Computer and Network Security I

Final Project: Building a Cryptocurrency

Benjamin Mourad
Mary Provencher (Team Leader)

https://github.com/mprovenc/comp5610-cryptoc

Abstract

The goal of this project was to build an application containing all the components of a simple cryptocurrency. The application is modeled after digital currencies such as Bitcoin, Ethereum, Litecoin, and others, all of which employ a similar, decentralized blockchain structure. Following this pattern, the cryptocurrency that we built in this project functions in the context of a peer-to-peer network and makes use of blockchain technology to track and verify transactions, utilizing public key cryptography to secure these transactions. Users are allowed to join this peer-to-peer network, make secure cryptocurrency transactions between themselves and other nodes within the network, and accurately track the balance of any node within the network, offering a comprehensive, lightweight cryptocurrency solution.

Project Overview

The project can be broken into three key network and security components, all of which were touched upon at some point during our class lectures this past semester:

1. A peer-to-peer network, which served as the foundational infrastructure for the project environment
2. Secure communications among all network nodes using PKC, specifically Elliptic Key Cryptography
3. A blockchain mechanism used for consensus and validation of transactions among nodes

Using these core components, we were able to successfully build a system that allows network nodes to manage digital assets and distribute them as desired to other nodes within the network in a secure manner.

Technologies Used

The code base is written entirely in Python and tested in a Linux environment. Additionally, the following Python libraries were used to assist with thread management and some of the cryptographic operations required for the implementation of PKC within our project:

- threading
- hashlib
- pynacl
- base64

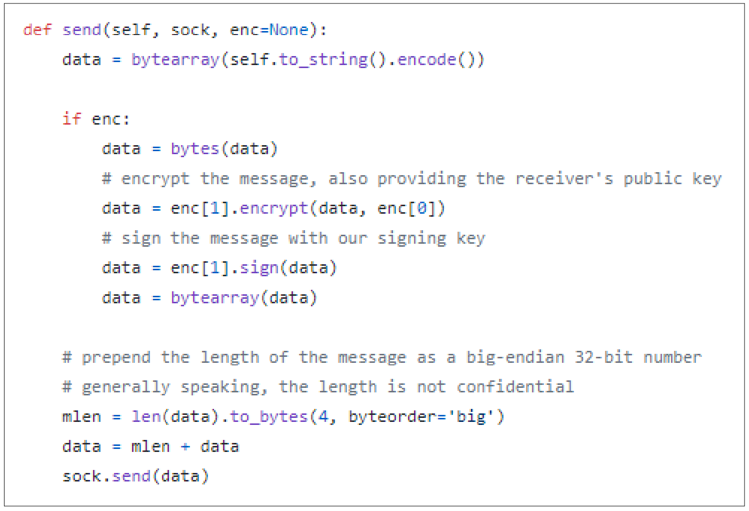
Implementation

Peer-to-Peer Network

The peer-to-peer network is implemented as a series of interconnected nodes with a "tracker" node which keeps a running list of nodes within the network. When a new node requests to join the network, it interfaces with the tracker node, which provides that new node with a unique ID and a list of peers within the network. The tracker node also supplies the new node with a current copy of its blockchain. Subsequently, this new node will only interface with other peers (unless it is providing the tracker with a new block to be added to the network blockchain), thus preserving the concept of a peer-to-peer network.

Public Key Cryptography

All network communications are secured using elliptic curve cryptography as implemented by the PyNacl library (PyNacl is a wrapper around libsodium, a widely used cryptographic library written in C). The tracker node generates for itself a public/private key pair. New nodes also generate a public/private key pair for themselves and share their public key with the tracker node when they request to join the network. The tracker node then shares its public key with the new node and signs all further communications with its private key. The tracker also shares public keys for all of the other nodes within the network, allowing for this new node to securely communicate with any of its peers. All communications are encrypted from this point on.



