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

Bitmunt - Task 1

General Notes

A GitLab repository has been created for each group of students. Please use the same repository for all tasks. We provide starting code in Go and TypeScript (See `assignment/skeleton/`). You can use this as a starting point to build upon.

Programming project: code of conduct

Our expectation is that you make this project in a group (with assigned group members). As a group you are allowed to collaborate in whatever way you think is most effective (e.g. do pair-programming, split up the work, etc.) Note however that we do expect ALL members of the group to be able to understand and explain the full codebase that you submit as a group!

You are allowed to discuss the tasks and solution strategies with your fellow classmates, but directly sharing solution code in whole or in part between groups is forbidden.

We will check code for plagiarism. Plagiarism is unacceptable and will lead to disciplinary actions as per university policy.

Use of GenAI tools to assist with coding is allowed within limits using the following guidelines:

- If your group makes frequent use of a GenAI tool, that tool should be briefly mentioned in your `submission.txt` notes.
- If you submit solution code where a GenAI tool has generated significant lines of code (e.g. a function body or a critical piece of logic) include a comment in the source code acknowledging use of GenAI, possibly including relevant prompts.
- Be aware of the university policy on GenAI. You remain personally responsible for everything you submit, and you cannot shift responsibility or blame for mistakes in your solution to third-party sources or tools (GenAI or otherwise).

Good luck with the project!

Submission Instructions

For each task, you must create a Git tag in your repository corresponding to the version of your code that you want to be graded. Follow the naming conventions below:

- For submission, use the format `taskX-submission`, where `X` is the task number (1 to 6). Only **correctly tagged** submissions before the hard deadline will be graded.

- For ungraded feedback, use the format `taskX-ungraded`. You need to submit it before the soft deadline.

There should be neither blocks (except the genesis block) nor transactions in the storage. More detailed instructions are available in the GitLab repository. This submission process will be used for all future tasks.

Grading Criteria

We will have a total of 6 tasks, with Task 6 being optional. Each task scores 2 points, making the maximum possible score 10 points. You can get 1 bonus point if you finish Task 6, allowing for a total score of up to 11 points.

High level Overview

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

- listen for incoming connections on port 18018 (already included in the skeleton).
- perform the handshake (already included in the skeleton).
- discover and connect to new peers.

1 Networking

Skeleton code provided

We have included starting code in the skeleton project, along with a solution for this section. Feel free to modify the code or create your own solution if you want.

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

1. Decide what programming language you will use (we recommend using either Go or Typescript, as these are the languages we provide help with).
2. Find a cool name for your node.
3. 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.

*Out of curiosity, the word "munt" means "coin" in Dutch.

4. Implement canonical JSON encoding for messages as per the format specified in the protocol.

5. 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.

6. 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.

2 Peer Discovery

1. Implement peer discovery bootstrapping by hard-coding some peers (for now, only hard-code the bootstrap node `172.22.29.47:18018`.

2. Store a list of discovered peers locally. This list should survive reboots.

3. Upon connection with any peer (initiated either by your node or the peer), send a `"getpeers"` message immediately after the `"hello"` message.

4. On receiving a `"peers"` message from a connected peer, update your list of discovered peers.

5. 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.

6. Devise a policy to decide which peers to connect to and how many to connect to. We suggest connecting to just a few nodes and not all of them.

7. If your node receives a valid message with a 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.

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)
- nodotindomain:1234 (no dot in domain)
- bitmuntnode.net (no port given)

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.

You can use the command "`nc -vvv ip_addr port`" to connect to a node at IP address `ip_addr` and port `port`, and also send and receive messages. Use this to test your node extensively. You can also use "`nc -vvv -l -p port`" to listen for connections on the port `port`.

3 Test Cases

Important: make sure your node is running all the time. 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.

[†]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.

- 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 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.

How to participate in the Bitmunt network

We expect you to host your implementation yourself and let it actively participate in the Bitmunt network. We have created a VM for each group on our departmental private cloud. You can find detailed instructions of how to connect to it in the GitLab repository.

The **soft deadline** for this task is **20th March, 2025, 11.59pm**.
The **hard deadline** for this task is **3rd 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.