Storm Project Report

COMP-4300 Computer Networks
14, Dec. 2022

Storm Repo: https://github.com/StoicDeveloper/storm

Goals

My goal for this project is to create a basic command-line messenger application using the Rust implementation of the bramble protocol stack. This project carries on from my work in the ongoing COMP-4560 Industry Project, in which I implement the bramble-sync protocol in Rust. The other Rust implementations of the bramble-stack protocols and utilites were created by Martinho Fernandez, my supervisor for COMP-4560. The scope of this project does not include the implementation of those protocols, except for the rendezvous protocol (Martinho only wrote the interface for that) except as far as any problems are found in those implementations which were revealed only through attempting to complete this project. The original specifications for the bramble protocols were created by the makers of the open source Briar application, and are available here: https://code.briarproject.org/briar/briar-spec

I will be using three protocols: bramble-rendezvous, bramble-transport, and bramble-sync. Each of these builds on the earlier protocol by exposing a socket interface. Rendezvous sockets can be passed to transport, and transport sockets can be passed to sync. My goal is to create a messenger application that can accept the public keys of peers, use them to create rendezvous sockets, which will be wrapped in transport sockets, and which can then be passed to sync to synchronize messages in groups.

Technology

As mentioned, the three most relevant protocols are bramble-rendezvous, bramble-transport, and bramble-sync.

Bramble-rendezvous abstracts the TOR network by allowing participants who have shared their public keys over some other medium to create a shared secret. This shared secret is broken into two public keys that are then used to generate Onion Service URLs. These URLs are then known to both of the contacts, and no one else. How the Tor network and connections over Onion Services works is complex, and available on the TorProject website, or the official specification documents. Each connecting peer creates an Onion Service at one Onion URL and attempts to connect to the other over a SOCKS proxy. If a connection is established before a timeout is reached, then the a TCP connection is created through which the peers can communicate, and which has all of the anonymity benefits that the TOR network provides.

Bramble-transport accepts a socket-like connection from a lower level (in this case it will be a rendezvous socket), and creates a new socket with additional guarantees about message authentication and integrity. The peers' public keys are used to generate a stream of tags that are appended to the encrypted messages. Each stream appears random, but for two peers, the sequence of tags will be the same, such that any message injected into the stream will not have a correct tag, and any intercepted message can be detected since a tag will be missing from a received message sequence.

Bramble-sync is the first application-layer protocol. It is responsible for synchronizing groups which contain graphs of messages on behalf of its clients. These clients determine when groups are created, with which peers the group is shared, what constitutes a valid message, and which messages should be shared. The protocol carries out these decisions to the specification of the clients. If an application is actually built using the bramble-stack, then the logic of the application is contained within one or more sync clients.

This project is interesting because this stack of protocols, once connected, can be used to create arbitrary secure, anonymous distributed applications.

For development tooling, I used the default Rust Analyzer tools, integrated into Vim with the Coc plugin. This tool, along with the included RustFmt, handles code completion and formatting. For linting no external tool is needed, as the Rust compiler has extensive warning options that contribute to code neatness.

Components

There are several components to this project worth examining. The most obvious is the storm-cli directory, which contains the source code for the CLI. Examining its main.rs file will show how the CLI works and what commands are available, which represent a narrow subset of the behaviour which a true messenger would have. For example, contacts can be added, and added to groups, but they cannot be deleted through the CLI.

The CLI mainly interacts with the bramble protocol stack through the controller files in storm-backend. Storm-backend also holds the clients and plugins directories which are for future use, currently only the default bramble-sync 'simple_client' is used. The controller files include tor.rs, which contains some utilities related to tor, including a minimal implementation of a Tor Controller (I couldn't get the library 'torut' working, which is a more comprehensive Rust Tor Controller), and the implementation for TorSocket, which is the struct that my implementation of bramble-rendezvous produces when making connections. That implementation is contained in rendezvous.rs. The file tor.rs also includes some basic tests related to starting up the TOR process, however they were written during an early stage of development and no longer work; the functionality they tested is now covered by the controller.rs tests. I've left the non-function tests in with an 'ignore' tag for instructive purposes.

The most important controller file is controller.rs. This file is responsible for creating connections with peers, and for starting and running the sync protocol and its clients, and receiving output from the latter. At the bottom of the file are its tests, some of which don't work due to flaws in the protocol implementations.

The storm-frontend directory contains the basic structure of the bridge between a storm controller and a flutter frontend. It is not relevant to this project, which only uses the CLI.

Difficulties

Numerous difficulties were encountered during the course of this project that limited its success. These can be broadly categorized as 1) Difficulties due to a lack of familiarity with Rust 2) Difficulties due to a lack of familiarity with Tor and the involved networking components, such as SOCKS, and 3) problems with the protocol implementations themselves.

