Report 3: Final Report

Jonatan Juhas

1 Documentation

1.1 Dependencies

The application is setup to use cargo (rust's package manager) which allows for simple managing of dependencies all of which will be downloaded automatically upon building. The app uses a few unstable rust features which require the nightly build. For the easiest setup we recommend using rustup[1]. During installation select the nightly build (cargo will be installed along rust)[2].

1.2 Building the app

Running the app in development mode is as simple as calling `cargo run -- -c` followed by the full path for the configuration file. For building the app use `cargo build --release` which will compile the application into an executable located inside `./target/release/`.

1.3 Issues while running the application

DISCLAIMER: The application does not fulfill the requirements of the specification

Due to various problems with designing a robust solution for connecting the event-loop cycle and the network protocol, are these two application parts NOT connected. The issues will be discussed in following sections. This means, the application will upon starting, get stuck in an infinite loop waiting for connections, unable to relay them to the event loop.

2 The Communication Protocol

All messages carry an ID number which marks this session's tunnel. Each ID is mapped to a thread ID which is responsible for this tunnel. That is each tunnel has its own thread. The main loop is responsible for mapping and dispatching ID numbers to threads. These are the messages that can be exchanged in the Protocol:

[1] `https://www.rustup.rs/`

[2] If you are working on Windows refer to `https://github.com/rust-lang-nursery/rustup.rs/blob/master/README.md#working-with-rust-on-windows` for more information.

2.1 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.

2.2 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).

2.3 Handshake

This message facilitates the handshake between the 2 nodes. It is part of the Auth mechanism. It is used for both the request (HS1) aswell as the response (HS2). The participating nodes are expected to keep track of which part of the Handshake is currently in progress.

2.4 Forward

This message requests the next hop to forward this message to the peer specified in the message. It carries the **Handshake** as its payload 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.

2.5 Incomming

The origin node packs the payload (real or cover data) into this message type and then encrypts it 'onion-wise' (via Auth) into a layered **Data** message type. All intermediate nodes know is that data is being transmitted. The last node decrypts the message type, extracts the payload and sends it over API to CM/UI.

2.6 Data

This message is simply used to transmit data encrypted by Auth. It is to be send to the next hop in the tunnel given the tunnel ID. The message types **Forward** and **Incomming** are send as its payload so that intermediate peers only see this message type and its encrypted content.

3 Problems with designing the Application

3.1 Network Connection

Designing the protocol in such a way that would facilitate TCP and UDP connections undermined our original implementation plan. We've decided to use TCP as a starting point but soon realized that separating the handshake from the actual data sending via UDP is not going to work reliably as the intermediate nodes do not and should not know if the transmitted data is part of a handshake or normal data transfer. The idea of using TCP only on part of the connection, as in transmit everything via UDP and only use TCP between the last hops, was discarded as it would throw away the benefits of both UDP and TCP. The current implementation therefore has UDP methods but is designed to use TCP for every data transfer. Since we (I) did not have prior knowledge in programming networking application this turned out to be a very tedious task[3].

3.2 Event Loop

The approach was to have a main event loop which would listen for incomming connections. Every connection would then be dispatched to a thread if it already is present in a global HashTable which would map message IDs to thread references (more specifically to thread channels that each thread would provide as means to communicate with it). Or if no thread exists which would accept this message ID a new one is spawned. This thread then either creates a new entry in the global reference table or signals an error if the message it received is not part of a standard protocol course[4].

The problem with creating this global reference table is difficult with rust's lifetime expectancy's as the table must be atomic, allocated on the heap as it's size will change and live for the entire time of the program[5]. The combination of all these problems resulted in a difficult setup which would need a global dispatcher taking care of managing access to such a resource.

The other problem with this approach was the complexity of associating messages with threads. Since messages are parsed by the message module they already contain all necessary information, but selecting which one to use for mapping is not trivial if the selection should be generic. A typical example is the P2P messages having an ID, the API messages for CM/UI a tunnel ID and the Auth messages a session ID.

3.3 Tying it all together

Combining these 2 problems resulted in the application not being able to communicate properly with its environment.

[3]The code responsible for creating network connections is inside the `brunch.rs` module

[4]The method for creating new threads and dealing with the initial choice of next action is `spinup_state_machine`

[5]This is a requirement of rust's compiler: Variables that might be accessible from other threads must have a static lifetime as other threads might still be running after the main thread dies.

4 Future Work

The app already features a robust and easily extensible error logging mechanism. Features such as logging to a file would be beneficial for analyzing network problems and could even be easily implemented.

A misbehavior detecting mechanism which would mark nodes that behave strangely (either take too long to connect, often disconnect from the network, drop packets . . .) was planned. Such nodes would be marked as untrustworthy and the origin node would try to avoid connecting to such nodes. Proof of bandwidth concept could be introduced, to for example further pose requirements on transfer speed for nodes.

5 Work division

I was the sole member of my team and therefore did all the work researching the topics, learning and programming in rust as well as writing reports.

6 Effort spent

The most difficult part, hands down, was programming the application. Coming up with the design and creating the documentation during the interim report was relatively easy. Rust's huge advantage is its amazing capability of detecting errors before the code compiles. I often found myself wondering why a certain part reported a mistake and wouldn't let me compile, only to after find out an object pointer would no longer be valid. But this was also the one thing that slowed me down considerably. I spend around 40% of the time writing code and 50% getting it to compile. Borrow checker requirements[6] and method signatures that sometimes spanned 3 lines were not uncommon. At one point I even encountered a compiler bug in the nightly build that crashed the compiler on a particular method signature I wrote (the issue on Github is still not resolved).

Some other unfortunate realizations were, for example, rust not supporting asynchronous IO operations. This was not a problem for direct connections to a target, but for the listener implementation a timeout was necessary. This would facilitate closing the listening stream once the application ends or killing an unresponsive listener. For this reason I had to switch to the low level API provided by the mio crate (rust's name for packages), but implementing its features instead of native methods was more complicated.

All in all, I spent around 15 days in total coding the application, not accounting for long reads of rust manuals and guidelines though.

[6]Rust's mechanism for associating an owner with a variable and preventing dropping a variable if references to it still [could] exist

Contents

Acronyms

Auth Onion Authenticate. 2

CM/UI Client/User Interface. 2, 3

Garlic Garlic (Onion Forwarding). 2