Midterm Report

Gossip Module
Deniz Etkar and Daniel Fomin

Assumption Changes

Compared to the initial report, we did some changes in our system. We changed from using the Brahms algorithm to using the Median-Counter Algorithm (https://zoo.cs.yale.edu/classes/cs426/2012/bib/karp00randomized.pdf). The change was initiated because Brahms is considered to be a Random Peer Sampling System and not a Gossip Protocol.

For the build system, we started to use a powershell script instead of using a make file as this was more familiar to us. The powershell creates an .exe file which starts the main function of our module. It is probable that we will still create a Makefile in the following time.

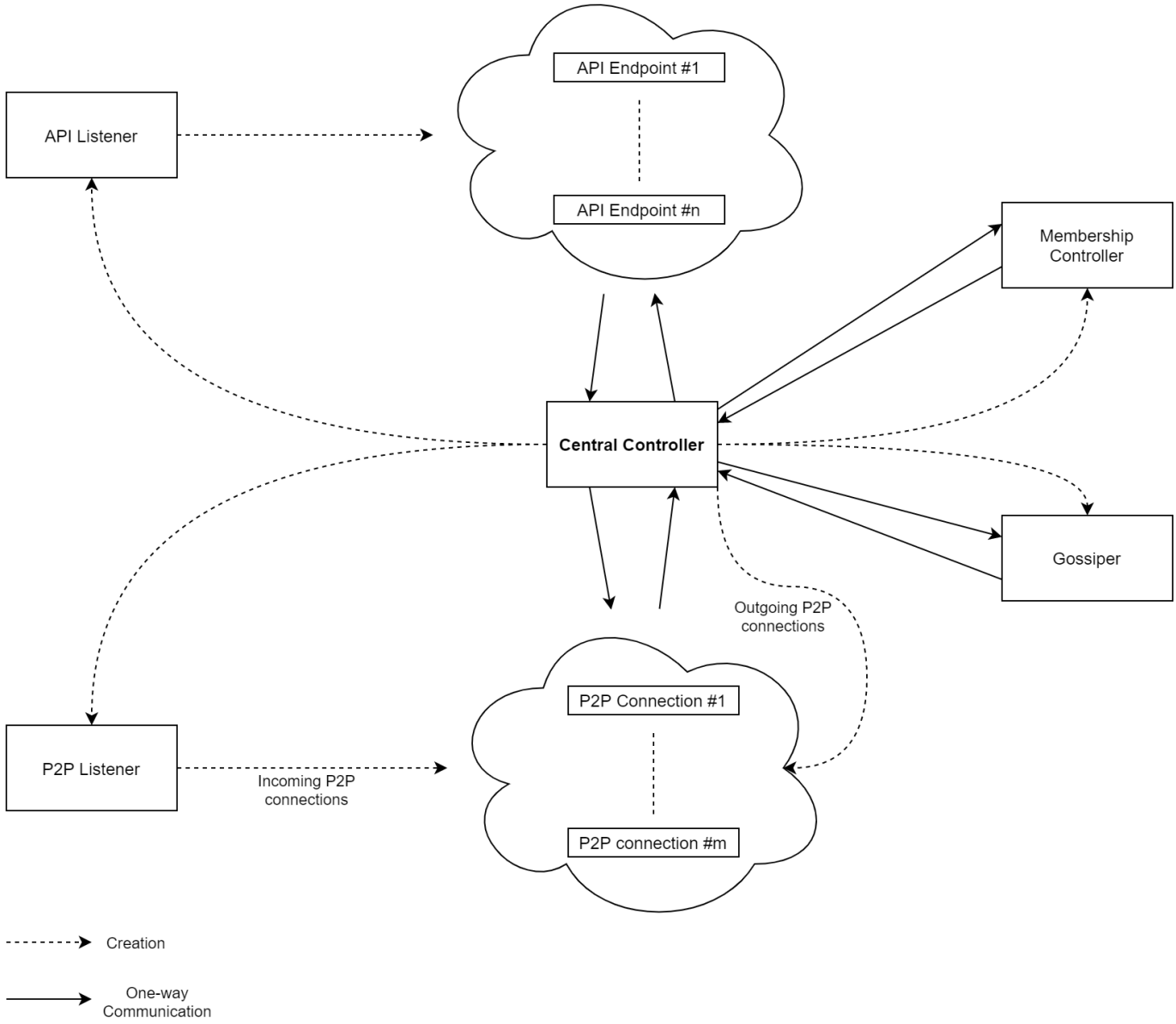
The last change that we did was using the golang library 'encoding/gob' instead of Google's protobuf. The gob library is already integrated into golang and thereby is also more optimized and intuitive to be used with this language. Protobuf on the other hand is implemented to work with a wide variety of languages, which has an advantage when the different communicating programs are written in different languages but this is not the case in our system because all of our peers run the same module, written in the same language.

