Homework 4

- Honor Code: You must work completely independently on this assignment. Do not discuss the questions or answers with each other before the assignment is due. Any breach of the honor code will be handled per the University's policy on academic honesty.
- Follow the instructions very carefully. Answers that do not conform to the instructions will not be given credit.
- Submit your solutions through Blackboard.
- Understand thoroughly all the code given to you in this lab. Search for documentation online if there is a primitive or API you have not encountered before.
- Use Java 8.
- Only use the external Java libraries provided to you in the lab, if any.

Overview

In this lab, you will design and implement a distributed consensus algorithm in a graph framework that's somewhat like the Ripple trust network. A trust network is an alternative method of resisting Sybil attacks and achieving consensus; it has the benefit of not wasting electricity like proof-of-work does.

Nodes in the network are either compliant or malicious. You will write a CompliantNode class (which implements a provided Node interface) that defines the behavior of each of the compliant nodes. The network is a directed random graph, where each edge represents a trust relationship. For example, if there is an A → B edge, it means that Node A listens to transactions broadcast by node B. We say that A trusts B, and that B is one of A's trusted nodes, and that A is one of B's listeners.

The provided Node class has the following API:

```
public interface Node {

    // {@code trustees[i]} is true if and only if this node trusts node {@code i}
    void setTrustedNodes(boolean[] trustedNodeFlag);

    // receive initial set of transactions
    void setInitialTransactions(Set<Transaction> initialTransactions);

    /**
     * @return proposals to send to nodes that trust this node.
     * REMEMBER: After final round, behavior of
     *          {@code getProposalsForTrustingNodes} changes and it should return
     *          the transactions upon which consensus has been reached.
     */
    Set<Transaction> getProposalsForTrustingNodes();

    //receive candidates from trusted nodes
    void receiveFromTrustedNodes(Set<Candidate> candidates);
}
```

Each node should succeed in achieving consensus with a network in which its peers are other nodes running the same code. Your algorithm should be designed such that a network of nodes receiving different sets of transactions can agree on a set to be accepted. We will be providing a Simulation class that generates a random trust graph. There will be a set number of rounds where during each round, your nodes will broadcast their proposal to their followers and at the end of the round, should have reached a consensus on what transactions should be agreed upon.

Each node will be given its list of trusted node via a boolean array whose indices correspond to nodes in the graph. A 'true' at index i indicates that node i is a trusted node, 'false' otherwise. That node will also be given a list of transactions (its proposal list) that it can broadcast to its listeners. Generating the initial transactions/proposal list will not be your responsibility. Assume that all transactions are valid and that invalid transactions cannot be created.

In testing, the nodes running your code may encounter a number (up to 45%) of malicious nodes that do not cooperate with your consensus algorithm. Nodes of your design should be able to withstand as many malicious nodes as possible and still achieve consensus. Malicious nodes may have arbitrary behavior. For instance, among other things, a malicious node might:

- Be functionally dead and never actually broadcast any transactions.
- Constantly broadcasts its own set of transactions and never accept transactions given to it.
- Change behavior between rounds to avoid detection.

The test scenario will have the following parameters:

- `p_graph`: the pairwise connectivity probability of the random graph: e.g. {.1, .2, .3}
- `p_malicious`: the probability that a node will be set to be malicious: e.g {.15, .30, .45}
- `p_txDistribution`: the probability that each of the initial valid transactions will be communicated: e.g. {.01, .05, .10}
- `numRounds`: the number of rounds in the simulation e.g. {10, 20}

Your focus will be on developing a robust CompliantNode class that will work in all combinations of the graph parameters. At the end of each round, your node will see the list of transactions that were broadcast to it.

Each test is measured based on:

- How large a set of nodes have reached consensus. A set of nodes only counts as having reached consensus if they all output the same list of transactions.
- The size of the set that consensus is reached on. You should strive to make the consensus set of transactions as large as possible

Hints:

- Your node will not know the network topology and should do its best to work in the general case. That said, be aware of how different topology might impact how you want to include transactions in your picture of consensus.
- Your CompliantNode code can assume that all Transactions it sees are valid -- the simulation code will only send you valid transactions (both initially and between rounds) and only the simulation code can create valid transactions.
- Use an iterative approach to development: start with a simple implementation of CompliantNode first and evaluate its effectiveness by running the scenarios and producing an overall raw score. Then, attempt to write a more complex implementation that achieves a higher overall raw score. Examine the outputted csv files after each run for scenario-specific information about the scenario's outcome.
- Ignore pathological cases that occur with extremely low probability, for example where a compliant node happens to pair with only malicious nodes. We will make sure that the actual tests cases do not have such scenarios.

Part 1 (80 points)

Instructions

1. Study the Ripple protocol by:
    a. Reading the original whitepaper on the Ripple consensus protocol: https://ripple.com/files/ripple_consensus_whitepaper.pdf

    b. Reading the illustrated article on the Ripple Ledger: https://ripple.com/build/ripple-ledger-consensus-process/
    c. Watching the video on how Ripple works

https://vimeo.com/64405422

2. Examine the BasicTests.writeStatistics method to understand how the raw score is calculated.
3. Implement an effective CompliantNode class by implementing each of the four methods defined in the Node interface.
4. Run the BasicTest Junit file to test your compliant node implementation on a set of provided trust graphs and transaction distributions. The test will calculate the raw score and generate a file that shows the different parameters of each test and how well your implementation performed. Note that grading will be based on different trust graphs, different transaction distributions, as well as different types of malicious nodes that have not been provided in the homework materials.

Optional: Devise up to three MaliciousNode implementations to ensure the effectiveness of your own CompliantNode implementation and to subvert the implementations of the other students in the class.

