Final Report: Onion Module

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

1. Overview over the architecture

This section aims to give an overview over the basic architecture and components of the final implementation of the Onion module. The interim report gave an abstract overview over the environment of the Onion Module, making clear that its central role in the VoidPhone application requires the module to have many interfaces. Figure 1 shows a more detailed representation of the final design. The figure can be seen as the inner workings of the *Onion Module* component depicted in the component diagram of the interim report (see Figure 1 of the interim report).

Each interface the Onion module has towards another module is managed by a respective interface class. Common base classes like for example an abstract `TcpClientInterface` for Authentication and RPS are used to encapsulate shared functionality among these interface classes. The common basis for all elements with networking code is the **Netty**[1] framework, an asynchronous event-driven network application framework based on Java's NIO.

Each of the interfaces references a parser specific for the data that is expected on this interface. The parser is given the raw bytes which is received by the interface and returns a serialized message instance or a parsing error. The parsers are also used by the refercing modules to build the message types they are sending. All references between components are setup via *Depedency Injection* using Google's **Guice** framework.

The Orchestrator acts as a starting point for the module and manages a total of three interfaces (`RPS Interface`, `Onion API Interface`, `Onion P2P Interface`). It's job is to issue the start of rounds and to manage data flow between interfaces. For this, the `Orchestrator` has references and callbacks towards interfaces. In case of an event on one interface that affects another one, an approriate transfer is done. An example would be that the UI/CM module wants to build a new tunnel which requires the sampling of hops via the `RPS Interface` as well as the tunnel build itself by the `Onion P2P interface`. The `Orchestrator` also references the `Configuration Provider` which reads the configuration file and parses it to provide the given data to all modules.

The Onion API Interface acts as a TCP server for a connecting UI/CM module. It is able to handle all messages specified as Onion API messages in the specification [1]. A callback connects the interface to the `Orchestrator` and notifies it of incoming requests.

The RPS Interface is the simplest interface, acting as a TCP client towards the RPS module. It only provides one method to query a random peer (using the `RPS QUERY` message type) which encapsulates the call to the other module.

The Onion P2P Interface is in charge of managing the peer-to-peer connections (tunnels) to other modules. It acts as a combined UDP server and client. Its parser handles our custom

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

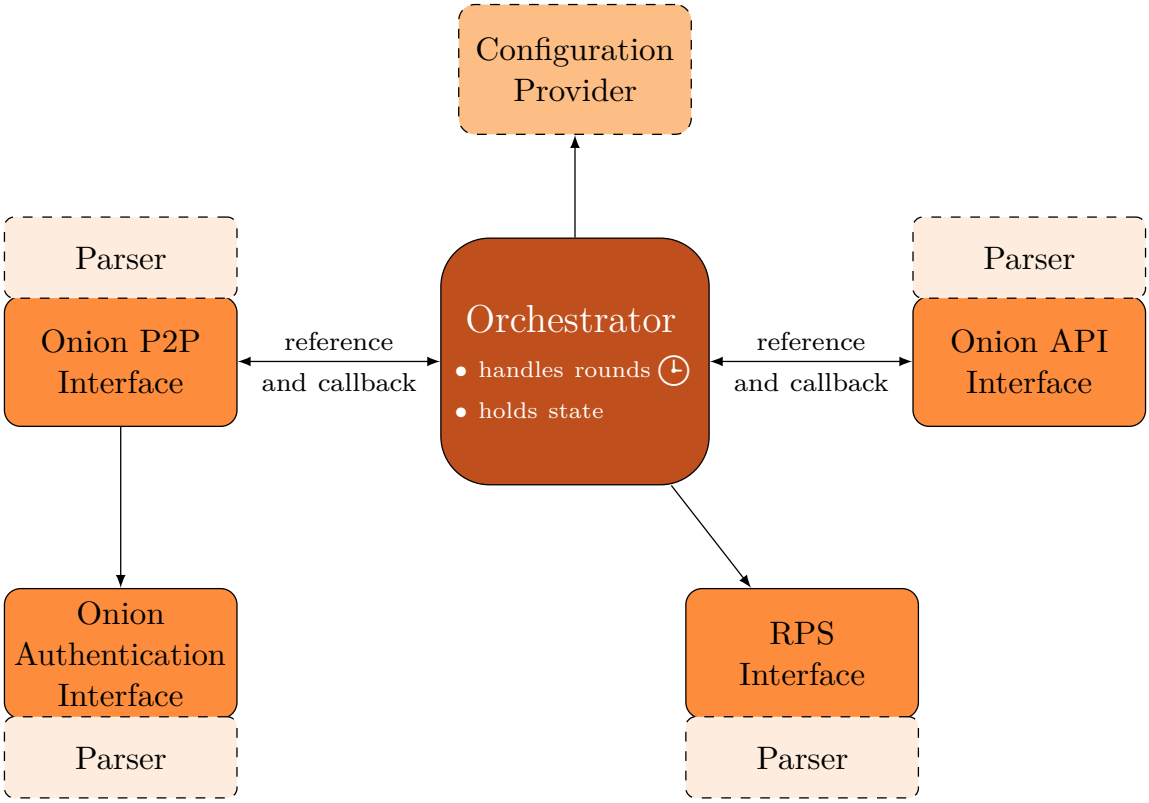


Figure 1: Architecture overview of the final implementation.

P2P protocol (see Section 2). The `Onion Authentication Interface`, a client to the Authentication module, is referenced directly. This accounts for the fact that both modules are strongly connected, even being combined into one in the past. The P2P interfaces is the only module that uses the `Onion Authentication Interface` and needs to call its functionality frequently during tunnel operations which do not trigger any callback to the `Orchestrator`. This way, the `Orchestrator` can simply issue a tunnel to be build, select the random peers and is not required to handle roundtrips to the authentication module for every encryption happening as a result.

