

Block Chain Assignment 2

In this homework, we will use the concepts we've explored in class to extend the previous assignment with networking, and explore topics in byzantine agreement and consensus.

Submit your work to **Canvas** as a zip file similar to the one we provide, containing an additional **solutions.pdf** file for written answers. **Please do not discuss solution details with others.** For all assignments, you can make use of published materials, but must acknowledge all sources. Additionally, you must ensure that you understand the material you are submitting; you must be able to explain your solutions to the course instructor or TA if requested.

Disclaimer

In general, many of the technologies we are using this semester will be poorly (if at all) documented, will be constantly changing, and will often suffer from broken or dead code or packages. This is par for the course for cryptocurrency. **Please start the assignment early.** We do not guarantee responses from the TAs or instructors on errors in the assignment or such broken packages without at least 72 hours lead time.

This assignment also implements networking functionality, which is generally failure-prone. We will support such issues, and will only test your code across local connections with little to no latency.

Problem 0 - Setting Up

First, download the provided zip in Canvas.

The grading and testing code will be written in Python 3.6. **Full setup instructions and documentation for the provided code is available by opening docs/index.html in any web browser.**

Your solutions can be written in any programming language; we strongly recommend Python for its ability to natively interface with the provided infrastructure. If you would prefer to use a different programming language, you can use `subprocess.communicate` or other Python methods to call out to external binary code, including interpreters for other languages (an example is provided here: <https://stackoverflow.com/a/16770371>).

If you do choose a different programming language, please note that support from course instructors for language-related bugs may be limited, and it is your responsibility to ensure interoperability of input and output functionality throughout your code. If using a language other than Python, you are also required to provide a `SETUP` file with a list of Ubuntu or Debian packages required for your solution. We will not deduct points for problems setting up your solution, but may contact you if such issues arise. You are responsible for remedying any problems in a timely way.

We encourage you to read and analyze the provided code that is not part of the assignment: please ask questions if anything is unclear to you!

Problem 1 - p2p Warm Up [15]

Last assignment, we created a mini toy cryptocurrency that included several features of a real cryptocurrency, including partial block and transaction validation rules, fork handling, proof of work and proof of authority mining, a block explorer, and more.

Several features we learned about in recent lectures are missing in this homework. We will first add a peer-to-peer network to our blockchain cryptocurrency, allowing several nodes to coexist across a network and synchronize their blockchains.

We have made several changes to the previous codebase in support of this we suggest you explore:

- The `run_webapp.py` script of previous homeworks has been replaced by `run_node.py`, a script that runs a self-contained node with a block explorer and connection to the peer-to-peer network. The operation of this script is explained in the web-based documentation under “Running Nodes”. Configuration of available nodes on the peer-to-peer network is provided in `config.py`.
- The file `p2p/gossip.py` has been added, and the `p2p.interfaces` module has also been added to serialize and deserialize blocks and transactions as they are exchanged over the network. We recommend inspecting the interfaces, though their modification is not required for the assignment. All functions to handle incoming messages are provided (`gossip.handle_message`), dispatching them to the appropriate submodule.
- The `generate_example_pow_chain.py` script from before will now also gossip blocks to any available nodes on the peer-to-peer network. Your code will have nodes re-gossip messages to any available nodes on *their* p2p network. Each message should only be forwarded once; code for this is provided in `handle_message` as part of re-gossip.

We recommend re-exploring the provided web documentation for a complete overview of the new modules added to this version. The blockchain module interfaces are unchanged from the previous assignment.

We now ask you to complete the following (with solutions to 2, 3 placed in a **solutions.pdf** to be submitted to Canvas):

1. In `p2p/gossip.py`, complete the `gossip_message` function
Provided tests: tests.gossip
2. First clear your databases for nodes 1-6 and the parent/master node (see the instructions on the web documentation under “Running Nodes”). Run 3-6 nodes in separate command line windows, and re-run the `generate_example_pow_chain.py` file. This file has been modified to broadcast its generated blockchain to all nodes in the gossip network. Open the block explorers of each of your node at the end of that script’s execution; do they appear synchronized? Now, repeat the experiment, but this time stop one of the nodes around halfway through the execution, and restart it at the end. What do you observe, and does this suggest any additions required to our gossip protocol (if so, what changes, otherwise, why not)? You may consult the documentation of Bitcoin’s gossip protocol, available at https://en.bitcoin.it/wiki/Protocol_documentation.
3. Notice that our node’s list of peers is hard-coded in `config.py`. Does this suggest another missing component required to achieve a permissionless blockchain? Why not, or if so, what is the closest analogous message type in the Bitcoin p2p protocol documentation linked above?

