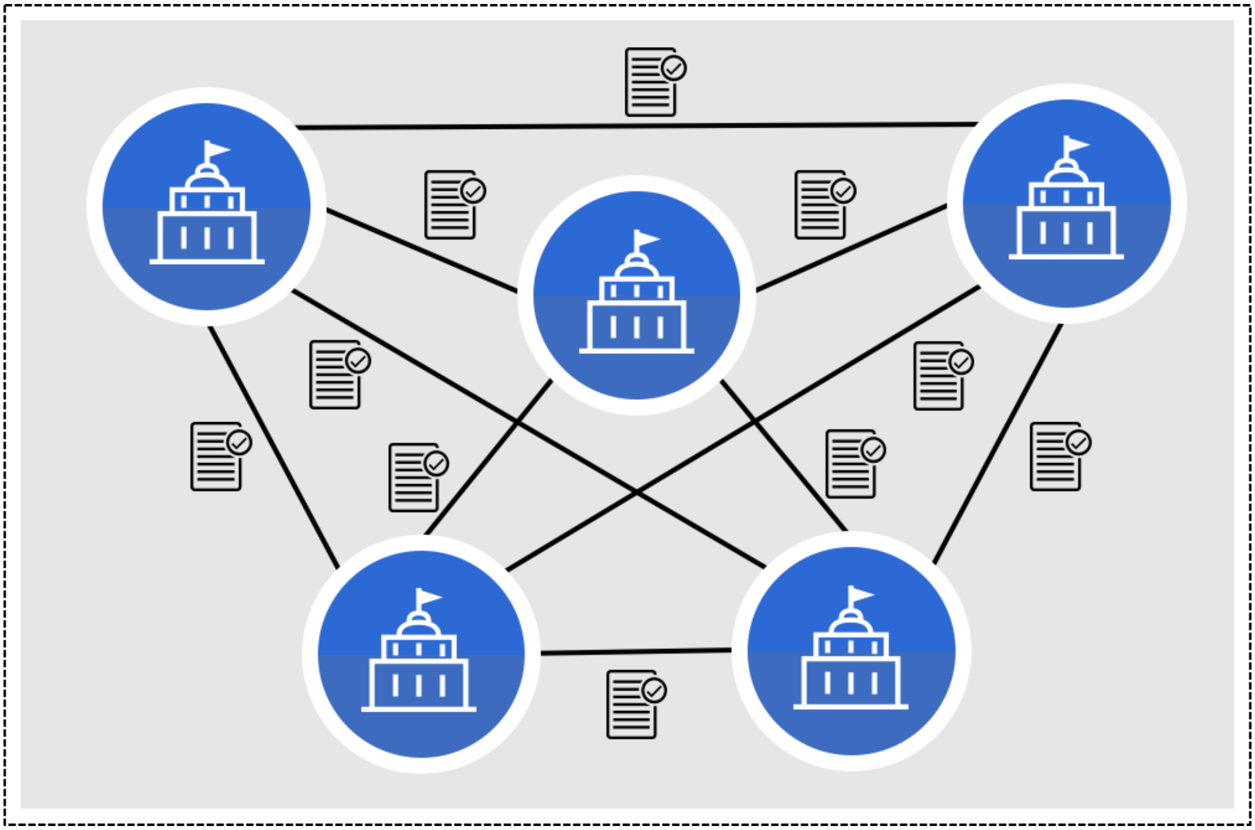


Term Project: A Simple Blockchain

26.05.2020

Team members:-

1) Ashraqat Sheta(13)
2) Rowan Adel(24)
3) Shereen elkordi(32)
4) Fatma ibrahim(42)
5) Nancy Abdelkarem(70)

Overview

it is required to implement a blockchain peer to peer network to handle a ledger of transactions collectively. A blockchain is a chain of blocks that continually grows as more transactions are added to the system. Each block references the block preceding it, and once the network agrees to a block, the transactions included in this block cannot be changed.

Description of coding

Organization

The project consists of three main packages.

- Client Package

  Client package Responsible for start client ,read transactions from File and broadcast transactions to all nodes.

- Node Package

  Client package Responsible for start the numbers of nodes specified

  make a peer to peer network between all nodes,collect transactions from clients , Each node builds up the block from collected transactions and builds block chains.

