



CHAIR OF NETWORK ARCHITECTURES AND SERVICES

TECHNICAL UNIVERSITY OF MUNICH

Project Documentation

Author: Thua-Duc Nguyen & Duc-Trung Nguyen



1 Architecture

The final architecture of our project is similar to the one we presented in our midterm project. As the specification requires, the gossip module runs as two independent protocols: one API protocol and one P2P protocol. These two protocols share some data to fulfill the functionality of the module.

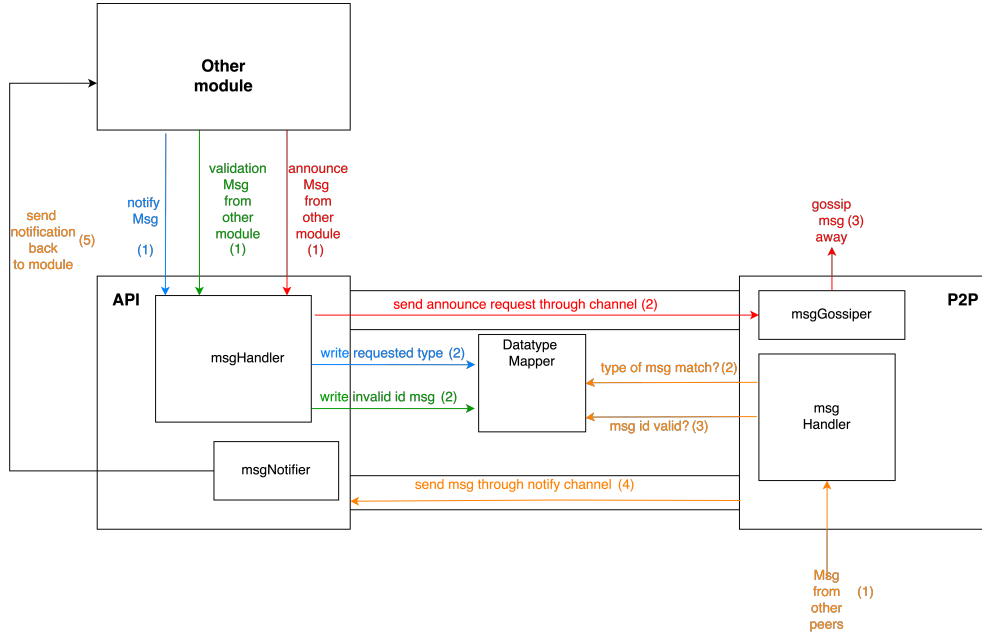


Figure 1: Structure of the gossip module

1.1 Shared data between API and P2P

AnnounceMsgChan

We have an announce Go channel [1] (marked in red in Figure 1) shared between the two protocols. Whenever the API protocol receives an announce message from another module, it processes the message immediately and sends it to the P2P protocol through this channel. The P2P protocol has an announce message handler running on a goroutine that always listens to this channel. When it receives an announce request, it will gossip this message away.

DatatypeMapper

We also need a DatatypeMapper (marked in black in Figure 1) shared between two protocols. Whenever API receives a notify message, it will write the message type that is valid into the mapper and hence should be propagated further. DatatypeMapper also contain a list of invalid message id, which is sent by other module through Validation endpoint. This list

is being used by P2P to check for the validity of a message before forwarding it through `NotiyiMsgChan`

This datatype mapper will, of course, own a mutex that guarantees no race condition between the two protocols.

NotiyiMsgChan

Thanks to the `DatatypeMapper`, the P2P protocol can recognize which kind of message it should propagate. When it receives a new message, it will check if this message type was requested by any module by reading the datatype mapper and if the message is valid. If that is the case, it sends this message through `NotiyiMsgChan` (marked in orange in Figure 1). API protocol also has a running goroutine that constantly listens to this channel. It can get those messages from P2P and send corresponding notification messages to the module requesting them.

1.2 Security

We changed our security mechanism according to our midterm feedback. We integrated POW in every gossip message. The hardness of this challenge is defined in the config file. Depending on the security requirement of the project, we can adjust the hardness, also known as the number of leading zeros, accordingly.

2 Software Documentation

2.1 Important Components and configurations

- **Node:** The core component responsible for the peer's logic and communication. The node handles traffic from other peers (internode communication) and interactions within the same peer (intranode communication) across the API and P2P modules. It can be customized using the `config.ini` file to control behavior and parameters.
- **Bootstrapping Service:** A central server responsible for providing initial knowledge to newly joined peers. This service distributes a partial view of the network and designates a set of trusted `SeedNodes`. The number of `SeedNodes` can be configured via the `enum/bootstrapper_config.go` file, ensuring flexible network deployment.

