P2P Gossip Protocol–
An Implementation discussing Architectural and Security Standpoints

Alexander Liebald, Kevin Ploch
contact.liebald@gmail.com, contact@ploch.me

Peer-to-Peer Systems and Security
Project Documentation
Summerterm 2021

Contents

1 API

This section refers to the API protocol provided by the Chair of Network Architectures and Services as part of the Peer-to-Peer-Systems and Security course. All rights lie with the chair, and any license provided with this project does not apply to this protocol.

As part of this course, it was assumed that all clients of this API behave honestly. Therefore, there are no security precautions to defend against attacks from clients in place. Security is focused on potential attackers within the peer to peer system (gossip to gossip).

1.1 General Format and Message Header

All API message integer fields are encoded in network-byte order (Big-endian) and do not contain floating-point numbers.

Every message starts with a header displayed in fig. 1, **size** allows to predict the amount of information to be read. The **type** indentifies the type of the message. These will be elaborated on in the following.

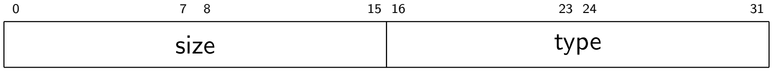

Figure 1: Message header format

type	field value
GOSSIP ANNOUNCE	500
GOSSIP NOTIFY	501
GOSSIP NOTIFICATION	502
GOSSIP VALIDATION	503

Table 1: Message types and their protocol field values

1.2 GOSSIP ANNOUNCE

Users can utilize this message to spread data in the network. It is sent to Gossip's listening API. No return message is provided.

The **TTL** tells Gossip how many hops this data should traverse. A value of 0 indicates unlimited hops. The **data type** is an identifier for the application data. Every subscriber (explained later) of this data type receives this application **data**.

The nature of this protocol is best-effort delivery, therefore no confirmations for delivery are sent.

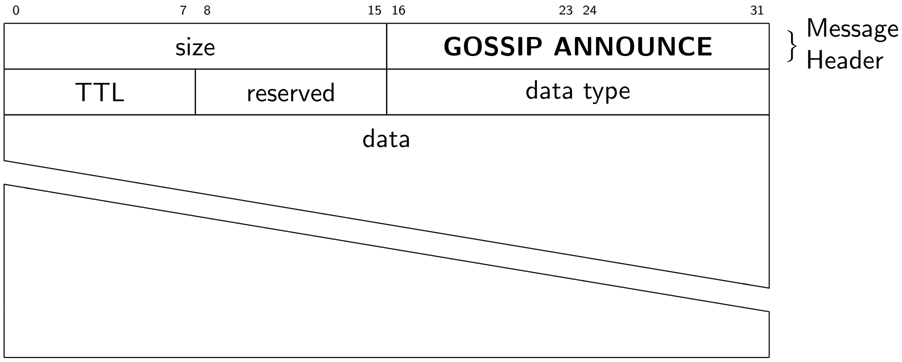

Figure 2: GOSSIP ANNOUNCE message format

1.3 GOSSIP NOTIFY

This message tells Gossip that the sender is interested in application data of type **data type**. Therefore Gossip will forward any received GOSSIP ANNOUNCES with type **data type** to the sender of this message. The second purpose of this message is that it lets Gossip know which data types are valid and should be spread further should the TTL field allow it.

This message does not evoke any response from Gossip and the subscription is cancelled should the API user disconnect.

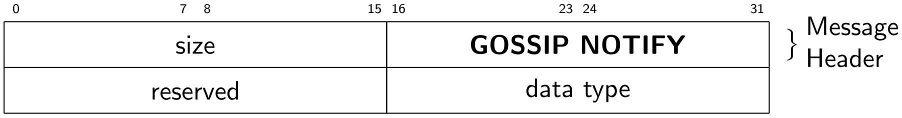

Figure 3: GOSSIP NOTIFY message format

1.4 GOSSIP NOTIFICATION

When Gossip receives a message from another Peer, it notifies all subscribers of the received data type by sending them a GOSSIP NOTIFICATION with the received data. Message ID is a random number identifying this message and the corresponding GOSSIP VALIDATION, which the client receiving this message has to reply with.

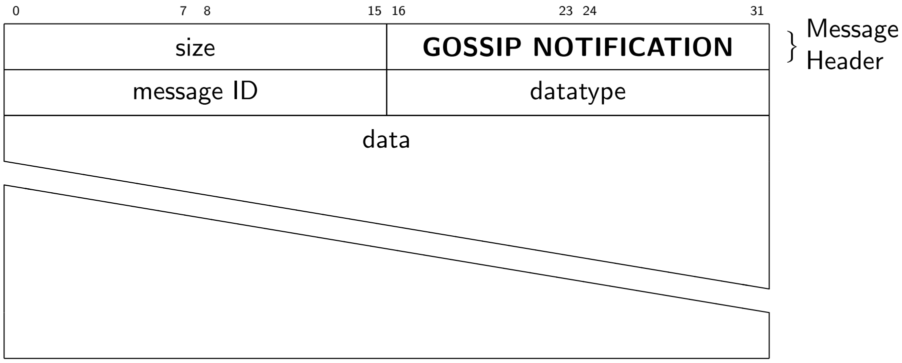

Figure 4: GOSSIP NOTIFICATION message format

1.5 GOSSIP VALIDATION

This message is send as a response to GOSSIP NOTIFICATION, telling Gossip whether the message with message ID is well-formed (V-bit = 1) or not (V-bit = 0). Only after all notified clients validate the message, it is propagated further. In case a client responds with invalid, the message with the given message id is discarded.

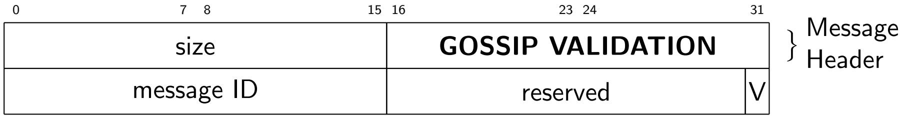

Figure 5: GOSSIP VALIDATION message format

2 Architecture

2.1 Project Folder Structure

The folder structure is split into the following:

config.ini Configuration file in Windows INI file format. Read by the program at startup, for a detailed description and a list of all possible parameters, see section 2.2.4

main.py Main file, execute to start. For details see section 3.3.

docs/ Documentation

modules/ Contains most of the code, file contents will be explained in this section and can also be seen in a short overview at the top of each module.

test_configs/ Configuration files for testing, see section 2.6

testing/ black box testing and unit tests, see section 2.6.

