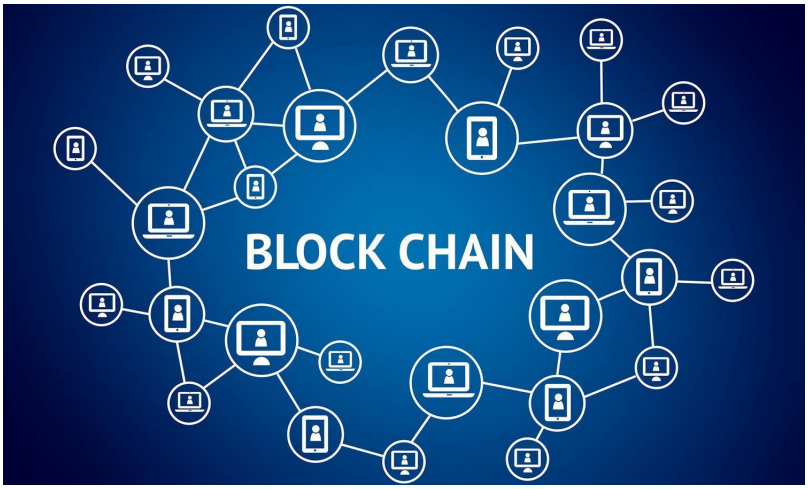
University of Liège

FACULTÉ DES SCIENCES APPLIQUÉES
Ecole d'ingénieurs et d'informaticiens

LIÈGE université

Project report

INFO8002 : Large Scale Data System
Keychain



Tom Crasset
Maxime Lamborelle
Antoine Louis

Professor : G. Louppe
Assistant : J. Hermans

2018-2019

1 Introduction

As the popularity of the Bitcoin grew, people became interested in the technology on which Bitcoin is built : the blockchain. The blockchain is a distributed and secure way to store data and can be used for many applications, ranging from virtual money transactions to key-value pairs storing, as in the context of this project.

1.1 Blockchain principle

The functioning of the blockchain is made possible thanks to a set of nodes connected to each other in a peer-to-peer network, sharing all copies of the same blockchain. Some of these nodes are called "miners" and have the particular task to validate newly committed transactions by hashing them in a particular way, in order to create a new block to add to the blockchain. Each block contains the hash of the previous block to which it points, so that a block in the chain cannot be changed unless the whole chain is changed. When the block is hashed, it can be broadcast in the network so that all nodes can verify this block and add it to their own chain. To reach consensus of the branch, each miner has to solve a complex mathematical problem whose difficulty is a parameter of the blockchain. This process is called the "Proof of Work".

2 Architecture

To implement our blockchain application, we divided the application into different layers. The layer stack is shown in Figure 1. The top application layer is a Flask application which implements the server side of each peer. We decided to put the server side (implemented in `blockchain_app.py`) on top of the stack so that each node running the Flask application can be controlled entirely by URL from a web browser or with any `HTTP` requests library. Each Flask application owns an instance of the Blockchain (implemented in `blockchain.py`). It can then use the methods implemented in the Blockchain Class in response to some `HTTP` requests. To perform these operations, the blockchain itself needs to perform GET or POST requests to request, for example, the peer nodes of a node. These requests are made in the client side layer (implemented in `broadcast.py`).

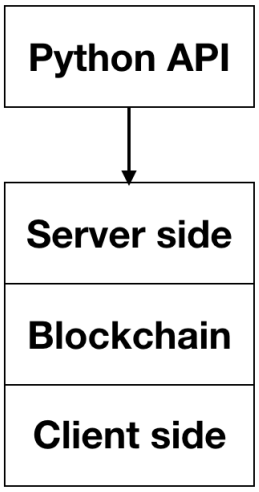


Figure 1 – Implementation stack if our blockchain

2.1 Python API

The Python API that we provide in the class `Storage` is not necessary. It is just an interface in Python using the API provided in the Flask application. This `Storage` class provides 3 methods : `put()`, `retrieve(key)`, `retrieve_all(key)` which respectively put a key-value instance in the blockchain, retrieve the latest value for a target key in the blockchain, and retrieve all values corresponding to the target key in the blockchain.

2.2 Flask : server side

To implement the server side of each node, we chose to use the `FLASK` library because is enables to easily create endpoints. The implementation of the server side can be found in `blockchain_app.py`. If the application is used with the python API, the `blockchain_app` is started in a background process. To talk to the Flask application, the user (or the Python API) only needs to know its ip address and can then perform a `GET` request on this URL with the path corresponding to a particular action. For example, if the node is located at address : 109.88.253.58 and the Flask application runs on port 5000, then making a GET request with the URL "http ://109.88.253.58 :5000/peers" results in getting the peers of this node.