- Utiles Package

  Client packages include the important functions that help both client and nodes,and model of block chain.

- Blockchain Package

  Responsible for building, mining, and verifying blocks and transactions.

Main Functions

Client Part

- **generateKeys()**

  It generates public and private keys using the “RSA” algorithm,

  input:a number of users.

  output: list of private and public keys.

- **hashTransaction()**

  It makes a hash to the transaction by using the “SHA-256” algorithm.

  Input: the transaction in the form of string

  output: hash in an array of bytes.

- **getSignature()**

  It returns the signature of the transaction in the form of a hash by using the “SHA256withRSA” algorithm.

  Inputs: private Key and hash of the transaction.

  Outputs: signature in array of bytes

- **verifySignature()**

  Verify the signature by using the ”SHA256withRSA” algorithm.

Blockchain Part

- **build()**
  - ★ Get a transaction from the transaction pool of the current node
  - ★ Validate the transaction
    1. verify its signature
    2. check double spending
  - ★ Add only the valid transactions to the block
  - ★ Repeat the above steps until reaching the predefined block size.
  - ★ establish the header of the block by setting its prevHash, merkleHash, time, nonce.
  - ★ Send the block to all other nodes.
- **validateTransaction()**
  - ★ Takes a transaction as input
  - ★ Check if the signature of the transaction is valid
  - ★ If the transaction is valid, checks the double spending.

- ★ For the double spending validation
    1. We have a map, its key, transaction number and its value, the output list of the transaction.
    2. Get the previous transaction number of the input transaction, search with it in the map to get the output list.
    3. If the output list is empty or the output index of the input transaction doesn’t exist in the list, then return invalid transaction.
    4. If the output index of the input transaction exists in the list then remove it from the output list in the map then return a valid transaction.

- **merkleTree()**
    - ★ It takes the block transactions as input.
    - ★ Add the hashes of the block transactions in a list
    - ★ Calculate hash of each pair of transactions, then calculate hash of each pair of hashes and so on.
    - ★ Finally, we construct a tree of the hashes with one tree root hash.
- **POW()**
    - ★ Takes the constructed block as input.
    - ★ Get the previous block hash, timestamp, merkle root hash.
    - ★ Set nonce with initial value 0.
    - ★ Calculate the hash of the previous variables and check if the first difficulty bits equal 0, otherwise increments the nonce and recalculates the hash.
    - ★ Once it reaches the valid hash, we set the nonce of the block and return the block hash.

- **We have 3 threads:**
    1. **One that receives the transactions from the client**
    2. **One that builds the block.**
    3. **One that receives blocks from the nodes.**

- **On receiving a block**
    - ★ Suspend the building block thread.
    - ★ Checks the level of the block, if the receiving node has a block in its blockchain with the same level then it discards the receiving block.
    - ★ Otherwise, Validate the block header hashing
    - ★ If the block is valid, it adds it to its blockchain.
    - ★ Finally, we resume the building block thread.

Compile and run

The program can be run in different machines located in the same network.

The program is set up using a configuration file that contains addresses of the nodes and clients in the network.

Steps to run:-

1. Locate the file of transaction database in the directory of the program
2. Set up addresses of the nodes and client in the configuration file
3. Start run the nodes and client of the network
4. When you start a node enter the name of the node.
5. Finally enter any key to start transferring transactions and blocks.

Component of Block chain

Client

scan transaction from database and broadcast transaction to all nodes one at a time with delay 0.05 ms.

Nodes

builds up the block from collected transactions and puts it in the Transactions pool.

