Cryptocurrencies (WS2025/26) Prof. Zeta Avarikioti, Prof. Matteo Maffei

Kerma - Task 1

1 General Notes

In your group of up to three students, you may use any programming language of your choice to implement the following task. We provide skeleton projects in Python and TypeScript (see `python-skeleton-for-task-1.tar.gz` and `typescript-skeleton-for-task-1.tar.gz` on TUWEL). You can use them to build upon. After the deadline, these skeleton projects will be updated and act as sample solutions for the previous tasks. These are guaranteed to pass all test cases for the previous tasks.

High level Overview

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

- listen for incoming connections on port 18018.
- perform the handshake.
- discover and connect to new peers.

Description

Start coding the implementation of your `Kerma`* node. First, start listening at port 18018 and exchange a `hello` message with any peer connecting to you. Then, you will extend your `Kerma` node so that it can exchange messages and perform peer discovery.

- Decide what programming language you will use (we recommend using either Python or Typescript, as these are the languages we provide help with and we release sample solutions).
- Find a cool name for your node.
- Implement the networking core of your node. Your submitted node must listen to TCP port 18018 for connections and must also be able to initiate TCP connections with other peers. Your node must be able to support connections to multiple nodes at the same time.
- Implement canonical JSON encoding for messages as per the format specified in the protocol.

*Kerma means “coin" in Greek.

- Implement message parsing, defragmentation, and canonical JSON encoding and decoding. On receiving data from a connected node, decode and parse it as a JSON string. If the received message is not a valid JSON or doesn't parse into one of the valid message types, send an `"error"` message with error name `INVALID_FORMAT` to the node. Note that a single message may get split across different packets, e.g. you may receive `"type":"ge` and `tpeers"` in separate messages. So you should defragment such JSON strings. Alternatively, a single packet could contain multiple messages (separated by `"\n"`) and your node should be able to separate them. Note that JSON strings you receive may not be in canonical form, but they are valid messages nevertheless.
- Implement the protocol handshake:
  - When you connect to a node or another node connects to you, send a `"hello"` message with the specified format.
  - If a connected node sends any other message prior to the hello message, you must send an `"error"` message with error name `INVALID_HANDSHAKE` to the node and then close the connection with that node. Note: Every message you send on the network must have a newline, i.e. `"\n"` at the end. Your node should use this to help parse and defragment valid messages. If you do not receive a `hello` message after 20s or receive a second `hello` message, you should send an error message (again with name `INVALID_HANDSHAKE`) and close the connection.
- Implement peer discovery bootstrapping by hard-coding some peers (for now, only hard-code the bootstrap node `128.130.122.73:18018`).
- Store a list of discovered peers locally. This list should survive reboots.
- Upon connection with any peer (initiated either by your node or the peer), send a `"getpeers"` message immediately after the `"hello"` message.
- On receiving a `"peers"` message from a connected peer, update your list of discovered peers.
- On receiving a `"getpeers"` message from a connected peer, send a `"peers"` message with your list of peers. Note that by specification of the protocol, a `"peers"` message must not contain more than 30 peers. If you have more than 30 peer stored, devise a policy on which 30 you send over the network.
- Devise a policy to decide which peers to connect to and how many to connect to. We suggest to connect to just a few nodes, and not all of them.
- If your node receives a valid message with different type than `hello`, `getpeers` or `peers`, you are not required to determine its validity in this task. You should not, however, close the connection if such a message is sent to you. For instance, if your node receives a `{"type":"getchaintip"}` message, then you should just ignore this message for now.
- Submit your implementation on TUWEL.

Important: Make sure that there are no bugs that crash your node and your node can run for a long time. 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.

- *The grader node "Grader" should be able to connect to your node. If you don't pass this test, Grader would not be able to grade the rest of the test cases. So make sure that you test this before you submit.*
- *Grader should receive a valid `hello` message on connecting.*
- *The hello message should be followed by a `getpeers` message.*
- *Grader should be able to disconnect, then connect to your node again.*
- *If Grader sends a `getpeers` message, it must receive a valid peers message.*
- *If Grader sends `{"type":ge`, waits for 0.1 second, then sends `tpeers"}`, your node should reply with a valid peers message.*
- *If Grader sends any message before sending `hello`, your node should send an error message with name set to `INVALID_HANDSHAKE` and then disconnect.*
- *If Grader sends an invalid message, your node should send an error message with the correct name. Some examples of invalid messages are:*
  1. `Wbgygvf7rgtyv7tfbgy{{{`
  2. `"type":"diufygeuybhv"`
  3. `"type":"hello"`
  4. `"type":"hello","version":"jd3.x"`
  5. `"type":"hello","version":"5.8.2"`
- *If grader sends a set of peers in a valid `peers` message, disconnects, reconnects and sends a `getpeers` message, it must receive a `peers` message containing (at least) the peers sent in the first message.*
- *Grader should be able to create two connections to your node simultaneously.*

1.1 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 public 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. [†]

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)
- 192.168.0.2:18018 (private ip)
- 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.

[†]If we would not require this property, then the checks for syntactically valid ip addresses would be unnecessary 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 are (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.

How to participate in the Kerma network

We suggest you to host your implementation yourself and let it actively participate in the Kerma network. There are multiple options to host your node:

- Use a static[‡] and public[§] IPv4 address. One way to get such an address is make use of free student offers available through (for instance) GitHub Student[¶], and to host your node on a VM in the cloud. Upload the static ip address to TUWEL.
- Host your node behind a dynamic[‖], public ip address and provide us with a domain entry that always points to this address.

How to Submit your Solutions We will grade your solution locally. Please upload your submission as a tar archive. When grading your solution, we will perform the following steps:

```
tar -xf <your submission file> -C <grading directory>
cd <grading directory>
docker-compose build
docker-compose up -d
# grader connects to localhost port 18018 and runs test cases
docker-compose down -v
```

When started in this way, there should be neither blocks (except the genesis block) nor transactions in the node's storage.

[‡] static means that it will not change over the duration of the course
[§] public means that it must be accessible from everywhere
[¶] See https://education.github.com/pack for more information
[‖] dynamic means that it may change over the duration of the course

If you use the skeleton templates, you can use the provided Makefile targets to build and check your solution. Note however, that this will only check if the container can be built, started and a connection can be established to localhost:18018. We highly recommend that you write your own test cases.

The deadline for this task is 31st October, 11.59pm. We will not accept any submissions after that. Each group member has to submit the solution individually. Plagiarism is unacceptable. If you are caught handing in someone else's code, you will receive zero points.