Final Report – GOSSIP

Daniel Fomin, Deniz Etkar

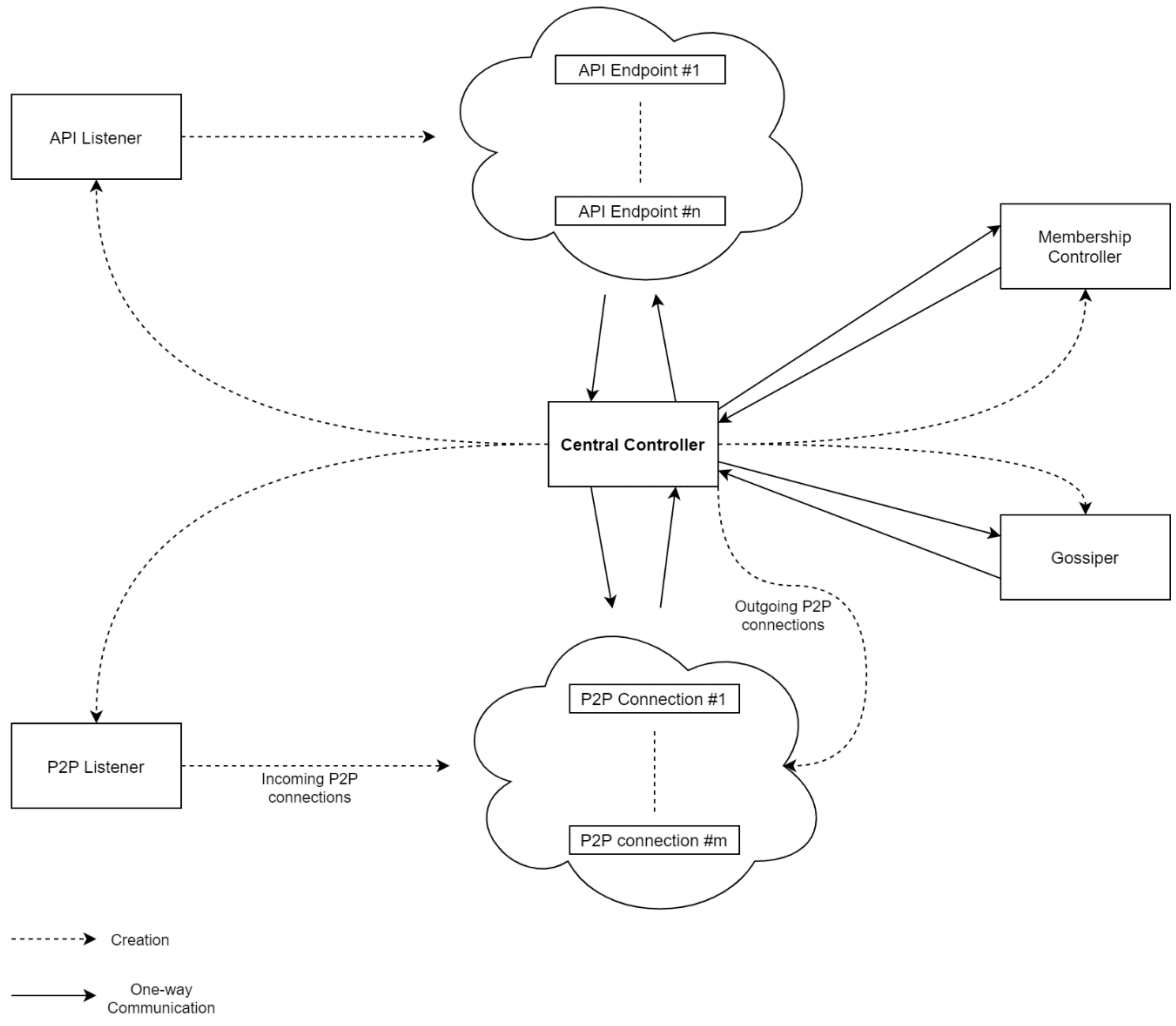
Sep 12, 2020

Contents

1. Module Architecture

The Gossip module is a single process that has many concurrent goroutines running in it. Some of these goroutines are special and are classified as a **submodule** in our documentation. There are 7 different types of submodule in our Gossip module, all of which are designed to run concurrently. Namely,

1) Central controller,
2) Membership controller,
3) Gossiper,
4) API listener,
5) P2P listener,
6) API endpoint,
7) P2P endpoint.