The application provides the following endpoints :
- `"\blockchain"` : returns the blockchain of the node as a `json` file.
- `"\addNode"` : adds the address of the node passed as URL parameters to the list of peers.
- `"\broadcast"` : endpoint when client broadcast messages, performing different operations regarding the type of message (transaction or block).
- `"\peers"` : returns the list of all peers of the node.
- `"\heartbeat"` : returns a response to the perfect failure detector in order to keep track of other nodes that it are still alive.
- `"\put"` : puts the key-value passed as argument in the blockchain.
- `"\retrieve"` : retrieves the most recent value corresponding to the key passed as argument.
- `"\retrieve_all"` : retrieve all the values corresponding to the key passed as argument, in reverse chronological order.

This interface provides a lot of facilities but is also a potential security failure of our blockchain.

2.3 Blockchain

Our blockchain implementation consists of three different classes : the `Blockchain`, the `Block` and the `Transaction` objects. Each block has a list of transactions and the Blockchain has a chain of blocks. A lot of helper methods are implemented but the most important part of the implementation is the consensus.

Figure 2 shows the choice we made regarding the consensus protocol.

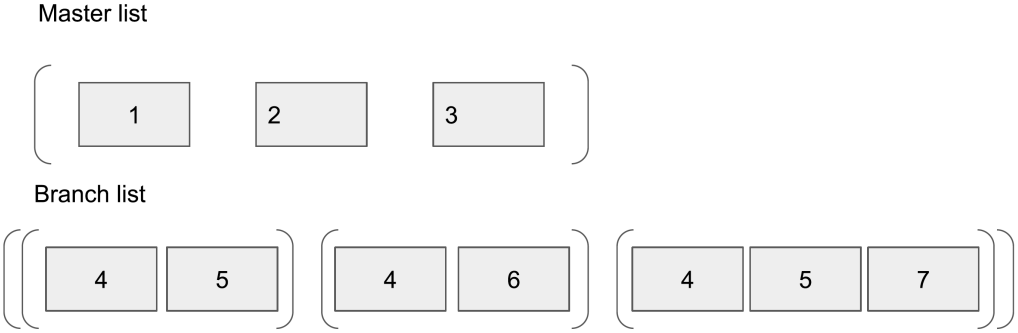


FIGURE 2 – Consensus implementation of the blockchain

We decided to keep a list of blocks, called the *Master list*, which keeps track of the agreed upon blockchain in the network of nodes. In addition to this, we store a list of branches, called the *Branches lists*, which keeps track of the diverging blockchains among all the nodes, starting from the last common block (block 4 in Figure 2). We purposefully decided that no block can attach to the blockchain if its parent is older than the last block on the master list, i.e. no block can attach on block number 2 in the given example. The only block to attach to is block number 3 or every block in the branches list. The branch where the consensus is reached on is determined using a parameter too. We chose to take the first branch that contains 2 blocks. When this happens, we extend the master list with this branch and we discard the other branches.

In contrast to the Bitcoin blockchain, we do not check index but only check previous hashes. If an incoming block has a previous hash that matches the hash of one of the blocks that can receive children, it attaches to that one, without regard to the index of either blocks.

Moreover, we do not check the validity of transactions, nor do we assure that every transaction ever broadcasted is going to be present in the master list. What we assure though is the fact that the transactions will be in chronological order, which is a minimum in this type of implementation

Further improvements include appending blocks to the chain following their indices and guaranteeing availability of every transaction ever broadcasted.

2.4 Broadcast class : Client side

Blockchains rely on the consensus of multiple nodes to create a robust storage space. Thus, information needs to be shared among all nodes. As soon as one node receives a transaction from a client, it broadcasts the transaction in the network. The same goes for blocks, i.e. if a block is mined, the miner will broadcast the block to every node so that they can in turn verify the validity of the block and add it to their local version of the blockchain.

To ensure that, when a node broadcasts a transaction or a new block, all other nodes will eventually receive this information, we needed to implement a reliable broadcast.

We chose to implement a lazy reliable broadcast which use perfect links and a perfect failure detector. We made this choice because of the topology of our network being a fully connected graph. The number of messages send by this type of broadcast is in the best case $O(N)$ and in the worst case $O(N^2)$, whereas other reliable broadcast algorithm, such as Eager reliable broadcast and All-ack uniform reliable broadcast, send in all cases $O(N^2)$ messages. We then chose the performance of the Lazy reliable broadcast over the robustness of a uniform broadcast. Note that in our case, the causality propriety is not needed. In further sections are detailed our implementation of the perfect link, the perfect failure detector and the lazy reliable broadcast.

Perfect link

The perfect link is already implemented with the TCP protocol. All of our communication between peers are done using the HTTP protocol which is based on the TCP protocol. So using the HTTP protocol, we can assume we have a perfect link.

Perfect failure detector

Each peer maintains lists of IP addresses, the first one being the list of all peers addresses the node knows and the other a correct list which is a subset of the peers addresses list containing only addresses of peers having responded well (with a HTTP response 200) to the last heartbeat.

Every 30s, each node sends a heartbeat to all IP addresses of its peers list and waits for the response. Then, depending on the response, different cases are taken into account :

— If the response comes and its status code is 200, the process is alive.

