COP 701

Distributed Ledger

Team 0:

Harish Chandra Thuwal	-	2017 MCS 2074
Shadab Zafar	-	2017 MCS 2076

Programming Language: Python 3.6

Development Environment: Ubuntu

stdlib modules

- `threading`
    - For the simulation, we'll use a thread per node.
    - None of the threads will share any data structures since they represent nodes connected via a network.
- `sockets, rpclib`
    - Nodes will talk to each other via sockets which will preferably be UDP because no setup/teardown packets are required.
    - RPC methods will be implemented to allow nodes to send commands to other nodes.
- `asyncio`
    - Each node needs to be able to perform multiple tasks like handle incoming connections, broadcasting messages etc.
    - An asyncio eventloop (per thread) will be used to perform these tasks concurrently without needing to run multiple threads per node.
    - This module is a recent addition to Python and is the reason we want to stick to Python 3.6

- `json, tnetstring`
    - These modules will allow serialization of objects such as transactions etc. so that they can be sent over the wire to other nodes.
    - We have not yet decided which of json or tnetstring we'll use.

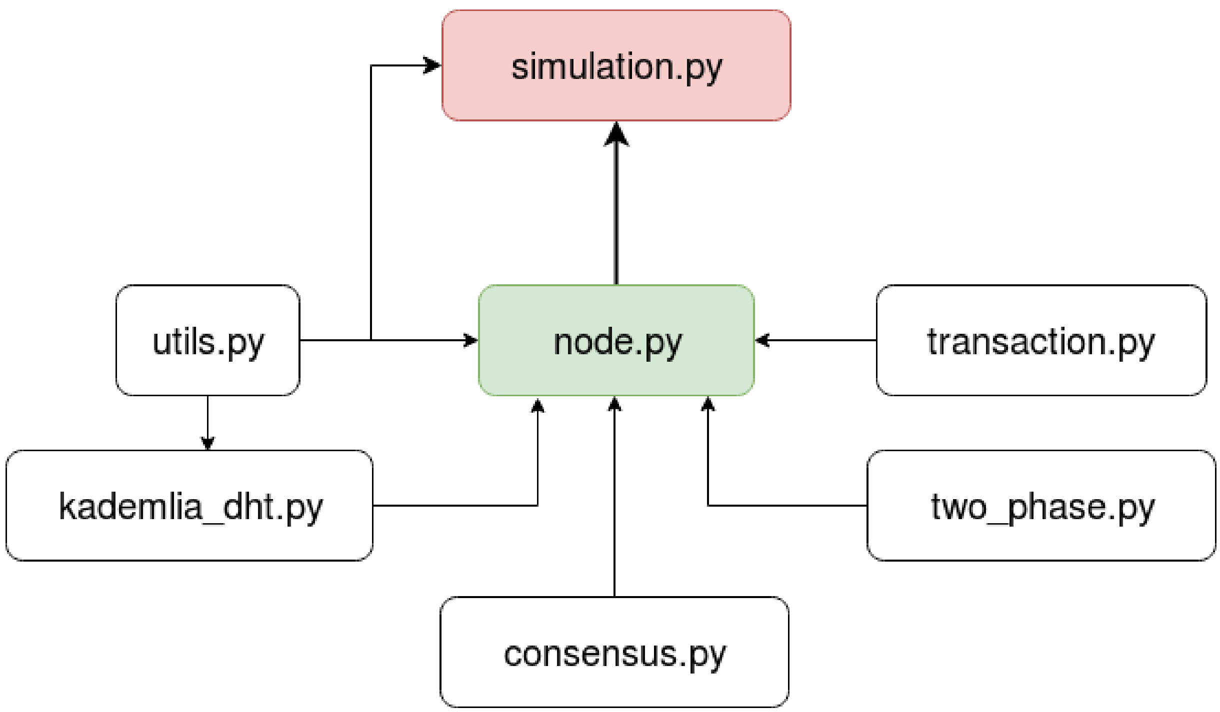
- `unittest, py.test`
    - `unittest` allows writing of test cases for the modules.
    - `py.test` is the associated test runner that runs those test cases and provides results.

