Noobcash

Georgios Kyriakopoulos
School of ECE, NTUA
Department of Computer Science
Athens, Greece
el18153@mail.ntua.gr

Eleni Tsertou
School of ECE, NTUA
Department of Computer Science
Athens, Greece
el18165@mail.ntua.gr

Serafeim Tzelepis
School of ECE, NTUA
Department of Computer Science
Athens, Greece
el18849@mail.ntua.gr

***Abstract*—The Noobcash project is a blockchain system designed to facilitate secure and decentralized transactions without the need for a central authority, implemented in Python [1]. The system achieves this by utilizing a distributed database, which ensures that all nodes within the network have access to the same information and can verify the validity of any transaction. As part of the project's development, several experiments were conducted to analyze the system's performance in terms of throughput and block times under varying conditions. These experiments involved testing different combinations of block capacity and mining difficulty. Additionally, the scalability of the system was tested by increasing the number of nodes within the network from 5 to 10. Through these experiments, valuable insights were gained into the system's behavior under different conditions, which can inform future improvements and developments in blockchain technology.**

***Index Terms*—blockchain, cryptocurrency, python, experiments, proof-of-work, throughput, block time, block capacity, mining difficulty, scalability.**

I. Introduction

Blockchain technology has become increasingly popular in recent years due to its ability to facilitate secure and decentralized transactions without the need for a central authority. One of the most well-known applications of blockchain technology is in the realm of cryptocurrencies, which rely on blockchain systems to enable users to transact with each other securely and efficiently.

The Noobcash project is a python-based implementation of a blockchain system that aims to provide a basic but functional example of a blockchain system. The system uses Proof-of-Work to ensure consensus between nodes, and it is designed to enable secure and decentralized transactions between users without the need for a central authority. Thorough explanation about the implementation of the system and how it works are provided in an upcoming section.

As part of the project's development, a series of experiments were conducted to analyze the system's performance under different conditions. These experiments included testing different combinations of block capacity and mining difficulty to determine the system's throughput and block times. Additionally, the scalability of the system was tested by increasing the number of nodes within the network from 5 to 10.

This paper is backed by a GitHub repository [2] [3], which contains the source code of the system, the scripts used for installation and testing, as well as analytics for the results.

II. Setup

In this section, we outline a brief explanation of the setup that was utilized to run the blockchain and the experiments.

We used the okeanos IAAS [4], which is GRNET's cloud service, for the Greek Research and Academic Community, to acquire 5 virtual machines for our study. Each machine has the same following specifications:

- CPU: 2 Cores - Intel® Xeon® CPU E5-2650 v3
- RAM: 2GB
- Disk: 30GB
- Operating System: Ubuntu Server 16.04.3 LTS

Those 5 machines were used to set up a 5-node cluster. We also set up a local network (192.168.2.0/24) so that the 5 nodes could communicate with each other. After setting up our machines, we used SSH in order to connect and have access to them and proceed with the configuration.

We then installed Python3.8 [1] by building it from source and pip [5] using a script for all nodes. At that point, we were ready to proceed with the implementation of our blockchain system in Python.

III. Implementation

We developed our blockchain source code by dividing it into several Python classes. The classes we used are:

- **Block**
- **Blockchain**
- **Node**
- **Transaction**
- **TransactionInput**
- **TransactionOutput**
- **Wallet**

On the following subsections, we will analyse each class separately.

A. Block

This class has the following instance variables:

- **index**: index of the Block
- **timestamp**: timestamp of the Block's creation
- **transactions**: list of the Block's transactions
- **nonce**: proof-of-work
- **previous_hash**: hash of the previous Block
- **hash**: hash of the Block

The following instance methods were also developed:

- **add_transaction_and_check**: adds a new transaction in the Block and checks if the Block has reached its capacity
- **hash_block**: calculates the hash of the Block

B. Blockchain

This class has the following instance variables:

- **blocks**: list of validated blocks in the chain

This class has no instance methods.

C. Node

This class has the following instance variables:

- **id**: id of the node
- **chain**: blockchain of the node
- **wallet**: wallet of the node
- **ring**: information about others (id, ip, port, public_key, balance)
- **filter_lock**: lock in order to provide mutual exclusion while filtering blocks
- **chain_lock**: lock in order to provide mutual exclusion while updating the chain
- **block_lock**: lock in order to provide mutual exclusion while updating blocks
- **unconfirmed_blocks**: queue that contains all the blocks waiting to be mined
- **current_block**: the block that the node currently fills with transactions
- **capacity**: max number of transactions in each block
- **stop_mining**: flag to stop mining when a confirmed block arrives

The following instance methods were also developed:

