Final Report

16 Onions

Charlie Groh Josef Stark

August 21, 2017

1 General

We are the team "16 Onions" consisting of Charlie Groh and Josef Stark, and our goal is to develop a prototype implementation of the Onion module.

2 Application Internals

In this section we will describe the general structure of the application, the network protocol we implemented, as well as an overview over how the source code is organized.

2.1 Network Protocols

For the communication between distinct onion instances we decided to use both TCP and UDP as underlying protocols in order to avoid reinventing the wheel, since they both fulfill the respective requirements perfectly.

Control messages, i.e. messages for authentication, tunnel construction, tunnel destruction and heartbeat are transferred and forwarded over TCP, because for those messages it is very important that they actually arrive or that we get feedback if one of those messages could not be delivered to the target, so we can react in an appropriate manner—e.g. assume that the corresponding node went down and construct an alternative route. TCP satisfies these requirements as it acknowledges the reception of messages, resends messages if necessary and reports a failure if a message still didn't provoke an acknowledgement after a few retries. If we used UDP instead, we would have had to re-implement all of these features by hand.

User data messages, i.e. messages containing VoIP and cover data, are transferred and forwarded over UDP, since for those a short delay is a requirement which UDP can satisfy. UDP is packet based and does not acknowledge the arrival of packets at all, so the lower delay that this causes comes at the price of possibly losing some packets which are not resent and therefore never reach their target, without the sender being informed about the loss. This is acceptable for VoIP and cover data.

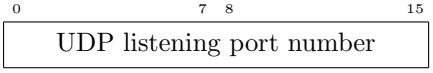
Other modules can use the onion module only to send user data messages (UDP), while control messages are exclusively used internally for tunnel management tasks.

2.1.1 Layer 0: Network Abstraction Layer

At the lowest level of our application lies the so-called multiplexer, which abstracts away low-level protocol details of the underlay connections and offers a clean interface for message interchange between nodes via logical overlay connections. The core tunnel mechanisms exclusively use this interface. The main tasks of the multiplexer are:

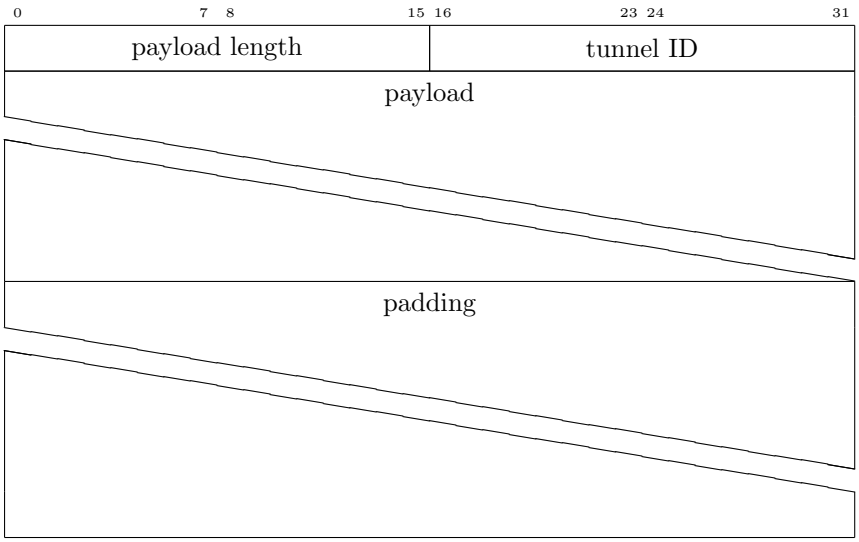
- Establishing and terminating control data channels to other peers using TCP
- Establishing and terminating user data channels to other peers using UDP
- Padding/Unpadding messages to/from a fixed, configurable size while offering transport functionality of variable-sized payload
- Correctly assigning and relating incoming and outgoing messages to the tunnel they belong to

Upon receiving the request to connect to a certain peer, the multiplexer first establishes a TCP connection to that peer for transporting control data and then immediately sends a packet to that peer containing the UDP port on which it is listening for user data:



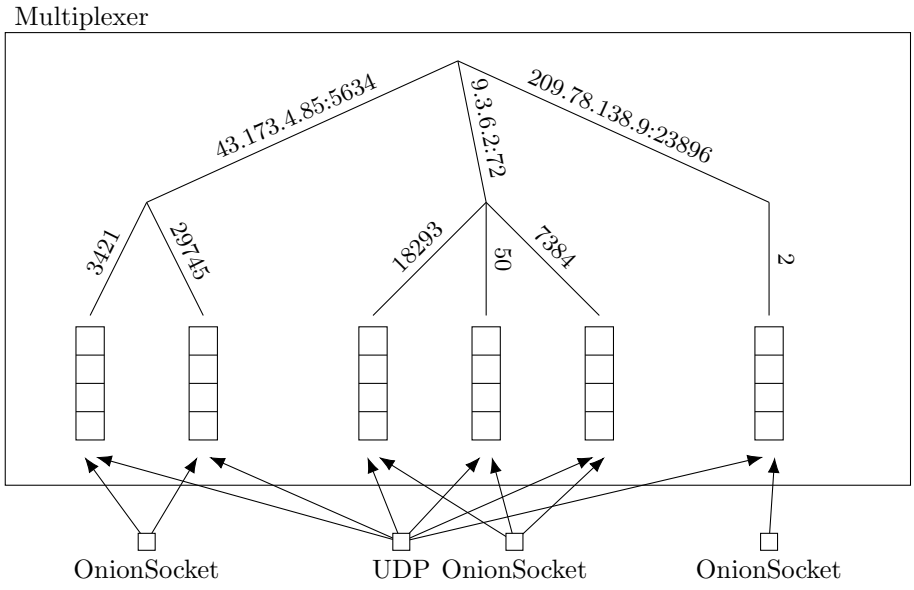
(TCP message)

This is the only message type that is exchanged on its own. After it has been sent, messages are only sent upon request of the core tunnel mechanism. They can be transmitted over TCP or UDP, which is controlled by a flag of the corresponding function. They have the following structure:



(TCP or UDP message)

Whenever a previously unknown peer—that is, a host with a source address-port combination, which did not communicate with the peer previously—connects (via TCP) to the peer, the multiplexer assigns an ID to that connection and creates a message queue for it. All incoming messages from the underlying connections are assigned and put into the respective queue belonging to the connection: Messages from `OnionSocket`, which are basically TCP messages, as well as UDP messages, whose filtering would otherwise need to be done by hand, since UDP is by nature connectionless. The queues, which are essentially the logical overlay connections exported by the multiplexer, can be accessed by their previously assigned ID to read and write TCP and UDP packets—or rather control and data messages, represented by the `OnionMessage` class. Even though there are no separate queues for TCP and UDP packets of a connection, their type can be determined to be control or data message via `type` field.



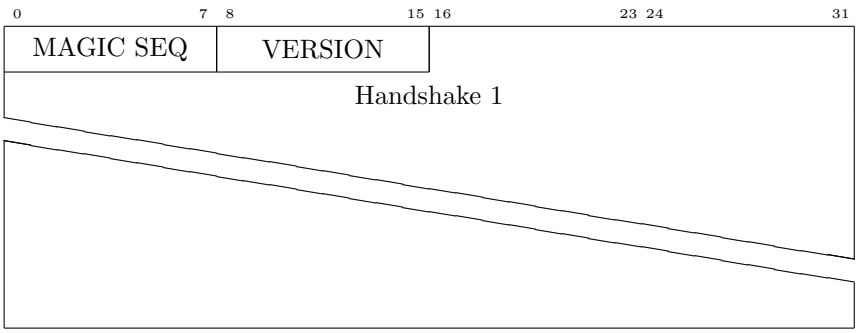
New connections are signaled by calling the `newConnection()` method, where almost all higher-level (Layer 1) messaging is done.

2.1.2 Layer 1: Tunnel Management Layer

The core tunnel mechanisms build upon the network abstraction layer provided by the multiplexer and use its interface to access the logical overlay connections and exchange messages. Those messages, which will be described in this section are thus free of metadata and padding from Layer 0. This information is added/removed and generally handled transparently by the multiplexer.

This section lists each message type and their exact structure used during all the phases of a tunnel lifespan. For the sake of readability, we will refer to the initiator of a connection as "peer A" and to the other node as "peer B".