Problem 2 - Adding Synchronous Rounds [20]

In our lectures on consensus, we often explored protocols in the *synchronous* model. The synchronous model assumes that the communication channel between participants (or network) has some bounded delay, Δ . Δ is known up-front and consistent among all participants in the protocol, and represents the maximum amount of time it may take a message to transmit across a network link.

A natural application of this model is to round-based protocols, in which nodes proceed in synchronized “rounds”, talking and reacting to each others’ messages while taking advantage of the assumption that no honest nodes will receive messages outside the intended round, if the messages were sent by an honest node. The protocols we discussed in class are generally round based. Consider the PKI model consensus protocol, we will now attempt to implement this protocol.

We hardcode our PKI in `inconfig.py` (PUBLIC KEYS and SECRET KEYS; since our nodes share config, we enforce in code that they only access the secret key of their own node ID; normally, the SECRET_KEYS vector would of course be limited to nodes a user controls).

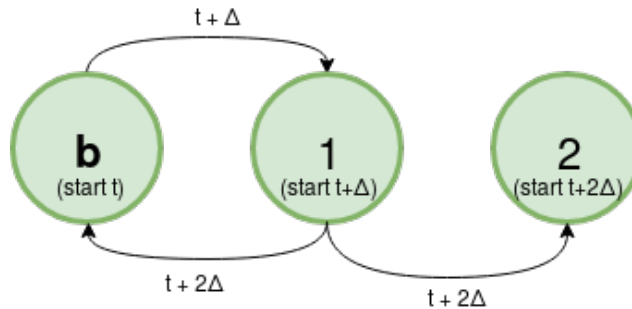
We must first implement the notion of synchronous rounds, assuming that our (actually asynchronous; the Internet provides no guarantees!) underlying network remains synchronous. Because we are doing all our testing locally on 127.0.0.1, this is a reasonable assumption, and messages are likely to be delivered quicker than our assumed Δ of two seconds.

We will also make an availability assumption, that is that all honest nodes will stay alive for the duration of the protocol; this is a relatively standard assumption, and can be removed through a “rejoin” protocol.

Lastly, we will make a clock synchrony assumption; this will allow us to start our series of rounds once and assume little divergence as a small number of rounds is completed, removing the need to re-synchronize each phase of each round, a slow process. This is also reasonable in our environment, as our nodes share the same local clock (you can run

these across computers too, and the small number of rounds we require likely means the assumption will hold).

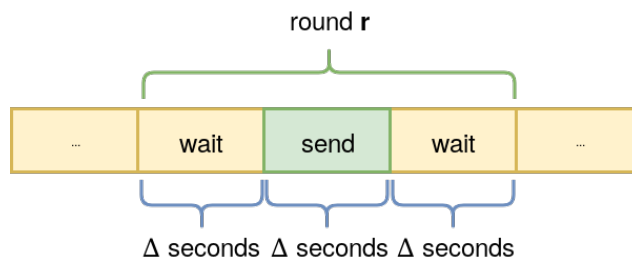
To start a synchronous round, any node can send a message **synchrony-start** to any other node. Honest nodes will gossip this message to the entire network. If any honest node gets a message at time t , all honest nodes will have the message by $t + \Delta$ (definition of synchrony). Therefore, honest nodes' internal clocks tracking their position in a round will always all differ by at most Δ . This is captured by the following diagram:



We can fairly simply deduct that any honest nodes' clocks will be at most Δ apart ($\pm \Delta$) relative to each other.

Consider the scenario in the diagram, where a byzantine node b broadcasts a start message to only one of two honest nodes. This node then gossips the message to all other honest nodes. The last honest node to receive it differs from the Byzantine node's start time by $2 * \Delta$, but from the honest node's start time by only Δ . Nodes that receive start messages when their synchrony rounds are already in progress simply ignore these messages, meaning 2's later gossip of the message has no effect on any nodes.

Intuitively, from this synchronization protocol, we can simulate non-overlapping rounds using the following technique:



Our final protocol will send at the beginning of the send period. This is similar to the clock synchronization technique explored in class. Assuming perfectly synchronized clocks, all other nodes will see the message in the sending period by the synchrony assumption. Because honest clocks are not perfectly synchronized and can be $\pm \Delta$ apart (by the above intuition), we add a buffer on each side of the sending period where nodes can receive but not send messages. Nodes start round 0 immediately on receipt of a start message.

