Assignment 1

200050039 : Ebrahim Sohail Haris
200050089 : Nishant Singh
2000500116 : Chetan Hiraman Rathod

February 2023

Contents

1 Introduction

In this assignment we built our own discrete event simulator for a P2P cryptocurrency network. There are several phases to this project which we are going to analyse in the upcoming sections. Some of them include simulating a connected P2P network, transactions and the latencies between the peers. We also simulate the Proof Of Work which is at the heart of a cryptocurrency network.

To implement the former mentioned simulation we have used Data stuctures and Object Oriented Programming concepts in C++. We also used xdot a widely used visualization tool to visualize the blockchain at the end of our simulation. By varying the paramenters in our simulation such as the link speed and the number of peers, allows us to draw various conclusions about a cryptocurrency network which we mention at the end of our report.

2 P2P network

2.1 Connecting the nodes

In this phase we analyse the simulation of a connected peer to peer network.There are some constraints in the network we are required to make. Each node in the network can have anywhere between 4 to 8 connections which is randomly assigned. Also the network parameters such as the number of nodes, the percent of high CPU nodes and fast nodes in our network are given in the command line. Using this we run a randomised algorithm which creates the required network. We iterate through each node and suppose that the current node already has x connections then we randomly select a number y such that:

$$4 \leq x + y \leq 8$$

. Say we got y=3, then we choose 3 random nodes other than the current node or nodes which already have 8 connections or the nodes that it is already connected to. Now we have a network, but however our job is not done yet. We need to check if the network is connected, this is done by running a DFS from any node in the network and checking if all nodes are reachable. If our network is not connected then we run this algorithm again untill we get a connected network. Since each node much have atleast four connections, the chances of creating a connected network in the first few trials itself is significantly high. We now list down some of the networks which we got after running this algorithm and visualising it through networkx library in python for various values of number of peers(n).

Figure 1: n = 5.

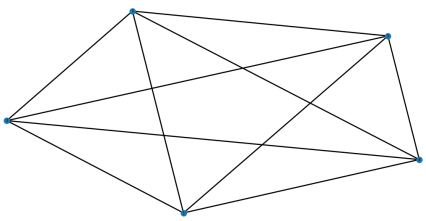


Figure 2: n = 10.

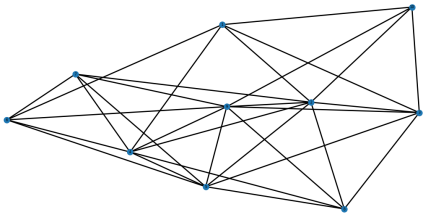
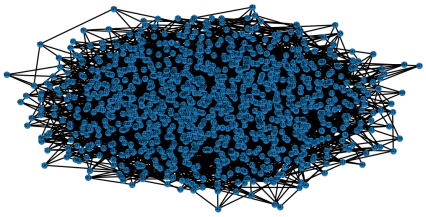


Figure 3: n = 1000.



2.2 Simulating the latencies

After connections have been established as given by the above algorithm we now simulate the latencies between each node. . First we need to assign fast and slow nodes and high cpu and low cpu nodes this is easily done by selecting z_0 and z_1 percentage of nodes randomly as fast and high cpu respectively, where z_0 and z_1 are the command line parameters as mentioned above. There are 3 variables which we have to calculate for simulating the latencies.

- ρ_{ij} (the speed of light delay from i to j) : This is a constant value for each i,j pair which is calculated from our random uniform distribution at the start of our simulation.
- c_{ij} (the link speed between i,j) : This is also calculated for each i,j pair as follows: if i and j are both fast nodes then we set it to 100Mbps else we set it to 5Mbps.
- d_{ij} (the queuing delay) : This is is randomly chosen from an exponential distribution with some mean 96kbits/ci,j

Then the latency between i and j is taken as: $\rho_{ij} + |m|/c_{ij} + d_{ij}$ where m is length of message.

