VoidPhone Project Specification

P2PSEC (IN2194), SoSe 2021
Chair of Network Architectures and Services
TUM

Sree Harsha Totakura Lukas Schwaighofer
Richard von Seck Prof. Dr.-Ing Georg Carle
p2psec@net.in.tum.de

June 15, 2021

1. Project Description

The aim of the VoidPhone project is to build an **anonymous and unobservable VoIP** application using a P2P architecture. For the system to provide anonymity to voice communication, the users of the system have to keep their peer running also during the times when they are not communicating. In those times their peers participate in the network by relaying data of other peers and by sending cover (fake) traffic. All the peers contribute their bandwidth (for routing other peers traffic), storage (for storing other peers' data), and computational power (needed for (de)encryption of data while routing traffic). We assume that all the peers have public IP addresses and are reachable from any other public IP (not behind NAT/or a firewall). We also assume that all peers run the same version of the developed software but may have differing configurations.

The underlying design of this project stems from Drac[1] which is based on APFS[3]. Our project builds on these ideas by having the following features/assumptions:

- Information is spread in the network using gossip. The project now has an experimental DHT module which can also be used for storing data.
- Time in the system is divided into equal intervals. We call each interval a *round*.
- Each peer participating in the network has a public-private key pair referred to as *hostkey*. The hostkey is generated with RSA and is 4096 bits long. It is stored on disk in the PEM format. The SHA256 hash of the public key gives the *identity* of the peer.
- A user is assumed to run a single peer at any time. Therefore at any moment a user is reachable through one peer.
- A peer is assumed to have a static network address as long as it is online. If a peer's network address changes, we consider that peer as going offline and coming back online with a new address.
- Users of the system share their hostkeys (the public keys) out-of-band.

Briefly, VoidPhone works as following:

1. Peers join at the beginning of a round. If a peer joins after the round, it is considered as online for the next round.
2. A new peer initially knows some of the existing peers. It can get this information from a bootstrapping service, or it is manually configured by the user while starting the peer. The new peer then connects to these peers, and asks them to give it some of the peers they know of. This way, the new peer acquires partial of knowledge of existing peers.
3. The new peer then establishes connections to some of the peers it knows.
4. Peers exchange information about existing peers using gossip.
5. At the beginning of each round, peers randomly choose two peers whom they already know of. They then attempt to build an onion tunnel through them.
6. If a peer has to make a call, it has to wait until the next round. It then uses the tunnel established in that round to call the destination.
7. If a peer is not actively communicating during the round, it will send cover traffic with characteristics of VoIP traffic through the tunnel. The cover traffic will be visible to the last hop in the tunnel and is then echoed back by it.

8. For a user to be able to receive calls the calling party should be able to know the network address of the user's peer. For this, the user's peer advertises through gossip its network address. [This information is encrypted with the private key corresponding to the hostkey, so that only the peers having the corresponding public key are able to decrypt this information. The information will contain, as part of the header information, the peer's identity (the hash of the public key), so that other peers can easily identify the data packet]

The project is divided into the following key modules. Detailed description of these modules is available in Section 4.

Gossip: This is the module responsible for spreading information in the network. Peers spread information that a user is online via this module. Other modules may base their functionality on this module; a mockup version of this module is provided as part of the testing module.

DHT: This module should implement a distributed hash table. This module should form a structured overlay.

NSE: The Network Size Estimation module should provide an estimate of the number of currently active peers. This estimate helps RPS to determine any skew in the distribution of sampled peers.

RPS: This module samples peers randomly (hence the name Random Peer Sampling) from those learnt by the gossip module. It is the responsibility of this module to ensure that the sample peers are uniformly distributed over currently online peers.

Onion: This module implements onion tunnels. It is responsible for building an onion tunnel to a destination using randomly chosen intermediate peers. It may use the RPS module for finding the random peers which will then be selected as hops in the onion tunnels.

CM/UI: This module does call management and interfaces with the user. Call management negotiates calls, *e.g.*, reject any other calls if the user is already in a call; signal the other party that the user rejected to take up a call, etc. Additionally, it interfaces with the sound subsystem of the operating system to capture and play audio frames corresponding to the call. **This module is not available as a sub-project for the teams to work on**.

Testing: This module ensures the conformance of the above modules' API. It contains dummy implementations for all the modules conforming to this specification. **This module can be downloaded from** `https://gitlab.lrz.de/netintum/teaching/voidphone_pytest.git` after successful team registration. To ensure its deployment on heterogeneous platforms it is developed in Python. You are advised to use it during the development of your modules to verify if your module is in conformance with this specification.

1.1. Configuration

Configuration for each of the modules developed by the sub-projects is administered through a single configuration file. This approach allows a module to read the configuration of another module when it is required. For example, Gossip module can get the address and port number on which it has to listen for API connections; later on, NSE module can lookup where the Gossip module is listening for API connections from the configuration and connect to that address. Additionally, this architecture also has the advantage to enable transparent injection of proxies for API communication and test API compliance of various modules.

All modules developed in the sub-projects have to accept a path to the configuration file via command-line parameter `-c`. They should then parse the configuration file to obtain relevant configuration parameters. The standard parameters in the configuration for each module are given in their respective sub-project description. In addition to the standard parameters you are free to define your own parameters.

The configuration file follows the Windows INI configuration file format. This is a simple format which defines parameter-value pairs with one pair per line, and groups the pairs into sections. Consider the example below:

```
hostkey = /hostkey.pem

[gossip]
cache_size = 50
max_connections = 30
bootstrapper = p2psec.net.in.tum.de:6001
p2p_address = 131.159.15.61:6001
api_address = 131.159.15.61:7001

[onion]
hops = 2
...
```

Here, the section `gossip` has parameters `cache_size`, `max_connections`, `bootstrapper` and, `p2p_address`; the section `onion` has the `hops` parameter.

The options `p2p_address` and `api_address` specify the network address where the module listens for P2P connections and API connections respectively. `api_address` should be adhered by all modules. The network address is given as `<hostname>:<port>`. IP address could be used in place of the hostname. With IPv6 addresses, we will use '[' and ']' to separate the address from port number, as in `[2001:4ca0:2001:11:226:b9ff:fe7d:84ed]:6001`. Apart from this options, you are free to decide which options go into the configuration section corresponding to your module.

All module-specific section names and option names should be in lower case.

2. Scope of Work

We understand that the effort required to complete this project exceeds the course requirements for a single student. For this reason students are encouraged to form teams of two. However, you can also choose to work on the project alone.

This document contains the API specification for each modules. The requirements and features that are to be implemented by the modules are listed in Section 4.

2.1. Reporting

Reporting for this project is accomplished through an initial report and a midterm report towards the end of the lecture period. They allow us to evaluate your progress and provide feedback regarding your design choices. Participation in the reporting process is **mandatory** to pass the project. Note that reporting is different from (code) documentation for the final submission of your project.

Both initial and midterm report are to be submitted by placing them in the `docs/` folder in the repositories allocated to you. For submission deadlines please refer to the Section 3.

The initial report should document your implementation approach to the chosen module. It should outline the chosen programming language and reasons for its choice, your development environment, libraries you plan to use and why.

The midterm report should document changes to your assumptions of the initial report. Furthermore it should introduce your module's design as well as the employed P2P protocol. If there are additional changes to the design presented in the midterm report, these should be outlined in the final project documentation.

Workload justification: We acknowledge that the project workload differs from student to student based on the choices they make. For example, if there is no compatible cryptography library available for the programming language a team has chosen, it could be a considerable effort to write wrappers for a library available for another language. Also, a student in a team could spend more time in designing the protocols or on reporting than on coding. Such differences are dealt individually and for us to evaluate correctly, we ask you to report the activities you have chosen to do and the time spent for each of them. We believe that this leads to a fair evaluation of your effort.

2.1.1. Initial Report

Before submitting your report please check if you have covered all the points in the following checklist:

1. Team name, names of team members and module, which implementation you have chosen to work on.
2. Chosen programming language and operating system and reasons for choosing them.
3. Build system to use in the project
4. Intended measures to guarantee quality of your software
   - How do you write testcases
   - Quality control (e.g. Valgrind, cppcheck, ...)
   - ...
5. Available libraries to assist you in your project
6. License you intend to assign your project's software and reasons for doing so
7. Previous programming experience of your team members, which is relevant to this project
8. Planned workload distribution

2.1.2. Midterm Report

Before submitting your report please check if you have covered all the points in the following checklist:

1. Changes to your assumptions in the initial report
2. Architecture of your module
   - Logical structure (classes, ...)
   - Process architecture (threading, multi-process, ...)
   - Networking (AsyncIO, goroutines, ...)