Data structures and identifiers: The application has a set of central data structures used to store its state. The major ones are collections of tunnels started by the running peer as initiator (`startedTunnels`), tunnels that this peer acts as an endpoint to (`incomingTunnels`) and states for this peer's role as an intermediate hop (`intermediateSegments`). The collections are created by the `Orchestrator` and also accessible by the `Onion P2P Interface`. From the perspective of the application, a tunnel is an object with a tunnel identifier and a list of tunnel segments. The segments have precise information about adjecent hops (address and port). A segment created for each hop of a tunnel, describing the state between the tunnel initiator and this hop. Therefore, a segment does also know about the session it belongs to and is identified by a **local identifiers (LIDs)**. In a peers role as an intermediate hop, each tunnel segment gets a matching counterpart segment assigned. With this, a peer is able to match forwarding of data with a certain LID in both ways through the tunnel. Each time, he

has to write the corresponding LID expected by the next peer inside the message header. It is noteworthy that for a peer's role as intermediate hop, we only store tunnel segments (not whole tunnels) as an additional grouping by tunnel ID, has no relevance for this peer. Tunnel IDs, and therefore whole tunnel instances, are only stored for tunnels a peer started and the ones he received as an endpoint, as these are scenarios in which the peer has to report to its respective UI/CM module stating a tunnel ID.

The LIDs have been introduced in the interim report and, as we noticed during the lectures on TOR, correspond to the circuit IDs in TOR. They are valid between the initiator and a specific hop of a tunnel. The LID allows an intermediate peer to match an incoming packet to a session and a tunnel-specific next hop for forwarding. Figure 2 taken from our interim report visualizes the concept and the validity of the identifiers in an Onion tunnel with two intermediate hops.

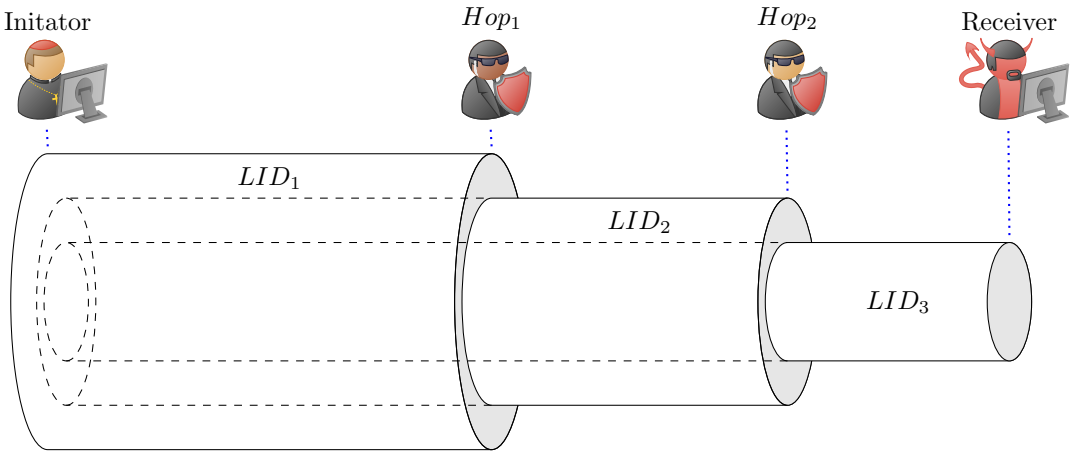


Figure 2: Structure of a tunnel and validity of the local tunnel identifiers (LIDs).

2. Final Onion P2P protocol

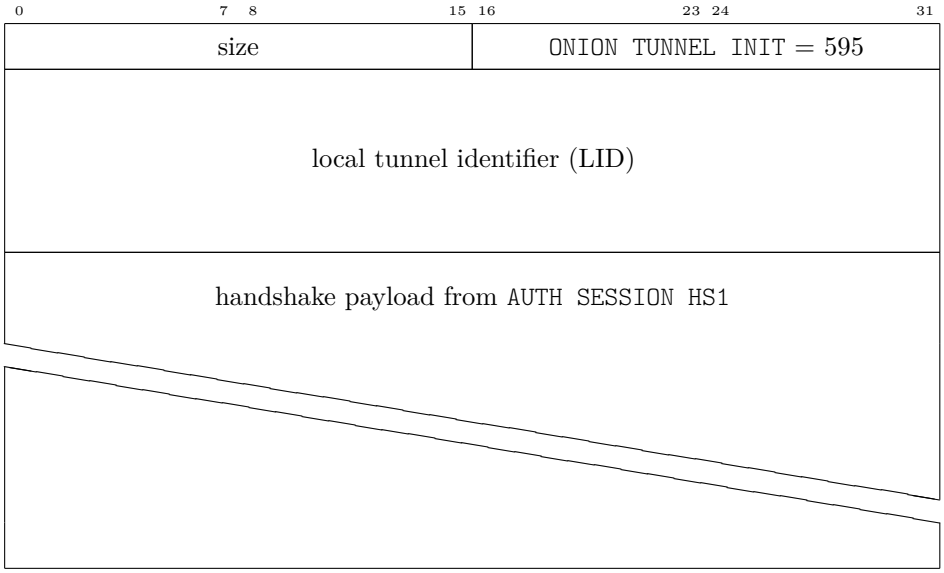
The protocol proposed in the interim report underwent minor improvements due to specification changes regarding the fixed size of packets. With new message types towards the authentication module, we are now able to maintain a fixed message size during onion layer encryption at each hop. With the lectures on TOR's onion protocol that happened shortly after the interim report's submission and the changes to the specification, we were able to refine our protocol.

This section describes the final message types for the Onion P2P protocol and the purpose of each. Not all message types underwent changes, however, this section still includes them to provide a complete replacement to the interim report. In order to maintain consistency with the inter-module protocols defined by the specification, each message type starts with a fixed four byte header that includes the message size and its type. We additionally specified a 128 bit local identifier (LID) to follow for every type.

