COL819: Bitcoin Assignment-3 Report

Garima(2018MCS2017), Anshu Bansal(2018MCS2142)

12th July, 2020

1 Introduction

We have successfully implemented a bitcoin system with 'n' number of nodes where 'n' is adjustable. There are 'n' independent threads in the system and the network between these nodes is assumed to be fully connected. A node can perform any number of transactions and the node that wins in the proof-of-work and satisfies consensus requirements finally gets to create a block which is added to the immutable block chain.

To maintain **integrity of the blocks**, we create Merkle tree of all the transactions present in the block. The tree stores hash pointers at each level. A block also stores hash pointer of the previous block in the chain. This ensures any tampering with a transaction or a block leads to disturbance in the hash values along the complete chain which can not go undetected. The hash value to be used is also an adjustable parameter.

Each node is given 5 wallets i.e. 5 pairs of $< publicKey, privateKey >$ to give multi-transaction support. We experimented with the different hyper-parameters of the block chain like *arity of merkle tree, size of nonce* etc. and captured results using a series of plots. The complete experiments along with the detailed implementation are provided in the coming sections.

2 Implementation Details

2.1 Merkle Tree

```
def generateMerkleTree(self, txns):
    child = []
    for i in txns:
        child.append((MerkleTree([i],True),0))
    while(len(child) > 1):
        level = child[0][1]
        merkleTreeChild = []
        for i in range(arity):
            if(len(child) > 0 and child[0][1] == level):
                merkleTreeChild.append(child[0][0])
                child.remove(child[0])
        child.append((MerkleTree(merkleTreeChild),level+1))
    return child[0][0]
```

This method takes a list of transactions as input and returns the root of constructed Merkle Tree, which is built in a bottom up manner with all the transactions present at the last level followed by $length(txnList)/arity$ number of nodes in the second last level of the tree and so on. The process is iterated until we are left with 1 node at the top, which serves as the root node of this tree.

2.2 Transactions

Each transaction has 2 major parts: Input & Output. The output of an earlier transaction acts as an input to a later transaction.

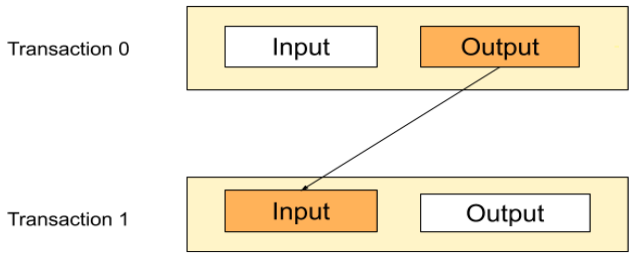

Figure 1: The Parts of a Transaction

We defined a class for Transactions where we store input as a list(because there can be multiple wallets of a payee) where each element of the list has 3 parameters -

`[(Previous Transaction's Hash Pointer, Index, Payee's Signature), ... , ..]`

The Output is a tuple of -

`(Amount, Receiver's Public Key)`

Here's an example on how transactions happen in our network.

Transaction Pointer	Index	Input	Signed by	Output
1	0	ϕ	-	- 25.0, PubKey(Alice)
2	0 1	1[0]	Alice	17.0, PubKey(Bob) 8.0, PubKey(Alice)
3	0 1	2[0]	Bob	8.0, PubKey(Carol) 9.0, PubKey(Bob)

`isValidTxn()`

A method that checks the validity of transactions by running Bitcoin Scripts, that verifies the signature, and also verifies the amount to be paid i.e. output value is $<=$ the current input of payee.

`hashVal`

A variable that stores the hash value (based on hash size provided in config) of the transaction.

`genesisBlockTxn = False`

A variable that indicates if a transaction belongs to the genesis block. It denotes a special transaction whose input list is empty. Default value is set to False.

2.3 Block

Each object of Block class has the following attributes:

`prevBlockPtr`

A pointer that points to the block before the current block in the Blockchain and it's value is fetched from *latestBlock* attribute of Blockchain class.