- Security measures (encryption/auth, ...)

3. The peer-to-peer protocol(s) that is present in the implementation
   - Message formats (as given in this document)
   - Reasoning why the messages are needed
   - Exception handling (Churn, connection breaks, corrupted data, ...)
4. Future Work: Features you wanted to include in the implementation but couldn't finish so far
5. Workload Distribution — Who did what
6. Effort spent for the project (please specify individual effort)

2.1.3. Project Documentation

For the final submission, we expect you to include documentation for your finished project. The project documentation is part of the final project grade. It should include the following points:

1. Latest version of your architecture / structure as introduced in the midterm report
2. Software Documentation
   - Building / running dependencies
   - How to install and run the software
   - Known issues
3. Future Work
4. Workload distribution — Who did what
5. Effort spent for the project (please specify individually)

2.1.4. Miscellaneous

Additionally, we will open a submission activity in Moodle where you can describe your experience with the project. This supports our grading process regarding the individual grades of team members. The following points should be covered:

1. How was the communication between you and your team member
2. Whether you felt you and your team partner are equally knowledgeable concerning the project
3. Whether you felt there was an unequal work distribution. If so, please state some objective facts for verification if any.
4. Any remarks/complaints. We will use this as feedback for next year.

3. Project Organisation

Git repositories will be created for each team to work on during the course of this project. It is advised that the teams commit their work into repositories allocated to them frequently. This helps in assessing the effort spent and also serves as a backup. After the project deadline the master branch of the repository is considered for evaluation. To avoid frequent merge conflicts, team members are encouraged to employ the following workflow while using the repositories:

1. Each team member starts his/her own working branch (possibly named after their lastname). This branch is to be used exclusively by that team member; the other team member should not rely on this branch, ever. So, if a member works on multiple computers (a latop and a desktop), he can use this branch to sync his work.
2. Each team member always commits his work to his working branch first.
3. The current branch is then changed to master and any new incoming commits from the remote are pulled into it.
4. The working branch is then rebased on top of the master branch.
5. The master branch is then fast-forward merged with the working branch.
6. The master branch is then pushed to the remote.

Please practice this workflow to get used to it and only proceed if you have understood it. Complaints about other team member reverting/overwriting your commits are frowned upon; and so are your computers exploding and/or your harddrive dying as you were about to push your work from last month. Please commit and push your changes regularly — if you are shy about making the master branch dirty, commit and push to your working branch.

3.1. Group Formation & Registration

You have to implement a TCP client which implements a simple protocol similar to that one used for API connections in the project. Using this client you will then have to connect to a server to register your group. The specification for this TCP client is described in Section 4.1.

3.2. Deadlines

1. Registration: 1. May 00:00 CEST
2. Initial Report: 31. May 23:59 CEST
3. Midterm Report: 12. July 23:59 CEST
4. Written exam: 20. July 08:00 - 09:30 CEST
5. Project Deadline: 12. September 23:59 CEST

4. Modules

This section documents the functionality of the modules. A special nature of this project is that it does not enforce any programming language/operating system requirement for developing the module. However, since we expect that the developed modules interface to one another, an interface is defined for every module in the form of a network communication protocol that the module has to adhere to. We refer to a module's interface and its Application Programming Interface (API). The API lies at the application layer of the OSI stack and uses TCP connections.

The modules could be seen as layers similar to those in OSI networking stack. For example, the RPS module samples random peers from the peer updates it learns from the Gossip module; the Onion module relies on the RPS module to acquire random peers which it can use to build onion tunnels.

Each module on a peer is expected to communicate with other modules of the peer. Some modules may need to communicate with their counterparts on other peers. For example, the Gossip module of a peer connects with the Gossip modules of peers who are its neighbours. Similarly, the Onion module connects to the Onion modules of other peers while building onion tunnels. The CM module on the caller peer, despite not having to connect to the CM module of the callee peer directly, will still need to communicate to the callee's CM module through the tunnel. We distinguish between these two type of connections by terming the former as *API connection* and the latter *P2P connection*.

API protocols are the communication protocols for API connections. These are specified for each module in this document. The protocols define message types and message formats. Similarly, communication protocols for P2P connections are termed *P2P protocols*. As a module developer, you have to design your module's P2P protocol. For data serialization you may use e.g. Protocol Buffers[1], Cap'n Proto[2], Binary JSON[3], or come up with your own protocol. Note that the implementation of the P2P protocol should be abstracted from the module's API.

Each module is to be implemented as a process which listens on a network socket for accepting API connections from other modules. This approach helps in two ways: 1. the modules could be programmed in any language and could be running on any operating system which supports network communication; 2. any bugs in a module leading to segmentation faults or crashes could be isolated since only the involved processes crash. The IP address and the port number on which the listening socket of a module is opened is determined by the `api_address` option in configuration.

It is a common assumption in P2P systems that misbehaving or malicious peers exist in the network. The core of P2P research lies in developing algorithms to tolerate the existence of such peers without making neither our peer nor the system vulnerable to them. We apply this to our project, too, and thus assume that all other peers we connect to or which connect to us could be either be malicious or misbehaving. Misbehaving peers may not adhere to protocols strictly, and/or may crash randomly. Malicious peers may drop messages, store and replay them, and/or modify them. Furthermore, they may exploit any vulnerabilities in protocols/implementations to crash or attack peers, *e.g.*, exploit buffer overflow vulnerabilities due to the usage of `strcpy`. The same applies for the module API communication, although it could be assumed here that the risk is mainly due to programming errors and crashes in the other module. Thus, care must be taken while designing and implementing the P2P protocols to make a module resistant towards such attacks. For example, modules may drop connections if they observe any discrepancy in the P2P protocol or after a response timeout. You are encouraged to come up with defence

[1] Google Protocol Buffers: `https://developers.google.com/protocol-buffers/`
[2] `https://capnproto.org/`
[3] `http://bsonspec.org/`

mechanisms for protection against these attacks.

Messages in all API protocols are restricted to 64KiB − 1 = 65535B in length and start with a header which contains a field to indicate the length of the message (including the header) and the type of the message. All integer fields are encoded in network-byte order (Big-endian). None of the fields in the API messages contain a floating-point number. The format of the message header is displayed in Figure 1.

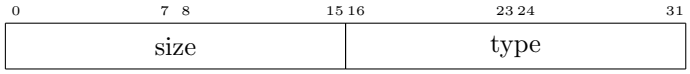

Figure 1: Message header format

Message types for the messages involved in the API have a globally unique number. To avoid confusion, it is advised that you do not use any of these numbers in your module's P2P protocols.

Bit-level formats for various messages involved in the module APIs are given in the following subsections. For illustration purposes (see Figures 1, 7), a message is split into 32 bit fields which are represented as a row. We call each such row a *word*. Words are further splitted to accommodate fields which are smaller than 32 bits. In that case, a vertical line is used inside the word to mark the beginning and ending of the field. Longer fields which typically encompass multiple words are illustrated by a space encompassing upto 3 words. The length of the field is then mentioned after the fields name. Variable length fields are illustrated by drawing a vertical cut through a similar space.

All the modules have the following common configuration parameters:

1. `api_address`: Used to specify the IP address or hostname and a port number to which the module must bind to for accepting API connections. See Section 1.1 for the format to be used.

Additionally, every configuration file will have the global configuration parameter `hostkey` to specify the hostkey of the peer.

4.1. Group registration

This module needs to be completed by everyone who wishes to take part in the course. The project teams are registered using this protocol. You need to implement a TCP client which connects to `p2psec.net.in.tum.de:13337`. As soon as the TCP connection is established the server initiates the communication by sending you the following message.

4.1.1. ENROLL INIT

This message is sent by the server after you have connected to it. It will contain a 64 bit challenge which is to be used in calculating a response for the subsequent `ENROLL REGISTER` 4.1.2 messages sent by the client.

