EN.601.441/641: Assignment 1b

September 2024

This is an pair assignment. Submit to Gradescope by September 25th at 11:59pm.

Specification. For this program you will write a simulation of a "Minimum Viable Blockchain" (MVB), a simplified version of the technology underlying Bitcoin. Your implementation will include nodes that validate transactions, perform proofs of work, and communicate in order to process transactions. This will emulate the mining and transaction verification process of Bitcoin.

Your MVB will implement the following:

- Authentic transactions that are resistant to theft
- Open competition amongst nodes to validate transactions
- Detection of double spending
- Use of proof-of-work to raise the cost of running attacks against the network
- Detection of and reaction to forks in the chain

You may work with a partner on this program.

Formats. The starter code that we provide gives a unique serialization for inputs, outputs, transactions, and blocks. Please refer to the starter code for how this is represented.

For pedagogical purposes, you can envision a transaction like so:

```
[
  {
    "number": <SHA256 hash of input, output, and signature fields>,
    "input": [
      {
        "number": <transaction number>,
        "output": {
          "value": <value>,
          "pubkey": <sender public key>
        }
      },
      ...
    ],
    "output": [{"value": <value>, "pubkey": <receiver public key>}, ...],
    "sig": <signature of input and output fields using sender private key>
  },
  ...
]
```

Note: ... implies that there could be an arbitrary number of similar elements in the list. The exception is the first transaction in the file, which will have an empty input list.

Likewise, you can envision a block as follows:

```
{
  "tx": <a single JSON transaction, using the format above>,
  "nonce": <the nonce value in hex, used for proof-of-work>,
  "prev": <SHA256 hash of the previous block in hex>,
  "pow": <the proof-of-work, a hash of the tx, prev, and nonce fields, in hex>
}
```

Hashes are 256 bits and should be written as 64-character hexadecimal values. You can compute these in Python as follows:

```
# python3
>>> from hashlib import sha256 as H
>>> computed_hash = H(b'hello') # note that the b makes the string a bytes literal
>>> computed_hash.hexdigest()
'2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824'
```

Similarly, nonces, signatures, and keys should be formatted as hex. We will use the EdDSA algorithm as implemented in pynacl. See `https://pynacl.readthedocs.io/en/latest/signing/` for an example of signing and verification, as well as the procedure to encode values into hex.

Genesis Block. The first transaction in the transaction file is special:

- it does not need to have a valid signature
- it will have an empty input list

The autograder will provide you with a genesis block to use.
The first block must contain the first transaction. This block, known as the "Genesis" block, has the following properties:

- The prev hash field may contain arbitrary hex data (of the appropriate length)
- The nonce is an arbitrary value (of the appropriate length)
- The pow hash may contain arbitrary hex data (of the appropriate length)

Node behavior. Your node will receive the global genesis block from the autograder. The autograder will verify your node can create valid transactions. It will also verify that your node can build (and mine) blocks from transactions. The autograder will also verify that your node accepts valid blocks from "the network." You do not need to keep track of any pending transactions.

In order to process a transaction, a node will have to perform the following steps:

1. Ensure the transaction is validly structured
   (a) `number` hash is correct
   (b) each `input` is correct
      i. each number in the input exists as a transaction already on the blockchain
      ii. each output in the input actually exists in the named transaction
      iii. each output in the input has the same public key, and that key can verify the signature on this transaction
      iv. that public key is the most recent recipient of that output (i.e. not a double-spend)
   (c) the sum of the input and output values are equal
2. Construct a block containing this transaction, setting `prev` to the hash of the most recent block

3. Create a valid proof-of-work for this block by setting `nonce` and `pow`

Additionally, each node must react to the broadcast of blocks from "other nodes":

1. Verify the proof-of-work
2. Verify the `prev` hash
3. Validate the transaction in the block
4. If all three checks pass, append this block to the node's instance of the blockchain, and continue work
5. Handle potential forks

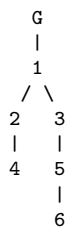
Proof-of-Work. In order to be a valid block, the pow value must be less than or equal to the hexadecimal value below:

```
0x07FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
```

Forking. In the operation of your nodes, forks may occur in your blockchain:

- The longest chain is the only valid blockchain
- Thus, at any point, any blocks can be invalidated, no block is ever "final"
- Your node should mine blocks based off of the longest chain it is aware of
- Spent outputs in an invalidated block become unspent transaction outputs

Forking example (numbers represent blocks in order they are broadcast, and lines represent prev links):



After 2 and 3 are broadcast (likely at roughly the same time), some nodes might be working off of 2 and some off of 3. Once 4 is broadcast, the transaction in 3 is no longer verified. It is possible (but unlikely) that blocks 5 and 6 might build off of 3 rather than 4, at which point the transactions in both 2 and 4 would be unverified – once that chain no longer represents the longest path back the Genesis block G.

Output. Follow the comments in the starter code.

Notes. Running your program should require no external dependencies or packages, but installation of Python 3 packages via `pip`/`pip3` is acceptable. Include a `requirements.txt` (generated via `pip list > requirements.txt`).

Resources. You may wish to consult the following resources:

- `https://www.igvita.com/2014/05/05/minimum-viable-block-chain/`
- `https://docs.python.org/3.5/library/venv.html`
- `https://docs.python.org/3.7/library/threading.html`
- `https://en.wikibooks.org/wiki/Python_Programming/Threading`
- `https://docs.python.org/3.7/library/json.html`
- `https://docs.python.org/3/library/hashlib.html`
- `https://pynacl.readthedocs.io/en/latest/`

This project was adapted from one developed by Professor Zachary Peterson.

Submission. Provide all code, `requirements.txt` to Gradescope. The TAs and CAs may ask for you to provide your code in a specific way to make the autograder happy.