Report 2: Interim Report

Jonatan Juhas

1 Architecture

As our module (code-named Garlic) needs to be able to handle 2 scenarios (Sending & Receiving) at the same time, we will implement a simple multi-threaded environment[1]. Sending and Receiving mechanisms will, apart from sharing a single state object containing information about current tunnels, not share other properties which simplifies the setup considerably.

2 Inter-Module protocol

2.1 Internal API specification

In this section, we will briefly explain what messages our module needs, how they look and what their natural flow would look like (that is in what order they arrive/are dispatched).

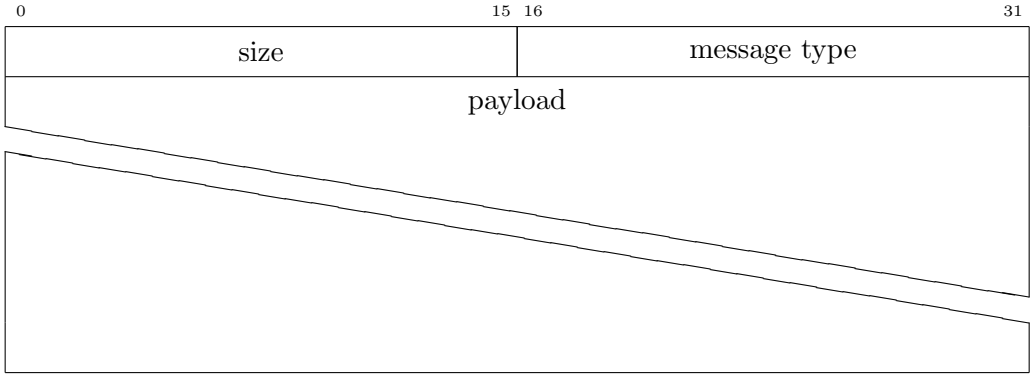
2.1.1 Onion messages

Our module "owns" the following messages:

- ONION TUNNEL BUILD
- ONION TUNNEL READY
- ONION TUNNEL INCOMING
- ONION TUNNEL DESTROY
- ONION TUNNEL DATA
- ONION TUNNEL COVER
- ONION TUNNEL ERROR

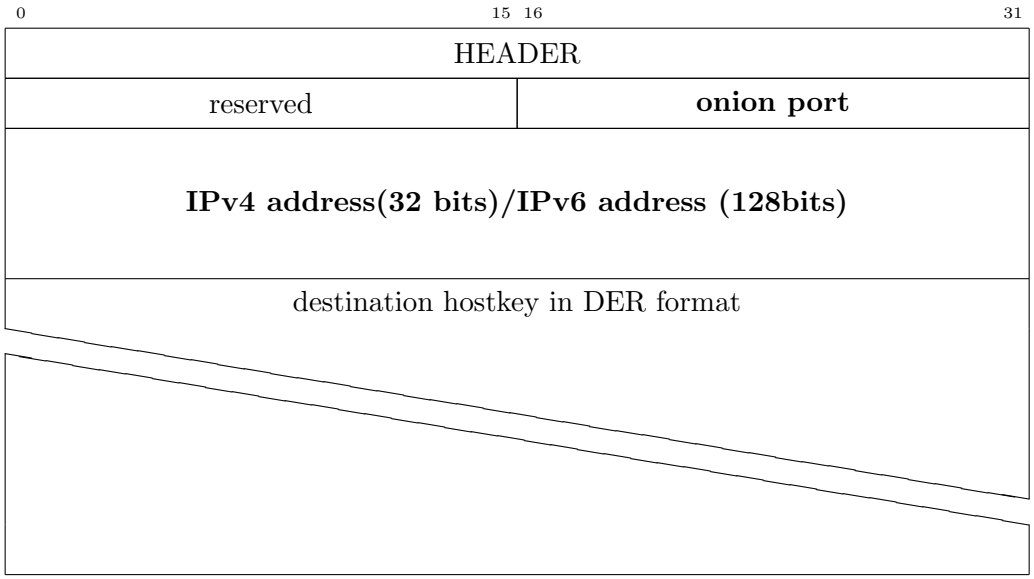
[1] It is potentially possible to fork other threads on every incoming/send request. Might be something to consider for the future

Each message is composed of a **header** and the payload. The header is composed of 32 initial bits which encompass the size (16 bits) and the message type (also 16 bits). It looks like this:

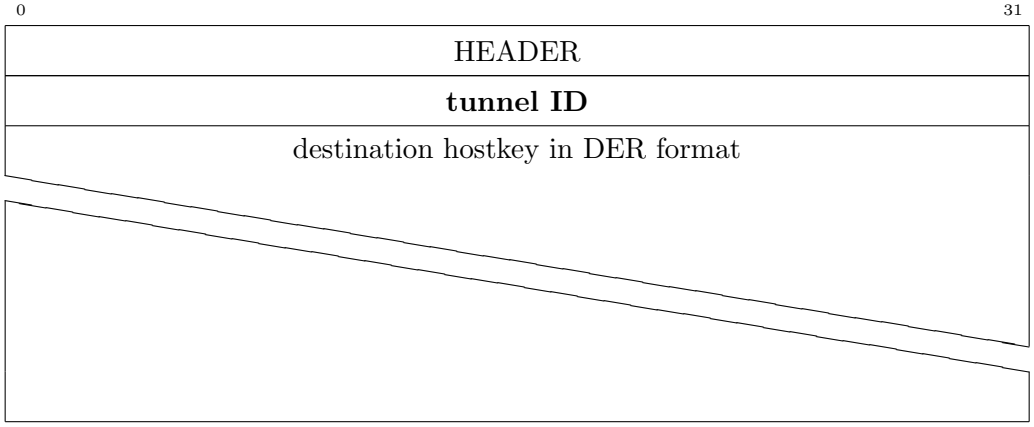


Now for the actual messages:

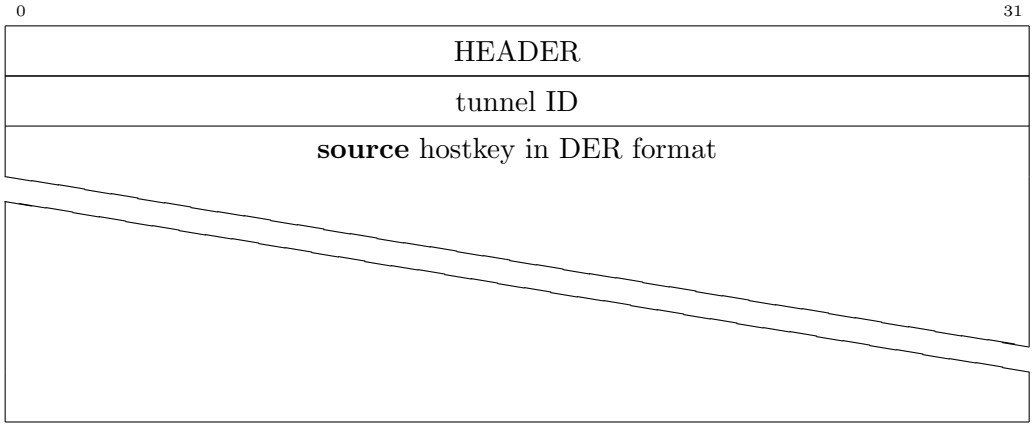
2.1.1.1 ONION TUNNEL BUILD The CM/UI will use this message to request a tunnel be opened in the next round to a specified host. This would be part of the Send thread. The message has the following form:



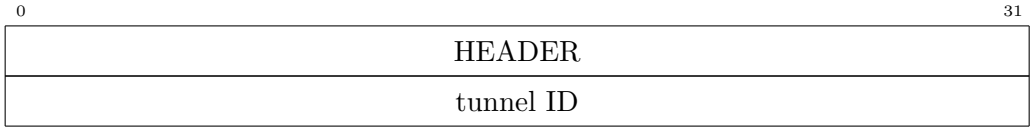
2.1.1.2 ONION TUNNEL READY This is a reply to the **TUNNEL BUILD** message carrying the tunnel ID that has been associated with the established tunnel.



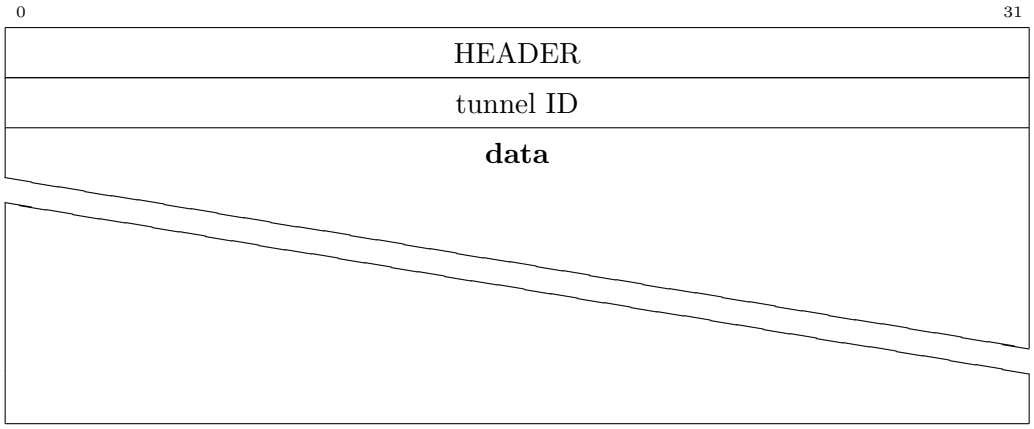
2.1.1.3 ONION TUNNEL INCOMING This is the counterpart of **TUNNEL BUILD** message as viewed by the peer receiving the connection. It indicates there is someone who is trying to connect to us. This message is passed to CM/UI.



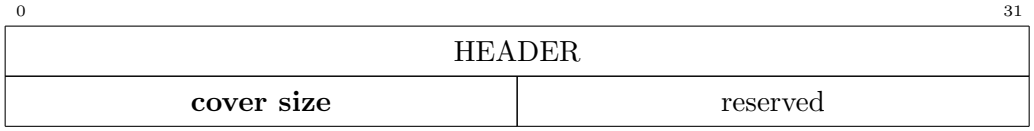
2.1.1.4 ONION TUNNEL DESTROY If a tunnel is no longer needed, this message can be used to notify Garlic it should close it. One other requirement is that the tunnel id **is valid**.



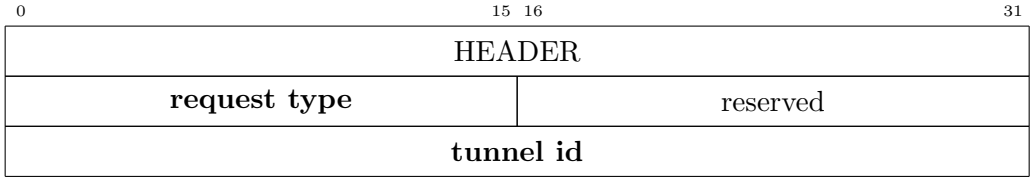
2.1.1.5 ONION TUNNEL DATA Used for forwarding data on the established tunnel. There is no requirement to verify if the data has been transmitted successfully - no 2 way check.



2.1.1.6 ONION TUNNEL COVER The CM/UI will with this message be asked to generate a fake payload of a given size to hide non-existent traffic.



2.1.1.7 ONION TUNNEL ERROR This final message is used to signal various problems that occurred. These do not include breach of API contract errors.



2.1.2 Other required messages

Apart from these messages, we still need a few more to properly integrate with our ecosystem. We will need the RPS messages:

- RPS QUERY
- RPS PEER

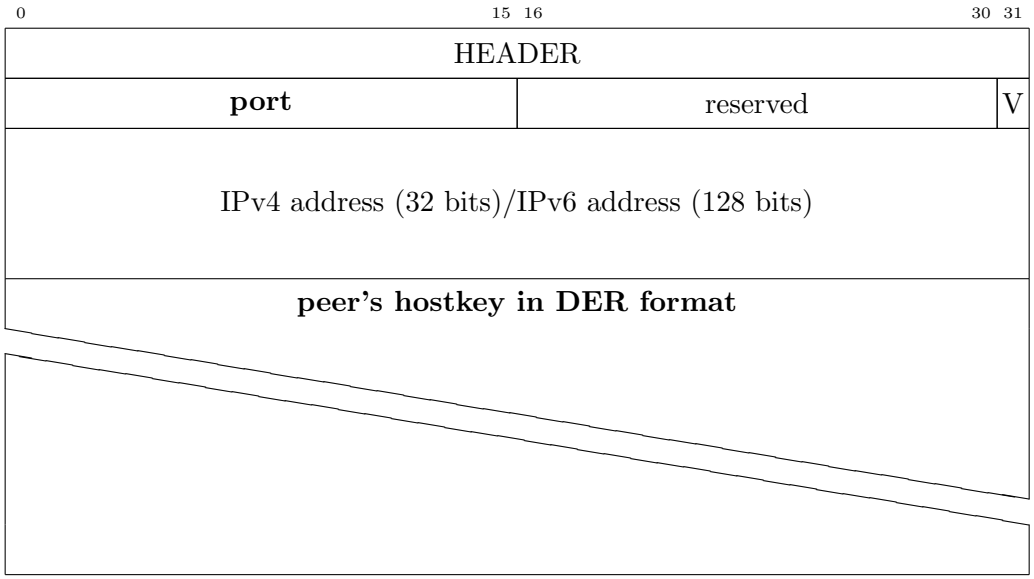
as well as the Onion Authenticate messages:

- AUTH SESSION START
- AUTH SESSION HS1
- AUTH SESSION INCOMING HS1

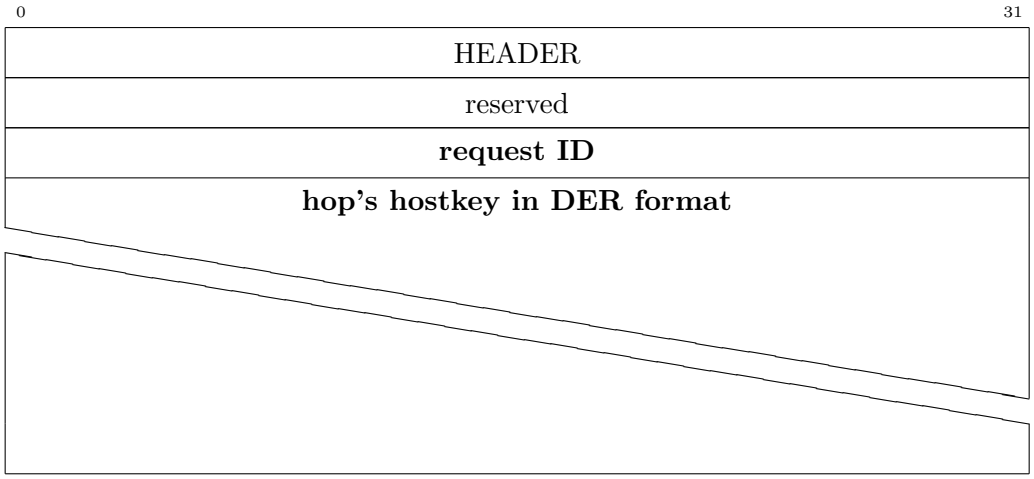
- AUTH SESSION HS2
- AUTH SESSION INCOMING HS2
- AUTH LAYER ENCRYPT
- AUTH LAYER ENCRYPT RESP
- AUTH LAYER DECRYPT
- AUTH LAYER DECRYPT RESP
- AUTH SESSION CLOSE
- AUTH SESSION ERROR

2.1.2.1 RPS QUERY Is composed of only the header. This message requests a random peer address from the RPS.

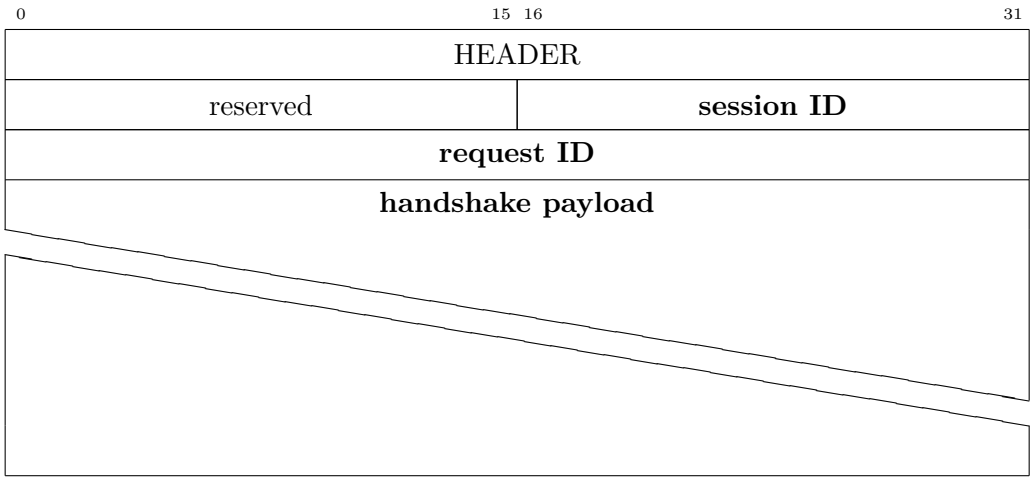
2.1.2.2 RPS PEER Response to **RPS QUERY** which contains a random peer's address. The IP version can be found in the **V field** of the message.



2.1.2.3 AUTH SESSION START This is the first message we need once we try to establish a secure tunnel (which is every tunnel because we like everything secure). The message requires the destination's hostkey in DER format. The Garlic module will use the **request ID** to facilitate mapping messages to their responses. The number can be random or monotonically increasing.