We will now ask you to complete the following:

1. In `p2p/synchrony.py`, complete `receive_start_message()`. This function should initiate the synchrony round tracker by setting the start time and running the logger.
Provided tests: `tests.synchrony_start`
2. In `p2p/synchrony.py`, complete `get_curr_round()`. Based on the start time and current time, this function should compute the return the current round number as an integer and a function of all the provided global constants.
Provided tests: `tests.synchrony_rounds`
3. In `p2p/synchrony.py`, complete `should_send()` as per the timing rules above, returning True if an honest node would be broadcasting in this part of the round.
Provided tests: `tests.synchrony_sends`

To test the complete protocol:

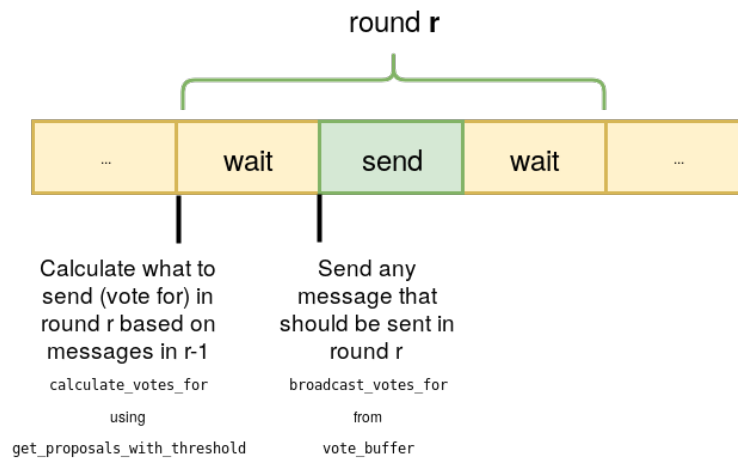
1. Run 2-6 nodes; node 1 should be among these nodes, unless you want to test what happens with a failed sender.
2. Once your nodes are running, run `start_synchrony.py` in the root directory.

Each node should produce logging output to the console tracking its current round / round state. These outputs should all be *synchronized* by our synchrony and clock synchrony assumptions above.

Problem 3 - Realizing BFT [25]

Having achieved synchronous rounds, we can now build a Byzantine fault-tolerant agreement protocol (“BA” protocol) for our nodes to come to consensus on a value. Notice that all such protocols we discuss have a sender and a message; for testing, we will have our nodes come to agreement on a random number proposed by the sender as the message, and choose Node 1 as an arbitrary sender.

Consider implementing the protocol in Figure 4.1 of the lecture notes. We will implement the protocol in two parts, as follows:



First, proposals that have reached the required signature threshold as described in the protocol (and are signed by the sender) are gathered, and “votes/signatures” for these proposals (consisting of a proposal, signature pair) are placed in the appropriate data structures. Then, a vote buffer is updated with any new votes that a node has not voted for, but should vote for in this round. Finally, at the start of the sending phase of the synchronous protocol we’ve built above, honest nodes broadcast all votes in their buffer to all other honest nodes, resetting their buffer.

(Note: Figure 4.1 describes coming to agreement on a bit, but coming to agreement on a message works identically).

We now ask you to complete the following (with solutions to 2, 3 placed in a **solutions.pdf** to be submitted to Canvas):

1. In `byzantine.agreement/simple_ba.py`, complete `get_proposals_with_threshold(self, round)`. This function returns all proposals a node should vote for in round r given current data structures containing messages from round $r - 1$.
Provided tests: `tests.ba_proposals`
2. In `byzantine.agreement/simple_ba.py`, complete `calculate_votes_for(self, round)`. This function updates internal data structures with the results of the above (add to S_i in the pseudocode, and add to voting queue and other data structures in implementation).
Provided tests: `tests.ba_votes`
3. In `byzantine.agreement/simple_ba.py`, complete `get_output(self)`. This function decides what to output if the protocol is completed.
Provided tests: `tests.ba_output`

Congratulations, you now have a well-isolated, terminating/synchronous chunk of code that allows a leader to have all their peers come to consensus on a proposed value, satisfying the properties we’ve studied of Byzantine agreement!

To test the complete protocol:

1. Run 2-6 nodes; node 1 should be among these nodes, unless you want to test what happens with a failed sender.
2. Once your nodes are running, run `start_ba.py` in the root directory. You can add Byzantine nodes to the protocol by modifying the `BYZANTINE` list in `config.py`; nodes with that ID will load code from `byzantine.agreement/byzantine_ba.py`. The current Byzantine behavior sends random data instead of signatures; we encourage you to experiment with arbitrary modifications to the superclass BA protocol’s behavior to test your code’s handling of Byzantine faults.

*Note that even this simple, non-consensus agreement protocol, which isn’t enough to build a blockchain since it says nothing about how to choose a sender and how blocks correspond to rounds, is **more robust** than the NEO network’s consensus protocol, which was recently revealed could not tolerate the crash (nevermind full Byzantine) failure of even a single one of the nodes running such a consensus protocol to secure their blockchain: <https://cryptovest.com/news/neos-consensus-glitch-what-really-happens-if-nodes-crash/>. Ready for your own 4 billion dollar currency?*