2.1. Onion Tunnel Init

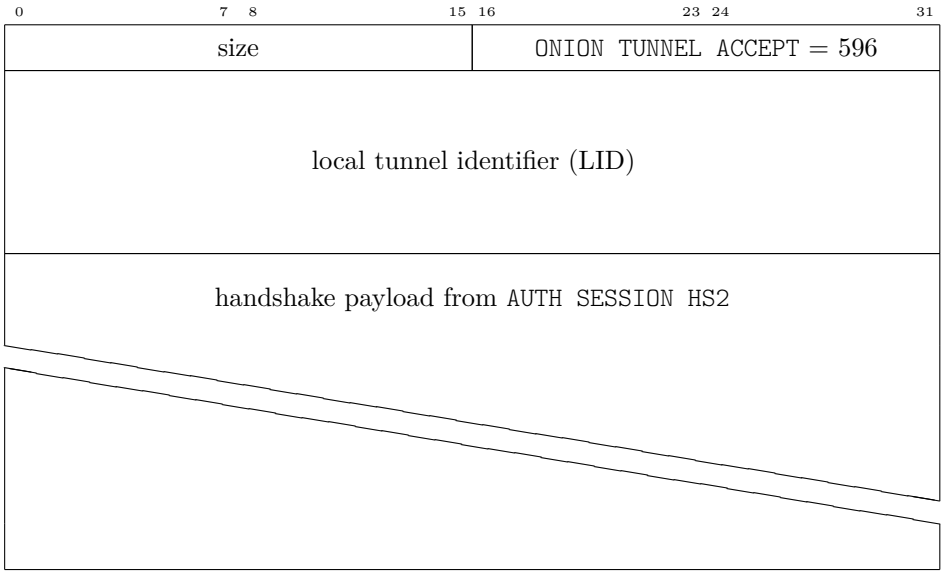
This messages indicates the beginning of a new tunnel segment and is created after the `Onion API Interfaces` receives the ONION TUNNEL BUILD message from its client UI/CM module or choses

to create a tunnel for cover traffic on its own. Messages of this type are sent iteratively by the tunnel initiator to the peers he choses to be part of the tunnel. Each encrypted depending on how far the tunnel is already setup. It is used to transmit the first handshake data given by the Onion Authentication module.



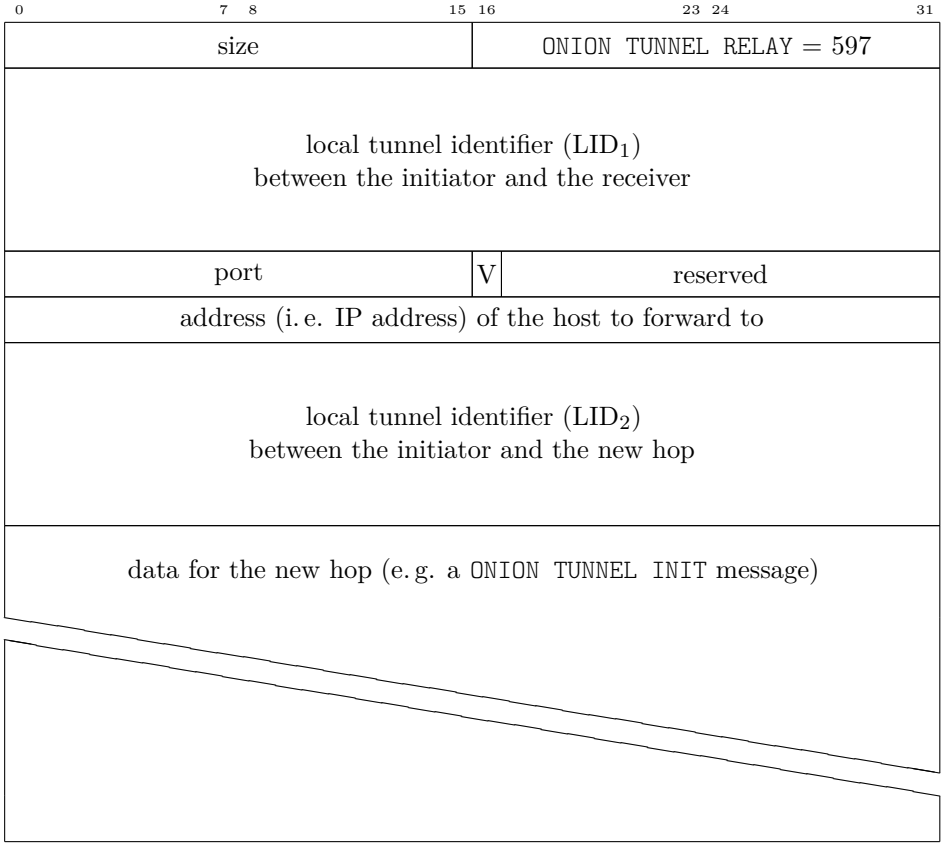
2.2. Onion Tunnel Accept

This messages is the expected response to the `ONION TUNNEL INIT` message. It contains 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`. This finishes the session establishment and allows for a subsequent encryption of messages between the initiator and the sending peer.



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 address and port the new peer is reachable on. 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 new peer. This LID allows the receiver of this message to create a mapping for further forwarding. The received data is sent to the specified peer.



2.4. Onion Tunnel Transport

The `ONION TUNNEL TRANSPORT` message is used to securely transmit arbitrary data over an already established tunnel. Therefore, it can possibly contain any other message type as data and is maintaining a fixed length in all cases in order to provide a defense against an attacker using traffic analysis to gain intel on anonymous communications.

