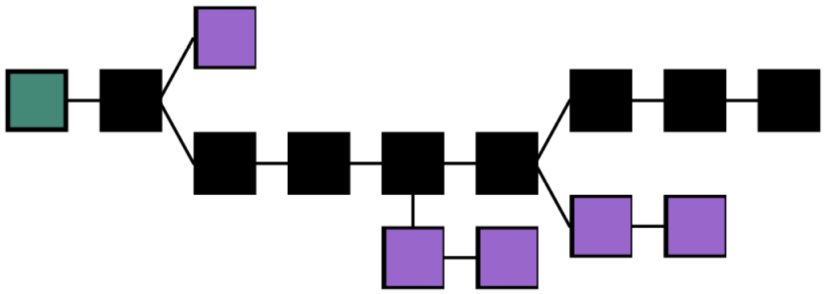
LINFO2345 Project

Charles Lohest : 17201800
Camille Caulier : 24701800

December 2021

Contents



Introduction

This project consists on the implementation of a block-Chain based on the Etherum election algorithm. We will learn how to build the structure of the blockchain and the algorithm to decide new pusher blocks with consensus mechanism. This project uses the Erlang programming language in order to understand the concurrency and the network handling functionalities of this language.

Execution

In order to execute the code you need to compile the three module `c(network). c(merkleTree). c(linkedList2).` then you can run the network with `network:network_start(N,V)`.

1 First Part

1.1 Creation of the network

In this part we created a main process `network_start(N,V)` which starts the network by creating N nodes with the `erlang:spawn` function whilst simultaneously storing the pid of the nodes. At the creation of each node we add the PID of this node in a list "Pid_list" which is broadcasted to the other nodes within the network with the use of `send_to_nodes(data,message, [H|T], N)`. The nodes wait to receive a message of the type {"pid list",Pid_List}. The List of PIDs is now broadcasted to the entire network and the network is active. The `my_node` function has a PID LIST argument which is set to the PID LIST when it receives it via message to store continuously this information.

1.2 node

Due to the nature of how we created the nodes it was considered worthwhile to explain into further detail the `my_node` function.
PARAMETERS:

- PID_LIST : This is the list of pids which is initially set to 0. When the nodes receive the list from the master they update this parameter with the pid list that they received.
- VCount : This is a parameter with the number of validators in the network. This parameter is usefull to know how many random numbers the pusher needs to receive.
- BlockChain : This is to store the blockChain in the network. When the blockchain is broadcasted all nodes update this parameter with the new blockchain.
- Random Value: This value is a parameter used by the elected node which is set to 0 at the beginning of each cycle. The parameter is incremented by all the random values that the pusher receives.
- Total received: This parameter is also set to 0 at the beginning of each cycle. It counts how many random numbers the pusher has received.
- Stop: The number of blocks that can be pushed in the blockchain before stopping the network

- V: This parameter is a validator status. Initially every node has a V set to 0 but when they are chosen as validator this parameter is switched to 1 to inform that they are validators.

MESSAGES:

- {"pid_list",Pid_list} : This message sends the pid list to every viewer node
- {"pid_list",Pid_list,"You are V"} : this message sensd the pid list and gives to the node the information that he is a validator
- {"stop", Empty} : This message informs the node that he needs to stop
- {"random_number",Random} : This message should be received by the pusher. It informs the node that he received a random number (Random).
- {"updated", New_blockchain,PID_pusher,Stopping} : This message is received by all the nodes it is the broadcasted blockchain.
- {"you are elected", Blockchain}: This message informs the node that he is elected and can push a new block into the blockchain.

1.3 blockchain Structure

To simulate the blockchain we used a linked list (stored in the module `linkedlist2`) using records. A block contains an id, id of previous block, and a timestamp. Note to reader that the ID of the block is the root of the merkle tree of the transactions. When we push a new block it is added to the head of the linked list we then proceed to elect a new node to push (see election section for more information).

We implemented a stop argument in the node function to know the maximum length of the blockchain and thus the maximum amount of push we can execute. When we reach this number a stop message is broadcasted in the network subsequently halting the system and disabling every node.

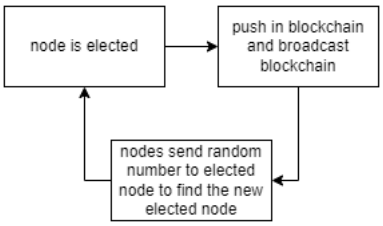


Figure 1: blockchain algorithm

1.4 Election

The system has to elect a new pusher_node after pushing a block. Upon using the master process `network_start(N,V)`, the first pusher_node created is elected randomly by the main process which takes a random node in the list of PID and send him a message {"You are Elected",BlockChain}. The Elected node receives an empty bock chain and pushes its own block to this blockchain. After

that it broadcasts the new blockchain to the rest of the nodes in the system. The pusher_node knows the PID of the other nodes thanks to the argument which stores the pid list. When the nodes receive the new blockchain via the message {"Updated", New_BlockChain} they choose a random number between 1 and N and send it to the last pusher_node. When this node has received all the random numbers it can elect a new pusher_node by calculating the sum and then doing the modulo of the number of validators. We then proceed to send the newly elected node an elected message to inform it.

2 Merkle Tree

In this part we changed the structure of our blocks to correspond to a specific form [2]. A block made is made of a root (MerkleTree), a timestamp, and the root of the previous block's merkle tree. Due to the structure of the blockchain the nodes can not know which node was the last pusher anymore, so we have to send the pusher pid with the broadcast so that each node receives the new blockchain with the pid where they need to send the random number.

