Simulation of a P2P Cryptocurrency Network

CS 765 : Introduction of Blockchains, Cryptocurrencies, and Smart Contracts
(Project Report)

Jitesh Gawas (22M0748) || Chaitanya Borkar (22M0810) || Mrunal Janbandhu (22M0811)

February 2023

1 Answers To The Questions

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

We use the exponential distribution because it has a few unique properties that make it a good fit for modelling certain situations, like how transactions might appear in a peer-to-peer cryptocurrency network.

For example, the exponential distribution has a "memoryless" property, which means that the chance of something happening in the future does not depend on how much time has passed since it last happened. This can help predict when transactions might show up in the network since we do not want the time since the last transaction to affect our predictions.

The exponential distribution is also suitable for working with continuous time (meaning time that does not stop and start), and it is relatively easy to use mathematically.

Therefore, the exponential distribution is a suitable choice for modelling transaction generation in the P2P cryptocurrency network due to its memorylessness, continuous time properties, and ease of use.

1.2 Why is the mean of dij inversely related to cij ?

The mean of dij (the time it takes for a message to travel from peer i to peer j) is related to cij (the link speed between peers i and j) in an opposite way. As the link speed between peers i and j increases, the time it takes to transmit a message decreases, so the time spent waiting in a queue at node i becomes a bigger part of the total delay. To ensure that the time spent waiting in the queue does not dominate the total delay of the message, the mean of dij is inversely related to cij.

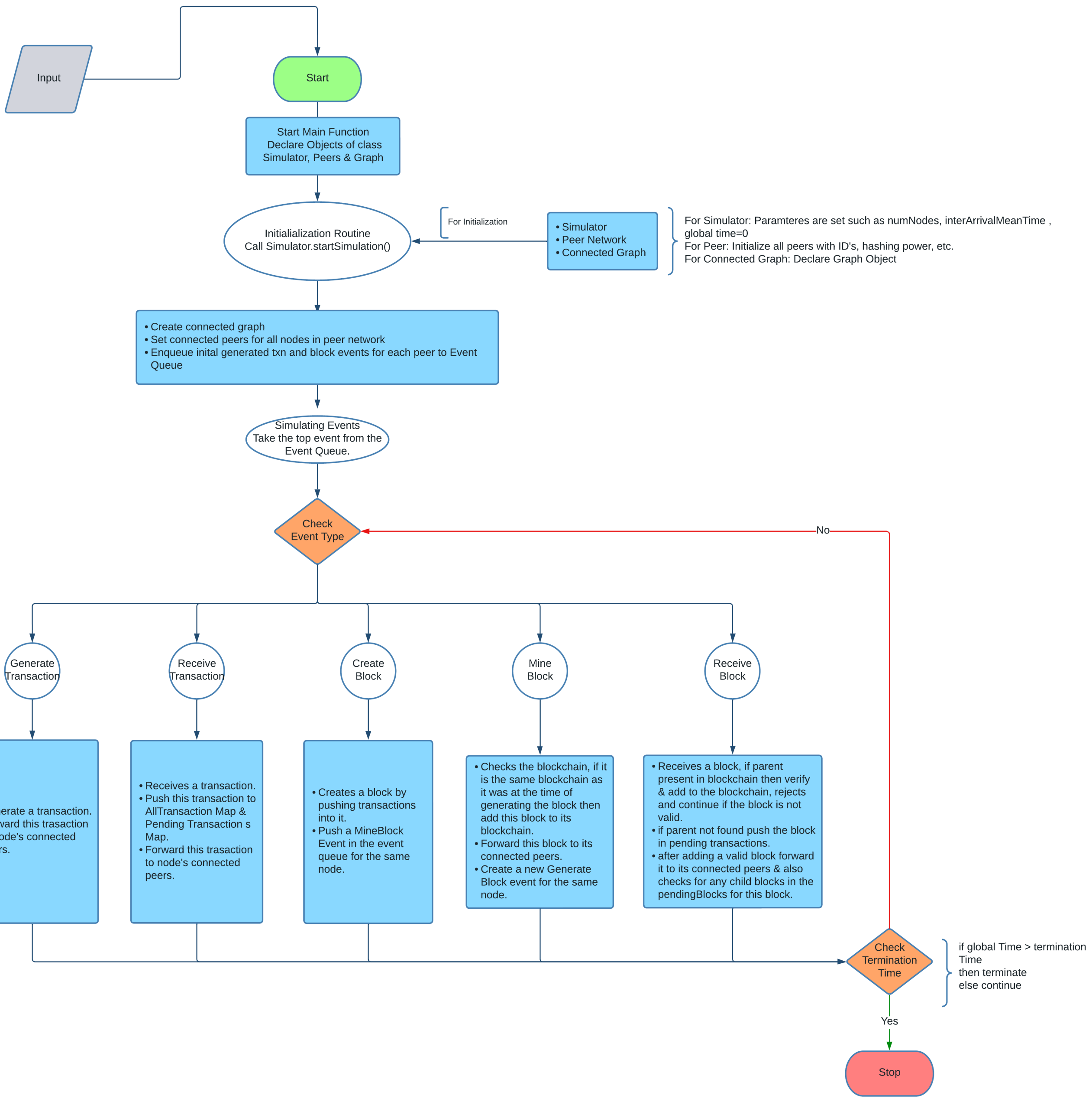
This ensures that the waiting time at node i becomes a minor part of the total delay as the link speed increases, which is more like how real networks behave. This also allows for more variation in the waiting time at node i for slower links, which helps to make the simulation more realistic.