Transaction

Consists of

1. Index
2. Payer Public key
3. Output index
4. Transaction hash
5. Transaction signature
6. List of outputs

Outputs

1. Value
2. Index
3. Payee public key

Block

1. hashPrevBlock:
2. hashMerkleRoot:
3. Timestamp
4. Nonce
5. Level
6. transactions

TransactionPool

1. Arraylist of transactions
2. Hashmap : key -> transaction number, value -> list of outputs

Problems

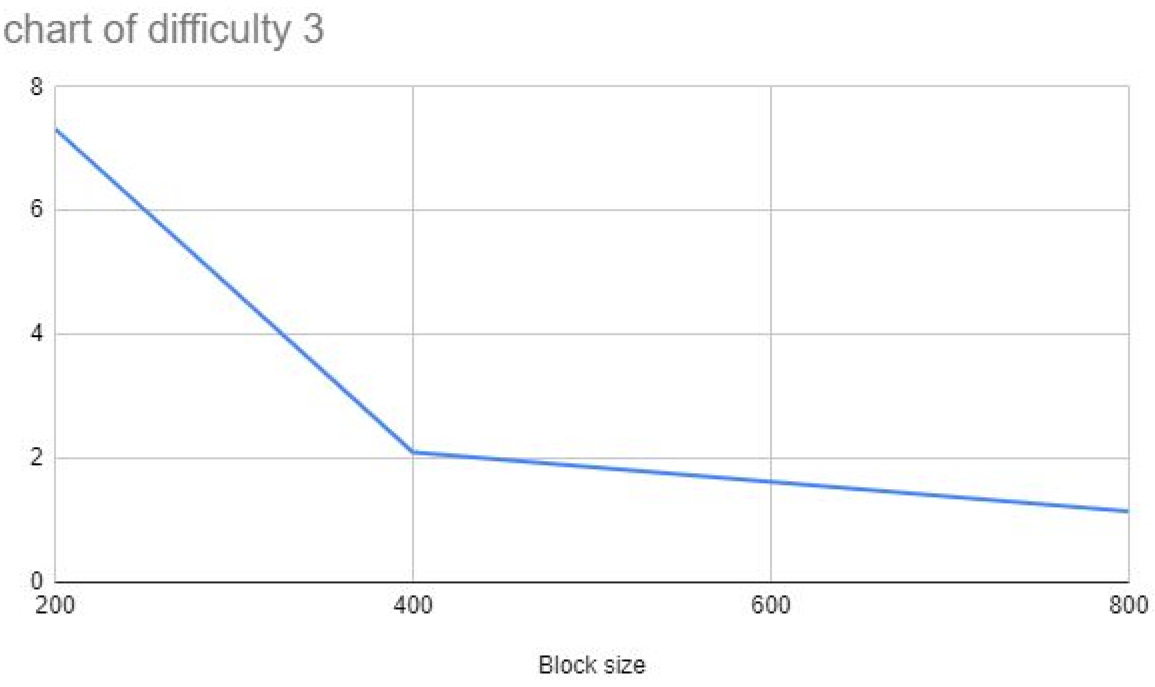
- Failed to use port forwarding as the most of the team didn't find the port forwarding section in their setting page and error in parameter configuration setup.
- It takes a time to load the transactions from the file then sending it to the nodes.
- It takes time from us to configure how to interrupt the thread that is responsible for building the block once the node receives a block then resume the thread.