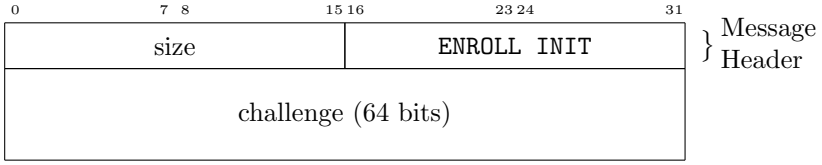

Figure 2: ENROLL INIT message format

Upon receiving this message the client should begin to compute a response according to the following protocol: Pack the challenge, your email, firstname, lastname, LRZ Gitlab username, team number, and project choice according to the structure shown in 3.

The team number is represented as an unsigned integer. If you are the first one in the group registering, set the team number to 0; the server will then allocate a team number for your group. If you do not have a team partner and would like to be randomly paired with another partner set the team number to 65535.

Project choice should take one of the following values:

- Gossip: 2961
- DHT: 4963
- RPS: 15882
- NSE: 7071
- Onion: 39943

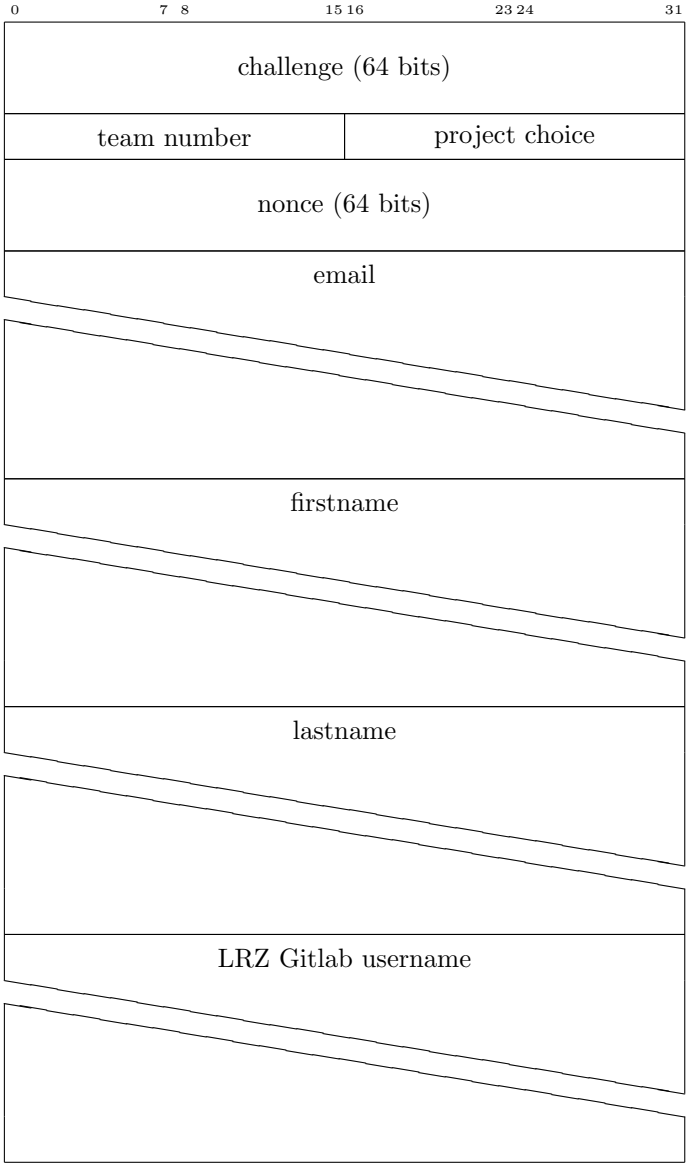

Figure 3: Packaging of team number, email, firstname, lastname, gluser and random nonce

Email, firstname, lastname and LRZ Gitlab username should be encoded with UTF-8 encoding. They are concatenated together with CRLF (carriage-return, line-feed, i.e "`\r\n`") as delimiter. The LRZ Gitlab username handle should be provided without the `@`, e.g. if your username is `@ga42hax`, you should provide `ga42hax` as username. Make sure to login at least once into LRZ Gitlab to activate your account before enrolling via this challenge. If you changed your default Gitlab handle, please provide your current handle instead.

With the nonce set to a random value, calculate the SHA256 hash (with padding in compliance with RFC 6234) of the structure show in Figure 3. If the resulting SHA256 value has the first `24` bits set to 0 then you can register by sending the `ENROLL REGISTER` (See: 4.1.2) message. If not, the nonce has to be changed to another random value and retried until one is found which gives one of the required SHA256 values. Note that SHA256 is considered a pseudo-random function. This means that there is no other possible way to find such values apart from randomly retrying them.

4.1.2. ENROLL REGISTER

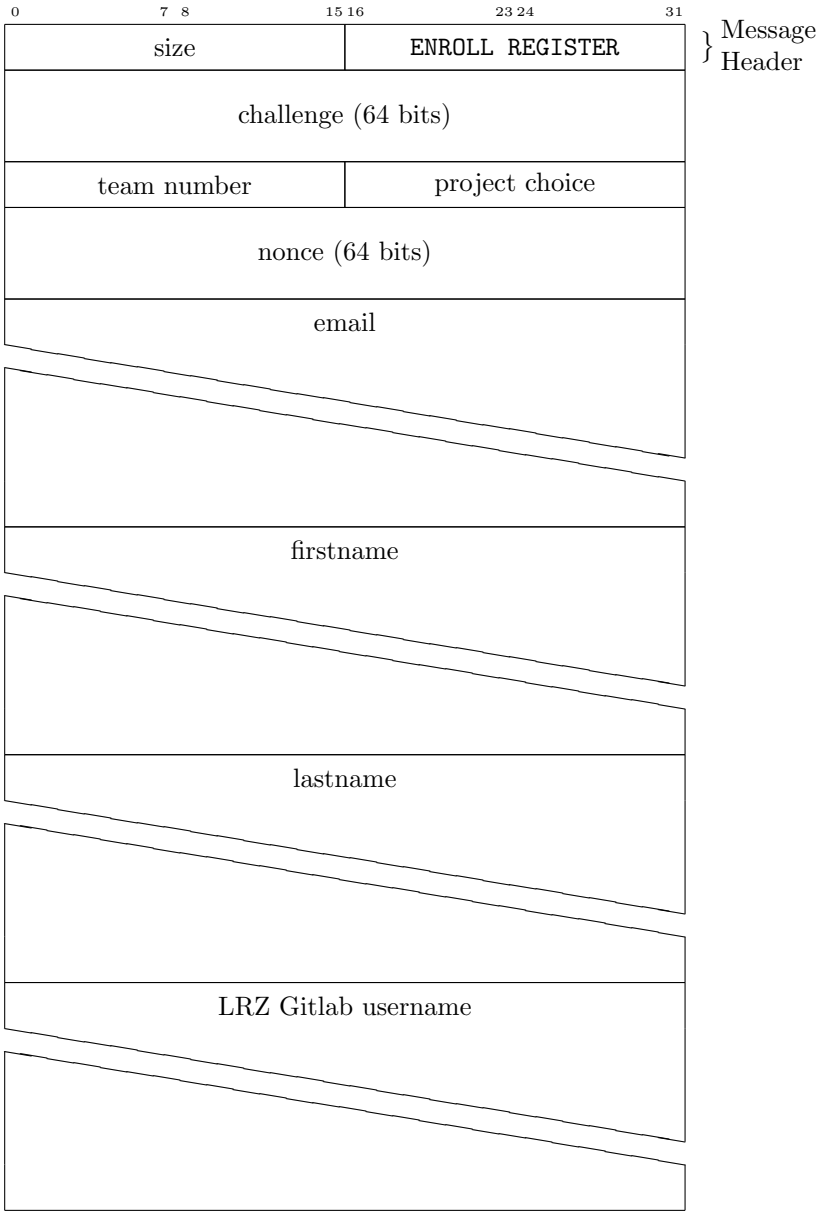

Figure 4: Packaging of team number, email, firstname, lastname, gluser and random nonce

4.1.3. ENROLL SUCCESS

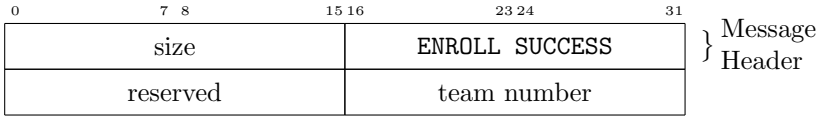

Figure 5: ENROLL SUCCESS message format

The team number allocated to you is given in an `ENROLL SUCCESS` message (see Figure 5). If you are the first member to register (have previously set team number to 0 in `ENROLL REGISTER`) you should give this team number to your partner and ask him to register with it (by setting the team number in `ENROLL REGISTER` to this value).

