Voidphone Project: Final Report

Oliver Jacobsen (03696596), Dominik Stolz (03696456)

August 28, 2020

Architecture

Logical Structure

Cargo – the default build system for Rust – allows to include a library as well as multiple binary artifacts in a single crate. So, we split our application into a binary handling API connections and a library servicing requests from these connections. This enables the possibility to reuse all of the onion functionality outside the context of the Voidphone project. For this reason, we tried to keep the public interface of the library as general as possible and confine all Voidphone specifics to the binary. This separation also reflects in our directory structure:

```
src
├── api
│   ├── config.rs
│   ├── mod.rs
│   ├── protocol.rs
│   ├── rps.rs
│   └── socket.rs
├── lib.rs
├── main.rs
├── onion
│   ├── circuit.rs
│   ├── crypto.rs
│   ├── mod.rs
│   ├── protocol.rs
│   ├── socket.rs
│   ├── tests.rs
│   └── tunnel.rs
└── utils.rs
```

The file `main.rs` contains the entry point for the binary and uses modules in the `api` directory. It also uses the included library, for which `lib.rs` is the root module. The library uses modules from the `onion` directory. The module `utils.rs` is used by both the binary and the library.

The library is in turn designed in multiple layers – the *P2P Protocol* constituting the bottom most one and the *Public Interface* the top most one. In between there is the *Circuit Layer* which is mostly concerned with relaying messages from one hop to the next. On top of the *Circuit Layer* resides the *Tunnel Layer* which represents the end-to-end view. The table in figure 11 shows an overview of all logical layers with the corresponding source files. The *Onion Socket* layer serves as glue between higher-level intentions, such as initiating a handshake, and actually writing serialized packets to a socket. The purpose of the *Round Handler* is to track all outgoing and incoming tunnels and schedule operations requested by the *Public Interface.*

Layer	Source File
Public Interface	`lib.rs`
Round Handler	`lib.rs`
Tunnel Layer	`onion/tunnel.rs`
Circuit Layer	`onion/circuit.rs`
Onion Socket	`onion/socket.rs`
P2P Protocol	`onion/protocol.rs`

Figure 1: Overview of layers and source files

Socket Layer

The socket layer serves as an abstraction of network-layer connections and as an interface for the protocol packet implementations.

Circuit Layer

A circuit is defined as the direct connection between two peers and is part of a tunnel involving multiple hops. In the code it is represented with the `Circuit` struct (cf. figure 2) which stores an unique identifier and a socket for communicating with the other peer.

```
struct Circuit {
    id: CircuitId,
    socket: Mutex<OnionSocket<TcpStream>>,
}
```

Figure 2: Circuit struct

For each incoming connection, a key exchange is performed first before handling arriving messages. Here it is important to note that keys are not necessarily exchanged with the connecting peer, but with the peer who is building the tunnel. After a successful key exchange, a `CircuitHandler` struct (cf. figure 3) is created, which also stores the negotiated session key.

```
struct CircuitHandler {
    in_circuit: Circuit,
    session_key: SessionKey,
    state: circuit::State,
    // omitted other fields for brevity
}
```

Figure 3: CircuitHandler struct

The `CircuitHandler` can be in one of three states:

```
enum State {
    Default,
    Router {
        out_circuit: Circuit,
    },
    Endpoint {
        tunnel_id: TunnelId,
        requests: Receiver<Request>,
    },
}
```

Figure 4: circuit::State enum

In the `Default` state, the handler just waits for a message instructing it to either extend the tunnel and become a `Router` or to be the `Endpoint` of a tunnel. In the `Router` state, the handler forwards messages in both directions until the tunnel is truncated or destroyed. In the `Endpoint` state, the handler connects its `in_circuit` to the *Tunnel Layer*.

Tunnel Layer

The *Tunnel Layer* is further subdivided into outgoing and incoming tunnels. In code, incoming tunnels exist implicitly as `CircuitHandlers` in the `Endpoint` state. Each outgoing tunnel built via the public interface is managed by a `TunnelHandler` which also handles the switch-over after each round. It associates the abstract concept of a tunnel persistent across many rounds with a concrete tunnel, which is represented through the `Tunnel` struct (cf. figure 5). This struct stores the circuit to the first hop in the tunnel as well as a list of the session keys negotiated with each hop.

```
struct Tunnel {
    id: TunnelId,
    out_circuit: Circuit,
    session_keys: Vec<SessionKey>,
}
```

Figure 5: Tunnel struct

A `TunnelHandler` (cf. figure 6) stores the concrete tunnel it is currently communicating over and the tunnel which will be used in the next round as `next_tunnel`. The tunnel for the next round will be constructed asynchronously during each round. At the end of a round, the `TunnelHandler` is signalled to replace its currently active tunnel with `next_tunnel`. A `TunnelHandler` can be in one of four states (cf. figure 7) beginning in the `Building` state. After the first switch-over it transitions into the `Ready` state and sends a corresponding event to the API. Upon receiving a request to destroy the tunnel, its state will be set to `Destroying` from which it will transition to `Destroyed` after one final switch-over. An overview of these transitions is illustrated in figure 8.

```
struct TunnelHandler {
    tunnel: Tunnel,
    next_tunnel: Option<Tunnel>,
    state: tunnel::State,
    // omitted other fields for brevity
}
```

Figure 6: TunnelHandler struct (simplified)

```
enum State {
    Building,
    Ready,
    Destroying,
    Destroyed,
}
```

