Peer-to-Peer Networks: Gossip-5 Documentation

Lagemann, Thomas
Peuker, Petra

September 2022

1 Introduction

This documentation describes the python project Gossip-5 which implements the VoidPhone Project Specification in version 2022-0.1. The project and its documentation are part of the joint work of both authors.

2 Usage

In the following section we will introduce into the python application and how to run and test it. For direct execution please refer to the Quick Start Guide.

2.1 Requirements

To run Gossip-5, Python 3.10 is needed. To install the required python packages run:

```
pip install -r requirements.txt
```

2.2 Configuration

To configure the Gossip module, a .ini file should be provided with the following parameters:

- `cache_size`: Maximum number of data items to be held as part of the peer's knowledge base.
- `degree`: Number of peers the current peer has to exchange information with.
- `p2p_ttl` (optional): Specifies the maximum TTL for `GOSSIP PUSH` updates. Default set to 5.
- `bootstrapper` (optional): If this option is not provided, operate as a bootstrapper. Otherwise connect peer to the bootstrapper node identified by IPv4 address and port.
- `p2p_address`: IPv4 address with port this module is listening on for P2P packages.
- `api_address`: IPv4 address with port this module is listen on on for API packages.

An example is given below:

```
[gossip]
cache_size = 50
degree = 30
p2p_ttl = 10
bootstrapper = 127.0.0.1:6001
p2p_address = 127.0.0.1:6011
api_address = 127.0.0.1:7011
```

For additional command line parameters please refer to `waiting_for_validation` and `spread`

2.3 Execution

To execute Gossip-5 a path to an configuration file is allways required. It can be specified by the `-c PATH/TO/CONFIG.ini` parameter. The optional parameters for the validation timeout `-v INT` and the circular spread suppression time `-s INT` can be given in seconds. For detailed information, please refer to `waiting_for_validation` and `spread`. Execution example:

```
cd src
python init.py -c ./config/bootstrap.ini -v 5 -s 60
```

2.4 Tests

To run the Tests please add the full path to Gossip-5/src to your PYTHONPATH:

```
export PYTHONPATH="<path-to-gossip>/Gossip-5/src/"
```

Then the tests can be executed by running:

```
cd tests/unit
python -m unittest
```

2.5 Quick start guide

To install the required packages please run:

```
pip install -r requirements.txt
```

And to run the first client of a network:

```
cd src
python init.py -c ./config/bootstrapper.ini
```

And to run another client please execute:

```
python init.py -c ./config/peer1.ini
```

3 Project Structure

3.1 Project Files

```
Gossip-5
├── docs
│   ├── documentation.pdf
│   ├── initial_report.md
│   └── midterm_report.md
├── src
│   ├── config
│   │   ├── bootstrapper.ini
│   │   ├── peer2.ini
│   │   ├── peer3.ini
│   │   └── peer4.ini
│   ├── log
│   │   └── server.log
│   ├── connection.py
│   ├── gossip_logic.py
│   ├── init.py
│   ├── packing.py
│   ├── server.py
│   └── until.py
├── tests
│   ├── unit
│   │   ├── __init__.py
│   │   ├── helper.py
│   │   ├── test_connection.py
│   │   ├── test_gossip_logic.py
│   │   ├── test_packaging.py
│   │   ├── test_server.py
│   │   └── test_util.py
├── README.md
└── requirements.txt
```

