Proof Of Work Consensus

CSE 535: Asynchronous Systems
Stony Brook University
December 10th, 2018

Project Information

Project Topic	Implementation of Proof of Work Consensus in Dist Algo
Technology	DistAlgo and Python
Team Members	Arun Swaminathan (SBU ID:112044697)
	Hardik Singh Negi (SBU ID:111886786)
	Shubham Jindal (SBU ID:112129688)
Github	https://github.com/unicomputing/proof-of-work-consensus-py

1. Introduction

Since its inception in 2008 the blockchain technology has found its applications in several sectors ranging from cryptocurrencies to healthcare. At the very core, this technology is a distributed ledger that works asynchronously in a trustless environment to handle large amounts of data efficiently. Due to its decentralized nature, the blockchain requires the use of robust consensus algorithms to perform desired actions. Several efficient and robust consensus algorithms like proof of work (PoW), proof of stake (PoS), PBFT etc. have been leveraged by existing blockchain implementations.

In this project, we plan to explore, implement and analyze the proof of work (PoW) consensus algorithm. For the implementation purposes, we plan to leverage the DistAlgo[1] language framework as it will enable us to focus on the high-level implementation and analysis of the algorithm, as the framework will itself handle the nuances of distributed implementations.

2. Problem

- The primary goal of this project is to implement the proof of work consensus algorithm in blockchain using the DistAlgo framework. According to the best of our knowledge, no such implementation exists in DistAlgo.
- Understand the concepts of Blockchain and fixed several issues with the existing implementation given by Amitai Porat et. al [2][3]
- Analyze the performance of the system and perform correctness testing as a secondary goal.
- As an extra feature, we have added visualization of the Blockchain using D3.js.

2.1 Input

The input of the system will consist of transaction ledger, generated by recording different transactions among different user nodes.

2.2 Output

The ideal output generated by the system is as follows:

- The miners present in the blockchain verify the transactions and the consensus is reached without violating any correctness property.
- Blocks of pending transactions are added to the blockchain.

3. State of the Art

The proof of work system was invented by Cynthia Dwork and Moni Naor and presented in a journal article[4] as a way to deter denial of service attacks and spams. However, in 2008 Satoshi Nakamoto leveraged this as a consensus algorithm for Bitcoin cryptocurrency blockchain[5][6]. Since then several other cryptocurrencies like Ethereum[7][8] have used PoW as a defacto consensus algorithm for their blockchains.

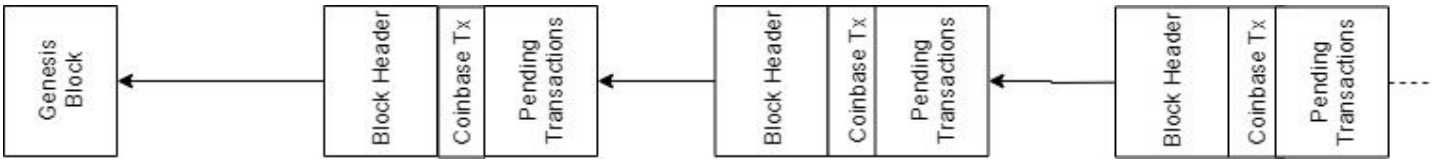
For the project, we focused on papers and implementations more centered on the PoW consensus and we found the implementation described in the paper by Amitai Porat et. al useful for our purpose and for the conceptual understanding we referenced Bitcoin Book.

Moreover, we also referenced following existing implementations for developing our system:

1. Implementation of Proof-Of-Work consensus protocol using Python (Ver. 2.7) by Amitai Porat et. al.
   This is a proof of work consensus implementation based on Ethereum platform developed in Python.
2. Justblockchain by Koshik Raj[10]
   This is a Python-based interactive implementation which allows users to interact with the blockchain via terminal.

4. Implementation Design

This design document explains the architecture and class module structure of our project. It starts by explaining the design of the blockchain and its constituent blocks. Then it describes the implementation of the transaction ledger in form of Merkle Tree and finally concludes with the class layout of miners and nodes involved in the blockchain.



