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

Bitmunt: Protocol Description

1 Project Description

For this course, we will develop our own blockchain. In groups of up to three members, students will write their own independent implementation of a node for our blockchain in their programming language of choice (we recommend using either Go or Typescript). This document specifies the protocol and defines how nodes will communicate.

During your implementation, be aware that neighbouring nodes can be malicious. Your implementation must be resilient to simple and complex attacks. Simple attacks can be the supply of invalid data. Complex attacks can involve signatures, proof-of-work, double spending, and blocks, all of which must be validated carefully.

The chain is a static difficulty proof-of-work UTXO-based blockchain over a TCP network protocol.

The protocol is called `Bitmunt`. The currency is called `bitmunt`, and its abbreviation is `bm`.

1.1 Notation disambiguation

`MUST` or `MUST NOT`: This term is used to specify a property that must hold, and a node that violates such a property is considered faulty.
`SHOULD` or `SHOULD NOT`: This term is used to specify intended behaviour. A node that does not adhere to that behaviour is not necessarily faulty. However, in order to get points for your solution, you must also adhere to these specifications.

1.2 Networking

The peer-to-peer network works over TCP. The default TCP port of the protocol is 18018, but you can use any port. If your node is running behind NAT, make sure your port is forwarded.

1.3 Bootstrapping

Your node will initially connect to a list of known peers. From there, it will build its own list of known peers. We will maintain a list of bootstrapping peer IPs / domains in GitLab as they become available.

1.4 Data Validation

Your client must disconnect from a peer it is connected to in case it receives invalid data from it. Be rigorous about network data validation and do not accept malformed data.

2 Cryptographic Primitives

2.1 Hash

We use BLAKE2s as our hash function. This is used both for content-addressable application objects as well as proof-of-work. When hashes appear in JSON, they should be in hexadecimal format.

2.2 Signatures

We use Ed25519 digital signature scheme. Public keys and signatures should be byte-encoded as described in RFC 8032. A library will typically take care of this process, e.g., `crypto/ed25519` in Go programming language. Once a signature or public key is byte-encoded, it is converted to hex in order to represent a string within JSON. Whenever we refer to a “public key" or a “signature" in this protocol, we mean the byte-encoded and hexified data.

2.3 Hexification

Hex strings must be in lower case.

3 Application Objects

Application objects are objects that must be stored by each node. These are content-addressed by the BLAKE2s hash of their JSON representation. Therefore, it is important to have the same JSON representation as other clients so that the same objects are addressed by the same hash. You must normalize your JSON and ensure it is in canonical JSON form. The examples in this document contain extra whitespace for readability, but these must not be sent over the network. The BLAKE2s of the JSON contents is the `objectid`.

An application object is a JSON dictionary containing the `"type"` key and further keys depending on its type.

There are two types of application objects: *transactions* and *blocks*. Their `objectids` are called `txid` and `blockid`, respectively.

An application object must only contain specified keys, i.e., it must not contain additional keys. The same applies to messages.

3.1 Transactions

This represents a transaction and has the type `transaction`. It contains the key `inputs` containing a non-empty array of inputs and the key `outputs` containing an array of outputs.

An input contains a pointer to a previous output in the `outpoint` key and a signature in the `sig` key. The `outpoint` key contains a dictionary of two keys: `txid` and `index`. The `txid` is the objectid of the previous transaction, while the `index` is the natural number (zero-based) indexing an output within that transaction. The `sig` key contains the signature.

Signatures are created using the private keys corresponding to the public keys that are pointed to by their respective `outpoint`. Signatures are created on the plaintext, which consists of the transaction they (not their public keys!) are contained within, except that the `sig` values are all replaced with `null`. This is necessary because a signature cannot sign itself.

An output is a dictionary with keys `value` and `pubkey`. The `value` is a non-negative integer indicating how much value is carried by the output. The `value` is denominated in *sassaman*, the smallest denomination in `Bitmunt`.

$$1 \text{ bm} = 10^{12} \text{ sassaman}.$$

The `pubkey` is a public key of the recipient of the money. The money carried by an output can be spent by its owner by using it as an input in a future transaction.