3.2 Architecture

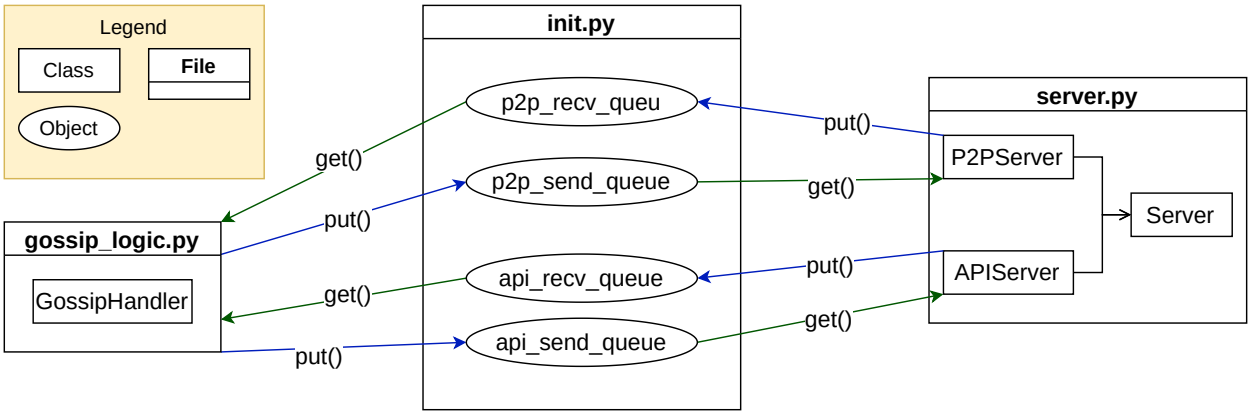

Figure 1: Communication between major project parts

This project uses Python 3.10 with asynchronous I/O through asyncio. With the use of asyncio the reception of messages is decoupled from their processing and a nearly non-blocking I/O can be provided. The program is started via src/init.py which starts the 2 main parts of this project: The servers for message and connection handling and the `GossipHandler` for message processing.

GossipHandler The `GossipHandler` decides how to deal with incoming messages depending on their message type and their source. All messages that do not fit into the defined protocol will be dropped and no response will be sent.

Incoming messages to the API server are directly passed via the `api_recv_queue` to the `GossipHandler`. `GOSSIP NOTIFY` messages will lead to a subscription to be stored in the `notify` list. On a `GOSSIP ANNOUNCE`

message, this list will be checked for registered data types. If the data type is subscribed, a `GOSSIP NOTIFICATION` with the given data will be put into the `api_send_queue` for every fitting subscriber. Additionally, one `GOSSIP PEER NOTIFICATION` message is packed and saved with a timestamp into the `waiting_for_validation` dictionary. If an `GOSSIP VALIDATION` message is then received, the validation will be checked. On a successful validation the waiting `GOSSIP PEER NOTIFICATION` message will be released. If the validation was unsuccessful, the waiting `GOSSIP PEER NOTIFICATION` message will be dropped. If more than one `GOSSIP VALIDATION` will be received for one message, only the first will be considered.

On the peer-to-peer level, the `GossipHandler` releases `GOSSIP PEER NOTIFICATION` from the `waiting_for_validation` dictionary after they were validated. The stored message will be queued into the `p2p_send_queue` to be handled by the P2PServer. Additionally, the `GossipHandler` deals with incoming `GOSSIP PEER NOTIFICATION` messages on the `p2p_recv_queue`. Those will be handled similar to incoming `GOSSIP ANNOUNCE` messages on the `api_recv_queue` with an additional check of the TTL: If the TTL is 0 the message will be announced to modules but not propagated into the network regardless of its validation.

waiting_for_validation and spread dictionaries The `GossipHandler` manages two caches with message IDs. The first one is the `waiting_for_validation` dictionary, which stores `GOSSIP PEER NOTIFICATION` messages until they are validated or expired. Those messages expire after a default time of 5 seconds. This default time can be overridden at the start of the Gossip peer with adding the command line parameter `-c INT` with an integer above 0. A lower value can be useful for fast connections to modules and low traffic on this peer. For high traffic and/or slow connections to modules, a higher rate could be necessary.

The `spread` dictionary caches the message ID of messages that have been forwarded to be sent to other peers. If a message shall be spread to the network, it will be dropped if the spread dictionary contains its message ID. On default, they are stored for 60 seconds, which can be overridden with the command-line parameter `-s INT` with an integer above 0. A high value can help to suppress circulating messages in a slow network, but can lead to dropped messages on reuse of message IDs. Therefore, a low value should be selected for high-speed networks with a high count of messages.