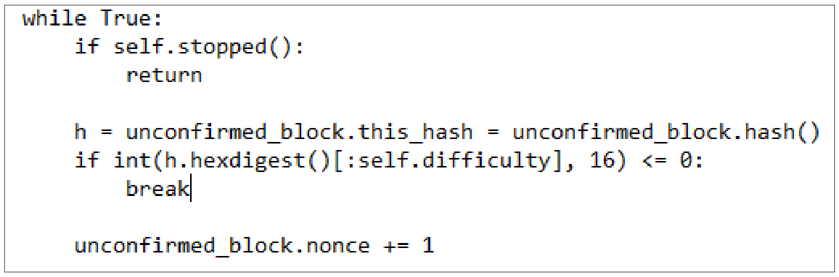
```python
def send(self, sock, enc=None):
    data = bytearray(self.to_string().encode())

    if enc:
        data = bytes(data)
        # encrypt the message, also providing the receiver's public key
        data = enc[1].encrypt(data, enc[0])
        # sign the message with our signing key
        data = enc[1].sign(data)
        data = bytearray(data)

    # prepend the length of the message as a big-endian 32-bit number
    # generally speaking, the length is not confidential
    mlen = len(data).to_bytes(4, byteorder='big')
    data = mlen + data
    sock.send(data)
```

Figure 1: All communications within the network are secured using PKC as illustrated in this send routine.

Blockchain

Each node within the network has its own blockchain. The blockchain consists of an ordered series of blocks, each of which contains one or more transactions. In addition to a list of transactions, each block contains a timestamp and a hash of the previous block in the chain (thus immutably linking the blocks in the chain together). When a new transaction occurs in the network, it is added to a list of unconfirmed transactions on each nodes blockchain. Each node then enters a mining phase, in which that node attempts to solve a difficult cryptographic problem. Once one of the nodes solves the problem, that node broadcasts a suggested block to the rest of the nodes in the network, which validate the transactions in that block and add the block to their blockchain, thus confirming the previously unconfirmed transactions.

In the mining phase, the proof-of-work algorithm used is modeled after Hashcash, the proof-of-work system used by cryptocurrencies such as Bitcoin. In this algorithm, the goal is to take a base string and find a variation of it such that the SHA-256 hash of that string is less than or equal to a fixed number[1]. In the case of our implementation of this proof-or-work scheme, we took the hash of the proposed block to be the base string. A nonce is added to the end of that proposed block. At each iteration, this nonce is incremented by 1 and then the hash of the base string plus the nonce is compared to a fixed number. This routine of trying to "crack the hash" can be either very quick or very slow depending on the size of the fixed number, so this part required some tuning. We settled on testing for 5 leading zeros.



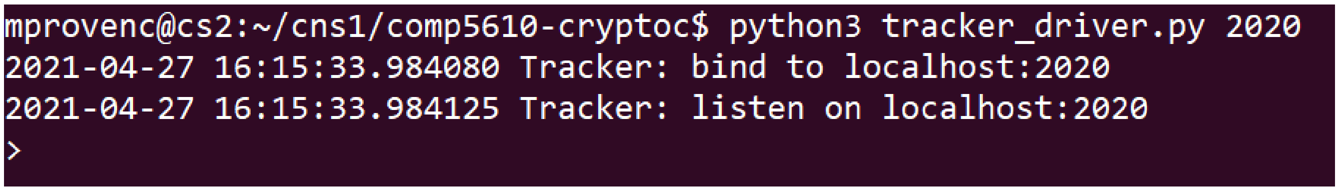
```
while True:
    if self.stopped():
        return

    h = unconfirmed_block.this_hash = unconfirmed_block.hash()
    if int(h.hexdigest()[:self.difficulty], 16) <= 0:
        break

    unconfirmed_block.nonce += 1
```

Figure 2: The proof-of-work algorithm used. The variable self.difficulty denotes the number of leading zeros to test for, thus controlling the difficulty of the cryptographic problem.

Before a block is added to the chain, the transactions are all validated. To validate transactions, a node loops through their blockchain to ensure that the sending user has enough balance to make the transaction. By default, all users begin with a balance of 10 coins.