3 Transactions

3.1 Generation of Transaction

We now proceed onto creating transactions between the peers. Simulating the transaction itself is not a very difficult job as we just assign a string as a transaction. Each transaction has to be of a particular type of string given by "TxnID: IDx pays IDy C coins". Since the transaction ID, which identifies a transaction needs to be unique we create a global variable which always returns a new value for the TxnID each time a transaction is created. Hence creating or simulating a transaction is a trivial task, but at the same time a **not** so trivial task is to check whether a transaction in a new block is valid or not as explained in the next section.

3.2 Validation of Transaction

To check for the validity of the transaction we need to check if IDx and IDy are valid miners, which can easily be done by parsing the transaction. We also need to check if the balance of x, bal[x](say) is such that, $bal[x] \geq c$, for which we could store the balance amount of each of the miner at each block for every miner. But this however creates a lot of overhead and a lot of wastage of memory and thus a very inefficient method. A more efficient method which we have used is for a node to store the balance of each node for only its current longest chain. Now it is trivial to check for the validity of a transaction if the transaction comes into the longest chain, we just have to check if the balance for the node who pays is greater than or equal to the amount paid. But if a block comes to a orphaned chain, now we need the balance of each miner in this chain to validate each transaction in this block. To do this we traverse from the last block in our longest chain to the block where the fork happened and along this path we modify the balance of each miner accordingly in a temporary variable we also traverse to this common block from the new block that just arrived and along the way add the amounts that each miner has lost or gained. Now let the balance of miner i at the common block be bal[i] and the money spent by i in the forked chain be x as calculated above using back-propagation then the balance of i when this new block emerged is now to be taken as:

$$bal'[i] = bal[i] - x$$

One may wonder how is it that we reached to the common block, for this we have a parameter called chain_length and last_block in each block which lets us easily reach the common parent by back-propagation, the parameters of the block will be explained in detail in the later sections. Since addition to an orphaned chain normally happens to a chain which is close to the main chain this method does not on average have to traverse a lot to reach the common parent. Also the memory overhead is exponentially decreased.

3.3 Loopless Transaction

Once we create a transaction we will broadcast it to all of the nodes we are connected to. Which then broadcasts it to its neighbouring nodes and so on. Ensuring looplessness in this transaction chain becomes vital as it may create a lot of overhead. This is done with the help of map data stucture which takes in a transaction and returns whether or not the transaction has already been listened to. If it is already listened to then we ignore this transaction else we send it to neighbours other than the one we received it from.

4 What are the theoretical reasons of choosing the exponential distribution?

There are various advantages for choosing exponential distribution. One of which is its memory less property and also that we in general can always expect to wait until the mean of the distribution(number of seconds) till we receive the next block another fact about the exponential distribution is that the number of transactions in a time interval can be depicted by a poison distribution.

5 Why is the mean of d_{ij} inversely related to c_{ij} ? Give justification for this choice.

The queuing time is inversely proportional to the link speed. If the link speed is high then the packet waiting in the queue will have to wait less time to since the packets ahead of it will get transmitted quickly(as compared to less link speed) on the network. for less link speed packet will have to wait more time.

6 The Block

Let us now dive deep into how a block is simulated and what are the parameters that each block owns. Each block is nothing but a derived data type, struct block, which we have implemented it as follows:

```
struct block {
int block_id;
int last_block;
int timestamp;
int chain_length;
std::vector<std::string> transactions;
std::string coinbase_transaction;
}
```

- block_id : Similiar to transaction_id, it uniquely identifies a block.
- last_block: An identifier which has the value of the block_id of the previous block our block is pointing to. The Genesis block has this parameter -1 as it does not have any previous block.
- time_stamp : The time at which this block was created.
- chain_length : As mentioned in the previous section chain_length is the length of the block from genesis block. Mathematically we can show that:

$$chain\_length(block\_id) = chain\_length(last\_block(block\_id)) + 1.$$

- transactions : It is an array of transactions in the block.
- coinbase_transaction: A single string which is the coinbase trasaction indicating the mining and transaction fee of the miner.