The P2P Server manages the peer's view and is responsible for spreading `PEER NOTIFICATION` packages as well as forwarding them to the `GossipHandler`. The peer's view is represented by a `Connections` object. An entry in the view consists of a peer's IPv4 address and port as an identifier. If there is an existing connection between the host and an peer, `Connections` also provides the current `asnycio StreamReader` and `StreamWriter` pair. According to the project specification, 'A user is assumed to run a single peer at any time'. Thus, while a connection is alive, the reader-writer pair cannot be updated for a peer in the view.

Maintaining the Peer's View The peer's view should only contain peers with active connections to them. Thus, if we want to add a new peer, we establish a connection and introduce ourselves with a `HELLO` message. Since the connection should only be active while there is a corresponding entry in the peer list, the peer to whom we want to connect (Peer2) should also add us to its view. To make DoS and Sybil Attacks more difficult, Peer2 sends us a nonce, we have to do a Proof of Work and send the result. If the result is verified, Peer1 is added to Peer2's view. In response to a correct challenge, we get Peer2's view. This protocol is also depicted in Figure 2.

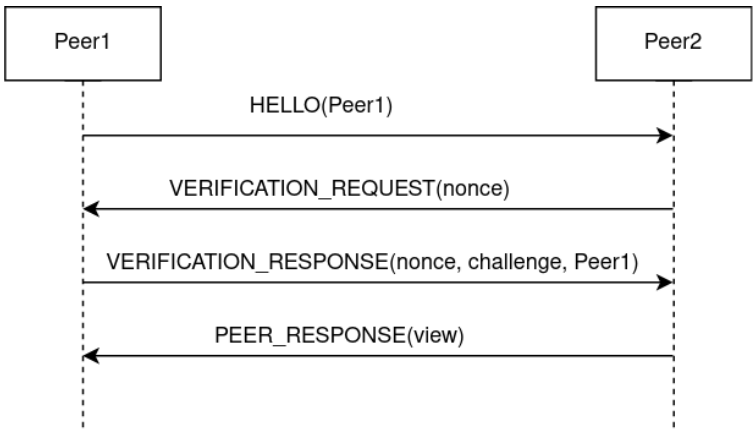

Figure 2: HELLO handshake

Our view will be updated with peers from the `PEER RESPONSE` as long as our view does not exceed the limit, but at most with half of the limit. On a successful addition of a new peer, we spread a `PUSH` message containing the new peer. Peers may also try to establish a connection with `HELLO` with the spread peer.

If the number of items in the view is less than a specific threshold, we send a `PEER REQUEST` message to a random neighbor. This neighbor answer us with `PEER RESPONSE` and its view. If the P2PServer should

spread messages, but the view is empty, the peer reconnects to the bootstapper.

If we have to remove a peer from the list, e.g. when the list size exceeds its limit, we choose a random peer and close the connection.

3.3 Network Packages

3.3.1 GOSSIP ANNOUNCE

Message to spread data in the network. Is sent from other modules to Gossip. For detailed information, review the project specification.

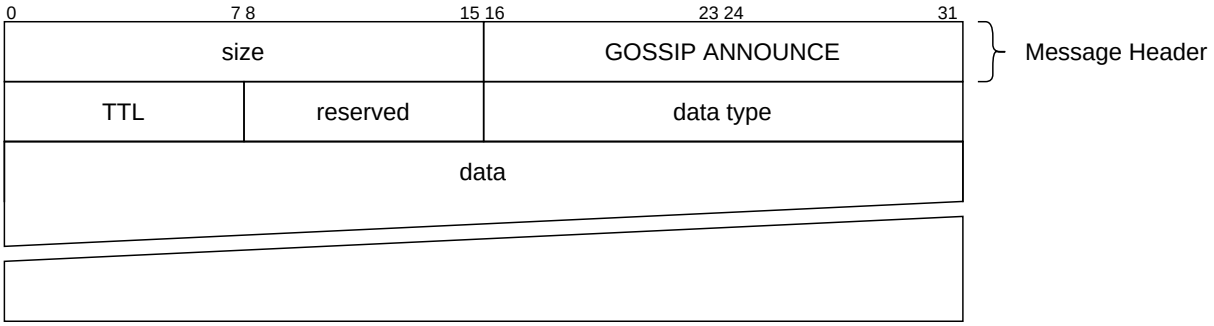

Figure 3: GOSSIP ANNOUNCE

3.3.2 GOSSIP NOTIFY

Message to subscribe for data that will be spread. Sent from other modules to Gossip. For detailed information, review the project specification.