`prevBlockHash`

A variable that stores hash value of the previous block in the Blockchain.

`txnList`

A list of all the transactions stored on this block.

`merkleTreeRoot`

A Merkle tree is constructed on all the transactions present in the block and the hash pointer of the root is stored in this field.

`nonce`

Every node is generated some random value based on the provided size.

`hashVal`

Variable to store hash value for this block, which is computed on the following fields: $prevBlockHash$, $nonce$, $length(txnList)$, and $merkleTreeRoot$.

2.4 Blockchain

Blockchain class consists of the following two attributes.

`rootBlock`

It stores the hash pointer of the Genesis Block.

`latestBlock`

It stores the hash pointer of the lastly added block in the Blockchain. This value is updated each time a new block is added to the Blockchain.

2.5 Genesis Block Creation

```
def createGenesisBlock(self):
    bitcoinvalue = 1000
    // random transactions
    for _ in range(numberOfNodes):
        randomNo = randrange(0,numberOfNodes)
        // random wallet selection
        randkeyno = randrange(0,5)
        node = self.allNodes[randomNo]
        recverpubkeyHash =
           SHA256.new(hashlib.sha256(node.pubKey[randkeyno]).hexdigest().encode())
        t1 = Transaction([],[],bitcoinvalue,recverpubkeyHash,None,True)
        self.transactions.append(t1)
```

```
txn = self.transactions.copy()
// build Merkle Tree
rootMerkleTree = self.generateMerkleTree(txn)
// create Block
blk = Block(None,rootMerkleTree,self.generateNonce(),txn)
// Broadcast the block and add to the Block chain
```

We perform some random transactions where one of the wallets of a node is chosen randomly and is assigned 1000 bitcoins.

A block is created that stores this set of transactions along with the root of merkle tree and other attributes. This block is then broadcasted to every node in the network and each node updates it's local $UTXO$(Unspent Transaction Output) which can now act as input to further transactions.

This acts as the genesis block and it's hash pointer is saved in *rootBlock* attribute of the *Blockchain* class.

2.6 Incentives

The node that wins in Proof-of-work gets the chance to add it's block to the Blockchain and is given some incentive to do so.

We have implemented incentive as regular transactions with an empty input list. The value of incentive is 1 but can be changed through the *configurations* file($config.py$).

2.7 Node

Every Node is implemented as a separate thread. These nodes do the transactions and send the information to every other node. Every node tries to get the chance to add the block to the blockchain, the chance will be given to the node on the basis of the following proof of work which is kept simple for the implementation purpose and receive consensus from other nodes:

2.7.1 Proof-of-work

following steps are used for this:

- Every node generate a random value $Nonce$ which is used while creating the Blocks
- Block's $hashVal$ generated by the node should be least among all.

2.7.2 Consensus

Any node will give its consensus to a block only if every input of all the transactions are valid according to its local $UTXO$.

3 Security

Immutable Blocks: Blocks can't be modified once added to the Block chain. A block stores hash value of its previous block, so if any attacker tries to modify a block, it leads to change in the hash values of the whole chain.

Double Spend Transaction: Every node is checking the validity of the transactions and if the node is honest, it will not allow to push this transaction into its local transaction list therefore and this transaction won't be added to the block chain by any node. In case of dishonest nodes, when they will try to add such a transaction then this node won't get the consensus from other node thus it will not be able to add the double spend transaction to the block chain until at least 51% of the network is controlled by the dishonest nodes.