This is an example of a (syntactically) valid transaction:

Valid normal transaction

```
{
   "type": "transaction",
   "inputs": [
      {
         "outpoint":{
            "txid": "1d9ddbdcbf1c1d531a2bedae08e809d2007da5bf48232a4aa03b645f2c30ebcb",
            "index": 0
         },
         "sig": "8c7881b6dc560b5f5ab930787409999df81437b4a99eca43ec2e50c55362f5af287
                 4f3e79d0652696716e940f8eb2f18347f77fe2ac32690b8ad3632c5da8a07"
      }
   ],
   "outputs": [
      {
         "pubkey": "2b5001f5aa6759139c61213e56e3df01574ea53ac5a936ff1ce9b53162d7703e",
         "value": 329999999999999
      }
   ]
}
```

Transactions must satisfy the weak law of conservation: The sum of input values must be

equal to or exceed the sum of output values. Any remaining value can be collected as fees by the miner confirming the transaction.

However, this example transaction could in principle, still be (semantically) invalid in the following cases:

- An object with id f7140..aa196 exists but is a block (INVALID_FORMAT).
- The referenced transaction f7140...aa196 has no output with index 0 (INVALID_TX_OUTPOINT).
- The output at index 0 in the referenced transaction f7140...aa196 holds less than 5100000000 sassaman (INVALID_TX_CONSERVATION).
- Verifying the signature using the public key in output at index 0 in transaction f7140...aa196 failed (INVALID_TX_SIGNATURE).

3.1.1 Coinbase Transactions

A coinbase transaction is a special form of a transaction and is used by a miner to collect the block reward. See below section Blocks for more info. If the transaction is a coinbase transaction, then it must not contain an `inputs` key, but it must contain an integer `height` key with the block's height as the value.

This is an example of a valid coinbase transaction:

Valid coinbase transaction

```
{
  "type": "transaction",
  "height": 133,
  "outputs": [
    {
      "pubkey": "2b5001f5aa6759139c61213e56e3df01574ea53ac5a936ff1ce9b53162d7703e",
      "value": 50000000000000
    }
  ]
}
```

3.2 Blocks

This represents a block and has the type `block`. It must contain the following keys:

- `txids`, which is a list of the transaction identifiers within the block.
- `nonce`, which is a 32-byte hexified value that can be chosen arbitrarily.
- `previd`, which is the object identifier `blockid` of the previous block in the chain. Only the genesis block is allowed to differ from that by having a `previd` set to null.

- `created`, which is an (integer) UNIX timestamp in seconds.
- `T`, which is a 32-byte hexadecimal integer and is the mining target.

Optionally, it can contain a `miner` key and a `note` key, which can be any ASCII-printable[1] strings up to 128 characters long each. It can also optionally contain a `studentids` field, which is a list of at most 10 student IDs, each being an ASCII-printable string of up to 128 characters long.

Here is an example of a valid block:

A valid block


```
{
  "T":"0000000abc0000000000000000000000000000000000000000000000000000000",
  "created":1740476899,
  "miner":"Bitmunt Bounty Hunter",
  "nonce":"0000000000000000000000000000000000000000000000000000000027e120d6",
  "note":"First block on genesis, got 50 bm reward!",
  "previd":"00000003aa05a8b3ec33a789d2a28a8ece1b33141eb23b4d4b5715685d7a8471",
  "txids":["644d9593a17873be3522e47f81775d10806b5f605b82a847d8b175777eeeb41c"],
  "type":"block"
}
```


Block validity mandates that the block satisfies the proof-of-work equation: `blockid` $<$ `T`.

The genesis block has a null `previd`. This is our genesis block:

Genesis Block


```
{
 "T": "0000000abc0000000000000000000000000000000000000000000000000000000",
 "created": 1740224556,
 "miner": "Bitmunt",
 "nonce": "0000000000000000000000000000000000000000000000000000000001d38904",
 "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"
}
```


All valid chains must extend genesis. Each block must have a timestamp that is later than its predecessor but not after the current time.

