

## Kerma: 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 Python 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 chain is called Kerma, but you should name your node something else.

#### 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 TUWEL as they become available.

#### 1.4 Data Validation

Your client must disconnect from a peer it is connect 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 as described in below.

### 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 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.

### 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 keys that are specified, 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 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.  $1 \text{ ker} = 10^{12} \text{ picaker}$ . The `pubkey` is a public key of the

recipient of the money. The money carried by an output can be spend by its owner by using it as an input in a future transaction.

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.

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.

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

### Valid normal transaction

```
{
  "type": "transaction",
  "inputs": [
    {
      "outpoint": {
        "txid": "f71408bf847d7dd15824574a7cd4afdfaaa2866286910675cd3fc371507aa196",
        "index": 0
      },
      "sig": "3869a9ea9e7ed926a7c8b30fb71f6ed151a132b03fd5dae764f015c98271000e7da322dbcfc97af7931c23c0fae060e102446ccff0f54ec00f9978f3a69a6f0f"
    }
  ],
  "outputs": [
    {
      "pubkey": "077a2683d776a71139fd4db4d00c16703ba0753fc8bdc4bd6fc56614e659cde3",
      "value": 5100000000
    }
  ]
}
```

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 referenced transaction f7140...aa196 is invalid (INVALID\_ANCESTRY<sup>1</sup>).
- The output at index 0 in the referenced transaction f7140...aa196 holds less than 5100000000 picaker (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": 1,
  "outputs": [
    {
      "pubkey": "3f0bc71a375b574e4bda3ddf502fe1afd99aa020bf6049adfe525d9ad18ff33f",
      "value": 5000000000000000
    }
  ]
}
```

## 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 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.

---

<sup>1</sup>Since your node will never store invalid objects, upon receiving the transaction, it will attempt to fetch the referenced transaction from its peer, leaving the object verification pending. When the peer sends the (invalid) referenced transaction, your node will find out that it is invalid and will release the pending object verification by rejecting the transaction with an INVALID\_ANCESTRY error



are "executed".

All blocks must have a target  $T$  of:

00000000abc000.

The genesis blockid is:

0000000052a0e645eca917ae1c196e0d0a4fb756747f29ef52594d68484bb5e2.

Check this to ensure your implementation is performing correct JSON canonicalization.

## 4 Mempool

Every node should store a list of transactions that are not yet confirmed in a block. This is called the mempool. It should satisfy the following properties:

- It is possible to extend the current longest chain by a block containing precisely all transactions in the mempool.
- If a node receives a transaction  $T_x$  from the network, it should check whether it can be added to its mempool. This is possible if  $T_x$  spends only from unspent outputs that are also not spent by any transaction currently in the mempool.
- If the current longest chain changes, i.e., a new block is found or there is a chain reorganization, the current mempool  $M$  needs to be re-computed. Then, attempt to add the transactions of  $M$  to  $M'$ . This is done in the following way. The node sets the mempool state to be equal to the state  $s(B)$  of the latest block  $B$ . Then, the node tries to apply all transactions in the old mempool against this new state  $s(B)$ , one by one, in the order they appeared in its old mempool. If a transaction from the old mempool can be applied, it is added to the new mempool state. Otherwise, if it cannot be applied, it is thrown away. For instance, if block  $B$  contains a transaction that was in the old mempool, that transaction is discarded and does not make it into the new mempool, because it spends from outputs which are already spent. Similarly, if a transaction that was in the old mempool conflicts with the new state  $s(B)$ , it is thrown away. This means that if two double spending transactions appear in a block and in the old mempool, the transaction in the block takes precedence.

It is not required to further optimize the placement of transactions in the mempool.

In case of reorgs the current state of the chain is rolled back until the latest common ancestor between the current longest chain  $C'$  and the new longest chain  $C$ . Let us consider the following example:  $C'$  has blocks  $\{B_0, B'_1, B'_2\}$ , and  $C$  has blocks  $\{B_0, B_1, B_2, B_3\}$ . When a reorg takes place, this is what happens:

- the latest common ancestor  $B_0$  between  $C'$  and  $C$  is identified, and its state  $s(B_0)$  is the new state of the chain;
- all transactions in blocks  $\{B'_1, B'_2\}$  are undone and transactions in blocks  $\{B_1, B_2, B_3\}$  are applied against the state  $s(B_0)$ . If at any point a transaction cannot be applied, the whole block containing it is discarded;

- if  $C$  is the new valid longest chain, the chain with state  $s(B_3)$  and ending with  $B_3$  is adopted: the new mempool can be computed by trying to apply transactions in  $\{B'_1, B'_2\}$  against the new state  $s(B_3)$  (in the same order as they appear in  $\{B'_1, B'_2\}$ ) and, finally, by trying to apply transactions in the old mempool (in the same order as they appeared in the old mempool).

## 5 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.

### 5.1 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 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 handshake was 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": "Kerma-Core 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.

## 5.2 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"
}
```

## 5.3 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.<sup>3</sup>

---

<sup>3</sup>If we would not require this property, then the checks for syntactically valid ip addresses would be unneces-



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.

sary 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 be (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.