A simple layout of Blockchain(without forks)

4.1 Overview

In this section, we will layout the basic implementation details of our implementation system and its components. These are as follows:

- **Block:** This is a constituent element of the blockchain we are implementing. Its implementation has properties like index, header, data etc. It is implemented in class Block(object) of our system. Each block within the blockchain is identified by a double hash, generated using the SHA 256 cryptographic hash algorithm on the header of the block. Each block also references a previous block, known as the *parent* block, through the "previous block hash" field in the block header.

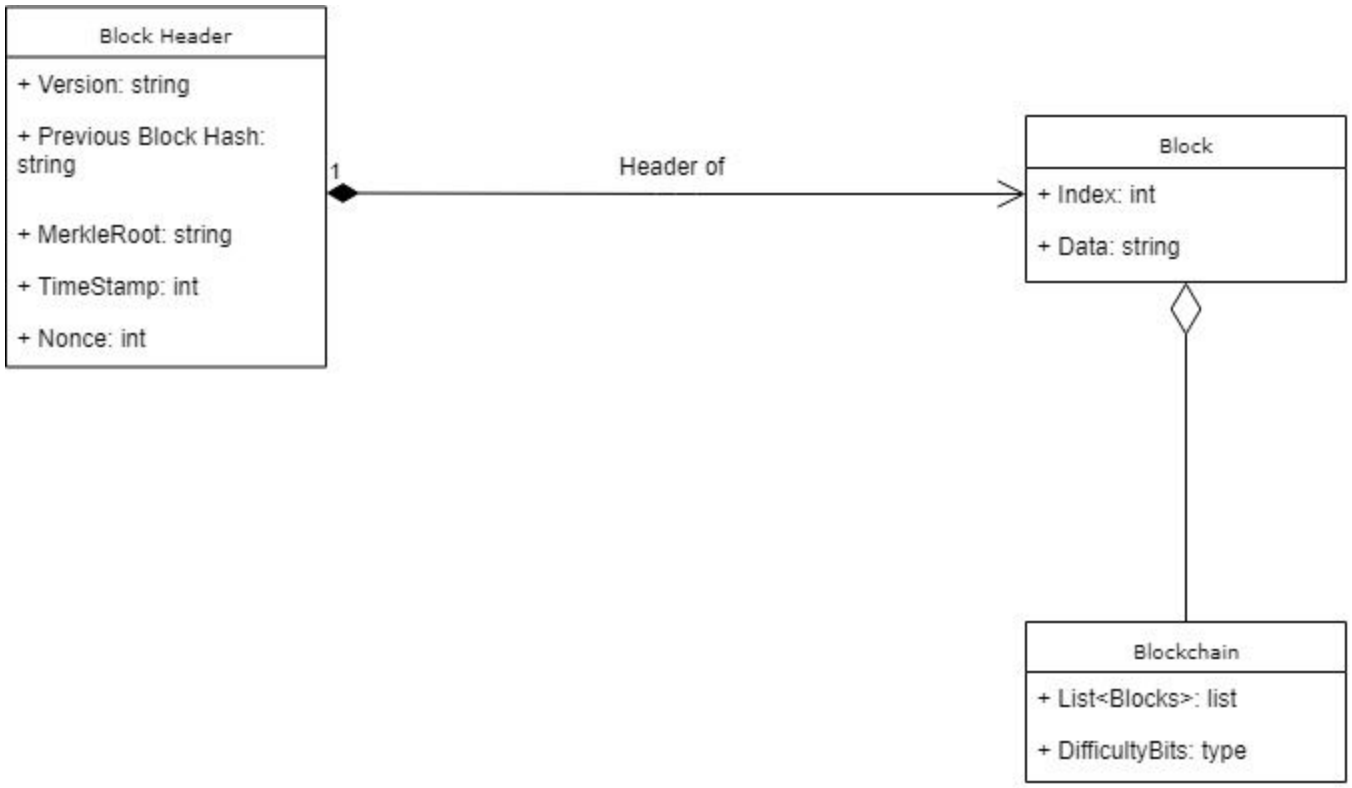
- **Blockchain:** The blockchain data structure is an ordered, back-linked list of blocks of transactions. The block object forms a constituent of a blockchain. Blockchain maintains the ledger of transaction blocks as a list and the first block is called Genesys Block. We implemented blockchain using class Blockchain(object), this class has methods to manage the blockchain, call block miners and most importantly apply proof of work consensus algorithm.