Table 1: Number of honest nodes(Threshold) required to maintain security and number of nodes to break security

Total nodes in system	**#Nodes to maintain security**	**#Nodes to break security**
10	5	6
50	25	26
100	50	51

4 Logs

All the Node ID's and wallet ID's used below are used only for the purpose of Log Printing. Logs printing is done in 3 parts:

- Node State Printing : printing the amount of all 5 wallets for every node
- Random Transactions Initiated : printing all the attempted transactions (sender, receiver, amount, valid/ Invalid transaction)
- Transactions Executed : printing all the valid transactions executed and added into the blockchain through a block(ID, IN/OUT , Wallet Id, Amount)
  - All the rows with same ID constitute 1 transaction
  - IN means input of transaction and OUT signifies output
  - i.j in Wallet Id implies j'th wallet of i'th node

```
-------- * * * * Genesis Block Added * * * * ---------

------------ * * * * Initial State of Nodes * * * * --------------
+---------+----------+----------+----------+----------+----------+
| Node ID | Wallet 0 | Wallet 1 | Wallet 2 | Wallet 3 | Wallet 4 |
+---------+----------+----------+----------+----------+----------+
|    0    |    0     |    0     |    0     |    0     |    0     |
|    1    |    0     |    0     |    0     |    0     |    0     |
|    2    |   1000   |   1000   |    0     |    0     |    0     |
```

3	0	0	0	0	0
4	0	0	0	0	1000
5	1000	0	0	0	0
6	0	1000	0	0	0
7	0	1000	0	0	0
8	0	1000	0	1000	1000
9	0	1000	0	0	0

-------------------- Random Transactions Initiated ---------------------

Sender	Receiver	Amount	Valid
6	4 . 0	11	True
8	4 . 1	42	True
4	8 . 0	33	True
5	7 . 4	45	True
1	7 . 1	27	False
2	6 . 4	30	True
0	2 . 2	37	False
3	5 . 1	12	False
9	5 . 2	28	True
0	3 . 3	11	False
0	5 . 3	43	False
1	0 . 3	18	False
3	2 . 4	24	False
0	2 . 4	20	False
3	7 . 1	24	False
0	4 . 1	17	False

-------- State of Nodes after addition of a New Block ---------

Node ID	Wallet 0	Wallet 1	Wallet 2	Wallet 3	Wallet 4
0	0	0	0	0	0
1	0	0	0	0	0
2	970	1000	0	0	0
3	0	0	0	0	0
4	11	42	0	0	967
5	955	0	28	0	0
6	0	989	0	0	30
7	0	1000	0	0	45
8	33	958	0	1000	1000
9	0	972	0	0	0

```
------------------Transactions Executed (Partial Table)----------

+----+--------+-----------+--------+
| ID | IN/OUT | Wallet ID | Amount |
+----+--------+-----------+--------+
| 0  |   IN   |   6 . 1   |  1000  |
| 0  |  OUT   |   4 . 0   |   11   |
| 0  |  OUT   |   6 . 1   |  989   |
| 1  |   IN   |   8 . 1   |  1000  |
| 1  |   IN   |   8 . 3   |  1000  |
| 1  |  OUT   |   4 . 1   |   42   |
| 1  |  OUT   |   8 . 1   |  958   |
| 1  |  OUT   |   8 . 3   |  1000  |
+----+--------+-----------+--------+
```

4.1 Multi-transaction

If multiple inputs are associated with one bitcoin transaction, this means that the amount being sent is coming from multiple bitcoin wallet addresses. this represents the multi-transaction Following Transaction shows the multi-transaction in Bitcoin as there are multiple inputs for the sender(node id 4 –> wallets 0,1,4), sending the amount of 41 bitcoins to the receiver(wallet 1 of node id 5):

```
+----+--------+-----------+--------+
| ID | IN/OUT | Wallet ID | Amount |
+----+--------+-----------+--------+
| 1  |   IN   |   4 . 0   |   11   |
| 1  |   IN   |   4 . 1   |   42   |
| 1  |   IN   |   4 . 4   |  967   |
| 1  |  OUT   |   5 . 1   |   41   |
| 1  |  OUT   |   4 . 1   |   12   |
| 1  |  OUT   |   4 . 4   |  967   |
+----+--------+-----------+--------+
```

5 Experiments

The main purpose of these experiments is to compare the transaction time (time taken to add the transaction to the Blockchain, as after the block gets added to the chain then only transaction is being finalised) and Block size (the memory used to store the transactions in terms of the Block including hash Size, Merkle Tree Size and Transaction size on the Blockchain) with the varying values of the different parameters used. Experiments has been performed considering the following parameters:

- Nonce Size
- Number of Nodes
- Hash Size
- Arity of the Merkle Tree
- Number of transactions in a Block

5.1 Experiment-1

This experiment shows the varying transaction time with respect to the value of the Arity of the Merkle Tree.

