Cryptocurrencies (WS2025/26) Prof. Zeta Avarikioti, Prof. Matteo Maffei

Kerma - Task 3

1 General Notes

In your group of up to three students, you may use any programming language of your choice to implement the following task. We provide skeleton projects in Python and TypeScript (see `python-skeleton-for-task-3.tar.gz` and `typescript-skeleton-for-task-3.tar.gz` on TUWEL). You can use them to build upon. After the deadline, these skeleton projects will be updated and act as sample solutions for the previous tasks. These are guaranteed to pass all test cases for the previous tasks.

2 High level Overview

After completing this task, your node should be able to:

- validate blocks with known ancestry.
- execute transactions.

3 Task Description

Important notice - Deviations from the protocol description

For this task, you do not have to implement recursive object fetching of blocks. This means that you can assume that your node will receive all blocks in the correct order during grading. Concretely:

- If (valid) block A is the parent of block B, then A will be sent to your node before B.
- If this is not the case, i.e. if your node receives this block B but has never received block A, then it must send an UNKNOWN_OBJECT error message.

Your node however must be able to fetch all unknown transactions in a block.
Other than these deviations, the protocol description has general precedence over task descriptions - when in doubt, follow the protocol description. Be sure to also read the respective sections before you start working on this task.

4 Block Validation

In this exercise, you will implement block validation for your `Kerma` node.

1. Create the logic to represent a block, as defined in the protocol description.
2. Check that the block contains all required fields and that they are of the correct format.
3. Ensure that the target is the one required, i.e., `"0000abc0000000000000000000000000000000000000000000000000000000"`
4. Check the proof-of-work.
5. Check that for all the txids in the block, you have the corresponding transaction in your local object database. If not, then send a `"getobject"` message to your peers in order to get the transaction and leave the validation of the block pending. Resume validation once you received all transactions.
6. For each transaction in the block, check that the transaction is valid, and update your UTXO set based on the transaction. More details on this in Section 5. If any transaction is invalid, the whole block must be considered invalid.
7. Check for coinbase transactions. There can be at most one coinbase transaction in a block. If present, then the txid of the coinbase transaction must be at index 0 in `txids`. The coinbase transaction cannot be spent in another transaction in the same block (this is in order to make the law of conservation for the coinbase transaction easier to verify).
8. Validate the coinbase transaction if there is one.
   (a) Check that the coinbase transaction has no inputs, exactly one output and a height. Check that the height and the public key are of the valid format.
   (b) Verify the law of conservation for the coinbase transaction. The output of the coinbase transaction can be at most the sum of transaction fees in the block plus the block reward. In our protocol, the block reward is a constant 50×10^{12} picaker. The fee of a transaction is the sum of its input values minus the sum of its output values.
9. When you receive a block object from the network, validate it. If valid, then store the block in your local database and gossip the block to all of your peers as you did with transactions in the last task.

5 Maintaining UTXO Sets

Overview - UTXO set

The UTXO set ("unspent transaction output set") contains all unspent transaction outputs and is used to quickly check if a new transaction may use a transaction output as an input.

The following scenario explains how a UTXO set should be maintained: Assume that a chain of blocks (g, b_0, b_1) with g the genesis block is valid. This defines an execution order of all transactions included in these blocks. Call these transactions (t_0, t_1).
Assume further:

- t_0 is a coinbase transaction and the only transaction in block b_0.
- t_1 is a transaction spending from the first output of t_0 and creating two outputs. It is contained in b_1.

Now assume that a node N regards (g, b_0) as the current longest chain, because it has not yet received block b_1.
The UTXO set of N at this point is $\{(t_0, 0)\}$, with $(t_0, 0)$ denoting the output at index 0 of transaction t_0.
Now one of its peers sends N block b_1. N fetches the transaction t_1 and validates it "in isolation", i.e. everything you did in the previous task. Additionally, it now has to check if this transaction may spend from its referenced inputs. It could be the case that t_1 would spend from an output of a transaction which is not confirmed in the chain or would spend from a transaction output that has already been spent by another transaction; in both cases the whole block must be considered invalid. t_1 spends from $(t_0, 0)$: This is possible, because this particular transaction output exists and is not yet spent. To determine this, N just checks if this transaction output is in the current UTXO set. Transaction t_1 then gets "executed", which means that N needs to update its UTXO set: Let u be the current UTXO set, i the set of inputs of t_1, and o the set of its outputs. Then, the updated UTXO state u' is defined as $u' = (u \setminus i) \cup o$.

