funion

A Tor Client in Elixir

About me

- Emil Engler (any pronouns)
- Graduated from high school this summer
- Studying pure mathematics and CS in October
- FLOSS-Contributor since 2018 (cURL, Rosenpass, …)
- Tor since 2022

- Fedi: `@engler@chaos.social`
- GitHub: `@emilengler`

Acknowledgements

- Many thanks to Alexander Færøy
  - Very supportive since I've first announced it
  - Invited me to do the talk here
  - Motivated me to continue working on it
- Many thanks to the Talla authors
  - Talla is a Tor relay implementation in Erlang
  - Helped me a lot with the Erlang crypto module and in understanding the Tor specifications
- Many thanks to the people in `#tor-dev`
  - Nick Mathewson, Roger Dingledine, Ian Jackson, Trinity Pointard, and others (no particular order)

Summary of my work

- Wrote a Tor Client (OP) in Elixir
- Implements only the core Tor network protocol
  - Cool kids call this `tor-spec.txt`
  - Also implements `cert-spec.txt`
- More of an educational project to learn from
- Still lacks many features
  - Directory support, path selection, hidden services, OR, …
  - No proper SOCKS proxy
  - Probably vulnerable to side-channel attacks
  - **Hackers welcome! I'll pay the Club-Mate**

Motivation


WRITING TOR CLIENT
PARTY ALL TIME
LEARNING FOR FINALS
imgflip.com

Motivation

- Finished school, lots of time before finals
- Background with low-level C and Rust programming
- Extreme OpenBSD diehard as a teenager

- Began contributing to Arti in autumn of 2022
- Saw Computerphile video on Erlang
- Decided to learn Elixir, functional programming, and Tor with this

What is Elixir/Erlang and why it matters

- Elixir and Erlang are functional languages compiling bytecode to BEAM
  - Similar to Kotlin/Java and JVM
- Extremely fault-tolerant; write once, run forever
- Emphasises **process separation** and **isolation**
- Makes IPC and process handling extremely easy
  - Makes the entire Unix process model look like garbage
- One of the most underrated technologies out there

Traditional approach vs. Elixir

Traditional application:

- One monolithic process
- Handling multiple connections through one event loop or async
- Event loop destroys linear execution
- Async is the herpes of programming

- No unique processes/threads per connection
  - Very expensive (8MB stack for each)
  - Notable exceptions: OpenSSH, Postgres

Elixir:

- One process per connection
- Not real OS processes, very cheap
- Offers isolation
- Makes code easier to understand
- Often makes code more secure

Isn’t this like Goroutines?

- Goroutines and Elixir processes are very cheap and similarly implemented
- Goroutines are threads, Elixir processes are processes

- Threads can access and manipulate all the resources inside the process
- Processes are isolated and cannot manipulate foreign state
  - No hurdle with mutex locks
  - **Extreme security benefit!** Reduces vulnerability impact significantly

A view from 10.000ft

- Developed using Elixir 1.15
- Supports link protocol version 4 of Tor only
  - Still lacks legacy/niche features, such as the TAP handshake
  - Lacks certain optional features and/or recommendations
- Consistent typespecs across the codebase
- Very modular design inspired by Arti
  - `tor_cell` – Implements (some) cells as Elixir structures
  - `tor_cert` – Implements the Tor Ed25519 certificate format
  - `tor_crypto` – Implements crypto primitives (e.g. ntor handshake, onion skins)
  - `tor_proto` – Implements the Tor protocol

A view from 10.000ft

- Developed using Elixir 1.15
- Supports link protocol version 4 of Tor only
  - Still lacks legacy/niche features, such as the TAP handshake
  - Lacks certain optional features and/or recommendations
- Consistent typespecs across the codebase
- Very modular design inspired by Arti
  - `tor_cell` – Implements (some) cells as Elixir structures
  - `tor_cert` – Implements the Tor Ed25519 certificate format
  - `tor_crypto` – Implements crypto primitives (e.g. ntor handshake, onion skins)
  - `tor_proto` – Implements the Tor protocol

`tor_cell` – Handling connection cells

- Implements the cells, Tor's basic unit of communication
- Parsing takes place with pattern matching
  - Elixir's unique feature of parsing protocols within the language
- A cell has three fields
  - `circ_id` – The circuit ID (32-bit integer)
  - `cmd` – An atom containing the command (e.g. :create2, :relay, :netinfo)
  - `payload` – A field containing the parsed cell (e.g.: %TorCell.Create2, %TorCell.Netinfo, …)
- Performs no cryptography, not even `CERTS` validation

`tor_cell` – Handling relay cells

- Decrypted relay cells are implemented within `%TorCell.RelayCell`
- Once encrypted, their ciphertext (onion skin) goes into `%TorCell.Relay`
- Contains three fields
  - `cmd` – The command of the cell as an atom (e.g. `:extend2`, `:begin`)
  - `stream_id` – The stream ID as a 16-bit integer
  - `data` – The data of the relay cell (e.g. `%TorCell.RelayCell.Extend2`)
- Has two public functions
  - `decrypt(onion_skin, kbs, db) :: {bool, %TorCell.RelayCell | onion_skin, db}`
  - `encrypt(relay_cell, kfs, df) :: {onion_skin, df}`