Fixed Parameters for this experiment:

- Nonce Size - 16
- Number of Nodes - 20
- Hash Size - 256
- Number of Transactions - 10

5.1.1 Results

The results obtained are summarized in the graph below-

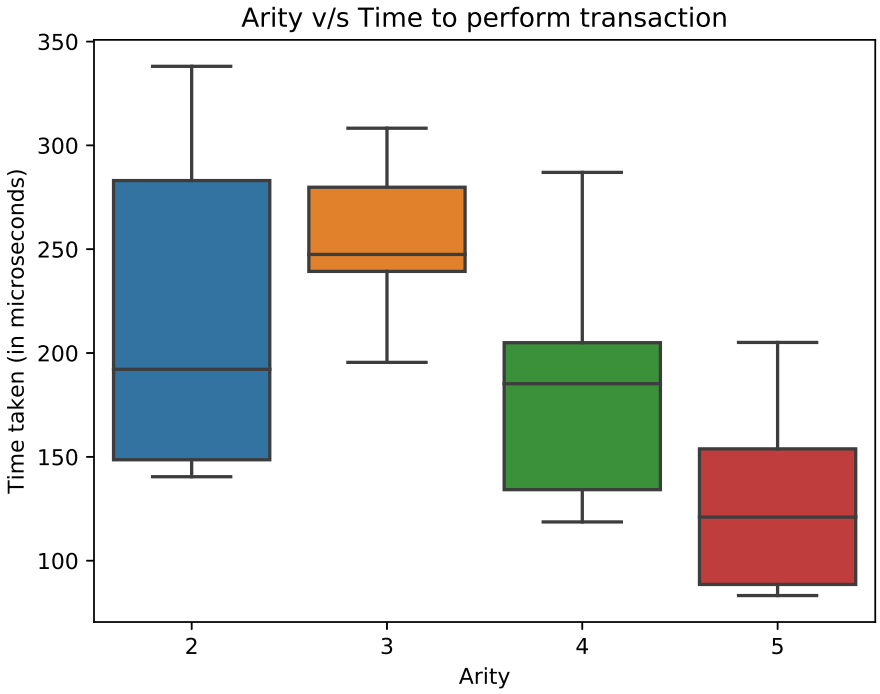

Figure 2: Box plot of the transaction time with varying arity

5.1.2 Observations

As the arity is increased, more number of transactions are merged to form a node of merkle tree. This in turn reduces the height of the tree, thereby, reducing the number of nodes present in the tree. Lesser nodes imply lesser computation and overhead of hash involved. Therefore, for a fixed number of transactions, transaction time has decreased with arity.

5.2 Experiment-2

This experiment shows the varying transaction time with respect to the value of the HashSize.
Fixed Parameters for this experiment:

- Nonce Size - 16
- Number of Nodes - 20
- Arity - 3
- Number of Transactions - 10

