CS620-Project: Algorand Simulator

Deadline: 15th April

1 ASim : An Discrete Event Simulator of Algorand

In this assignment you will build your own Discrete Event Simulator for the Algorand protocol [2]. This assignment can be done in groups of at most three. Let's call this simulator ASim. It must simulate following properties.

1.1 Network

1. Simulate a overlay network consisting of N nodes where each node controls a unique ECDSA public-private key pair. Unless otherwise stated use $N = \cancel{2000}$ **256**. You can use an already existing Cryptographic library for key generation. One such open source library for python can be found here [1].

2. For correct execution of your Simulator nodes must form a connected network - in other words there must be no partitions. Each node should be connected to m other nodes where m should be uniformly distributed among $\cancel{\{4, 5, \ldots, 8\}}$ $\{\mathbf{2}, \mathbf{3}, \mathbf{4}\}$.

3. In your simulator nodes will relay/broadcast two kinds of messages: Block messages and non-block messages. Delays on each edge should be drawn from $\max\{0, \mathcal{N}(200 \text{ ms}, 400 \text{ ms})\}$ and $\max\{\mathcal{N}(30 \text{ ms}, 64 \text{ ms})\}$ for Block and non-Block messages respectively. Here $\mathcal{N}(\mu, \sigma)$ refers to a Gaussian distribution with mean μ and standard deviation σ. You can use the same delay for both up-link and down-link. Also, these delays should be picked once at the beginning of your simulation and must remain fixed for the entire duration of your simulation.

4. Ignore bandwidth on each edge and assume that the any message relayed on an edge gets atomically delivered at the end of the link delay.

1.2 Gossip Protocol

1. To avoid message forgery, each message must be signed by the message originator. Relaying nodes must first validate the signature on the message before forwarding it. Data structure for each message should follow the format <Payload||Public Key|| Meta data for Signature verification> where "Payload" refers to the content of the message.

2. On receiving a message a node first validates its signatures and on correct validations gossips to all of its neighbors.

3. To avoid forwarding loops, a node gossips a message at most once.

1.3 Setup

1. At the beginning of the simulation, assign each node a stake amount chosen randomly and uniformly from the integers 1 to 50 (both inclusive).

2. All nodes must start with the genesis block at the beginning of the simulation. Your genesis block should contain the string "We are building the best Algorand Discrete Event Simulator".

3. Implement a function that works as a Pseudo-Random Generator (PRG). Each node should be able to call this function individually with their local input as seed for the PRG. On each input, your PRG should return a 256 bit integer. You can use an appropriate library of your choice for source of randomness.

1.4 Cryptographic Sortition

1. Use the PRG described in the previous section to generate a random string for each step in a round. The seed for the PRG should be:

   $$< \text{Hash of previous block}||\text{Round Number}||\text{Step Number} >$$

2. Use Asymmetric key signature as the Verifiable Random Function (VRF), i.e use the signature on the PRG output as the hash output of the VRF. Note that in class we used a Hash function of the signature. Here we are leaving out the Hash function.

3. Treating signature on the PRG output as the hash output of the VRF, implement sortition as mentioned in Section 5 of the Algorand paper [2]. Note that the number of selected sub-users of a node should be proportional to the amount of stake the node controls.

4. Consider three different threshold parameter $\tau_{\text{proposer}}, \tau_{\text{step}}, \tau_{\text{final}}$ to vary the number of nodes selected using sortition for role of Block proposer, Per step voting committee and Final voting committee respectively.

1.5 Block Proposal

1. After consensus on a block in the previous round, each node queries PRG with:

   $$< \text{Hash of the previous Block} \,||\, \text{Current round} \,||0 >$$

   Here round is equivalent to the block height. Consider block height of the genesis block as 0.

2. With the output of the PRG and $\tau_{\text{proposer}} = \cancel{20}\ \mathbf{5}$, each node should check whether it has been selected as a block proposer or not.

3. In case a node is selected as a block proposer, compute priority for each of the selected sub-user. Priorities should be computed as:

   $$\text{SHA256}(\text{VRF hash output}||\text{sub-user index})$$

   Lower the output of the SHA256 higher is the priority of the sub-user.

4. After computing priority, gossip a message with the highest priority sub-user information. Payload for the priority message should be :

   $$< \text{Round Number}||\text{VRF hash output}||\text{sub-user index}||\ \text{priority} >$$

5. Each proposer node will then wait for time $\lambda_{\text{proposer}} = 3$ seconds to hear priorities broadcast by other proposer nodes of the network. After $\lambda_{\text{proposer}}$ period, nodes using the received messages check whether it is the highest priority node or not.

6. The node with the highest priority creates a block proposal and broadcast it to the network. Format of Block-proposal message is:

   $$< \text{SHA256}(\text{Previous block})||\text{256 bit long random string} \,||\, \text{Node's Priority payload} >$$

   Here priority payload is the message content described in point (4) above. You should treat block proposal message as a Block message and consider its transmission delay accordingly.