The main thread of our module runs the **Central controller** which governs the creation, state maintenance, closure and inter-message passing of every other submodule. All submodules, except for the 2 endpoints, are meant to have only 1 unique running instance in the Gossip module process. The basic working principle is based on **asynchronous-IO** whereby every interaction between submodules is an **event**. Every event is handled by an event handler function. There are separate input and output message queues of each submodule except for the 2 listeners (they only have output message queue).

Any design choice that might require mutex/semaphore locking mechanisms is avoided. Instead, as per the design principles suggested by the Go language, our submodules "**Do not communicate by sharing memory; instead, share memory by communicating.**" [1] . That is why we utilized input and output message queues, namely **channels**.

The **Membership controller** is responsible for maintaining a random list of active peers from the Gossip network. In order to maintain this list, the Membership controller of a peer communicates with its counterpart in other peers. This communication is classified as **peer to peer communication** and it involves 6 types of messages:

1) Membership push (one type for outgoing, one type for incoming),
2) Membership pull request (one type for outgoing, one type for incoming),
3) Membership pull reply (one type for outgoing, one type for incoming).

In addition to the p2p communication, it also periodically **probes** all peers that are in its sample list.

The **Gossiper** is responsible for spreading the gossip items that are either announced by the API clients or by other remote peers (gossip modules) as well as notifying API clients that are subscribed to the gossip items with a certain data type as well as taking account of the validation replies coming from API clients. In order to achieve rumor spreading among peers, the Gossiper of a peer communicates with its counterpart in other peers. This communication is also classified as **peer to peer communication** and it also involves 6 types of messages:

1) Gossip push (one type for outgoing, one type for incoming),
2) Gossip pull request (one type for outgoing, one type for incoming),
3) Gossip pull reply (one type for outgoing, one type for incoming).

[1] https://blog.golang.org/codelab-share

For the remaining 2 responsibilities, the Gossiper communicates with API clients which is classified as **api communication** and it involves all of the API messages defined in the **project specifications document**.

The **api listener** is responsible for accepting incoming api client connections, creating an api endpoint for each of them and notify the Central controller by sending the created api endpoint.

The **p2p listener** is responsible for accepting incoming p2p connections, creating a p2p endpoint for each of them and notify the Central controller by sending the created p2p endpoint.

An **api endpoint** is responsible for reading from its api connection and delivering incoming messages to the Central controller as well as writing the messages coming from the Central controller into its api connection. It has two running goroutines: one for reading from the connection and one for writing to the connection. The purpose of creating such a submodule is to **relieve** the Central controller of these IO bound tasks (reading and writing).

A **p2p endpoint** is responsible for reading from its p2p connection and delivering incoming messages to the Central controller as well as writing the messages coming from the Central controller into its p2p connection. It has two running goroutines: one for reading from the connection and one for writing to the connection. The purpose of creating such a submodule is to **relieve** the Central controller of these IO bound tasks (reading and writing).

Every peer to peer connection is a secure connection that implements the custom protocol we designed called "**Gossip-6 LAYER4 Secure Communication Protocol**". Further details of the protocol can be found in **comms_protocol.pdf** file in **docs** folder of our project. Essentially what this secure communication implementation, in short **securecomm**, allows us to do is to avoid using a certificate authority (**CA**). This is achieved by utilizing proof of work (**PoW**) in both the client side and the server side for each connection attempt. To avoid using a certificate authority means to avoid centralization. Any form of centralization in a software architecture presents a singular point of failure which would be the center of attention for attackers. In this way, our secure communication alleviates a singular point of failure in our software architecture and provides both the client and the server with a method of authentication.