4.1.4. ENROLL FAILURE

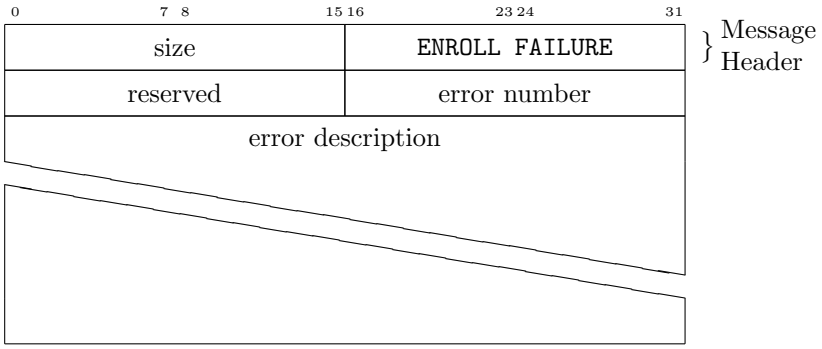

Figure 6: ENROLL FAILURE message format

If something goes wrong, the server will send you an `ENROLL FAILURE` message, as given in Figure 6. **NOTE:** Make sure to read the given error description string, to find out what is wrong with your code.

4.2. Gossip

This module aims to disseminate local knowledge of a peer and maintain membership of the peers in the network. Peers exchanges their local knowledge with other peers in the network. These peers could be immediate neighbors or random peers. The knowledge may contain arbitrary data; for example, the network addresses of available peers. This functionality is also used by the RPS, NSE and CM/UI modules for disseminating their knowledge. For example, RPS may use this layer to learn about new peers in the network; CM/UI may use this layer to spread the information that its user is now online and is ready to receive calls.

Gossiping is prone to Sybill and, Eclipse attacks: if you are bootstrapping through an attacker peer, the attacker peer can give you the network addresses of peers controlled by her, effectively eclipsing your peer by creating a lot of sybill peers to which your peer may connect to. These attacks could be countered by enforcing Proof-of-Work-based peer identities or by generating the peer identities based on network addresses.

The configuration for Gossip contains the following parameters:

- `degree`: Number of peers the current peer has to exchange information with
- `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.

The API for the Gossip layer has the following messages:

4.2.1. GOSSIP ANNOUNCE

Message to instruct Gossip to spread the knowledge about given data item. It is sent from other modules to the Gossip module. No return value or confirmation is sent by Gossip for this message. The Gossip should put in its best effort to spread this information.

The TTL field specifies until 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 protocols, whereas the data type is used to identify the application data Gossip spreads in the network.

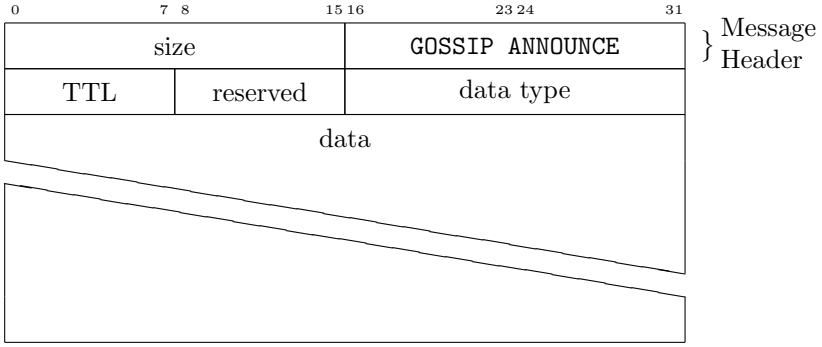

Figure 7: GOSSIP ANNOUNCE message format

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.

4.2.2. GOSSIP NOTIFY

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 given type is received by it. The new data message could have been received from another peer or by another module of the local peer. The caller of this API will be notified by the Gossip through `GOSSIP NOTIFICATION` messages (See Section 4.2.3). 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. This means only messages for which a module has registered a notification from Gossip will be propagated by Gossip.

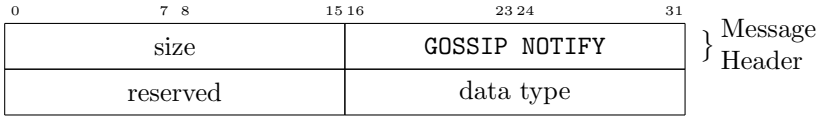

Figure 8: GOSSIP NOTIFY message format

This message does not evoke any response from Gossip. The notification is cancelled 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.

4.2.3. GOSSIP NOTIFICATION

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 conveyed its interest using `GOSSIP NOTIFY` message.

Message ID is a random number to identify this message and its corresponding response.

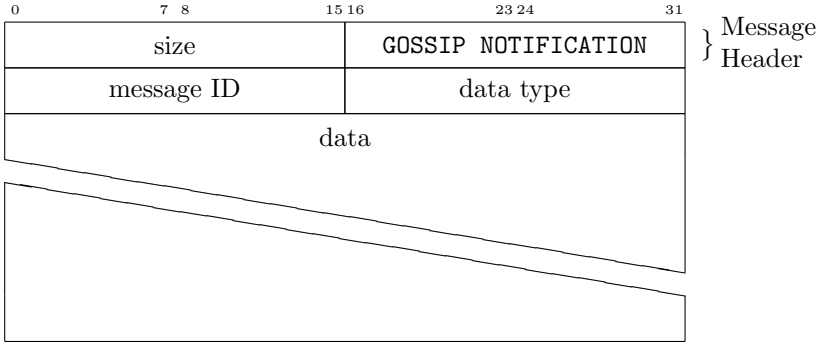

Figure 9: GOSSIP NOTIFY message format

The module receiving this message has to reply with `GOSSIP VALIDATION` message to signify whether this message is well-formed or not. Well-formedness of a message could be defined as whether the mesage's data is according to the expected format, its fields make sense, any digital signatures in it verify OK or not. Gossip then propagates this message to other peers if it is well-formed.

4.2.4. GOSSIP VALIDATION

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 the the notification is well-formed; otherwise it is to be deemed invalid and hence should not be propagated further.

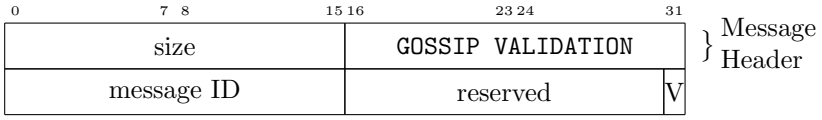

Figure 10: GOSSIP VALIDATION message format

4.3. DHT

A distributed hash table (DHT) is a distributed storage for key-value pairs. An implementation of a DHT should form a structured overlay to optimize the placement and retrieval of the key-value pairs. Some designs for DHT are the Chord, Pastry, Kademlia, and GNUnet's R5N. You can choose to implement any of them or your own design.

A DHT implementation should support storing a key-value pair and retrieval of the value given the corresponding key at any later point. The key-value pair can expire after a predefined timeout. To ensure the availability of the key-value pair, the peer storing/caring for that pair needs to refresh it by storing it at regular intervals.

In this specification the keys are all 256 bit in length. Your implementation may use shorter or longer lengths. To account for churn your implementation should support redundancy while storing a key-value pair.

4.3.1. DHT PUT

This message is used to ask the DHT module that the given key-value pair should be stored. The field TTL indicates the time in seconds this key-value pair should be stored in the network before it is considered as expired. Note that this is just a hint. The peers may have their own timeouts configured which may be shorter than the value in TTL. In those cases the content could be expired beforehand. The DHT does not make any guarantees

about the content availability, however it should exercise best effort to store it for that long. Similarly, the replication field indicates how many times (by storing the content on different peers, or under different keys, etc) this content should be replicated. This value should also be treated as a hint. The DHT may choose replicate more or less according to its parameters.