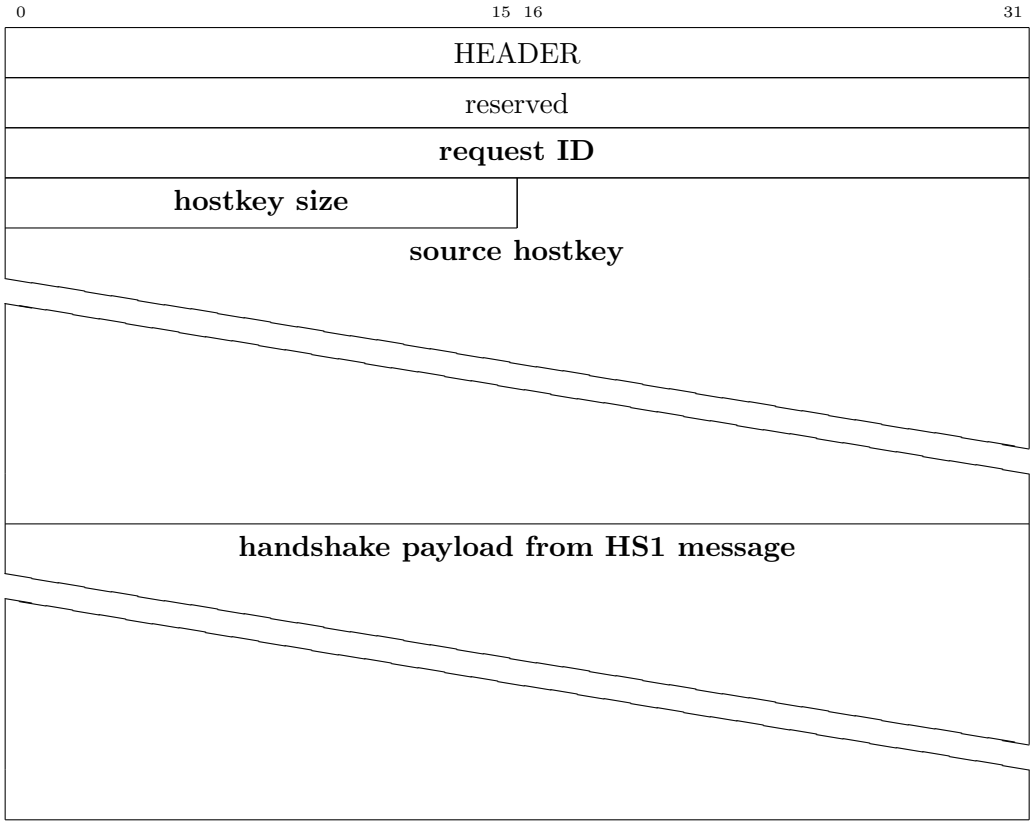


2.1.2.4 AUTH SESSION HS1 This is a reply to **AUTH SESSION START** which carries the **payload** to send to the peer as well as a **session ID** used to identify this tunnel's encryption session along with **request ID**[2]. This message's payload is forwarded to the next hop.

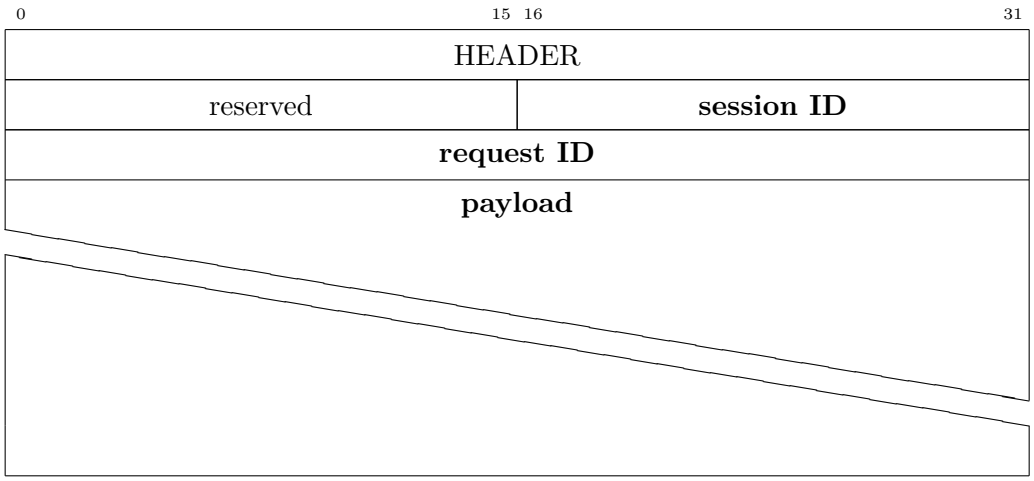


2.1.2.5 AUTH SESSION INCOMING HS1 This is the first *proper* message on the receiver end and is forwarded to Onion Authenticate. It doesn't contain a **session ID** as this is yet to be allocated but it does contain a unique **request ID**.

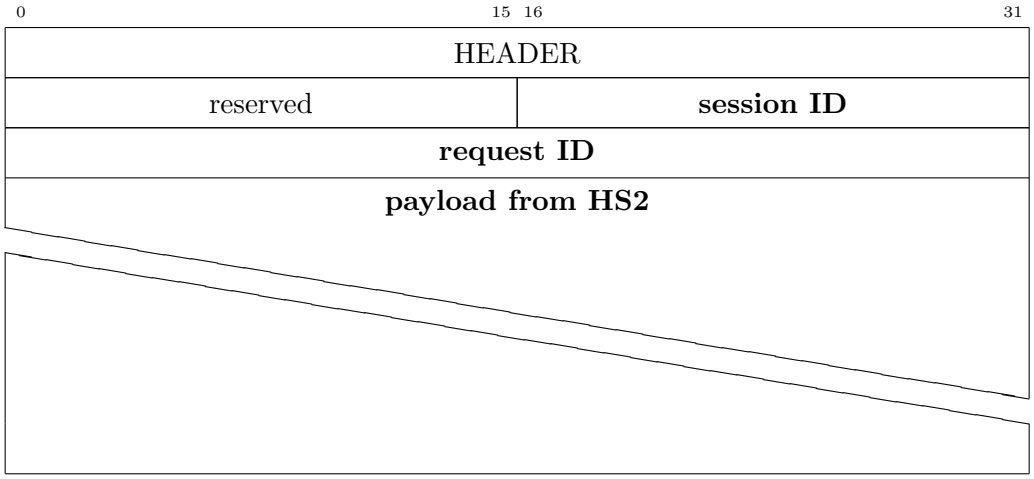
[2]If I understood it correctly.



