TUM

Peer-to-Peer Systems and Security

IN2194, SoSe 2023

Gossip Module

for Anonymous and Unobservable VoIP Application

Project Documentation

Team #97

1. API Protocol Specification:

This section specifies and explains the API protocol that is used for communication between Gossip and the other modules within the same local peer.

1.1. GOSSIP_ANNOUNCE Message:

This message is used to instruct Gossip to spread the knowledge about given data items. It is sent from other modules to the Gossip module. No return value or confirmation is sent by Gossip for this message.

The TTL field specifies how many hops the overlying application requires this data to be spread. A value of 0 implies unlimited hops. The data type field specifies the type of the application data. The data type should not be confused with the message type field: While they are similar, message types are used to identify messages in the API or the P2P protocols, whereas the data type is used to identify the application data Gossip spreads in the network.

Since this message does not evoke a response from Gossip, no assumptions about successful spreading of the data in the message can be made. Knowledge spreading is best effort and can only be seen as probabilistic. However, with enough cache size and well connectivity, it is very likely to achieve a good chance of knowledge spreading in the network.

When a peer receives this message, It will transform it to a GOSSIP_NOTIFCATION message [1.3] and send it to all subscribed modules on the same local peer which subscribed through a GOSSIP_NOTIFY message [1.2], and and it will also transform it to a PEER_ANNOUNCE message [2.4] and send it to all connected external peers in the network.

Size (16 bits)		**GOSSIP ANNOUNCE** (16 bits)
TTL (8 bits)	Reserved (8 bits)	Data Type (16 bits)
Data		

1.2. GOSSIP_NOTIFY Message:

This message serves two purposes. Firstly, it is used to instruct Gossip to notify the module sending this message when a new application data message of a given type is received by it. The new data message could have been received from another peer through a PEER_ANNOUNCE message [2.4] or from another module of the local peer through a GOSSIP_ANNOUNCE message [1.1]. The sender of this message will be notified by the Gossip through GOSSIP NOTIFICATION messages [1.3]. The data type field specifies which type of messages the caller is interested in being notified. The second purpose of this message is to tell Gossip which message types are valid and hence should be propagated further to other peers. This means only messages for which a module has subscribed a notification from Gossip will be propagated by Gossip to other peers.

This message does not evoke any response from Gossip. The notification is canceled when the API connection is closed. Note that there is no API action to explicitly cancel the notification. To cancel it, just close the API connection and open a new one.

Size (16 bits)	**GOSSIP_NOTIFY** (16 bits)
Reserved (16 bits)	Data Type (16 bits)

1.3. GOSSIP_NOTIFICATION Message:

This message is sent by Gossip to the module which has previously asked Gossip to notify when a message of a particular data type is received by Gossip. The receiver should have previously subscribed to this data type through a GOSSIP NOTIFY message [1.2]. As mentioned before, the data sent in this message is received either from other peers in the network by a PEER_ANNOUNCE message, or from other modules on the local peer by a GOSSIP_ANNOUNCE message.

If the data was received from other peers, the message ID in this message will be a random number which will be used to identify this message and its corresponding GOSSIP_VALIDATION response. Every subscribed module needs to validate this message before we can propagate it further to other peers.

On the other hand, data received from other modules by a GOSSIP_ANNOUNCE will be sent in the GOSSIP_NOTIFICATION with a message ID = 0. We do not expect the subscribed modules to validate this message and we can directly propagate it further to other peers since we assume that all API connections are honest.

Size (16 bits)	**GOSSIP_NOTIFICATION** (16 bits)
Message ID (16 bits)	Data Type (16 bits)
Data	

1.4. GOSSIP_VALIDATION Message:

The message is used to tell Gossip whether the GOSSIP_NOTIFICATION with the given message ID is well-formed or not. The bitfield V, if set, signifies that the notification is well formed; otherwise it is to be deemed invalid and hence should not be propagated further.

Size (16 bits)	**GOSSIP_VALIDATION** (16 bits)	
Message ID (16 bits)	Reserved (15 bits)	V (1 bit)

2. P2P Protocol Specification:

This section specifies and explains the P2P protocol that is used for communication between different peers in a network.

2.1. PEER_INIT Message:

For peers to connect to each other, the initiator node will connect to the socket on which another peer is listening on. When the receiver node receives a connection, it will immediately send this message to the initiator, which will contain a random 64-bits challenge that is part of the Proof of Work Puzzle.

Size (16 bits)	**PEER_INIT** (16 bits)
Challenge (64 bits)	

2.2. PEER_VERIFY Message:

When a peer initiates a connection with another peer and receives the PEER_INIT message [2.1] with the 64-bits challenge, the initiator peer then has to solve this challenge by finding a nonce such that if the challenge, the nonce, and the p2p listening port of the initiator are concatenated together and then hashed using the SHA 256 algorithm (with padding in compliance with RFC 6234), the resulting hash should have a certain number of leading zero bytes. This certain number of leading zero bytes is the “challenge_difficulty” which is configured in the config file of the module.