The reason for maintaining a UTXO set is because verification of new transactions is as simple as a set inclusion check - otherwise you might need to traverse the whole chain to determine if an output is unspent.

In this exercise, you will implement a UTXO set and update it by executing the transactions of each block that you receive. This task will not yet cover all the features of the UTXO set. We will revisit this part in future homeworks.

1. For each block in your database, store a UTXO set that will be computed by executing the transactions in that block. This set is not modified when you receive transactions, only when they get executed during handling of new block objects.
2. When you receive a new block, you will compute the UTXO set after that block in the

following way. To begin with, initialize the UTXO set to the UTXO set after the parent block (the block corresponding to the `previd`). Note that the UTXO set after the genesis block is empty. For each transaction in the block:

(a) Validate the transaction as per your validation logic implemented in Task 2. Additionally, check that each input of the transaction corresponds to an output that is present in the UTXO set. This means that the output exists and also that the output has not been spent yet.

(b) Apply the transaction by removing UTXOs that are spent and adding UTXOs that are created. Update the UTXO set accordingly.

(c) Repeat steps a-b for the next transaction using the **updated** UTXO set. This is because a transaction can spend from an output of a non-coinbase transaction in the same block.

For testing your node, you can use the genesis block which is a valid block. Below is another block mined on the genesis block that is valid. You can also mine more blocks to test your validation.

Example valid coinbase transaction

```
// objectid is 6ebfb4c8e8e9b19dcf54c6ce3e1e143da1f473ea986e70c5cb8899a4671c933a
{
  "height": 1,
  "outputs": [
    {
      "pubkey": "3f0bc71a375b574e4bda3ddf502fe1afd99aa020bf6049adfe525d9ad18ff33f",
      "value": 50000000000000
    }
  ],
  "type": "transaction"
}
```

Example valid block

```
// objectid is 00004d07d79f728a27ee72334b17425b83e5d59e1f1abad46f9f1936eeb6a517
{
  "T": "0000abc0000000000000000000000000000000000000000000000000000000",
  "created": 1671148800,
  "miner": "grader",
  "nonce": "000000000000000000000000000000000000000000000000000000004e315",
  "note": "This block has a coinbase transaction",
  "previd": "0000000052a0e645eca917ae1c196e0d0a4fb756747f29ef52594d68484bb5e2",
  "txids": [
    "6ebfb4c8e8e9b19dcf54c6ce3e1e143da1f473ea986e70c5cb8899a4671c933a"
  ],
  "type": "block"
}
```

The UTXO set after this block is $\{(6ebfb4c8...71c933a, 0)\}$.

6 Sample Test Cases

IMPORTANT: Make sure that your node is running at all times! Therefore, make sure that there are no bugs that crash your node. If our automatic grading script can not connect to your node, you will not receive any credit. Taking enough time to test your node will help you ensure this. Below is a (non-exhaustive) list of test cases that your node will be required to pass. We will also use these test cases to grade your submission. Consider two nodes Grader 1 and Grader 2.

1. On receiving an `object` message from Grader 1 containing any invalid block, Grader 1 must receive a correct error message and the transaction must not be gossiped to Grader 2. Beware: invalid blocks may come in many different forms! Some examples are as follows:
   (a) The block has an incorrect target.
   (b) The block's proof-of-work is not valid.
   (c) There is an invalid transaction in the block.
   (d) There are two transactions in the block that spend the same output.
   (e) A transaction attempts to spend an output that is not in the UTXO.
   (f) The coinbase transaction has an output that exceeds the block rewards and the fees.
2. On receiving an `object` message from Grader 1 containing a valid block, the block must be gossiped to Grader 2 by sending an `ihaveobject` message with the correct blockid.

How to Submit your Solutions We will grade your solution locally. Please upload your submission as a tar archive. When grading your solution, we will perform the following steps:

```
tar -xf <your submission file> -C <grading directory>
cd <grading directory>
docker-compose build
docker-compose up -d
# grader connects to localhost port 18018 and runs test cases
docker-compose down -v
```

When started in this way, there should be neither blocks (except the genesis block) nor transactions in the node's storage.

If you use the skeleton templates, you can use the provided Makefile targets to build and check your solution. Note however, that this will only check if the container can be built, started and a connection can be established to localhost:18018. We highly recommend that you write your own test cases.

The deadline for this task is 30th November, 11.59pm. We will not accept any submissions after that. Each group is required to submit **one** solution. Plagiarism is unacceptable. If you are caught handing in another group's code, your members will receive zero points.