5.2.1 Results

The results obtained are summarized in the graph below-

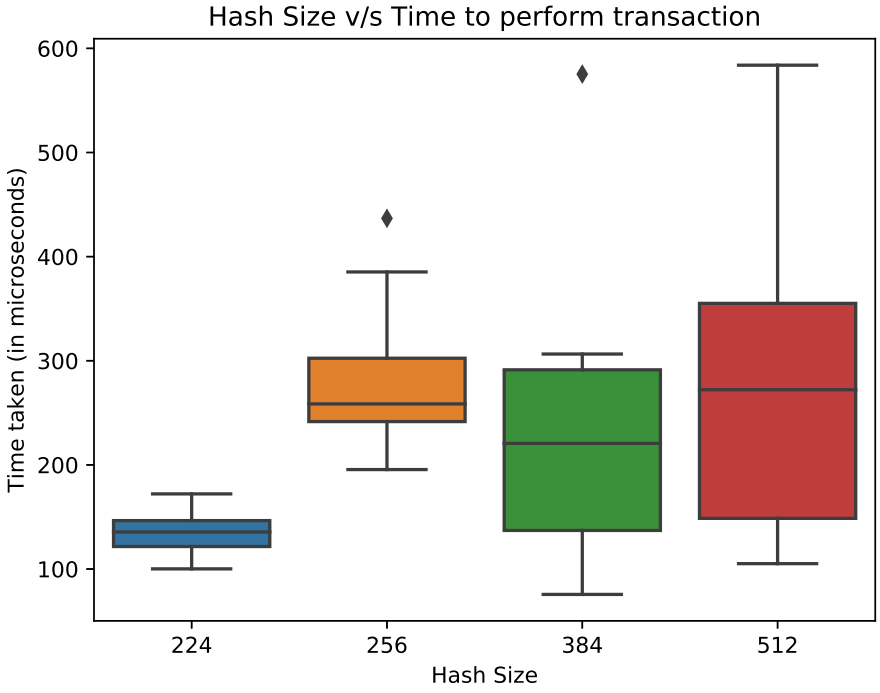

Figure 3: Box plot of the transaction time with varying HashSize

5.2.2 Observations

The hash size is used to compute hash values of transactions, merkle tree nodes and the block. As the hash size increases, the complexity of computing hash at each step is increased which then leads to increase in the transaction time.

5.3 Experiment-3

This experiment shows the varying transaction time with respect to the value of the Nonce Size.
Fixed Parameters for this experiment:

- Arity - 3
- Number of Nodes - 20
- Hash Size - 256
- Number of Transactions - 10

5.3.1 Results

The results obtained are summarized in the graph below-

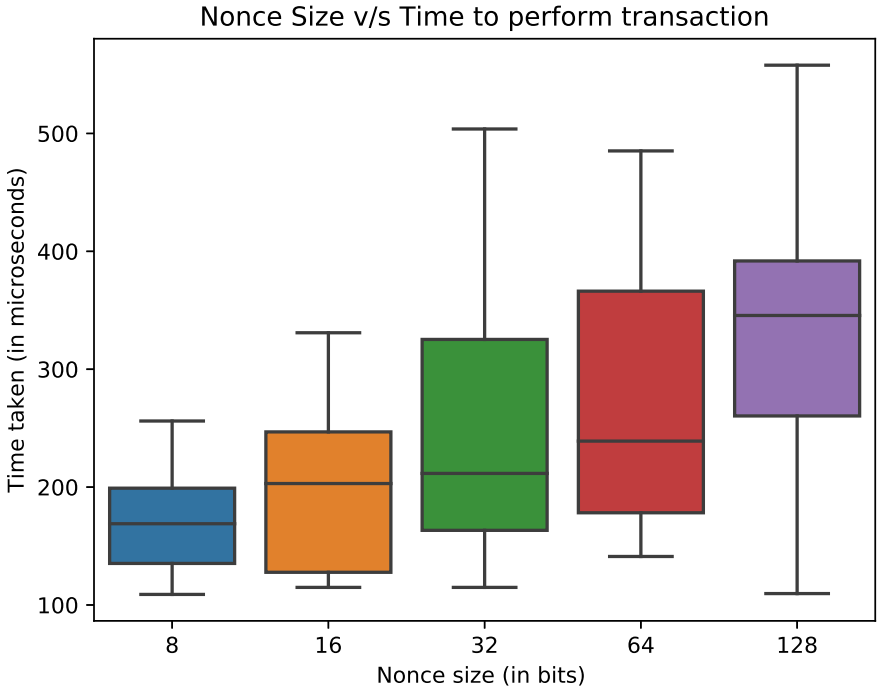

Figure 4: Box plot of the transaction time with varying Nonce Size

5.3.2 Observations

The value of nonce is generated randomly in the range $[0, 2^{sizeOfNonce}]$. As the number of bits increase, the range increases exponentially. It increases the time complexity involved in the proof-of-work. Also, larger value of nonce results in increased computation of hash values which further increases the transaction time.