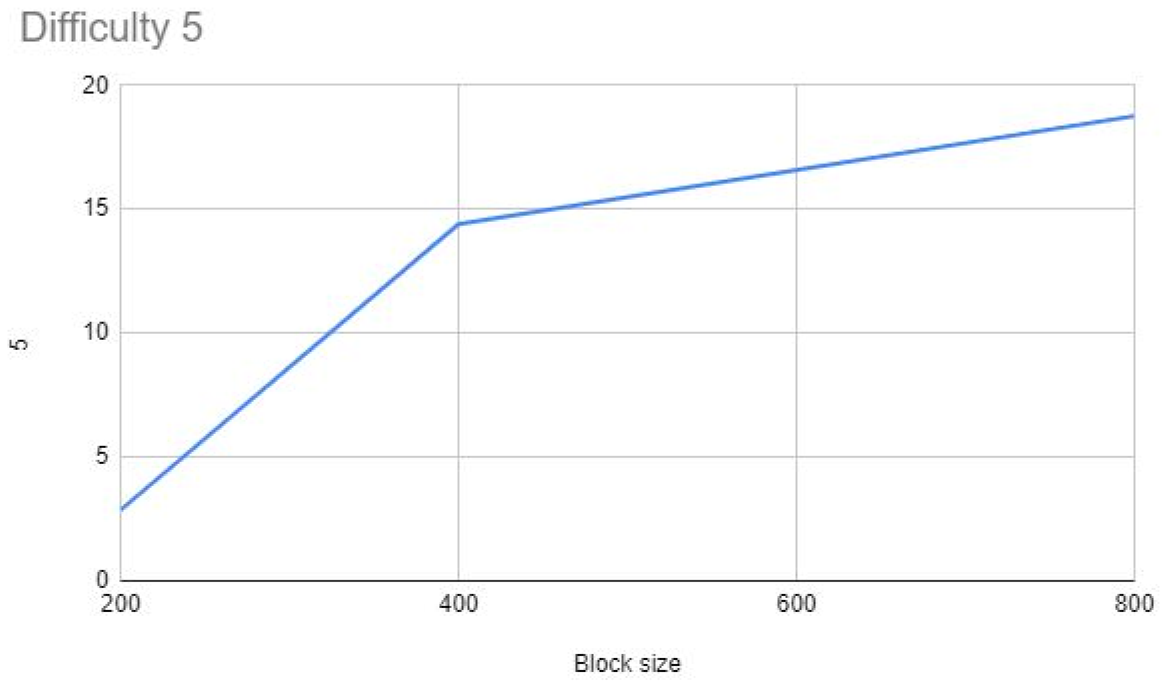
Results

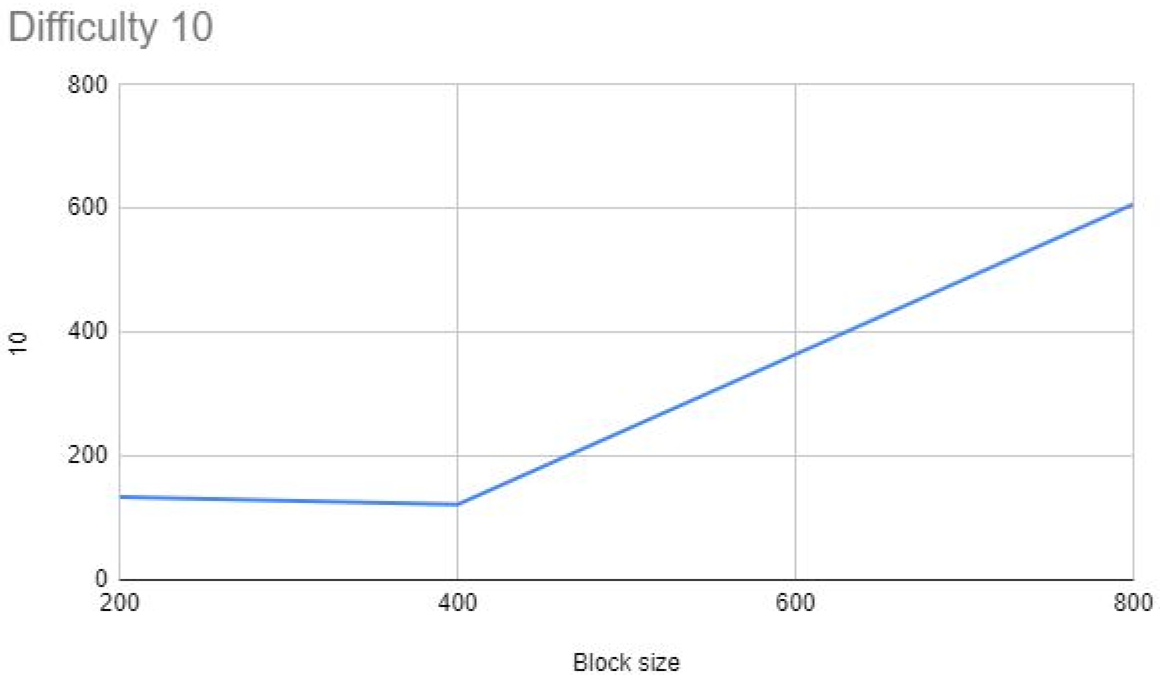
Average time to mine a block

All measures are in Milliseconds and using 1 node as client and 3 other nodes as miners

Difficulty / Block size	200	400	800
3	7.32	2.0964	1.1492
5	2.8626	14.40452	18.7686
10	134.0378	121.48	606.7522