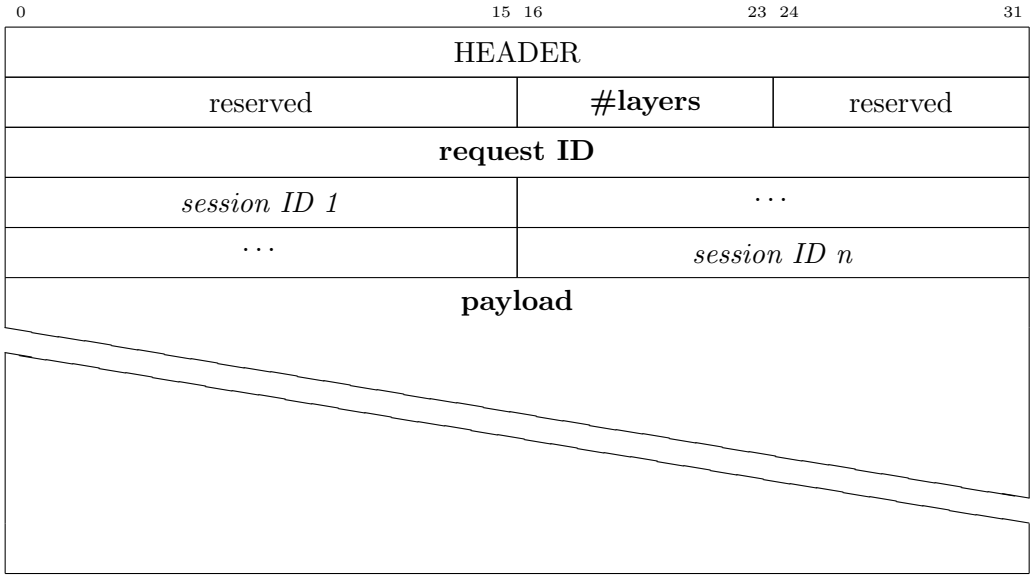
2.1.2.6 AUTH SESSION HS2 This is the reply to **INCOMING HS1**. Session key is now established.



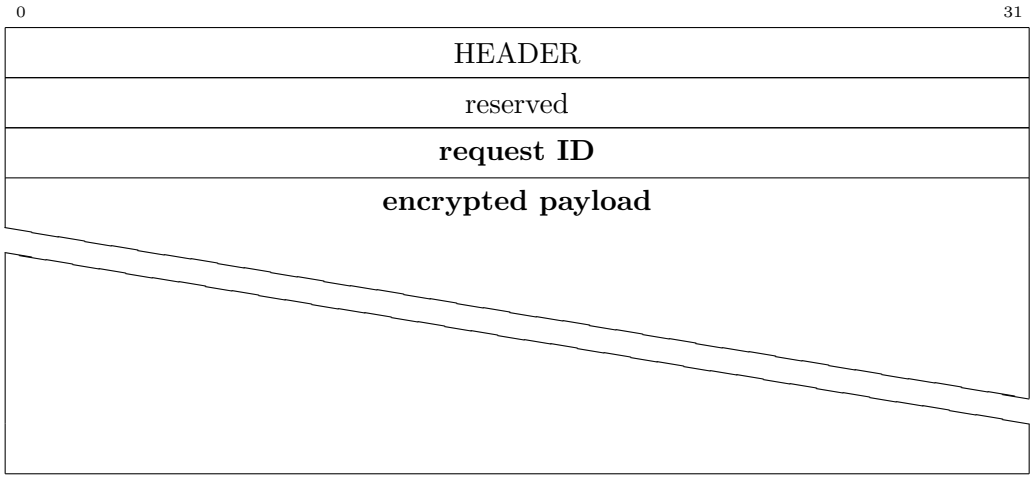
2.1.2.7 AUTH SESSION INCOMING HS2 Complete the session handshake on host.



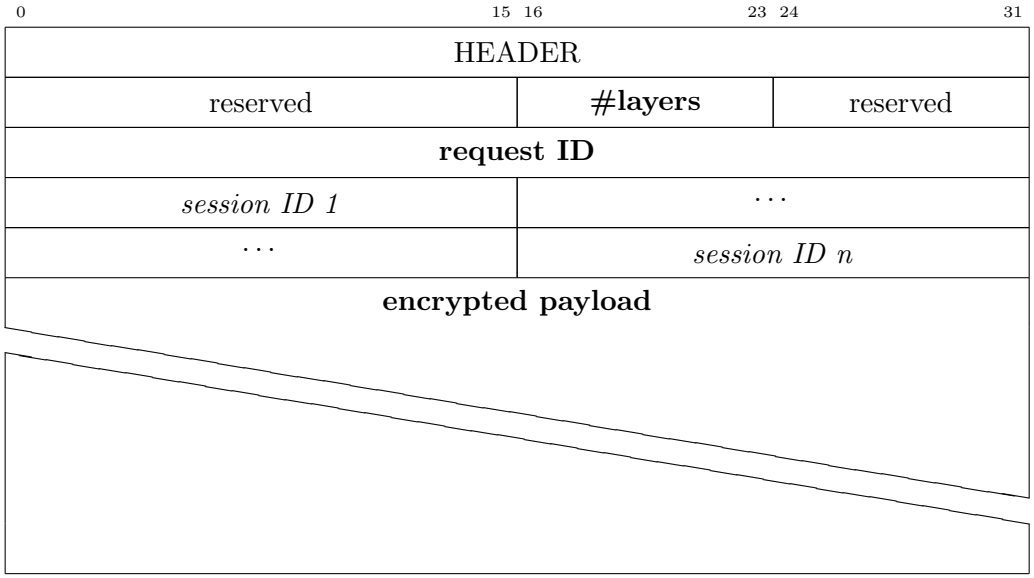
2.1.2.8 AUTH LAYER ENCRYPT This message is used once all session keys have been established, that is a tunnel of a required length has been built. This message contains the number of peers in the **layers** field and all **session IDs** in a row - padded to 1 word - that is why at least 2 session IDs (2 hops) are required. The message looks like this:



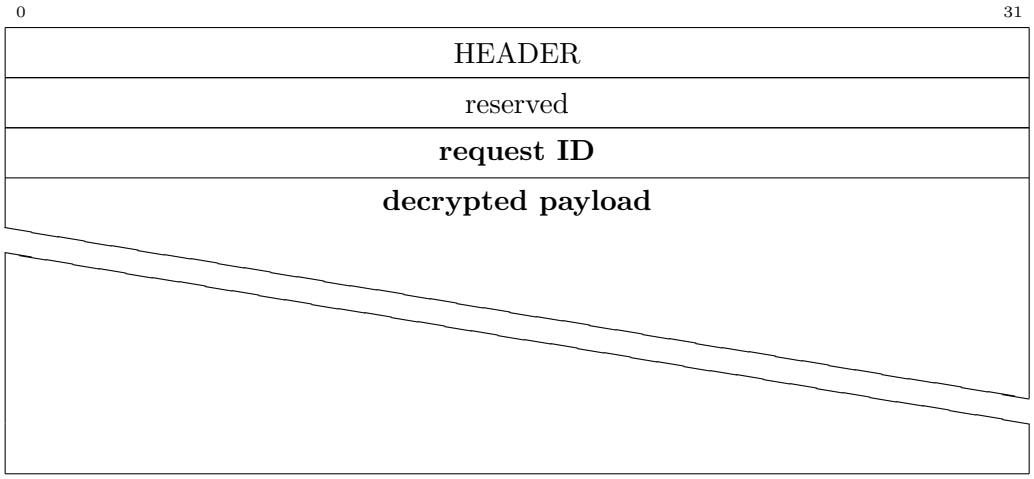
2.1.2.9 AUTH LAYER ENCRYPT RESP This is the response to **ENCRYPT** message. It contains the **encrypted payload** which is forwarded to peers.



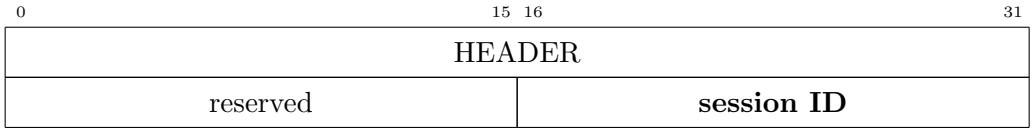
2.1.2.10 AUTH LAYER DECRYPT The peers then pass the **encrypted payload** to Onion Authenticate for decryption.



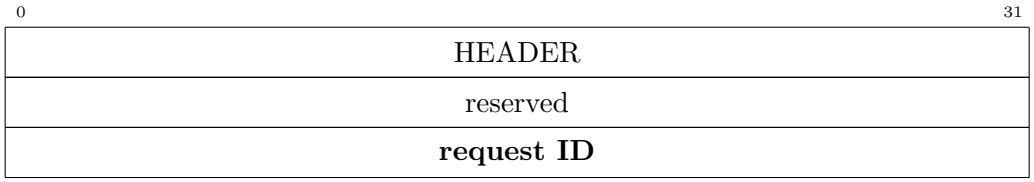
2.1.2.11 AUTH LAYER DECRYPT RESP This is the response to the **DECRYPT** message. It's similar to **ENCRYPT RESP**.



2.1.2.12 AUTH SESSION CLOSE This message signals Onion Authenticate to destroy all session keys that belong to that session.



2.1.2.13 AUTH ERROR This message is sent by Onion Authenticate and means an error occurred somewhere in the P2P Protocol.



Whew - that was a lot, was it not? Let us move on to message flow then ...

2.2 API Flow

Now that we have covered how the messages look like it's time to talk about how and when we use them.

2.2.1 Connection to external peer

Once a connection to a peer was requested by the CM/UI we first find t intermediate hops[3] to connect through using RPS and finally our destination hop. The length t is given by the configuration file in the field minimum_hop_count. You can see how the protocol behaves in

[3]This is indicated in the diagram by the cycling arrow connections with RPS

this case in figure 1. After we gather enough intermediate hops $P_1 \dots P_n$ we start connecting to each one of them in the following way:

1. Connect to the first peer P_1 and form an ephemeral session key K_1 with this peer using Onion Authenticate.
2. Send necessary meta-data encrypted with K_1 to P_1 to instruct it to connect to P_2.
3. During handshake with P_2 form the next session key K_2. The key is encrypted on both host and P_2 in a way P_1 cannot gain access to it as it mediates the traffic.
4. Repeat steps $1 \dots 3$ until enough hops have been connected to the tunnel and the destination node has been reached.

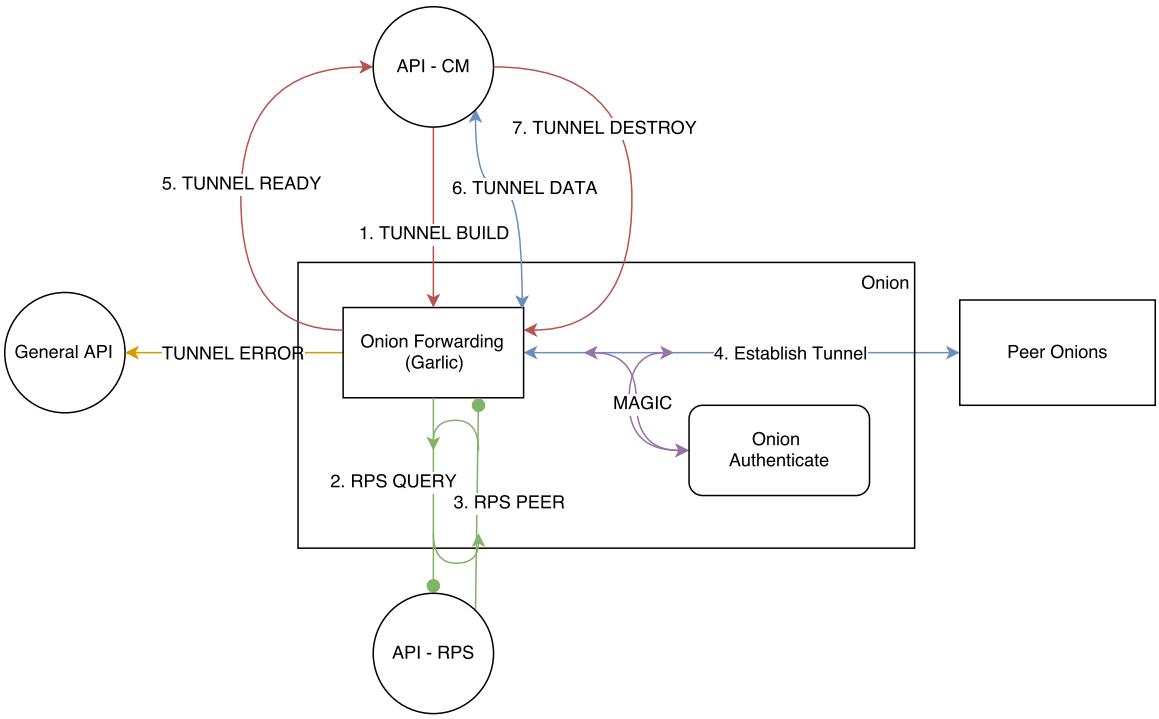


Figure 1: Message flow for connecting to other peers

2.2.2 Receiving connection from other peer

Once the host tries to connect to a destination peer, instead of an intermediate hop, this peer needs to handle the connection and pass data to CM/UI. This is shown in figure 2. The optional destroy message can be sent to Garlic to indicate the CM/UI didn't accept this connection (eg. call has been rejected etc.).