5.4 Experiment-4

This experiment shows the varying Block Size having different number of transactions with respect to the value of the arity.

Fixed Parameters for this experiment:

- Nonce - 32
- Number of Nodes - 50
- Hash Size - 256

5.4.1 Results

The results obtained are summarized in the graph below-

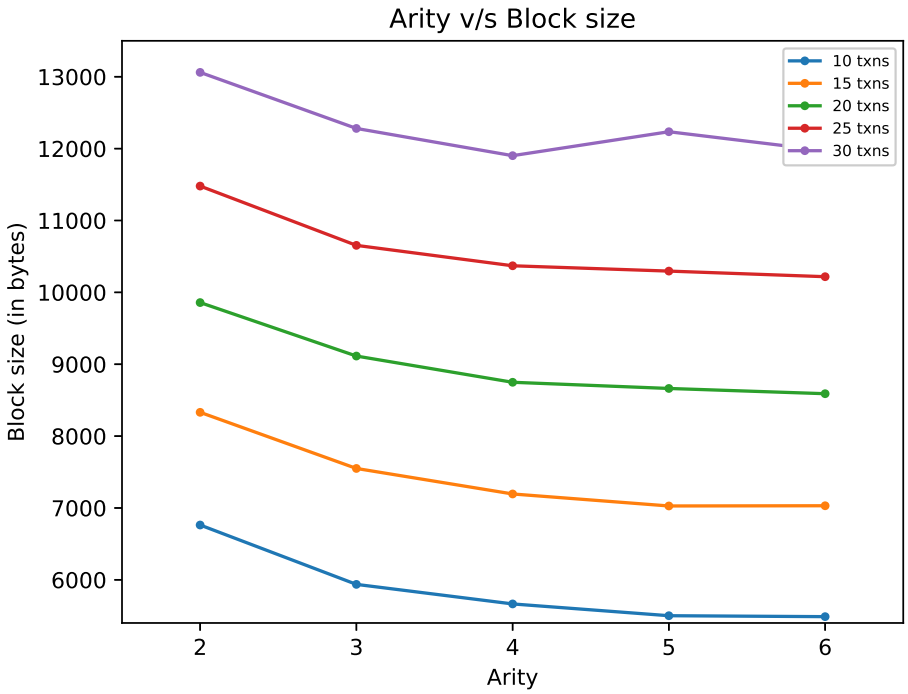

Figure 5: Block Size having different number of transactions with varying arity

5.4.2 Observations

A block stores the list of all transactions and the Merkle Tree constructed over them along with all other attributes explained above. As the arity increases, the number of nodes in Merkle Tree decreases and the space complexity to store the tree is reduced. We ran the experiment for different number of transactions and observed this trend.

The value for k (i.e. the number of transactions in a block) can be computed on the basis of the Block Size. We can limit the maximum size of the block to be added on the block chain and can compute the value of k.

5.5 Experiment-5

This experiment shows the varying Block Size having different number of transactions with respect to the value of the Hash Size.