1.3 Explanation for the choice of a particular mean.

We choose a specific value for the average time between block arrivals to simulate the time it takes for new blocks to be added to a blockchain network. We call this value "Tk". To determine this value, we consider the average time it takes for blocks to arrive and the amount of computational power each node has. Nodes with more computational power are more likely to mine a new block and add it to the blockchain, so we adjust the mean value of Tk based on the fraction of each node's total hashing power.

This approach ensures that the time between two consecutive blocks in the blockchain is what we expect it to be, and it also considers the impact of computational power on block mining.

2 Design Document



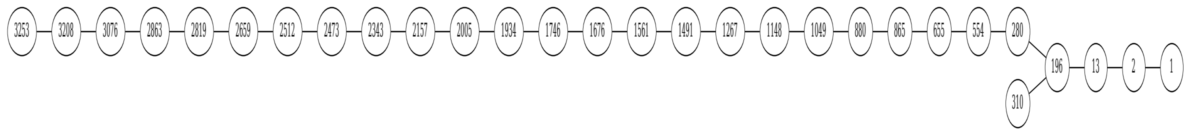
Input
Start
Start Main Function
Declare Objects of class
Simulator, Peers & Graph
Initialialization Routine
Call Simulator.startSimulation()
For Initialization
• Simulator
• Peer Network
• Connected Graph
For Simulator: Paramteres are set such as numNodes, interArrivalMeanTime ,
global time=0
For Peer: Initialize all peers with ID's, hashing power, etc.
For Connected Graph: Declare Graph Object
• Create connected graph
• Set connected peers for all nodes in peer network
• Enqueue inital generated txn and block events for each peer to Event Queue
Simulating Events
Take the top event from the
Event Queue.
Check
Event Type
No
Generate
Transaction
Receive
Transaction
Create
Block
Mine
Block
Receive
Block
• Generate a transaction.
• Forward this trasaction to node's connected peers.
• Receives a transaction.
• Push this transaction to AllTransaction Map & Pending Transaction s Map.
• Forward this trasaction to node's connected peers.
• Creates a block by pushing transactions into it.
• Push a MineBlock Event in the event queue for the same node.
• Checks the blockchain, if it is the same blockchain as it was at the time of generating the block then add this block to its blockchain.
• Forward this block to its connected peers.
• Create a new Generate Block event for the same node.
• Receives a block, if parent present in blockchain then verify & add to the blockchain, rejects and continue if the block is not valid.
• if parent not found push the block in pending transactions.
• after adding a valid block forward it to its connected peers & also checks for any child blocks in the pendingBlocks for this block.
Check
Termination
Time
if global Time > termination
Time
then terminate
else continue
Yes
Stop