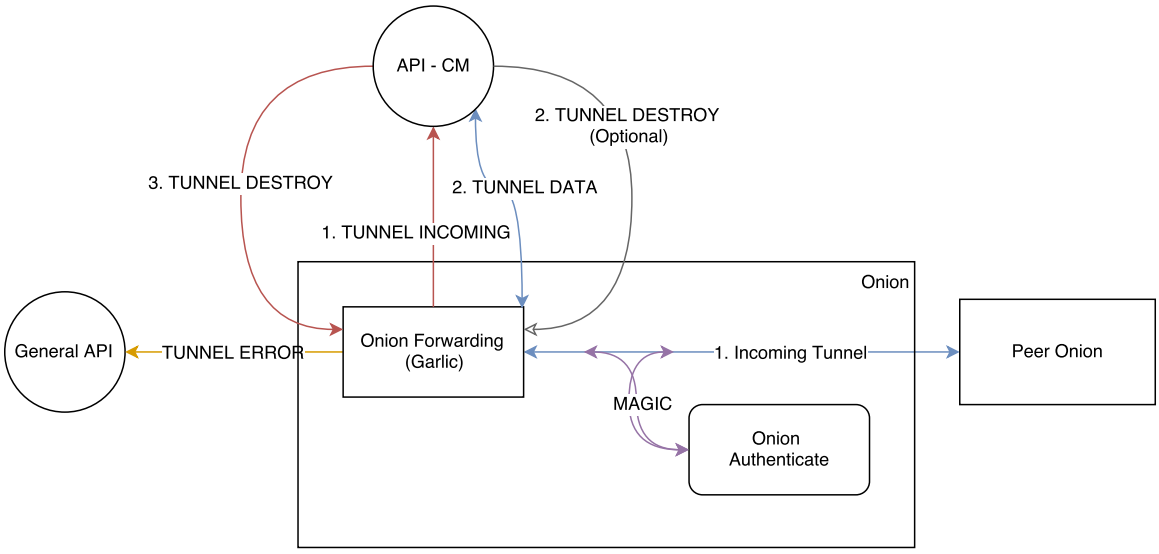


Figure 2: Message flow for receiving incoming tunnel connection

2.2.3 Faking traffic

As our ecosystem inevitably generates traffic while a call is in progress a malicious node can gather information about which nodes are active. To counteract this we introduce cover traffic to our ecosystem. This traffic is to mimic a real call and therefore confuse possible eavesdroppers. How such traffic is generated is shown in figure 3. Random peers - inclusive a random destination peer - are gathered from RPS and with cover payload from CM/UI P2P traffic is sent across them. The intermediate nodes do not know the connection is fake. The destination node forwards the data to CM/UI without knowing either.

2.2.4 Magic behind Onion Authenticate

So far we have looked at the Onion Authenticate module as "MAGIC" happening on the tunnel. Now we look closer at how the message exchange (depicted in figure 4) actually looks like.

We start of with a **Start** message containing the hop's, which we are connecting to, hostkey. Onion Authenticate responds with an **HS1** message. We extract and save its *session ID* - this is local to each peer - for later use. We then forward the payload to the next hop.

This hop sends an **Incoming HS1** to its Auth module with the source hostkey and the received payload. Onion Authenticate responds with an **HS2** message. We again save the local *session ID* and return the payload to the host.

Using its own *session ID* it sends the payload from the response to the Auth module. As long as no errors are reported by Auth the ephemeral key for this connection has been

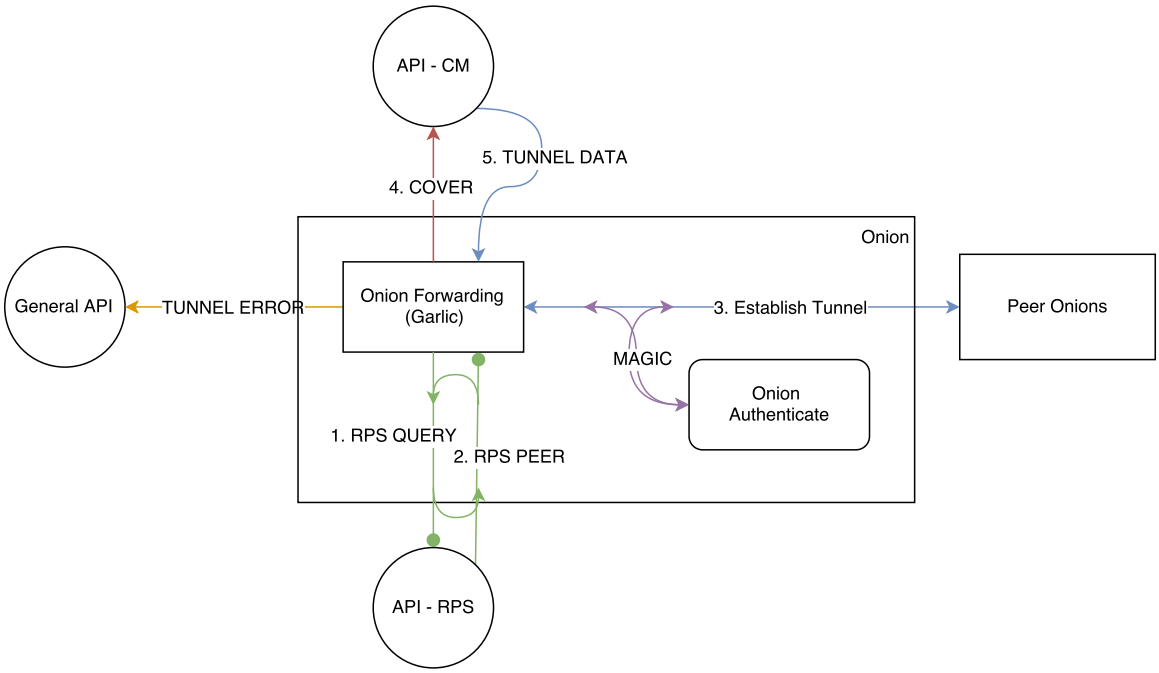


Figure 3: Message flow for generating fake connections

established. This is repeated for each hop as explained in section 2.2.1.

