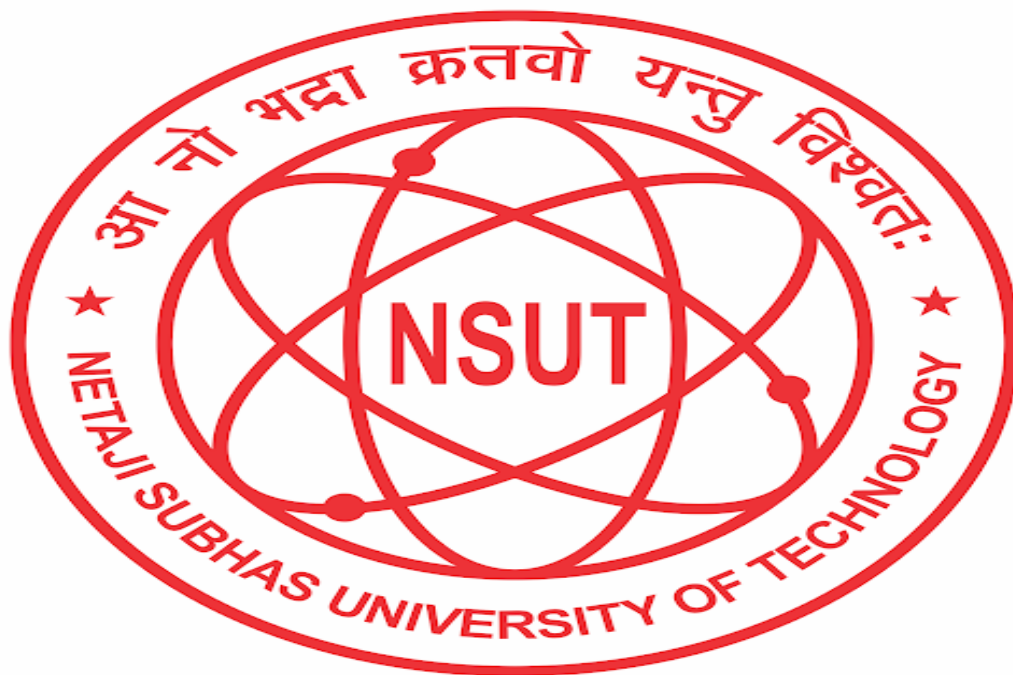


Simulation And Modeling

PROJECT REPORT

“Discrete Event Simulator for A P2P Cryptocurrency Network”

MATHEMATICS AND COMPUTING ENGINEERING



SUBMITTED TO:-

Ms. Riya Goyal

Ms. Surbhi Sharma

Team Members :-

Vikas(2019UMC4555)

Aditya(2019UMC4556)

Abhijeet Singh(2019UMC4559)

Discrete Event Simulator for A P2P Cryptocurrency Network

In this project we will build our own *discrete-event* simulator for a P2P cryptocurrency network. This project can be done in groups consisting of at most 3 persons.

Implementation Details

We have implemented the project in Python, and tested it on window machines. The core of the simulation is a priority queue which holds events prioritized by their scheduled time.

There are 4 types of events: Transaction Generate, Transaction Receive, Block Generate, Block Receive.

Since nodes keep generating events and the simulation is potentially infinite, we used a limit on the number of events that are run. This can also be passed as a parameter to the program.

At the end of the simulation, each node dumps its blockchain tree in the DOT graph description language, which are rendered into .png images using the Graphviz tool. We also created a graph depicting the connections between nodes to visualizing the network.

Libraries used :

- 1) [random](#) :- Python has a built-in module that we can use to make random numbers.
- 2) [argparse](#) :- The [argparse](#) module makes it easy to write user-friendly command line interfaces. The program defines what arguments it requires, and [argparse](#) will figure out how to parse those out of [sys.argv](#). The [argparse](#) module also automatically generates help and usage messages. The module will also issue errors when users give the program invalid arguments.
- 3) [os](#) :- This module provides a portable way of using operating system dependent functionality.

- 4) [queue](#) :- The [queue](#) module implements multi-producer, multi-consumer queues. It is especially useful in threaded programming when information must be exchanged safely between multiple threads.
- 5) [Collections](#) :- This module implements specialized container datatypes providing alternatives to Python's general purpose built-in containers, [dict](#), [list](#), [set](#), and [tuple](#).