Figure 4: GOSSIP NOTIFY

3.3.3 GOSSIP NOTIFICATION

Message to hand knowledge over to modules. Sent from Gossip to modules that subscribed previously via `GOSSIP NOTIFY`. The knowledge to forward can be received from the modules via `GOSSIP ANNOUNCE` or from peers via `GOSSIP PEER NOTIFICATION`

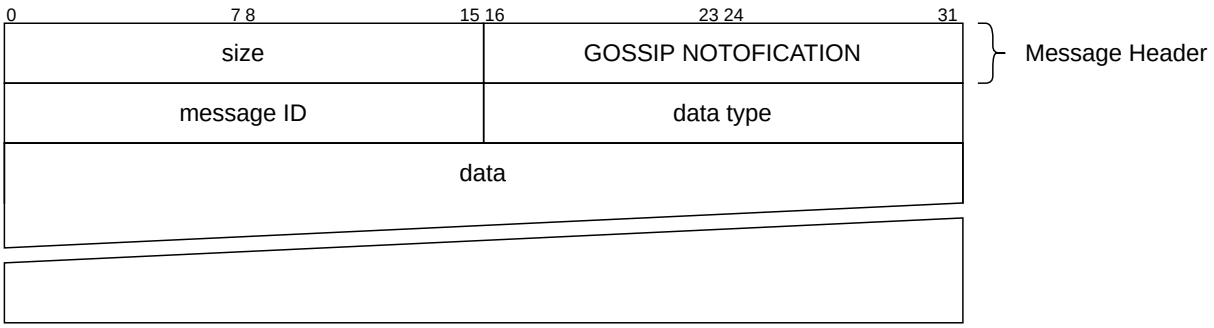

Figure 5: GOSSIP NOTIFICATION

3.3.4 GOSSIP VALIDATION

Message to inform Gossip about valid formatted messages. Sent from the module to Gossip as a response to `GOSSIP ANNOUNCE` and indicates whether to spread a `GOSSIP PEER NOTIFICATION` to the network.

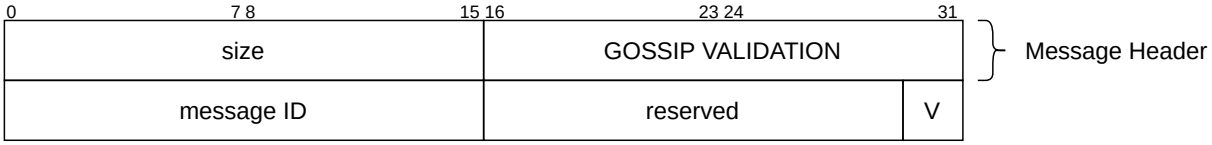

Figure 6: GOSSIP VALIDATION

3.3.5 GOSSIP PEER REQUEST

Ask the peer for its peer list. Should response with a `GOSSIP PEER RESPONSE` message.

Figure 7: GOSSIP PEER REQUEST

3.3.6 GOSSIP PEER RESPONSE

Sends own peers to the receiver.

