Principles of Distributed Ledgers: Tutorial 1

For the purpose of this tutorial, we use a very simplified representation of a blockchain, and rely on encoding mechanisms that would not be used in production systems (e.g. hashing hex strings instead of bytes). These are to simplify the implementation. Switching to a production-ready encoding should be relatively easy given a decoupled enough implementation of this system.

Task

The aim of this tutorial is to write the code for a simple prototype blockchain client. To avoid having to deal with networking, everything will be read from and written to JSON files on the filesystem. The client should support the following functionalities:

1. Load the most recent block header hash from the `blockchain.json.gz` file.
2. Produce a new block (see **Block specs** from a set of pending transactions stored in a JSON file (`mempool.json.gz`)) using a proof-of-work algorithm. For the purposes of this tutorial, we assume that a block has at most 100 transactions. The steps for this should be
   (a) Find the set of executable transactions (see **Transaction specs**) that are the most profitable (have the highest fees). To find the transactions to include, you should filter the mempool for transactions executable by the time the next block will be mined (compare the `lock_time` in the transaction and the `timestamp` of the block you will mine) and then select up to 100 of the most profitable ones (highest `transaction_fee`). Each block should contain 100 transactions if possible. For this step, you do not need to verify that the transactions are valid, this is part of the bonus section.
   (b) Construct the Merkle tree for the set of transactions that should be included in a block to get the Merkle root (see **Merkle root**).
   (c) Create a new block using the transactions and the constructed block header. See the steps for creating the **hash** in the **Block specs**.
   (d) Mine the block, i.e. find a hash of the new block that respects the current chain difficulty by altering the nonce in the block header (see **Proof-of-work algorithm**). We recommend starting from 0 and incrementing the nonce. The final hash should be hex-encoded, prefix by `0x` and go in the "`hash`" field of the new block header.
3. Write the current state of the blockchain to a JSON file.
4. Produce an inclusion proof, proving that a transaction is included in a particular block.
5. Verify the validity of an inclusion proof.

All hex strings should always be pre-fixed with `0x`. In the JSON file with the current state of the blockchain, this prefix is already included. Note that all the JSON files provided are gzipped. The current state of the blockchain is stored in `blockchain.json.gz` and the mempool in `mempool.json.gz`.

Block specs

For the purpose of this tutorial, the genesis (first) block is produced at timestamp `1697412600` and the next block is then always produced 10 seconds later.

In this tutorial, a block contains a list of `transactions` (see **Transaction specs** for the details of these) and a `block_header` that contains the following fields:

`height` The height (number) of the current block, starting from 0 and incrementing by 1 for each new block produced

`previous_block_header_hash` The hash of the previous block header. The only exception is for the genesis (first) block, where it will be 0

`timestamp` The UNIX timestamp (in seconds) at which the block has been produced (as mentioned above, this should always be 10 seconds later than the previous block).

`transactions_merkle_root` The Merkle root of all the transactions included in the block

`transactions_count` The number of transactions included in a single block. The maximum number of transactions in a block is 100 `miner` The address of the miner who mined the block. For this tutorial, you can simply use any address as the miner. All the keys in `keys.json.gz` are valid addresses.

`nonce` A nonce that is modified to satisfy the proof-of-work constraint. To generate a nonce, you should start with the nonce being 0 and increment by 1 during each try when performing the proof-of-work (see **Proof-of-work algorithm**).

`difficulty` The difficulty of the current block. The difficulty is expressed as the number of leading 0s that the block header hash must have. The difficulty starts at 1 on the genesis block and will be incremented by 1 every 50 blocks (e.g. the first time when the block height is 50). Once the difficulty reaches 6 it should remain constant indefinitely.

`hash` The hash of the current block header. This hash is created by performing the following steps:

1. Sort all the above fields in alphabetical order by their key.
2. Produce a comma-separated string containing all the values, without any space. Numbers (`height`, `timestamp`, `nonce`, `transaction_count`, `difficulty`) should be encoded as decimal value without any leading 0s. Hashes (`previous_block_header_hash`, `transactions_merkle_root`) and addresses (`miner`) should be hex-encoded and prepended by `0x`.
3. Hash the string produced in step 2 using the SHA-256 hash function.