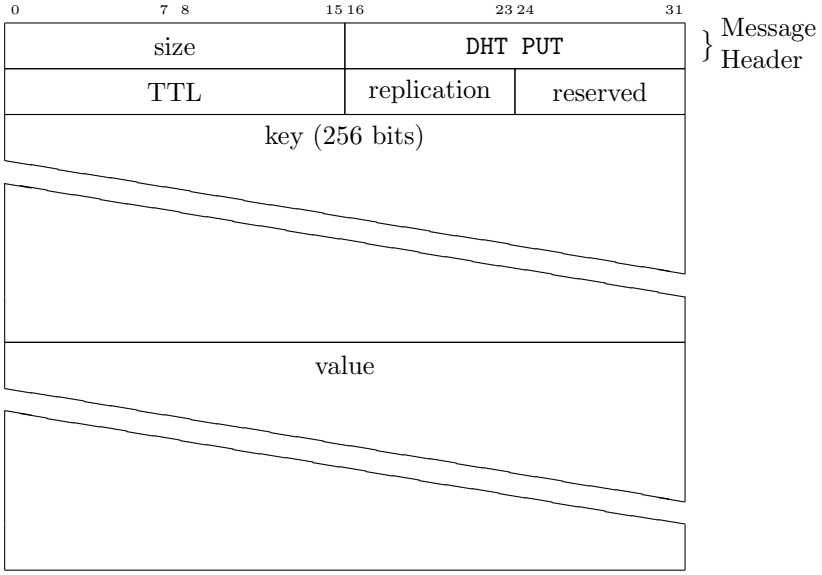

Figure 11: DHT PUT message format

It is expected that the DHT module upon receiving this message does its best effort in storing the given key-value pair. No confirmation is needed for the PUT operation. Request of and storage of empty values is not allowed.

4.3.2. DHT GET

This message is used to ask the DHT method to search for a given key and provide the corresponding value, if it can be found in the network.

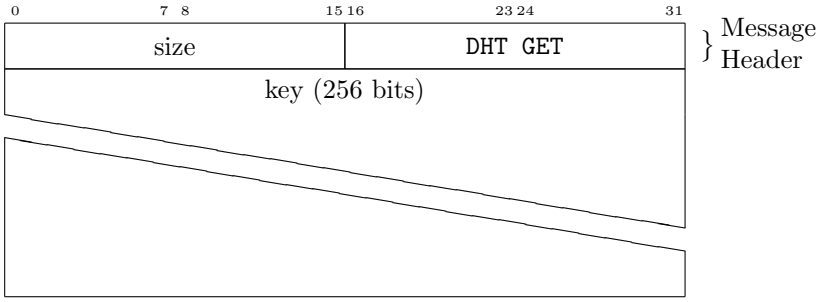

Figure 12: DHT GET message format

No immediate is reply is expected after sending this message to the DHT module. The module should however start with its best effort to search for the given key.

4.3.3. DHT SUCCESS

This message is sent when a previous DHT GET operation found a value corresponding to the requested key in the network.

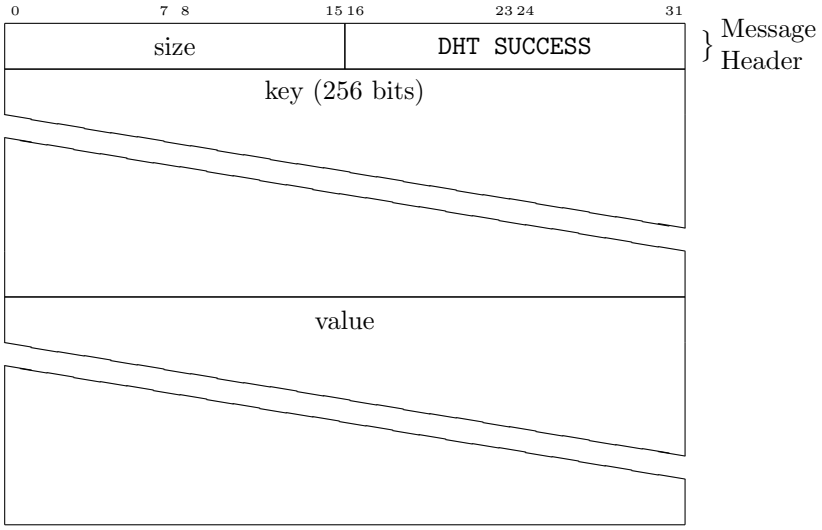

Figure 13: DHT SUCCESS message format

4.3.4. DHT FAILURE

This message is sent when a previous DHT GET operation did not find any value for the requested key.

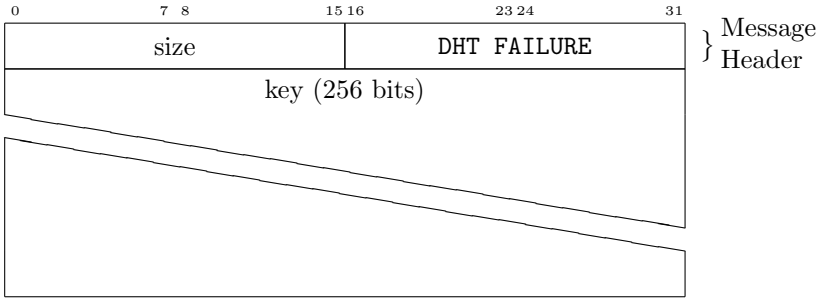

Figure 14: DHT FAILURE message format

4.4. NSE

Estimating the size of the network is valuable for many P2P applications and defence mechanisms. In our application, we can derive a good value of TTL. As a defence, we can infer if our sample of random nodes are random or if there is any skew due to an attacker carrying out an eclipse attack on us.

The task of approximating the size of the network, *i.e.* the number of peers participating in the network at any time is given to the NSE module. Our reliance on Gossip module means that algorithms for estimating network size based on structured overlays cannot be used easily without the NSE module implementing its own structured overlay. However, other algorithm such as Gossipico [4] and GNUnet's NSE [2] could be used here.

The API for the NSE module is simple and contains a single message for which the NSE module replies with an estimated value of the network size.

4.4.1. NSE QUERY

Query the NSE module for current estimate. After receiving this message on a API connection the NSE module should reply with its current estimate.

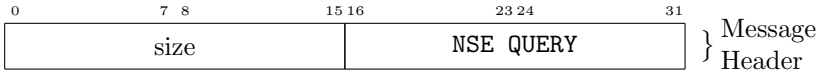

Figure 15: NSE QUERY message format

Upon receiving this message NSE responds with `NSE ESTIMATE`. Only one response is sent per query message.

4.4.2. NSE ESTIMATE

This message is sent as a response to the `NSE QUERY` message by the NSE module. The fields *estimated peers* and *estimated std. deviation* contain the estimated number of peers and the standard deviation of the current estimate from its previous values. The format of the message is shown in Figure 16

For the calculation of the standard deviation, the NSE module should maintain a history of previous estimates which it should have calculated periodically. The length of this history and the periodicity should be configurable in the configuration.

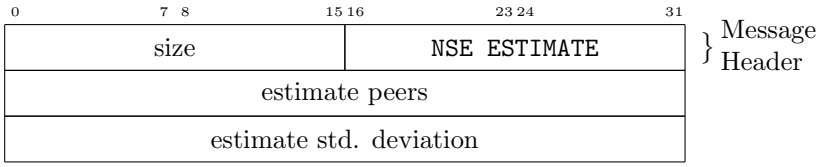

Figure 16: NSE ESTIMATE message format

The NSE module does not solicit any response for this message.

4.5. RPS

This module is geared towards helping us find peers at random. The implementation of this module could implement their own P2P protocol for spreading peer membership information or rely on Gossip to spread the information. Note that this choice is determined by the type of algorithm in use. In both cases, this module should store the identifiers and network addresses of peers it learns into persistent storage for later usage.

The RPS module may interface with the NSE module to obtain an estimate on the size of the network. Several peer sampling algorithms rely on this estimate to detect certain attacks and also to work optimally.

The API for RPS module is simple and consists of two messages.

4.5.1. RPS QUERY

This message is used to ask RPS to reply with a random peer. This message is short and consists only of the header. The format is shown in Figure 17.

Figure 17: RPS QUERY message format

4.5.2. RPS PEER

This message is sent by the RPS module as a response to the RPS QUERY message. The format is shown in Figure 18. It contains the peer identity and the network address of a peer which is selected by RPS at random. The version of the network address is specified by the flag `V`; it is set to 0 for IPv4, and 1 for IPv6. In addition to this it also contains a portmap for the P2P listen ports of the various modules on the random peer. The RPS module of a peer should get the listen port addresses from its configuration.