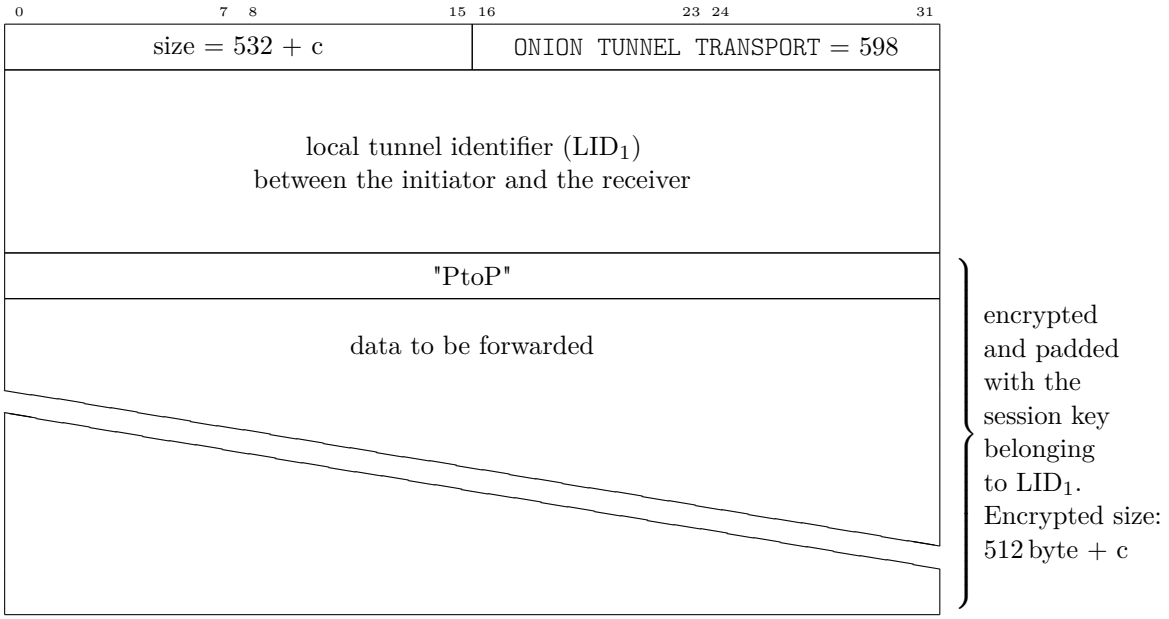
Our calculation provided with the interim report regarding the size of data such a message has to be able to contain, still holds. For this, we considered average human speech in combination with bitrates of common VoiP codecs. This was to make sure a single lost packet does not contain a crucial amount of voice data which would hurt the quality of the phone call. However, with the new changes introduced to this message type, we do not rely on a padding scheme like PKCS #7

any longer. Instead, the size of the inner real data is given explicitly by its length field and random padding data can be stripped, just like in TOR.

Every time an arbitrary other Onion P2P message type is to be encapsulated the new procedure for transport messages is:

1. The inner packet has to have a maximum size of 508 byte (including its header, larger voice data is fragmented).
2. A fixed four byte magic string is added (with that a peer can quickly check if he is the receiver).
3. The resulting 512 byte block is encrypted with either `AUTH LAYER ENCRYPT` or `AUTH CIPHER ENCRYPT` and its E flag set to 0
4. The resulting cipher data is put into a `ONION TUNNEL TRANSPORT` message after the LID.

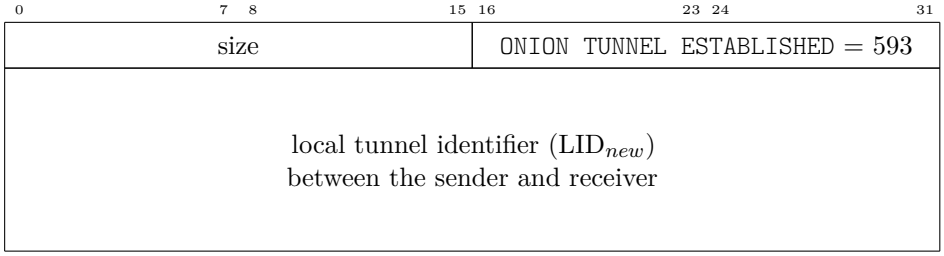
   Note: the cipher can be larger than 512 byte due to an overhead (c) the authentication module might produce due to IVs oder MACs. However, further en- and decryption is designed to keep this size constant. With a fixed plain input length of 512 byte to the authentication module's encryption mechanism, we are guaranteed to have a fixed message size.
5. Receivers that decrypt the message once, determine if they are left with plain data by checking the "PtoP" string.
6. Random padding can be stripped by parsing the length field of the contained message type.



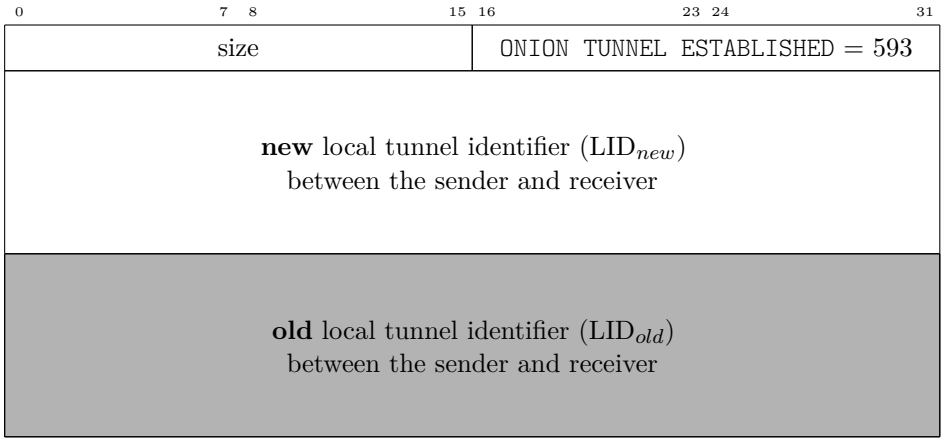
2.5. Onion Tunnel Established

The `ONION TUNNEL ESTABLISHED` message type is used to notify a peer that it is the final hop in a tunnel. This is necessary as peers have to send a `ONION TUNNEL INCOMING` message to their UI/CM in this case. Without a separate message type, a peer is unable to know about its role as an endpoint, as an extension of the tunnel towards a new peer can happen at any time. An alternative to this

message type would be to treat the first incoming voice data as a signal of being the final hop. However, a called peer might also want to start sending voice data before receiving any. Section 3.1 displays how this message is used during the tunnel building phase.