3 Visualization

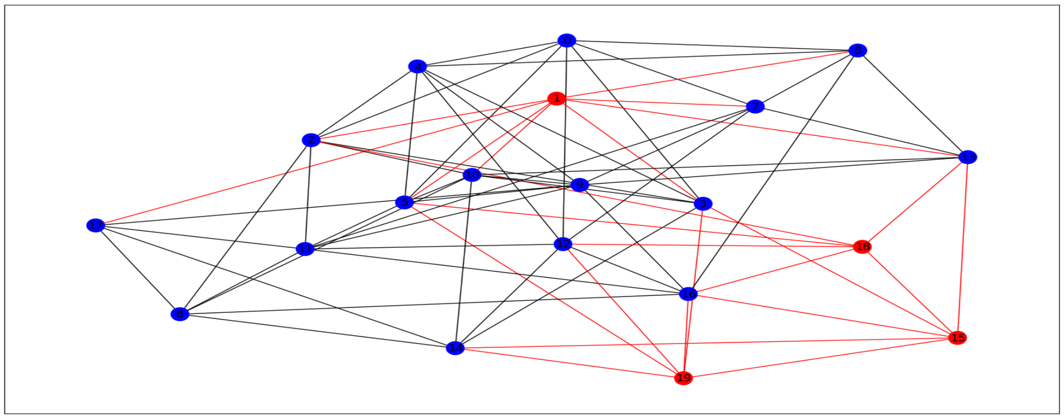
For all the visualizations the number of nodes is 20.

3.1 Parameters : $Z_0 = 0.2, Z_1 = 0.6, T_{tk} = 10, T_k = 100$

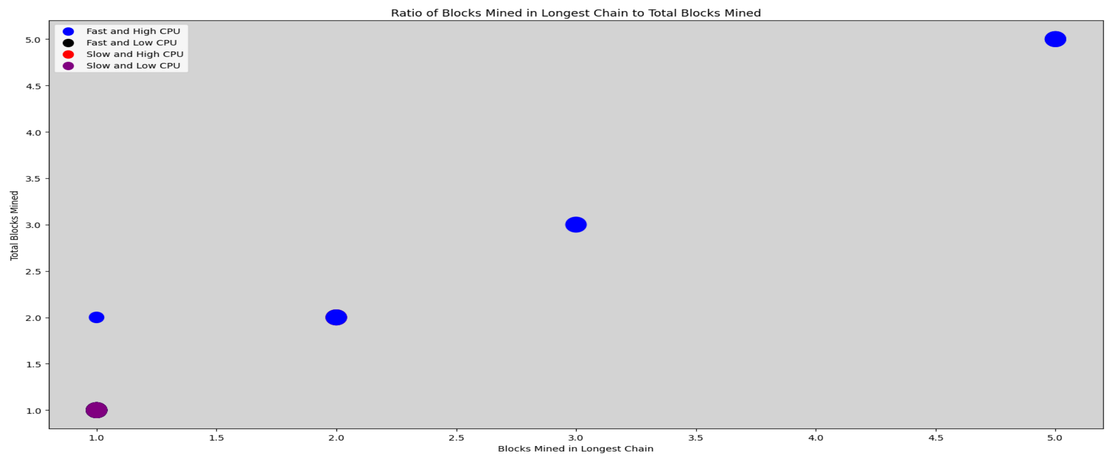
Blockchain



Peer Network



Mined in longest chain vs Total Blocks Mined



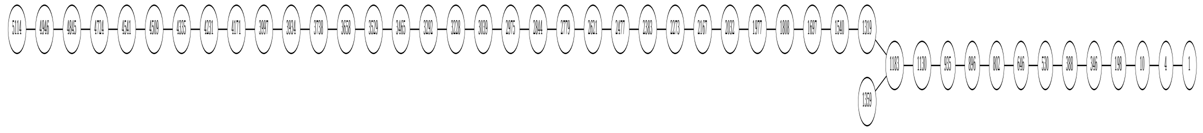
Note: Ratio = Blocks mined in longest chain vs Total blocks mined in blockchain
Network Speed : 0 - Slow | 1 - Fast || CPU Usage : 0 - Low | 1 - High

Node	**Network Speed**	**CPU Usage**	**Blocks mined (longest chain)**	**Total Blocks mined**	**Ratio**
1	1	0	0	0	NA
2	0	0	1	1	1
3	1	0	0	0	NA
4	1	0	0	0	NA
5	1	0	1	1	1
6	1	0	0	0	NA
7	1	1	1	1	0.5
8	1	1	5	5	1
9	1	0	1	1	1
10	1	0	0	0	NA
11	1	1	5	5	1
12	1	0	1	1	1
13	1	1	1	1	1
14	1	1	2	2	1
15	1	1	3	3	1
16	0	1	2	2	1
17	1	0	1	1	1
18	1	1	2	2	1
19	0	0	0	0	NA
20	0	0	1	1	1