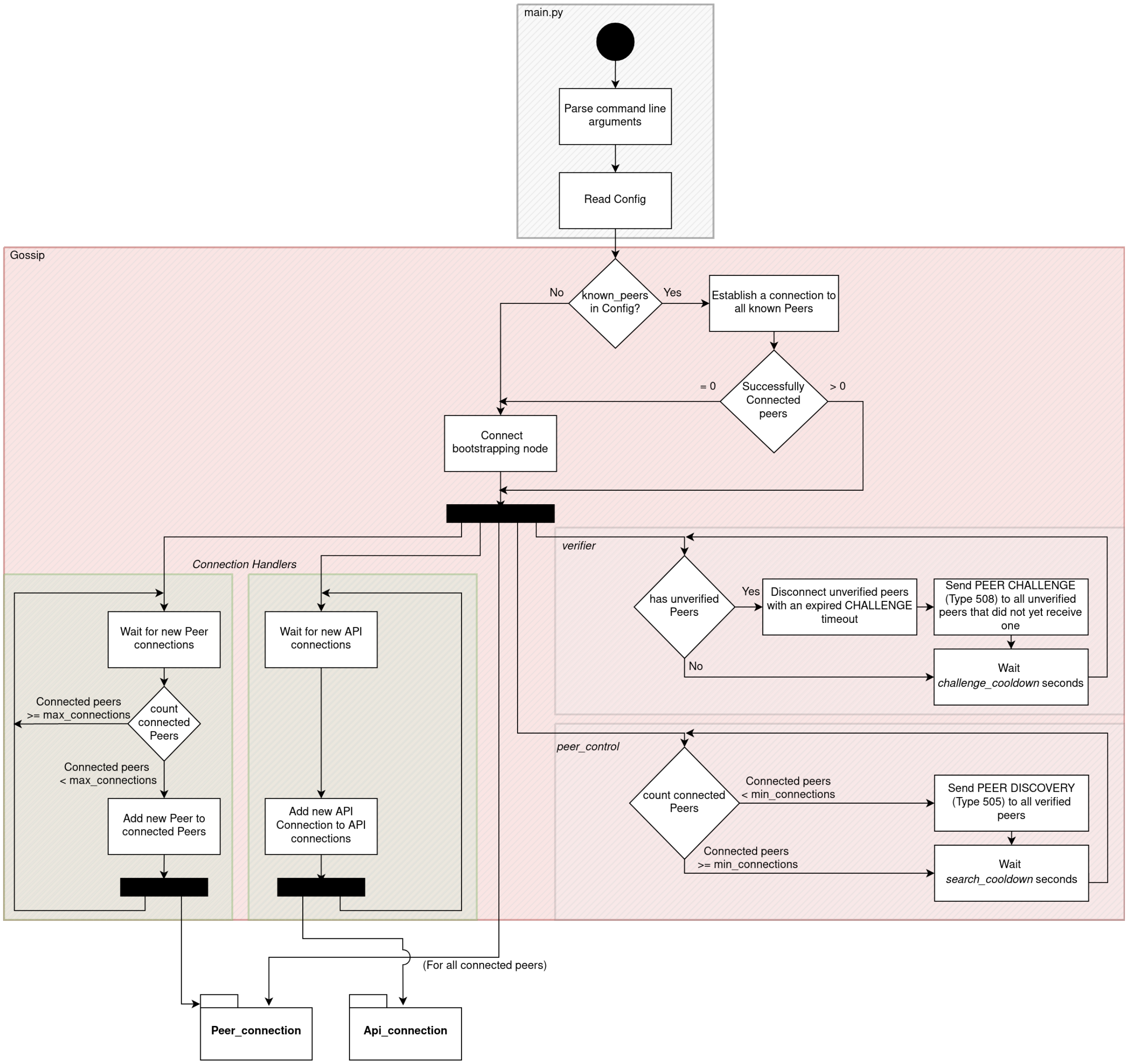
main.py
Parse command line arguments
Read Config
Gossip
known_peers in Config?
No
Yes
Establish a connection to all known Peers
Successfully Connected peers
= 0
> 0
Connect bootstrapping node
Connection Handlers
Wait for new Peer connections
count connected Peers
Connected peers >= max_connections
Connected peers < max_connections
Add new Peer to connected Peers
Wait for new API connections
Add new API Connection to API connections
verifier
has unverified Peers
Yes
No
Disconnect unverified peers with an expired CHALLENGE timeout
Send PEER CHALLENGE (Type 508) to all unverified peers that did not yet receive one
Wait challenge_cooldown seconds
peer_control
count connected Peers
Connected peers < min_connections
Connected peers >= min_connections
Send PEER DISCOVERY (Type 505) to all verified peers
Wait search_cooldown seconds
(For all connected peers)
Peer_connection
Api_connection