Reasons of choosing the exponential distribution for inter-arrival between transactions:

The question concern with the time we need to wait before a given event (generate transaction) occurs. If this waiting time is unknown, it is often appropriate to think of it as a random variable having an exponential distribution. Roughly speaking, the time X we need to wait before an event occurs has an exponential distribution if the probability that the event occurs during a certain time interval is proportional to the length of that time interval. More precisely, X has an exponential distribution if the conditional probability:

$$P(t < X \leq t + \Delta t | X > t)$$

is approximately proportional to the length of Δt the time interval comprised between the times t and $t + \Delta t$, for any time instant t . In most practical situations this property is very realistic and this is the reason why we choose the exponential distribution for waiting time for generation of next event (transaction).

One of the most important properties of the exponential distribution is the **memoryless property**. When the event can occur more than once and the time elapsed between two successive occurrences is exponentially distributed and independent of previous occurrences, the number of occurrences of the event within a given unit of time has a Poisson distribution. The exponential distribution is also related to the Poisson distribution.

Node connection's

Each peer is connected to a random number of other peers. **For this** We are using **Uniform Distribution** with minimum value as $n/2$ and maximum value as n . The constraints on the range ensures that the graph is always connected.

latency calculations

Simulate latencies L_{ij} between pairs of peers i and j . Latency is the time between which a message m was transmitted from sender i and received by another node j . Choose the latency to be of the form $\rho_{ij} + jmj = cij + dij$, where ρ_{ij} is a positive minimum value corresponding to speed of light propagation delay, jmj denotes the length of the message in bits, cij is the bottleneck link speed between i and j in bits per second, and dij is the queuing delay on the path randomly chosen from an exponential distribution with some mean.

In real world scenario, the transmission depends on bandwidth. So to simulate that we use bottleneck link speed in latency calculation.

The bottleneck link speed Cij determines how fast a node can process its queue which in turn will determine the queue delay each message will have. Therefore Dij mean will depend upon Cij .

Summary of effects of choosing different values of simulation parameters:

Number of Nodes (N):

With z value fixed ($z = 0.3$), we observe the following.

When $n = 8$, the chain lengths of slower nodes relatively same as that of faster nodes.

As n value increases the difference between chain lengths of slower nodes and faster nodes increases, with faster nodes having longer chains.

With even higher values of n (~ 20), the slower nodes have very short chain lengths as compared to the faster nodes.

Fraction of Slow Nodes (Z):

With n value fixed ($n = 10$), we observe the following.

When $z = 0.1$, i.e. 10 % nodes are slow, the longest chain length of slow nodes is much shorter than that of the faster nodes.

As z increase the chain lengths of the slower nodes starts catching up to the faster nodes.

At around $z = 0.7$ to 0.8 the slower node chain length is almost the same as faster nodes.

Mean transaction interarrival time (T_m):

With $N = 10$, $Z = 0.3$, $B_m = 10$, and running simulation for 5000 events, we observe:

When $T_m = 1$, a lot of transactions are generated, resulting in the events queue being flooded with TransactionReceive events (66% of all events) This in turn results in not a lot of blocks getting propagated and the trees at all nodes are relatively sparse.

As T_m is increased the trees all nodes get relatively denser since lesser number of TransactionReceive events happen. In general, increasing T_m results in less transactions being generated for every node and vice versa.

Mean block interarrival time (B_m):

With $N = 10$, $Z = 0.3$, $T_m = 5$, and running simulation for 5000 events, we observe:

When $B_m = 1$, a lot of blocks are generated in a relatively small amount of time and since nodes aren't waiting to hear from other nodes and start extending the longest chain they have, the branching factor of the

resulting tree is high at both slow and fast nodes as a result of which the maximum chain length is relatively low.

As **Bm** is increased to be the branching factor of the tree decreases and the chains start getting relatively longer. In general, increasing **Bm** results in less blocks being generated for every node and vice versa.

Summary of the structure of trees for different types of nodes:

Fast: Block chain is of longer length.

Slow: Block chain is of shorter length.

Faster nodes basically have a high-speed network among themselves, as such they are able to receive more blocks than the slower nodes, so they have a longer chain length.