- **Transactions:** This is the implementation of the transaction ledger which contains transaction history of the blockchain. This is implemented in using class Transaction(object) in Merkle Tree form for ease of access.

- **Nodes and Miners:** Nodes are the sites participating in the blockchain and miners perform the transaction ledger validation. These are implemented in class Miner(process) and class Node(process).

In the upcoming section, we will describe the implementation structure and design of these classes along with a simple architectural layout.

4.2 Blockchain Architecture



Architectural Diagram of Blockchain and Block Class

4.3 Block

This section describes the API layout of Block Class with important properties and parameters.

class Block(object)

Parameters	Use
index	Index of the block in the blockchain
data	Data related to the transactions
header	Block header has parameters like timestamp, previous_hash, nonce, merkle_root

Properties/Methods	Use
blockhash(self)	Other Block instance

__eq__(self, other_block)	Check if the other_block is equal to the given block or not

Following is the list of parameters in the **Block Header**

Parameters	Use
Version	The version of the Blockchain
previous_hash	Hash of the previous block
timestamp	Timestamp of the block generation
merkel_root	Contains the root of the Merkel tree based transaction ledger
nonce	Nonce generated from Proof Of Work

4.4 Blockchain

This section describes the API layout of Blockchain Class with important properties and parameters.

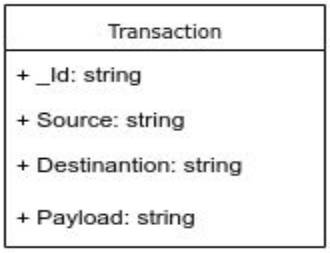
class Blockchain(object)

Parameters	Use
_chaingraph	Defaultdict used to implement blockchain having forks
blockmap	Maps blockhash to the block object
difficulty_bits	Difficulty level for the nonce calculation using Proof Of Work Consensus algorithm
primary_blockchain_block	Latest block in the longest chain
max_height	Length of the longest chain

Properties/Methods	Use
get_latest_block(self)	Return the latest block in the blockchain list
create_block(self, block_data)	Create a new block with json provided in block_data parameter

add_block(self, transactions, block)	Adds the block to the longest chain
block_ancestory(self, block)	Returns the path to the block from the Genesis block

4.5 Transactions



This class represents transactions taking place in the blockchain.

class Transaction(object)

Parameters	Use
Id	Unique Transaction Id
source	Source of the transaction
destination	Recipient of the transactions
payload	Payload/Object involved in transactions

In our system, we have implemented the transaction ledger in form of a Merkle Tree.