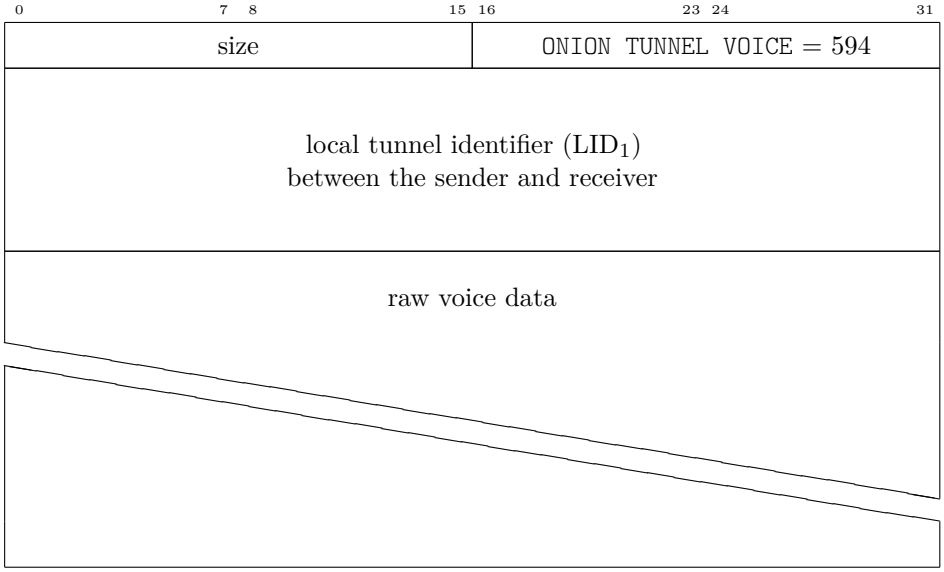


The `ONION TUNNEL ESTABLISHED` does also come in a second version including two local identifiers (additional, optional LID is marked in gray). This is used when we transparently switch over an ongoing call to a fresh tunnel. To do so, this message type indicates a replacement from the old to the new identifier to be mapped to an existing call. Section 3.4 displays how this message is used during tunnel switching.



2.6. Onion Tunnel Voice

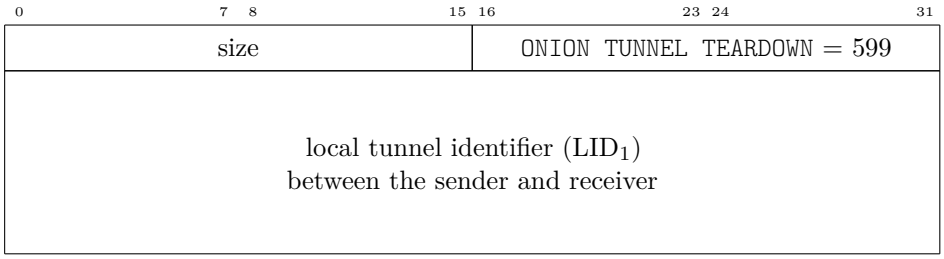
This message type is used to transmit voice data over an established tunnel. It is always encrypted inside a `ONION TUNNEL TRANSPORT` message. A peer retrieving this message from encrypting a transport message type expects to be a tunnel endpoint regarding the given LID. The included voice data can then be forwarded to the UI/CM module with the tunnel ID associated to the included LID in the peer's state.



2.7. 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 to a specific peer in the tunnel. Unencrypted and therefore unauthenticated tear downs would introduce a huge weakness to denial of service attacks on existing tunnels. To tear down a tunnel, this message is iteratively sent through the tunnel starting from the last hop.

If the endpoint of a tunnel is instructed to tear down a tunnel by its UI/CM module, he uses the same message type. This then propagates through the tunnel until it reaches the hop that identifies itself as initiator. The initiator is then able to destroy the tunnel as described (see sequence in Section 3.3).



3. Functioning

This section describes how the individual steps of tunnel handling (build, usage, teardown, transparent switching, cleanup) function in the final implementation. For each step, a protocol sequence with only one intermediate hop is given. This is only done due to the limited space on a portrait oriented page. Additional intermediate hops do not add any additional behavior, and only lead to more forwarding and layer encryption. The identical headers with size and type are abbreviated by a shorthand for the respective message types. The also identical local identifiers of each message type

are specified to make clear which LID is used.

Note that data inside a `transport` message after the initial LID is always layer encrypted with the next required decryption key being the one of the session associated with the specified LID. After that decryption, subsequent encryption layers up to the designated receiver follow.

3.1. Tunnel initiation

Tunnel building involves the iterative usage of `ONION TUNNEL INIT` and `ONION TUNNEL RELAY` messages which can be encapsulated in `ONION TUNNEL TRANSPORT` for equal size and encryption.

Figure 3 displays the sequence with one intermediate hop. Note that `ONION TUNNEL INIT` and `ONION TUNNEL ACCEPT` are transmitted in plain for each hop as there hasn't been a full handshake beforehand. Therefore, using padding to get a fixed message size is not helpful.

Note that the relay message contains the new local identifier (LID_2) as well. The intermediate hop does not only forward the inner message, but is also required to add the new LID to its state to be prepared to map responses with it, like the `accept` message, back to the initiator and LID_1.

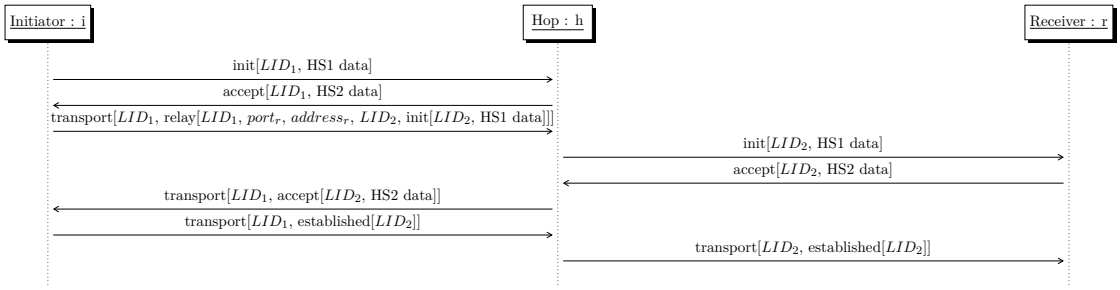


Figure 3: Protocol sequence for building a new tunnel.

After a session is established with the designated endpoint of the tunnel, an additional `ONION TUNNEL ESTABLISHED` message is sent. This notifies `r` to be the endpoint and triggers it to select a tunnel ID and to associate a tunnel object with the tunnel segment that has been created during the handshake. This enables the receiver to report the incoming tunnel to its superordinate module (UI/CM) and finish the tunnel establishment.

