Interim Report: Onion Module

Marko Dorfhuber (03658730) ✉marko.dorfhuber@tum.de
Christoph Rudolf (03662116) ✉christoph.rudolf@tum.de

1. Overview over the architecture

The Onion module has a central position in the VoIP application and communicates with three other modules on the same peer, as well as with itself on other peers. For other local modules, depending on the relationship with other modules, the Onion module acts as either a client, requesting functionality (RPS, Onion authentication) or as a server offering its functionality (CM/UI). Figure 1 shows the environment and adjacent modules of the Onion module.

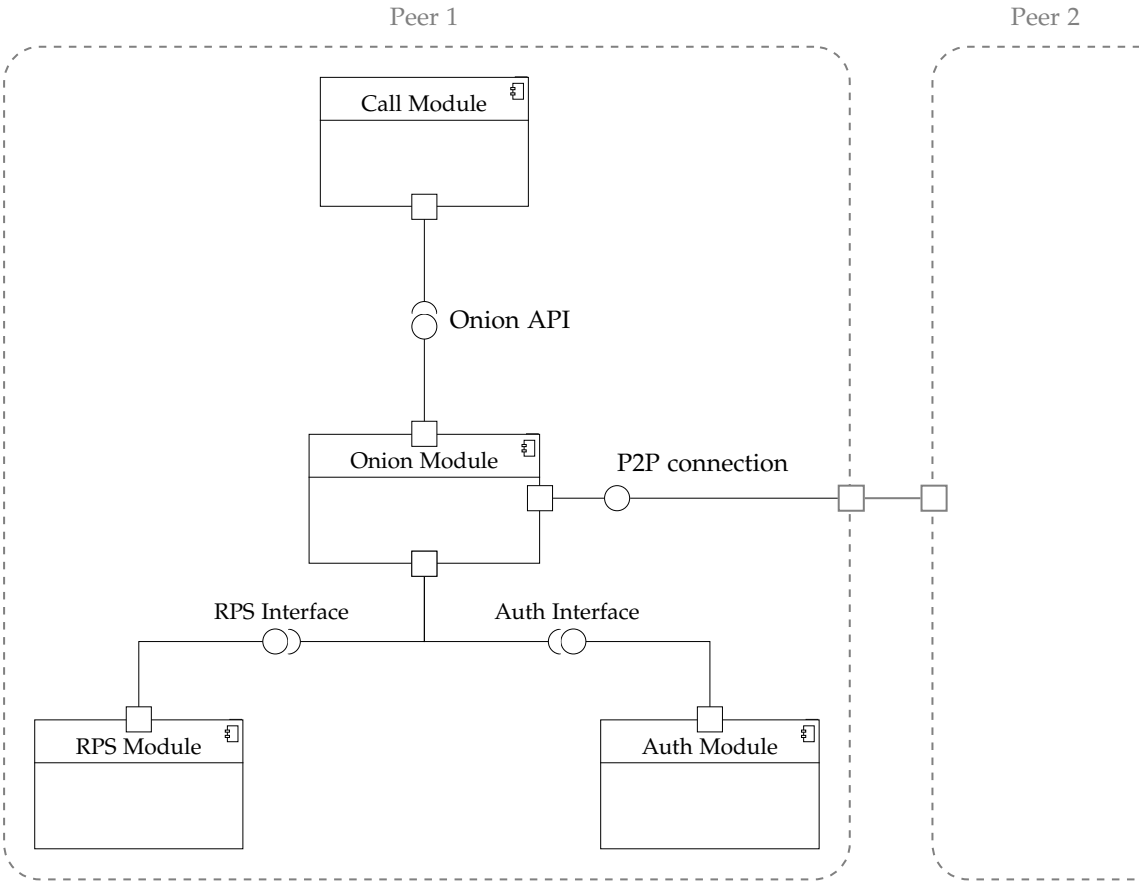


Figure 1: Environment of the Onion module in the Voidphone application.