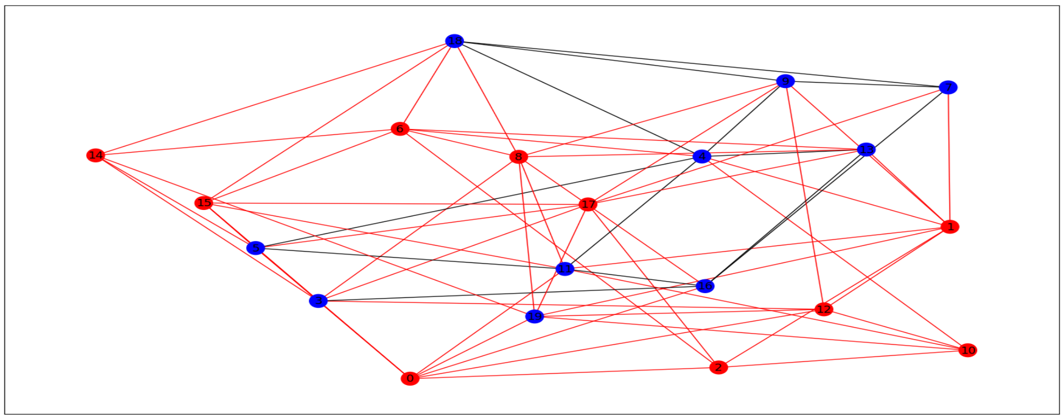
Table 1: $Z_0 = 0.2, Z_1 = 0.6, T_{tk} = 10, T_k = 100$

3.2 Parameters : $\mathbf{Z}_0 = 0.5, Z_1 = 0.6, T_{tk} = 10, T_k = 20$

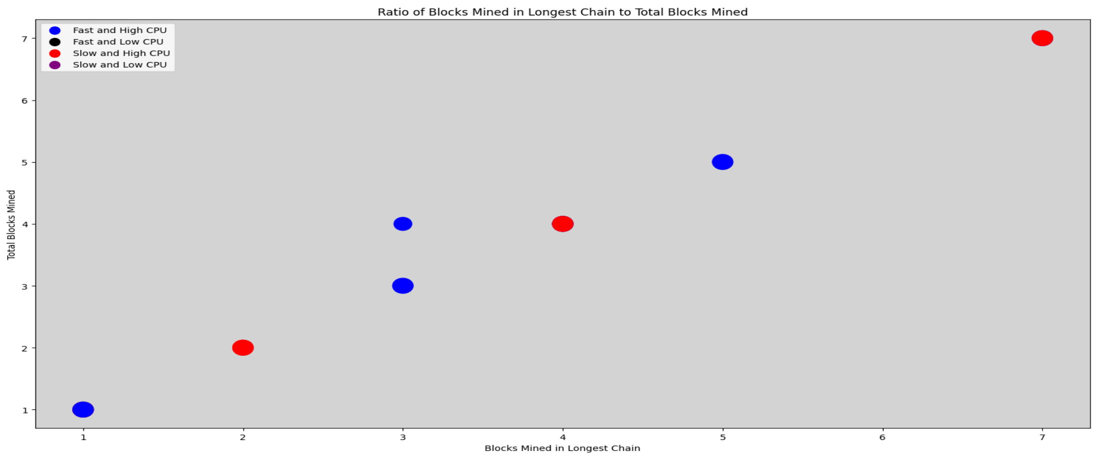
Blockchain



Peer Network



Mined in longest chain vs Total Blocks Mined



Note: Ratio = Blocks mined in longest chain vs Total blocks mined in blockchain
Network Speed : 0 - Slow | 1 - Fast || CPU Usage : 0 - Low | 1 - High

Node	**Network Speed**	**CPU Usage**	**Blocks mined (longest chain)**	**Total Blocks mined**	**Ratio**
1	0	0	0	0	NA
2	0	0	0	0	NA
3	0	1	4	4	1
4	1	1	3	4	0.75
5	1	0	0	0	NA
6	1	0	0	0	NA
7	0	0	0	0	NA
8	1	1	7	7	1
9	0	1	7	7	1
10	1	1	4	4	1
11	0	1	1	1	1
12	1	0	0	0	NA
13	0	1	2	2	1
14	1	1	3	3	1
15	0	1	2	2	1
16	0	0	1	1	1
17	1	1	5	5	1
18	0	1	4	4	1
19	1	0	0	0	NA
20	1	1	1	1	1

Table 2: Parameters : $Z_0 = 0.5, Z_1 = 0.6, T_{tk} = 10, T_k = 20$

4 Observations

1. If the Inter arrival mean time of block is very close to the Inter arrival mean time of transaction, then most of the mined blocks have very few transactions.

