Cryptocurrencies (WS2025/26) Prof. 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.tar.gz` and `typescript-skeleton-for-task-2.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 IP addresses of peers
- store objects.
- validate transactions.
- send and receive objects over the network.

3 Task Description

3.1 Peers

Every node should keep a set of known peer addresses to be able to actively connect to other nodes running the protocol.
The `peers` message can be volunteered or sent in response to a `getpeers` message. It contains a `peers` key which is an array of size in range [0, 30], i.e. contains at most 30 entries, but an empty array is also valid. Every `peer` is a string in the form of `host:port`. `port` is a valid port, i.e. a decimal number in range [1, 65535]. The default port is 18018. You can host your node on any port, but your submission must listen at port 18018. `host` is either a valid DNS entry or a syntactically valid IPv4 address in decimal form. A `DNS entry` in our protocol is considered valid if it satisfies the following properties:

- it matches the regular expression `[a-zA-Z\d\.\-\_]{3,50}`, i.e. it is a string of length in range [3, 50] and contains only letters (a-z and A-Z), digits (0-9), dots (.), hyphens (-) or underscores (_).

- there is at least one dot in the string which is not at the first or last position.
- there is at least one letter (a-z or A-Z) in the string. *

As an example, the following `peers` are invalid:

- 256.2.3.4:18018 (neither a valid ip address nor a valid DNS entry)
- 1.2.3.4.5:678 (neither a valid ip address nor a valid DNS entry)
- 1.2.3.4:2000000 (invalid port)
- nodotindomain:1234 (no dot in domain)
- kermanode.net (no port given)

If a peer in a `peers` message is not syntactically valid, you must send an INVALID_FORMAT error message and consider your communication partner faulty. Otherwise, add all peers in this message to your known peers.

Optional - Further address checks

You are allowed to perform further checks and to apply heuristics to keep your list of known peers as best as possible by choosing not to store a new peer in your known peers. "As best as possible" means that you want to have as many reachable and correctly working nodes and as few unreachable or faulty nodes as possible in your known peers. However, your node must satisfy the following condition: As long as your node does not know more than 30 peers, if it receives a peer which has a syntactically valid address, is reachable and running the Kerma protocol, then your node must include this peer address in any subsequent reply to a `peers` message.

A node should include itself in the list of peers at the first position if it is listening for incoming connections. Your node should always listen for and accept new incoming connections.

*If we would not require this property, then the checks for syntactically valid ip addresses would be unnecessary because the ip address format satisfies the first two properties, and e.g. 300.300.300.300 would be a valid host.

Explanation - Why you should add yourself to `peers` messages

Consider the following scenario: Assume you have hosted your node on a server that has a non-static ip address which is currently 1.2.3.4 and the hostname "mykermanode.net" always points to your server. If you connect to the network for the first time, no one knows that the address of your node can be determined by looking up "mykermanode.net". The only thing your connected peer knows about you is that you connected from 1.2.3.4. Provided your node is listening on the standard port 18018, the connected node can guess correctly that your node can be reached at 1.2.3.4:18018. This knowledge will eventually reach other nodes, which might want to connect to you. But in the meantime, your ip address might have already been updated, causing any connection attempt to fail. Therefore, your node should behave in the following way:

- It should not try to guess the listening address of a communication partner.
- It should add its DNS entry + listening port at the first position to every "peers" message it sends. If you do not use a DNS entry but have a static ip address, add this ip + port instead.

It is fine if you hardcode these values or pass them as cli arguments, they don't have to match the ip address of the machine on which we will locally test your node during grading.
The same argument also applies if you host your node on a static ip address on a non-standard port: The port used for outgoing connections will be different from your listening port, therefore any node you connect to will not know on which port your node is listening for incoming connections.

Here is an example of a valid `peers` message:

Valid peers message

```
{
   "type":"peers",
   "peers":[
      "kermanode.net:18017",
      "138.197.191.170:18018"
   ]
}
```

If you find that a node is faulty, disconnect from it and remove it from your set of known peers (i.e., forget them). Likewise, if you discover that a node is offline, you should forget it. You must not, however, block further communication requests from this node or refuse to add this node again to your known nodes if another node reports this as known. Note that there may are (edge) cases where forgetting a node is not possible - we will not check this behaviour. The idea behind this is that your node will not create lots of traffic by running in

an infinite loop: Connecting to a node, downloading its (invalid) chain, disconnecting because an erroneous object was sent, and connecting again.

Explanation - Edge cases when forgetting nodes

Whenever a remote node that initiated a connection to your node turns out to be faulty, you cannot correctly "forget" it. If the remote node is hosted on a dynamic ip address and a DNS entry points to it, you will only get the current ip address from the connection, not the hostname of this node. You then would need to look up all DNS entries know to you and if you find a match, you know the hostname of the remote node. Still, if multiple nodes are hosted behind the same ip address, you would not be able to deduce which node connected to you. Therefore, this case also applies if the remote node is hosted on a static ip address known to you. Relying on the first entry in the peers message which should be the hostname of the remote node or even on the agent key would be possible, but very easily abusable by a dishonest adversary. Because of this, you should (at least for now) only forget the addresses of peers which you used yourself to initiate the connection, thus knowing exactly who you are connected to.

3.2 Object Exchange and Gossiping

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.

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.

3.3 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.

3.3.1 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[†] 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"
}
```

[†]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.

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 14th 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.