MP2: Cryptocurrency

CP1: Due Mar 25, 11:55pm

CP2: Due Apr 8, 11:55pm

For this MP, you are going to implement some basic functionality of a cryptocurrency. In particular, you will implement a peer-to-peer protocol to communicate transactions and blocks, rule verification to ensure transactions are well-formed, and Nakamoto consensus and its longest chain rule.

To save the CPUs of the CS VM cluster, you will not be implementing proof of work, instead you will be relying on a service, which we will provide, that will emulate this for you. This service will also provide you with an introduction functionality and generate client transactions.

In addition to building the protocols, you will be asked to perform some experiments to analyze the performance of your cryptocurrency implementation. You will get full marks for a correctly functioning and well-analyzed implementation, but you will get bonus points (and perhaps some VC investors!) for a design that exhibits high performance.

So get coding—your virtual billions are waiting!

Checkpoint 1: Transaction propagation

To maintain a distributed ledger, all nodes must agree on a set of transactions. The first step towards this is to distribute transaction information to all of the nodes. A typical approach taken by cryptocurrencies is to use gossip, but you can consider other designs too.

Node connectivity

The set of nodes in your system is dynamic, with new nodes joining and old nodes leaving. You will need to ensure that the overall network maintains connectivity despite this. To help you in this task, we will provide you with an introduction service.

When you first connect to the introduction service (over TCP), you should send a `CONNECT` command.

```
CONNECT node1 172.22.156.2 4444
```

The command should include a symbolic name for your node, followed by information about your node that other nodes will need for connection. Typically this would be the IP address and port on which the node is listening for connections.

After receiving a `CONNECT` command, the service will introduce the new node to up to 3 existing nodes:

```
INTRODUCE node2 172.22.156.3 4567
INTRODUCE node7 172.22.156.99 8888
INTRODUCE node12 172.22.156.12 4444
```

You may connect to one or all of these nodes. You will need to use these initial nodes to bootstrap connectivity to the rest of the network. Your network should be remain connected even if a large number of nodes (up to half) fail. A common approach is to periodically query your neighbors for a list of known nodes and randomly connect to several.

Transaction broadcast

The service will also act as a client and send transactions to the nodes. A transaction looks like:

```
TRANSACTION 1551208414.204385 f78480653bf33e3fd700ee8fae89d53064c8dfa6 183 99 10
```

The parameters of the transaction are:

- a timestamp, represented as a floating point number. (Seconds since January 1, 1970, with microsecond precision.)
- a 128-bit unique transaction ID, encoded as hexadecimal digits
- a source account number
- a destination account number
- transaction amount encoded as an integer

Note that for this checkpoint you do not need to worry about the semantics of the transaction. Your goal, instead, is to ensure that every single node in the system receives a copy of every transaction. Note also that the transaction messages are sent asynchronously, not in response to any command sent by the node.

A typical approach would be to gossip transaction information among the nodes, but you can implement any dissemination strategy you wish.

Node termination

The service can tell a node to terminate itself by sending either a `QUIT` or a `DIE` message. `QUIT` allows for graceful termination, where a node can perform some cleanup tasks. `DIE` *must* exit the process immediately without any further cleanup actions.

You may use the same implementation for `QUIT` and `DIE`, but if you are having trouble getting your system to be robust with abrupt node failures, you may try to implement cleanup, in which case you will lose some robustness points.

Logging and evaluation

Your nodes should perform logging to be able to analyze correctness and performance. You should log the connections made between nodes, and transactions received from either the service or other nodes. Your logs should be detailed enough so that you can a) verify that a transaction reaches all nodes in the system, b) track the propagation speed of a transaction (how long until it reaches all nodes? how long until it reaches half the nodes?), c) track the bandwidth used by the system. You may want to write some scripts to process the logs and extract the relevant information.

To evaluate your system, you should perform the following experiment:

1. Start 20 nodes
2. Configure the service to generate 1 transaction per second
3. Run for a few minutes to measure transaction propagation delay
4. Kill half the nodes
5. Run for a few minutes to verify functionality and measure transaction propagation delay

You should then repeat the same experiment, scaling up to 100 nodes initially and 20 transactions per second.

Your CP1 report should include a graph of the propagation delay and bandwidth used by your system. The suggested format is to use either the transaction number (implicit) or its timestamp on the x axis, and create graphs plotting the minimum, maximum, and median propagation delay, as well as the number of nodes the transaction reached (to validate reachability). You should also plot the bandwidth used by your system in aggregate across time.

Running the introduction service

To run the introduction service, you will need to install python36 if it is not installed already:

```
sudo yum install python36
```

and then download the service here

To run it, you will need to run:

```
python36 mp2_service.py <port> <tx_rate>
```

The *port* is the TCP port that it will listen on, and the *tx_rate* will be the rate at which the service will generate transactions. Note that this is a global rate, so each node will receive new transactions at the appropriate fraction of the rate. The transactions arrive using a randomized process, so they may arrive somewhat more or less frequently in any given period.

The service will print diangostic messages to the console about which nodes it is connected to and the transactions it is generating. You may also type `thanos` into the terminal (followed by a newline) to cause it to send a `DIE` command to half the nodes, selected at random—this may be helpful for you to carry out the experiments as described above.

Note that we will be using our own version of the service in testing, which may have slightly different behavior.

CP1 submission

Please submit CP1 using Gradescope. Your submission should include the following information:

1. The list of people in your group with names and NetIDs
2. The URL to your GitLab repository. You must use the Engineering GitLab server and give the course staff Reporter access to your repository.
3. Your group number
4. A description of how to compile (if needed) and run your nodes

You should then have a description of the design of your MP2. Your description should explain:

1. How your nodes keep connectivity; how they discover nodes beyond the originally introduced ones, and how they detect failed nodes. You should justify why you think your design is robust to failures.
2. How transactions are propagated. Describe the algorithm you are using, any parameters and how you arrived at them.
3. What information is logged and how you used this to generate the graphs. Please make sure that the logs you used in your experiments are checked into the git repository. If you wrote any scripts to analyze the logs please include them in the repo and describe how they work.