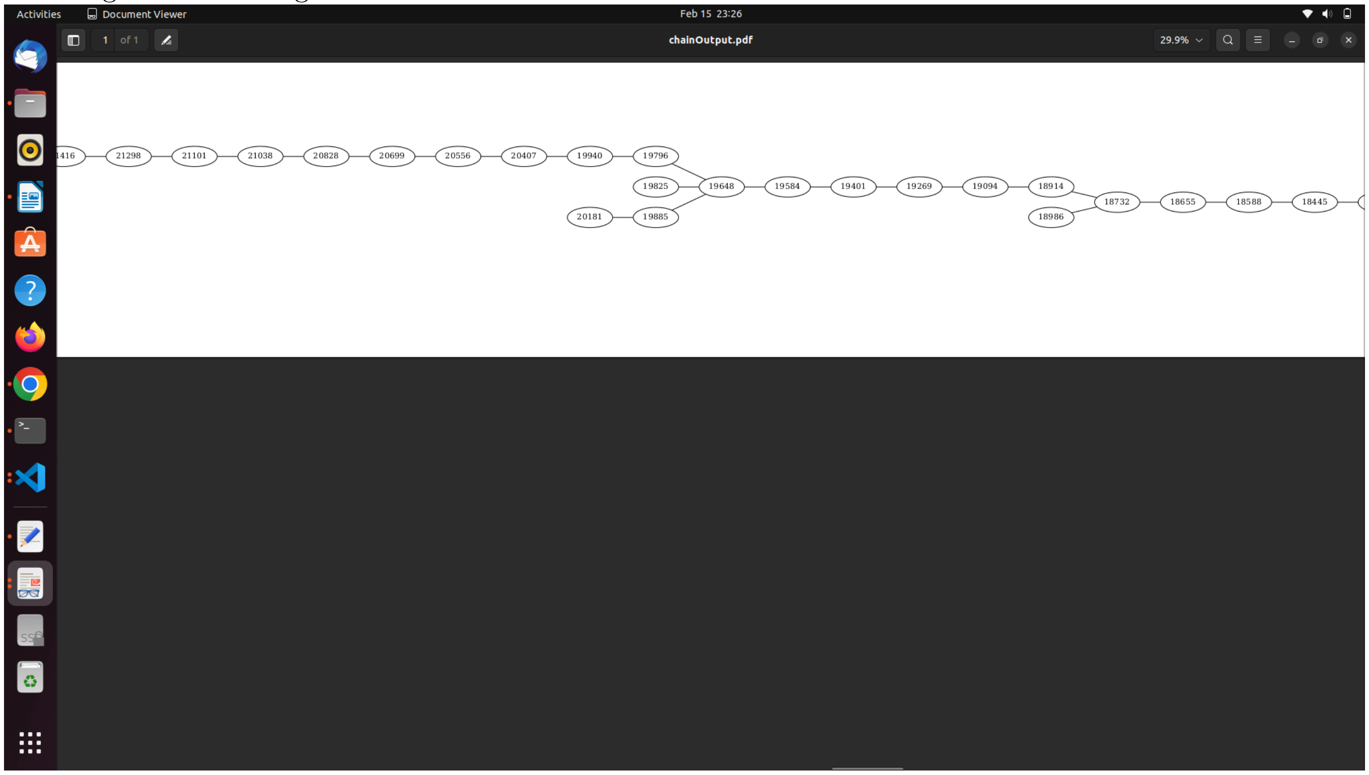
2. If a node is both fast in terms of network speed of high cpu usage, then such nodes are most likely to mine the majority of blocks.

3. A node with slow network speed low cpu usage, often ends up in mining zero blocks.

4. The max size of a fork branch observed while trying out simulation with various parameters was of length three.

5. At termination time, slow network speed node don't have the ending blocks in its blockchain as compared to the blockchain at a fast node.