Low CPU (high Bm): The branching factor is lower

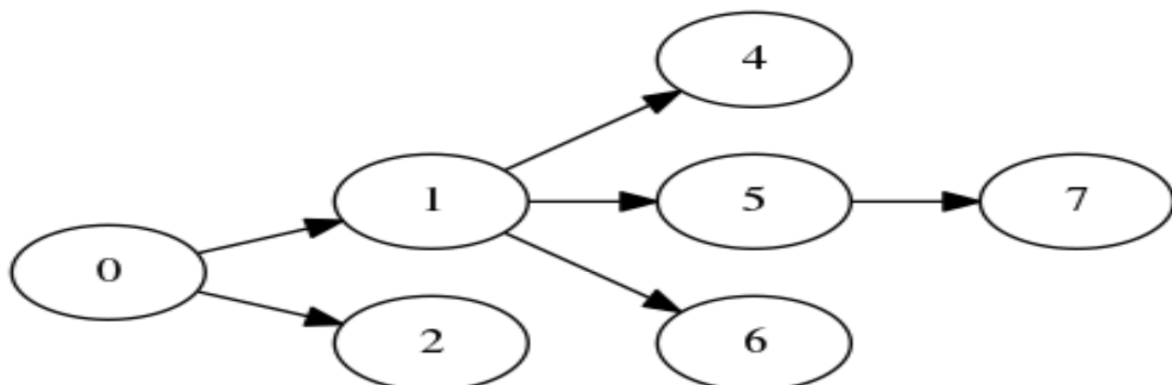
High CPU (low Bm): The branching factor is higher

If nodes have higher CPU power, they can generate more blocks in less time intervals so more duplicate blocks are generated that leads to more forks in the tree. As such the branching factor is higher.

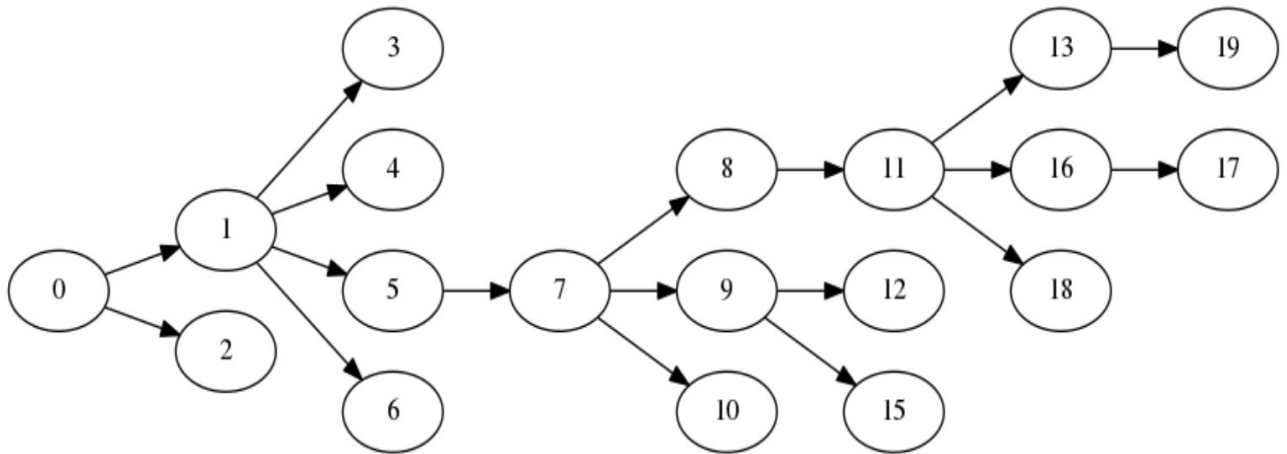
Pictures of typical trees

With parameters, $N = 10$, $Z = 0.3$, $T_m = 3$, $B_m = 10$ and running the simulation for 2000 events, we observe the following blockchain trees.