— If the process was alive at previous step and an error is raised (or the timeout exceed), the process is marked as not correct anymore and a counter begins.
— If the process was incorrect and a error is raised, the counter is incremented for this process. If it reached 10 (the process does not respond to 10 consecutive heartbeats), it is removed from the list of peers addresses.
— If the process was marked as incorrect and the heartbeat responds without raising any error, the process is marked as correct.

Lazy reliable broadcast

Each peer maintains a list of broadcasted messages it already received from each other peer. When it has to broadcast a message, it adds it to its own list and uses the best effort broadcast, which simply sends the message in an HTTP request to all other correct processes. When it receives a broadcasted message, if the message has already been received (it is in the list of the received messages of the sender), the message is discarded. But if it is a new message, the peer will check if the sender is still a correct process. If it is, the lazy reliable broadcast simply delivers the message. If it is not, it re-broadcasts the message to all other peers.

2.5 Bootstrap procedure

When a new node wants to enter in our blockchain network, it needs to bootstrap to the network. In order to be able to do that, it needs an IP address of an active node in the network. It will first ask this node for its list of peers. Then, it will send a HTTP request with its own IP address to the endpoint "\addNode" of all the peers. These peers will add the new node to their own peers list and send a HTTP responds with their last hash of their blockchain. The new node will collect all these hashes and select one node which have as last hash the most common one, considering that this node is not a corrupted one. When the new node has selected a non-corrupted node, it will ask its blockchain through an HTTP resquest on the "\blockchain" endpoint and it will copy this chain as its own chain.

3 Discussion

In this section, we discuss the scalability of our blockchain and its applicability for a key-value store. We also highlight the main defaults of our implementation.

3.1 Scalability

This blockchain can handle many nodes. But because it is a fully connected graph, it results that the number of total messages send only for heartbeat will be $O(N^2)$ each 30 seconds. So if the numbers of nodes grows to much, to network will be overloaded by these messages. A possible improvement would be to connect a node only to a subset of the peers on which it bootstrap, but it would need more complex rules to decide how big is this subset so that the load is correctly distributed in the graph.

3.2 Applicability

Using a blockchain as a key-value store is useful if one needs to keep track of the history of a key-value pair. One can take snapshots of the past by iterating through the chain and stopping at a certain timestamp in the past. This is similar to a multi-version distributed system like Google's Spanner system.

However, for most applications where historical data is not needed, this is over the top. One needs to make many assumptions for the different parts of the blockchain, namely the consensus and the

broadcasting part, which for our application is just not needed. Furthermore, because of the CAP theorem, one needs to be ready to lose out on one of the following : consistency, availability and risk to partitioning. A simple local storage would suffice as a key-value store, which doesn't have the disadvantages (nor the advantages) of network storage.

4 Experiments

An example application can be found when running the `application.py` script with the following command : `python application.py -port 5000 -bootstrap 127.0.0.1:5000 -miner True`.

Furthermore, very basic unit tests were done in the `unit_testing.py` script.

For the throughput analysis, the code for gathering the data and plotting it isn't part of the submission but it simply consisted in writing the number of transactions to a file, which were later plotted using `matplotlib`.

4.1 Faults

Unfortunately, we ran out of time to implement the tests.

4.2 Difficulty

To measure the throughput of the blockchain under different difficulties, we chose to count the number of transactions the blockchain stored at a given timestamp. Transactions were submitted to the network of nodes at intervals of 1 second and the number of transactions in the blockchain were measured every second too.

Figure 3gives an overview of the throughput for difficulties ranging from 1 to 4, values greater than 4 were not able to mine the blocks sufficiently quickly, so we abandoned further testing.

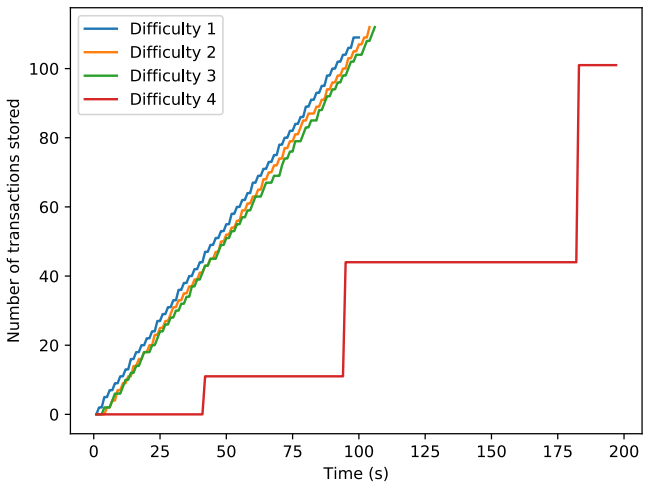


FIGURE 3 – Number of transactions stored in the blockchain for different values of difficulty

The throughput can be seen by the slope of the lines. As expected, the greater the difficulty, the slower the throughput. Eventhough a slower mining process meant more transactions per block mined, it didn't compensate the slow mining rate.

4.3 51% Attack

Unfortunately, we didn't implement the blockchain in a way that would facilitate doing a 51% percent attack. It could have been done with great effort but we ran out of time.