6. The longest branch length observed was 3.



5 Class Declarations

5.1 Class DiscreteEventSimulator

Data Members

1. numNodes: It stores the number of nodes in the network.
2. z_0 : represents the percentage of *slow* nodes in the network.
3. z_1 : represents the percentage of *low CPU usage* nodes in the network.
4. globalTime: to simulate *time* in our simulation.
5. transaction_Counter : to keep the count of transactions generated while simulating the events.
6. txnInterArrivalMeanTime: Inter Arrival Mean Time for choosing random arrival times between two transactions from a distribution whose mean equals this value.
7. blockInterArrivalMeanTime: Used for simulating Proof-Of-Work.
8. DateTime: Used for creating different logs for different program builds.
9. EventQueue : Queue which maintains events in ascending order of their event time.
10. currEvent: Pointer to an Event object, used to access the event popped from the EventQueue at any given time.

Member Functions

1. DiscreteEventSimulator(int numPeers, float z0, float z1, float txnMean, float blkMean): Constructor functions to initialize the simulator's parameters.
2. PrintParameters(): Prints the parameters like numNodes, z_0, z_1 etc.
3. startSimulation(Graph &adjMatrix, Peers &PeerNetwork): call to this functions starts the simulation.
4. writeBlockArrivalTimes(Peers &PeerNetwork, string DateTime): After the simulation, this function can be called to write the block arrival times of each node to a separate file.
5. writeBlockChain(Peers &PeerNetwork, string DateTime): This function writes the Blockchain formed at each peer in the form of BlockId,ParentId to a separate file after the simulation is completed.

5.2 Class Block

Data Members

1. blockId: ID of the block.
2. PrevHash: ID of the parent block.
3. minedId: ID of the node which has mined the block.
4. blockLevel: The level of the blockchain tree at which this block is supposed to be present or attached.

5. Transactions: Collections of all transactions included while creating the block.

6. NodeBalances: Collection of balances of all the nodes, values in this collection are the values after the transactions of the block will be performed.

Member Functions

1. Block(): Default constructor of block class.

2. Block(int blockID, int PrevHash, int blockLevel): Parameterized constructor of block class.

3. operator=(const Block *rhs): Overloaded "=" operator in order to perform "B = B_dash" where B_dash is a pointer to a block object B is a block object.

4. operator=(const Block &rhs): Overloaded "=" operator in order to perform "B = B_dash" where B_dash are block objects.

5.3 Class Transactions

Data Members

1. senderId: Id of the Node which is the sender/payee of this transaction (exceptions in case of coinbase transaction, where this becomes the receiver of coinbase/mining fee).

2. receiverId: Id of the Node, which is the receiver of this transaction.

3. txnId: Id of the transaction.

4. type: Type of the transaction.

5. coins: Amount of coins exchanged in this transaction.

6. txnTime: time of the execution of the transaction.

Member Functions

1. Transaction(): Default constructor of transaction class.
2. Transaction(string message, float timeStamp): Parameterized constructor of transaction class.
3. HashFunction(string Message, float timeStamp): Returns Hash(Message||timeStamp)/ || - Concatenation.
4. getMessage(): Returns the message form of a transaction.
5. Split(string Message): splits the message into individual words, used for internal purposes.
6. operator=(const Transaction *rhs): Overloaded "=" operator in order to perform "T = T_dash" where T_dash is a pointer to a block object T is a transaction object.
7. operator=(const Transaction &rhs): Overloaded "=" operator in order to perform "T = T_dash" where T_dash T are transaction objects.

5.4 Class Event

Data Members

1. senderId: Id of the node, which is a sender of the event.
2. receiverId: Id of the node, which is a receiver of the event.
3. eventTime: Execution time of the event.
4. type: Type of event.
5. Transaction T: If the event is of type "Generate/Receive" Transaction, it represents that transaction.
6. Block B: If the event is of type "Generate/Mine/Receive" Block, it represents that Block.

Member Functions

1. Event(int NodeId, float eventTime, string type): Parameterized constructor used for creating initial events like createBlock.
2. Event(Transaction *T, string EventType, Node *sender, Node *receiver, float prop_delay): Parameterized constructor used for creating an event related to the transaction.
3. Event(Transaction T, string EventType, Node *sender, Node *receiver, float prop_delay): Parameterized constructor used for creating an event related to the transaction.
4. Event(Block B, float eventTime, string EventType, Node *sender, Node *receiver): Parameterized constructor used for creating an event related to Block.
5. Event(Block *B, float eventTime, string EventType, Node *sender, Node *receiver): Parameterized constructor used for creating an event related to Block.
6. calculate_Latency(int senderNWspeed, isnt receiverNWspeed, int numTransaction): Used to calculate the network latency between sender & receiver except the propagation delay.