Figure 6: Flow chart for main.py and Gossip class

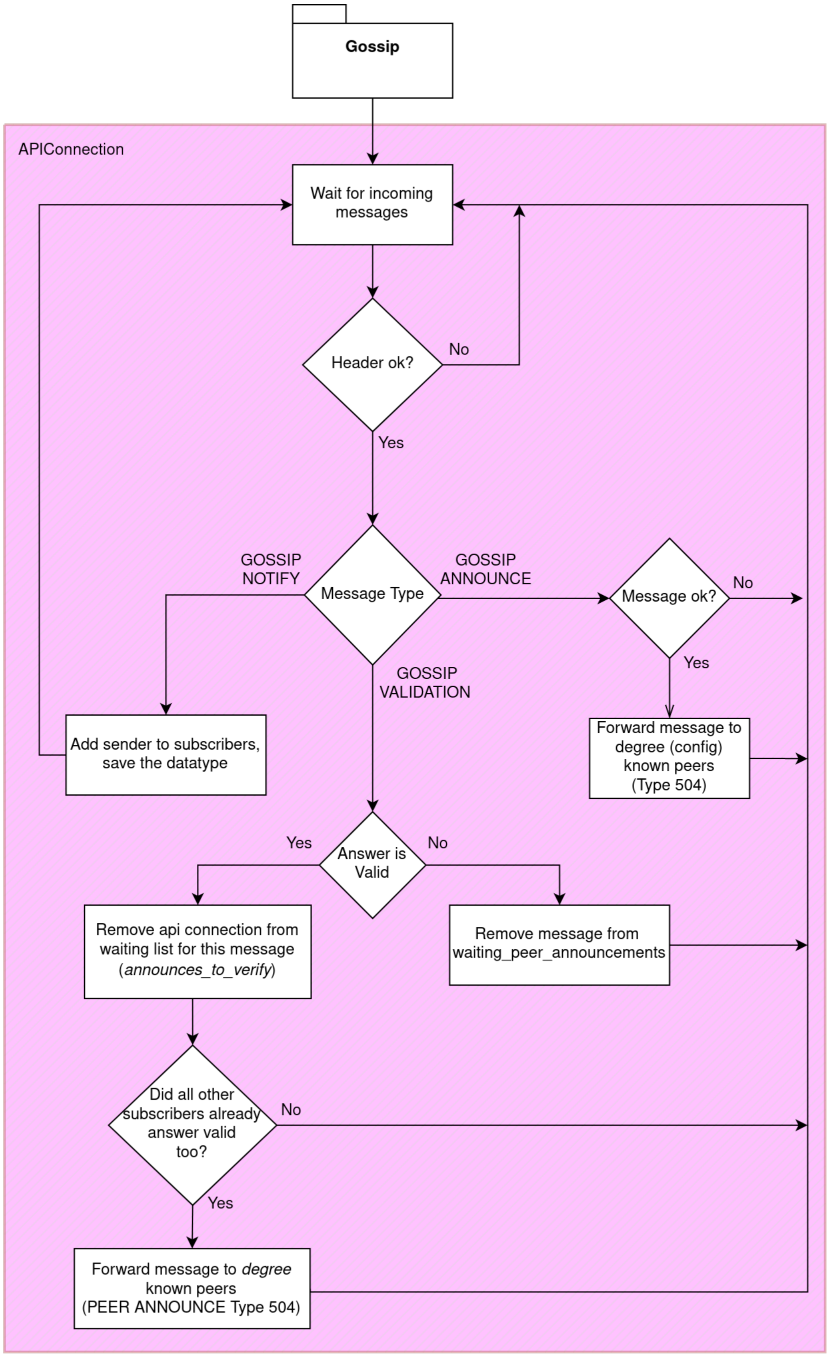
Gossip
APIConnection
Wait for incoming messages
Header ok?
No
Yes
Message Type
GOSSIP NOTIFY
GOSSIP ANNOUNCE
GOSSIP VALIDATION
Message ok?
No
Yes
Add sender to subscribers, save the datatype
Forward message to degree (config) known peers (Type 504)
Answer is Valid
Yes
No
Remove api connection from waiting list for this message (announces_to_verify)
Remove message from waiting_peer_announcements
Did all other subscribers already answer valid too?
No
Yes
Forward message to degree known peers (PEER ANNOUNCE Type 504)

