OPID><ETX>
```

- ***PUBLIC_KEY*** - This protocol is to handle the public keys of the peers and store them for mining purposes.

```
    JSON IMPLEMENTATION -->
    <PAYLOAD>
        <OPID> - 'p'
        <OBJ> - {"PK": "0xabc"} // OUR WALLET PUBLIC KEY
    BYTE ENCODING IMPLEMENTATION -->
    <PAYLOAD>
        <OPID> - 'p'
        <PUBLIC_KEY>
    FINAL PROTOCOL -
        <STX><LEN><OPID><PUBLIC_KEY><ETX>
```

3 III - Limitations of Current Implementation

- ***Size Limitations*** - The message size is restricted due to a lack of message pipelines or buffers. In byte encoding, there is a limitation with the <LEN> attribute passed and after some point in the future, the blockchain may come to a halt.
- ***Security Limitations*** - The public-private key pair generated is secured with no encryption or authentication. There is a possibility to perform a Denial of Service attack on the miner.
- ***Storage Limitations*** - No data structure implementation like Merkle trees to store and handle transactions. All the transactions and account databases are not backed up. This might lead to a failure when the chain stops. but that is not a serious concern as the chain is not meant to go down.
- ***Synchronization Limitations*** - There are no techniques like pruning to help synchronize the blocks and transactions between the peers. As soon as the chain grows the peers will find it difficult to keep up with the blocks.

4 IV - Evaluation

Consensus Mechanism	Txn Model	AVG CPU Usage
PoW	Account	24%
PoT	Account	13%
PoW	UTXO	26%
PoT	UTXO	15%

Table 2: Evaluation Metrics

The above is the average CPU usage of the SBB blockchain in various configurations. The Byte encoding is not tabulated due to incomplete implementation. From the above table 2 we can see that the PoT has a significant advantage when it comes to efficient use of resources and mining is more resource extensive. The UTXO model also contributes to significant resource consumption due to the traversal of the chain. The numbers in the table might get higher when the chain gets longer and busier.

5 V - Un-Finished Implementation

- ***UTXO Database*** - The UTXO Database version is not implemented due to my irresponsible time management by myself. The *accountDB.py* holds the implementation of a similar approach for database implementation.
- ***PoT InEffeciency*** - The implementation of Proof of Turn (PoT) is inefficient and choppy when it comes to its execution. Instead of running the calculation of the next miner in all peers every time in a thread, we can run it in a single peer that mines the latest block. This change has not been made due to timeline restrictions.

- ***Well Defined Byte Encoding*** - The protocol has been implemented only in JSON format and not on Well defined Byte Encoding decided as a group component. The byte encoding decided upon needed some updates and changes in the protocol structure. Due to a lack of expertise, knowledge, and time, this task was not implemented.
- ***Error Handling*** - The implementation must have try-expect blocks and Exception handling and type errors.
- ***Cosmetic Changes*** - The implementation needs more print statements and CLI logs. The Interface can be Improved and information can be delivered more clearly.
- ***High Coupling & Low Cohesion*** - The modularity of the code is sub-par and the code is coupled a little higher than recommended. This can be resolved by dedicating more time to the modularity of our code.

6 VI - Merkle Tree

6.1 Introduction

A Merkle tree, also known as a hash tree or binary hash tree, is a tree structure in which every leaf node is labeled with the cryptographic hash of a data block, and every non-leaf node is labeled with the cryptographic hash of the concatenation of its child nodes' labels. The tree is constructed in a way that allows for efficient verification of the integrity and consistency of large amounts of data.

The primary purpose of a Merkle tree is to provide an efficient and secure way to verify the integrity of data. It achieves this by creating a hierarchical structure where the integrity of the entire data set can be verified by checking just a small number of hash values.

6.2 Working

There are three major steps that can help us understand how the Merkle tree works.

Data Segmentation - The data is divided into fixed-size blocks (usually called leaves), and the cryptographic hash of each block is calculated.

Building the Tree - The leaf nodes are arranged at the bottom level of the tree. If the number of leaves is odd, the last leaf is duplicated to create an even number of leaves. The parent nodes are then created by concatenating and hashing the labels of their child nodes. This process continues until a single root node is obtained.

Root Hash - The root node of the tree, also known as the Merkle root, represents the overall integrity of the entire data set.

To verify the integrity of the data, one can compare the computed root hash with a trusted root hash. By comparing the root hashes, it is possible to determine if any of the data blocks have been tampered with or if the data set as a whole has changed.

6.3 Advantages

Merkle Trees offer several practical applications:

Storage - Instead of storing all individual transactions in memory, we can store a single Merkle root representing those transactions. This helps conserve memory space.

Light Nodes - To optimize network communication, we can implement light nodes that only exchange the Merkle root instead of transmitting all transactions. Full nodes, which hold complete data, can serve as trusted sources for the light nodes.

Mining Efficiency - Miners can benefit from Merkle trees to enhance their efficiency and speed. Miners can calculate a hash in real time while receiving new transactions, making the process more efficient.

Easy Verification - Additionally, transaction verification becomes easier with Merkle trees. Merkle trees enable us to verify if a transfer actually occurred by checking if the transaction is included in a block.

By utilizing Merkle trees, we can optimize memory usage, streamline network communication, ensure transaction integrity, and improve the efficiency of mining operations.