3.2. Sending data

Sending voice data over an established tunnel can happen after the `ONION TUNNEL ESTABLISHED` message as seen in the previous Section 3.1 has been sent. The process of sending voice data does not differ from sending any other data over hops to which a session has been established. If the initiator of the tunnel sends the message, the data inside the transport message is encrypted once for every hop that is part of the tunnel. This starts with the last hop and finished with encrypting the data using the key established with the first hop. That way, all intermediate hops can remove one layer of encryption using their key for this tunnel. Although Figure 4 only shows the LID which specifies the next encryption session key to be use, the initial `transport` message sent by the initiator is already encrypted twice.

For data that is sent by the receiver or rendezvous point, it is only encrypted once with the key known to this peer. Every peer in the tunnel than adds his layer of encryption until the whole

Onion encrypted data reaches the initiator who then has the knowledge of all keys to strip every layer again.

Figure 4 displays this message flow which is possible just after the previous tunnel build sequence shown in Figure 3 has been finished.

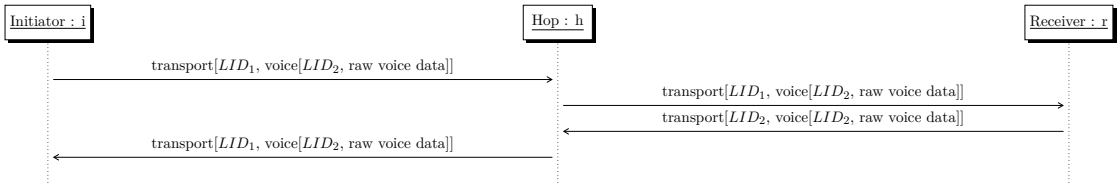


Figure 4: Protocol sequence for sending data over an established tunnel.

3.3. Teardown

The destruction of a tunnel is done iteratively and always started by the initiator. This is required as only the initiator knows the full list of peers in the tunnel and is able to start closing sessions from the very end. Starting the tear down with the last peer in a tunnel is necessary in order to be able to layer encrypt the teardown messages towards the last remaining peer.

As specified in the applications specification document, the rendevouz point at the end of a tunnel can also receive a `ONION TUNNEL DESTROY` message from its superordinate UI/CM module. To allow for the tear down process of a tunnel starting from this point, an `ONION TUNNEL TEARDOWN` message is propagated to the initiator who then starts the described procedure from his side.

Both versions of how a tunnel can be destroyed are shown in Figure 5.

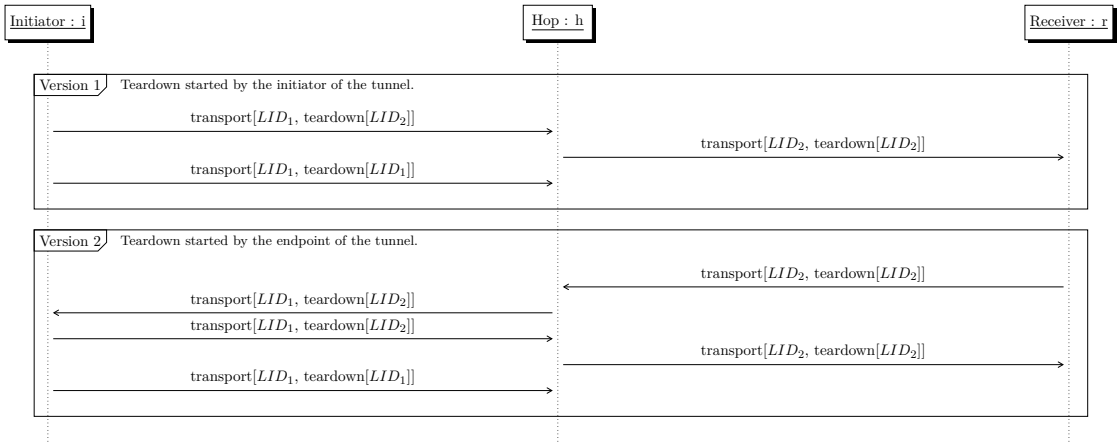


Figure 5: Protocol sequence for destroying a previously established tunnel.

3.4. Transparent switching of tunnels

The specification of the VoidPhone application states that:

> Tunnels created in one period should be torn down and rebuilt for the next period. [...] Onion should ensure that this is done transparently to the modules using these tunnels.

Our implementation achieves this by recreating tunnels a peer started as initiator to the existing destination over new hops on every new round. With this, some time of the round passes until the rebuilt tunnel is finished. This slight offset will also happen in future rounds so that the effective time a tunnel is used equals the timespan of a round.

New tunnels that are recreations of existing once, are finalized via the `ONION TUNNEL ESTABLISHED` message containing two local identifiers (see Section 2.5). This allows the receiver to map future data with a new identifier to an existing tunnel ID for forwarding to UI/CM.

The process of rebuilding a tunnel (shown in Figure 6) is therefore basically equivalent to the tunnel build sequence shown in Section 3.1 with the only difference being the established message.

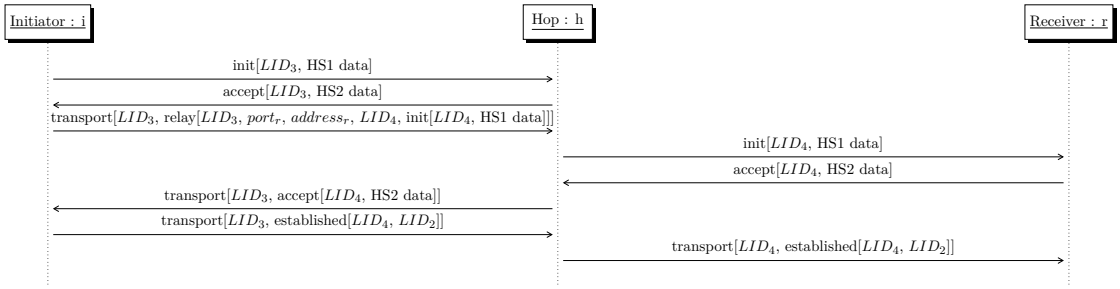


Figure 6: Protocol sequence for rebuilding a new tunnel, precisely the one build in Figure 3.