- `secrets, hashlib`
    - SHA 1 hashes will be used for node_ids and as the key space for the Kademlia DHT.

- `ecdsa, pycrypto`
    - Generation of public, private key pair and digital signatures.

- `logging`
    - Each node will log its output/status to a separate file named `<node_id>.txt` in a common `logs` directory.

System Design



`simulation.py`

This is the main caller (akin to a `main` function) that will control the simulation and in turn perform all the tasks defined in the assignment.

For Task 1:

- Initialize and run 1st node. This will act as a bootstrapper for the rest of the nodes.
- Initialize and run the other N-1 nodes.
- Because this is a simulation, we'll maintain a list of all nodes (threads) currently running, the port they're listening on etc.
    - So we can send messages to nodes to perform the tasks that follow.

For Task 2:

- Arbitrarily decide 3 nodes to act as a sender, receiver & witness.
- Let the 3 nodes know what their roles are so they can begin a 2 Phase Commit.
- Since any of the nodes can fail and 2PC may not succeed - the simulation controller will keep track of it. And restart this task if it failed.

For Task 3:

- Once a transaction is successfully committed the sender will broadcast the transaction to its neighbors which will in turn verify it and broadcast it to their neighbors.
- Ensure that the broadcast reaches every node.
- Ensure that the broadcast does not keep looping around in the network. Once the broadcast has reached everyone.
- A node should discard the broadcast message if it has already received and forwarded that message.

For Task 4:

- Check that each node on reception of the broadcast first verifies it (both the digital signature as well as the input transactions) and then adds it to its transaction list

- An unverified message should not be present in the transaction list.

For Task 5:

- Initiate several transactions (2 Phase commits) so that multiple transactions are broadcasted concurrently.

- Compare the hashes of transaction lists of all the nodes to ensure all nodes have perceived the same order of all broadcasts.

For Task 6:

- Deliberately initiate a double spend transaction.

- That is initiate a transaction with at least one already used input transaction.

- Ensure that the nodes detect this double spend during verification and do not accept the transaction.

`node.py`

A node represents an individual system running independently of, but in conjunction with, other nodes.

Since we're using threads to simulate a node, the `Node` Class will inherit from `threading` and from the `Kademlia` class since each node will also be a part of the DHT.

The core of each node will consist of the following asynchronous functions running inside an event loop.

- Server
    - Continuously listens for new connection requests from other nodes.
    - On accepting a connection, a new async task is created that handles the connection.
    - The server then continues to listen for new requests.
- Handler
    - The handler will consist of several Remote Procedure Calls (RPCs).
    - The RPCs will be used by the nodes involved in the connection to perform actions and issue commands to each other.
- Failure
    - This method will cause the node to exit with a random probability.
    - This will simulate a real-world scenario where each node can suddenly fail or may choose to become online without informing anyone (fail stop failure mode).

kademlia_dht.py

After researching about various DHT implementations, we've decided to stick with Kademlia as this is what BitTorrent and a lot of other Peer to Peer systems use and so there's a lot of documentation available.

This module will implement the Kademlia Protocol including the routing etc.

transaction.py

This module will have a class representing a Transaction object with methods that allow us to create an empty transaction, serialize the transaction, and verify it.

The verification process will iterate over all the transactions in a node's ledger and ensure that no double-spend is possible.

We plan to keep a separate `unspent_transactions` set to improve the efficiency of the verification process.

utils.py

This module will contain functions that don't seem to be a part of any other module and will be used by multiple modules.

Some examples:

- generate_public_private_pair()
- generate_sha_hash()
- compare_hash()

`two_phase.py`

This module will deal with the distributed two-phase commit protocol including methods for both coordinator & cohorts.

`consensus.py`

To attain a common ordering of transactions on all nodes - a consensus algorithm will be required. We're not too sure of what we will use to achieve this.

Possible candidates are:

- ABCAST (and therefore Virtual Synchrony)
- Paxos
- Raft

Rather than implementing these algorithms from scratch we'll understand their workings and then use an external library.

This module will mostly contain the helper functions that integrate such a library with our code.