Figure 7: Flow chart, Api_connection

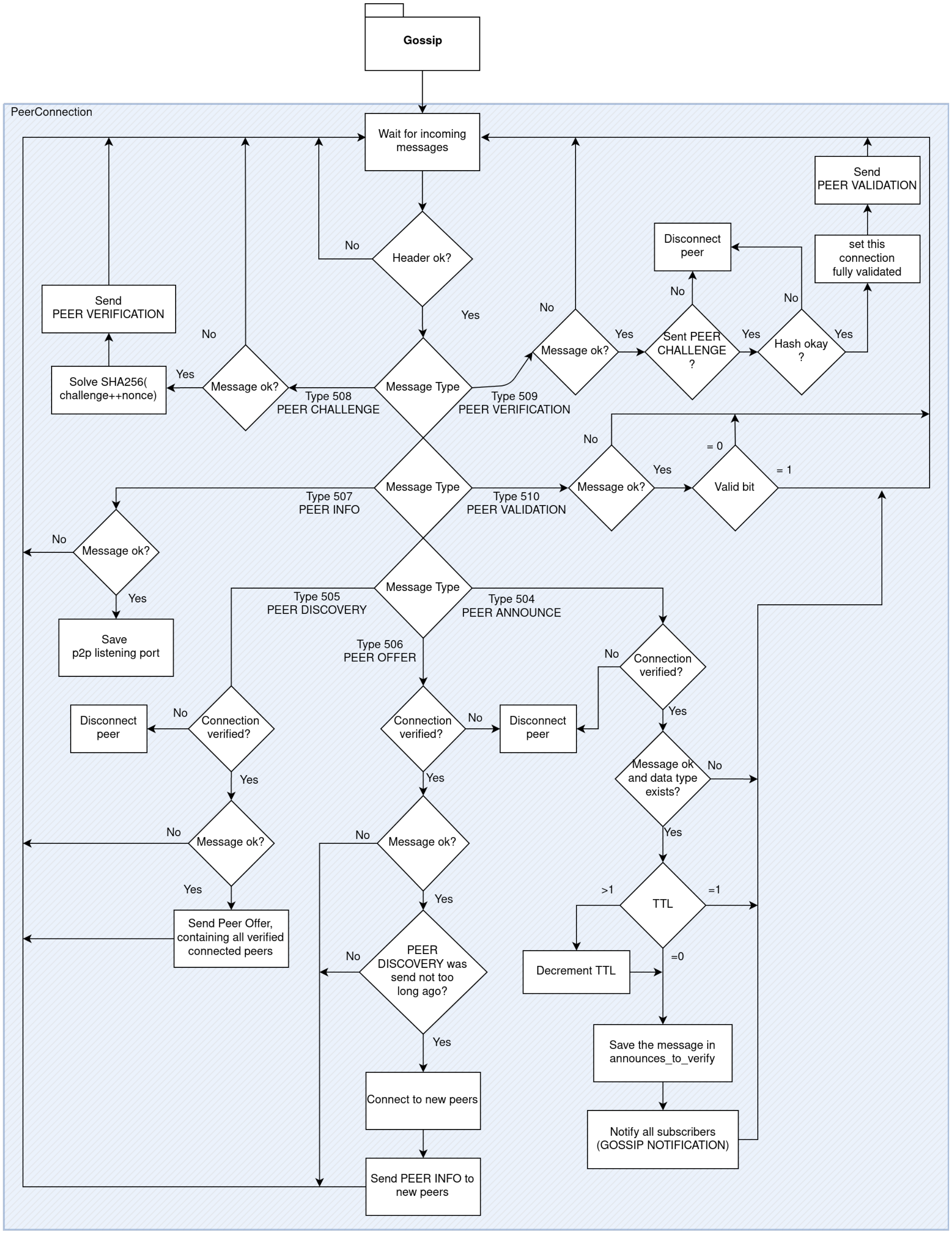

Figure 8: Flow chart, Peer_connection

2.2 Logical Structure

To provide a clear and understandable architecture, we decided to use a class based approach. A UML diagram of all classes can be seen in figure 9. Besides these classes we also used smaller modules, which provide some functionality, which is used in multiple parts of the program.

2.2.1 Gossip Class

The Gossip class is central to the software. After initializing it and calling its run() function, the program starts socket listeners for peer and API connections. Should a new connection be received, an instance of either Api_connection or Peer_connection is initiated (see section 2.2.2 and section 2.2.3). Also the bootstrap procedure is started by connecting to either peers given in the config or a bootstrapping node (see 2.2.4). To further maintain a certain level of connectivity to the P2P network it also starts peer_control, which is responsible for searching new pull peers using PEER DISCOVERY every `search-cooldown` seconds, and peer_verifier, which periodically sends out PEER CHALLENGE-s every `challenge-cooldown` seconds to unverified push peers to initiate the verification process (see section 2.4.2). Gossip manages all central information about API and peer functionality in several data structures, see table 2.

In general, whenever a message arrives in Peer/Api_connection and we need to update a data structure e.g. by removing an API connection from the *apis* list, these changes are handled by the Gossip class as it contains these data structures.

Variable	**Purpose**	**Data structure**
push_peers	peers aquired from PUSH Gossip	deque
pull_peers	peers aquired from PULL Gossip	list
unverified_peers	PULL peers without completed handshake	deque
apis	connected API users	list
datasubs	API subscribers of a datatype	dictionary
peer_announce_ids	Prevent spreading loops	Setqueue (2.2.5)
announces_to_verify	PEER ANNOUNCES we are awaiting API confirmation for	dictionary

Table 2: Gossip Class Datastructure Overview

2.2.2 Peer_connection Class

An instance of Peer_connection represents a connection to another peer. The class provides the functions to communicate with the other peer and a run() function, which waits for incoming messages from the connected peer. As soon as a message arrives it passes one central handling function, which determines the next step by checking the messsage type in the header. The peer message types are listed in table 3. Next the individual handling functions of each type message type checks whether the size field matches the actual size of the packet. Then the program flow continues depending on the message.

2.2.3 Api_connection Class

Api_connection works similarly to its peer counterpart, representing an active connection to a single API. Instead of peer protocol messages it handles API protocol messages specified in the course

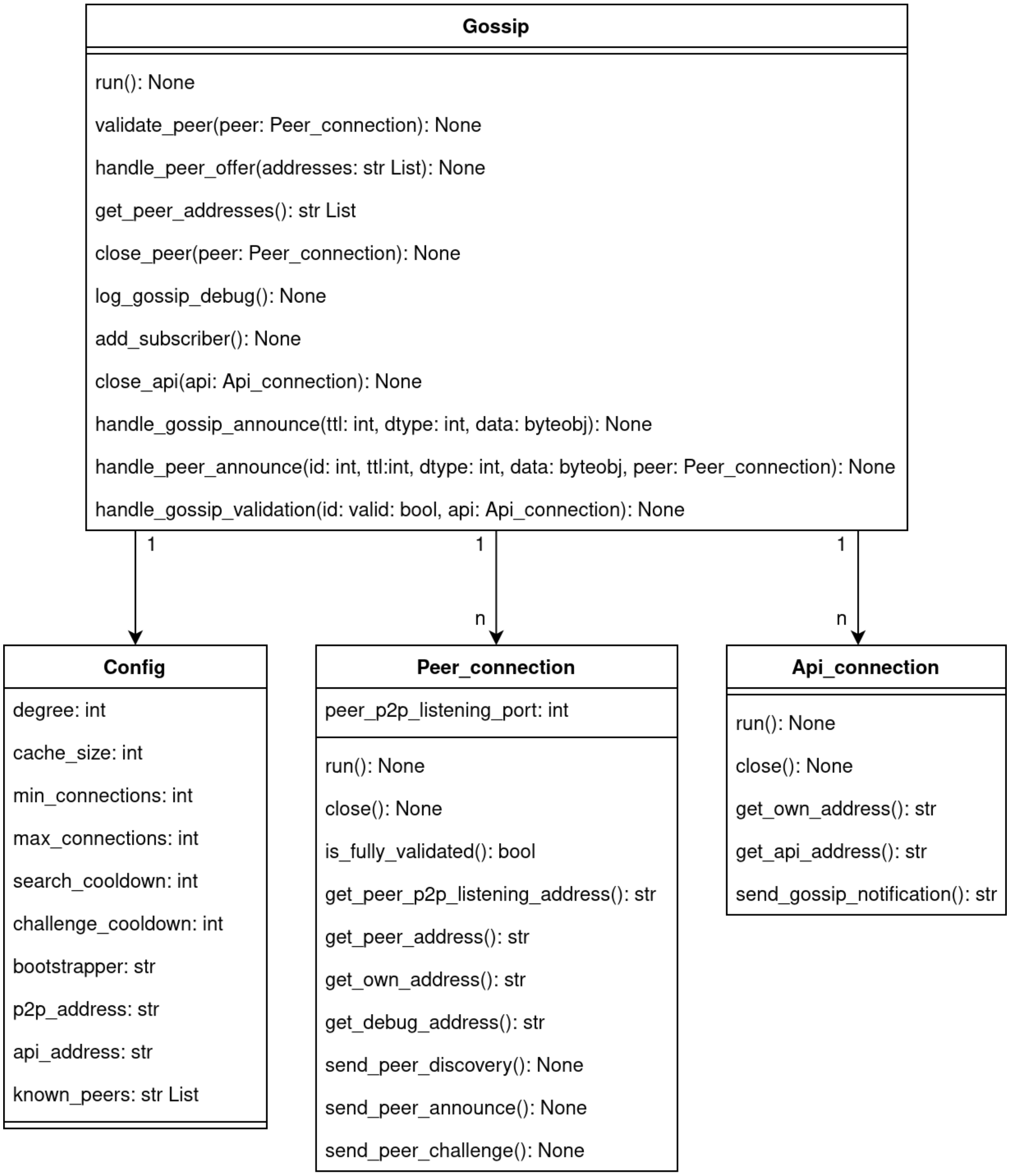
Gossip
run(): None
validate_peer(peer: Peer_connection): None
handle_peer_offer(addresses: str List): None
get_peer_addresses(): str List
close_peer(peer: Peer_connection): None
log_gossip_debug(): None
add_subscriber(): None
close_api(api: Api_connection): None
handle_gossip_announce(ttl: int, dtype: int, data: byteobj): None
handle_peer_announce(id: int, ttl:int, dtype: int, data: byteobj, peer: Peer_connection): None
handle_gossip_validation(id: valid: bool, api: Api_connection): None
1
1
n
1
n
Config
degree: int
cache_size: int
min_connections: int
max_connections: int
search_cooldown: int
challenge_cooldown: int
bootstrapper: str
p2p_address: str
api_address: str
known_peers: str List
Peer_connection
peer_p2p_listening_port: int
run(): None
close(): None
is_fully_validated(): bool
get_peer_p2p_listening_address(): str
get_peer_address(): str
get_own_address(): str
get_debug_address(): str
send_peer_discovery(): None
send_peer_announce(): None
send_peer_challenge(): None
Api_connection
run(): None
close(): None
get_own_address(): str
get_api_address(): str
send_gossip_notification(): str