However, to achieve complete transparent switching, a problem remains regarding data that is being sent by the rendevouz point after the initiator has already switched its state and before the rendevouz point itself received the established message with the LID remapping. Figure 7 displays this issue in which, matching the sequence of Figure 6, LID_2 is replaced with LID_4 for the receiver.

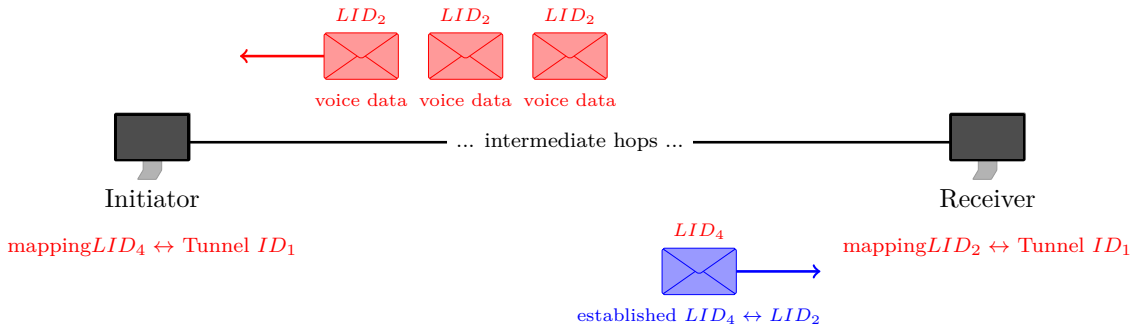


Figure 7: Issue with non-attributable old messages sent during rebuild of a tunnel.

To resolve this issue, the initiator holds a collection named `toBeDestroyed` that keeps the state of an old and now rebuild tunnel **until the first message with a new identifier arrives on his P2P interface**. This way, the initiator can be sure the receiver has received note of the refresh and all packets with old LIDs that come even later are out of order anyway and can be dropped. The initiator then cleans up this old state on the first witness of the new identifiers and also tells the intermediate hops to remove their state concerned with the old tunnel.

3.5. Cleanup of leftover states

Additionally to the cleanup and replacement of old identifiers in case of a switched tunnel, peers have be able to entirely remove states as well. Peers are required to be able to clean up data regarding

the state of a tunnel or a mapping they have in their role as an intermediate hop. This is especially important as tunnels can be broken by peers leaving the network at any point in time. To do so, every peer keeps track of the last time data has been seen associated to a certain LID. If this time exceeds a whole round, the state is being removed. The time check is done on every round start by the peer.

4. Security Properties

The Onion module's purpose is to provide security to the users of the VoidPhone application. We analyze the protection of sensible user data regarding confidentiality, authenticity, integrity and privacy in this section.

Privacy The main purpose of onion routing is to allow anonymous communication in the system. Every peer in a tunnel, except the initiator, only knows the addresses of its direct predecessor and successor. As a result, no peer, besides the initiator, knows both, sender and receiver address, if we are using at least two intermediate hops in a tunnel. In addition, we do not transfer any information that allows the identification of the tunnel initiator. Thus, the Onion module provides sender anonymity. The usage of a rendevouz point allows receiver anonymity, too.

We utilize layer encryption and have an identifier (LID) that is different for every tunnel segment. In addition, a `ONION TUNNEL TRANSPORT` message only contains the LID corresponding to the next hop. Therefore, a passive adversary is not able to directly track a tunnel through the system.

However, it is possible to track a tunnel during tunnel establishment using `ONION INIT` and `ONION ACCEPT` messages or by a timing attack. Such attacks are prevented if a lot of peers are sending data or establishing tunnels over the same re-router at the same time. I. e., we have to rely on a large anonymity set, to prevent timing- and pattern-based attacks.

By using the same round interval on every peer and synchronizing round transitions, we could increase the anonymity set during tunnel establishment. Nevertheless, this was not possible in this version of the Onion module, as the round interval has to be configurable regarding the supervisors.

The Onion module has at least one active tunnel at a time. This is achieved by building a cover tunnel if no user tunnel is established. In addition, the CM module is able to instruct the Onion module to sent random cover data through a tunnel which allows link padding. As we also refresh cover tunnel at round transitions and have fixed size packets, an attacker is not able to differentiate if a peer is currently communicating or not.

Authenticity The VoidPhone application uses pre-shared public keys for the identification of peers. In addition, the Onion module transfers two handshake messages during tunnel establishment. The onion authentication module can use those handshake messages and the pre-shared keys to provide authentication of the communication partners.

Confidentiality During tunnel establishment, the onion authentication module is given the possibility to exchange handshake messages. As those messages are exchanged between the initiator and all hops in the tunnel, a secure ephemeral key can be established for every tunnel segment, e. g. by the use of the Diffie-Hellmann key excahnge algorithm. As the Onion module layer encrypts all data sent through a tunnel using those keys, no user information is sent in plain text.

The Onion module builds a new tunnel for every user communication in every round. As a result, the onion authentication module has the possibility to update its keys in a fixed interval.

Integrity The onion authentication module is allowed to add additional information to a message sent through a tunnel. This information can be used to provide integrity protection.

5. Installation and Execution

In this section we describe how to install and run the Onion module. The required sources are available on our git repository[2].

5.1. Execution

We used `Java` as programming language for our Onion module. `Java 8` is required for execution. It is possible to either run the pre-built version contained in the `release/` directory or build from source as explained in Section 5.3.

Listing 1 shows the command to run the Onion module. The `config` command line argument specifies the configuration file that shall be used by the Onion module. We present the required configuration parameters in Section 5.2.

```
release/$ java -jar Onion_1.0.jar --config path
```

Listing 1: Command to execute the Onion module

Following command line parameters are currently supported:

help	—	Print the help text
config	path	Path to the configuration file
loglevel	level	Used log level (Optional)

5.2. Configuration

The VoidPhone application uses a configuration file in the Windows INI format. Table 1 lists all parameters required by the Onion module and their corresponding section. A sample configuration file is contained in the `config/` directory in the repository.

