Cryptocurrencies (WS2022/23) Dr. Zeta Avarikioti, Prof. Matteo Maffei

Kerma: Protocol Description

Project Description

For this course, we will develop our own blockchain. Each student will write their own independent implementation of a node for our blockchain in their programming language of choice. This document outlines how the system will work and 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 chain is called `Kerma`, but you should name your node something else.

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.

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 TUWEL as they become available.

Data Validation

Your client must disconnect from its peer in case it receives invalid data. Be rigorous about network data validation and do not accept malformed data. Log any incorrect data received to help us with debugging.

1 Cryptographic Primitives

Hash

We use SHA256 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 as described in below.

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 as 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.

Hexification

Hex strings must be in lower case.

2 Application Objects

Application objects are objects that must be stored by each node. These are content-addressed by the SHA256 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 should normalize your JSON and ensure it is in canonical JSON form. The examples in this document contain extra whitespace for readability, but these should not be sent over the network. The SHA256 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 `objectid`s are called `txid` and `blockid`, respectively.

Transactions

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

An input contains a pointer to a previous output and a signature. An input is a dictionary containing two keys: `outpoint` key (i.e., thee previous output) and a `sig` key. The `outpoint` key contains a dictionary of two keys: `txid` and `index`. The `txid` is the transaction identifier 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 *picaker*, the smallest denomination in `Kerma`. 1 ker = 10^{12} picaker. The `pubkey` is a public key of the

recipient of the money.

```
{
   "type":"transaction",
   "inputs":[
      {
         "outpoint":{
            "txid":"f71408bf847d7dd15824574a7cd4afdfaaa2866286910675cd3fc371507aa196",
            "index":0
         },
         "sig":"3869a9ea9e7ed926a7c8b30fb71f6ed151a132b03fd5dae764f015c98271000e7
                da322dbcfc97af7931c23c0fae060e102446ccff0f54ec00f9978f3a69a6f0f"
      }
   ],
   "outputs":[
      {
         "pubkey":"077a2683d776a71139fd4db4d00c16703ba0753fc8bdc4bd6fc56614e659cde3",
         "value":5100000000
      }
   ]
}
```

Transactions must satisfy the weak law of conservation: The sum of input values must be equal or exceed the sum of output values. Any remaining value can be collected as fees by the miner confirming the transaction.

If the transaction is a coinbase transaction, then it will not contain an `inputs` key but it will contain a `height` key with the block's height as the value. The coinbase transaction cannot be spent in the same block.

```
{
   "type":"transaction",
   "height":128,
   "outputs":[
      {
         "pubkey":"077a2683d776a71139fd4db4d00c16703ba0753fc8bdc4bd6fc56614e659cde3",
         "value":50000000000
      }
   ]
}
```

Blocks

This represents a block and has the type `block`. It contains the following keys: `txids`, which is a list of the transaction identifiers within the block, `nonce`, which is a 32-byte hexified value, `previd`, which is the block identifier of the previous block in the chain, `created`, which is an (integer) UNIX timestamp in seconds, and `T` which is a 32-byte hexadecimal integer and is the mining target. Optionally it can contain a miner field and a note field, which can be any

ASCII-printable strings up to 128 characters long each.

```
{
   "type":"block",
   "txids":[
      "740bcfb434c89abe57bb2bc80290cd5495e87ebf8cd0dadb076bc50453590104"
   ],
   "nonce":"a26d92800cf58e88a5ecf37156c031a4147c2128beeaf1cca2785c93242a4c8b",
   "previd":"0024839ec9632d382486ba7aac7e0bda3b4bda1d4bd79be9ae78e7e1e813ddd8",
   "created":1622825642,
   "T":"003a0000000000000000000000000000000000000000000000000000000000000",
   "miner":"****",
   "note":"A sample 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:

```
{
   "T":"00000002af0000000000000000000000000000000000000000000000000000",
   "created":1624219079,
   "miner":"dionyziz",
   "nonce":"0000000000000000000000000000000000000000000000000000002634878840",
   "note":"The Economist 2021-06-20: Crypto-miners are probably to blame for the graphics-chip shortage",
   "previd":null,
   "txids":[

   ],
   "type":"block"
}
```

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

The transaction identifiers (`txids`) in a block may contain one coinbase transaction. This transaction must be the first in `txids`. That transaction has no inputs but has a `height` key containing the height of the block the coinbase transaction included in. It has exactly one output which generates $50 \cdot 10^{12}$ new picaker 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 it can be less than this. 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.

All blocks must have a target `T` of:
`00000002af0000000000000000000000000000000000000000000000000000`.
The genesis `blockid` is:
`00000000a420b7cefa2b7730243316921ed59ffe836e111ca3801f82a4f5360e`.
Check this to ensure your implementation is performing correct JSON canonicalization.

3 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.

Hello

When you connect to another client, you must both send a `{ "type": "hello" }` message. The message must also contain a `version` key, which is always set to 0.8.0. If the version you receive differs from 0.8.x you must disconnect. The message can also contain an agent key, with a string description of the node software name and version the node is running.

You must exchange a hello message both ways before you exchange any other message. If a message is sent prior to the hello message, you must close the connection. Messages can be sent in any order after that.

```
{
   "type":"hello",
   "version":"0.8.0",
   "agent":"Kerma Core Client 0.8"
}
```

Error

Optionally, you can send objects with implementation-specific error messages to describe any exceptions encountered. An error object should be of type `error` and contain an `error` key with a string value that describes the error at hand.

```
{
   "type":"error",
   "error":"Unsupported message type received"
}
```

GetPeers

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

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

Peers

This message can be volunteered or sent in response to a `getpeers` message. It contains a peers key which is an array of peers. Every peer is a string in the form of <`host`>:<`port`>. The default port is 18018 but other ports are valid. Inclusion of the port in the string is mandatory. Examples for different types of hosts are provided below.

```
{
   "type":"peers",
   "peers":[
      "****.com:18018", /* dns */
      "138.197.191.170:18018", /* ipv4 */
      "[fe80::f03c:91ff:fe2c:5a79]:18018" /* ipv6 */
   ]
}
```

When sending a peers message, a node can include itself in the list if it is listening for connections.

GetObject

This message requests an object addressed by the given hash. It contains an `objectid` key which is the address of the object.

```
{
   "type":"getobject",
   "objectid":"0024839ec9632d382486ba7aac7e0bda3b4bda1d4bd79be9ae78e7e1e813ddd8"
}
```

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.

```
{
  "type": "ihaveobject",
  "objectid": "0024839ec9632d382486ba7aac7e0bda3b4bda1d4bd79be9ae78e7e1e813ddd8"
}
```

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

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.

```
{
  "type": "object",
  "object": {
    "type": "block",
    "txids": [
      "740bcfb434c89abe57bb2bc80290cd5495e87ebf8cd0dadb076bc50453590104"
    ],
    "nonce": "a26d92800cf58e88a5ecf37156c031a4147c2128beeaf1cca2785c93242a4c8b",
    "previd": "0024839ec9632d382486ba7aac7e0bda3b4bda1d4bd79be9ae78e7e1e813ddd8",
    "created": "1622825642",
    "T": "003a0000000000000000000000000000000000000000000000000000000000000000"
  }
}
```

GetMempool

Request the mempool of the peer with a message of type getmempool. There is no payload in this message. The peer responds with a `mempool` message.

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

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 transaction identifiers that the sending peer has in its mempool, i.e., are not yet confirmed. These are included in an array with the key `txids`.

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

GetChainTip

Request the current blockchain tip of the peer with a message of type `getchaintip`. There is no payload in this message.

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

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 block identifier of the current tip.

```
{
  "type": "chaintip",
  "blockid": "0024839ec9632d382486ba7aac7e0bda3b4bda1d4bd79be9ae78e7e1e813ddd8"
}
```

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