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

Bitmunt - Task 3

High level Overview

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

- validate blocks with known ancestry.
- execute transactions in a block.

1 Block Validation

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 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.

In this exercise, you will implement block validation for your `Bitmunt` 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 in the correct format. Otherwise, send back an `INVALID_FORMAT` error.
3. Ensure if present that the `note` and `miner` fields in a block are ASCII-printable strings up to 128 characters long each. ASCII printable characters are those with decimal values 32 up to 126. If this is not the case, send back an `INVALID_FORMAT` error.
4. Ensure if present that the studentids field is an array with at most 10 ASCII-printable strings each containing up to 128 characters. If the studentids field is present but does not follow these constraints, send back an `INVALID_FORMAT` error.

5. Ensure that the target is the one required, i.e.
   `"0000000abc000000000000000000000000000000000000000000000000000000"`
   Otherwise, send back an `INVALID_FORMAT` error.

6. Check the proof-of-work. If not valid, send back an `INVALID_BLOCK_POW` error.

7. 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, or abort the validation if a timeout occurs to prevent indefinite waiting. If you cannot find the transaction and none of your peers have found it, then send an `UNFINDABLE_OBJECT` error to the peer who sent you the block.

8. 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 2. If any transaction is invalid, the whole block must be considered invalid. Since you should not add such invalid transaction to your database and other peers should not send it back to you, send back an `UNFINDABLE_OBJECT` error in the case of an invalid transaction in a block.

9. For all the transactions in the block, if there is more than one transaction spending from the same output, then send back an `INVALID_TX_OUTPOINT` error.

10. After implementing your UTXO set, for each transaction, check that it only spends an output that is in the UTXO set. Otherwise, send back an `INVALID_TX_OUTPOINT` error.

11. 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`. Otherwise, send back an `INVALID_BLOCK_COINBASE` error.

12. 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). Otherwise, send back an `INVALID_TX_OUTPOINT` error.

13. 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. (We will check if the height is correct in the next homework when we validate chains, not now.) Otherwise, send back an `INVALID_FORMAT` error.

    (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} sassaman. The fee of a transaction is the sum of its input values minus the sum of its output values.
    Otherwise, send back an `INVALID_BLOCK_COINBASE` error.

14. 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 by using an `ihaveobject` message.

2 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 the 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;
- t_1 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. Right now, you don't have to maintain the mempool UTXO set.

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. If the output is not present in the UTXO set, send back an `INVALID_TX_OUTPOINT` error.

   (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 now, you may assume that the previous block was sent to you beforehand so that you have the UTXO set after the previous block. You may ignore the block if you do not have its previous block. In the next homework, you will also make sure to download and validate the previous 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 in block 1

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

Example valid block with height 1

```
// objectid is 000000000ea8b85a361f7640d12467a6eb68825b454b3ff10174ca5f6f8b064c
{
  "T": "0000000abc000000000000000000000000000000000000000000000000000000",
  "created": 1740476899,
  "miner": "Bitmunt Bounty Hunter",
  "nonce": "000000000000000000000000000000000000000000000000000000000027e120d6",
  "note": "First block on genesis, got 50 bm reward!",
  "previd": "00000003aa05a8b3ec33a789d2a28a8ece1b33141eb23b4d4b5715685d7a8471",
  "txids": [
    "644d9593a17873be3522e47f81775d10806b5f605b82a847d8b175777eeeb41c"
  ],
  "type": "block"
}
```

3 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. (`INVALID_FORMAT`)
   (b) The block's proof-of-work is not valid. (`INVALID_BLOCK_POW`)
   (c) There is an invalid transaction in the block. (`UNFINDABLE_OBJECT`)
   (d) There are two transactions in the block that spend the same output. (`INVALID_TX_OUTPOINT`)
   (e) A transaction attempts to spend an output that is not in the UTXO. (`INVALID_TX_OUTPOINT`)
   (f) The coinbase transaction has an output that exceeds the block rewards and the fees. (`INVALID_BLOCK_COINBASE`)
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.

The **soft deadline** for this task is **3rd April, 2025, 11.59pm**.
The **hard deadline** for this task is **10th 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.