The transaction identifiers (`txids`) in a block may (but are not required to) contain one coinbase transaction. This transaction must be the first in `txids`. It has exactly one output, which generates $50 \cdot 10^{12}$ new sassaman and also collects fees from the transactions confirmed in the block. The value in this output cannot exceed the sum of the new coins plus the fees, but

[1] ASCII-printable characters have hexcodes `0x20` to `0x7e`

it can be less than that. The `height` in the coinbase transaction must match the height of the block the transaction is contained in. This is so that coinbase transactions with the same public key in different blocks are distinct.

The coinbase transaction cannot be spent in the same block. However, a transaction can be spent from a previous, non-coinbase transaction in the same block. The order of the identifiers of the confirmed transactions in a block defines the order in which these transactions are "executed".

All blocks must have a target `T` of:
`0000000abc0000000000000000000000000000000000000000000000000000`.

The genesis `blockid` is:
`00000003aa05a8b3ec33a789d2a28a8ece1b33141eb23b4d4b5715685d7a8471`.
Check this to ensure your implementation is performing correct JSON canonicalization.

4 Messages

Every message exchanged by two peers over TCP is a JSON message. These JSON messages are separated from one another using '\n.' The JSON messages themselves must not contain new line endings ('\n'), but they may contain escaped line endings within their strings.

Every JSON message is a dictionary. This dictionary always has at least the key `type` set, which is a string and defines the message type. Each message may contain its own keys depending on its type. Fields that are required must be present, fields that are optional can be present but are not mandatory, and other fields that are not mentioned should not be present.

4.1 Hello

When you connect to another client, you and the other client must perform a simple handshake by both sending a `hello` message. The message must also contain a `version` key, which is always set to 0.10.x with x a single decimal digit. The message must also contain an `agent` key, which is an ASCII-printable string of up to 128 characters that can be chosen arbitrarily.

The `hello` message must be the first message sent by any communication partner. You can send subsequent messages immediately after the hello message, even before you receive the `hello` message of your communication partner. You must send an `INVALID_HANDSHAKE` error message and consider your communication partner as faulty in the following cases:

- A non-hello-message is sent prior to the hello message.
- No hello message has been sent 20s after connection.
- A hello message is sent after the handshake is completed.

You must send an `INVALID_FORMAT` error message and consider your communication partner as faulty in the following cases:

- The version does not match the specified format.
- The `agent` key contains non-printable characters or is longer than 128 characters.
- Any required key is omitted, or there are additional keys.

This is an example of a valid `hello` message:

Valid hello message

```
{
  "type":"hello",
  "version":"0.10.0",
  "agent":"Bitmunt Client 0.10"
}
```

After a handshake is completed, the connection should be kept alive by both communication partners. So, in principle, unless an error on the network layer occurs, a connection between two nodes should only be closed when one node detects that the other is faulty.

4.2 GetPeers

If you want to know what peers are known to your peer, you send them a `getpeers` message. This message has no additional keys and should be responded to with a `peers` message.

Valid getpeers message

```
{
 "type": "getpeers"
}
```

4.3 Peers

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 the 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 the range [1, 65535]. The default port is 18018. You can host your node on any port, but your

submission must listen to port 18018. `host` is either a valid DNS entry or a syntactically valid IPv4 address in decimal form.
Here is an example of a valid `peers` message:

Valid peers message

```
{
  "type":"peers",
  "peers":[
    "bitmuntnode.net:18017", /* dns */
    "201.39.11.167:18018" /* ipv4 */
  ]
}
```

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.

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.

4.4 GetObject

This message requests an object addressed by the given hash. It contains an `objectid` key, which is the address of the object. If the receiving peer knows about this object, it should respond with an `object` message. If the receiving peer does not know about this object, it should send an error message with the name `UNKNOWN_OBJECT`.