Figure 9: Gossip UML class diagram

project specification.

2.2.4 Config Class

The Config provides an abstraction for config files. It contains parsing and allows for quick access to config parameters. The Config module also contains a ‘config_config‘ variable, which is used as a blueprint for parsing config files. It allows quick and easy addition of new parameters, setting default values, types and constraints / checks.

The following variables are available:

- `cache_size`: Maximum number of data items to be held as part of the peer's knowledge base. Older items will be removed to ensure space for newer items if the peer's knowledge base exceeds this limit. Is used for `unverified_peers` capacity and `peer_announce_ids` cache size
  Constraints: must be greater than 0.

- `degree`: Number of peers this peer exchanges information with. Relevant for PEER ANNOUNCE.
  Constraints: must be smaller or equal to `min_connections`, `min_connections` and grater than 0.

- `min_connections`: Minimum amount of alive connections this peer should try to keep.
  Constraints: must be greater than 0 and bellow or equal to `min_connections`.

- `max_connections`: Maximum amount of alive connections this peer should keep.
  Constraints: Must be greater than or equal to 2 and `min_connections`.

- `search_cooldown`: In this interval it is checked whether we have `min_connections` peers. If not start search.
  Constraints: must be greater than 0. If this variable is not given the default value of 1 minute is used.

- `challenge_cooldown`: Every challenge_cooldown seconds the verification process (section 2.4.2) is initiated for all unverified peers. Since this is a security feature, it is not recommended to be experimented with.
  Constraints: must be greater than 0. If this variable is not given the default value of 5 minutes is used.

- `bootstrapper`: One trustworthy bootstrapping node, used as a fallback in case no known_peers are given or none can be reached.
  Constraints: must be a valid IPv4, IPv6 address or domain in the format `<address>:<port>`

- `p2p_address`: Listening ip and port number for other Gossip peers. Must be a valid and unused port.
  Constraints: must be a valid IPv4, IPv6 address or domain in the format `<address>:<port>`

- `api_address`: Listening ip and port number for API users. Must be a valid and unused port.
  Constraints: must be a valid IPv4, IPv6 address or domain in the format `<address>:<port>`

- `known_peers`: Comma separated list of peers to bootstrap through, if it is empty consult `bootstrapper`.
  Constraints: must have the format: `<address>:<port>,<address>:<port>`. Can also be left out / not required

All variables, besides `search_cooldown`, `challenge_cooldown` and `known_peers` are required. If a variable is missing or malformed, the program will log an error and exit.
To reload the config, the program must be restarted.

2.2.5 Utility Module

The utility module (*util.py*) provides utility functions which wouldn't explicitly fit into another module. It contains a custom datastructure, the Setqueue, which is a modified FIFO queue with the property of not having duplicate elements in it. It is used for storing routing IDs of already received packets to avoid message loops. The utility module also contains functions for generating and checking Proof of Work, parsing and validating ip addresses as well as resolving domains to ip addresses.

2.2.6 Packet Parser Module

The packet parser module provides functions for parsing byte-objects to tuples consisting of the specified protocol fields and vice-versa for building byte-objects for protocol messages. Whenever a parameter is wrong or a byte-object doesn't fit the protocol message criteria, *None* is returned.

2.3 Process Architecture

We utilize pythons *Asyncio* library for asynchronous code execution. The Gossip class runs two parallel tasks with sockets waiting for connections on the API or peer listening port. Additionally, tasks for verifying peers within the `challenge_cooldown` interval and finding new peers within the `search_cooldown` exist. If new connections are established, an instance of Peer- or Api_connection is initiated and its run() method is started as a new parallel task. There, new messages from this peer or API are awaited, received and handled.