The field `#portmap` contains the total number of portmap records. Each port record is of 4 bytes long; the first 2 bytes identifies the module (shown in the figure as `App`), the next 2 bytes contain the listen port number of that module. The modules are identified by the following encoding:

- DHT: 650
- Gossip: 500
- NSE: 520
- Onion: 560

RPS should sample random peers from the currently online peers. Therefore the peer sent in this message is very likely to be online, but no guarantee can be made about its availability.

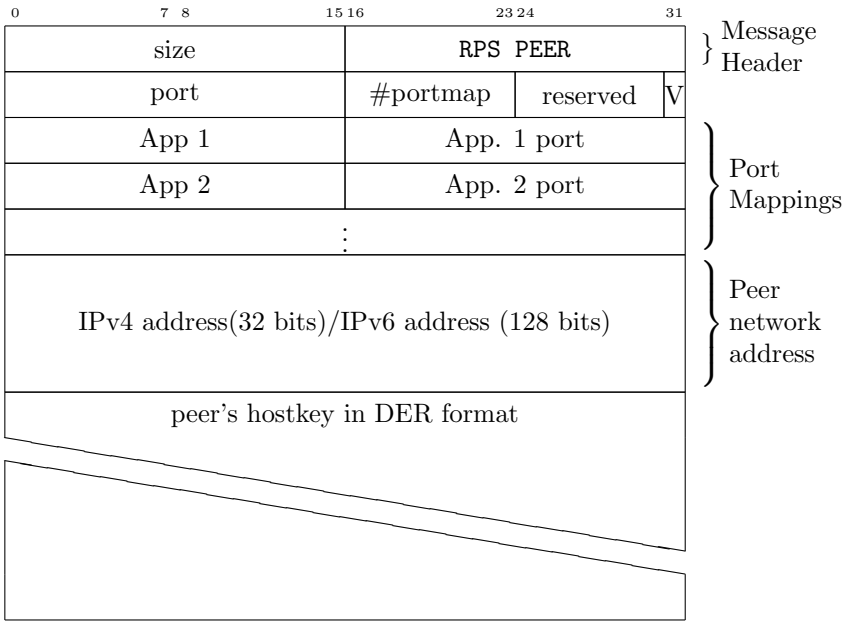

Figure 18: RPS QUERY message format

4.6. Onion

The Onion module deals with constructing onion tunnels for tunnelling data streams from CM/UI module. In our application, at the beginning of each round the Onion module on each peer should start constructing a tunnel with at least two intermediate hops which are selected at random. This means that on an average each peer can expect to be used as an intermediate hop by at least two other peers. The destination of a tunnel could be the call recipient or a random peer. This is decided in a previous round by the CM/UI module. If the CM/UI module has asked Onion to create a tunnel to a peer, it will be this peer; if not, it will be a random peer selected by the Onion module with the help of RPS module. The actual round duration is arbitrary but fixed between all peers, using the same Onion module. Configuration could be realized by adding another configuration file option.

The Onion module should read its peer's hostkey from the file given in the configuration. It should then listen on a predefined network socket and should accept P2P connections from Onion modules of other peers from this socket. The Onion modules of two peers use their respective hostkeys to form an ephemeral session key which they then use to encrypt all communication between them.

Tunnel building is done iteratively, i.e., a peer O building a tunnel selects the first hop peer H_1 at random (from RPS), it then forms an ephemeral session key K_1 with the peer. It then uses K_1 to encrypt necessary meta-data to H_1 to instruct it to connect to the next hop peer H_2 and establish an ephemeral session key K_2. Note that K_2 is generated by the hostkeys of O and K_2, so that H_1 does not learn about it. The process continues until the tunnel has enough number of hops and the destination peer is added.

While building the onion tunnels there is a chance that attacker peers are picked at random. Building a tunnel through them has the disadvantage that if all of the peers in the tunnel are attacker peers or co-operate among themselves, our communication can be de-anonymised. Thus, the more peers participate in the tunnel the smaller the risk, as a single honest peer in the tunnel is enough to scramble your communication. On the other hand, having more peers increases the latency of the communication and may effect the call quality. For this reason, we restrict to a minimal hop count of 2 hops in a tunnel (the 3rd hop being the destination). Combined with cover traffic, small duration of rounds, and end-to-end encryption between the caller and the callee, even with attacker peers in the tunnel, the attacker still has to overcome the dilemma whether the traffic observed is cover traffic or real traffic. The actual hop count may be parameterised via configuration.

Since we are building an application employing anonymity, it is important to defend against certain types of traffic analysis attacks which make use of packet sizes to analyze communication, even though it is encrypted. For this reason, the Onion to Onion communication should employ packets of fixed sizes. The size of the packet is up to the module developer to decide. It is wise to choose neither of a too big nor too small value. This also means that packets should be padded accordingly at each hop to maintain their fixed size. The packet size should be strictly enforced by the protocol — any peer deviating from this should be disconnected immediately.

The tunnels the Onion modules build may have unrealiability metrics similar to that of the UDP protocol. This means that the Onion modules put their best effort in forwarding the packets but no guarantees need to be made on packet delivery: packets may be lost, delayed arbitrarily and/or received out of order. If a module requires TCP-style reliability for the tunnelled data, the module has to employ its own protocol, *e.g.*, by using a userspace TCP stack.

Tunnels created in one period should be torn down and rebuilt for the next period. However, Onion should ensure that this is done transparently to the modules, using these tunnels. This could be achieved by creating a new tunnel before the end of a period and seamlessly switching over the data stream to the new tunnel once at the end of the current period. Since the destination peer of both old and new tunnel remains the same, the seamless

switch over is possible.

The configuration for Onion should have the following standard parameters:

- `p2p_port`: This is the port for Onion's P2P protocol *i.e.*, the port number on which Onion accepts tunnel connections from Onion modules of other peers. This is different from the port where it listens for API connections. This value is used by the RPS module to advertise the socket the onion module is listening on, so that other peers' onion modules can connect to it.
- `p2p_hostname`: Similar to `p2p_port` this parameter determines the interface on which Onion listens for incoming P2P connections.

4.6.1. ONION TUNNEL BUILD

This message is to be used by the CM/UI module to request the Onion module to build a tunnel to the given destination in the next period. The message is identified by `ONION TUNNEL BUILD` message type. The version of the network address is specified by the flag `V`; it is set to 0 for IPv4, and 1 for IPv6. See Figure 19 for the message format.

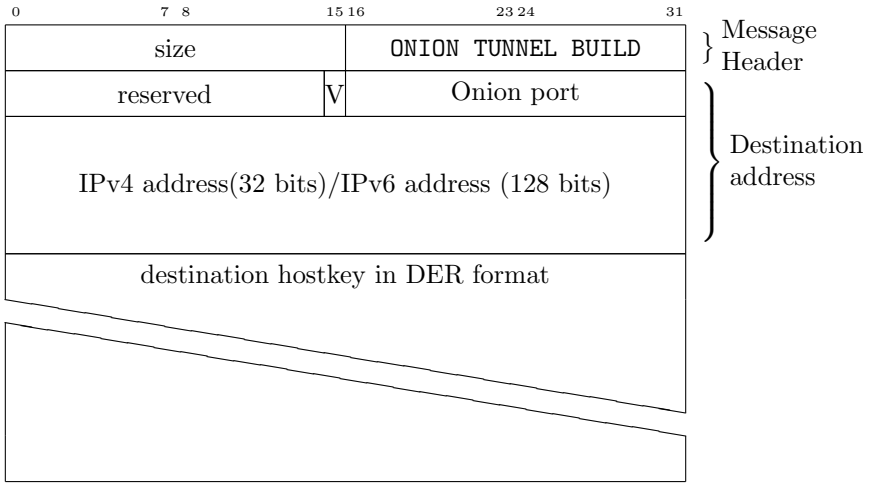

Figure 19: ONION TUNNEL BUILD message

The message contains the identity and network address of the destination.

4.6.2. ONION TUNNEL READY

This message is identified by the message type `ONION TUNNEL READY` and is sent by the Onion module when the requested tunnel is built. The recipient of this message is allowed to send data in this tunnel after receiving this message. It contains the identity of the destination peer and a tunnel ID which is assigned by the Onion moduel to uniquely identify different tunnels. See Figure 20 for the message format.