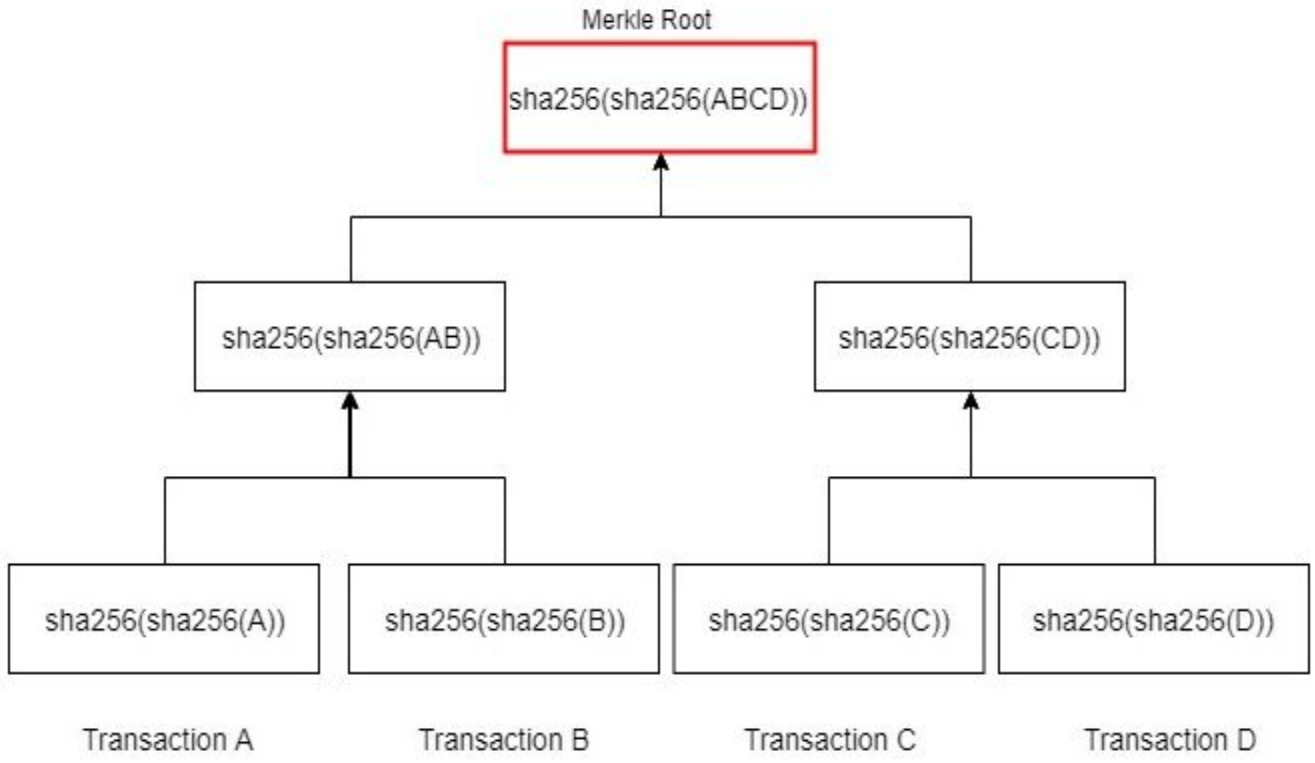
4.6 Merkle Tree

Each block in the bitcoin blockchain contains a summary of all the transactions in the block using a *Merkle tree*, also known as a *binary hash tree*, a data structure used for efficiently summarizing and verifying the integrity of large sets of data.

Every node in the Merkle Tree contains the hash of one transaction. Because the Merkle tree is a binary tree, it needs an even number of leaf nodes. If there is an odd number of transactions to summarize, the last transaction hash will be duplicated to create an even number of leaf nodes, also known as a *balanced tree*.

The transaction hashes are then combined, in pairs by simply adding the hash and double hashing it again, creating each level of the tree, until all the transactions are summarized into one node at the "root" of the tree.

In a **Merkle path** used to prove inclusion of a data element, a node can prove that a transaction K is included in the block by producing a Merkle path that is only four 32-byte hashes long (128 bytes total). The path consists of the four hashes. With those four hashes provided as an authentication path, any node can prove that HK is included in the Merkle root by computing four additional pair-wise hashes.



Merkle Tree for Transactions

4.7 Nodes

This class represents essential processes carried out by the node in a system.

Parameters	Use
_id	The unique id of Nodes
neighbors	Set of all neighboring nodes
blockchain	A local copy of the blockchain
count_live_miners	Number of miners still alive

Properties/Methods	Use

broadcast(msg)	Broadcast a transaction to all neighboring nodes
receive_block(block)	Receive completed blocks from neighbors and add to its blockchain

4.8 Miners

Miners are specialized nodes which also do the proof of work and update the block in the blockchain. They have the same properties and parameters as above but also contain the below:

Parameters	Use
pending_transactions	List of transactions received by the Miner to be added to the next block
orphan_block_pool	Blocks received by the miner whose parent blocks have not yet been received.

Properties/Methods	Use
proof_of_work(self, candidate_block)	Implementation of proof of work consensus algorithm in the blockchain, which generates a nonce for the new block
validate_transaction(transaction)	Validate the transaction received from the neighbors and add to the pending transactions.
validate_block(block)	Validate the transaction received from the neighbors and add to the longest chain in its blockchain.
receive_transaction(transaction)	Receive transaction from its neighboring nodes.
mine_block()	Mine the next block to be added to the blockchain
broadcast()	Broadcast a mined block to its neighbors
print_status()	Prints the final blockchain of the miner just before the miner dies.
operate_miner()	Await for a new broadcasted block to be received and parallelly mine a block at specific intervals.