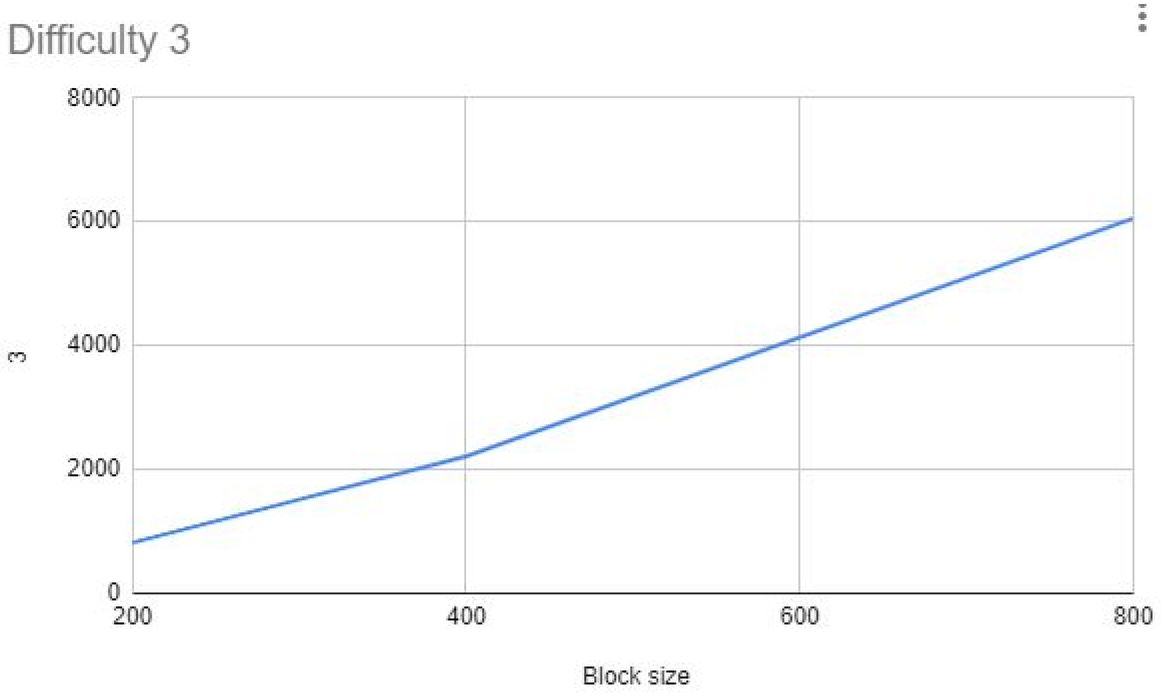


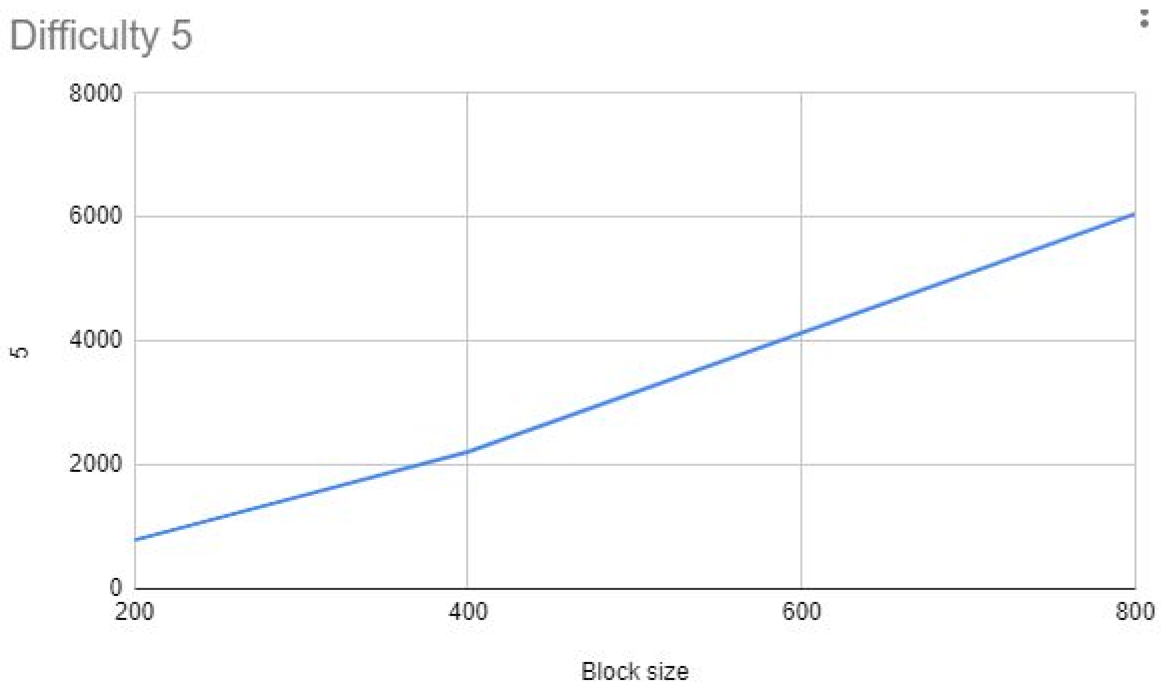

Difficulty 5
20
15
10
5
0
5
200
400
600
800
Block size


Difficulty 10
800
600
400
200
0
10
200
400
600
800
Block size