## 5.4 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 name `UNKNOWN_OBJECT`.

#### Valid getobject message

```
{
  "type": "getobject",
  "objectid": "0000000052a0e645eca917ae1c196e0d0a4fb756747f29ef52594d68484bb5e2"
}
```

## 5.5 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": "0000000052a0e645eca917ae1c196e0d0a4fb756747f29ef52594d68484bb5e2"
}
```

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

## 5.6 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": "00000000abc00000000000000000000000000000000000000000000000000000",
    "created": 1671062400,
    "miner": "Marabu",
    "nonce": "0000000000000000000000000000000000000000000000000000000021bea03ed",
    "note": "The New York Times 2022-12-13: Scientists Achieve Nuclear Fusion Breakthrough With Blast of 192 Lasers",
    "previd": null,
    "txids": [],
    "type": "block"
  }
}
```

## 5.7 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 with a `chaintip` message.

#### Valid GetChainTip message

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

## 5.8 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 block identifier of the

current tip.

### Valid ChainTip message

```
{
  "type": "chaintip",
  "blockid": "0000000052a0e645eca917ae1c196e0d0a4fb756747f29ef52594d68484bb5e2"
}
```

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.

## 5.9 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"
}
```

## 5.10 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 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`.

### Valid Mempool message

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

## 5.11 Error

If your node receives an invalid message or object, you must send a message with type error. An error message must contain a standardized key name. Possible values for name are:

- INVALID\_FORMAT:
  - A message was sent that could not be parsed as valid JSON.
  - A message is not a JSON object.
  - A required key is missing or an additional key is present.
  - An incorrect data type is encountered (e.g., a number was expected in the created key but a string was sent instead).
  - A key that should hold a fixed value contains a different value (e.g., different block target) or does not meet the required format (e.g., non-printable characters in key note).
  - A block was referenced at a position where a transaction was expected, or vice versa.
  - A non-coinbase transaction has no inputs.
  - A transaction has an output that holds a negative amount of coins.
  - A peer in a peers message is syntactically invalid.
- INVALID\_HANDSHAKE: A (semantic) error occurred related to the handshake, as specified in section Hello.
- INVALID\_TX\_CONSERVATION:
  - A transaction creates outputs holding more coins than the sum of the inputs.
  - A transaction itself double spends, i.e. it uses the same output multiple times as input.
- INVALID\_TX\_SIGNATURE: A signature is not valid.
- INVALID\_TX\_OUTPOINT:
  - A transaction in a block spends from the coinbase transaction in the same block.
  - A transaction in a block double spends or references a transaction output that is not in the current chain.
  - A transaction references an outpoint (*txid, index*) but there is no such outpoint in the transaction with id *txid* (i.e., *index* is too high).
- INVALID\_BLOCK\_POW: A block does not meet the required mining target.
- 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\_COINBASE:

- The coinbase transaction creates more coins than allowed.
  - A coinbase transaction was referenced but is not at the first position.
  - More than one coinbase transaction is referenced in a block.
  - The height of a coinbase transaction does not match the height of the block that references it.
- **INVALID\_GENESIS:** A block other than the genesis block with parent set to null was sent.
  - **UNKNOWN\_OBJECT:** An object was requested from a node that does not know about it. This does not indicate a faulty node.
  - **UNFINDABLE\_OBJECT:** The node could not verify an object because it failed to receive a dependent object after 5s. Note that when this error occurs, it does not necessarily mean that your communication partner is faulty - therefore you should not close the connection and remove your communication partner from your known nodes. Send this error also for every object whose verification you put on hold and must discard for now.
  - **INVALID\_ANCESTRY:** If verification of an ancestor of an object failed because it was found out to be invalid, send this error for every object whose verification you put on hold and can now be considered invalid.

The `msg` key of the error message contains a specific explanation explaining why this error occurred. You do not have to reply to an error message that is sent to you or perform any action. You can rely on TCP to detect if your communication partner has closed the connection because it considered your node as faulty. However, it might be interesting to log errors you receive because they might lead you to implementation errors in your node.

#### Valid error message

```
{
  "type": "error",
  "name": "INVALID_BLOCK_POW",
  "msg": "Block 000b7379afe3c40e6f19677d2e6e098c631eb495d13b24dc8998091f91c4be26 does not meet mining target"
}
```

## 6 Changelog

This section tracks changes made to this document.

- v1: Released 11.10.2023.
- v2: Released 18.10.2023. Introduced the following changes:
  - Clarified how to handle "peers" messages (please re-read section Peers)
  - Added new error case in the "peers" message for **INVALID\_FORMAT** also to the section Errors, and how to handle error messages

- Clarified that subsequent messages can be sent immediately after the hello message
- Clarified that connections should be kept alive
- Clarified edge cases when forgetting nodes
- Corrected type "mempool" in the "mempool" example message (was "getmempool")
- Added colorboxes around sections of interest
- (19.10.2023) Fixed regex in peers message
- v3: Released 25.10.2023. Introduced the following changes:
  - Fixed "miner" and "note" key restriction in section Blocks
  - Clarified that objects with additional keys that are not specified by this protocol description are invalid in the "Application Objects"
  - Minor restructuring of the "Blocks" section
  - Removed weird-looking object { "type": "hello" } in section Hello
  - Re-arranged and further clarified error names in section Errors
  - Grayed-out sections that are not required for the upcoming task.
  - Added an example on transaction validation.
- v4: Released 08.11.2023. Introduced the following changes:
  - Some typos corrected
  - Clarified the INVALID\_ANCESTRY and INVALID\_TX\_OUTPOINT error example in the "Transactions" Section
  - Updated grayed-out sections for Task 3