4.9 Proof Of Work

We have implemented Proof Of Work in Blockchain class itself, however, in this section we will give some more details to proof of work implementation.

Proof of Work through mining secures the blockchain system and enables the emergence of network-wide consensus without a central authority. We have used SHA 256 as a hashing algorithm to generate a nonce for the new block. The difficulty_bits, control the difficulty of the hashing problem used to generate a nonce for a new block. Difficulty bits help in setting a target and for generating new block we need to loop over the nonce space of 4 billion to find a valid nonce which generates a valid nonce for the new block. Increasing the difficulty by 1 bit causes a doubling in the time it takes to find a solution.

4.10 Visualization

The visualizer uses the force layout from D3.js to generate a collection of nodes and links aka blocks which make the blockchain. The current code assumes 5 miners i.e 5 blockchains and creates the layout for each of them.

These nodes can be **dragged** around according to the users convenience to analyze forks and other properties.

On **hovering** the mouse on each node, we can see the transactions and its parent hash.

5 Implementation Details

5.1 Files Included

We have implemented the above-described design as following class package:

- **block.py:** This has class and methods to generate, manipulate and verify the blocks of the blockchain.
- **blockchain.py:** This file has the class implementation of the blockchain system.
- **miner.da:** This file has the implementation of miners of the blockchain
- **node.da:** These are the simple user nodes in the blockchain.
- **transaction.py:** This file has the class to generate transaction objects.
- **merkletree.py:** This file has Merkle tree implementation which is to be used for transaction ledger.
- **constants.py:** This has configuration constants like DIFFICULTY_BITS, MAX_BLOCKCHAIN_LENGTH.
- **visualization.html:** Blockchain visualization file.

GitHub Link of the Implementation: https://github.com/unicomputing/proof-of-work-consensus-py

5.2 Implementation Challenges

During the implementation we faced the following challenges:

Challenge 1: How to parallelly check for new received blocks from the other miners as well as mine a new block. Ideally, if a miner has received a valid block, it should **abandon** the mining for the current block, this was not happening in the existing solution.

Solution: We shifted proof of work from block.py to miner.da so that for each iteration, the control would check for a received block in the received array. We did this by adding yield point in the loop of the proof of work.

Challenge 2: How to enable forking in the blockchain. The existing implementations were using a linear list, which is not the correct data structure to implement forking.

Solution: We converted the blockchain list into a directed graph using adjacency list. This adjacency list has been implemented with the help of a defaultdict blockchain._chaingraph (**<blockhash, set<blockhash>>)** and also another dictionary blockchain.blockmap (**<blockhash, block_object>),**

Challenge 3: How to visualize and debug the blockchain easily. The existing implementation was just dumping the blockchain data on the terminal.

Solution: Although original scope did not include visualization, we added a whole visualization module custom made from scratch using D3, to help us easily debug the blockchain.

5.3 Insights

Language and tools: DistAlgo and Python
Code sizes and Number of Files:

Language	files	blank	comment	code
XML	7	0	0	748
Python	7	169	207	313
DAL	3	72	2	215
HTML	1	23	0	106
DOS Batch	1	8	1	26
Markdown	1	4	0	10
make	1	4	6	9
JSON	5	0	0	5
Bourne Shell	1	0	0	3
SUM:	27	280	216	1435

Number of commits: 93

6 Results

6.1 Blockchain visualization

Red nodes represent the Genesis blocks and the graphs represent the blockchain for each miner. We show 2 sample visualization of the blockchains in which we changed difficulty bits from 10 to 15.

a)