2.2 How to configure and boot up more nodes

By default, we have set up the `docker-compose.yml` file to boot up the bootstrapping service along with three independent nodes. Below is an example of the `docker-compose.yml` file configuration.

For more details, take a look at the `docker-compose.yml`

Ports: For each node, we map local ports to container ports. For example:

- "9002:9000" maps traffic from `localhost:9002` to `node:9000`.
- "9003:9001" maps traffic from `localhost:9003` to `node:9001`.

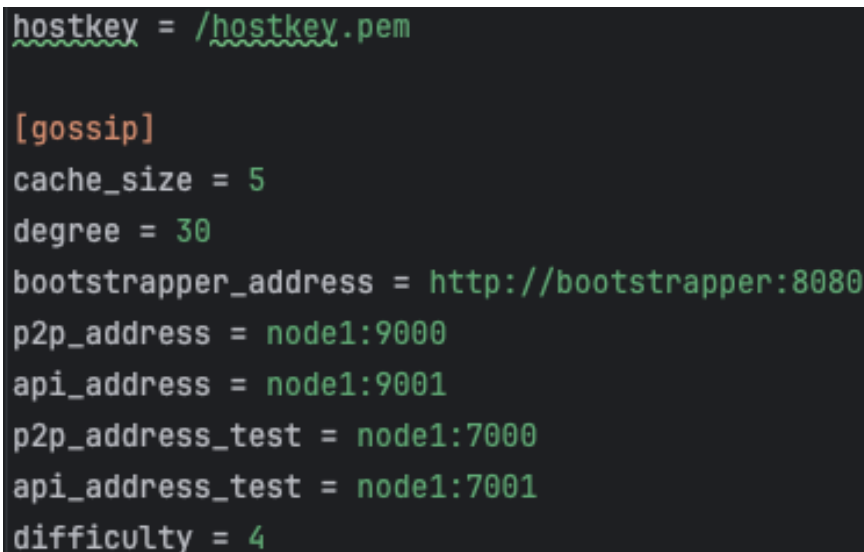
It's crucial to set these ports correctly to enable the client code to interact with the respective nodes.

Container Names: Each container has a unique name, such as `node`, `node1`, and `node2`, ensuring no conflicts.

Networks: All containers must be in the same network (e.g., `my-network`) to correctly enable communication.

Volumes: Each node's configuration is mounted from the `docker-container-config` folder to the container, e.g., `./docker-container-config/config-node:/root/configs`. Ensure you provide the correct path to the configuration file.

Additionally, the `config.ini` file for each node should have a unique `p2p_address` to prevent conflicts, as shown in the sample `config.ini`:



```
hostkey = /hostkey.pem

[gossip]
cache_size = 5
degree = 30
bootstrapper_address = http://bootstrapper:8080
p2p_address = node1:9000
api_address = node1:9001
p2p_address_test = node1:7000
api_address_test = node1:7001
difficulty = 4
```

Figure 2: Sample configuration file

For example: `node1:9000` and `node1:9001` for `node1`. And `node2:9000` and `node2:9001` for `node2`.

Boot up more nodes:

- **Step 1:** add new config.ini file to `./docker-container-config` Make sure config.ini resides in a uniquely named folder according to the node you want to inject the config.int to. For example: node10-config should be injected into node10
- **Step 2:** adjust `docker-compose.yml` accordingly. Add new entry to `services:` section. Make sure the path to config.ini is set correctly.

2.3 Additional Logic To P2P Protocol

Peer Join:

When a peer joins the network, it needs to notify other peers to become part of the distributed system. This is done by sending a `PeerJoinAnnounce` message to all known peers. The joining peer sends out its address, which is then propagated through the network using the gossip protocol. This ensures all active peers update their local peer list with the new node.

Key steps:

- The new peer sends a `PeerJoinAnnounce` message.
- Existing peers receive this message and add the new peer to their local list.
- The message propagates via the gossip protocol to ensure network-wide awareness.

Peer Leave:

When a peer leaves the network, it announces its departure by sending a `PeerLeaveAnnounce` message. This ensures the departing peer is removed from the other peers' lists. The peer sends this message to its known peers, who propagate the information using gossip to ensure all peers update their lists.