2. Software Documentation

i. Dependencies

There are no runtime dependencies for the Gossip module. The module simply consists of a single process with a single executable file which is started from a terminal. However, there are compile time dependencies all of which are defined in the **go.mod** file at the root directory of our project with their version numbers and dates.

ii. How to build

Our project uses Go modules provided by the Go language itself (currently with version 1.14). So, the building process is made as simple as the following steps:

1) Download and install any Go language with version 1.14 from https://golang.org/doc/install,
2) Download the source code of our gossip module project,
3) Open a terminal into the root directory of the project,
4) Run **go build** command.

In the root directory, an executable file with name **gossip** will be created.

iii. How to run

1) Move the gossip executable file created in the root directory to a permanent directory,
2) Change current directory to the path of the gossip executable file,
3) Run **gossip -config_path=%path_of_config%** where **%path_of_config%** must be replaced with the valid path of a "**.ini**" formatted configuration file (see **config.ini** file in the config folder for an example).

Some important things to note before running the gossip module are as follows:

- Make sure to use your own RSA 4096bit private and public keys and not the sample keys that are given as part of the project source code,
- In the configuration file, double check the paths **hostkey** and **pubkey** which are supposed to respectively point to "**.pem**" formatted RSA 4096bit private key and "**.pem**" formatted RSA 4096bit public key,

- In the configuration file, check the path **trusted_identities_path** which is supposed to point to a folder with files such that each file name is the **identity** (as defined by the **project specifications document**) of a trusted peer and that the identity of the **bootstrapper** peer exists in there.

iv. Known issues

1) There is no mechanism for neither the api listener nor the p2p listener to reject an incoming connection request which might be necessary in certain cases such as connection capacity exhaustion and denial of service attack (**DOS**).
2) When one of the following submodules crashes, the Central controller hence the process is also forced to crash as well: The Membership controller, the Gossiper, the api listener, the p2p listener submodules.
3) All of the logging inside the gossip module have the same level of verbosity. Therefore, there is no choice on the level of verbosity for logging and every log is written to the **stdout**.
4) The gossip module has not been extensively tested to prove maturity, yet. This means there are minimal guarantees on the correct functioning and security of this software.

3. Future Work

Obviously, the current known issues must be addressed in the near future. Also, it is currently not possible for a peer to join a Gossip network without sharing its identity (SHA256 hash of its 4096bit RSA public key, as defined by the **project specifications document**) which makes joining a hassle. So, out-of-band identity sharing mechanism can be automated by using hard enough proof of work or by using a distributed ledger technology such as blockchain etc. Another improvement would be to give the flexibility of using different cryptographic authentication algorithms other than RSA 4096bit keys with SHA256 hash for identities. The development was done in VS Code Windows 10 environment. Since the project uses Go language, the project would certainly compile in any platform for which Go language has installation support. Nonetheless, it is still necessary to compile the gossip module and test it in various platforms such as Linux and MacOS. Also, creating binaries for different platforms and distributing them is needed in order to make our project even easier to use.

4. Workload Distribution

Workload of team members are shown below.

Deniz Etkar:

- Literature search about gossip protocols and random membership sampling,
- Initial report and final report writing,
- Producing software architecture ideas,
- Initialization of the internal messaging system,
- Template code creation for all structures/types,
- Implementation of the Membership controller and the Gossiper,
- Design of the custom secure communication protocol.

Daniel Fomin:

- Literature search about gossip protocols and random membership sampling,
- Interim report writing,
- Producing software architecture ideas,
- Implementation of the api listener, p2p listener, api endpoint, p2p endpoint and the secure communication package.

5. Individual Effort Spent

Our effort that we spent on this project, counted in total manhours since the project registration is estimated on an individual basis as follows:

- Deniz Etkar: 100h
- Daniel Fomin: 80h