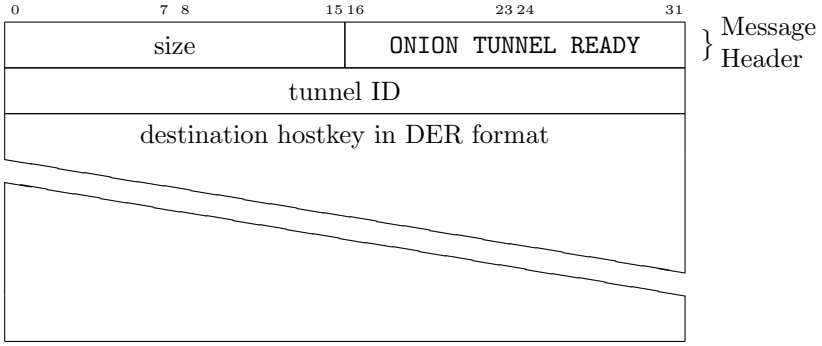

Figure 20: ONION TUNNEL READY message format

4.6.3. ONION TUNNEL INCOMING

This message is sent by the Onion module on all of its API connections to signal a new incoming tunnel connection. The new tunnel will be identified by the given tunnel ID. The format of this message is shown in Figure 21.

No response is solicited by Onion for this message. When undesired, the tunnel could be destroyed by sending `ONION TUNNEL DESTROY` message.

Incoming data on this tunnel is duplicated and sent to all API connections which have not yet sent an `ONION TUNNEL DESTROY` for this tunnel ID. An incoming tunnel is to be destroyed only if all the API connections sent a `ONION TUNNEL DESTROY` for it.

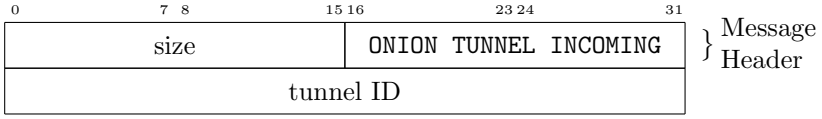

Figure 21: ONION TUNNEL INCOMING message format

4.6.4. ONION TUNNEL DESTROY

This message is used to instruct the Onion module that a tunnel it created is no longer in use and can now be destroyed. The message is identified by the message type `ONION TUNNEL DESTROY`. The tunnel ID should be valid, *i.e.*, it should have been solicited by the Onion module in a previous `ONION TUNNEL READY` or `ONION TUNNEL INCOMING` message.

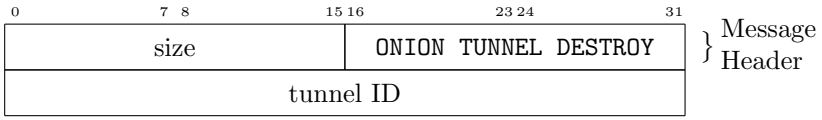

Figure 22: TUNNEL DESTROY message format

4.6.5. ONION TUNNEL DATA

This message is used to ask Onion to forward data through a tunnel. It is also used by Onion to deliver data from an incoming tunnel. The tunnel ID in the message corresponds to the tunnel which is used to forwarding the data; for incoming data it is the tunnel on which the data is received.

For outgoing data, Onion should make a best effort to forward the given data. However, no guarantee is given: the data could be lost and/or delivered out of order.

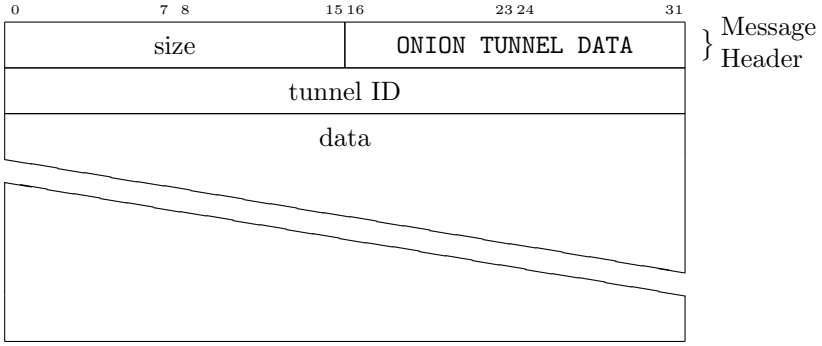

Figure 23: ONION TUNNEL DATA message format

4.6.6. ONION ERROR

This message is sent by the Onion module to signal an error condition which stems from servicing an earlier request. The message contains the tunnel ID to signal the failure of an established tunnel. The reported error condition is not be mistaken with API violations. Error conditions trigger upon correct usage of API. API violations are to be handled by terminating the connection to the misbehaving client.

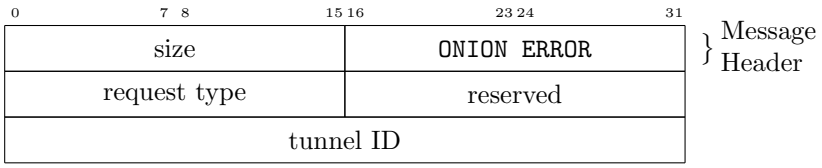

Figure 24: ONION ERROR message format

The message is identified by the type `ONION ERROR` and contains the message type and the tunnel ID used in the request to which error message corresponds to.

4.6.7. ONION COVER

This message identifies cover traffic which is sent to a random destination by the Onion module. The CM/UI module uses this message to fabricate cover traffic, mimicking the characteristics of real VoIP traffic. Upon receiving this message, the Onion module should send the given amount of random bytes on the tunnel established to a random destination in a round. It is illegal to send this message when a tunnel is established and Onion has replied with `ONION TUNNEL READY`.

Cover traffic that is received by a tunnel endpoint via your P2P protocol, needs to be echoed back to the initiator to ensure a bidirectional data flow akin to real VoIP traffic.

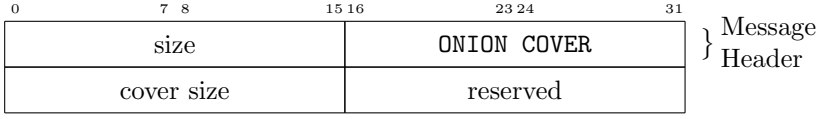

Figure 25: ONION COVER message format

4.7. CM/UI

The expected functionality of the CM/UI module is that it provides an interface for the user for initiating and receiving calls. This could either be a graphical or command line interface. The CM/UI module should also integrate with the underlying operating system's audio interface, *e.g.*, PulseAudio [4] for GNU\Linux, to play the received audio stream from the other communication party and to record the local microphone input stream to send it to the other party. Additionally, CM/UI module needs to compress the audio packets using a frame size of less than 30ms to keep the end-to-end latency low. On the receiver side, the CM/UI has to reorder the received stream packets, decompress the frame and send the audio samples to the playback device.

Call management usually involves a control and data channel. In RTP for example, RTCP is used for control whereas RTP is used for streaming voice/video content. The control channel is used for signalling call initiation, rejection, busy, waiting, termination, etc., and also for establishing a session key with the call receiver in case of end-to-end encryption.

The control channels are usually realised over TCP due to its reliability guarantees, whereas the data channels are realised over UDP to achieve less latency. In our project the tunnels built by the Onion modules do not provide guaranteed packet delivery. This should be taken into consideration while designing the protocol for call management because a reliable transport is not provided by Onion.

A simple protocol in this case would be to start sending the voice stream to the destination irrespective of whether the destination has picked up the call or not. The destination CM/UI module would then see one of the packets belonging to the voice stream and will consider as an incoming call unless it has already accepted another call, in which case, it will reply with a noise stream. A more sophisticated way is to combine both data stream and signalling. An example is to have a fixed 64-bit field in the packets of voice stream for signalling.

It is optional for the CM/UI module to perform End-to-End encryption of the call streams since the onion module ensures that the tunnelled data is delivered with End-to-End encryption.

For users to know whom of their friends are currently online, the CM/UI module must announce presence information on the network using Gossip and also listen for presence messages from Gossip. See Gossip API 4.2 for how this can be done. The presence message should contain the identity of the peer and the network address of the peer's Onion module. It may contain other fields required to prevent replay attacks and may be signed by the peer's hostkey.

Adversaries snooping on users' Internet connection can observe when a user is using the system by observing the outgoing/incoming bandwidth on the link. The adversary can then use this information to perform confirmation attack (where the adversary can confirm whether a suspected pair of users are communicating). To mitigate such attacks cover traffic is to be employed. This traffic's characteristics such as packet size, frequency of packets, etc., should mimic the characteristics of a call stream. Since the characteristics of the call stream are determined by the CM/UI module, it is also its responsibility to fabricate the cover traffic. For this reason, it uses `ONION COVER` messages (see Section 4.6.7) to signal Onion module to send a given number of random bytes through the tunnel established in that round.