Each of the Onion module's four interfaces utilizes a parser instance that is injected via *Dependency Injection*. The parser is able to parse and deserialize the packets that are expected to

be received on the respective interface. As stated in the initial report, all network programming is done via **Netty**[1], an asynchronous event-driven network application framework that can use non-blocking I/O (NIO) in Java.

2. Design of the Onion protocol

The following section gives an overview over the design choices made when designing the inter-module protocol for the Onion module. Based on the specification, the Onion module's peer-to-peer protocol does not have to be reliable and is therefore based on UDP.

Even though UDP in itself is connection-less, a minimal state has to be stored by each peer. This state indicates that the peer is part of a tunnel. It holds its predecessor and successor to forward data and the session key the peer agreed on with the tunnel initiator. In order to allow a peer to quickly parse and find the correct tunnel dataset for an incoming packet, the Onion protocol uses a so called *local tunnel identifier (LID)*. The *LID* is a 128 bit identifier (UUID) that acts as a key for the tunnel data on a peer and can be chosen randomly with a sufficiently small probability of conflicts. The local identifier is valid for a segment of the tunnel between exactly two hosts which both know this identifier. Such an identification solves the problem arising if the same predecessor and successor of a hop are part of multiple tunnels, as this introduces an ambiguity regarding the en- and decryption. Figure 2 displays the usage of local tunnel identifiers to tag tunnel fragments. The initiator is always aware about all LIDs, whereas both peers adjacent to a tunnel fragment do also store the LID.

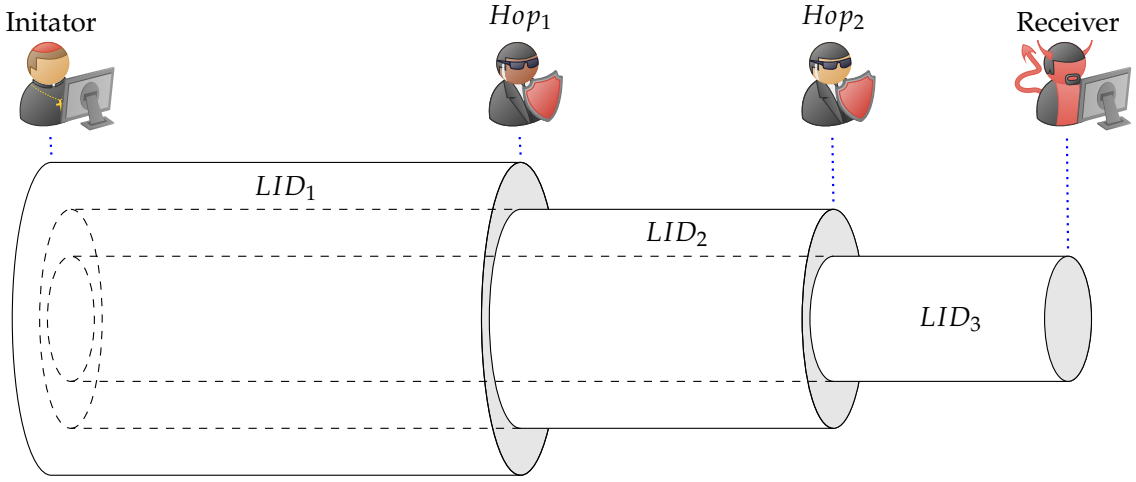


Figure 2: Structure of the tunnels and tunnel identifiers.

A local identification was chosen in favor of a global identifier for a tunnel to avoid leaking private information. As the identifier is sent during the tunnel building process, it reaches a new hop that is appended to the tunnel in plain text. This is necessary because a session key is yet to be established in this phase. An attacker sniffing network traffic could trace a unique global tunnel identifier over multiple hops and derive the complete path of the tunnel.