Valid getobject message

```
{
  "type":"getobject",
  "objectid":"00000003aa05a8b3ec33a789d2a28a8ece1b33141eb23b4d4b5715685d7a8471"
}
```

4.5 IHaveObject

This message advertises that the sending peer has an object with a given hash addressed by the `objectid` key. The receiving peer may request the object (using `getobject`) in case it does not have it.

Valid IHaveObject message


```
{
 "type": "ihaveobject",
 "objectid": "00000003aa05a8b3ec33a789d2a28a8ece1b33141eb23b4d4b5715685d7a8471"
}
```


In our gossiping protocol, whenever a peer receives a new object and validates the object, then it advertises the new object to its peers.

4.6 Object

This message sends an object from one peer to another. This can be voluntary or as a response to a `getobject` message. It contains an `object` key, which contains the object in question.

Valid object message


```
{
 "type": "object",
 "object": {
  "T": "0000000abc0000000000000000000000000000000000000000000000000000",
  "created": 1740224556,
  "miner": "Bitmunt",
  "nonce": "000000000000000000000000000000000000000000000000000001d38904",
  "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"
 }
}
```


4.7 GetChainTip

Request the current blockchain tip of the peer with a message of type `getchaintip`. There are no further keys in this message. A peer should respond to it with a `chaintip` message.

Valid GetChainTip message


```
{
 "type": "getchaintip"
}
```

4.8 ChainTip

This message with type `chaintip` is sent as a response to the `getchaintip` message, or it can be volunteered. It includes a single key called `blockid` with the `blockid` of the tip of the current longest chain.

Valid ChainTip message

```
{
 "type": "chaintip",
 "blockid": "00000003aa05a8b3ec33a789d2a28a8ece1b33141eb23b4d4b5715685d7a8471"
}
```

The receiving peer can then use `getobject` to retrieve the block contents and then recursively follow up with more `getobject` messages for each parent block to retrieve the whole blockchain if needed.

4.9 GetMempool

Request the mempool of the peer with a message of type getmempool. There are no additional keys in this message. The peer should respond with a `mempool` message.

Valid getmempool message

```
{
 "type":"getmempool"
}
```

4.10 Mempool

This message with type `mempool` is sent as a response to a `getmempool` message, or it can be volunteered. It includes a list of all txids that the sending peer has in its mempool, i.e., are not yet confirmed. These are included in an array with the key `txids`.

Valid Mempool message

```
{
 "type": "mempool",
 "txids": []
}
```

The receiving peer should attempt to add these transactions into their own mempool.

4.11 Error

You should send objects with implementation-specific error messages to describe any exceptions encountered. An error object should be of type “error” and contain a name key containing a predefined string value (all possible options are listed below) and a description key that describes the error.

Valid error message

```
{
  "type":"error",
  "name":"INVALID_FORMAT",
  "description":"The note field in the block message contains more than 128 characters."
}
```

Here is the complete list of accepted error names:

- INTERNAL_ERROR: Something unexpected happened.
- INVALID_FORMAT: The format of the received message is invalid.
- UNKNOWN_OBJECT: The object requested is unknown to that specific node.
- UNFINDABLE_OBJECT: The object requested could not be found in the node’s network after a timeout.
- INVALID_HANDSHAKE: The peer sent other validly formatted messages before sending a valid `hello` message.
- INVALID_TX_OUTPOINT: The transaction outpoint index is too large.
- INVALID_TX_SIGNATURE: A signature is not valid.
- INVALID_TX_CONSERVATION: The transaction does not satisfy the weak law of conservation.
- INVALID_BLOCK_COINBASE: The block coinbase transaction is invalid.
- INVALID_BLOCK_TIMESTAMP: The `created` key contains a timestamp not strictly greater than that of the parent block or is in the future.
- INVALID_BLOCK_POW: A block does not meet the required mining target.
- INVALID_GENESIS: A block other than the genesis block with `previd` set to null was sent.

You do not have to reply to an error message that is sent to you or perform any action. It might be interesting to log errors you receive because they might lead you to implementation errors in your node.