After finding a suitable nonce, It is then sent back to the other peer along with the port on which the current peer is listening on for p2p connections. It is important for the receiver peers to know the listening ports of peers connecting to it, so that it can share it with other peers in the PEER_BROADCAST message [2.6].

Sending this message has to be done within a specific time after receiving the PEER_INIT message otherwise the verification process will fail and the connection will be dropped. This specific time is configured as “challenge_timeout” in the config file.

Size (16 bits)	**PEER_VERIFY** (16 bits)
Reserved (16 bits)	P2P Listening port (16 bits)
Nonce (64 bits)	

2.3. PEER_OK Message:

After the initiator peer sends back the PEER_VERIFY message [2.2] to the receiver peer, the receiver peer has to check if the nonce it received is indeed valid and if it was sent within the allowed time frame. To do this, the receiver peer has already stored in its memory the challenge it previously sent, along with a timestamp.

To check if the nonce is valid, the receiver peer will use the challenge it has stored and concatenates to it the nonce and the port it received from the PEER_VERIFY message. It will then hash the result and check if the final result has a certain number of leading zero bytes according to the “challenge_difficulty” config variable.

If the nonce is valid and was sent within the allowed time frame, the receiver peer will send back the PEER_OK message to notify the initiator that it considers the connection between them valid. Both peers are now trusting each other, and as a security measure: a peer will only start accepting other messages types from a certain connection only after it verifies this connection and it will drop the connection otherwise.

Size (16 bits)	**PEER_OK** (16 bits)

2.4. PEER_ANNOUNCE Message:

This message is used to spread data between connected peers in the network. When a peer receives this message, it will first check if it has any modules that are previously subscribed to the data type specified in this message via a GOSSIP_NOTIFY message [1.2]. It will then make sure that it has not already received this data before, by checking if the hash of the data does not exist in its cache. If any of these conditions are not met, the receiver peer will discard this

message, otherwise, it will continue by sending a GOSSIP_NOTIFICATION message [1.3] to all the currently subscribed modules on the API layer.
If the TTL in the received message is 1, the current peer will only send it to its subscribed modules and it will not propagate it further to other peers in the network. If the TTL received in this message is either 0 (unlimited hops) or more than 1, the current peer will propagate this message further with a decremented TTL (in case it is more than 1) when all the subscribed modules respond with a GOSSIP_VALIDATION message [1.4] signifying that the data is valid. If any of the subscribers indicates that the data is not valid, the data will not be propagated further and the connection with the sender peer will be dropped as a security measure.

A peer will also form this message and send it to all peers it is connected to if it receives a GOSSIP_ANNOUNCE message [1.1] from other modules via the API layer.

<table>
<tr><td colspan="2">Size (16 bits)</td><td>PEER_ANNOUNCE (16 bits)</td></tr>
<tr><td>TTL (8 bits)</td><td>Reserved (8 bits)</td><td>Data Type (16 bits)</td></tr>
<tr><td colspan="3">Data</td></tr>
</table>

2.5. PEER_DISCOVER Message:

This message is sent by every peer in the network to all other peers they are connected to every certain amount of time, if they do not already have a full connections list. This amount of time is configured as “discovery_cooldown” in the config file.

The message is used to learn about other peers in the network by asking known peers for the addresses of other peers they know.

<table>
<tr><td>Size (16 bits)</td><td>PEER_DISCOVER (16 bits)</td></tr>
</table>

2.6. PEER_BROADCAST Message:

This message is sent as a response to the PEER_DISCOVER message [2.5]. The data field holds a list of all the addresses and ports of all the peers that the sender of this message is connected to.

The receiver of this message will then try to connect to these peers if it is not already connected to any of them.

Size (16 bits)	**PEER_BROADCAST** (16 bits)
Data	

3. Architecture:

Our implementation consists of 6 main classes: Config, Gossip, APIServer, P2PServer, APIConnection and P2PConnection. In this section we are going to explain the functionality of each of them and how they interact with each other.

3.1. Config Class:

Initializing the Config class takes as an input the config file's path. This class is then responsible for parsing and validating this config file, which includes making sure all required config parameters exist in the right format, data type and follow other specific constraints. The class provides an easy way to define new config parameters and their constraints.

After validation, each required parameter is then stored as an instance variable of this class, and an instance of this class is initialized during the initialization of the Gossip Class [3.2] and is stored as an instance variable of the Gossip Class. This allows us to easily access these parameters anywhere in the program whenever needed.

The following are the required parameters that should exist in the config.ini file:

- **cache_size:** The maximum number of data messages that will be stored in the peer's cache. The cache stores data received from other peers through the PEER_ANOUNCE message [2.4]; this is needed to be able to detect when the same data is received again and hence not propagate it again. We hash the data and only cache the hash to save memory space.
  Constraints: should be an int.
- **degree:** The maximum number of peers the current peer can be connected to.
  Constraints: should be an int.
- **bootstrapper:** The address of the first peer to connect to when the current peer starts.
  Constraints: should be a valid ip_address:port
- **p2p_address:** The address and port on which the current peer should listen on for p2p connections.
  Constraints: should be a valid ip_address:port
- **api_address:** The address and port on which the current peer should listen on for api connections.
  Constraints: should be a valid ip_address:port
- **challenge_timeout:** The maximum time in seconds allowed for other peers that are trying to connect to the current peer to solve the challenge by finding a valid nonce.
  Constraints: should be an int.
- **challenge_difficulty:** The number of zero leading bytes that are required to solve the Proof of Work Puzzle for other peers to validate their connection with the current peer.
  Constraints: should be an int, should be >=0 and <= 64.
- **discovery_cooldown:** The time in seconds to wait between each attempt to discover more peers in the network by asking for the current connections' knowledge.
  Constraints: should be an int.