Module Architecture

Our gossip module consists of five logical structures: Endpoints, Listeners, the Central Controller, Membership Controller and Gossiper.
The **Central Controller** is used to coordinate every task that is done in other structures.
It will be implemented in such a way, that each task that the controller has to manage, will be done in a non-blocking way, so that new tasks can be received and initiated.
The **Membership Controller** manages peers by for example distributing the peer list to other peers or sorting out peers that are not online anymore.
The **Gossiper** is used to manage gossip data, i.e. doing the distribution of the data, e.g. requesting or sending data to a peer, or managing notifications to other peers. The last two logical structures are the Listeners and the Endpoints. These can be further divided into API Listener and P2P Listener, and API Endpoints and P2P Endpoints.
The **API Listener** is used to create API Endpoints for the modules that want to communicate with our Gossip module and the **P2P Listener** is used to create P2P Endpoints for the remote peers that want to communicate with the local peer.
Each **Endpoint** is specific to a communication with an endpoint of another peer.
The endpoints act as an interface for the Central Controller, implementing every read and write

operation, so that the Central Controller can use the internal messaging system to initiate a data transfer.

Every goroutine will communicate over an Internal Messaging System.
The messages are transmitted over golang channels and consist basically of a message type and a data payload that should be transmitted.



P2P Protocol

As already discussed we use for our P2P communication a combination of two endpoints and a secure communication protocol, which sits on top of the tcp-layer. The messages are encoded using 'encoding/gob' which allows us to work not on a byte level but on a higher level by sending structs over the network, at least from the programmers point of view. Similar to the Internal Messaging System, these messages are also a struct that consist of a type and a payload as can be seen in the following code snippet:

```
type P2PMessage struct {
    Type     P2PMessageType
```

```
    Payload AnyMessage
}
```

The type and the data in the payload corresponds to the type and data we receive from the other modules.

Message types

The messages itself can originate either from the gossiper or from the membership controller. The messages that both goroutines initiate are listed in the following:
The **Gossiper** sends at each gossip round push messages as well as pull requests of rumours to existing peer connections, following the "Median-Counter Algorithm". The Gossipers on the other peers then reply with Pull Replies when they receive a pull request.
The **Membership controller** acts in a similar way by sending membership related push/pull messages at every round as well as receiving Disconnect messages from the Central controller for when a remote peer abruptly disconnects.
There is a significant difference between the gossiper and the membership controller rounds. That is that the rounds in the membership controller are much longer, i.e. 30s vs 500ms.

Exception handling

To mitigate problems because of exceptions, we use the following techniques:
If an Endpoint does not reply within a specific timeout, it is deemed to be down and the connection is terminated. Thereby peers can be detected that did not correctly disconnect and be removed from the peer list. To test for corrupted data, a message is forwarded to the destination module which checks the validity of the data, thereby forwarding the problem to the module that should receive the message. To prevent DoS attacks we use 'io.LimitedReader' which allows us to read just a limited amount of bytes from the communication buffer. Not doing this could lead to too much data processing and thereby possibly to crashes of the system. If too much data is transmitted 'io.LimitedReader' throws an error so that the connection can then be terminated.

P2P Communication Tunnel

The communication between the peers will be done by implementing an own level 4 protocol. This protocol includes a Proof of Work (PoW) concept, so that every time an unknown peer is contacted, both peers would need to show that they are legitimate, thereby preventing Sybil attacks. PoW works by choosing a Nonce such that the cryptographic hash the handshake message has certain commonly expected properties (being less than a predetermined number 'k'). The handshake includes a DHE public key, a RSA public key which is shared out of band, a time stamp, ip:port and a nonce. For the specific hash algorithm for PoW, we settled for Scrypt as it is difficult to create Application-specific integrated circuits (ASICs) for that algorithm, giving no peer a too big disadvantage.

As already discussed, too much transmitted data is handled by terminating the communication endpoint as we expect such a situation only to be created with a malicious intent. The communication will furthermore be encrypted using an AES256 key, which is shared through a Diffie Hellman key exchange.

Future work

In the last segment of the project time we need to finalize the P2P and API Listener/Endpoints as well as to finalize the Central Controller. We also need to implement the Secure Communication Channel.

Workload Distribution

In the following, the workload distribution after the initial report is shown.
Deniz Etkar:

- Template creation for all structures
- Implementation of the Membership controller and the Gossiper
- Initialization of the Internal Messaging system

Daniel Fomin:

- Report writing
- API Listener implementation

Effort spent

Our effort that we spent on the projects, counted in hours in total since the initial report is described in the following:

- Deniz Etkar: 45h
- Daniel Fomin: 35h