Université de Liège

INFO8002-1
Large-scale data systems

Project : *Decentralized Blockchain key-value store.*

BULUT Stephan - s172244
HELD Jan - s165516
MUKOLONGA Jean-David - s170679
SIBOYABASORE Cedric - s175202

Academic year 2021-2022

To install the requirements needed to execute our codes, the line

```
pip install -r requirements.txt
```

has to be executed first, where `requirements.txt` contains `Flask` and `requests`.

To execute our code, the first node is launched with the following command line :

```
python app.y --p port --m miner --d difficulty
```

Then, following nodes can be added in the network as follows :

```
python app.py --p port --m miner --d difficulty --b bootstrap
```

1 Architecture

In this section, we will talk about the architecture of our blockchain implementation.

1.1 Blockchain implementation

Our blockchain implementation consists of 5 different classes. The Block, the Peer, the Transactions, the Callback and the Blockchain classes :

- **Transaction**
  This class contains all the information about a transaction. A transaction typically involves a key, a value, an origin (the host who puts it onto the storage) and a timestamp.
- **Block**
  This class contains all the information that we save inside a block. This class contains an index, an address, a list of transactions, a timestamp, a nonce, and a hash of the previous block to ensure immutability of the entire blockchain.
  Furthermore, as the data in each block is immutable, we implement a cryptographic hash function. To calculate the hash, we created the method `_hash` inside our class. Inside this method, we use the hash function SHA-256 to calculate the hash.
- **Callback**
  This class is responsible for remembering that a transaction has been added and waits until the transaction has been added to the blockchain. It contains the considered transaction and the blockchain.
- **Blockchain**
  The main functionality of the Blockchain class is that it contains the chain of blocks and the host who owns it. Furthermore, it contains a variable called difficulty which is a measurement of how difficult it is to mine a block. A high difficulty results in an increase of computing power to mine the same number of blocks but making the network more secure against attacks but also, making it slower.
- **Peer**
  The Peer class is the most important class as it contains all the information needed to maintain the correctness of our blockchain implementation. The Peer class contains : the address of the host, his peers, the blockchain, if the node is a miner, if the node is currently mining, his memory pool, if the node is ready to mine, two lists containing the peers who answered to the heartbeat signal and their counts, the historic of transactions, two lists containing the number of transactions and the time measured (for the experiment on the throughput), the time when the mining starts, and three components throughput,attacker and victim to perform

experiments.
A new node joins the network using the address of another node. This bootstrapping node will let the new node access its peer and ask the peer with the longest chain for its blockchain. During mining, to check if the hash of the candidate block is below a target, we check if its hash begins with a number of zero bits [1]. If that condition is not satisfied, the nonce field of the block is incremented to obtain a new hash. We describe more detailed implementations of Peer methods in the pseudo-codes in the [Annex] section.

An additional class called **PerpetuelTimer** was created to handle the `heartbeat` function.
We assume that when a node enters the network, no peer is corrupted.

We executed our codes on address `localhost:port` where port is the port given as argument.

1.2 Flask

In order to be able to use our blockchain implementation, we needed an interface on which multiple users could interact with each other. We decided to use Flask, which is a web application framework written for Python that enables us to create routes in a very simple manner. All our API calls can be found in the `app.py` script :

- / : Retrieves the data from the forms for the different requests (`put` which stores the value with the associated key, `retrieve` which searches the store for the latest value with the specified key which is given as argument, `retrieve_all` which retrieves all values historically recorded in the store for the key which is given as argument and `network` which displays the peers of the host).
- /**testing** : Allows to perform the experiments via `experiment.py`.
- /**peers** : Returns a list of all peers.
- /**addNewNode** : Adds the address of a node to the list of peers. The address is passed as a parameter in the URL.
- /**keyChain** : Returns the blockchain.
- /**memoryPool** : Returns the memory pool. The memory pool contains a list of transaction.
- /**addTransaction** : Adds a transaction to the current list of transactions and broadcasts it to the network.
- /**heartbeat** : Requests a heartbeat signal from the peer.
- /**addNewBlock** Adds a new block to the blockchain.

2 Experiments

In this section, we performed three different experiments in order to analyse the performance.

2.1 Resistance of the key-value store against faults

We implemented several mechanisms to make the key-value store resistant against faults.

Regarding transactions, we encountered a cycle problem due to broadcasting. Indeed, in the case where a miner who has a particular transaction in his memory pool creates a block encapsulating this

1. Satoshi Nakamoto, Bitcoin : A Peer-to-Peer Electronic Cash System, 2009

transaction, the node must first broadcast the transaction on reception. Then, as soon as the block is mined, the miner will remove the transaction from the memory pool. A peer node receiving the transaction will in turn broadcast the transaction. Thus, we implemented a `transaction_historic` list in order to discard transactions that have already been mined.
Likewise, to avoid cycle problems due to the broadcasting of mined blocks, we first check if the block is not in the current blockchain before adding it.
Regarding the memory pool, we simply remove transactions that are encapsulated in each block we add to the blockchain, while discarding already-received transactions thanks to the `transaction_historic`.