3.2. Gossip Class

This class can be seen as the main backbone of the whole module. To initialize it, it only takes the config_file_path as an input which it uses to initialize the Config class as mentioned in [3.1]. Along with the Config class instance, it also stores as instance variables other data structures that are essential for the whole program to function, making it the main store for the module's state data. This single Gossip instance is then passed down to all instances of all the other classes, making its data structures available to read or manipulate anywhere and anytime.

The following are the data structures that this class stores as its instance variables:

- **api_connections:** A list that holds information about all the current API connections in the form of APIConnection class instances [3.5].
- **subscriptions:** A map to hold information about which API connections have subscribed to which data types through the GOSSIP_NOTIFY message. The keys of the map are the data types of type int, and the values are lists of subscribed APIConnection.
- **unverified_p2p_connections:** A deque that holds information about the current P2P connections that are not yet verified by solving the Proof of Work Puzzle. These can be either connections initiated by the current peer or by other peers. It has a maximum size of "degree" [3.1].
- **p2p_connections:** A deque that holds information about all the current verified P2P connections in the form of P2PConnection class instances [1.2.6]. It has a maximum size of "degree" [3.1].
- **unvalidated_announces:** A map to hold information about announces received from other peers through the PEER_ANNOUNCE message. The keys of the map are the message_ids of type int, and the values are lists of 5 values: ttl of type int, data_type of type int, data in bytes, sender of type P2PConnection and subscribers which is a list of APIConnection. This map is mainly used to hold data until it is validated by all api connections that subscribed to its data_type. A message is only propagated further if all subscribers validated it, or it is discarded if one subscriber said it is not valid.
- **cache:** A deque to hold the hash values of data received from other peers through the PEER_ANNOUNCE message. If a message is received that is already in the cache it will be discarded and not propagated further to other peers on api connections that are subscribed to its data_type.

Each of these data structures also have their own locks to ensure that only one coroutine can modify or read it at a time to avoid race conditions.

Running this class after initializing it will then also initialize and run both the APIServer Class [3.3] and the P2PServer Class [3.4]

3.3. APIServer Class

This class is responsible for running a TCP server on the specified API address and port in the config file [3.1]. It then listens for incoming connections and creates an instance of the APIConnection Class [3.5] for each new connection it receives and adds it to the list of api_connections in the Gossip instance [3.2].

3.4. P2PServer Class

This class is responsible for running a TCP server on the specified P2P address and port in the config file [3.1]. It then listens for incoming connections and creates an instance of the P2PConnection class [3.6] for each new connection it receives and adds it to the list of unverified_p2p_connections in the Gossip instance [3.2].

After starting the server, this class is also responsible for initiating a connection to a bootstrapping node configured in the config file [3.1]. Moreover, this class will also send a PEER_DISCOVER message [2.5] to all verified connected peers every certain amount of time if the current peer does not already have a full list of connections. This certain amount of time is configured as discovery_cooldown in the config file [3.1].

3.5. APIConnection Class

This class is responsible for listening for messages received from a single API connection. It is then responsible for parsing the message, validating its format, and handling the message depending on the message type according to the API and P2P protocols specified earlier in [1] and [2]. The connection will be dropped if a message received is malformed or is of an unknown type.

3.6. P2PConnection Class

This class is responsible for listening for messages received from a single P2P connection. It is then responsible for parsing the message, validating its format, and handling the message depending on the message type according to the API and P2P protocols specified earlier in [1] and [2]. The connection will be dropped if a message received is malformed, is of an unknown type or if a certain message is received at the wrong time (ex: when connection is not yet verified).

If an instance of this class is created when another peer is connecting to us (not a connection that the current peer is initiating), the class will start by sending the PEER_INIT message [2.1].

4. Running The Program:

Follow the following steps to run the program without docker:

1) Make sure you have python 3.9 or above installed on your machine
2) Run the following command from the main directory of the project to install the required dependencies:
   python3 -m pip install -r requirements.txt
3) Run the following command to start the program (the -c argument is optional and will default to config.ini):
   python3 run.py -c config.ini
4) You can run multiple nodes on different terminals with different config files that have different api and p2p addresses/ports.

Follow the following steps to run multiple nodes of the program with docker compose:

1) Make sure you have docker installed on your machine
2) (Optional) Edit the docker-compose.yml if needed to add/remove nodes or to change their config file path (through environment -> conf). Make sure the ports and ipv4 address matches the config files.
3) Run the following command to build a docker image (should be run every time config files are modified):
   docker build -t peer .
4) Run the following command to start docker compose using the image created:
   docker-compose up