Key steps:

- The leaving peer sends a `PeerLeaveAnnounce` message.
- Existing peers remove the peer from their local peer lists.
- The message is gossiped across the network, updating all peers.

Request Peer List:

A peer can request the peer list from another node to maintain an up-to-date view of the network. The requesting peer sends a `PeerListRequest` message, prompting the target peer to respond with its list of known peers. This ensures nodes remain aware of each other, particularly after membership changes.

Key steps:

- The requesting peer sends a `PeerListRequest` message.
- The target peer responds with its current list of peers.

Response Peer List:

When receiving a `PeerListRequest`, the peer responds with a `PeerListResponse` message, containing its current list of known peers. This allows the requesting peer to update its peer list accordingly and keeps the network synchronized.

Key steps:

- The peer receiving the request sends a `PeerListResponse` with its peer list.
- The requesting peer updates its list based on the response.

3 How to install and run

To install and run the project, follow the steps below:

1. `git clone` the project repository to your local machine:

```
git clone <repository-url>
```

2. Install Docker Engine on your system if it is not already installed. Please follow the instructions on Docker's official site.
3. Make the build script executable by running (make sure you are in the project directory):

```
chmod +x build.sh
```

4. Run the build script that includes all necessary commands to pull build docker images and run them up:

```
./build.sh
```

After the step above, you should be able to observe the behavior of the containers

The steps underneath are optional if you want to use our Go client. You can also interact with the running Nodes by using the Python client provided in the course.

5. (Optional) Install Go to ensure the Go environment is properly set up.
6. (Optional) Run Go Client to interact with node. For more details, take a look at `Readme.md`.

3.1 Known Issues

- **Node Leave:** Currently, the node leave mechanism is not functioning correctly. The node fails to send a gossip message before shutting down completely. The reason may be due to using TCP instead of UDP for sending messages, which leads to delays and failed messages when a peer tries to read PeerLeaveAnnounce from the closed TCP connection.
- **Bootstrapping Inefficiencies:** The bootstrapping strategy is not optimized. Peer discovery can be slow and inefficient, especially in larger networks.
- **Seed Node Refresh:** There is no mechanism for peers to refresh their view of seed nodes if one of the seed nodes goes offline. This can lead to outdated peer lists and network instability.
- **Seed Node Assignment:** Any peer who joins the network first becomes a seed node. If a seed node goes offline, the next joining node becomes a seed node, which can yield issues with network stability.

4 Future Work

- **Improve Node Leave Mechanism:** Implement a more robust mechanism to handle node departure. Currently, nodes fail to send the gossip message before shutting down completely. Switching from TCP to UDP for leave notifications would allow faster, more reliable communication before the node exits.
- **Optimize Bootstrapping Strategy:** The current bootstrapping strategy has inefficiencies and should be optimized to prevent issues like inconsistent peer discovery and poor handling of network partitioning.
- **Seed Node Refresh Mechanism:** Develop a dynamic seed node refresh mechanism to allow peers to refresh their view of seed nodes. This is especially necessary when one of the seed nodes goes offline, so the network can remain stable without depending on unreachable nodes.
- **Redesign Seed Node Assignment:** The current logic assigns the first peer as a seed node, and if a seed node goes offline, the next joining node becomes a seed node. This can lead to instability. The seed node assignment mechanism should be more robust, ensuring stable and capable nodes are assigned as seed nodes.
- **Peer View Synchronization:** Work on synchronizing peer lists more efficiently across nodes, particularly in the event of network partitions or if a large number of nodes join

simultaneously.

- **Security Enhancements:** Explore more advanced security protocols for P2P communication beyond proof of work, such as mutual authentication or encryption, to enhance the integrity and confidentiality of gossip messages.

5 Workload Distribution

Our previous distribution of tasks has worked very well, ensuring smooth collaboration and efficient development. The responsibilities were divided as follows:

- **Thua Duc Nguyen:** in charge of implementing the API server, designing the overall architecture of the peer, and developing security measures. Additionally, Duc implemented the Go client to facilitate testing and provided scripts that streamlined Docker container building and execution processes.
- **Duc Trung Nguyen:** responsible for designing and implementing the core P2P protocols, including the bootstrapping mechanism. Additionally, Trung handled dockerizing the application, allowing seamless config file injection into containers and ensuring flexible deployment strategies.

This clear division of tasks allowed us to effectively meet our deadlines and achieve the project goals.

Bibliography

- [1] “Go channel”. In: (). Accessed: 2024-09-08. URL: <https://go.dev/tour/concurrency/2>.