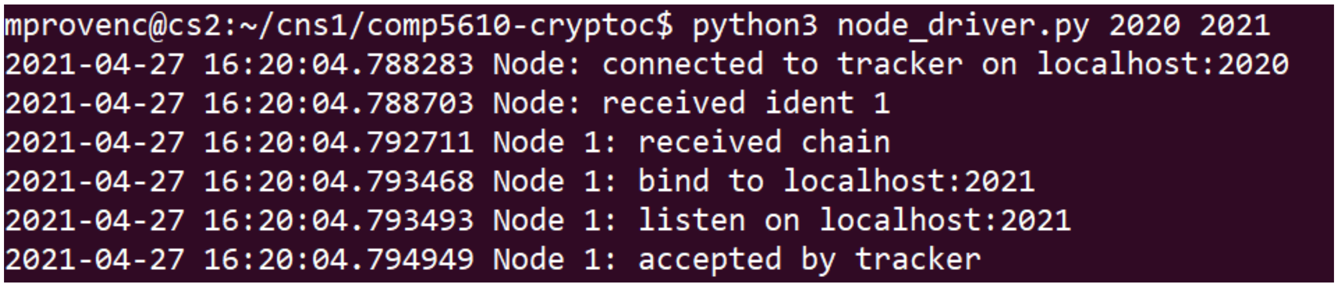
An Example Transaction

Setting up the peer-to-peer network begins with the creation of the tracker node. In the sample below, a driver program is used to initialize the tracker node and bind it to localhost:2020.



```
mprovenc@cs2:~/cns1/comp5610-cryptoc$ python3 tracker_driver.py 2020
2021-04-27 16:15:33.984080 Tracker: bind to localhost:2020
2021-04-27 16:15:33.984125 Tracker: listen on localhost:2020
>
```

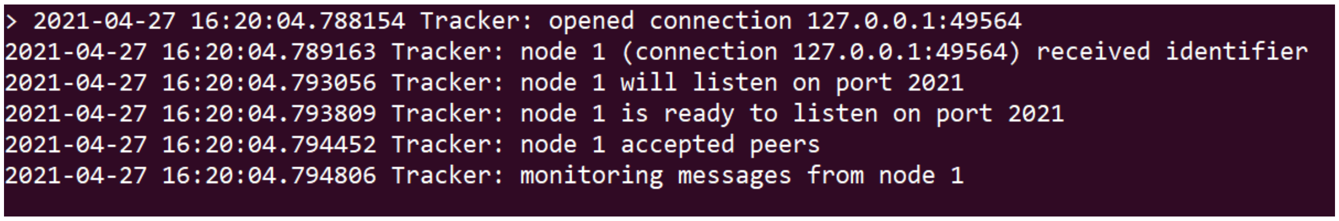
The first node joins the network. The node driver program in the sample below is used to initialize a new node, binding it to localhost:2021 and connecting it to the tracker node running on localhost:2020. Note that this node receives a new ID as well as the tracker's blockchain. This node will be referred to from this point on as Node 1.



```
mprovenc@cs2:~/cns1/comp5610-cryptoc$ python3 node_driver.py 2020 2021
2021-04-27 16:20:04.788283 Node: connected to tracker on localhost:2020
2021-04-27 16:20:04.788703 Node: received ident 1
2021-04-27 16:20:04.792711 Node 1: received chain
2021-04-27 16:20:04.793468 Node 1: bind to localhost:2021
2021-04-27 16:20:04.793493 Node 1: listen on localhost:2021
2021-04-27 16:20:04.794949 Node 1: accepted by tracker
```

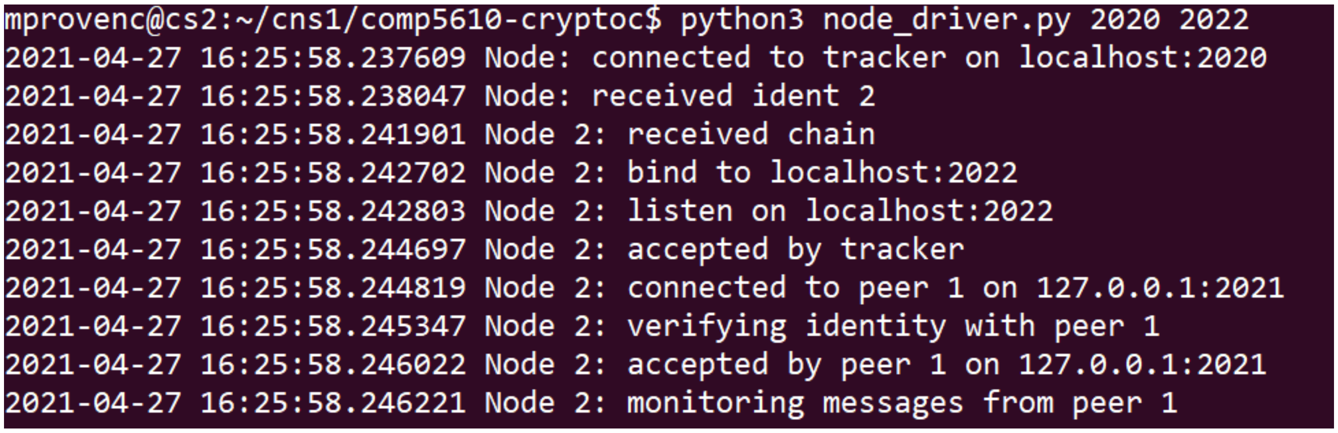
1 A. Van Wirdum. "The Genesis Files: Hashcash or How Adam Back Designed Bitcoin's Motor Block." bitcoinmagazine.com. https://bitcoinmagazine.com/technical/genesis-files-hashcash-or-how-adam-back-designed-bitcoins-motor-block. (accessed Apr. 27, 2021).