The Onion module is primarily concerned with building onion tunnels and transferring voice data during phone calls in an anonymous way. Therefore, we need messages to build tunnels in addition to messages for transmitting encrypted voice data. For each message type of the protocol

[1] `http://netty.io/`

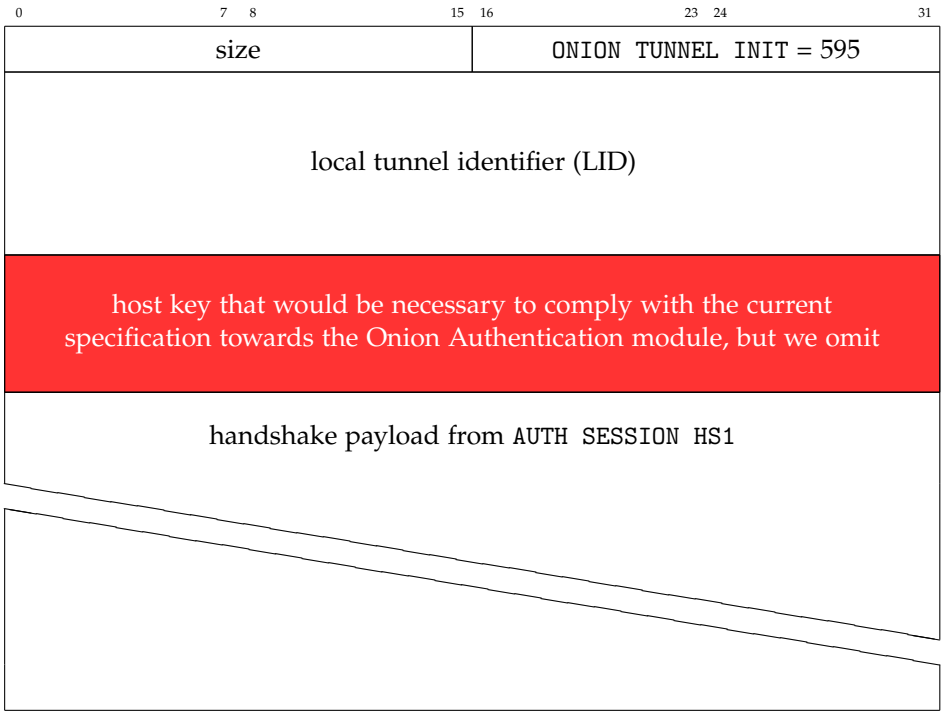
we provide an explanation of all individual fields and their purpose. For this description, peer building a new tunnel will be referred to as *initiator*.

2.1. Onion Tunnel Init

This messages indicates the beginning of a new tunnel segment and is created after an Onion module receives the `ONION TUNNEL BUILD` message from its *Call module* or choses to create a tunnel for cover traffic on its own. Messages of this type are then sent iteratively by the tunnel initiator to the peers he choses to be part of the tunnel. The message deploys a size and type header field to be compliant with all other packets defined in the project's specification. The subsequent 128 bit define the *local tunnel identifier* to use for this tunnel segment.

