CHAIR OF NETWORK ARCHITECTURES AND SERVICES

TECHNICAL UNIVERSITY OF MUNICH

Miterm Report

Author: **Thua-Duc Nguyen & Duc-Trung Nguyen**

1 Change of assumptions

2 Architecture of your module

The module `gossip` has been developed using the Golang programming language. It relies heavily on Go features like goroutines, Go channels, and multiple Go libraries.

2.1 The whole picture

As the specification requires, the `gossip` module runs as two independent protocols: one API protocol and one P2P protocol. However, these two protocols share some data to fulfill the functionality of the module.

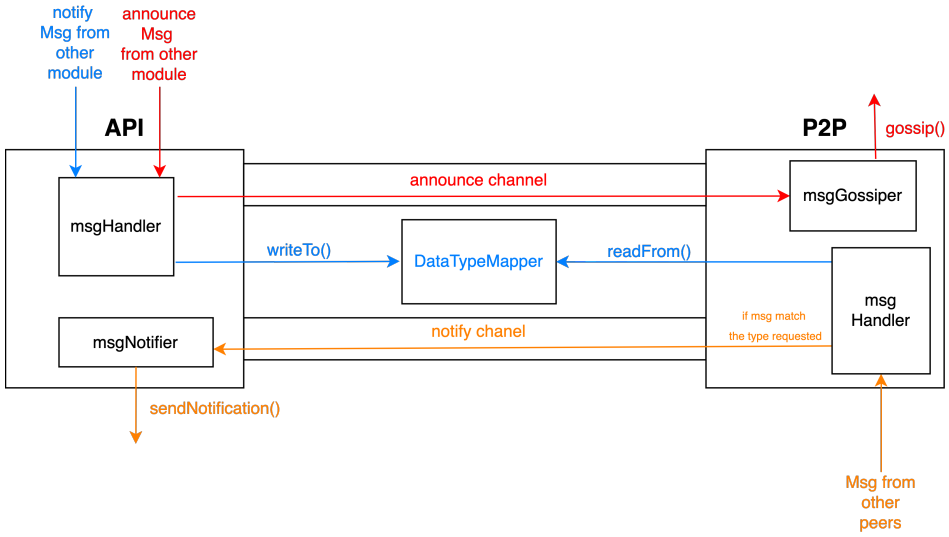

Figure 1: Structure of the gossip module

1. Announce messages Go channel

To make the announce functionality work, we need an **announce Go channel**[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.

```
1 announceMsgChan := make(chan enum.AnnounceMsg)
```

[1]Marked in red in Figure 1

2. Datatype mapper

To make the notify functionality work, we need a **datatype mapper**[2] 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. This datatype mapper will, of course, own a mutex that guarantees there is no race condition between the two protocols.

```
type DatatypeMapper struct {
    mutex sync.RWMutex
    data  map[net.Addr]map[enum.Datatype]bool
}
```

3. Notify messages Go channel

Thanks to the datatype mapper, 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. If that is the case, it sends this message through **notify message Go channel**[3]. 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.

2.2 API

The API is designed to facilitate a Gossip-based protocol in a distributed system, leveraging Go's robust features for concurrency and networking. At its core, the Server listens on a specified TCP address for incoming connections, using Go's net package to manage network communications. Once a connection is accepted, the Server hands it off to a Handler, which processes messages according to their type.

The Handler utilizes a custom logger for monitoring and error reporting, enhancing the system's reliability and debuggability. It reads incoming messages, verifies their size and type using the bytes and encoding/binary packages, and then routes them to the appropriate handler functions. These functions handle specific message types such as announcements or notifications, updating the datatypeMapper, or sending messages to the announceMsgChan channel as necessary.

This seamless interaction between the Server and Handler ensures the system can efficiently process and route messages, maintaining data integrity and system state across distributed nodes. The use of Go's concurrency primitives, like goroutines and channels, allows the API to handle multiple connections simultaneously, making it scalable and robust for real-time, distributed communication.

[2]Marked in blue in Figure 1
[3]Marked in orange in Figure 1

2.3 P2P

Bootstrapping strategy:

Bootstrapping service is one of the important components of a P2P network that helps newly joined Node to get initial knowledge on current active peers in the network. In our current implementation, we use a static bootstrapping method to ensure that new nodes can join the network and connect to existing peers. The bootstrapping process involves the following steps:

1. **Registration**: When a new node starts, it registers itself with the bootstrapper server. The server maintains a list of all registered peers.
2. **Fetching Initial Peers**: After registration, the new node fetches an initial list of peers from the bootstrapper server. This list is used to establish initial connections and begin participating in the gossip protocol.

Certainly, here is the shortened version:

Gossip Node:

The `GossipNode` implementation supports key functionalities for managing peer-to-peer communication. When a node starts, it registers with the bootstrapper server and fetches an initial list of peers to establish connections. When a node leaves, it announces its departure to its known peers, updating their peer lists.

The node disseminates information using a gossip protocol, spreading data such as new peers joining or leaving, and application messages to a random subset of peers. It processes and forwards incoming gossip messages to ensure widespread distribution. Nodes periodically exchange peer lists to maintain up-to-date network knowledge, dynamically adding new peers and removing inactive ones as necessary.