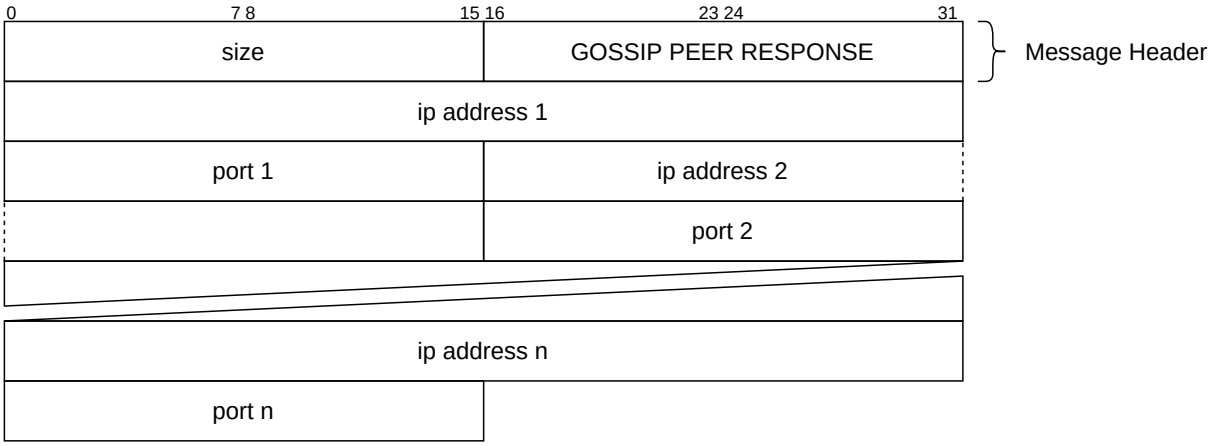

Figure 8: GOSSIP PEER RESPONSE

3.3.7 GOSSIP VERIFICATION REQUEST

Sends a 64-bit nonce that is to be used to generate a response for the `GOSSIP VERIFICATION RESPONSE`. We have to maintain the relation between the receiver and the nonce to check whether the response is valid.

Figure 9: GOSSIP VERIFICATION REQUEST

3.3.8 GOSSIP VERIFICATION RESPONSE

The task for the proof of work is to find a challenge c with $sha256(nonce.concat_{big\_endian}(c)) = r$ so that the first 24 bits of r are 0. The nonce is provided by `GOSSIP VERIFICATION REQUEST`.

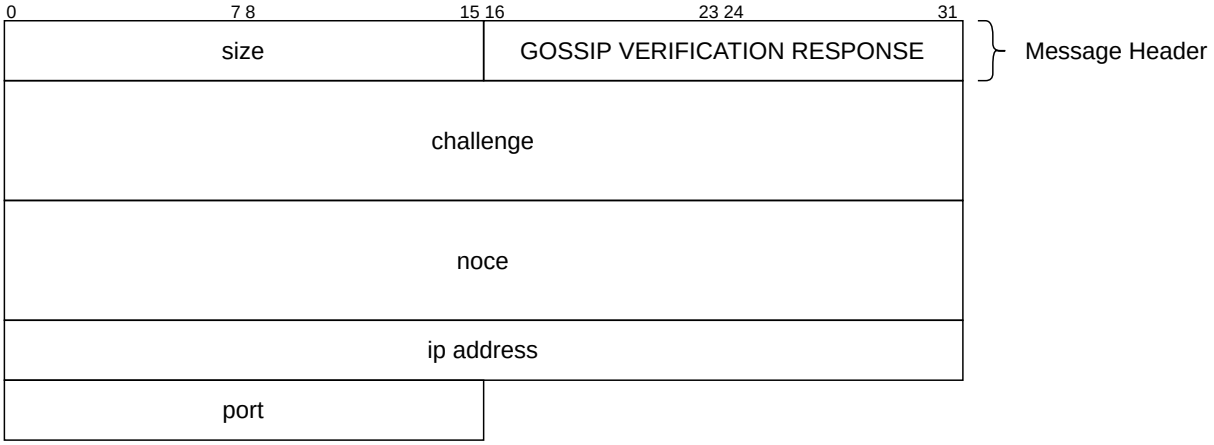

Figure 10: GOSSIP VERIFICATION RESPONSE

3.3.9 GOSSIP HELLO

Signals to the receiver that we added him to the peer view and he should also do this to relate the established connection with the sender's host IP and port. But before adding to the view, the sender has to provide some proof of work.