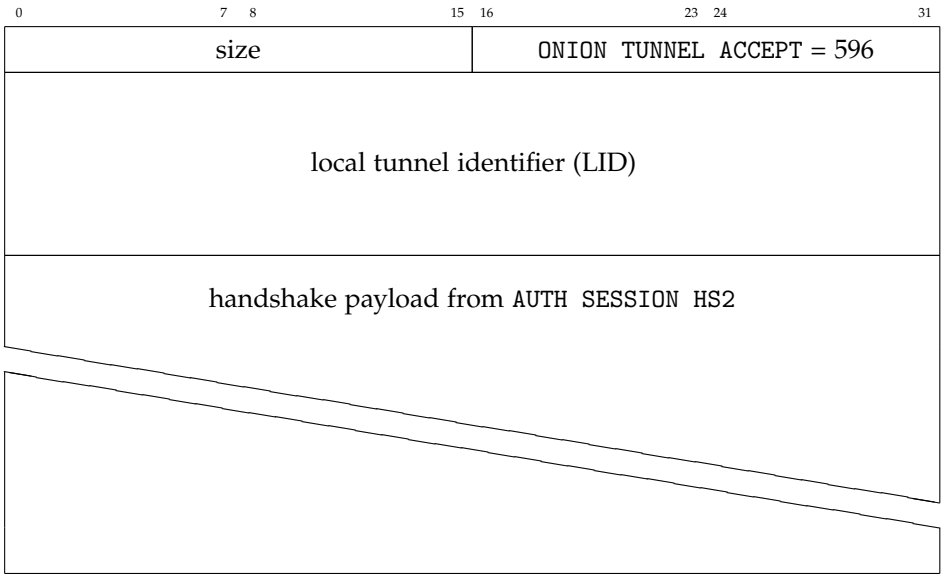
The majority of the message's data is the first part of the authentication handshake. Here we transmit the payload of the `AUTH SESSION HS1` which is given by the tunnel initiator's Onion Authentication module.

Important: If we want to create the `AUTH SESSION INCOMING HS1` message on the receiver's side for its Onion Authentication module, we need to transmit the initiators hostkey as well. However, this leaks information about the initiator to the target for this tunnel, which is a rendezvous point. An alternative discussed with the project's originator Sree Harsha Totakura is to omit the host key and don't verify the initiator on the side of the peer. This does not introduce a security issue because the initiator is still able to verify the authenticity of the session key parameter. In case of Diffie-Hellman, this is sufficient to prevent Man-in-the-Middle attacks. Therefore, our protocol omits the host key of the initiator in this message and therefore also in `AUTH SESSION INCOMING HS1`.



2.2. Onion Tunnel Accept

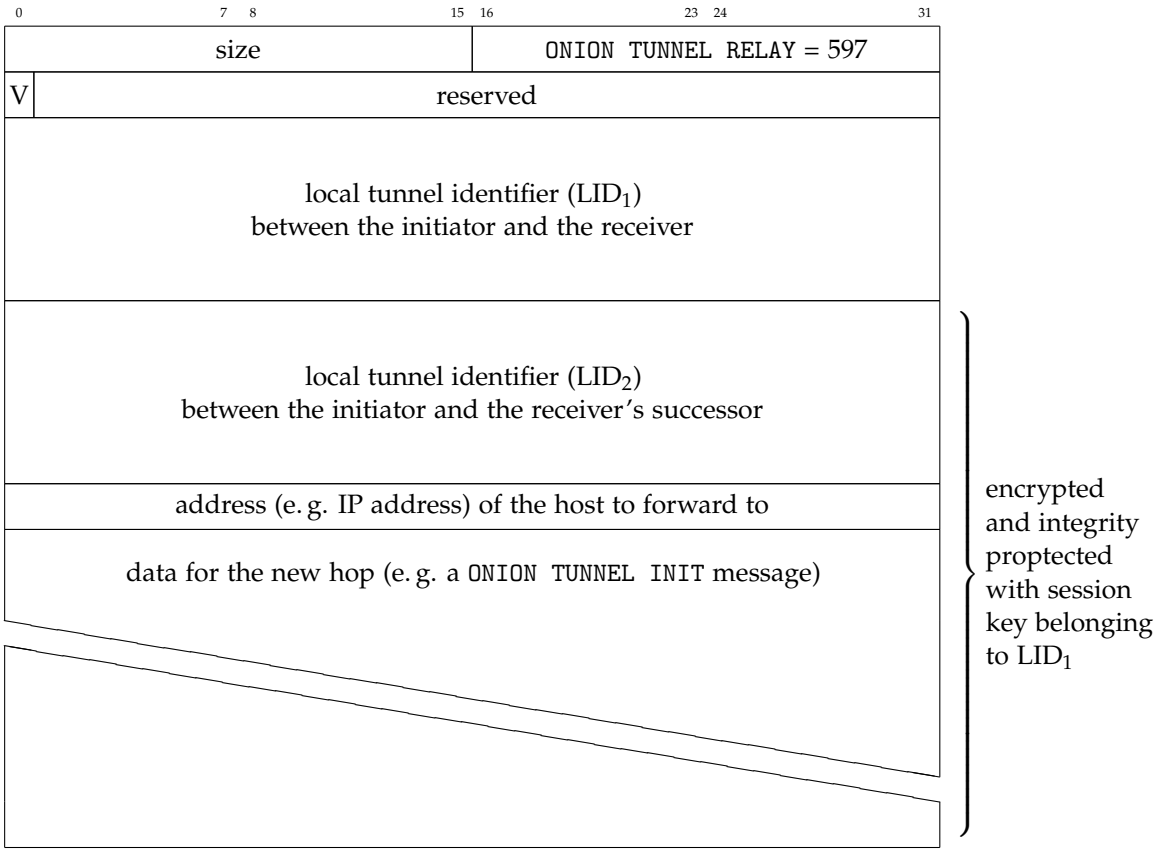
This messages is the expected response to the `ONION TUNNEL INIT` message. It sends the local tunnel identifier specified in the matching `ONION TUNNEL INIT` and forwards the data this peer received from its Onion Authentication module in `AUTH SESSION HS2`. `AUTH SESSION HS2` contains the response to the first authentication handshake and finishes the session establishment. After this message is received, data between the tunnel initiator and the hop sending this, can be encrypted.