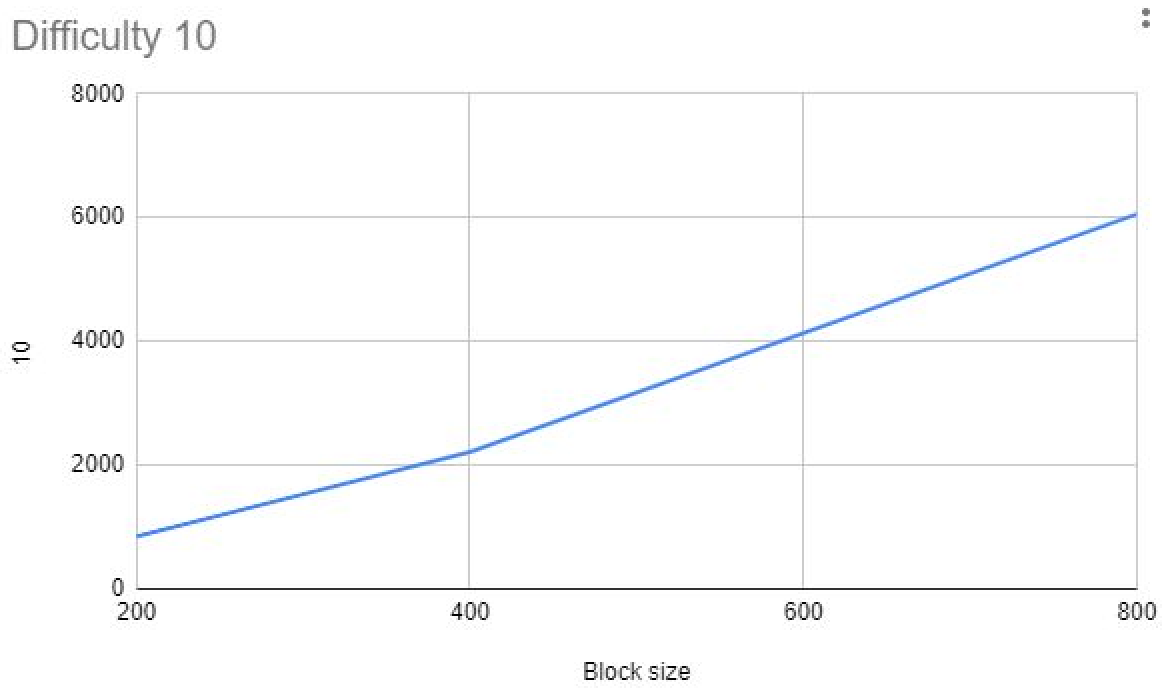
Average message complexity per block

Difficulty / Block size	200	400	800
3	819	2212	6056
5	786	2212	6053
10	845	2212	6056





Difficulty 5



Difficulty 10