Having written bramble-sync in Rust, I am now relatively familiar with the programming language, however there are still many detail that I'm not used to. I had some difficulty getting the CLI input/output to work correctly, and in determining whether there was an existing library that would help. I spent some time setting up the Shellfish library that is used to create CLIs, but because Shellfish async commands access a state object that must be able to move between threads; and my StormController object cannot meet that requirement, that time ended up being wasted. In the end I did not use a CLI library, instead I parse the input directly. This shows in that the output is sometime janky, such as when Tor creates output that overwrites the prompt. In that case the CLI is still usable but doesn't look right. However, getting it to look good wasn't a priority.

Setting up Tor was difficult. Eventually I found an adequate asynchronous SOCKS library that enabled me to make TCP connections to an Onion Service URL. Some of the sources I found useful in figuring out how to do that are listed at the top of controller/tor.rs.

This project revealed some problems in the existing bramble implementations. Bramble-rendezvous was only previously an interface, which I implemented in rendezvous.rs. Bramble-transport was fully implemented by my COMP-4560 supervisor Martinho Fernandez, however I was not successful in wrapping TorSocket rendezvous connections into bramble-transport sockets. The controller.rs test “wrap_rendezvous_in_transport” demonstrates this failure. When the below command is run, the output shows that data is written to the write-end of the transport sockets, but the connection will hang forever; the read-end will never notice that data has arrived. I attempted to use Wireshark to determine whether the data was actually sending and receiving over the network, but due to the large amount of Tor traffic, that investigation was not revealing. Here is the command:

```
$ RUST_LOG=TorSocket,rendezvous,network cargo +nightly test wrap_rendezvous – --nocapture
```

There is a further problem whose exact cause I have not been able to determine. While the CLI works well for exchanging messages, when too many messages are being synced at one, several read ‘futures’ in the sync protocol will hang. These represent in-progress reads from peers. Even though data has been sent, the receiving end will not receive that data, even though the socket is still open and shows no errors. This problem can be demonstrated with the test “exchange_many_between_two” and “exchange_several_in_small_network”. The other tests in controller.rs are successful however.

Accomplishments

I have succeeded in creating a basic messenger application that connects to peers over the TOR network with my implementation of bramble-rendezvous and synchronizes groups of messages using bramble-sync; the latter was created as part of COMP-4560 and is largely not intended as part of the submission for this project, though I did make improvements to bramble-sync when its uses revealed problems that the mocked tests contained in that library could not reveal. In particular, I wrote the bramble-sync file sync_states.rs as a more complex method of maintaining the state of each message’s synchronization with each sharing peer as a solution to race conditions that only appeared when messages were transmitted over Tor’s high-latency network, but had not appeared when using the bramble-common Duplex object, which simple stores read-write data in a buffer.

Only the most basic bramble-sync functionality is demonstrated through this CLI. It allows peers to be added, groups created, peers added to groups, and messages sent, which are then received by peers, if the peer is also in that group (ie. the group must be created by both peers, a future version may include an “invite” feature, where a special message is sent over DM, which will then trigger the specified group to be created). Stored messages that are unsynchronized will also be synchronized when new connections are made (that is, a message can be created either before or after a connection is made; the result is the same. Functionality that is not included is creating messages with dependencies, removing peers from groups, deleting groups, etc.

Demonstration

On debian 10, the following instructions work to run the demo.

```
Sudo apt-get install curl
curl https://sh.rustup.rs -sSf | sh
git clone https://github.com/StoicDeveloper/storm.git
cd storm
./storm-backend/src/controller/setup.sh

rustup toolchain install nightly
```

```
cargo +nightly run -p storm-cli
```
To send messages using the CLI, first login.
```
$ login <name>
```
Then find your key.
```
$ key
```
Use the key from another, simultaneously running CLI on another VM to add a peer.
```
$ add <name> <hex_key_encoding>
```
Wait for the connection to be made. This may be quick or it may be slow. It may also not work at all, in which case either restart, or wait for the Tor network to have lesser load.
Create the same group on both peers.
```
$ newgroup <group_name>
```
Add the remote peer to the group.
```
$ share <group_name> <other_peer>
```
Enter the group.
```
$ entergroup <group_name>
```
Send a message
```
$ send <message>
```
See the message appear on the CLI of the remote peer.

I’ve enabled a logger in Storm in case you’d like more insight into its inner workings. To use it you can run the CLI with a prefix of RUST_LOG=<comma delimited target list>. Available targets include those in the following example:

```
$ RUST_LOG=interactive_operations,rendezvous,network,sync_states,wire_events cargo +nightly run -p storm-cli
```

There are more, but these are the most informative.