We performed unit tests thanks to `unitTest.py` to ensure modules functionalities.

2.2 Transaction throughput with respect to the difficulty level

We perform the throughput experiment thanks to the following command lines :

```
python app.py --p port --m miner --d difficulty --t True
python experiment.py --t True
```

where `-t` is the boolean argument denoting the wish for conducting the experiment showing the transaction throughput with respect to the difficulty level.

To measure the transaction throughput which corresponds to the number of transactions per second, we used Peer attributes *nb_ transactions* which is a list containing accumulated number of transactions, *timing* which is a list containing the corresponding timestamps and *start* which is the time at which the node starts mining.
To conduct the experiment, we instantiated a sole node in mining mode in the network. We execute a `experiment.py` script that automatically adds a new transaction with a different value every 1 second for 3 minutes (180 seconds). Each time a block is mined and is about to be added to the blockchain, the number of transactions inside that block is appended to *nb_ transactions* and the timestamp at which this appending occurs (<180 seconds) is appended to `timing`.
We repeat this process for difficulties ranging from 1 to 5 and obtain the graph in Figure 1 in which we plotted the number of transactions and the respective timestamps at which they are about to be added to the blockchain.

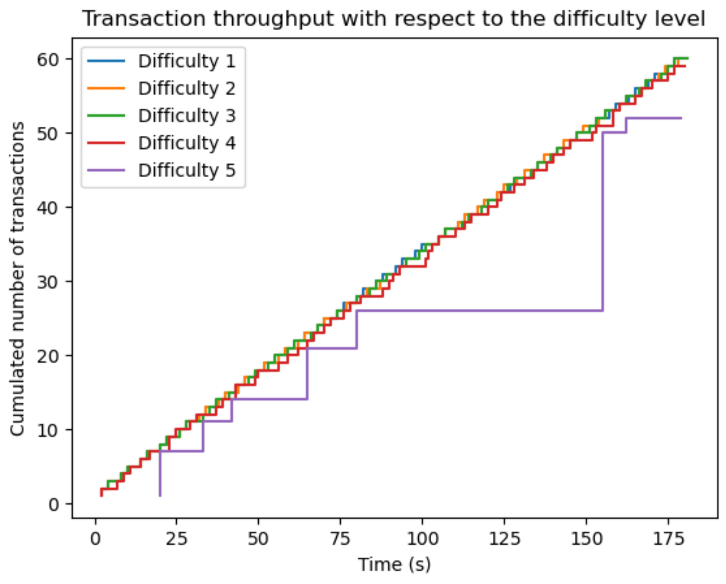

FIGURE 1 – Transaction throughput with respect to the difficulty level

We can observe that the greater the difficulty level, the longer it will take for a block to be mined and added to the blockchain and thus the lower the throughput. This is logical since the mining node takes more time to find a node with an hash below the target in a setting with a greater difficulty level.

2.3 51% attack

To perform the 51% attack, we need to enable at least 2 nodes. The first one being the victim and the second one being the attacker, the experiment is performed thanks to the following command lines :

`python app.py --p 5000 --a True --v True --m True --d 4` (first node)
`python app.py --p 5001 --a True --v False --m True --d 4` (second node)

```
python experiment.py --a True
```

where boolean argument `-a` denotes whether we're in a 51% attack simulation or not and `-v` denotes whether the instantiated node is a victim or the attacker.
One assumption we made was that a majority of miners, weighted by hash power, follow the protocol. Indeed, the system is vulnerable to attackers that would detain 51% or more of the total hashing power. In this experiment we demonstrate a 51% attack by rewriting an older version of a key.

To simulate the attack, we instantiated two nodes. One node is in attack mode and is the one that will hold the majority of the nodes while the other is the victim. The victim node calls the `time.sleep()` function in its mine method and the other doesn't, which means the attacker will mine faster and have the ability to build a new longer blockchain for the victim to adopt.

To conclude, we can say that if the attackers have 51% or more of the total hashing power, it becomes very easy for them to create new blocks because on average more than one every two blocks is created by them. They could decide to fork, even if there is a long branch, since they detain this power, eventually that fork will become the longest branch in the system regardless of what the other do. As consequence, they would be able to take control of the transactions that are included in the chain. A relevant note is that any miner on the network can theoretically perform this attack. However, in practice, if no miner has the majority of the mining power, it will be exponentially more difficult to perform successfully the attack.

3 Discussion

3.1 Consequences and scalability of your broadcast implementation

As described previously in section 2.1, we had to implement several mechanisms to avoid problematic consequences of broadcasting.

