CCS 5594

Homework Assignment 2

Given: Feb 29, 2024 **Due:** Mar 21, 2024

General directions. The point value of each problem is shown in [ ]. The completed assignment must be submitted on Canvas as **a ZIP containing all code files of your working framework** by 11:59 PM on Mar 21, 2024. **No late homework will be accepted.**

---

[75] 1. Implement the logic used to process transactions and produce the ledger as follows.

In ScroogeCoin, the central authority Scrooge receives transactions from users. Scrooge organizes transactions into time periods or blocks. In each block, Scrooge will receive a list of transactions, validate the transactions he receives, and publish a list of validated transactions.

Note that a transaction can reference another in the same block. Also, among the transactions received by Scrooge in a single block, more than one transaction may spend the same output. This would of course be a double-spend, and hence invalid. This means that transactions cannot be validated in isolation; it is a tricky problem to choose a subset of transactions that are together valid.

You will be provided with a Transaction class that represents a ScroogeCoin transaction and has inner classes Transaction.Input and Transaction.Output.

A transaction output consists of a value and a public key to which it is being paid. For the crypto functions, we use the PyCryptodom library (https://pycryptodome.readthedocs.io/en/latest/src/installation.html)

A transaction input consists of *(i)* the hash of the transaction that contains the corresponding output, *(ii)* the index of this output in that transaction (indices are simply integers starting from 0), and *(iii)* a digital signature. For the input to be valid, the signature it contains must be a valid signature over the current transaction with the public key in the spent output.

More specifically, the raw data that is signed is obtained from the getRawDataToSign(index: int) method. To verify a signature, you will use verifySignature() method included in the provided file Crypto.py:

verifySignature(pubkey: RSA.RsaKey, message: bytes, signature: bytes)

This method takes a public key, a message and a signature, and returns true if and only if signature correctly verifies over message with public key.

Note: You are only given code to verify signatures, and this is all that you will need for this assignment. The computation of signatures is done outside the Transaction class by an entity that knows the appropriate private keys.

A transaction consists of a list of inputs, a list of outputs and a unique ID (see the getRawTx() method).

The class also contains methods to add and remove an input, add an output, compute digests to sign/hash, add a signature to an input, and compute and store the hash of the transaction once all inputs/outputs/signatures have been added.

You will also be provided with a UTXO class that represents an unspent transaction output. A UTXO contains the hash of the transaction from which it originates as well as its index within that transaction. We have included eq, hash, and compare functions in UTXO that allow the testing of equality and comparison between two UTXOs based on their indices and the contents of their txHash arrays.

Further, you will be provided with a UTXOPool class that represents the current set of outstanding UTXOs and contains a map from each UTXOs to its corresponding transaction output. This class contains constructors to create a new empty UTXOPool or a copy of a given UTXO, and methods to add and remove UTXOs from the pool, get the output corresponding to a given UTXOs, check if a UTXO is in the pool, and get a list of all UTXOs in the pool.

In summary, you will be responsible for creating a file called TxHandler.py that implements the following API:

```
class TxHandler:

    # Creates a public ledger whose current UTXOPool (collection of unspent * transaction
    outputs) is utxoPool. This should make a defensive copy of utxoPool by using the
    UTXOPool(UTXOPool uPool) constructor. */

    def __init__(self, pool: UTXOPool):

  # Returns true if
  # (1) all outputs claimed by tx are in the current UTXO pool,
  # (2) the signatures on each input of tx are valid,
  # (3) no UTXO is claimed multiple times by tx,
  # (4) all of tx's output values are non-negative, and
  # (5) sum of tx's input values is greater than or equal to sum of its output value;
  # and false otherwise. */
    def isValidTx(self, tx: Transaction) -> bool:


   # Handles each epoch by receiving an unordered array of proposed * transaction
 # checking each transaction for correctness, returning a mutually valid array of
 # accepted transactions, and updating the current UTXO pool as appropriate.*/
  def handleTxs(self, txs):
```

Your implementation of handleTxs() should return a mutually valid transaction set of maximal size (one that can't be enlarged simply by adding more transactions). It need not compute a set of maximum size (one for which there is no larger mutually valid transaction set).

Based on the transactions it has chosen to accept, handleTxs should also update its internal UTXOPool to reflect the current set of unspent transaction outputs, so that future calls to handleTxs() and isValidTx() are able to correctly process/validate transactions that claim outputs from transactions that were accepted in a previous call to HandleTxs().

[50] 2. 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:

```
class Node:
  # NOTE: Node is an interface and does not have a constructor.
  # However, your CompliantNode.py class requires a 4 argument constructor as defined in
  # the provided CompliantNode.py.
  # 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}
  def setFollowees(self, followees):

    # initialize proposal list of transactions
    def setPendingTransaction(self, 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.
  def sendToFollowers(self) -> set:

    # receive candidates from other nodes.
  def receiveFromFollowees(self, candidates: set):
```

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.

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.py	a basic interface for your CompliantNode class
CompliantNode.py	A class skeleton for your CompliantNode class. You should develop your code based off of the template this file provides.
Candidate.py	A simple class to describe candidate transactions your node receives
MaliciousNode.py	A very simple example of a malicious node
Simulation.py	A basic graph generator that you may use to run your own simulations with varying graph parameters and test your CompliantNode class
Transaction.py	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 problem*)

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 should be within reason.

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.
- Malicious nodes are more destructive, sometimes they act as honest nodes, sometimes not. Thus, identifying the malicious node is difficult and resource-consuming. At this level, you do not need to give an optimal solution, just propose the one with your assumptions and arguments that are reasonable to demonstrate that it yields a good result.