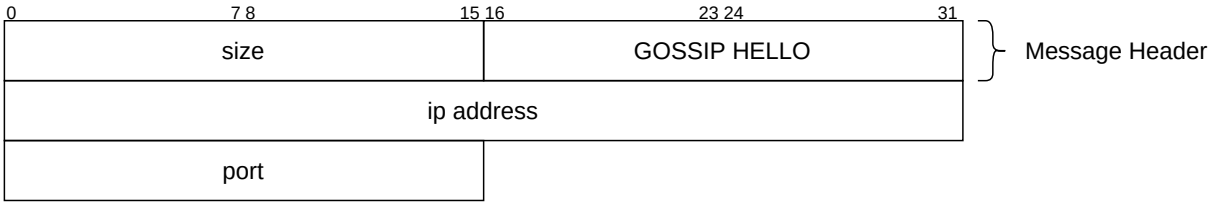

Figure 11: GOSSIP HELLO

3.3.10 GOSSIP PUSH

This message contains the new peer that should be spread. The maximal TTL can be configured in the .ini file. Only establish a connection to the given peer if the message was sent from a neighbor with an existing connection.

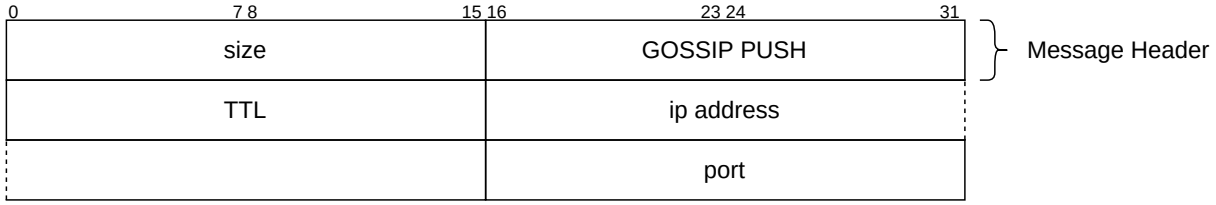

Figure 12: GOSSIP PUSH

3.3.11 PING

This message is sent to verify whether the receiver is still alive.

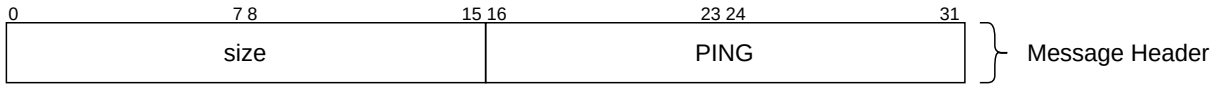

Figure 13: PING

3.3.12 GOSSIP PEER NOTIFICATION

This message spreads knowledge between peers. It is sent between Gossip modules. A received `GOSSIP PEER NOTIFICATION` will be spread to other modules via `GOSSIP ANNOUNCE` When the `GOSSIP ANNOUNCE` message has been verified by a module via `GOSSIP VALIDATION` and the TTL is higher than 0, the message will be spread in the network as new `GOSSIP PEER NOTIFICATION` message with reduced TTL. If the message ID of an incoming `GOSSIP PEER NOTIFICATION` was received and is already waiting for validation or was spread recently, the message will be dropped. The duration until those message IDs expire can be set with the validation time and the spread suppress time.