The corresponding messages on the tracker's console show that it has accepted Node 1 into the network:



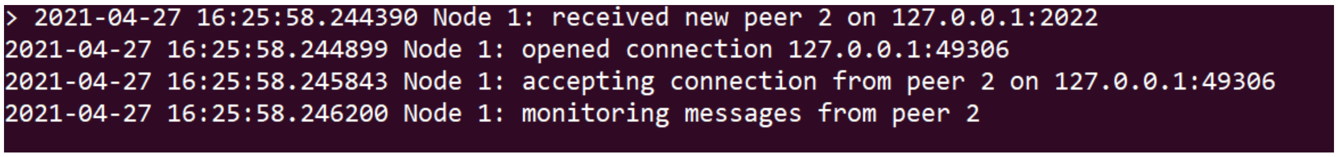
```
> 2021-04-27 16:20:04.788154 Tracker: opened connection 127.0.0.1:49564
2021-04-27 16:20:04.789163 Tracker: node 1 (connection 127.0.0.1:49564) received identifier
2021-04-27 16:20:04.793056 Tracker: node 1 will listen on port 2021
2021-04-27 16:20:04.793809 Tracker: node 1 is ready to listen on port 2021
2021-04-27 16:20:04.794452 Tracker: node 1 accepted peers
2021-04-27 16:20:04.794806 Tracker: monitoring messages from node 1
```

When a second node binds to the tracker node, it receives an ID of 2 and starts to monitor messages from Node 1:



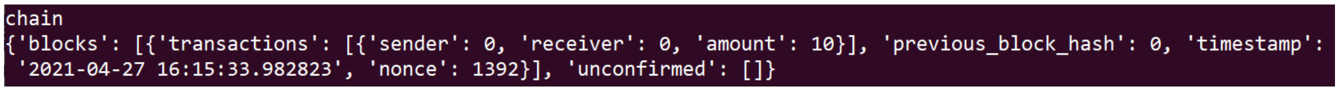
```
mprovenc@cs2:~/cns1/comp5610-cryptoc$ python3 node_driver.py 2020 2022
2021-04-27 16:25:58.237609 Node: connected to tracker on localhost:2020
2021-04-27 16:25:58.238047 Node: received ident 2
2021-04-27 16:25:58.241901 Node 2: received chain
2021-04-27 16:25:58.242702 Node 2: bind to localhost:2022
2021-04-27 16:25:58.242803 Node 2: listen on localhost:2022
2021-04-27 16:25:58.244697 Node 2: accepted by tracker
2021-04-27 16:25:58.244819 Node 2: connected to peer 1 on 127.0.0.1:2021
2021-04-27 16:25:58.245347 Node 2: verifying identity with peer 1
2021-04-27 16:25:58.246022 Node 2: accepted by peer 1 on 127.0.0.1:2021
2021-04-27 16:25:58.246221 Node 2: monitoring messages from peer 1
```

We can also see that Node 1 has accepted the new node as it's peer:



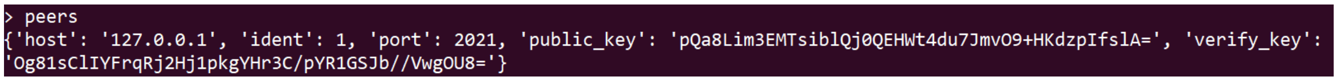
```
> 2021-04-27 16:25:58.244390 Node 1: received new peer 2 on 127.0.0.1:2022
2021-04-27 16:25:58.244899 Node 1: opened connection 127.0.0.1:49306
2021-04-27 16:25:58.245843 Node 1: accepting connection from peer 2 on 127.0.0.1:49306
2021-04-27 16:25:58.246200 Node 1: monitoring messages from peer 2
```

In our code, we added read-evaluate-print loops to allow real-time interaction with each of the nodes in the network. By running the command "chain" on Node 2, we can view Node 2's blockchain. The sample below shows see that this node currently has an empty blockchain, except for the initial balance of 10 coins granted to it by the tracker node when it was initialized. The blockchain is the same on Node 1.



