# Assignment -1 CSL7490
Instructor: Nitin Awathare

Due Date: 24/09/2024 11:59 PM SGT

**Objective**: Utilize OMNeT++ Discrete Event Simulator to simulate the blockchain on top of the P2P network.

**Overview**: The network consists of peer node and seed node. Each peer must connect to at least $\lfloor (n/2) \rfloor + 1$ seeds out of $n$ available seeds. Note that $n$ will vary for each instance of the peer-to-peer network and the config file should be changed accordingly. Config file contains the details (IP Address:Port) of seeds. The definitions of "seed","peer" etc. are given below. Each "peer" in the network must be connected to a randomly chosen subset of other peers. The network must be connected, that is the graph of peers should be a connected graph. In order to bootstrap the whole process, the network should have more than one *Seed* node which has information (such as IP address and Port Numbers) about other peers in the network. Any new node first connects to seed nodes to get information about other peers, and then connects to a subset of these. These peer nodes are actually a blockchain nodes that create a block and maintains the blockchain.

Once the network is formed, the peers broadcast messages in the network and keep checking the liveness of connected peers in regular intervals. If a node is found to be dead, the details of that node needs to be sent to the seeds.

Each peer node would run a proof-of-work blockchain protocol. The Blockchain should start with a genesis block whose hash is "0x9e1c". Each hash is 16 bit in length and hence can be represented in hexadecimal with four letters. Every block consist of following fields:

- Block Header
  - Previous block hash (16 bits)
  - Merkel root (16 bits)
  - Timestamp - Unix timestamps, which is the number of seconds that have elapsed since January 1, 1970 at 00:00:00 GMT.
- Block Body : For now we leave the Body empty (with no transactions). So the block is just the block header. The block hash which you put in a block is the last 16 bits of the SHA-3 hash of the previous block. The Merkel root is any arbitrary value for this assignment.

**Network setup**
Your network will consist of the following type of nodes:

1. *Seed Nodes*: A *Seed* node is a node that is used by a peer to get into the P2P network. There should be n (n>1) seed nodes active, this number n will be chosen arbitrarily for different instances of P2P network. A *Seed* node maintains a list of IP address and port number pairs of peers that have connected to it. This list is called *Peer List (PL)*. On startup, any new peer registers itself with at least $\lfloor (n/2) \rfloor + 1$ randomly chosen seed nodes. Registration request triggers an event at the seed node to add an entry containing the peer's IP and port number to the PL. For simplicity, in this project, seed nodes are not *peers*, that is, they do not perform actions of peers as described below. Seed Node helps in building a Peer to Peer Network by giving details about already registered Peer Nodes to new Peer Nodes.
   When a seed receives a 'dead node' message (details given below) from any peer, it should remove the dead node's details from the PL if present.

   Please note that for this to happen, the peers will not close connection formed initially with the seed nodes.

2. *Peer Nodes*: On launching the *peer* node, it first reads the IP Addresses and Port number of seeds from a config file. The IP Addresses and Port Numbers of the seeds will be hardcoded in this config file. You will be allowed to change only this config file at the time of evaluation (demo of project).

   The peer then randomly chooses $\lfloor (n/2) \rfloor + 1$ seed nodes and registers its identity (IP address: port) with them. After registration with the chosen seed nodes, a peer requests the list of peers available at the seed node. In response, the seed nodes send their peer list. The peer node then takes a union of all the peer lists that it received from different seeds and randomly selects a maximum of **4 distinct** peer nodes and establishes a TCP connection with each of them.

Along with this, every node maintains a data structure called *Message List(ML)* to efficiently broadcast a gossip message so that every message travels through a link (pair of connected peers) at most once. ML consists of the hash of the message, and information about which of its connected peers it has sent the message to or received the message from. ML can also have other fields as per your needs.

Also, the peer nodes have to check for the liveness of its connected peers periodically. So a node will send a liveness request message to all the peers it is connected to every 13 seconds. If a node gets this message, they have to reply only to the sender with the liveness reply message. If 3 consecutive liveness messages are not replied to, the node will notify the seed nodes that it is connected to, that this particular IP Address is not responding. The peer will then close the TCP connection with the node that isn't alive. Upon getting the message that a particular node is not alive, the seed will remove the details of the 'dead' node from it's PL.

Removing dead nodes from a PL ensures that peers do not try to connect to nodes which are offline. However, the scheme described above to remove dead nodes is open to security attacks by malicious nodes. Try to think of how a malicious peer can launch some type of attack, and also think of a solution to the attack which you propose. You need not implement the attack or the solution, but be prepared with answers at the time of the evaluation (demo).

**Gossip Message format**
A peer should generate a message of the form

<self.timestamp>:<self.IP>:<self.Msg#>

Such messages will be generated by each peer node every **5** seconds. The first message should be generated by a peer after it connects to selected neighbors after registration. A peer will stop message generation after it generates 10 messages.

**Gossip protocol**
After a node (peer) generates a message $M$, it transmits $M$ to all its adjacent nodes. On receipt of a message for the first time, a node makes an entry in the ML(<hash($M$),*true*>) and forwards it to all peers except the peer from which it got the message. On receiving the same message subsequently, the node just ignores it. Maintaining ML will prevent the nodes from forwarding the same Message more than once. This is to avoid unnecessary forwarding, and to prevent loops.

**Liveness Message Format**
The liveness request message format should be:

Liveness Request:<self.timestamp>:<self.IP >

The liveness reply message format should be:

Liveness Reply:<sender.timestamp >:<sender.IP >:<self.IP >

The 'self.timestamp' in liveness request must match 'sender.timestamp' in liveness reply. Similarly, 'self.IP' should match 'sender.IP' in liveness requests and replies respectively.