2.3. Onion Tunnel Relay

The relay message is used to instruct a host that is already part of the tunnel, to send data to a new host currently outside of the tunnel. This instruction can already be encrypted with the session keys established so far. With this message, the forwarding host is also instructed to assign the new host as its successor for this tunnel segment. This message is necessary for tunnel building when the currently last hop expands the tunnel with a new hop.

The message includes the typical header with size and type. An additional bit `V` defines the usage of IPv4 (= 0) or IPv6 (= 1). Subsequently, the message includes the local identifier matching the connection from the initiator to the receiver. This LID allows the receiver of this message to chose the correct session key for decrypting the subsequent data. The encrypted data includes the destination for this message and an arbitrary nested onion tunnel message type. With this, the initiator picks all local tunnel identifiers.



2.4. Onion Tunnel Transport

The `ONION TUNNEL TRANSPORT` message is used to securely transmit data over an already established tunnel. As specified by the project specification [4], a fixed length is the best choice regarding anonymity. Otherwise, an attacker can use traffic analysis to gain intel on which data is real voice data and which is merely cover traffic.

The best packet size for the Onion to Onion data packets depends on various parameters: quality, time frame per packet, payload size and bandwidth. A packet should transmit a relatively small time frame of voice data to be negligible when being lost. The packet, however, should still be big enough to avoid unnecessarily large overheads.

For padding the payload, we aim to use PKCS#7, which is a well known and easy to parse padding-scheme. With PKCS#7, to pad a total number of n bytes, one simply adds n bytes with the value n. This limits the amount of data that can be padded to 256 bytes. However, a payload of exactly 256 bytes would lead PKCS#7 to add another 256 byte long block, which would result in two packets for our scenario. Therefore, the maximum payload length we aim to use is 255 bytes.