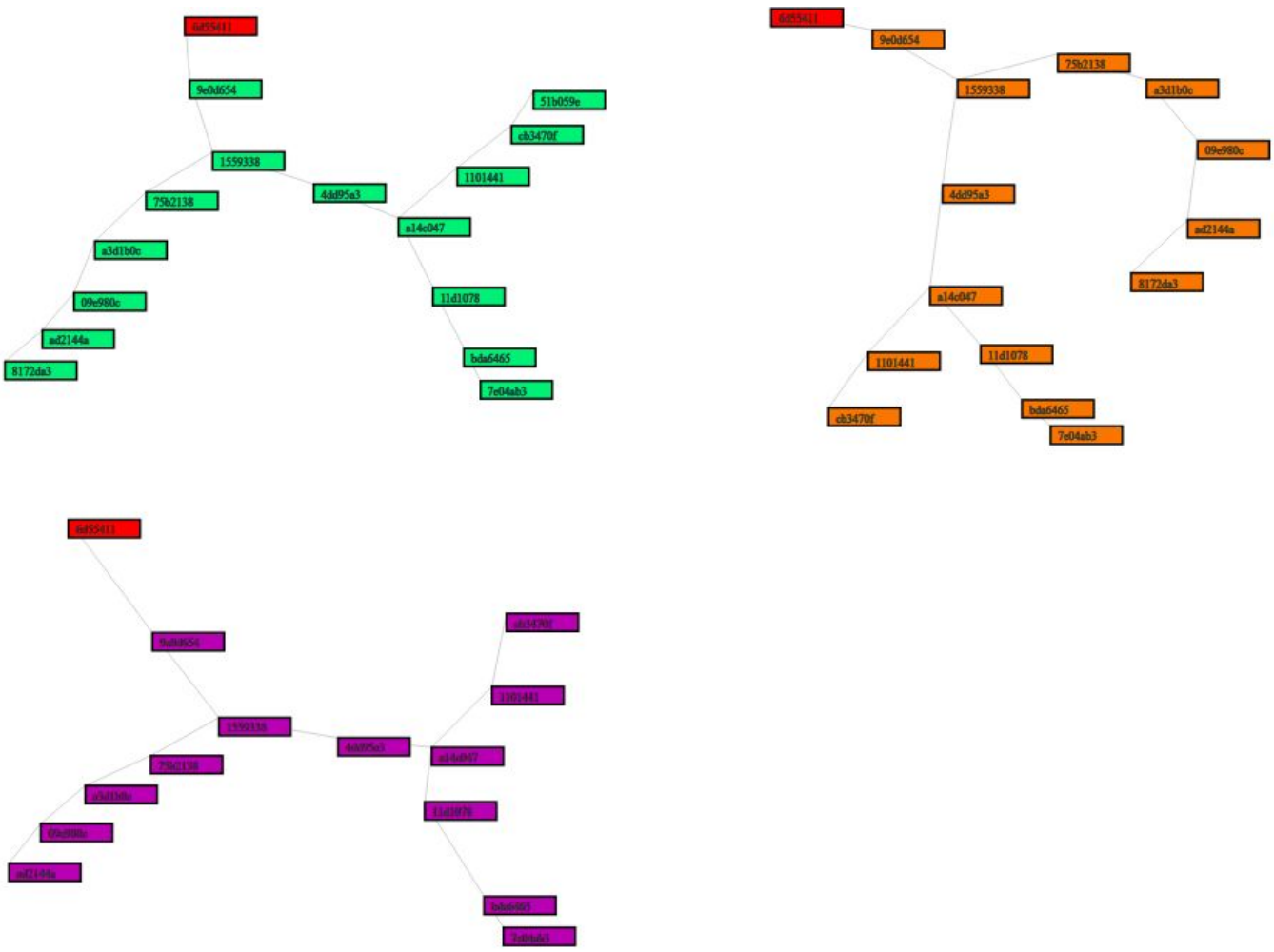


Fig. Miners: 3, Nodes: 4, Blockchain length: 8, Difficulty bits: 10

Analysis: We observed that in this case 15 blocks were mined since the difficulty was low, miners had a higher chance of mining the blocks simultaneously.

b)

Fig. Miners: 3, Nodes: 4, Blockchain length: 8, Difficulty bits: 15

Analysis: In this case 7 blocks were mined since the difficulty was high, any one of the miner would compute the current block first and broadcast it, making other miners to abandon their computation. So, the blocks had a very low chance of mining the blocks simultaneously.

c) Miners: 5, Nodes: 2, Blockchain length: 25, Difficulty bits: 15
Rather than having a snapshot, we added a mov video file to show you how big and complex blockchain was made by the 5 miners. The file name is m5_d15_l25.mov. It had just one fork as seen in the video.

d) Miners: 5, Nodes: 2, Blockchain length: 25, Difficulty bits: 10
Rather than having a snapshot, we added a mov video file to show you how big and complex blockchain was made by the 5 miners. The file name is m5_d10_l25.mov. This had multiple forks.