7 Proof Of Work(PoW)

The final yet vital part is to simulate the PoW. We know that every miner is in a rush to create its own block and its probability to create the next block is proportional to its computational power. This is what we will mimic. Remember that while creating the network we had assigned nodes as high CPU and low CPU. We will now use this fact. Let a miner receive a block at its chain at time t_k. Then this miner k(say) will start creating a new block. But we know that the time taken by this miner to mine a new block T_k(say) should be proportional to its computational power. hence T_k is taken from a exponential distribution with mean I/h_K where h_k is calculated as follows:

- h_k = 100/n(1000-9z_1) if k is low CPU
- h_k = 1000/n(1000-9z_1) if k is high CPU

where h_k is the hashing power of the miner k and I is the average inter arrival time between the blocks and z_1 hence the event k mines new block is pushes into the event queue at time t_k+T_k.If no other block was received by k between t_k to $t_k + T_k$ then we accept this event else we ignore it. Since I/h_k is lower for high CPU miners, it will create blocks more often. Hence we properly demonstrate the PoW.

8 Conclusion

We get to observe how the block chain behaves by tweaking the values of n,z_0 and z_1 and I, the average inter arrival time. These results can be verified from the graphs generated while running the script. For these cases we then calculate the ratio of blocks generated by a miner in the longest chain to total number of blocks in the longest chain for high CPU(Rhc), low CPU(Rlc), fast(Rf), slow(Rs).

case 1: n=10, z_0=0, z_1=0, I=600sec In this case a linear chain is observed attributing to the fact that z_0 and z_1 are both 0 meaning all nodes are equal and I=600 is quite high compared to the latencies. Hence theoretical knowledge can be verified by our simulation.

- $Rhc \approx 0$ because there is no high cpu nodes
- $Rlc \approx 0.1$
- $Rf \approx 0$ because there is no high cpu nodes
- $Rs \approx 0.1$

case 2: n=10, z_0=0, z_1=100, I=600sec Even in this case we made all nodes slow and high CPU we get the chain to be linear since each node is identical. Something we verify from this is that if the nodes all have same hashing power then there is almost 0 forking.

- $Rhc \approx 0.1$
- $Rlc \approx 0$ because there is no high cpu nodes
- $Rf \approx 0$ because there is no high cpu nodes
- $Rs \approx 0.1$

case 3: n=10, z_0=0, z_1=90, I=10sec Here we notice that most of the blocks are created by the miner "5". This is because only this miner has high CPU and others are low CPU. Hence this is the expected result.

- $Rhc \approx 0.6$
- $Rlc \approx 0.05$
- $Rf \approx 0$ because there is no high cpu nodes
- $Rs \approx 0.1$

case 4: n=10, z_0=0, z_1=0, I=.01sec Now we make a rather interesting change and keep I=.01 which is comparable to the latencies. In this case we see a lot of forking happen as expected. Hence it is important for the average inter arrival time to be of higher order than the latency between the miners. Many more such fascinating results can be observed by varying the parameters of our blockchain.

- $Rhc \approx 0$ because there is no high cpu nodes
- $Rlc \approx 0.1$
- $Rf \approx 0$ because there is no high cpu nodes
- $Rs \approx 0.1$

By varying the parameters more we can observe many more such facinating results. Let us analyse a bit more into what does the values of the ratios that we got for each case indicate. One thing we notice first is that if we introduce high cpu nodes then it has most of the ratio of the blocks generated this is expected and consistent with the theory. The ratios of blocks generated for each slow node or fast node is approximately equal to the ratio of number of slow_node/ fast_node and fast_node/slow_node respectively.

9 Control Flow

Figure 4: Control Flow.