- **create_new_block**: creates a new block
- **register_node_to_ring**: registers a new node in the ring, called only by the bootstrap node
- **create_transaction**: creates a new transaction, after gathering the inputs from the utxos
- **add_transaction_to_block**: adds a transaction to a block, check if mining is needed and update the wallet and balances of participating nodes
- **broadcast_transaction**: broadcasts a transaction to the network, utilizing threads
- **validate_transaction**: validates an incoming transaction, by checking the signature, the inputs and the outputs
- **mine_block**: implements the proof-of-work algorithm
- **broadcast_block**: broadcasts a transaction to the network, utilizing threads
- **validate_previous_hash**: validates the previous hash of an incoming block
- **validate_block**: validates a block, by validating its hash and its previous hash
- **filter_blocks**: filters the queue of the unconfirmed blocks, by removing the transactions that are included in a mined block
- **share_ring**: shares your ring to a specified node
- **validate_chain**: validates all the blocks of a chain
- **share_chain**: shares your blockchain to a specified node
- **resolve_conflicts**: resolves conflicts of multiple blockchains, by keeping the longest chain when a new block that can't be validated is received

D. Transaction

This class has the following instance variables:

- **sender**: sender's public key
- **sender_id**: id of the sender
- **receiver**: receiver's public key
- **receiver_id**: id of the receiver
- **amount**: amount of nbc to transfer
- **total**: total amount that sender sends
- **inputs**: list of TransactionInput
- **id**: hash of the transaction
- **outputs**: list of TransactionOutput
- **signature**: signature of the transaction

The following instance methods were also developed:

- **convert_to_list**: list representation of a Transaction
- **hash_transaction**: calculates the hash of the Transaction
- **calculate_outputs**: computes Transaction outputs
- **sign_transaction**: signs the Transaction using a private key
- **verify_signature**: verifies the signature of a Transaction

E. TransactionInput

This class has the following instance variables:

- **output_id**: id of the TransactionOutput that is used as TransactionInput

This class has no instance methods.

F. TransactionOutput

This class has the following instance variables:

- **transaction_id**: id of the transaction
- **target**: target of the TransactionOutput
- **amount**: amount of nbc to be credited to the target
- **unspent**: boolean of whether this output has been used or not

This class has no instance methods.

G. Wallet

This class has the following instance variables:

- **private_key**: private key of the node
- **public_key**: public key of the node, also its address
- **transactions**: list that contains the transactions of the wallet

The following instance methods were also developed:

- **wallet_balance**: calculates balance of the wallet based on the utxos

IV. REST API AND CLIENT

Now, after introducing the classes and methods implemented for the blockchain, it is important to highlight the parts that bring the system to life. These include the endpoints we developed for the REST API, a simple client in the form of a CLI (command-line interface) and a script to test some sample transactions.

A. Endpoints

The REST API we developed utilizes the endpoints below:

- **register_node**: registers a new node in the network, called only by the bootstrap node
- **validate_transaction**: validates an incoming transaction
- **receive_transaction**: receives a transaction and adds it to a block
- **receive_block**: receives a block, validates it and adds it to the blockchain
- **receive_ring**: receives the ring from bootstrap
- **receive_chain**: receives the blockchain
- **send_chain**: sends your chain to another node
- **create_transaction**: creates a new transaction
- **get_balance**: gets balance of the node
- **get_transactions**: gets transactions of the last confirmed block
- **get_id**: gets id of the node
- **get_metrics**: gets metrics of the network

B. Noobcash Client

To develop a client for our blockchain, we opted for a simple CLI client, using the PyInquirer module [6]. The client has the following commands available:

- **new transaction**: create a new transaction, specifying the amount of NBC coins to send and the receiver's id
- **view last transactions**: print the transactions contained in the last validated block of the Noobcash blockchain
- **balance**: print the balance of the wallet
- **help**: explanation of the above commands

C. How to Run and Test

We have also implemented a `main.py` file that starts the REST API on the node and then, based on the command line arguments used when running it, initializes a node.

More specifically, when running the `main.py` file, the following arguments can be defined:

- **-p** `<port>`: port to listen on
- **-n** `<number-of-nodes>`: number of nodes in the blockchain
- **-c** `<capacity>`: capacity of a block
- **-b**: flag that is set if current node is the bootstrap

The first 3 arguments are required for every node, while the last one should only be set on the bootstrap node (the first one on the network).

In the case of the bootstrap node, it registers itself on the ring, creates the genesis block and the first transaction, adds it in the genesis block and then waits for other nodes to join. When another node initializes, it sends a request to the bootstrap node to be registered on the ring. When all nodes have joined, the bootstrap node broadcasts the ring, the blockchain up to that point and also sends each node 100 NBC coins.