A message handler is integrated into the `GossipNode`, which picks up the message type from the gossip message and handles it accordingly. This handler ensures that different types of messages (such as announcements, departures, data spreads, and peer list requests) are processed correctly and efficiently within the network.

Properties

The `GossipNode` has several properties. The fanout, set to n=2, determines the number of peers a node gossips to during each round. The gossip interval, set to m=5 seconds, specifies the frequency of gossip messages. The message cache prevents redundant processing by storing recently seen messages using a map data structure. The peers list, updated dynamically, maintains known peers for communication. The bootstrap URL is used for initial registration and fetching peers when nodes join the network.

Gossip Message

For efficient communication between peer, we define a structrue for Gossip Message(figure below):

- **MESSAGE_TYPE**: Predefined type of gossip message.
- **TTL (Time to Live)**: Get from API call, reduce by each hop.
- **RESERVED**:
- **PAYLOAD**: Data get from API call

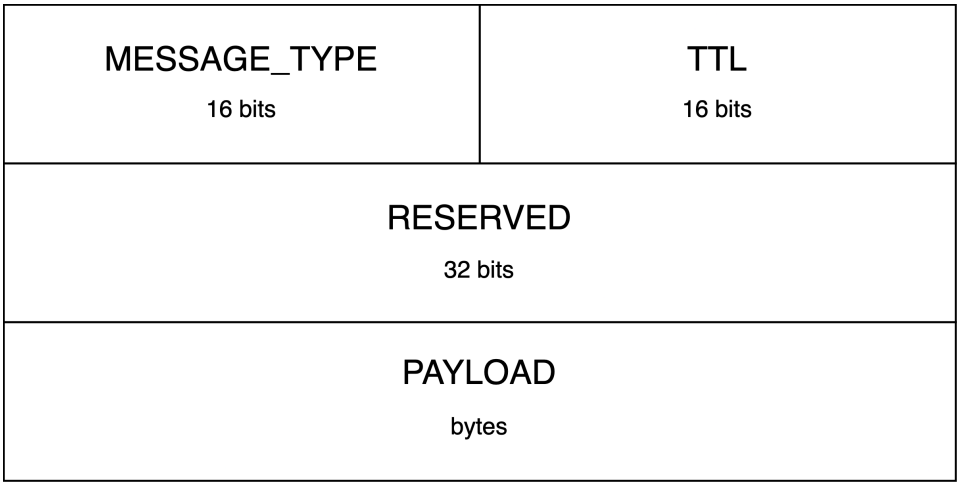

Figure 2: Gossip Message

- **GossipMessageType**:
  - **PEER_ANNOUNCE**: Message announcing a new peer.
  - **PEER_LEAVE**: Message announcing a peer leaving the network.
  - **DATA_SPREAD**: Message containing data to be spread across the network.
  - **PEER_LIST_REQUEST**: Message requesting the list of known peers.
  - **PEER_LIST_RESPONSE**: Message containing the list of known peers.

3 Security Measures

3.1 Bootstrapper Proof-Of-Work (POW):

For security of Bootstrapper, we implement POW mechanism for the registering process. To avoid sybil attack, we define target difficulty, which is the number of leading zeros of a

hash that a peer should calculate in order to register successfully. Details of the process are belows:

BOOTSTRAP INIT

When a new peer opens a connection to Bootrapping Server, the server sends an INIT message to the peer. This message contains a challenge and its target difficulty which this peer needs to use to calculate the hash.

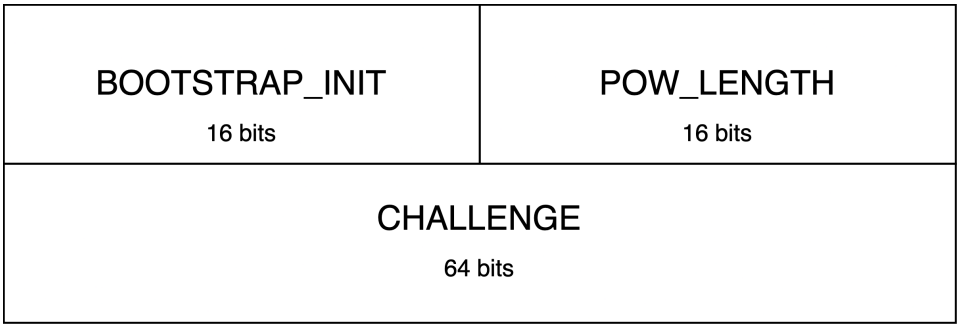

Figure 3: Bootstrap Init Message

BOOTSTRAP REGISTER

The peer calulate the SHA256 hash of the concatination of nonce (random number) and the challenge. If the resulting hash has the first n bits set to zeros (wheras n = target difficulty), then the peer can register by sending the BOOTSTRAP REGISTER message. If not, the nonce has to be changed to another random value and retried until one is found which gives one of the required SHA256 values. Note that SHA256 is considered a pseudo-random function. This means that there is no other possible way to find such values apart from randomly retrying them.