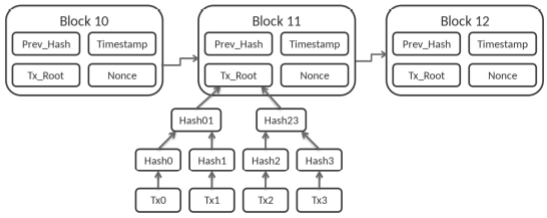


Figure 2: merkle tree diagram

In order to construct the MerkleTree we made some research and found a tutorial about the implementation of the merkle tree that we decided to follow: MerkleTree.
To create the Merkle tree, we first created the function `create_tree(TransactionData)` which takes in a list of transactions `transaction data`. We then proceed by calling `createFirstHash(TransactionData,[])` which takes in the transaction data and an empty list (this empty list is uses an accumulator the make the function tail recursive). The function `createFirstHash` will return a list of nodes with hashed transactions called `ListOfHash`. [3]

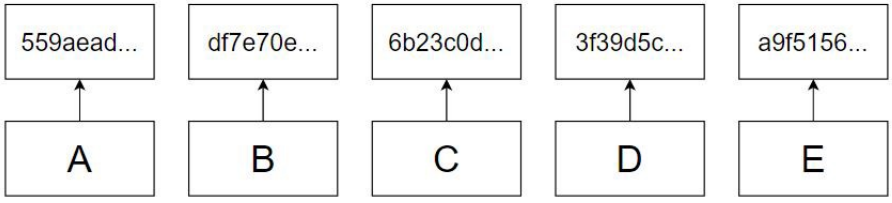


Figure 3: the transaction data is hashed

After that we call `recursiveRoot(ListOfHash)` which uses a list of hash and create the parent nodes by hashing the concatenated hashes of the children ie: the nodes in ListOfHash. We continue to do this recursively [4] until we obtain a list containing a single node aka : the root node.

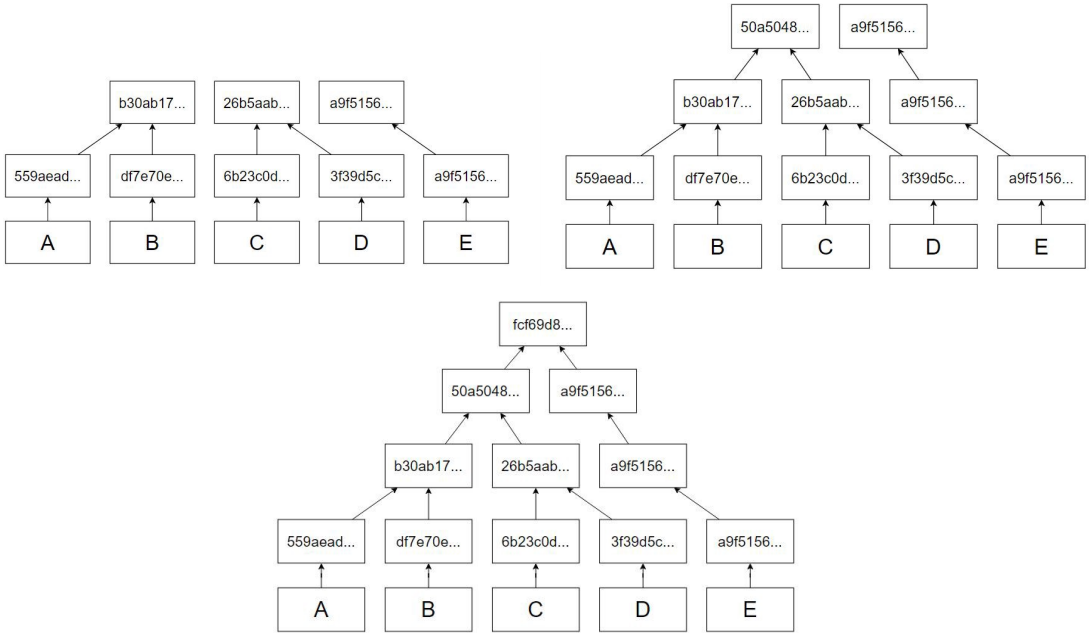


Figure 4: Merkle tree algorithm

In order to simulate the merkle tree in our blockchain we created `makeFakeTransactions()` to create a list of random fake transactions of size varying from 1 to 10 fake random transactions.

3 Sybil attack protection

In this part we have implemented a protection against Sybil attack by setting up a V number of validators. At the launch of the system the out of the N+V nodes created, the V first nodes are set as validators and the other N nodes are set as viewers. The validators have an argument V set to 1 to remember that they are validators. When the nodes receive the broadcasted blockchain we check if the nodes are a validator or not. The validators will send a random number to the last pusher node. The last pusher node will receive all the validator random numbers and sum them up together and then calculate the next pusher by computing the modulo of the sum by the number of validators to insure we choose a valid validator.

4 Conclusion

This project brought forth many challenges. With erlang being a functional language we had to find new ways of being able to store data within functions. We also had the opportunity to further understand the concept of concurrency between different processes and how to direct and command a network. The project was a very good learning tool, teaching us the internal mechanisms of a blockchain such as the merkle tree and consensus mechanisms in order to provide security in a network.