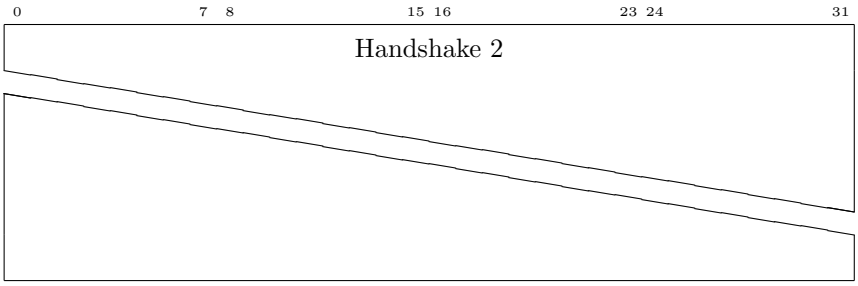
- Handshake 1



(TCP message)

This message is sent from peer A to initiate the authentication process. MAGIC SEQ is a bit sequence identifying the connection initiator as onion peer; VERSION contains the P2P protocol version. Handshake 1 is obtained from the ONION AUTH module of peer A and forwarded to peer B via this message, where it is passed to the local ONION AUTH module—given that it is actually a onion node, MAGIC SEQ matches and VERSION is compatible.

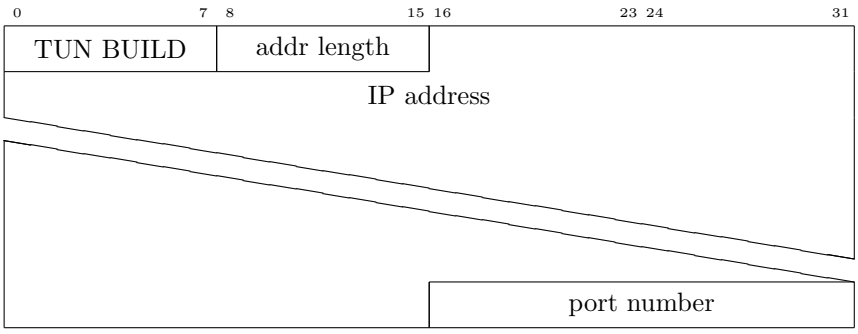
- Handshake 2



(TCP message)

After having processed Handshake 1, peer B obtains Handshake 2 from its ONION AUTH module—given that the Handshake was valid. It then sends this Handshake 2 back to peer A. ONION AUTH of peer A now checks the received Handshake for validity. If positive, the handshake is further processed and an encrypted connection between peer A and peer B has been established. In any other case—peer A received an invalid handshake or no answer at all and timed out—the authentication is aborted and adequate measures are to be taken, e.g. selecting a different node and starting over the procedure.

- Tunnel building/expansion request



(TCP message)

Peer A can send this message to an authenticated peer B to instruct it to connect to another peer C, whose address is specified in this message. Upon receiving it, peer B will now forward all incoming messages between

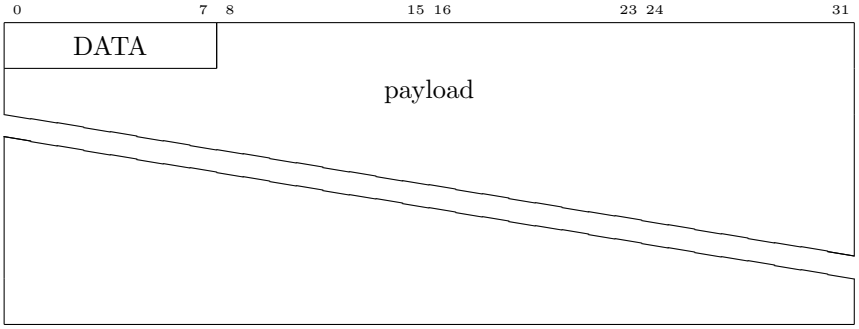
peer A and peer C, so peer A can begin authenticating with peer C over peer B.

- Incoming tunnel notification

(TCP message)

This message is sent from peer A to the peer D at the end of the tunnel, so that peer D knows that the established tunnel ends at peer D and that the further incoming data is destined to him and must not be forwarded. After receiving this message, no more tunnel building/expansion requests can be sent to peer D.

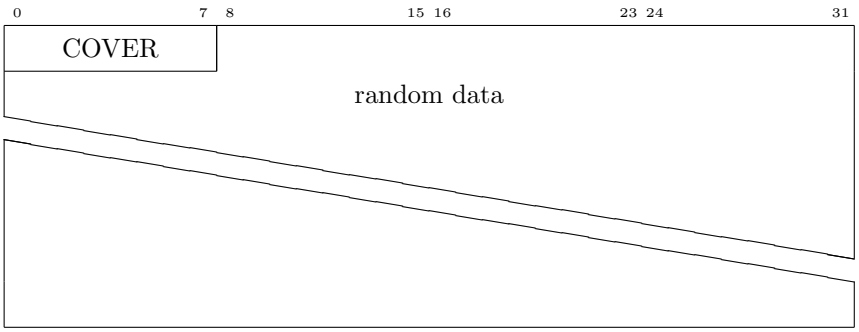
- User data



(UDP message)

Contains real user (i.e. VoIP) data, which was requested to be sent by another local module. Requires the previous reception of an incoming tunnel notification in order to be interpreted.