5.5 Class Graph

Data Members

1. adjMatrix: Adjacency Matrix representing the connected graph of peers.
2. visited: Information about whether a node is visited while checking the graph's connectivity.
3. numNodes: Number of nodes in the network.
4. minDegree: Minimum number of peers to which a node should be connected in the final network.
5. maxDegree: Maximum number of peers to which a node should be connected in the final network.

Member Functions

1. Graph(int numNodes, int minDegree, int maxDegree): Parameterized constructor for the class Graph.
2. createGraph(): Creates a network where each node is connected to peers in the range of minDegree maxDegree.
3. isConnected(): Checks the connectivity of the graph.
4. dfs(int v): Depth First Search, used for checking the connectivity.
5. does_exist(const vector<vector<int» &adjList, int Node1, int Node2): Used to check whether a *Node1* is connected to *Node2* or not.

5.6 Class Node

Data Members

1. NodeId: Id of the node.
2. NWspeed: 0 - Node is slow, 1 - Node is fast.
3. CPU_Usage: 0 - Low CPU usage, 1 - High CPU Usage.
4. lastBlockId: Id of the last block of the longest chain of the blockchain.
5. balance: Balance of node.
6. hashing_power: Hashing Power of the node.
7. blockChainLength: Length of the longest chain of blockchain.
8. connectedPeers: Lists all the nodes connected to this node.
9. map<string, Transaction> AllTransactions: Collection of all transactions received to this node.
10. map<string, Transaction> PendingTransaction: Accumulates the transactions that remain unadded to any block until now.
11. map<int, Block> Blockchain: Tree of Blocks.

12. map<int, Block> PendingBlocks: Received Blocks which are not added to the blockchain due to the absence of their parent block.

13. map<int, bool> ReceivedBlocks: ID's of all received blocks.

14. map<int, float> BlockArrivalTimes: Stores arrival time of all the blocks received during the simulation.

Member Functions

1. bool isConnected(const Graph &adjMatrix, int peerId): Returns true if Node with ID = peerID is connected with this Node.

2. GenerateTransaction(DiscreteEventSimulator *Simulator, string TxnType): Generates transaction for the node (sender, which is this node only).

3. ReceiveTransaction(DiscreteEventSimulator *Simulator, Event *currEvent): Receives a transaction at this node's end.

4. RandomInterArrivalTxnTime(float InterArrivalMean): Returns the random duration from an exponential distribution whose mean equals *InterArrivalMean* for the transaction time.

5. RandomInterArrivalBlockTime(float InterArrivalMean): Returns the random duration from an exponential distribution whose mean equals *InterArrivalMean* for the block time.

6. GenerateBlock(DiscreteEventSimulator *Simulator, int *BlockCounter): Creates a block for this node & pushes a *MineBlock* event to the *EventQueue*

7. MineBlock(DiscreteEventSimulator *Simulator, Event *E, int *BlockCounter): Mines a block for the node & pushes a *createBlock* event to the *EventQueue*

8. BroadcastBlock(DiscreteEventSimulator *Simulator, Event *E): Broadcasts a block to the connected peers.

9. ReceiveBlock(DiscreteEventSimulator *Simulator, Event *E, int *BlockCounter): Receives a blocks, checks whether it needs to be added to the blockchain or not & pushes *createBlock* event to the *EventQueue.*

10. VerifyAddBlock(Block B): Verifies all the transactions of the block used while receiving a block.

5.7 Class Peers

Data Members

1. PeerVec: Pointers to the Node objects representing the Nodes in the peerNetworks.
2. numNodes: Number of nodes of the peer network.
3. BlockCounter: Used for setting unique block ids to each block.
4. z0_Set: Node ids of the nodes with slow network speed.
5. z1_Set: Node ids of the nodes with low CPU usage.
6. slow_HashPower: hashing power with low CPU usage.

Member Functions

1. Peers(int numNodes, DiscreteEventSimulator &Simulator): Initializes the data members and randomly sets some nodes to have slow network speed and low CPU usage. It also distributes the hashing power among the nodes such that nodes with high CPU usage have ten times more hashing power than nodes with low CPU usage.
2. PeerInfo(): Displays the information for each node in the system, such as node id, balance, network speed, CPU usage, and hashing power.
3. setConnectedPeers(Graph &adjMatrix): This method sets the connected peers in a graph data structure by using a reference to the adjacency matrix representation of the graph.