Cryptocurrencies (WS2023/24) Dr. Zeta Avarikioti, Prof. Matteo Maffei

Kerma - Task 2

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-2.tgz` and `typescript-skeleton-for-task-2.tgz` 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:

- store objects.
- validate transactions.
- send and receive objects over the network.

3 Task Description

Important notice - Deviations from the protocol description

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

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

Furthermore, you are not required to implement verification of blocks yet. If your node receives an object with a type not equal to "transaction", you may assume this object is of INVALID_FORMAT. During grading of Task 2 your node will not receive block objects.

Other than these deviations, the protocol description has general precedence over task descriptions - when in doubt, follow the protocol description.

3.1 Object Exchange and Gossiping

In this exercise, you will extend your `Kerma` node to implement content addressable object exchange and gossiping.

1. Maintain a local database of known objects. The database should survive reboots.
2. Implement a function to map objects to objectids. The objectid is obtained by taking the blake2s hash of the canonical JSON representation of the object. You can test your function using the Genesis block and its blockid given in the protocol description.
3. Implement object exchange using the `getobject, ihaveobject, object` messages.
   a) On receiving an `ihaveobject` message, request the sender for the object using a `getobject` message if the object is not already in your database.
   b) On receiving an object, ignore objects that are already in your database. Accept objects that are new and store them in your database if they are valid.
   c) Implement gossiping: Broadcast the knowledge of newly received valid objects to your peers by sending an `ihaveobject` message to all your connected peers.
   d) On receiving a `getobject` message, send the requested object if you have it in your database.

3.2 Transaction Validation

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

1. Create the logic to represent a transaction. The protocol description defines the structure of a transaction.
2. Create the logic for transaction validation as specified by the protocol description. Transaction validation contains the following steps:
   a) For each input, validate the outpoint. For this, ensure that a valid transaction with the given txid exists in your object database and that it has an output with the given index.
   b) For each input, verify the signature.
   c) Outputs contain a public key and a value. The public keys must be in the correct format and the values must be a non-negative integer.
   d) Transactions must satisfy the weak law of conservation: The sum of input values must be equal or exceed the sum of output values.

   For now, assume that a (syntactically valid) coinbase transaction is always valid. I.e., you do not have to check how many new coins have been created or what the height is set to. We will validate these starting in the next homework.

3. When you receive a transaction object, validate it. If the transaction is valid, store it in your object database and gossip it using an `ihaveobject` message. If it is invalid, send an `error` message to the node who sent the transaction and do not gossip it. In case the other node sent you an invalid transaction, you should consider the other node faulty. If you could not verify a transaction because it references an object not known to you, this does not indicate a faulty communication partner and you should not close the connection, just send an UNKNOWN_OBJECT error message (for now).

You should test your transaction validation by generating different valid and invalid transactions, signed using a private key of your choice.

3.2.1 Example

Here is a simple example that you can use for testing (the first is a object message containing a valid coinbase transaction, the second contains a valid transaction that spends[1] from the first):

Example valid object message with coinbase transaction

```
// object id is d46d09138f0251edc32e28f1a744cb0b7286850e4c9c777d7e3c6e459b289347
{
  "object": {
    "height": 0,
    "outputs": [
      {
        "pubkey": "85acb336a150b16a9c6c8c27a4e9c479d9f99060a7945df0bb1b53365e98969b",
        "value": 50000000000000
      }
    ],
    "type": "transaction"
  },
  "type": "object"
}
```

[1] Note that this transaction only really "spends" from the coinbase transaction if it is included in a block. This means it is perfectly fine if you receive transactions A, B and C that all spend from the same output - you should consider them all valid, gossip and store them in your database. The logic how transactions are confirmed in blocks will be implemented in the next tasks.

Example valid object message with transaction

```
// object id is 895ca2bea390b7508f780c7174900a631e73905dcdc6c07a6b61ede2ebd4033f
{
  "object": {
    "inputs": [
      {
        "outpoint": {
          "index": 0,
          "txid": "d46d09138f0251edc32e28f1a744cb0b7286850e4c9c777d7e3c6e459b289347"
        },
        "sig": "6204bbab1b736ce2133c4ea43aff3767c49c881ac80b57ba38a3bab980466644
                cdbacc86b1f4357cfe45e6374b963f5455f26df0a86338310df33e50c15d7f04"
      }
    ],
    "outputs": [
      {
        "pubkey": "b539258e808b3e3354b9776d1ff4146b52282e864f56224e7e33e7932ec72985",
        "value": 10
      },
      {
        "pubkey": "8dbcd2401c89c04d6e53c81c90aa0b551cc8fc47c0469217c8f5cfbae1e911f9",
        "value": 49999999999990
      }
    ],
    "type": "transaction"
  },
  "type": "object"
}
```

4 Sample Test Cases

Important: make sure your node is running all the time. 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. Object Exchange:

   - If Grader 1 sends a new valid transaction object and then requests the same object, Grader 1 should receive the object.
   - If Grader 1 sends a new valid transaction object and then Grader 2 requests the same object, Grader 2 should receive the object.

- If Grader 1 sends a new valid transaction object, Grader 2 must receive an `ihaveobject` message with the object id.
- If Grader 1 sends an `ihaveobject` message with the id of a new object, Grader 1 must receive a `getobject` message with the same object id.

2. Transaction Validation:

- On receiving an object message from Grader 1 containing any invalid transactions, Grader 1 must receive an error message and the transaction must not be gossiped to Grader 2. Beware: invalid transactions may come in many different forms!
- On receiving an object message from Grader 1 containing a valid transaction, the transaction must be gossiped to Grader 2.

5 Grading

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 7th November, 11.59pm. We will not accept any submissions after that. One submission per group is sufficient. Plagiarism is unacceptable. If you are caught handing in someone else's code, you will receive zero points.