On a slow node:

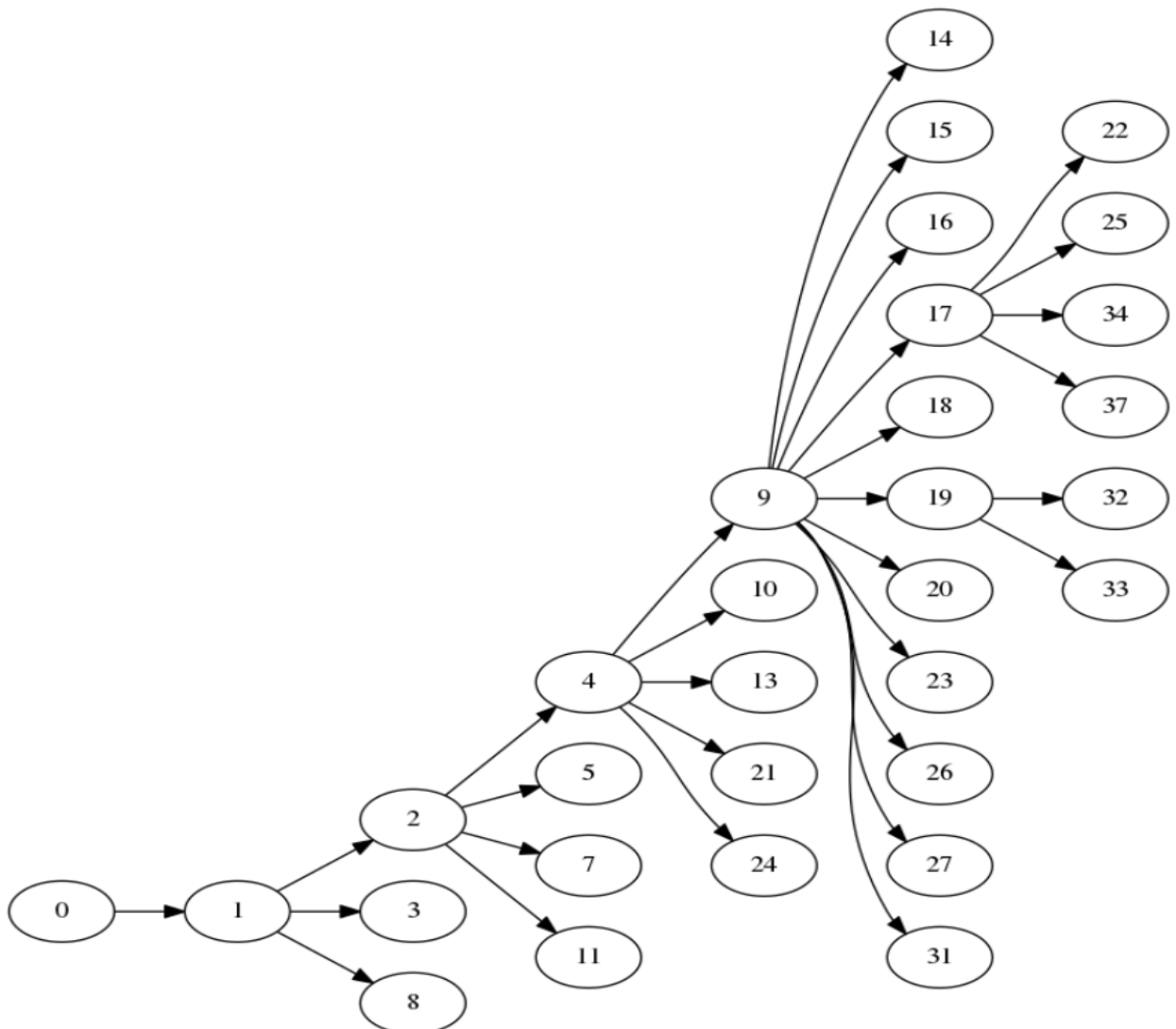


On a fast node:



This shows how fast nodes have denser trees and longer chains because they have the network advantage.

When lowering $B_m = 1$, we get high branching factors and shorter chains:



Running the code:

In the source directory, run: `python3 run.py [n] [z] [tm] [bm]`

Where, n is the number of nodes, z is the fraction of slow nodes and tm, bm are the mean transaction & block interarrival times respectively.

By default, simulations run for a maximum of 1000 events. To change that pass `--until` parameter and to suppress printing of event log use `-q`:

For eg: **`python run.py 10 0.3 3 10 -q --until 5000`**

Generated charts are stored in **output** directory.