The size of 255 bytes for voice data per packet has to be reasonable regarding the quality and length of the voice data transmitted. In order to find suitable values, we have to start with the question of how much time a single packet should contain. The average rate of speech for an English speaker has a magnitude of around **150 words per minute** [3]. This results in:

$$150 \frac{\text{words}}{\text{min}} \cdot \frac{1}{60} \frac{\text{min}}{\text{s}} = 2.5 \frac{\text{words}}{\text{s}}$$

Considering the roughly 2.5 $\frac{\text{words}}{\text{s}}$, the timespan covered in one packet has to be short enough to avoid losing meaningful fractions of words in case of a packet loss. Vendors of VoIP applications and telephones implement packets to contain between 20 ms and 40 ms [1] of audio. With our 255 bytes of payload data per packet we are able to transmit

$$p_{size} = 255\,\text{byte} = 2040\,\text{bit} \qquad \textit{(per packet)}$$

Assuming we receive a packetization rate of $\Delta t = 30\,\text{ms}$ from the Call module, we can reach a bitrate r of

$$r = \frac{p_{size}}{\Delta t} = \frac{2040\,\text{bit}}{30 \cdot 10^{-3}\,\text{s}} = 68 \cdot 10^3\,\frac{\text{bit}}{\text{s}} = 68\,\text{kbps}$$

This bitrate is sufficient to support all common VoIP codecs which use bitrates up to a maximum of 64 kbps [5].

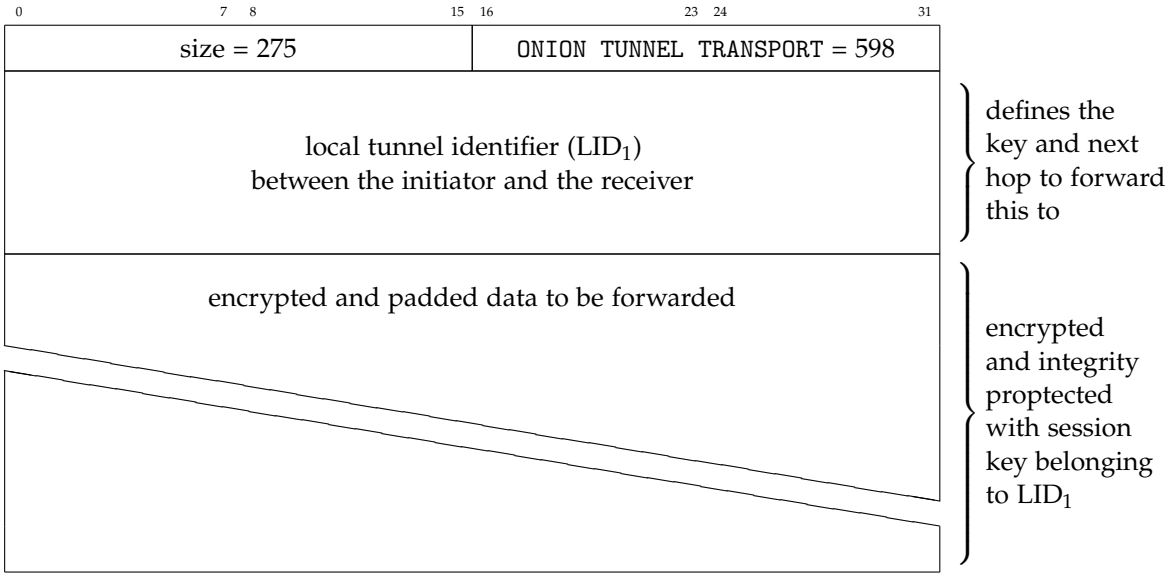
The last parameter to consider is the necessary bandwith. We have 255 bytes of voice payload and an additional header size of 20 bytes for our application protocol. This protocol is based on UDP which adds an additional 8 bytes for its header. The network layer protocol header can be either 20 bytes (IPv4) or 40 bytes (IPv6). Assuming IPv6 to be used we get a total packet size s per 30 ms of speech:

$$s = 255\,\text{byte} + 20\,\text{byte} + 8\,\text{byte} + 40\,\text{byte} = 323\,\text{byte}$$

To ensure this data can be delivered in the necessary amount of time Δt, we need a network bandwidth B of:

$$R = \frac{s}{\Delta t} = 323\,\text{byte} \cdot \frac{1}{30 \cdot 10^{-3}}\,\frac{1}{\text{s}} = 10\,767\,\frac{\text{byte}}{\text{s}} \approx 0.09\,\text{MBit/s}$$

A necesssary data rate of 0.09 MBit/s is easily within the limitations of current Internet connection speeds. This makes a fixed packet size of 255 byte + 20 byte = **275 byte** for the Onion to Onion protocol suitable.