Below is an example of how to execute step number 3 of the block header hash creation in Python, given the following block header:

```
{
  "difficulty": 5,
  "height": 203,
  "transactions_merkle_root":
    "0xddba0c2d7d38a9bc8ba357d1fcb4a4be339ab5fddf8cdcc4419970e4746d1f6e",
  "miner": "0xdc45038aee5144bbfa641912eaf32ebf2bad2bd7",
  "previous_block_header_hash":
    "0xb2448304889df2935277464e90a73e53b9d2c5820c48de4a40d4fa5b844c7b57",
  "timestamp": 1697412660,
  "transactions_count": 97,
  "nonce": 0
}
```

```python
from hashlib import sha256

string_to_hash = (
    "5,203,0xdc45038aee5144bbfa641912eaf32ebf2bad2bd7,0,"
    "0xb2448304889df2935277464e90a73e53b9d2c5820c48de4a40d4fa5b844c7b57,"
    "1697412660,97,"
    "0xddba0c2d7d38a9bc8ba357d1fcb4a4be339ab5fddf8cdcc4419970e4746d1f6e"
)

print(sha256(string_to_hash.encode()).hexdigest())

# 073c348de2486c616699fcd8267dc895f2d8b43355b126295da92df2961f8a87
```

Only valid blocks should be produced and appended to the blockchain. For the purposes of this tutorial, a block is considered valid if the following conditions are met:

- The `height` of a new block is equal to the current block height plus one.
- The `previous_block_header_hash` is the correct hash of the previous block's header.
- The `block_header_hash` has at least as many leading 0s as given by the `difficulty`.
- The `transactions_merkle_root` is the valid Merkle root for the set of transactions included in the block.

For the genesis block, transactions may send funds from address 0 (`0x00...000`) to non-zero addresses so that accounts are funded. The `previous_block_header_hash` will be 0 for the genesis block.

Proof-of-work algorithm

In this tutorial, the difficulty of the block expresses the minimum number of leading 0s that the block header hash must have. In the previous example, the difficulty is 5 and there is only one leading 0 in the header hash, which means that the header would be an invalid solution to the proof-of-work. To find a valid solution, the nonce must be adjusted until a valid solution is found by incrementing it by 1 during each attempt.

In the above example, by having the nonce set to 173082, the block header hash would be a valid solution:

```
0x00000689f1d49ba1fadfe02b69e5e2894b266fa5fecb69720036195ca19f6fbc
```

Transaction specs

A transaction contains the following fields:

`sender` The address of the sender of this transaction

`receiver` The address of the receiver of this transaction

`amount` The amount being transferred from sender to recipient

`transaction_fee` The amount of fees for this transaction

`lock_time` The UNIX timestamp after which the transaction can be included in a block

`signature` The signature of the transaction.

A **transaction hash** is created by performing the following steps:

1. Sort all the above fields in alphabetical order by their key.
2. Produce a comma-separated string containing all the values, without any space. Numbers (`amount`, `lock_time`, `transaction_fee`) should be encoded as decimal value without any leading 0s. The `signature` and addresses (`sender`, `receiver`) should be hex-encoded.
3. Hash the string produced in step 2 using the SHA-256 hash function (remember to ensure that the hex string starts with `0x`).

All transactions provided in the `mempool.json.gz` file are valid (i.e. the accounts always hold sufficient funds for the transactions they perform and the signature for a given transaction is always valid).

Merkle root

The aim is to create a Merkle tree that represents the set of all transactions in a block. The root shall be included in the block header, as specified earlier. Each transaction is represented by its hash. The hash of a transaction is found by using the encoding specified, taking all the fields of the transaction (including the signature).

The leaves in the Merkle tree are the transaction hashes. If at any step in the Merkle tree creation, there is an odd number of hashes, a null string (`0x` followed by 64 0s) should be appended to the list of hashes.