2.4 Security Measures

2.4.1 Attacker model

As this is a peer to peer application, we want others to participate by running own nodes and so form a P2P network. What we do not want is one entity controlling a lot of nodes in the network, which gives them the ability to isolate nodes, a so called *Sybill attack*.
When looking at the analogy of the OSI model, the lower layers (1-3) take on the responsibility of connectivity through single and multihop communication. Authentication and encryption in TCP/IP is done in upper layers. We apply the same for Gossip as a bottom level protocol for an application. It exists to build new connections and hold them, aswell as keeping enough connectivity to the network. Further security can be built on top of Gossip. Therefore we decided **not** to include node authentication nor communication encryption between peers.

Security Mechanisms Overview:

1. When receiving a malformed or unknown message, the corresponding connection is terminated immediately. Since TCP is used as an underlying networking protocol, it is assumed that malformed messages are not the result of an networking error but an error that occurred on the side of the connected peer / a malicious peer.

2. **Connection establishment**: In an interval we perform a Proof of Work handshake on all unverified peers that connected to us in this interval to limit short-term sybil attacks.

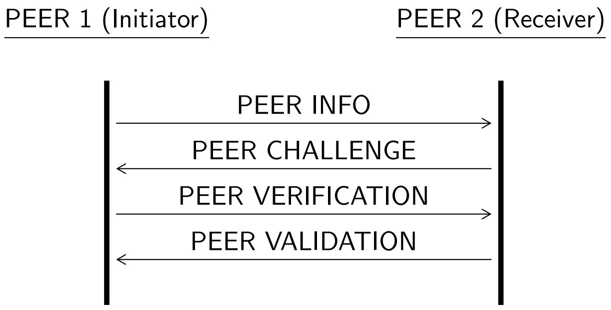

Figure 10: Handshake upon connection establishment

3. **FIFO push peers**: Oldest push connections get replaced first, should new incoming connections arrive and the push peers capacity is reached. This makes long-term attacks even more difficult.

4. **PUSH/PULL Gossip**: PUSH and PULL peers are differentiated, each with their own capacity. For more details see section 2.4.3.

2.4.2 Connection Establishment / Handshake

If one node (initiator) wants to connect to another (receiver), it will open up a connection to the listening port of the node and start the handshake depicted in fig. 10.

The initiator starts with a PEER INFO message, which lets the receiving node know of the initiators listening port (see section 2.5.4). Now begins a break. Connections that get opened by an initiator will be put on hold by the receiver. In a regular interval, defined with the *challenge_cooldown* variable (see section 2.2.4), the receiver will challenge all initiators that connected in this interval at once with a Proof of Work puzzle. The duration of the interval is configurable, see section 2.2.4. This Proof of Work procedure begins with a PEER CHALLENGE message (see section 2.5.5), which contains a 64 bit challenge. The challenge is unique for each initiator to prevent sharing of solutions. A correct solution consists of SHA512(challenge,nonce) (with padding in compliance with RFC 6234), where the input are the byte-objects of challenge and nonce concatenated, with the first **24 bits** set to zero in the hash.
For future improvements see section 4.

Upon finding such a nonce, the initiator sends the nonce in a PEER VERIFICATION message (see fig. 16) to the receiver. Should this verification take longer than 5 minutes, the challenge is considered expired and the connection is closed by the receiver.

The receiver checks the nonce by trying to solve the challenge with the given nonce. He will finish the handshake by sending a PEER VALIDATION message, which gives the initiator the feedback whether he solved the challenge correctly and now is a peer of the receiver or it failed (see section 2.5.7). Only after receiving a valid VALIDATION all other messages, which are not required in the handshake, are allowed.

2.4.3 PUSH / PULL Gossip

Each peer maintains two different types of connections which depend on the perspective. When learning about a node and opening a connection to it, this node is considered a **pull peer**. If another node is opening a connection to us, this node is a **push peers**. Until a peer is verified (section 2.4.2), push peers are considered **unverified peers**. These do not count towards the capacity of push peers and communication is limited to the verification process.

As a defence precaution, the maximum peer capacity given by max_connections (section 2.2.4) is split up between pull and push peers. In case max_connections is uneven, the capacity of pull peers will be larger by one compared to the capacity of push peers.

After a unverified peer has been verified, it is added to push peers. In case push peers has reached its capacity, the oldest push peer will be closed. This, in combination with the verification process, helps to defend against eclipse attacks by updating possible routes, therefore making it harder to continuously stay in contact with a peer (see lecture 6).

2.5 Protocol Messages

Number	Name	Purpose
504	PEER ANNOUNCE	spreading of knowledge
505	PEER DISCOVERY	discovering new peers
506	PEER OFFER	spreading information about other peers
507	PEER INFO	Proof of Work Handshake, listening port information
508	PEER CHALLENGE	Proof of Work Handshake, challenge
509	PEER VERIFICATION	Proof of Work Handshake, nonce
510	PEER VALIDATION	Proof of Work Handshake, true/false

Table 3: Peer Message Overview

2.5.1 PEER ANNOUNCE

The PEER ANNOUNCE message is central in spreading information from one peer to another. GOSSIP ANNOUNCE messages from APIs are translated to this message for further spreading. As can be seen in fig. 11, it has similar protocol fields. One addition is the 64 bit id field, which is used to differentiate messages from each other and prevent message spreading loops. The corresponding datastructure in the Gossip class is *peer_announce_ids* (see section 2.2.1).

In case we receive a PEER ANNOUNCE from a peer we keep spreading it if and only if we know about this datatype by having API subscribers to it. If we have a subscriber to this datatype, we transform the PEER ANNOUNCE into a GOSSIP NOTIFICATION. If we receive a negative PEER VALIDATION from the API that received the NOTIFICATION, we do not propagate it further.

The TTL field functions equally to the GOSSIP ANNOUNCE TTL field. A value of zero equals to infinite and one means it is ending at this node. It is decremented on every hop.

2.5.2 PEER DISCOVERY

Figure 12 shows this message. It tells the receiver that this node wants to extend his connectivity by asking for new pull peers.