Submission: Your submission for this part will only be the CompliantNode.java class and any number of MaliciousNode_<NAME>_<X>.java classes, where <NAME> is your name (use only alphabet letters) and <X> is any number (e.g. MaliciousNode_RalphMerkle_5.java). Do not modify any of the other existing classes, as the auto-grading script will use the original implementations of all the other classes. Do not zip your files. Upload each file separately.

Grading: Your grade will be relative to other students. A raw score will be calculated that measures how effective your CompliantNode implementation is at achieving consensus in a network with malicious nodes. The student with the highest raw score R will receive a 100%. The rest of the students will have their raw scores scaled by 100 – R. For example, if Alice has the highest raw score of 80 and Bob has a raw score of 70, then Alice gets a 100% on the assignment, and Bob will get **at least** a 90%. See BasicTests.writeStatistics for how raw scores are calculated.

Additionally, each student can **optionally** propose up to three MaliciousNodes to be used in the grading process for all students. Consequently, each student's grade will depend on the effectiveness of his/her CompliantNode implementation as well as the MaliciousNodes proposed by other students. Three malicious nodes (MaliciousNode0, MaliciousNode1, and MaliciousNode3) have already been provided.

Thus, you are motivated to devise effective MaliciousNodes to test both your own CompliantNode implementation as well as to undermine the CompliantNode implementations of other students, thereby increasing your grade.

The BasicTests.test0 must take less than 5GB of memory to run and complete in less than 10 minutes.

Part 2 (20 points)

Instructions

1. For MaliciousNode3, and generate a line graph that visualizes the relationship between the p_graph parameter and the maximum number of nodes in consensus. Similarly, generate line graphs for the p_malicious parameter, p_txDistribution parameter, and the numRounds parameter.
2. Clearly label the title and axes of each graph, and include information about the fixed values for the other parameters. An insufficiently labelled graph will not receive credit.
3. Use the Simulation class to generate trust graphs scenarios. For each data point, generate 10 random scenarios and average their outcomes to produce a single value. For example, for the p_malicious line graph, you could use values of .10, .20, .30, .40. Fix the values for the rest of the parameters, and then run 10 random scenarios where p_malicious=.1 and average the 10 values you produced for the maximum number of nodes in consensus in each scenario.
4. Explain conceptually why the relationships visualized each graph for each parameter makes sense.

Submission

Include all four graphs in a single PDF.

Frequently Asked Questions

Question 1: I think the fact that all the transactions are valid is confusing me. If everything is valid what are we agreeing/disagreeing on?

Answer: A transaction is only valid in the sense that its structure follows the protocol's specification. For example, its signature is verifiable, and the transaction references existing accounts on the ledger.

The purpose of the distributed consensus protocol isn't to check the validity of transactions. Remember that in a decentralized system, like the Ripple network, there is no centralized database that serves as the single source of truth for the ledger.

The nodes must agree, every epoch, on what the current state of the ledger is by only communicating with their immediate neighbors in the network. Specifically, the nodes need to agree about which transactions will be added to the ledger during the current epoch, and which transactions will not. A valid transaction may very well be rejected during the current epoch, but accepted in a later epoch.

Question 2: Where is the ledger?

Answer: In the actual Ripple network, each node keeps its own copy of what it believes is the ledger as a result of participating in the distributed consensus protocol every epoch. However, this assignment only requires you to simulate a single epoch. Your goal is to calculate the "last-closed ledger" corresponding to this epoch. See the Ripple consensus white paper and Ripple video for an explanation on what the "last-closed ledger" means.

Question 3: What is the mechanism that (as a node) says this transaction is agreed on?

Answer: This is your task for the assignment. You will devise a mechanism that a node can use to determine which transactions to accept as part of its proposal for what the consensus should be. All honest nodes run the same code. Here's a trivial example: suppose honest nodes never agree to accept any transaction to the ledger. This is a legitimate strategy that results in guaranteed consensus among all honest nodes: they all agree to an empty set of transactions. Obviously, this isn't a useful consensus protocol, but it's a consensus nonetheless. Your goal is to devise a strategy the honest nodes can use to maximize the number of transactions that all nodes agree should be part of the consensus for this epoch.

Question 4: Can you explain the difference between a proposal and a candidate? From what I understand the proposal is what the network is trying to agree on, and the candidate is the other nodes' "votes". So at the end of the first round the nodes with at least 50% of votes will become proposals again?

Answer: A node receives candidate transactions from its trusted nodes. A node sends proposal transactions to nodes that trust it.

Your goal is to write code so that the node can intelligently decide which of the candidate transactions it receives from its trusted nodes should be sent as proposals nodes that trust it. In other words, a node needs to decide which candidates it has received thus far this epoch should become proposals.

Question 5: We are given access to each Scenario whose members are mostly non-public. How much of that can we access?

Answer: The non-public access is intentional and resembles reality. Nodes in a real network have limited, if any, information about the extent of the network and the configurations of other nodes. You do have access to several public methods in Scenario, which may be useful.

Question 6. I have a solution that works well but can't beat the trivial solution with respect to the raw score formula.

Answer: If you have developed a strong compliant node that has advantages not fully appreciated by the raw score formula, please do submit it with a write up justifying its effectiveness. The intent of the raw score formula is emphasize that a meaningful consensus maximizes both the nodes in consensus as well as the number of transactions in consensus. However, ultimately, the number of nodes in consensus is more valuable than the absolute number of transactions in consensus, as long as, in future epochs, the missed transactions eventually become part of the consensus.

Question 7: Do we need to submit any code for part two?

Answer: Part 2 does not require you submit any code. You only need to submit the graphs which can be made using excel. Your explanations of the graphs shouldn't need to exceed more than three sentences.