Fixed Parameters for this experiment:

- Nonce - 32
- Number of Nodes - 50
- Arity - 4

5.5.1 Results

The results obtained are summarized in the graph below-

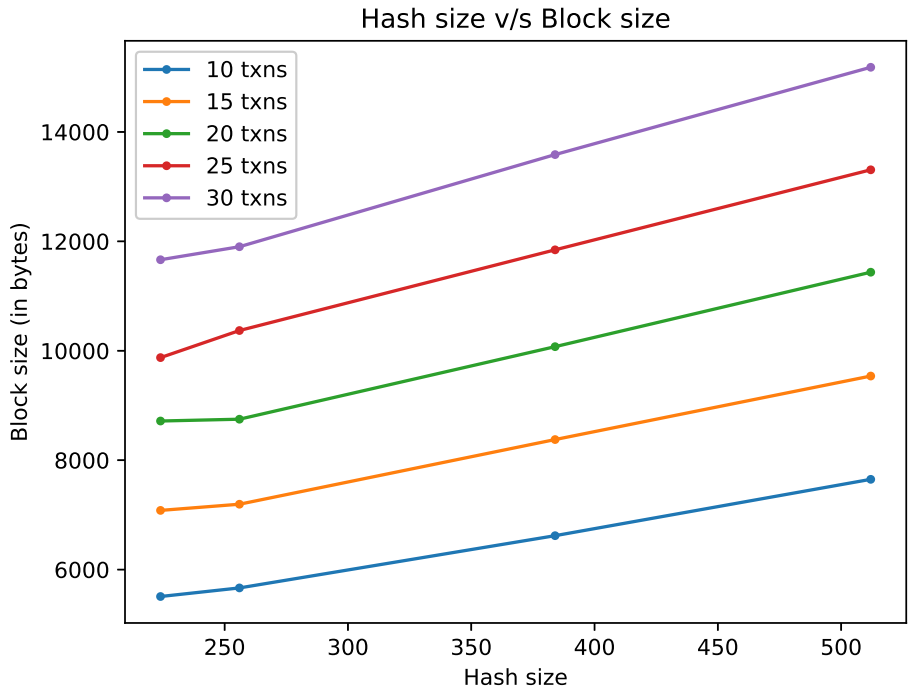

Figure 6: Block Size having different number of transactions with varying HashSize

5.5.2 Observations

As the hash size increases, the space complexity to store a block increases. This is due to the fact that a block stores the hash value of every transaction, every node of merkle tree and itself. Higher hash size means more memory required to store the block.

5.6 Experiment-6

This experiment shows the varying Block Size having different number of transactions with respect to the value of the Nonce Size.

Fixed Parameters for this experiment:

- Arity - 4
- Number of Nodes - 50
- Hash Size - 256

5.6.1 Results

The results obtained are summarized in the graph below-

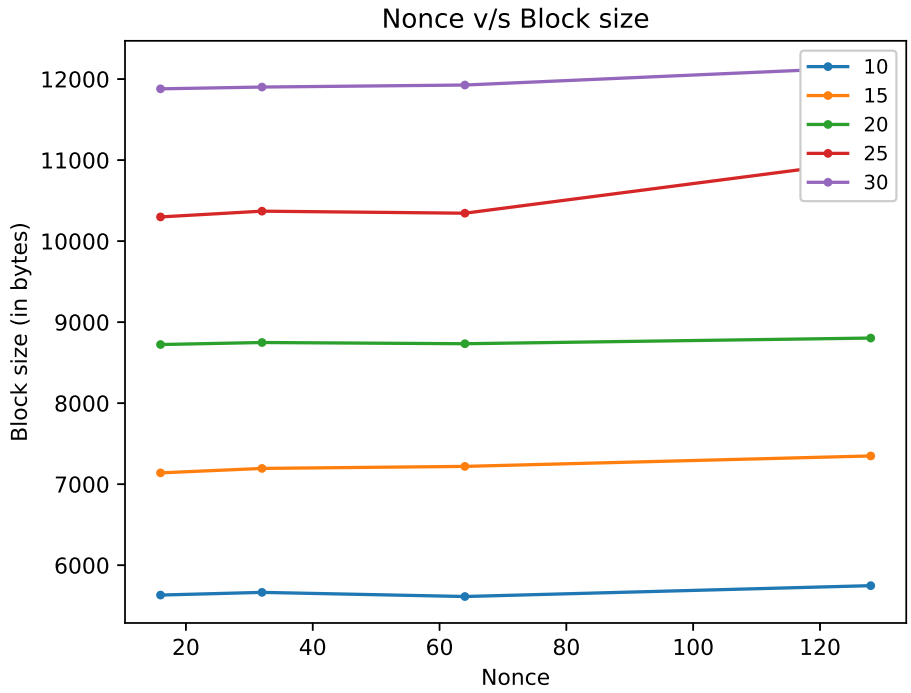

Figure 7: Block Size having different number of transactions with varying Nonce Size