Bear in mind that in the case of a network with 5 nodes, each machine needs to run the REST API once, using the `main.py` file on a port of choice (default 5000). However, if the number of nodes is 10, then each machine needs to run the REST API twice, on two different ports (e.g. 5000 and 6000). These 2 different ports are used to distinguish between the two nodes that need to run on the same machine.

Then, the noobcash client (CLI) can be used to interact with the blockchain (creating transactions, checking the balance etc), or a script that runs a few sample transactions may be executed, to test the system.

The client can be run by executing the `noobcash.py` file, specifying the **-p** `<port>` argument, which is the port that the REST API is listening on that node. The client will then ask the user to input a command, which can be one of the above mentioned commands.

The test script can be used on any node by executing the `run_test_files.py` file and will run 100 transactions sending NBC coins to other nodes. It requires a **-d** `<directory>` command line argument, specifying the path to the directory of the transactions and a **-p** `<port>`, specifying the port that the REST API is listening on that node.

V. Experiments

A. Metrics

We ran some experiments on our blockchain, testing different combinations of block capacity and mining difficulty. These tests were conducted to measure the system's performance using two metrics, throughput and block times. We also tested the scalability of our system, by running the blockchain with 5 and 10 nodes in the network.

B. Graphs

The results of the above tests can be summarized in the following graphs. The first graph is comparing the throughput for 5 and 10 nodes based on the different configurations of block capacity and mining difficulty. The second graph is doing the same but for block time. The next two graphs provide a clearer view of how the scaling of the system's number of nodes affects the throughput and block time.

C. Comments

Based on Figure 1, we can see that the throughput of the system increases as the block capacity increases, since less time is spend mining, because more transactions can fit into one block and mining is needed less frequently. As difficulty increases, the throughput decreases, since mining is now harder and requires more time to finish.

About Figure 2, it is clear that as difficulty increases, the block time will increase as well, considering that mining is harder and will certainly take up more time on average to find the right nonce and hash. Block time also increases as capacity increases, since now the block contains more transactions and all the operations on it, including calculating the hash, are expected to take up more time.

For Figure 3, in most configurations there is a small but noticeable increase in the system throughput from 5 to 10 nodes. When increasing the number of nodes, there are more

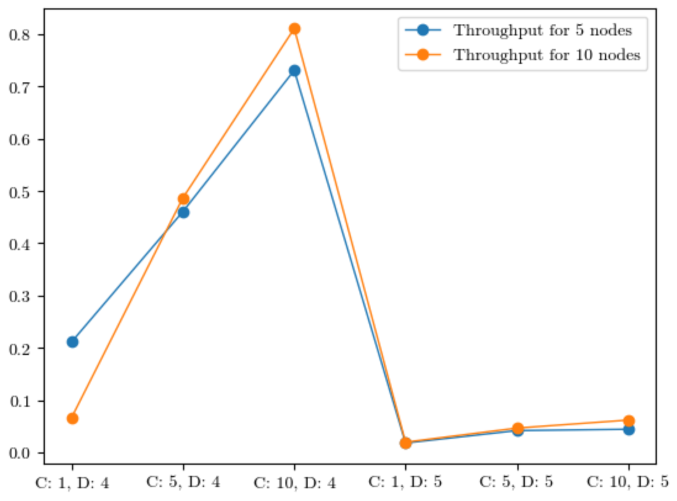

Fig. 1. Throughput relative to Capacity (C) and Difficulty (D)

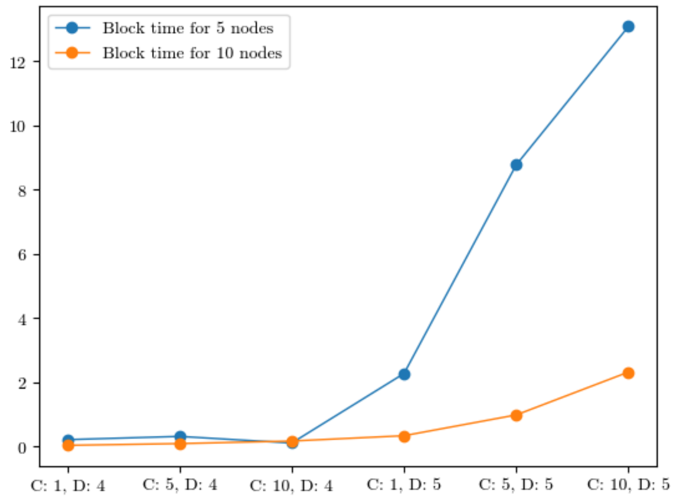

Fig. 2. Block time relative to Capacity (C) and Difficulty (D)