The CM/UI module does not have an API interface as no further modules are envisioned to use it. This module just needs to interface with the Gossip and Onion modules using their respective APIs.

The configuration of CM/UI as of now does not have any additional standard parameters. Any future parameter requirements will be listed here.

[4] https://wiki.freedesktop.org/www/Software/PulseAudio/

4.8. Testing

The aim of this module is to test and evaluate the system. It provides dummy implementations of the modules so that module developers can start testing their implementations. The module is developed in Python and should be able to run on operating systems where Python VM is available.

This module is available at this repository: `https://gitlab.lrz.de/netintum/teaching/voidphone_pytest.git` For instructions on how to use this module, please refer to the `README.md` in the repository.

A. Message Types

The preceding sections define formats for API messages. The following enumeration defines the utilized message types. All values are given in decimal notation.

500. `GOSSIP ANNOUNCE`
501. `GOSSIP NOTIFY`
502. `GOSSIP NOTIFICATION`
503. `GOSSIP VALIDATION`

⋮

519. *reserved until here for* Gossip
520. `NSE QUERY`
521. `NSE ESTIMATE`

⋮

539. *reserved until here for* NSE
540. `RPS QUERY`
541. `RPS PEER`

⋮

559. *reserved until here for* RPS
560. `ONION TUNNEL BUILD`
561. `ONION TUNNEL READY`
562. `ONION TUNNEL INCOMING`
563. `ONION TUNNEL DESTROY`
564. `ONION TUNNEL DATA`
565. `ONION ERROR`
566. `ONION COVER`

⋮

599. *reserved until here for* Onion
600. `AUTH SESSION START`
601. `AUTH SESSION HS1`
602. `AUTH SESSION INCOMING HS1`
603. `AUTH SESSION HS2`
604. `AUTH SESSION INCOMING HS2`
605. `AUTH LAYER ENCRYPT`

606. `AUTH LAYER DECRYPT`
607. `AUTH LAYER ENCRYPT RESP`
608. `AUTH LAYER DECRYPT RESP`
609. `AUTH SESSION CLOSE`
610. `AUTH ERROR`
611. `AUTH CIPHER ENCRYPT`
612. `AUTH CIPHER ENCRYPT RESP`
613. `AUTH CIPHER DECRYPT`
614. `AUTH CIPHER DECRYPT RESP`

⋮

649. *reserved until here for* Onion Auth
650. `DHT PUT`
651. `DHT GET`
652. `DHT SUCCESS`
653. `DHT FAILURE`

⋮

679. *reserved until here for* DHT
680. `ENROLL INIT`
681. `ENROLL REGISTER`
682. `ENROLL SUCCESS`
683. `ENROLL FAILURE`

⋮

689. *reserved until here for* ENROLL

Note: To avoid confusion, please refrain from using these message types for messages in your P2P protocols if you are using a message format similar to that one specified in this document.

B. Change Log

2021-0.3: Added final project deadline. Released on 15.06.2021.
2021-0.2: Clarifications regarding config option hierarchy. Added bonus exam date. Released on 12.05.2021.
2021-0.1: Clarifications regarding the registration challenge
2020-0.3: More clarifications on submodule implementations and change of enrollment challenge formats (28.10.2020)
2020-0.2: Clarified description of general API message length. Released on 01.07.2020.
2020-0.1: Fixed erroneous description of midterm report content. An additional, faulty enumeration item has been removed. Released on 22.06.2020.
2020-0.0: Restructured project organization for remote work. Updates to references to external resources and bugfixes.
2019-0.0: Modified project organisation. Teams now enhance existing implementations.
2018-2.0: Changed RPS API to include a port map. Released 26, June 2018.
2018-1.0: Changed DHT API to include some new fields. Released 30. May 2018.
2018-0.4: Removed references to Onion Forwarding module as that module's functionality is now merged into onion. Released 22. May 2018.
2018-0.3: Fix project deadlines. Released 17. May 2018.
2018-0.2: Added Enrollment API. Released 19 April 2018.
2018-0.1: Added DHT. Released 12 April 2018.
2018-0.0: Adapted from 2017-4.0. Removed Onion Authentication module. Not released.
2017-4.0: Remove source hostkey from `ONION TUNNEL INCOMING` message as this is a privacy leak. Released 07 August 2017.
2017-3.1: Add flag to `ONION TUNNEL BUILD` to distinguish between v4/v6 IP addresses. Released 19 July 2017.
2017-3.0: Remove sending source hostkey in `AUTH SESSION INCOMING HS1`; else we will be leaking the identity of the initiator. Add 4 new message types (`AUTH CIPHER` ...) to perform encryption and decryption on encrypted payloads. Released 17 July 2017.
2017-2.0: Introduce a new message type `AUTH ERROR` to signal P2P protocol errors at AUTH. Introduce a flag in `RPS PEER` to determine the version of the IP address. Fix configuration example — our config file format does not have parameters without a section name.
2017-1.0: Onion Auth protocol changed to include Request IDs also in `SESSION_START` messages; request IDs are made 32-bit while Session IDs 32-bit. Offsets in some Onion Auth messages changed.
2017-0.1: Add testing module's repository. Fix submission deadlines.
2017-draft-0.0: Adapted from last year's project specification. Not released.
2016-draft-0.1: Project architecture is modified with new modules and message formats. Public draft; released on 19 April 2016.
2016-draft-0.2: Added API for Onion Authentication module. Extended deadline for intial approach document until 4 May 2016. Public draft; released on 26 April 2016.
2016-draft-1.0: Fixed inconsistency in referring to the API socket address; thanks to Daniel Sel for pointing this. Fixed `ONION ERROR` message to contain tunnel ID of the faulty tunnel; also added a reserved field. Added `AUTH LAYER ENCRYPT RESP` message. Public draft; released 09 May 2016.
2016-draft-1.1: Added message format for `AUTH LAYER DECRYPT` and added `AUTH LAYER DECRYPT RESP`, `AUTH SESSION CLOSE`. Changed the semantics of `GOSSIP NOTIFY`, `GOSSIP NOTIFICATION`. Added new messsage `GOSSIP VALIDATION.` Public draft; released 24 May 2016.
2016-draft-1.2: Added `GOSSIP VALIDATION` to the enumeration of messages; Public draft; released 24 May 2016.
2016-draft-2.0: Changed message formats for `RPS PEER`, `ONION TUNNEL BUILD`, `ONION`

`TUNNEL READY`, `ONION TUNNEL INCOMING`, `ONION ERROR`: these messages now contain. Fixed typos in `ONION COVER`, `ONION TUNNEL DATA`. Public draft; released 27 May 2016.
2016-draft-3.0: Fixed `AUTH SESION CLOSE`; previously it was errorneously referred to as `AUTH LAYER CLOSE` and the session ID was only 2 bytes. Public draft; released 28 May 2016.
2015-0.1: Added checklist for final report. Promoted from draft status.
2015-draft-2.1: Added error message types in DHT and KX. Defined message types.
2015-draft-2.0: Changed format for `MSG_KX_TN_READY`; IP address management for tunnels is simplified.
2015-draft-1.2: Introduce configuration in Sections 1,4 and in all module descriptions.
2015-draft-1.1: Added checklist for interim report.
2015-draft-1.0: Address information fields added to DHT TRACE and KX TUNNEL BUILD messages. Refined KX and VoIP description.

References

[1] George Danezis et al. "Drac: An Architecture for Anonymous Low-Volume Communications". In: *Privacy Enhancing Technologies*. Springer. 2010, pp. 202–219.

[2] Nathan S. Evans, Bartlomiej Polot, and Christian Grothoff. "Efficient and Secure Decentralized Network Size Estimation, Tech Report". In: (May 2012).

[3] Vincent Scarlata, Brian Neil Levine, and Clay Shields. "Responder anonymity and anonymous peer-to-peer file sharing". In: *Network Protocols, 2001. Ninth International Conference on*. IEEE. 2001, pp. 272–280.

[4] Ruud Van De Bovenkamp, Fernando Kuipers, and Piet Van Mieghem. *Gossip-based counting in dynamic networks*. Springer, 2012.