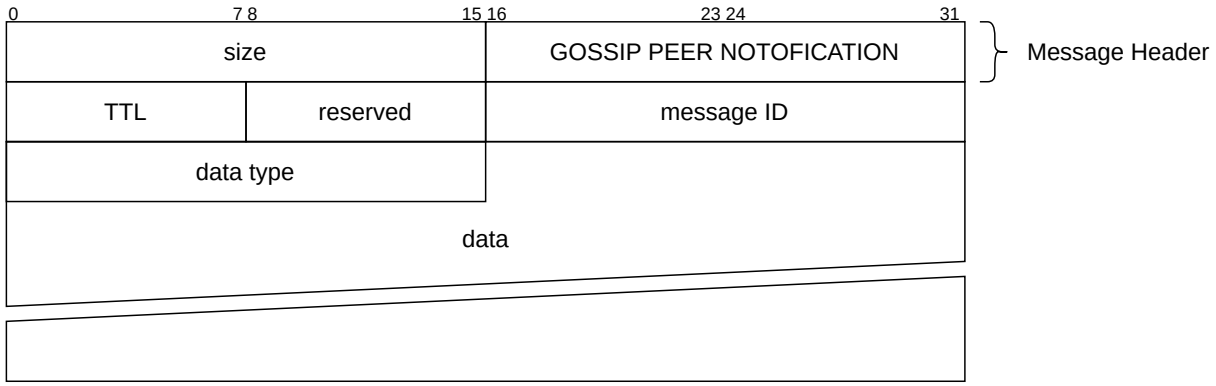

Figure 14: GOSSIP PEER NOTIFICATION

4 Security

We assume that sensitive data is encrypted by higher layers.

To mitigate Sybil attacks, we require a PoW from peers that send us a request to add them to our view. Peers sent with `GOSSIP PEER RESPONSE` and `GOSSIP PEER PUSH` messages are only accepted and further processed if they are sent to us from a known peer. We also only accept `GOSSIP VERIFICATION REQUEST` messages from connections registered in the peer list. This should mitigate a DoS attack where we are forced to solve many PoW challenges.

5 Known Issues

Currently, one problem with the connection handling is regarding the locking in the P2PServer. Since the peer view and other data structures are accessed by multiple coroutines, we prevent race conditions with `asnycio.Lock`, which are mutexes. However, this decreases the positive impact of coroutines on the execution time. This could be mitigated by using, for example, Read-Write locks.

Another problem occurs while handling the state of connections to whom we sent a nonce for validation. Since we should verify that the nonce sent in the `GOSSIP VERIFICATION REQUEST` is the same as in the corresponding `GOSSIP VERIFICATION RESPONSE`, we need a data structure to keep track of it. When a `GOSSIP VERIFICATION REQUEST` is sent, we add an entry to this data structure. However, if there is no `GOSSIP VERIFICATION RESPONSE` sent from the corresponding connection, the entry remains in the data structure forever. This can be easily be abused to DoS the Gossip module by sending a lot of `GOSSIP HELLO` messages to the peer.

A further problem, correlating with the last, is that we currently do not support any timeout for the `GOSSIP VERIFICATION RESPONSE`. This is quite critical, since Sybil attacks are easier now.

The `GossipHandler` stores subscriptions in a list(`notify`) with the corresponding receiver. This list grows with every new `GOSSIP NOTIFY` message, but never decreases. This could be a problem on peers that do not restart regularly.

6 Future Work

To solve the issue regarding the verification timeout previously described, future work includes establishing a timestamp for every pending verification and reject those that send a `GOSSIP VERIFICATION RESPONSE` not in time.

To solve problems with the constantly growing `GossipHandler.notify` list, the status of a subscription could be either checked on a regular base inside of the `GossipHandler` or handed by the `APIServer`.

The timings for `waiting_for_validation` and `spread` could be tested in real network condition to receive knowledge about better default values for various network types. Until now, those are only tested on local networks.

Support for IPv6 can be added to be more flexible in network selection. Support for Microsoft Windows can be added and especially tested. The current state of functioning on MS Windows is not known.

7 Workload Distribution

7.1 By Structure

Petra Peuker Implemented networking and Server structures. Created program bootstrapping/init process.

Thomas Lagemann Implemented the Gossip logic as specified in the project specification. Main work in files `exgossip_logic.py` and `test_gossip_logic.py`.

Shared Workload Multiple planning and debugging meetings. Creation of documentation.

7.2 By Time

Petra Peuker Between 3 and 4 weeks á about 7 hours a day

Thomas Lagemann About 2 weeks á 6 to 8 hours a day