Each successive message needs to be first encrypted with **Encrypt** from Auth passing the number of hops, all *session IDs* associated with this connection and finally the payload. As response it receives the encrypted payload.

The peer receives this payload and decrypts it in a similar way using **Decrypt**.

3 Inter-Process communication (P2P)

3.1 Transport protocols

We will use a combination of TCP & UDP connections for our needs. TCP for the initial key exchange and tunnel initialization and UDP for sending data from CM/UI. The reasoning behind this is, we want to make sure our tunnel is properly setup (using UDP for this would be a pain as dropped packets would require additional effort on our side). On the other hand once a tunnel is setup (since we are programming this module for a VoIP application) a 100% delivery rate is not necessary and we save on bandwidth with UDP.

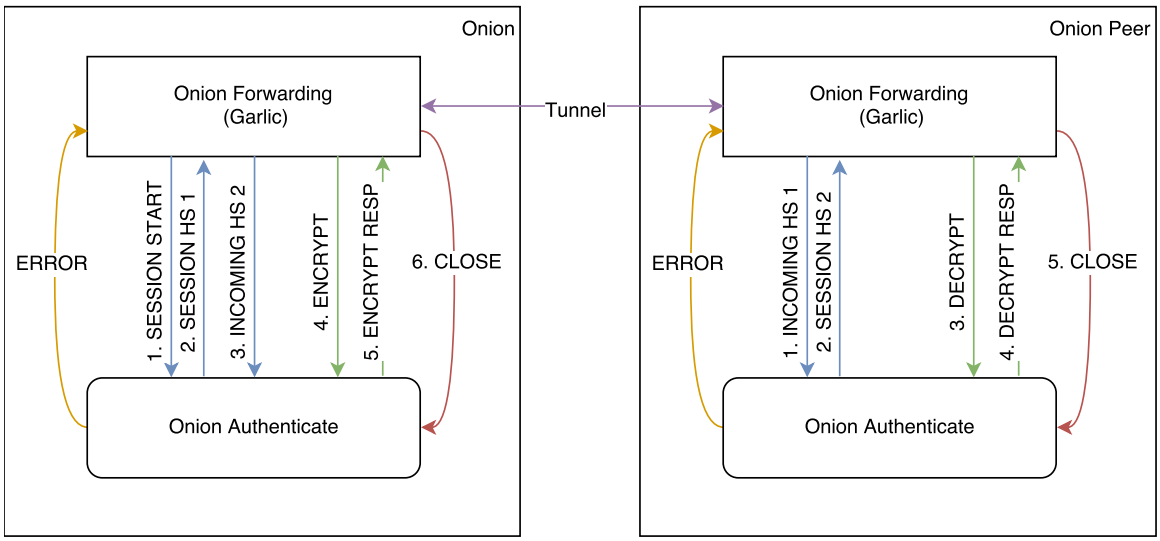


Figure 4: Message flow between Garlic (Onion Forwarding) and Onion Authenticate

3.2 P2P Protocol

The protocol in which messages will be encoded is going to be Cap'n Proto[4] (as I absolutely adore their website & their promise of being insanely fast). The library capnproto-rust[5] is what we will use as implementation.

3.3 P2P Messages

All messages carry an identification number which marks this session's tunnel. Every node can map this ID to a list of hops in the current session's tunnel.

3.3.1 Knock

This message will knock on the destination peer and expect a **Who's There?** message as a response. This protects us from accidentally communicating with nodes that have a different port configuration or nodes which don't support Garlic.

3.3.2 Who's There?

This is the only correct response to the **Knock** message a peer can send to successfully continue the process of establishing a tunnel. The 2 messages **Knock** and **Who's There?** are required whenever a new peer is contacted (as part of one session: non-persistent).

[4] `https://capnproto.org/`
[5] `https://github.com/dwrensha/capnproto-rust`

3.3.3 Handshake

This message facilitates the handshake between the 2 nodes. It is part of the Auth mechanism (Section 2.2.4). It is used for both the request (HS1) aswell as the response (HS2).

3.3.4 Introduce me to...

This message requests the next hop to forward this message to the peer specified in the message. It carries the same information as **Handshake** plus the destination of the peer whose "hand we want to shake". The intermediate node knows it is a handshake between the other 2 nodes but as the payload has been encrypted on both sides it cannot extract the session key of this handshake even though it facilitates its transmission.

3.3.5 Did you know?

This message is simply used to transmit data which was either returned by **Onion Data** or **Onion Cover** (we do not care which one) and encrypted by Auth. It is to be delivered to the destination node at the end of the tunnel.

4 Error handling

Error handling will be facilitated with Rust's Result pattern. Every exception that occurs is logged and handled by the appropriate module. A list of possible errors and their counter measures[6]:

- Peer goes down: The tunnel is immediately torn down and a new one is established
- Connection breaks: Garlic will try to reconnect. If the re-connection fails the tunnel is declared dead and an error is broadcast on the API.
- Corrupted Data: Corrupt data reported by Auth are dropped. If the other module breaks contract on the messages the connection is immediately terminated and the offending node is marked as untrustworthy - Garlic will try to avoid sending traffic through a tunnel which consists of only untrustworthy nodes.

5 Disclaimer

This report will serve as documentation for Garlic and will therefore change accordingly until the project is finished. More changes and, more importantly, additions are to be expected.

[6] Other errors will possibly be added to this document at a later date

Contents

Acronyms

Auth Onion Authenticate. 4, 6, 9–15

CM/UI Client/User Interface. 2–4, 10–13

Garlic Garlic (Onion Forwarding). 3, 5, 11, 14, 15

RPS Receive Packet Steering. 4, 5, 10, 12