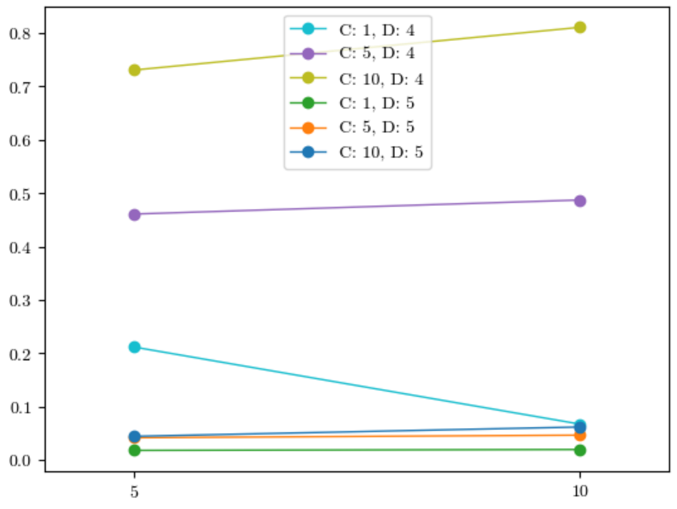

Fig. 3. Throughput relative to Number of Nodes

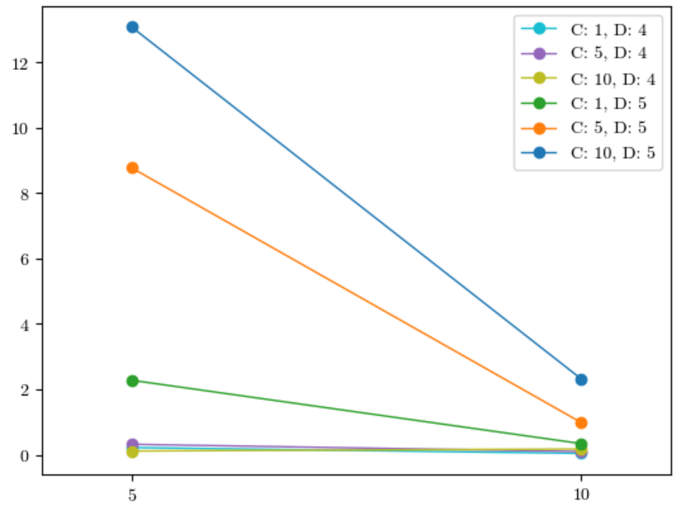

Fig. 4. Block time relative to Number of Nodes

miners available to help speed up the mining process and therefore increase the throughput of the system. However, with more nodes, there are also more blocks created and there is added network latency and communication overhead. So, an increase is possible and expected, but not to a great extent.

Finally, on Figure 4, block time is significantly reduced. That is more noticeable on the configurations with higher mining difficulty. This behavior is expected, when considering that there are more miners available to mine every block, which will lead to lower block time.

D. *Conclusion*

Based on the experiments conducted on the blockchain system, it is evident that the block capacity and mining difficulty have a significant impact on the system's throughput and block time. Increasing the block capacity can increase the throughput, while increasing the mining difficulty can decrease the throughput and increase the block time. Additionally, increasing the number of nodes in the network can slightly increase the throughput, but will also heavily reduce the block time, more so when the mining difficulty is higher.

Overall, these experiments highlight the importance of carefully optimizing the block capacity, mining difficulty, and network infrastructure to achieve optimal performance in a blockchain system.

References

[1] "Welcome to Python.org," Python.org. [Online]. Available: https://www.python.org/.

[2] G. Kyriakopoulos, E. Tsertou, S. Tzelepis, "geokyr/ntua-distributed-systems: Simple blockchain system project in Python, that records transactions between participants and ensures consensus using Proof-of-Work for the Distributed Systems course at ECE NTUA," GitHub. [Online]. Available: https://github.com/geokyr/ntua-distributed-systems.

[3] S. Tzelepis, "SerTze/ntua-distributed-systems: Blockchain project in Python. Noobcash is a simple blockchain system that records transactions between participants and ensures consensus using Proof-of-Work," GitHub. [Online]. Available: https://github.com/SerTze/ntua-distributed-systems.

[4] “ Okeanos,” okeanos Dashboard. [Online]. Available: https://accounts.okeanos.grnet.gr/.
[5] “Pip,” PyPI. [Online]. Available: https://pypi.org/project/pip/.
[6] “Pyinquirer,” PyPI. [Online]. Available: https://pypi.org/project/PyInquirer/.