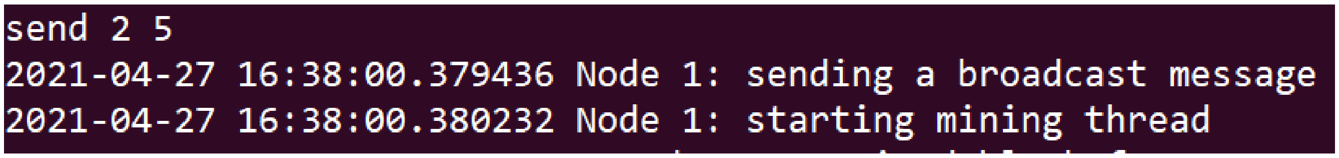
```
chain
{'blocks': [{'transactions': [{'sender': 0, 'receiver': 0, 'amount': 10}], 'previous_block_hash': 0, 'timestamp':
 '2021-04-27 16:15:33.982823', 'nonce': 1392}], 'unconfirmed': []}
```

By running the REPL command "peers", we can view a node's peers. Running this command on Node 2 shows that it has Node 1 as a peer.



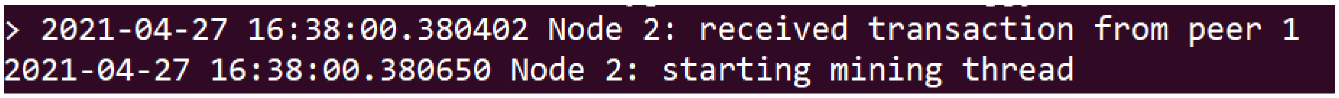
```
> peers
{'host': '127.0.0.1', 'ident': 1, 'port': 2021, 'public_key': 'pQa8Lim3EMTsiblQj0QEHWt4du7JmvO9+HKdzpIfslA=', 'verify_key':
'Og81sClIYFrqRj2Hj1pkgYHr3C/pYR1GSJb//VwgOU8='}
```

The REPL command “send 2 5” can be run from the console on Node 1 to denote that Node 1 wants to send 5 coins to Node 2. Node 1’s console shows that it is broadcasting this transaction to all the nodes in the network. Node 1 also starts trying to mine a new block right away:



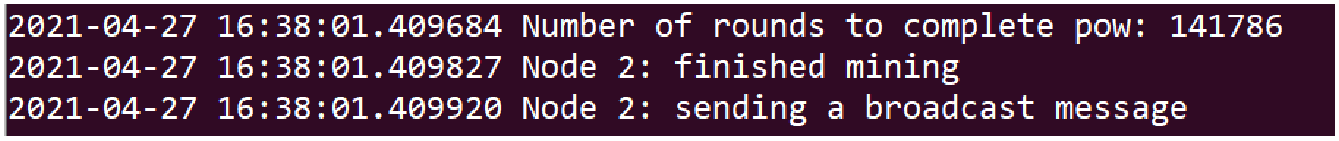
```
send 2 5
2021-04-27 16:38:00.379436 Node 1: sending a broadcast message
2021-04-27 16:38:00.380232 Node 1: starting mining thread
```

Node 2’s console shows that it has received this transaction and it also starts trying to mine a new block right away:



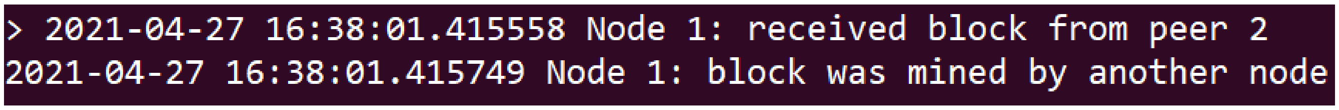
```
> 2021-04-27 16:38:00.380402 Node 2: received transaction from peer 1
2021-04-27 16:38:00.380650 Node 2: starting mining thread
```

Both Node 1 and Node 2 are mining, and in this sample, it seems that Node 2 finishes mining before Node 1. Node 2’s console shows that it completed it’s proof-of-work routine in 141,786 iterations, and it broadcasts the new block to all other nodes in the network.



