Bitcoin and Cryptocurrency Technologies

Assignment 2: Consensus from trust

For this assignment, you will design and implement a distributed consensus algorithm given a graph of "trust" relationships between nodes. This 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 B listens to transactions broadcast by node A. We say that B is A's follower and A is B's followee.

The provided `Node` class has the following API:

```java
public interface Node {

    // NOTE: Node is an interface and does not have a constructor.
    // However, your CompliantNode.java class requires a 4 argument
    // constructor as defined in the provided CompliantNode.java.
    // This constructor gives your node information about the simulation
    // including the number of rounds it will run for.

    /** {@code followees[i]} is true if this node follows node {@code i} */
    void setFollowees(boolean[] followees);

    /** initialize proposal list of transactions */
    void setPendingTransaction(Set<Transaction> pendingTransactions);

    /**
     * @return proposals to send to my followers. REMEMBER: After final round,
     * behavior of {@code getProposals} changes and it should return the
     * transactions upon which consensus has been reached.
     */
    Set<Transaction> sendToFollowers();

    /** receive candidates from other nodes. */
    void receiveFromFollowees(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 followees via a boolean array whose indices correspond to nodes in the graph. A 'true' at index *i* indicates that node *i* is a followee, 'false' otherwise. That node will also be given a list of transactions (its proposal list) that it can broadcast to its followers. 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.

You will be provided the following files:

`Node.java`	a basic interface for your `CompliantNode` class
`CompliantNode.java`	A class skeleton for your `CompliantNode` class. You should develop your code based off of the template this file provides.
`Candidate.java`	a simple class to describe candidate transactions your node recieves
`MaliciousNode.java`	a very simple example of a malicious node
`Simulation.java`	a basic graph generator that you may use to run your own simulations with varying graph parameters (described below) and test your `CompliantNode` class
`Transaction.java`	the `Transaction` class, a transaction being merely a wrapper around a unique identifier (i.e., the validity and semantics of transactions are irrelevant to this assignment)

The graph of nodes will have the following parameters:
- the pairwise connectivity probability of the random graph: e.g. {.1, .2, .3}
- the probability that a node will be set to be malicious: e.g {.15, .30, .45}
- the probability that each of the initial valid transactions will be communicated: e.g. {.01, .05, .10}
- 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.
- Execution time, which should be within reason (if your code takes too long, the grading script will time out and you will be able to resubmit your code).

Some 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 has the ability to create valid transactions.
● 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.