[2] `https://gitlab.lrz.de/ga78sil/p2p-2017-group17-onion`

Table 1: Configuration parameters required by the Onion module

Section	Parameter	Value	Description
onion	listen_address	<public_ip>:<port>	P2P address of this peer's Onion module
	api_address	<pubic/private_ip>:<port>	Address used for API connections to the Onion module
	hostkey	<path_to_pem_key>	Path to the key used by this peer
	round_interval	<interval_in_seconds>	Round interval the Onion module shall use
	intermediate_hops	<number_of_intermediate_hops>	Number of hops in the tunnel between this peer and the receiver
rps	api_address	<pubic/private_ip>:<port>	Address and port the RPS module uses for API connections
auth	api_address	<pubic/private_ip>:<port>	Address and port the ONION AUTH module uses for API connections

5.3. Compilation

Maven[3] is required for building from source, as it is used for dependency and build management. We tested the build using Maven-3.3.9 (Apache License 2.0).

The Java source code of the project is contained in the `src/` directory in the repository. Execute the command shown in Listing 2 in the `src/` directory, to build the project from source. After a successful build, Maven creates `Onion_1.0.jar` in the `src/target/ directory`. This jar file contains all necessary dependencies.

```
1 src/$ mvn clean package
```

Listing 2: Build the project from source

All unit tests and a large integration test that builds a tunnel, sends data and destroys it afterwards using mockups, are executed during build. To skip all tests use the command in Listing 3.

```
1 src/$ mvn clean package -DskipTests
```

Listing 3: Build the project from source without tests

5.4. Dependencies

Table 2 lists all dependencies of our implementation of the Onion module. As we are using Maven, all dependencies are downloaded during the build.

[3] `https://maven.apache.org/`

Table 2: Dependencies of the Onion module

Library	Version	License
jUnit	RELEASE (currently 4.12)	Eclipse Public License 1.0
Bouncy Castle	1.57	MIT X11 License
Netty	4.1.12.Final	Apache License 2.0
Google Guice	4.1.0	Apache License 2.0
log4j	2.8.2	Apache License 2.0
ini4j	0.5.1	Apache License 2.0

All dependencies are already integrated in the `jar` of the pre-built version. Therefore, they do not have to be downloaded separately.

6. Time Management

We tried to clearly separate the effort the team members spent to the project. In contrast to Figure 8, this was not always possible as we helped each other in implementing different parts and also fixed bugs in the other team member's code. Nevertheless, we have the opinion that both team members spent an equal amount of time and the work was distributed fairly.
The average effort spent on the project was about 4–5h per week for development, 20h for the reports and additional 30h in the final phase for each team partner. This corresponds to $13 * 4.5h + 20h + 30h = 108.5h$ of work for the project. In addition, we spent $(3h + 2h) * 13 = 65h$ on the lecture and follow-up and $35h$ on exam preparation per person. Therefore, each team member spent about $108.5h + 65h + 35h = 208.5h$ of effort for the lecture which is slightly higher than the expected workload of $180h$ for an 6ECTS module.

Figure 8 summarizes the work distribution during the project. In the following paragraphs we describe some steps of Figure 8 in more details.

Christoph

- Netty Server and Client: We decided to utilize Netty pipelines to handle network traffic. As we have to handle connections to other local modules and other onion instances, Christoph developed a wrapper to simplify message sending and receiving.
- Callbacks: We use a lot of callbacks to handle asynchronous communication between the modules. Therefore, we use Netty pipelines for receiving and message parsing. The orchestrator uses callbacks to react on those asynchronous messages and connects the different components shown in Figure 1.
- Internal Message Handling: Christopher implemented a large part of the internal onion logic, e. g. tunnel set-up.

Marko

- Message Parsing: Marko developed parsers for the interfaces to other modules. Each parser offers build methods for outgoing messages and a central parse method for incoming packets.
- Onion Message Parsing: Marko added a parser for the messages exchanged between two onion instances. In addition, some changes at the Netty pipelines and internal logic were necessary as the used UDP handlers blocked message reception.

- Unit: Unit tests ensure correct message serialization and parsing of messages exchanged between two onion instances.
- Integ: Marko developed mockups for all interfaces except the onion-to-onion interface to have an integration test for the internal logic.

Together We use this point to summarize all outcomes that we developed together.

- Inital/Interim/Final: All reports for this project were written by both team members.
- Correct Onion Protocol: The first version of the onion protocol, which was developed for the interim report, had some issues, e. g. with the implementation of equal size packets. Therefore, we had to set-up a new onion protocol (see Section 2).
- Internal Logic: We completed the internal message handling and introduction of the round concept together.
- Testing: We used the integration test implemented by Marko and a test set-up using real hardware and the mockups developed by the project's supervisors to test the internal logic.

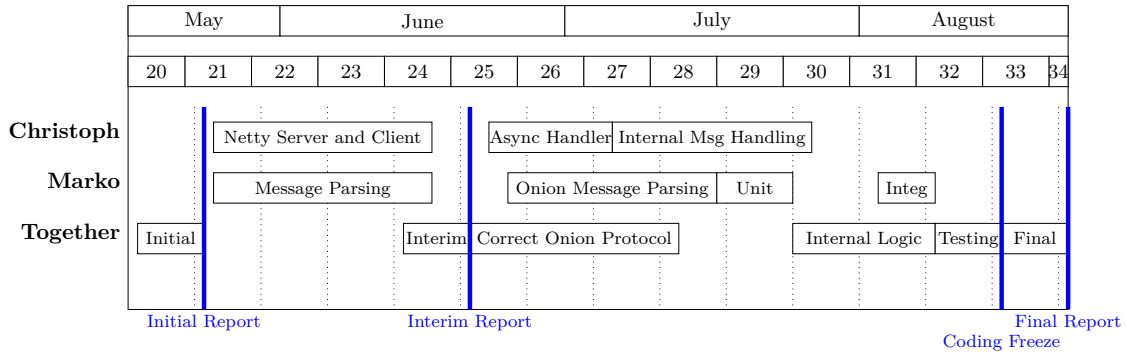


Figure 8: Work distribution during development

References

[1] S. H. Totakura, L. Schwaighofer, and G. Carle. Project Specification (P2PSEC SoSe 2017) – Version: 2017-4.0, Aug. 2017. Accessed: 2017-08-20.