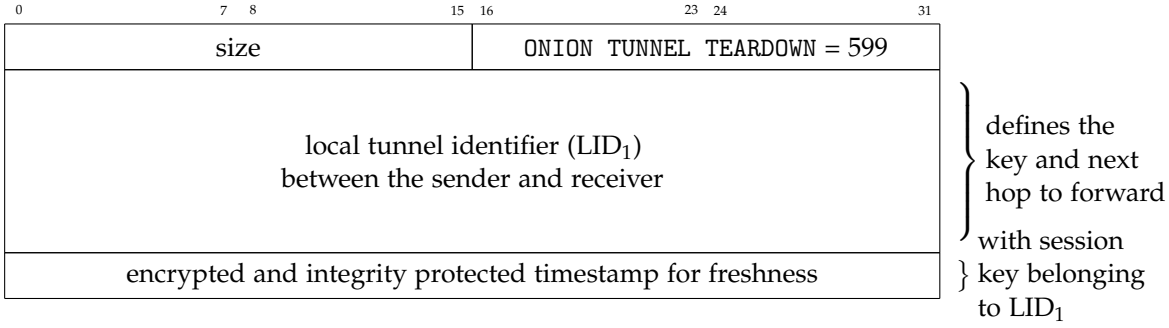


2.5. Onion Tunnel Teardown

The teardown process of a tunnel can only be issued by its originator. This is necessary because a tunnel has to be teared down beginning from very the last hop and only the originator can send an encrypted teardown message through the whole tunnel. Unauthenticated tear downs would introduce a huge weakness to denial of service attacks on existing tunnels.

As peers might be malicious or randomly disconnect before terminating their tunnels, peers used as intermediate hops need to be able to remove old tunnel states on their own. To avoid running out of memory, a peer can remove its tunnel state safely after two rounds. The timespan of two rounds makes sure that even with the maximum possible offset between the round starting points on two peers, no tunnel is prematurely eliminated.

To ensure freshness and avoid replay attacks against open tunnels, the payload of this message is an integrity protected timestamp. Due to the fact that we can only use the interface to the Onion Authentication module for encryption and authenticity, both is applied to the timestamp even though authenticity (e. g. via an HMAC) would be enough.



3. Protocol sequence

The final protocol sequence composed of all aforementioned message types is given below in Figure 3. As Onion routing hasn't been covered in the lecture so far, the basic idea of repeated encapsulation of encrypted data has been derived from the TOR paper by Dingledine et. al. [2].