- User data



(UDP message)

Contains cover traffic of a specified size, which was requested to be sent by the local CM/UI module. Since it contains no meaningful data, the message is discarded upon reception.

- Heartbeat

(UDP message)

If a time-out was triggered, meaning that a peer has not received data of any kind from a certain tunnel in a predefined amount of time, it sends this message to request a sign of live from the node at the other end of the tunnel. This sign of live can be any data, but if there is currently no real user data available for being sent, it is simply cover traffic. Each time-out triggers the sending of such a heartbeat message and the incrementation of a retries-counter. Once a response has been received, that counter is reset to zero; if the counter exceeds five, the tunnel is presumed dead. This message (as well as cover traffic and real data) can only be sent by the two end nodes of a tunnel. An intermediate hop also counts the time-outs and disconnects from its predecessor and successor if necessary, but does not send messages on its own, it only forwards them.

2.2 Mode of Operation

After the application has been initialized, it accepts requests from the CM/UI to:

- Build tunnels
- Destroy tunnels
- Send data through tunnels
- Send cover traffic through tunnels

Apart from that, it listens for new incoming tunnels and data arriving through tunnels and notifies the CM/UI for any of those events.
Time in the system is divided into equal intervals called rounds. New tunnels are established at the beginning of the next round and destroyed upon request. If at the beginning of a round there is not any tunnel in existence or pending to be built, the module builds a tunnel to a random destination, which is automatically destroyed at the end of the round. This kind of tunnel can only be used to send cover traffic.

2.2.1 Tunnel Construction Process

The class used for constructing outgoing tunnels is called `OnionConnectingSocket`. For incoming as well as intermediate tunnels (when a peer is only forwarding

traffic), `OnionListenerSocket` is used. The process of constructing it is as follows:

1. An array of intermediate hops is constructed by repeatedly asking the RPS module for random peers. The amount of hops is determined in the config file. The last node is the specified target.
2. The first unconnected node in the hop array is selected.
3. Handshake 1 is generated and sent to the selected hop.
4. Handshake 2 is received.
5. Test, if there are any unconnected hops remaining in the array:
   (a) If yes: A tunnel building/expansion request, containing the next hop as target, is sent to the current hop. That hop then connects to the specified target and forwards all traffic coming through that tunnel to it and vice-versa. The next unconnected node in the array is connected and the next loop iteration executed, starting at point 3.
   (b) If not: The tunnel endpoint is reached, so the incoming tunnel notification is sent.

If during this construction process an unexpected packet is received or an answer takes too long, a different intermediate peer is selected or, if the node in question is the tunnel endpoint, the construction is aborted and the failure, if required, reported to the CM/UI module.

Once established, a tunnel can and will be used to transfer data-messages, heartbeats and cover-traffic.

While the CM/UI cannot explicitly send cover data over tunnels whose construction it requested, cover messages are used internally if a node needs to respond to a Heartbeat and has no real user data available to be sent.

2.2.2 Tunnel Destruction Process

Tearing down a tunnel happens in the reverse order of construction: The respective message is sent to all hops, starting with the tunnel node to the node which is directly connected to the initiator. Once a hop receives the message, it closes the connection to its predecessor in that tunnel. Should an error happen while sending the message

2.3 Testsuite

The module contains a testsuite that tests artificial and real world situations. The testsuite is located in the package `com.voidphone.testing`. The class `Testing` is a unit test that tests the module for API conformance and some basic functionality like building a tunnel between two hops. The class `Test` launches multiple hops and tries to build multiple tunnels between them. When the test terminates, it will print the number of successful calls.

In order to avoid stale files, the testsuite will delete generated config files in `tmp/` after executing the test. If you want to keep them, you must set `deleteConfigAfterTest` to `false` in the `Helper` class. To prevent stale open ports, every started process is killed after usage. In addition, every test is terminated after at most two minutes. Unfortuanly, we were not able to add satisfying support for the `SecurityHelper` class in the `testing`-module. So it may be necessary to hardcode the keystore path in the `propertiesPath` variable.

2.4 Bugs and Limitations

3 Project Work

Although there are still some bugs in our module, we consider the project as successful.

3.1 Work Distribution

3.2 Spent Effort

Since we mostly worked on the project at the same time, the effort spent by each individual is almost identical. On average, every team member spent around three hours per week for the project.