As a guideline, each of the three points above should be one or two paragraphs.

Finally, you should have the graphs of transaction propagation and bandwidth from the experiments described above.

Checkpoint 2: Nakamoto Consensus

In this checkpoint, you will be responsible for "mining" blocks of transactions. Each node should collect previously unmined transactions, verify their correctness, and put them into a block. It will then use the service to try to get a proof-of-work token for the block, allowing it to "mine" it. If it gets the token, it will then broadcast the block to all the other nodes, using the same mechanism as in CP1. In case a node receives several conflicting blocks, it will use the longest-chain rule to pick between them.

Solving Puzzles

The mp2 service will solve the puzzles for you. To start, you will need to compute the SHA256 hash of the first parts of the block: the reference to the previous block and the list of transactions you want to include. (You will need to serialize your block to a sequence of bytes.) You will then need to request a puzzle solution from the service by issuing a `SOLVE` command, e.g.:

```
SOLVE 86777674e3fe09e0da911be4c7bce219794a8988955508d3e9433d8584630b1f
```

The service will start trying to solve the puzzle. When a solution is available, it will return it to you:

```
SOLVED 86777674e3fe09e0da911be4c7bce219794a8988955508d3e9433d8584630b1f 251a76a3bc860cb9f599bb2d6e183753120dcd26c211e3c20f16337b53e4
```

The first hash is the puzzle input that you provided, and the second hash is the solution to the puzzle. You should add the solution to your block and broadcast it to all the other nodes.

Note that a solution may take some time to compute; in the mean time the service may end up sending more transactions to your node and you should process them. (In other words, the puzzle solution will arrive asynchronously.) If you want to start working on a new block (e.g., if you received a block from another node, or if you added some transactions to your current block), you should issue a new `SOLVE` command; the service will then abandon your previous `SOLVE` request and start trying to solve the new puzzle.

Because we are simulating proof-of-work here, you will also need to use the service to verify the solutions. When you receive a block, you should compute the SHA256 hash of the block minus the solution component, and then issue a verify command to the service, supplying the block hash and the solution included in the block.

```
VERIFY 86777674e3fe09e0da911be4c7bce219794a8988955508d3e9433d8584630b1f 251a76a3bc860cb9f599bb2d6e183753120dcd26c211e3c20f16337b53e4
```

The service will verify that the solution is correct and reply:

```
VERIFY OK 86777674e3fe09e0da911be4c7bce219794a8988955508d3e9433d8584630b1f 251a76a3bc860cb9f599bb2d6e183753120dcd26c211e3c20f16337b5
```

If you give the wrong input, the verification will fail; e.g.:

```
VERIFY FAIL 86777674e3fe09e0da911be4c7bce219794a8988955508d3e9433d8584630b1f 351a76a3bc860cb9f599bb2d6e183753120dcd26c211e3c20f16337
```

Block chaining and broadcast

Once a block has been created, it should be broadcast to the rest of the network. You can use the same protocol you used in CP1 for broadcasting transactions, or you may choose to update it to optimize it for block transmission.

Each node should try to create the longest chain of *valid* (see below) blocks. If it receives a block with a higher chain "height" (number of blocks since the beginning), it should immediately switch to solving a puzzle to extend the chain from that block. In case of a chain fork, a node may receive two blocks with the same length. Such ties may be resolved in an arbitrary manner.

Initially you will be mining the first block. For that one, the previous block hash entry should be set to 0.

Consensus Rules

To ensure that your cryptocurrency operates properly, you should make sure that your nodes enforce consensus rules when creating new blocks or receiving blocks from other nodes. The main rule you need to check is that a transaction does not create a negative balance. You will therefore need to keep track of the balances of all accounts and reject any transactions that "double-spend" money and cause the account balance go negative. Accounts are represented by a number. Initially all accounts have a zero balance, except for the special account `0`, which has an effectively infinite balance, i.e., a transfer from account 0 to any account always succeeds.

As an example, if the following transactions were received:

```
TRANSACTION [...] 0 1 100
TRANSACTION [...] 0 2 50
TRANSACTION [...] 2 3 100
TRANSACTION [...] 1 4 75
TRANSACTION [...] 1 5 75
```

the 3rd and 5th transaction should be rejected as there is insufficient balance to fund them, and the remaining balances should be 25 in account 1, 50 in account 2, and 75 in account 4. Note, however, that the order in which transactions are received changes *which* transaction is rejected, and the goal of the consensus protocol is to ensure that all nodes eventually agree on the order of all transactions.

The other consensus rules that need to be enforced are:

- Each block can hold at most 2000 transactions
- Each block must include a hash of the previous block
- Each block must include a puzzle solution that is correctly verified using the VERIFY service

Experiments and Evaluation

As in CP1, you will need to log events and analyze them to produce graphs in the report. In addition to the metrics in CP1, you should keep track of the following metrics:

- How long does each transaction take to appear in a block? Are there congestion delays?
- How long does a block propagate throughout the entire network?
- How often do chain splits (i.e., two blocks mined at the same height) occur? How long is the longest split you observed? (i.e., smallest distance to least common ancestor of two nodes)

The suggested experiments for your system are same as in CP2. However, if you have trouble getting your system to be stable at the high transaction rate and system size (100 nodes / 20 tps), try to find an intermediate rate / system size where your system still functions well.xk