**Liveness Testing**
The liveness messages should be sent every 13 seconds, even after the gossip broadcast has stopped. The liveness messages won't stop until the node goes offline.

**Reporting the node as 'Dead'**
When 3 consecutive liveness requests do not receive a reply, the peer sends a message of the following format to all the seeds it is connected to:

Dead Node:<DeadNode.IP>:<DeadNode.Port>:<self.timestamp>:<self.IP>

**Block Mining**
We will simulate PoW mining in the following way. When a miner receives a block at say time $t$, it will generate an exponential random variable exponential random variable, say $\tau$. Note that $\tau$ depends on the hash power assigned to the miner. The miner will wait up to time $t + \tau$. If it has not yet received another block by then, it will generate a block and broadcast it. If on the other hand, it receives a block (which creates a longer chain than its current chain) before $t + \tau$, it will generate a new exponential random variable and the process repeats.

If *interarrivaltime* represents the average block interarrival in the entire network, then you can use the following steps to generate the exponential random variable described above.

- Overall block generation rate
  $meanT_k = 1.0/interarrivaltime$
- Scale individual miners lambda parameter based on the percentage of hash power it has.
  $lambda = nodeHashPower * meanT_k/100.0$
- Appropriately scale the the exponential waiting time
  $T_k := time.Duration(rand.ExpFloat64()/lambda)$

A newly joined node, say $N$, should sync with the blockchain's current height in the network. $N$ will receive the recent block $B_k$ where $k$ is the height, from the network immediately after joining the network. $N$ won't push the $B_k$ on to the chain; it will be in the pending queue. $N$ will request blocks from $B_0$(genesis block) to $B_{k-1}$ from the network and put each on the chain, post validation. All the block received after $B_k$ will enter to the pending queue. After the node $N$ sync till $k-1$, it starts processing the pending queue. For simplicity, the limit on the size of the pending queue is unrestricted. An honest node will begin creating and mining the block once the pending queue is empty.

While a node is mining two events can occur a) The node itself comes up with the solution, in our case, the timer expires, or b) The node receives the block from another node in the network and this block creates a longer chain than it had before receiving the block.

a)After node creates the block, it should follow the following steps :

- Store the block in the database.
- Broadcast the block to neighbors.

b)After receiving a block, the node should insert the block into the pending queue and abort the mining process if it's running.

Node validates the blocks in the pending queue one at a time. Validation consists of the following steps :

- Validate the received block (check if the hash in the block is the hash of some other exisiting block; timestamp was generated within 1 hour (plus or minus) of current time.) Reject the block if it fails the validation test, else do the following.
- Store it to the database (you are free to use any database which you are comfortable with).
- Broadcast the block to neighbors in the P2P network.
- If the block creates a chain longer than the longest chain currently and pending queue is empty, reset the timer for next block waiting time.

Note that node should always mine on the **longest chain** available locally.

**Block flooding attack :**
An adversary can flood the network with invalid blocks(Do not mine) to keep other node's pending queue full and hence suspend their block creation process. Note that an honest node won't forward the invalid blocks; thus, an adversary should explicitly flood a particular set of nodes with the sequence of blocks. Note that an invalid block won't go into the main chain. An adversary also parallelly mines and sends the valid blocks that follow the same process as the honest miner. You can implement this attack by making a node behave as an adversary and test the severity of attack for different inter-arrival times(starting from 2 sec), keeping mining power assigned to the adversary and % of nodes flooded in the network constant. Assume that the node behaving as an adversary consume 33% of the total hashPower. Severity is defined in terms of fractions of blocks mined by an adversary that ends up in the main chain with and without attack. You also have to observe the effect of the mentioned attack strategy on the mining power utilization(defined below).

**Definition 0.1. Mining power utilization :** Mining power utilization is the ratio of the number of blocks in the longest chain to the total number of blocks in the blockchain (including forks).

**Program Output**
You should submit a report along with your code on google classroom (submit a zip file). The report must contain the following plots :

- Mining power utilization(defined in Definition 0.1) Vs inter-arrival times.
- A Fraction of main chain blocks mined by the adversary Vs inter-arrival times
- A para clearly explaining your criterion determining the longest chain from the blocks stored in your database.

Run the experiment for 10%, 20%, and 30% of node flooded in the network and draw the above plots for each. All the plots should result from a minimum of 10 minutes run of the experiment(which we will verify from the timestamp maintained in the log file). Please note that the length of the longest chain and the entire blockchain that individual node see will be different, so you have to take the average over all the nodes while plotting the graph the mining power utilization.

While giving your demo, you will have to show the blockchain tree at any node using any graphics tool. You are free to use any graphics tool to draw the tree. We have mentioned a few potential graphics tools in the links below. Also, on the program's exit, you should be able to pop out the listed plots for one value of % of the node flooded.

**Useful Links** You can follow the links below to brush up your knowledge about some graph plotting tools.

1. Numpy.
2. Latex graph

**Submission Instructions:**

1. The assignment should be done in a group consisting of a maximum of **TWO** students.
2. The code should follow programming etiquette with appropriate comments.
3. Add a **README** file which includes a description of the code and gives detailed steps to compile and run the code.
4. Zip all your code and the report as a single file with the name **rollno1-rollno2.tar.gz** and upload it to google classroom. Only one group member should upload the file.
5. You should be able to **demonstrate** the code.