7. Simultaneously, each node queries PRG with seed given as:

   $$< \text{Previous block hash}|| \text{Current round} \,||1 >$$

   Using the PRG output above, nodes run Cryptographic Sortition to check its selection in vote committee. You should use τ_{step} for this part.

8. Committee members wait for a period $\lambda_{\text{proposer}} + \lambda_{\text{block}} = 33$ seconds. If they do not hear a Block proposal from the highest priority block-proposer within this period they commit vote for a Empty Block. Structure for empty block vote is given as:

   $$< \text{Previous Block Hash}|| \text{ Round } || \text{ Step } ||\text{“Empty”}||\text{VRF hash output} >$$

   In the above treat “Empty” as a string.

9. For a given block proposal, the block structure is defined as:

   $$< \mathsf{SHA256}(\text{Previous block})||\text{256 bit long random string mentioned in proposal} >$$

   Similarly structure of an empty block in a particular round is:

   $$< \mathsf{SHA256}(\text{Previous block})||\text{“Empty”} >$$

   Here “Empty” is a string.

1.6 Voting and Reduction

On hearing a block proposal from the highest priority proposer, each node must proceed to the Reduction step of the protocol whose details are given below.

1. On hearing a block, each node must validate sortition of the block proposer. On successful validation, committee members cast vote for the block whose format is given as:

   $$< \text{Prev. Block Hash}||\text{Curr. Block Hash}||\text{Round}||\text{Step}||\#\text{Sub-user}||\text{VRF output} >$$

2. In case the Block is invalid, a committee node votes for an empty block as described earlier.

3. At the end of $\lambda_{\text{proposer}} + \lambda_{\text{block}}$ seconds, nodes wait for a additional period of $\lambda_{\text{step}} = 3$ seconds and count received votes for step one as given in Algorithm 5 of [2]. Note that as an inherent part of the Algorand protocol, beginning of the timer could be different for each node. Specifically, a node starts the timer at the instant when it realizes that a consensus has been reached on a block in the previous. This consensus could be either tentative or final.

4. Nodes then re-run Cryptographic Sortition with PRG seed:

   $$< \text{Previous block hash}|| \text{ Current round } ||1 >$$

   and proceed to step 2 of the reduction as given in Algorithm 7 of the paper.

1.7 BA⋆

1. With the output of the Reduction step, each node proceeds to Binary BA⋆ as shown in Algorithm 8 of the paper. Note the in Algorithm 8, the author initialize $step \leftarrow 1$. You should initialize *step* with 3 just to be consistent with your previous implementation. Also use MAX_STEPS = ~~15~~ **10**.

2 Experiments and Evaluations

With ASim you should run the following experiments and create a report containing the required graph and explanation. During these experiments, unless otherwise stated throughout your experiments keep your network fixed. Also in every round you should log the following information for each node:

- Cryptographic sortition results at each step
- Received proposal from other nodes (Sort them in descending order of their priority)
- Set of proposed block
- Received votes per proposed block in every step
- Block on which consensus is achieved

2.1 Stake based Cryptographic Sortition

Run ASim for ~~500~~ **64** blocks with the initial stake distribution as described above and plot the following graph.

1. For each step across all rounds, sum the number of sub-users selected for all nodes for each particular stake value between 1 and 50. Compute mean of these sums across all the steps in all rounds and plot the computed mean. Your x-axis should present stake $1, 2, \ldots, 50$ and y-axis should present the mean number of sub-user selected as computed above. Plot the result of this experiment for $\tau_{\text{step}} =$ ~~200~~ **32** and ~~400~~ **64**.

2. For each step across all rounds, sum the number of sub-users selected for all nodes for each stake value between 1 and 50. Evaluate the fraction of sub-users for each stake value in a single step. Then compute the mean of these fractions across all the steps in all rounds. and plot the computed mean. Your x-axis should present stake $1, 2, \ldots, 50$ and y-axis should present the mean fraction of sub-user selected for each stake as computed above. Plot the result of this experiment for $\tau_{\text{step}} =$ ~~200~~ **32** and ~~400~~ **64**

2.2 Highest Proposer and Network Delay

In this experiment we will evaluate how the view of each proposer changes regarding the highest proposer in a given round at the end of $\lambda_{\text{priority}}$. We want you to plot the following graphs:

1. Let r be the current round number and assume final consensus has been achieved on a block at round $r-1$. For this specific experiment you should add sufficient quiescent period after consensus on Block at round $r-1$ such that all nodes are aware of it.

2. Pick a global identical time instant after all nodes gets synchronized with the consensus of the previous round and let's call this time instant t_{init}. At t_{init} each node runs Cryptographic sortition for the current round and gossips their priority message immediately (if any).

3. At time $t_{\text{priority}} + t_{\text{init}}$, each proposer should have received priorities of a subset of other proposers.

4. For each proposer count the number of other proposer that considers it to be the highest priority proposer. Your x-axis should represent public keys of the proposers and y-axis should present the measured count. You can ignore the values with 0 count. You should also mention sum of all the counts. For cleaner representation of x-axis in your graph, you can put any pseudonym for the proposers as long as they are unique.