Figure 7: tunnel::State enum

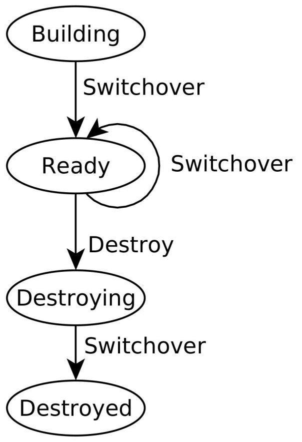

Figure 8: Layered view of the architecture with communication channels

Round Handler

The `RoundHandler` (cf. figure 9) keeps track of all outgoing tunnels built by the API and periodically signals them to switch-over to a new tunnel. To prevent idle tunnel connections from timing out, it also ensures keep alive messages are sent. When there are no tunnels, it constructs a cover tunnel on which cover traffic can be sent when requested by the API.

```
struct RoundHandler {
    tunnels: Map<TunnelId, Sender<Request>>,
    cover_tunnel: Option<Sender<Request>>,
    // omitted other fields for brevity
}
```

Figure 9: RoundHandler struct (simplified)

Onion Module and RPS Module

The `OnionModule` (cf. figure 10) is part of the binary part of this project. It forwards API requests to the library and library events to the API. For that reason, it needs to keep track which API clients are interested in which tunnels.

```
struct OnionModule {
    connections: Map<SocketAddr, ApiSocket>,
    tunnels: Map<TunnelId, Vec<SocketAddr>>,
    // omitted other fields for brevity
}
```

Figure 10: OnionModule struct (simplified)

The `RPSModule` is used to request and buffer random peers from the RPS module to avoid any bottleneck issues. Random peers are provided as a stream to the library, which uses them during tunnel construction.

Utilities

Some cryptographic functionality is shared across all layers and hence extracted into the `crypto.rs` module. The `config.rs` module is used by the binary to read and parse the configuration file.

Process Architecture

Tasks and Channels

The process architecture of our application is based on a single asynchronous Rust program spawning various concurrent tasks. These tasks are executed on an event loop provided by the `tokio` crate, which internally uses a thread pool for scheduling. The main tasks are:

- For each incoming API connection a handler task is spawned, which calls the corresponding methods in the library.

- The library spawns a `RoundHandler` task which keeps track of all created tunnels, coordinates switch-over and creates a cover tunnel if needed.

- The `OnionListener` waits for incoming peer-to-peer connections, for each of which it spawns a `CircuitHandler` task.

- A `TunnelHandler` task is spawned for each created tunnel, to manage a tunnel over many rounds including seamless switch-over.

Communication in the Public Library Interface between tasks is done in a message passing fashion using channels. Synchronization using message passing often allows to avoid shared state, which is a common source of race conditions and related bugs.

For instance, calling the `build_tunnel` method in the library sends a `Build` request over a channel which is then consumed by the `RoundHandler`. To notify the API on incoming tunnels or received data, another channel is used, which propagates events up from the circuit and tunnel handlers.

An overview of the communication schema is shown in figure 11.

Networking

Networking is also done in an asynchronous manner using the methods provided by `tokio`. A call to `read` creates a future, which can be awaited inside of a coroutine. The `tokio` runtime then uses `epoll` to resume the coroutine once the I/O operation completed.

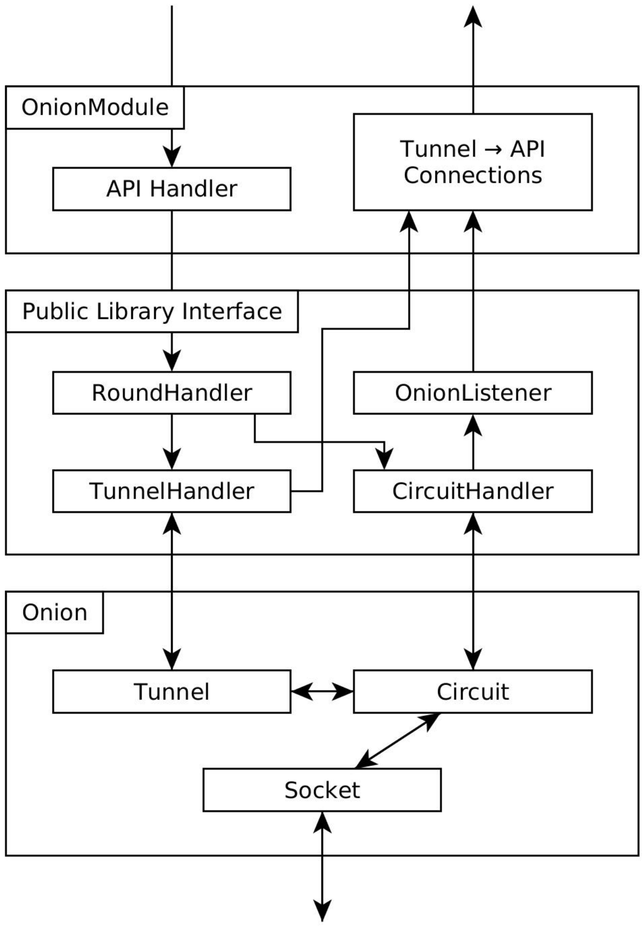

Figure 11: Layered view of the architecture with communication channels

Software Documentation

Please refer to the `README.md` file accompanying the source code.

Future Work

Please refer to the `README.md` file accompanying the source code.

Workload Distribution

While Oliver was especially concerned with error handling and the onion protocol, Dominik focused on implementing the API protocol and designing the concurrency model. However this is only a rough separation as we generally distributed the workload evenly among ourselves. Instead of designating exclusive areas of focus to each team member, smaller tasks were dynamically assigned on a by-need basis. This also ensured, that both of us maintained an overview of the project and were able to contribute to each part.

To put the effort spent in numbers, one can take the Git commit history into account, counting over 100 commits since May 8th. Some of larger refactorings were approached in pair-programming and thus were only committed from one account.

- Dominik: ca. 100 hours
- Oliver: ca. 100 hours