Assignment 1: CS765

190050113 Shivam Raj
190050017 Aniket Agrawal
190020010 Aman Singh

September 7, 2020

Part 2

The reason to choose exponential distribution for the interarrival b/w transactions generated by any peer is the fact that it is memory-less, and the previous result doesn't affect the upcoming ones.

Part 4

We made the connected graph by adding one node at a time such that it is randomly connected to x number of nodes from the graph formed till now, and x is lying in uniform distribution from 1 to min(log(n), number of nodes present at that time). By this it is sure that after adding the node graph is still connected and also that the maximum degree is somewhat restricted due to condition of log(n). We choose the uniform distribution here to ensure that the degree is of the range log(n) which happens generally in reality.

Part 5

The reason to choose d and inversely proportional to c is because as the speed of network link will increase the queuing will be less and thus the value of d will be less, and if c is more then more queuing will lead to more d. Therefore, they are inversely proportional.

Part 7

The latency in the transmission b/w the networks is around x ms, so we need to keep the mean of mining time as 10 to 100 times the latency and thus we computed the mean of mining time by theoretically computing the latency.

Insights

Visualizing Network :-

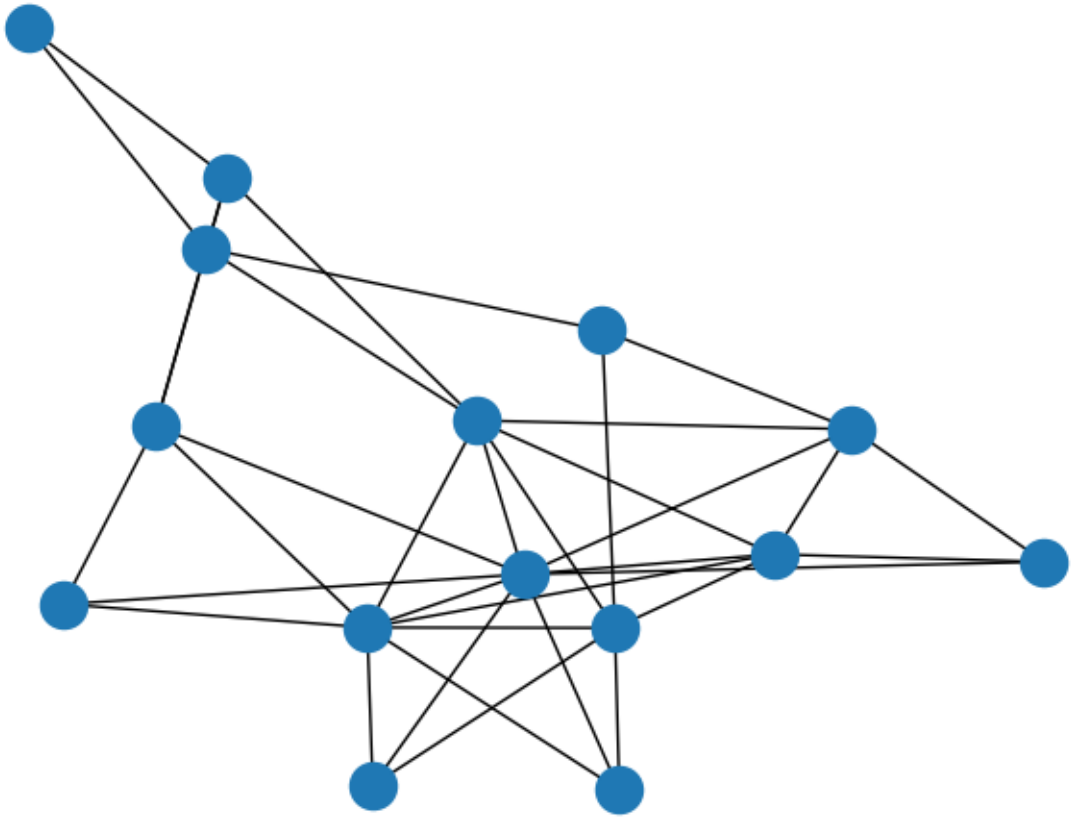


Figure 1: Generated graph for 10 nodes

As compared to total number of blocks in the primary chain of the blockchain at any node, total number of blocks stored at that node is very high.