6.2 Performance and Testing Metrics

We know that, from our study, higher branching factor/ higher forking reduces the performance of a blockchain. Existing realtime blockchains dynamically adjust the difficulty bits to reduce these factors. Below we have calculated these factors by changing the difficulty bits statically.

We also performed testing and benchmarking of the implemented system on the following metrics:

- The system must not violate correctness property and must also be live. (This can also be observed from the visualization)
- Average time to mine a block.
- Average branching under various levels of difficulty
- Maximum secondary chain length on varying difficulty bits

Observing various parameters for varying Difficulty Bits

Total Miners - 5
Max Blockchain Length - 25 (blocks)
Total Runs: 5

Difficulty Bits	Avg Time (s)	Fastest Miner (s)	Slowest Miner (s)	Avg. Stale Blocks Mined
5	0.0038	0.0026	0.006	10
10	0.062	0.015	0.103	33
15	0.782	0.661	1.01	3
18	3.149	1.187	4.457	0
20	23.471	4.272	38.48	0

Observations:

- For the case of 5 Difficulty Bits, we got stale blocks to be 0 for several runs, as each miner had mined it's own primary chain due to lower value of difficulty.
- From the table it can be observed that as the difficulty bits increase the stale block count tends to 0 as the chance of several miners mining a valid block at the same time is quite less.
- Since the feature for dynamically adjusting the difficulty bits is not present so the consensus was not attained for lower values of the difficulty bits.

Observing Highest Branching Factor for varying Difficulty Bits

Difficulty Bits (Miners = 5, Blockchain Length = 10)	Highest Branching Factor
5	4
10	4

15	1
18	1
20	1

Observing Highest Branching Factor for varying no of Miners

No. of Miners (Difficulty Bits = 10, Blockchain Length = 10)	Highest Branching Factor
3	3
5	4
7	5
9	5
11	6
15	6

Observing Fork Length for varying Difficulty Bits

Difficulty Bits	Fork length
5	4
10	2
15	1
18	1
20	1

7 Future Work

Following tasks can be performed as an extension to this project:

1. Simulate increase in processing power of the different miners, and dynamically adjust difficulty.
2. Make multiple types of miners with different iteration logic to find the nonce for the proof of work.
3. Highlight forking and show more statistics in the visualization of blockchain.

8 References

[1]Liu, Annie et. al. “DistAlgo Language”, https://github.com/DistAlgo September 2018
[2]Porat, Amitai et. al. “ Blockchain Consensus: An analysis of Proof-of-Work and its applications”, http://www.scs.stanford.edu/17au-cs244b/labs/projects/porat_pratap_shah_adkar.pdf, December 2017
[3]Porat, Amitai et. al. “ Blockchain Consensus: An analysis of Proof-of-Work and its applications”, https://github.com/shahparth95/CS244Bproject, December 2017
[4]Dwork, Cynthia; Naor, Moni. "Pricing via Processing, Or, Combatting Junk Mail, Advances in Cryptology". *CRYPTO'92: Lecture Notes in Computer Science No. 740*. Springer: 139–147. 1993
[5]Nakamoto, Satoshi. "Bitcoin: A peer-to-peer electronic cash system." (2008).
[6]Mastering Bitcoin, https://github.com/bitcoinbook/bitcoinbook, November 2018
[7]Ethereum Homestead, http://ethdocs.org/en/latest/introduction/index.html, 2018
[8]Ethereum, https://github.com/ethereum, 2018
[9]Olivera,Matias “Interactive blockchain built with Node.js”, https://github.com/olistic/simplechain, April 2018
[10]Raj, Koshik “Justblockchain”, https://github.com/koshikraj/justblockchain, March 2018