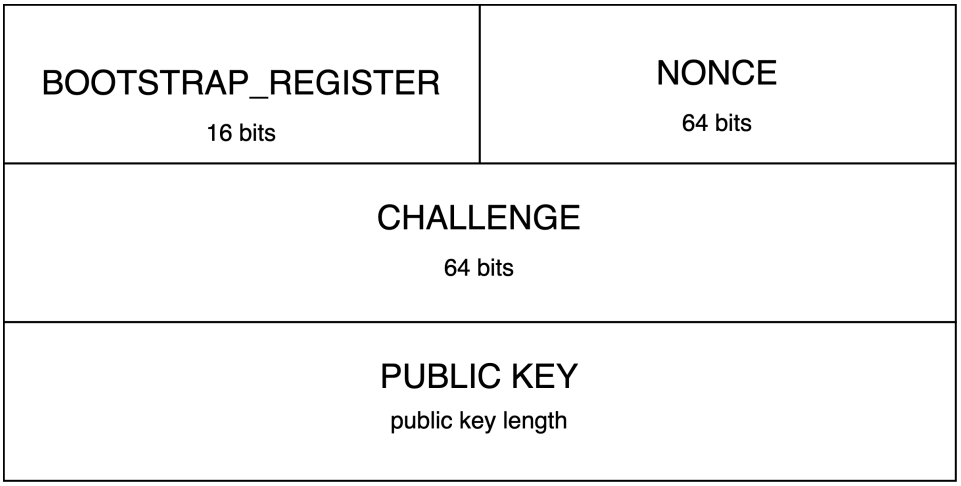

Figure 4: Bootstrap Register Message

BOOTSTRAP SUCCESS

If registeration is successful, meta data of the new peer got saved in local memory of the bootstrapping service and a BOOTSTRAP SUCCESS message is returned to the peer. The message also contains initial peer list.

Figure 5: Bootstrap Success Message

BOOTSTRAP FAILURE

If registeration is unsuccesful, meta data of the new peer got saved in local memory of the bootstrapping service and a BOOTSTRAP FAILURE message is returned to the peer with error message.

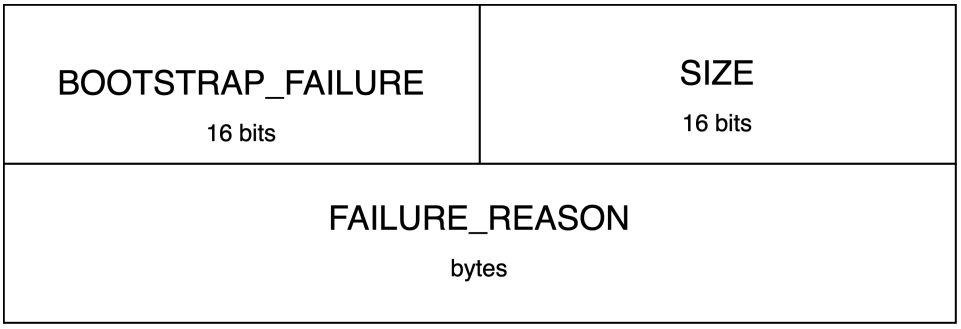

Figure 6: Bootstrap Failure Message

4 Specification of the peer-to-peer protocol that will be implemented

4.1 Message Types

500-503: API SPECIFICATION

504: BOOTSTRAP INIT

505: BOOTSTRAP REGISTER

506: BOOTSTRAP SUCCESS

507: BOOTSTRAP FAILURE

508: PLACE_HOLDER

509: PLACE_HOLDER

510: DATA_SPREAD

511: PEER_ANNOUNCE

512: PEER_LEAVE

513: PEER_LIST_REQUEST

514: PEER_LIST_REQUEST

5 Future Work

In the upcoming weeks, there are still a few missing functionalities that need to be implemented:

- **Gossip Message Handler in GossipNode**: Implement a handler that can pick up the message type from a gossip message and handle it accordingly.

- **Peer List Sharing Between Nodes**: Develop functionality for periodic peer list exchange to maintain up-to-date network knowledge.
- **Bootstrap Server to Ping Nodes**: Add a feature in the bootstrap server to ping nodes periodically to check for their vitality.
- **Node Join/Leave Announcements**: Ensure that nodes announce their presence when joining and notify the network when they leave.
- **Further Security Measures**: Implement additional security measures to enhance the robustness and security of the P2P network.

In the upcoming weeks, there are still a few missing functionalities

6 Workload distributed

The initial distribution works out very well as we are responsible for each part as following: Duc TrungTrung (design and implement P2P Protocols, including bootstrapping mechanism), Thua Duc (implement API server, design overall architecture of the Peer, and security measurement). We will continue with that strategy.

7 Effort spent for the project

Throughout the project, we maintained regular communication and coordination to ensure that each team member was contributing equally. Weekly meetings were held to discuss progress, allocate tasks, and address any challenges faced. This collaborative approach allowed us to leverage each member's strengths and work effectively as a team.

The project was a result of the collective effort of all team members, with each individual contributing equally to its success.