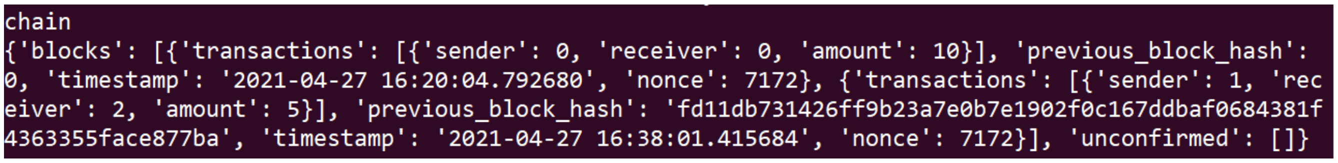
```
2021-04-27 16:38:01.409684 Number of rounds to complete pow: 141786
2021-04-27 16:38:01.409827 Node 2: finished mining
2021-04-27 16:38:01.409920 Node 2: sending a broadcast message
```

Node 1 receives the proposed block and ends its mining routine early.



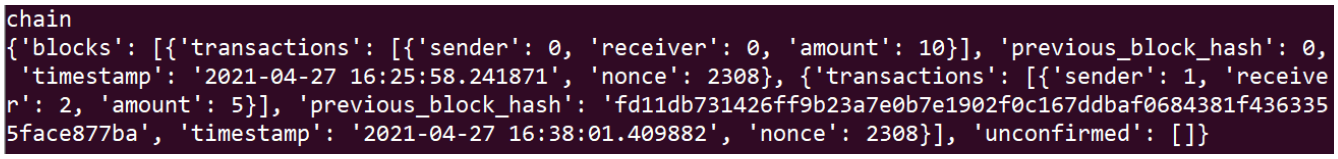
```
> 2021-04-27 16:38:01.415558 Node 1: received block from peer 2
2021-04-27 16:38:01.415749 Node 1: block was mined by another node
```

Running the “chain” command again on Node 1 shows that it has added the new block to its chain successfully.



```
chain
{'blocks': [{'transactions': [{'sender': 0, 'receiver': 0, 'amount': 10}], 'previous_block_hash':
0, 'timestamp': '2021-04-27 16:20:04.792680', 'nonce': 7172}, {'transactions': [{'sender': 1, 'rec
eiver': 2, 'amount': 5}], 'previous_block_hash': 'fd11db731426ff9b23a7e0b7e1902f0c167ddbaf0684381f
4363355face877ba', 'timestamp': '2021-04-27 16:38:01.415684', 'nonce': 7172}], 'unconfirmed': []}
```

Node 2 has also added the new block to its chain:



```
chain
{'blocks': [{'transactions': [{'sender': 0, 'receiver': 0, 'amount': 10}], 'previous_block_hash': 0,
 'timestamp': '2021-04-27 16:25:58.241871', 'nonce': 2308}, {'transactions': [{'sender': 1, 'receive
r': 2, 'amount': 5}], 'previous_block_hash': 'fd11db731426ff9b23a7e0b7e1902f0c167ddbaf0684381f436335
5face877ba', 'timestamp': '2021-04-27 16:38:01.409882', 'nonce': 2308}], 'unconfirmed': []}
```

This concludes the example showing a successful transaction using our cryptocurrency application.

Future Work

Some areas of potential future work include:

- **Ensuring fault tolerance.** Our current system does not take into account the possibility of network latency issues that could impact the transaction and mining components of our application. For example, a race condition could occur if there is a delay in transmitting a block or receiving a transaction. Such network latency issues are common, putting this issue at the forefront of our future plans.
- **Larger-scale testing.** All of our testing thus far has involved only a few nodes at a time. It would be beneficial to test with many nodes to see how well our application scales and what types of adjustments we may need to make to improve the scalability.
- **Variable starting balance.** With the current setup, all nodes start with the same initial balance. We would like to add more flexibility in the future to provide the possibility of different starting balances.
- **More flexible proof-of-work routine.** The difficulty of our proof-of-work mechanism is currently a fixed difficulty. We would like to be able to increase the difficulty based on some function of how many nodes are simultaneously mining – the more nodes are mining, the more difficult the proof-of-work mechanism ought to be.