The branching of blockchain decreases as we increase the inter-arrival time of transactions. This can be justified by the fact that increased inter-arrival time give enough time to nodes to mine a block.
The change in the in percentage of slow nodes does not have any significant effect on branching, if we run the simulation for significant mount of time.
The increase the inter-arrival time of blocks again give enough time to nodes to mine a block and hence decreasing the branches.
As the branching decrease the ratio of number of blocks generated by each node in the Longest Chain of the tree to the total number of blocks it generates at the end of the simulation get close to 1.

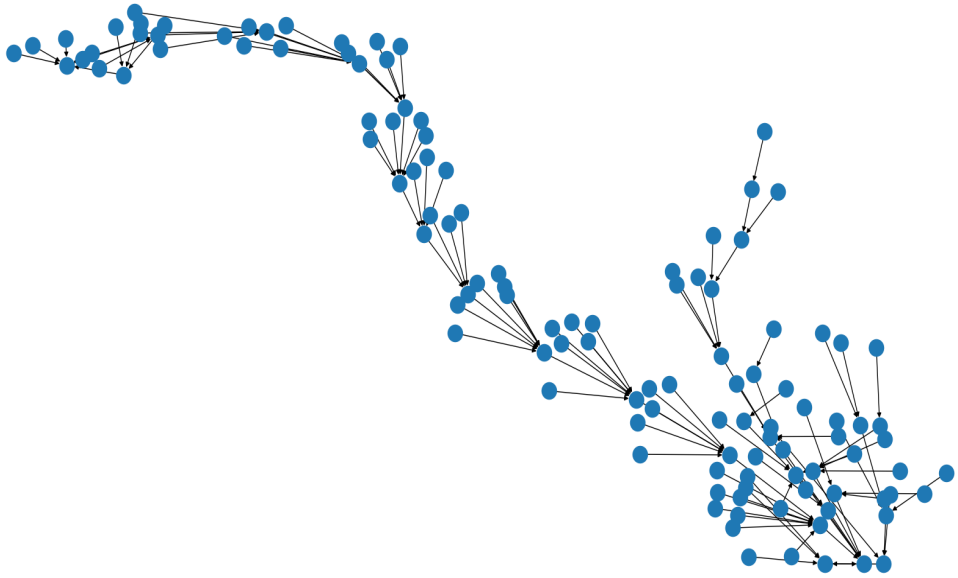


Figure 2: All received blocks (High mean mining time)

Design Doc

There are 4 important classes in our solution : `Transaction`, `Block`, `Event`, `Node` and `Simulation`. The `Transaction` and `Block` classes model their standard meanings. `Transaction` has a sender, a receiver, a value and a unique id (tid). `Block` has a block id, a parent block id, transactions included in the block and a miner.

```
class Transaction:
    def __init__(self, tid, sender, receiver, value):
        self.sender=sender
        self.receiver=receiver
        self.value=value
        self.tid=tid
class Block:
    def __init__(self, bid, pbid, txnIncluded,miner):
        self.bid = bid
        self.pbid = pbid
        self.txnIncluded = txnIncluded
        self.miner = miner
```

The `Event` class is inherited by 4 other classes `TxnGen`, `TxnRecv`, `BlockRecv` and `BlockMined`. The event class has attributes such as a sender, receiver, time and eventid. The 4 subclasses of `Event` class form all the events that we need to add to our Simulation object's event queue.