On a high-level, given five transaction hashes t_1, t_2, t_3, t_4, t_5 and a hash function H, the Merkle tree would be constructed as follow:

$$\begin{aligned}
n_{11} &= H(t_1, t_2) \\
n_{12} &= H(t_3, t_4) \\
n_{13} &= H(t_5, 0) \\
n_{21} &= H(n_{11}, n_{12}) \\
n_{22} &= H(n_{13}, 0) \\
n_{31} &= H(n_{21}, n_{22})
\end{aligned}$$

where n_{ij} is the jth node at depth i, $i = 0$ being the leaves (transaction hashes). In this example, the Merkle root is n_{31}. For this tutorial, given $+$ being the string concatenation operator, we will use the following hash function:

$$H(a,b) = \begin{cases} \text{SHA256}(a+b) & \text{if } a < b \\ \text{SHA256}(b+a) & \text{otherwise} \end{cases}$$

In order to prove that, for instance, the transaction hash t_1 is included in the Merkle tree, one would require the following nodes: t_1, t_2, n_{12}, n_{22}. From this information, you can then arrive at a Merkle root. If this root constructed from the proof matches the Merkle root in some block header, the proof is valid. When validating a Merkle proof, you are free to choose a leaf node, i.e., a transaction in some block, and create the appropriate proof from this.

Bonuses

- Keep track of the balances of all addresses as blocks are mined.
- Verify signatures of transactions (the signature must be valid for the sender of the transaction).

  The `signature` field in the transactions that you are given was computed using the following steps:

  1. Encode all the above fields using the same encoding as the block above.
  2. Hash the resulting string using SHA256.
  3. Sign the hash using the private key of the sender and ECDSA with the NIST256p curve and hex-encode the result.
  4. Hex-encode the DER-encoded public key of the sender.
  5. Concatenate the results 4 and 3 (in this order), separated by a comma.

  Note that an address is obtained as follows:

  1. DER-encode the ECDSA public key
  2. SHA-256 the result of 1
  3. Take the last 20 bytes of 2
  4. Hex-encode the result of 3

  To verify the validity of a signature, you need to perform the following:

  1. Extract the public key and the signature from the `signature` field
  2. Verify that the signature is valid for the public key, given the hash of the transaction as the message to sign
  3. Check that the public key does correspond to the address of the sender

- Generate new transactions that can be included in a further block. Note that all the private keys of the addresses used are stored in `keys.json.gz` and DER-encoded, then hex-encoded. A private key can be loaded and used with the following code snippet:

```
import ecdsa

private_key_str = (
  "0x30770201010420ee13c39177bbef8319c304412bd1b75ace9c1f031b3b167ada68984d94ed738da00a06082a8648ce3d030107"
  "a14403420004bffb73356f634aa98ba07defc0bd7e943833fb93b02f0f4cd1ab0c0f25c79ee801f0ff9ed071fba5e63e66613332"
  "8eb50f24e783f12797f6f0489d568d1a8a10"
)

private_key_bytes = bytes.fromhex(private_key_str[2:])
private_key = ecdsa.SigningKey.from_der(private_key_bytes)

tx_hash_to_sign = "0x...." # hash of encoded transaction as described in the previous bonus task
signed_hash = "0x" + private_key.sign_digest(bytes.fromhex(tx_hash_to_sign[2:])).hex()
```

Note that this uses the ecdsa library

Sample usage

We do not enforce any way to interact with the code you created but here is a sample usage of the CLI tool used in the sample solution (that will be made public later).

```
# produce 15 blocks, reading the state from ./data/blockchain.json.gz
# the mempool from ./data/mempool.json.gz
# writing the new state to new-blockchain.json.gz and
# the new mempool to new-mempool.json.gz
blockchain-poc --blockchain-state ./data/blockchain.json.gz \
               produce-blocks \
               --mempool ./data/mempool.json.gz \
               --blockchain-output new-blockchain.json.gz \
               --mempool-output new-mempool.json.gz \
               -n 15

# get the hash of the 7th transaction in block 18
blockchain-poc --blockchain-state ./data/blockchain.json.gz get-tx-hash 18 7

# generate the inclusion proof for the 7th transaction in block 18 (using
   hash retrieved above)
# and save it to proof.json
blockchain-poc \
    --blockchain-state ./data/blockchain.json.gz \
    generate-proof 18 \
    0x1a7812629c3dc0badb786d2de1e77e4a81cc7eb89c2afb2fb71da7c9a6427a6b \
    -o proof.json

# verify the inclusion proof saved in proof.json
blockchain-poc --blockchain-state ./data/blockchain.json.gz \
               verify-proof proof.json

# generate 2000 new transactions using accounts in data/keys.json.gz and
# save them to new-mempool.json.gz
blockchain-poc --blockchain-state ./data/blockchain.json.gz \
               generate-txs -n 2000 -o new-mempool.json.gz \
               -a data/keys.json.gz
```