To find new pull peers, do the following:

```
capacity        = current push peers < max push peers
below_min      = current push & pull peers < min_connections
```

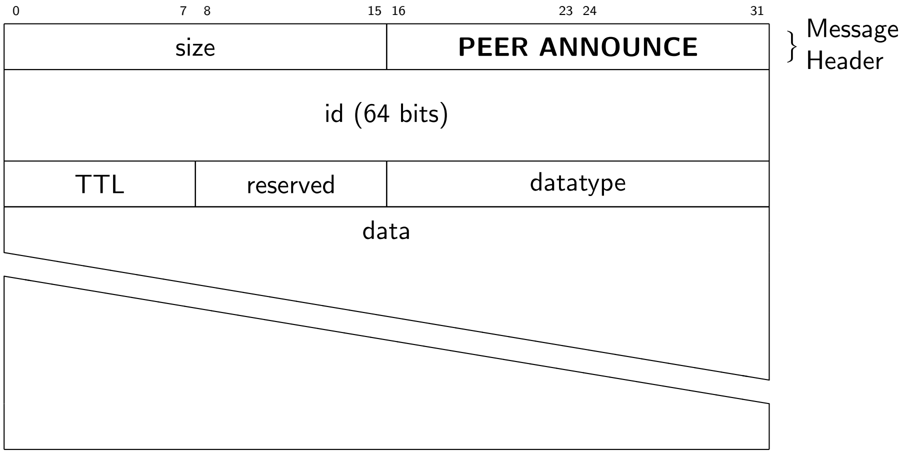

Figure 11: PEER ANNOUNCE message format

```
below_min_push = current pull peers < min_connections / 2

while running
    if capacity and (below_min or below_min_push)
        1. Send PEER DISCOVERY to all verified peers
        2. Peers respond (PEER OFFER) with a list
           containing all their known peers
        3. Connect to peers in all given lists in a random order
           until we reach max push peers.
    sleep for search_cooldown seconds
```

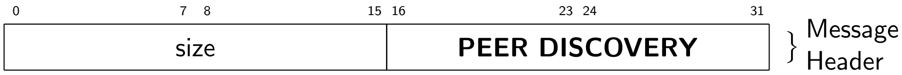

Figure 12: PEER DISCOVERY message format

2.5.3 PEER OFFER

Figure 13 shows this message. Answer to a PEER DISCOVERY message. Includes the senders peer list. We remove the sender of the PEER DISCOVERY from the list before sending it.

2.5.4 PEER INFO

The PEER INFO message is part of the connection establishment handshake and tells the opposing end of this nodes listening port for new connections. It's structure can be seen in fig. 14. This message is necessary since the port used to connect to another peer is dynamic and not equal to the p2p address port. This port number is also spread further in the network through each peers list when answering with PEER OFFER to PEER DISCOVERY to enable connection establishment.

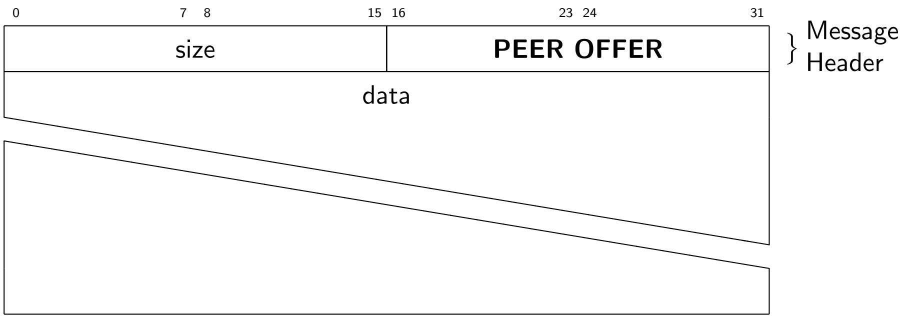

Figure 13: PEER OFFER message format

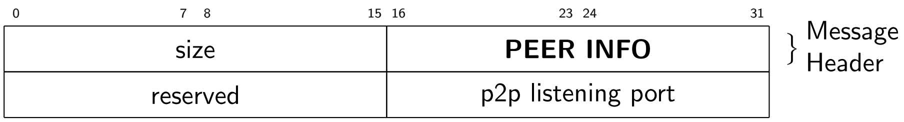

Figure 14: PEER INFO message format

2.5.5 PEER CHALLENGE

The PEER CHALLENGE is the second message of the connection establishment handshake. As can be seen in figure 15 it mainly consists of its header and a 64 bit challenge. For more details on the proof of work handshake see section 2.4.2.

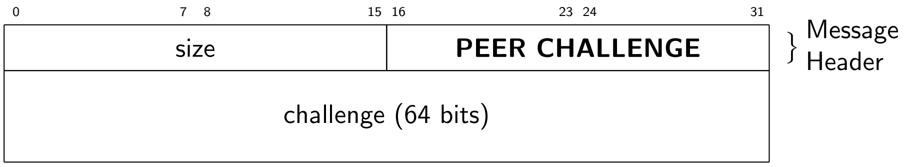

Figure 15: PEER CHALLENGE message format

2.5.6 PEER VERIFICATION

The PEER VERIFICATION is the third message in the handshake and is used to send a found nonce in the Proof of Work procedure (see section 2.4.2). Its simple structure can be seen in figure fig. 16.

2.5.7 PEER VALIDATION

The PEER VALIDATION message is the last message in the handshake and tells the receiver whether he is now a verified peer for the sender or not. This is indicated in a bitfield, as can be seen in fig. 17. A set bit indicates a positive feedback.

The structure of the PEER VALIDATION message can be seen in figure 17.

2.6 Testing

2.6.1 Unit Tests

packet_parser.py Unit tests for the packet parsing functions are defined in **/testing/test_packet_parser.py**

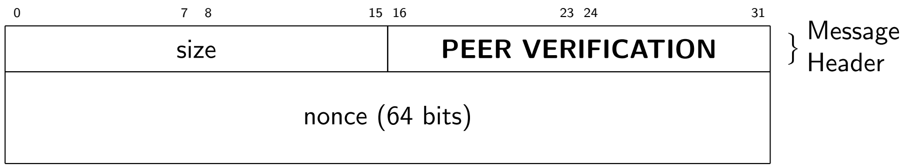

Figure 16: PEER VERIFICATION message format

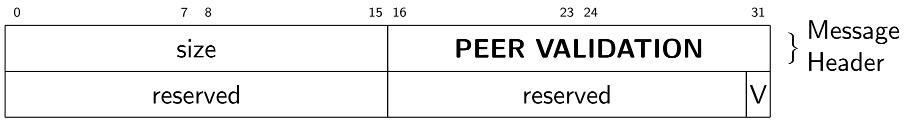

Figure 17: PEER VALIDATION message format

The test can be executed with:

```
python3.9 testing/test_packet_parser.py
```

test_config.py Unit tests for the Config class are defined in **/testing/test_config.py**
The test can be executed with:

```
python3.9 testing/test_config.py
```

2.6.2 Blackbox Tests

Blackbox tests help in testing parts of the application by creating mock API connections and/or peers, send data and observe how the system behaves.