5. For a fixed $\tau_{\text{proposer}} =$ ~~20~~ **5** plot three different sub-graphs with non-Block message delay drawn from $\max\{0, \mathcal{N}(40ms, 64ms)\}$, $\max\{0, \mathcal{N}(60ms, 64ms)\}$ and $\max\{0, \mathcal{N}(100ms, 64ms)\}$.

6. Keeping link delay for non-block messages from $\max\{0, \mathcal{N}(40ms, 64ms)\}$, plot three graphs for $\tau_{\text{proposer}} =$ ~~10, 20 and 40~~ **5**, **10**, and **15**. You can re-use one plot from your previous graph.

2.3 Fail-Stop Adversary

In this experiment you will consider presence of adversary $\mathcal{A}$ who controls f fraction of the total nodes in every round. The choice of nodes for the adversary remain fixed in every round. Unless otherwise stated assume that $\mathcal{A}$ has to abide by the network protocol. In this experiment the adversary adopts the following strategy:

1. Just like any other node, in every round $\mathcal{A}$ performs cryptographic sortition to check for proposers for all the node it controls.

2. In case, any adversarial node gets selected as a proposer, $\mathcal{A}$ correctly computes the priority and gossips it to the network.

3. At this point, $\mathcal{A}$ decides to stop generating new messages like block-proposal and does not participate in consensus in that round. $\mathcal{A}$ however relays all messages by the honest nodes and start afresh his strategy in the next round.

With above adversarial strategy in place, run your simulator to generate ~~1000~~ **64** blocks (Including empty blocks) and plot the following graphs as well as answer questions.

1. Keeping $\lambda_{\text{proposer}}$ fixed vary f as ~~$\{0, 0.5, 0.10, \ldots, 0.25\}$~~ $\{\mathbf{0}, \mathbf{0.5}, \mathbf{0.10}\}$ and for each f compute the number of empty blocks in the above generated ~~1000~~ **64** blocks. Plot f on x-axis and Number of empty blocks on y-axis. Note that, you should pick appropriate τ_{proposer} to avoid "No-consensus" scenario.

2. Mathematically explain your reasoning behind picking a particular τ_{proposer}. Is there a bound on minimum τ_{proposer}? Why?

3. If your answer to the above is Yes, call τ^*_{proposer} as the minimum number of proposers and repeat your experiment for τ^*_{proposer} and $2\tau^*_{\text{proposer}}$ and plot them. Explain your observation.

4. Keeping $\tau_{\text{proposer}} = \tau^*_{\text{proposer}}$ fixed, plot the above graphs for non-Block message delay drawn from $\max\{0, \mathcal{N}(40ms, 64ms)\}$, $\max\{0, \mathcal{N}(60ms, 64ms)\}$ and $\max\{0, \mathcal{N}(100ms, 64ms)\}$. Explain your observation.

2.4 Byzantine Adversary

In this experiment you will Simulate an Byzantine adversary $\mathcal{A}$ and measure the impact of $\mathcal{A}$'s behavior on the protocol. Our Byzantine adversary $\mathcal{A}$ controls a f fraction of the node (fixed at the beginning of the protocol) in the system and adopts the following strategy:

1. All adversarial nodes honestly follow the protocol till gossiping of priority messages. In case one of the adversarial nodes is chosen as the highest-priority proposer, it proposes two conflicting blocks as the next block. Let's denotes these blocks with $\mathbf{B_1}$ and $\mathbf{B_2}$. Also this highest priority proposer magically transmits its blocks to all other adversarial node immediately. This is equivalent of a scenario where $\mathcal{A}$ is equipped with a very low-latency network connecting all the adversarial nodes.

2. All adversarial nodes on hearing the blocks proposed by $\mathcal{A}$ randomly picks one of them to gossip among its honest neighbors. Also, they don't relay the other Block at all.

3. During committee voting, all adversarial nodes vote for both the Blocks and relay them to other honest nodes. However, for all the honest committee votes, an adversarial node only relay votes for the block it gossiped during block proposal.

4. In case an honest node is selected as the block-proposer, $\mathcal{A}$ honestly follows the protocol.

With the above adversary in the system Plot the following:

1. Generate enough blocks such that adversarial nodes gets selected as the block proposer for ~~20~~ **5** rounds. Here you should work with $\tau^*_{\text{committee}}$ for a given f.

2. In each such round, plot the number of steps Algorand took to reach consensus. In case, Algorand reaches MAX_STEPS number of steps, (i) stop consensus protocol for that round, (ii) Unanimously select one of the proposed block to extend and (iii) initiate the protocol for next round.

3. Repeat the above experiment for ~~5~~ **3** different initial sets of adversarial nodes picked uniformly randomly from the entire network for both $f =$ ~~0.10~~ **0.05** and $f =$ ~~0.15~~ **0.10**.

4. Does changing the set of adversarial node affect the results? Explain your answer qualitatively. How does the number of required steps vary with increasing f.

References

[1] Pure-python ecdsa signature/verification. `https://github.com/warner/python-ecdsa`.

[2] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 51–68. ACM, 2017.