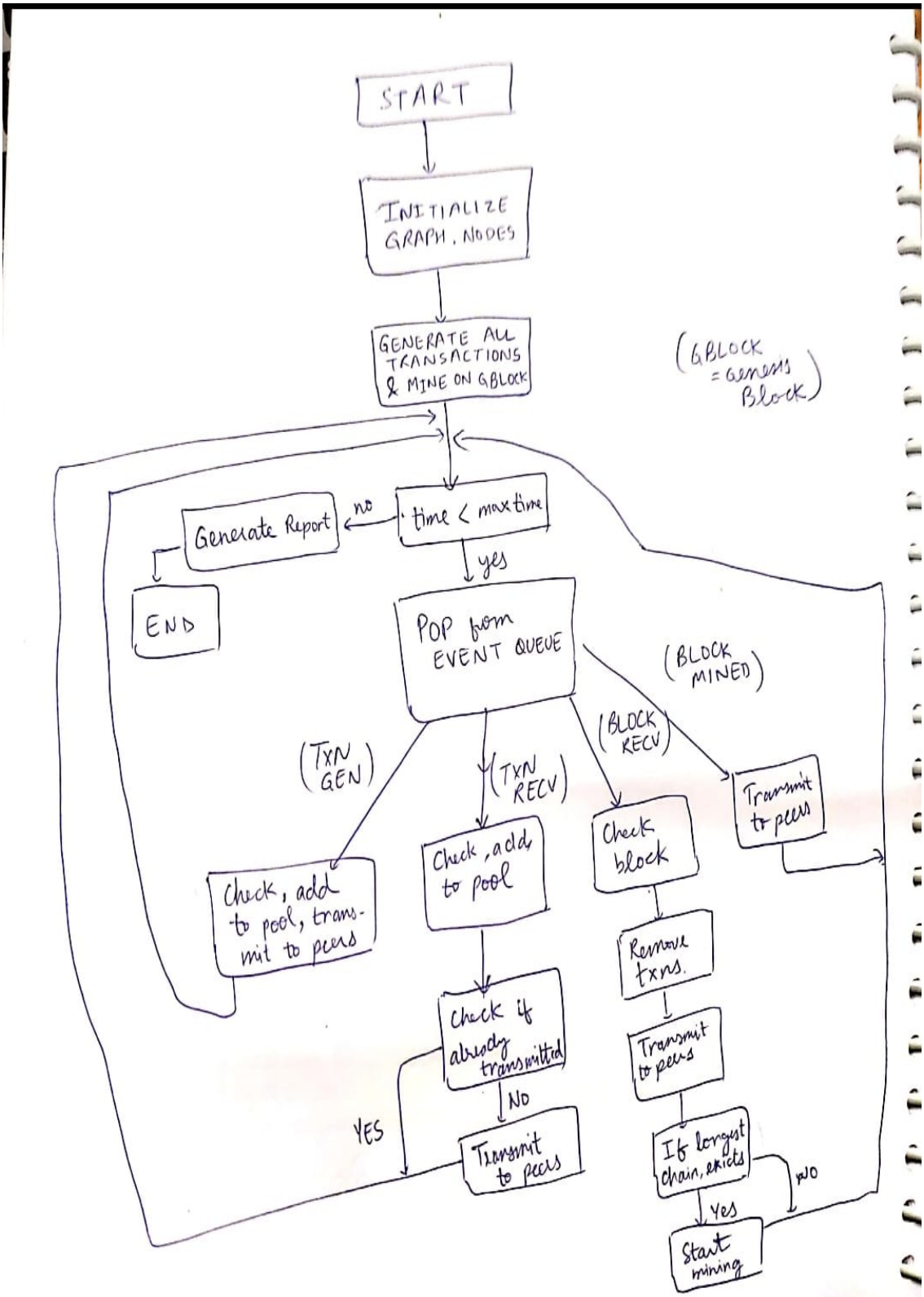
```
class Event:
    def __init__(self, time,eventId, sender=-1, receiver=-1):
        self.time=time
        self.sender=sender
        self.receiver=receiver
```

```
        self.eventId=eventId
```

The `Node` class has all the relevant fields like the mean of generating time required to mine a block, it's id, whether or not it is "fast" etc. This class also includes all the event handler functions. Whenever an event (object of the `Event` class) is popped from the event queue, we inspect the details of of the event and call the relevant node's event handler. The event handlers then add other events to the event queue at appropriate times. The time attribute of the events which are added are calculated using appropriate latencies (block size, link speed, speed of light delay, time required to mine etc.).

```
class Node:
    def __init__(self, nid, speed, genesis, miningTime):
        self.nid=nid
        self.speed=speed
        self.lbid=genesis.bid
        self.blockChain[genesis.bid]=copy.deepcopy(genesis)
        self.miningTime=miningTime
```

The `Simulation` class maintains the simulation parameters, initialzes nodes and connects the nodes appropriately. It also maintains an event priority queue from which it pops events and calls relevant event handlers passing in the appropriate parameters.


START
INITIALIZE GRAPH, NODES
GENERATE ALL TRANSACTIONS & MINE ON GBLOCK
(GBLOCK = Genesis Block)
time < max time
no
Generate Report
END
yes
POP from EVENT QUEUE
(TXN GEN)
(TXN RECV)
(BLOCK RECV)
(BLOCK MINED)
Check, add to pool, transmit to peers
Check, add to pool
Check block
Transmit to peers
Remove txns.
Transmit to peers
Check if already transmitted
YES
NO
Transmit to peers
If longest chain, exists
NO
Yes
Start mining