Running Blackbox Tests

1. Start Gossip with the special test config:

   ```
   $ python3.9 main.py -p test_configs/blackbox_config.ini
   ```

   It has a low challenge cooldown for a quick handshake after starting the tests instead of waiting for a full intervall of e.g. 5 minutes

2. In a new shell, start the test.

Handshake functionality To test the handshake of Gossip run the test **testing/test_handshake.py** according to the description above:

```
# 1. Start Gossip
$ python3.9 main.py -p test_configs/blackbox_config.ini
# 2. In a second shell
$ python3.9 testing/blackbox_test_handshake.py
```

This test uses a mock peer to connect to the running Gossip software and performs the connection establishment handshake.

PEER ANNOUNCE functionality To test the correct handling of PEER ANNOUNCEs run the python program **testing/blackbox_test_pannounce.py** according to the description above:

```
# 1. Start Gossip
$ python3.9 main.py -p test_configs/blackbox_config.ini
# 2. In a second shell
$ python3.9 testing/blackbox_test_pannounce.py
```

The test program is going to test a scenario with 2 connected mock Peers and 2 connected mock API users. They will connect to our Gossip instance we started before, which is listening for new connections. The peers will complete the handshake and the APIs will subscribe to the datatype 1 by sending a GOSSIP NOTIFY. After this setup one peer sends a PEER ANNOUNCE with the datatype 1. We check whether this triggers GOSSIP NOTIFICATIONs to the APIs and if thats the case we answer with a GOSSIP VALIDATION from both. Now Gossip should spread the PEER ANNOUNCE in the network by taking a sample of size degree, which will effectively be the peer that we not send the PEER ANNOUNCE from (we do not send it back to the sender). Lastly we check whether this peer received said PEER ANNOUNCE.

GOSSIP ANNOUNCE functionality

```
# 1. Start Gossip
$ python3.9 main.py -p test_configs/blackbox_config.ini
# 2. In a second shell
$ python3.9 testing/blackbox_test_gannounce.py
```

This program tests the handling of GOSSIP ANNOUNCE messages. Two mock API users and one mock peer connect to the running Gossip program. The peer completes the handshake and both APIs send a GOSSIP NOTIFY with datatype 1 to subscribe to said datatype. One API now sends a GOSSIP ANNOUNCE. We test whether the peer receives the now transformed GOSSIP ANNOUNCE in form of a PEER ANNOUNCE.

3 Software Documentation

3.1 Excursus: Virtual Environments in Python

Although the software runs on windows and linux, this section is targeted at GNU/Linux users. For windows reference see here.

Your global python installation includes executables and libraries that could impact running this project by e.g. choosing the wrong library version. To avoid headaches like this python provides the useful module **venv** to create a completely isolated python environment with its own executable and libraries in the project folder. This is accomplished by running

```
$ python3.9 -m venv /path/to/project/.venv
```

in the Gossip Project folder. This creates the folder ".venv" at the specified path with its own python executable and libraries should you now decide to install some.
To enter the environment simply run:

```
$ source /path/to/project/.venv/bin/activate
```

and to exit it:

```
(venv)$ deactivate
```

You can easily uninstall it by removing the created folder, ".venv" in our example.

3.2 Dependencies

To execute the program, Python 3.9 or above is required. To install all our library dependencies simply run:

```
$ python3 -m pip install -r requirements.txt
```

to install them globally or run it inside a virtual environment (Section 3.1) to install them for the project only.

3.3 Executing the program

To run the software, the **main.py** file must be executed using the Python interpreter.

```
$ python3.9 main.py
```

It is possible to change the used configuration file (section 2.2.4) by adding -**p** or --**path** followed by a path to an existing ini file. The default configuration file is the config.ini in the root folder.
By adding the -**v** or --**verbose** flag, additional debug information will be displayed during execution. This Information can give a better and more detailed insight into the execution, but is not required for a overview over events during execution.
With -**l** or --**logfile** followed by a valid path, all logging will be written into the file at the end of the path. If the given file does not yet exist, it will be created. Otherwise, new logs will be appended to the current content. Note that the folder structure given in the path must already exist.

```
$ python3.9 main.py -p config.ini -v -l example.log
```

3.4 Code Conventions

3.4.1 Code Style

For a homogenous and readable code style we settled for Python's PEP 8.

3.5 Known issues

Mixed Gossip networks with both IPv4 and IPv6 addresses are prone to errors. If a IPv4 peer finds out about a IPv6 peer, he might still connect using an IPv6 and therefore appear twice in the newtork, once with its IPv4 and once with its IPv6.

4 Further Improvements

4.1 More Tests

Generally improving test coverage by adding more tests could improve the project. For example, a blackbox test for PEER DISCOVERY/PEER OFFER interaction would help in debugging problems more easily. The coverage of unit tests for PEER packets in testing/packet_parser.py could be increased.

4.2 Multiprocessing in Proof of Work

The current Proof of Work implementation runs on a single core and therefore uses a rather few leading zero bits of 24 bits. A security improvement would be to adjust to multiprocessing through higher amount of leading zero bits in the target hash. In its current form, solving a challenge is also blocking the rest of the program, which could be solved using an external process which could be awaited.

4.3 Peer Blacklist

Peers that failed to solve the handshake in a moderate time and peers that act maliciously by e.g. sending wrong message formats could be put on a blacklist, therefor denying future communication. Currently we just terminate the connection.

4.4 Peer List History Sample

In addition to filling the peer list with PUSH and PULL peers, a history sample could be included which only includes peers we were previously connected to.