The blockchain can handle many blocks from many different peers. We chose to build a fully connected graph with each node peering with every other. This choice of implementation was motivated by its simplicity but makes the scalability limited. Indeed, every broadcasting operation (e.g. Heartbeat, Transaction broadcasting, ...) is $O(N^2)$.

3.2 Efficiency and scalability of our `put` and `retrieve` operations

Both operations are fully effective.

Regarding the put operation, we divided the work into several functions. We first call add_transaction where we check the current memory pool list as well as the transactions historic list to eventually broadcast the transaction (or not). Checking that the transaction we want to add is not in those lists is $O(N)$ in average. Broadcasting the transaction is $O(N^2)$ for N transactions.
If the callback procedure is enabled, we also have to take in account the complexity of mining. In fact, mine time complexity depends first on the difficulty of the miner. We can estimate it by the number of operations to find a hash under the current target.
Then, we add the mined block to the blockchain thanks to add_block which is $O(N^2)$ because of the broadcasting. Finally, the updating of the memory pool is $O(N)$.

Concerning the retrieve operation, the time complexity is determined by the 2 nested loops. To retrieve a value with the key, we analyze each block of the blockchain as well as all the transactions it contains. So, considering N transactions and B blocks in the blockchain, the resulting time complexity is $O(N * B)$.

3.3 Applicability of a blockchain to this problem

Using blockchain to hold a decentralized key-value store has strengths and drawbacks.

In fact, centralized data is vulnerable. It makes it possible for hackers to hack data significantly. Thus, we ensure more security by storing key-value data in a decentralized way. New key-value instances always being linked to the previous and next one, no block can be modified or deleted without a trace. In other words, keeping the store decentralized ensures their persistence and integrity. Only authorized parties are allowed to access the data.

Additionally, storing key values using blockchains is useful in cases where tracking the history of key value instances is important.

However, in order to maintain a stable, fast and fair network, we need a lot of nodes and especially miners. Indeed, competitive mining, which ensures faster and less resources-consuming implementation of the data store, is only effective when a lot of miners are in the network. In addition to that, regarding the 51% attack, the main prevention is to have a lot of miners in the network that can focus their energy on building the chain, which makes it incredibly more difficult for the attacker to build longer chains and replace the existing one.

4 Annex

```
# Implementation of the bootstrapping procedure.

bootstrapProc(address):
    response = requests('http://{address}/peers')
    if response.status == 200 then
        peers.append(address)
    else
        return

    for peer in peers then
        response = requests('http://{peer}/addNewNode?address={self.address}')
        chain_sizes[peer] = response.json()
    longestSize = max(chain_sizes.values())

    p = list(chain_sizes.keys())[list(chain_sizes.values()).index(longestSize)]

    response = requests('http://{p}/keyChain')
    blockchain = Blockchain(response.json())

    response = requests.get(f'http://{p}/memoryPool')
    for t in response.json():
        memoryPool.append(Transaction(t))

    ready = True



# Implementation of the put procedure.

put(key, value, time,block=True):

    transaction = Transaction(address, key, value,time)
    add_transaction(transaction)

    if block then
        callback = Callback(transaction, blockchain)
        callback.wait()



# Implementation of the consensus procedure.

consensus(block, address):

    index_last_block = blockchain.last_block.index

    if block.index - index_last_block >= 2 then
        response = requests('http://{address}/keyChain')
        blockchain = Blockchain(response.json())
        return True

    return False



# Implementation of the mining procedure.

mine():

    if not miner then
        return False

    a, b, c = mining, memoryPool == [], ready
    if c and (not a or b) then
        sleep(2)
        return mine()

    mining_pool = memoryPool.copy()
    candidate_block = Block(len(blockchain.blocks),address, mining_pool,
        blockchain.last_block._hash,time)

    while True then
        if mining then
            hash = candidate_block._hash

            if hash.startswith('0' * blockchain.difficulty) then
                if experiment_throughput then
                    if len(nb_transactions) == 0 then
                        nb_transactions.append(len(candidate_block.transactions))
                    else then
                        nb_transactions.append(len(candidate_block.transactions) +
                            nb_transactions[-1])
                    timing.append((datetime.now() - start).seconds)

                blockchain.add_block(candidate_block)
                handle_memoryPool(candidate_block)

                for peer in self.peers then
                    requests('http://{peer}/addNewBlock?block={str(candidate_block)}')

                return mine()

            candidate_block.nonce += 1
        else then
            return self.mine()


# Implementation of the heartbeat procedure.

heartbeat(removed):
    try:
        if not removed then
            peers_heartbeat = peers.copy()

        for peer in peers_heartbeat then
            current_peer = peer
            requests('http://{peer}/heartbeat')
            heartbeat_count[peer] = 0

    except Exception:

        peers_heartbeat.remove(current_peer)
        heartbeat_count[current_peer] += 1

        if heartbeat_count[current_peer] == 10 then
            peers.remove(current_peer)
            del heartbeat_count[current_peer]

        heartbeat(1)
```