5.6.2 Observations

The value of nonce is generated randomly in the range $[0, 2^{sizeOfNonce}]$. Every block stores one nonce value and therefore we see a minor effect in block size with increasing nonce value.

5.7 Experiment-7

This experiment shows the Transaction time with respect to the varying number of nodes in the network. The number of transactions in a block is in the range [14-25].

Fixed Parameters for this experiment:

- Nonce - 32
- Arity - 4
- Hash Size - 256

5.7.1 Results

The results obtained are summarized in the graph below-

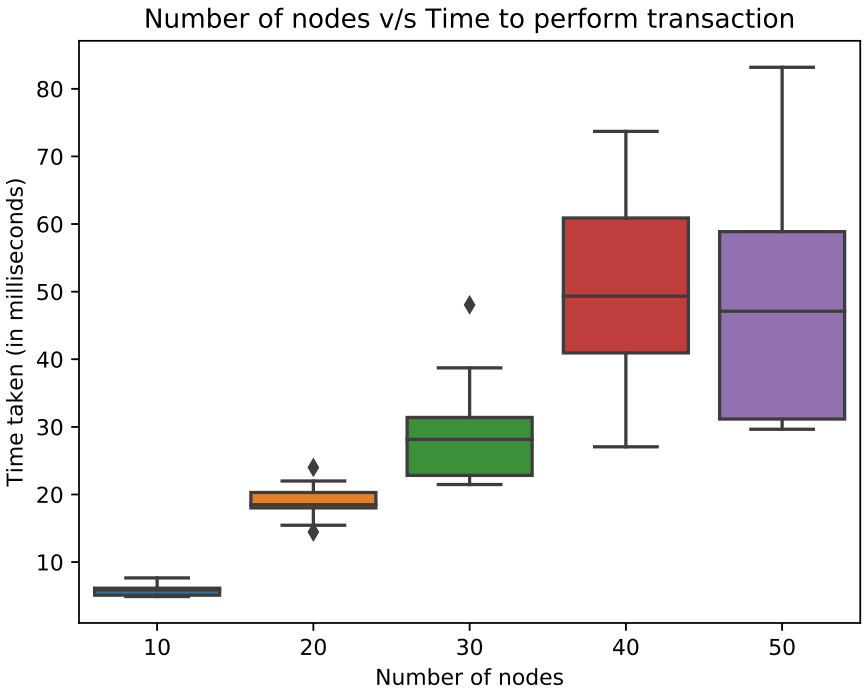

Figure 8: Transaction time with varying number of nodes

5.7.2 Observations

The transaction time has increased with increasing number of nodes in the network. This is because of the fact that more nodes imply more time taken to get consensus. Time complexity involved in proof-of-work also increases.

6 How to run code

The values of hyper-parameters can be adjusted through the config.py file. To run the code, type:

```
$ python3 Node.py > log.txt
```

The code runs infinitely as a bitcoin system is supposed to do. Random transactions would keep taking place and the nodes winning hash puzzle will keep on adding blocks to the block chain.

To view the logs after addition of 1-2 blocks in the block chain, force quit the program "$Ctrl + C$" after 2 minutes of execution.

7 References

- Bitcoin: A Peer-to-Peer Electronic Cash System by Satoshi Nakamoto
- Bitcoin and CryptoCurrencies Technology course by Coursera
- Internals of Transactions in Bitcoin