SPCDSS'25 Prof. Tom Van Cutsem, Dr. Davy Preuveneers

Bitmunt - Task 2

High level Overview

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

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

1 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 yet required to implement verification of blocks. If your node receives an object with a type not equal to "transaction", you may assume this object is of `INVALID_FORMAT`. During the 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. Be sure to also read the respective sections before you start working on this task.

In this exercise, you will extend your `Bitmunt` 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. Canonical representation is important so that each object has a unique objectid. You can test your function using the Genesis block and its blockid given below.

Genesis Block

```
// The genesis blockid is 00000003aa05a8b3ec33a789d2a28a8ece1b33141eb23b4d4b5715685d7a8471
{
  "T": "0000000abc0000000000000000000000000000000000000000000000000000",
  "created": 1740224556,
  "miner": "Bitmunt",
  "nonce": "000000000000000000000000000000000000000000000000000000001d38904",
  "note": "VRT News 2025-02-02: 2 op de 5 IC-treinen en geen P-treinen: nmbs verwacht maandag grote hinder",
  "previd": null,
  "txids": [],
  "type": "block"
}
```

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.

2 Transaction Validation

In this exercise, you will implement transaction validation for your `Bitmunt` 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.
   Otherwise, send an "`UNKNOWN_OBJECT`" error message.

   b) For each input, if a transaction with the given txid exists but the outpoint index is invalid, send an "`INVALID_TX_OUTPOINT`" error message.

   c) For each input, verify the signature. Our protocol uses ed25519 signatures. If the transaction signature is `null` or if it is not a valid signature given the public key, you should send an "`INVALID_TX_SIGNATURE`" error message.

   d) Ensure that a transaction does not have multiple inputs that have the same outpoint. Otherwise, send an "`INVALID_TX_OUTPOINT`" error message.

d) 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.
Otherwise, send an "`INVALID_FORMAT`" error message.

e) Transactions must satisfy the weak law of conservation: The sum of input values must be equal or exceed the sum of output values.
Otherwise, send an "`INVALID_TX_CONSERVATION`" error message.

f) In other cases, or if the transaction has an incorrect JSON format or invalid properties, you should send an "`INVALID_FORMAT`" error message. Note that this error message also encompasses basic parsing errors such as ensuring that the inputs' outpoints' indexes and the outputs' values are non-negative.

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.

2.1 Example

Here is a simple example that you can use for testing (the first is a valid coinbase transaction, the second contains a valid transaction that spends* from the first):

Example valid coinbase tx

```
// txid is 644d9593a17873be3522e47f81775d10806b5f605b82a847d8b175777eeeb41c
{
  "height": 1,
  "outputs": [
    {
      "pubkey": "7cb057a09fb1c0b38b430e3c9deae5607c32665d526ee75724cb0615319a87d7",
      "value": 50000000000000
    }
  ],
  "type": "transaction"
},
```

*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 tx that spends from the coinbase tx above

```
// txid is 694330761ff59515171b16274ff5b2160312620f7e9aea2bbde15aaaf04f4f5b
{
  "inputs": [
    {
      "outpoint": {
        "index": 0,
        "txid": "d46d09138f0251edc32e28f1a744cb0b7286850e4c9c777d7e3c6e459b289347"
      },
      "sig": "8c7881b6dc560b5f5ab930787409999df81437b4a99eca43ec2e50c55362f5af2874f3e
            79d0652696716e940f8eb2f18347f77fe2ac32690b8ad3632c5da8a07"
    }
  ],
  "outputs": [
    {
      "pubkey": "26e13b5ecebcb5b828b809372951ad2a6cc9c892d68c79c8d79221fb7a520001",
      "value": 30000000000000
    },
    {
      "pubkey": "7cb057a09fb1c0b38b430e3c9deae5607c32665d526ee75724cb0615319a87d7",
      "value": 10000000000000
    }
  ],
  "type": "transaction"
}
```

3 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.

The **soft deadline** for this task is **27th March, 2025, 11.59pm**.
The **hard deadline** for this task is **3rd April, 2025, 11.59pm**.
We will not accept any submissions after the hard deadline. Please ensure you correctly tag your commit and push it to the `main` branch.