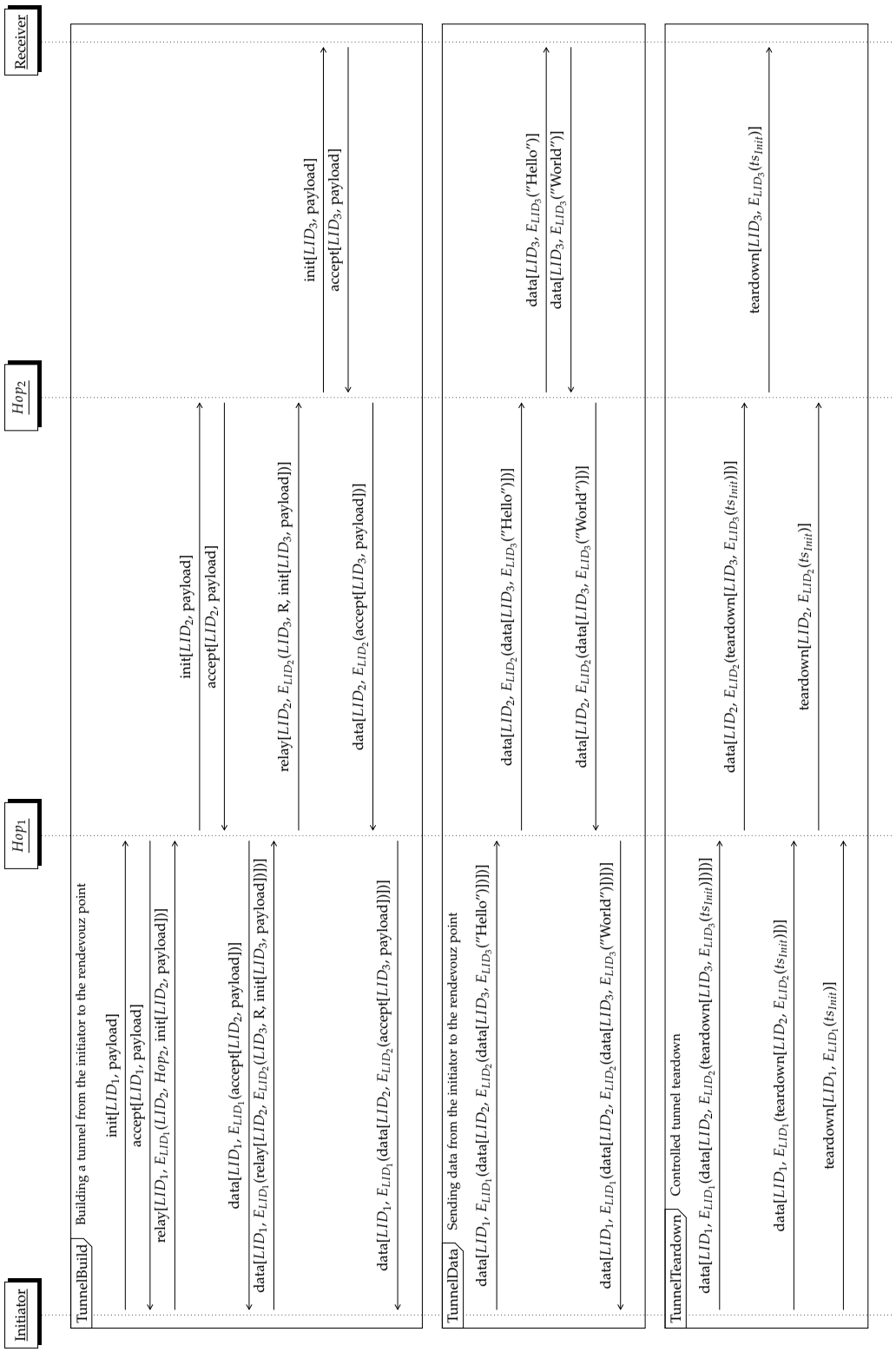


Figure 3: Protocol sequence of the complete Onion peer-to-peer protocol.

References

[1] G. Audin. An introduction to bandwidth for VoIP. http://searchunifiedcommunications.techtarget.com/tip/Whats-in-a-VoIP-packet-An-introduction-to-bandwidth-for-VoIP, Jan. 2007. Accessed: 2017-06-17.

[2] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The Second-generation Onion Router. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, SSYM'04, pages 21–21, Berkeley, CA, USA, 2004. USENIX Association.

[3] National Center for Voice and Speech. Voice Qualities. http://www.ncvs.org/ncvs/tutorials/voiceprod/tutorial/quality.html. Accessed: 2017-06-17.

[4] S. H. Totakura, L. Schwaighofer, and G. Carle. Project Specification (P2PSEC SoSe 2017) – Version: 2017-2.0, June 2017. Accessed: 2017-06-18.

[5] N. Unuth. Most Common VoIP Codecs. https://www.lifewire.com/voip-codecs-3426728, Apr. 2017. Accessed: 2017-06-17.