EN.601.441/641: Assignment 1b

September 2024

This is an pair assignment. Submit to Gradescope by September 18th 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
- Rather than communicating over a network, your simulated nodes will run in threads. Transactions, either valid or intentionally invalid, will be defined in a file which is read by the simulation and provided to the nodes. Your threading should be non-cooperative, which is to say that there should be no synchronization primitives such as locks. This simulates nodes running independent of one another.

You may work with a partner on this program.

Formats. You must format messages *exactly* as given above for the autograder to work. We are using a JSON format for this assignment, but there are multiple JSON representations that denote the same object (due to whitespace, field ordering, etc.). Place newlines after the curly braces and commas, use double quotes, indent with two spaces, and order the fields exactly as given below.

A single transaction will be represented using text/plain JSON data structure containing a single list as follows:

```
[
  {
    "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.

A block will be a text/plain JSON structure constructed 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 should be written as 64-character hexadecimal values with a double quotation mark on each side. 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 with single quotes. We will use the EdDSA algorithm as implemented in pynacl. See `https://pynacl.readthedocs.io/en/stable/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 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. Each node will receive the global Genesis block, and then begin processing transactions from the global unverified transaction pool. Each node should have its own representation of the blockchain at any given time. In order to process a transaction, a node will have to perform the following steps:

1. Ensure the transaction is not already on the blockchain (included in an existing valid block)
2. 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

3. Construct a block containing this transaction, setting `prev` to the hash of the most recent block
4. Create a valid proof-of-work for this block by setting `nonce` and `pow`

Additionally, each node must periodically check for and 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

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

```
0x07FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
```

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"
- Each node should work off of the longest chain it is aware of
- The contained transaction in an invalidated block become unverified and must be re-added to the global unverified pool by the nodes, if not already there

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

```
  G
  |
  1
 / \
2   3
|   |
4   5
    |
    6
```

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.

Driver Program. Finally, you are required to write a driver program which, after starting at least 8 nodes, will begin reading the transaction file and populating the global unverified transaction pool. The driver should also create the Genesis block and make it available to the nodes. Nodes should continue processing until all valid transactions in the transaction file are verified. You may implement this by having the driver terminate the node threads. The driver should sleep for a random (changing) time up to 1 second between placing each transaction into the global unverified pool.

Upon termination, each node should write its blockchain to a file. If all has gone well, these files will all be identical, and contain every valid transaction in the transaction file. In the very rare case that two equal-length chains exist when all valid transactions are verified, re-run the entire program.

Output. The blockchain output of each node must be JSON, a list of the described `block` structures.

Notes. It is strongly preferred and recommended that you use Python 3 for this project. That said, speak with the TA if you would like to make a case for using another language, but note that likely no debugging/library questions will be considered if you do.

You may wish to implement broadcast as each thread having a thread-safe queue which any node writes to but only it reads from. To broadcast, a node would write to each other node's queue.

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/stable/`

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

Submission. Provide all code, `requirements.txt`, and instructions for running your driver program.