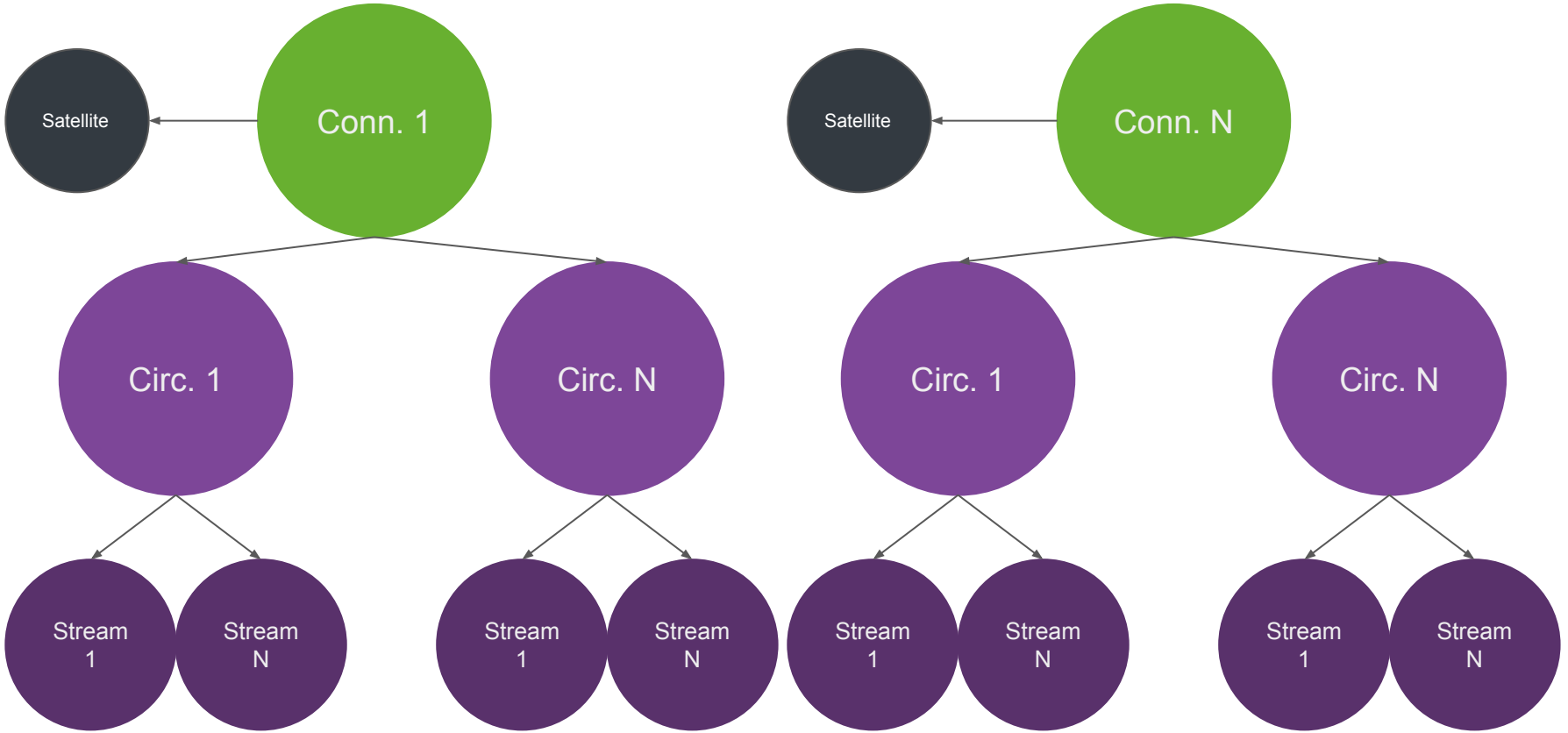
`tor_crypto` – Performing the low-level cryptography

- Performs various cryptographic operations required for Tor
  - The ntor handshake
  - A small abstraction for the digest
  - The encryption/decryption of onion skins, called onion stream
- Pretty small, only about 150 LOC

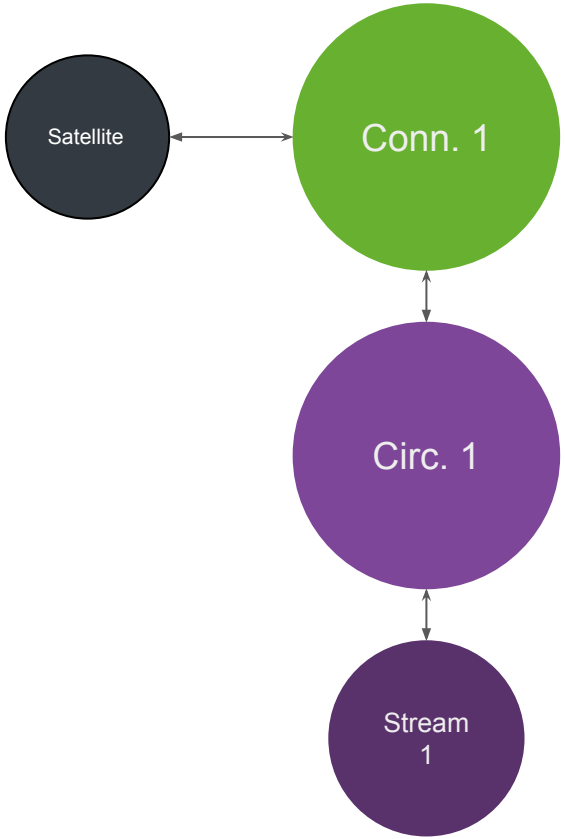
`tor_proto` – The actual Tor implementation

- Implements the actual Tor protocol
- Implemented using GenServer, providing a client API
- Highly process oriented, very resource separated

tor_proto – The actual Tor implementation



`tor_proto` – The actual Tor implementation



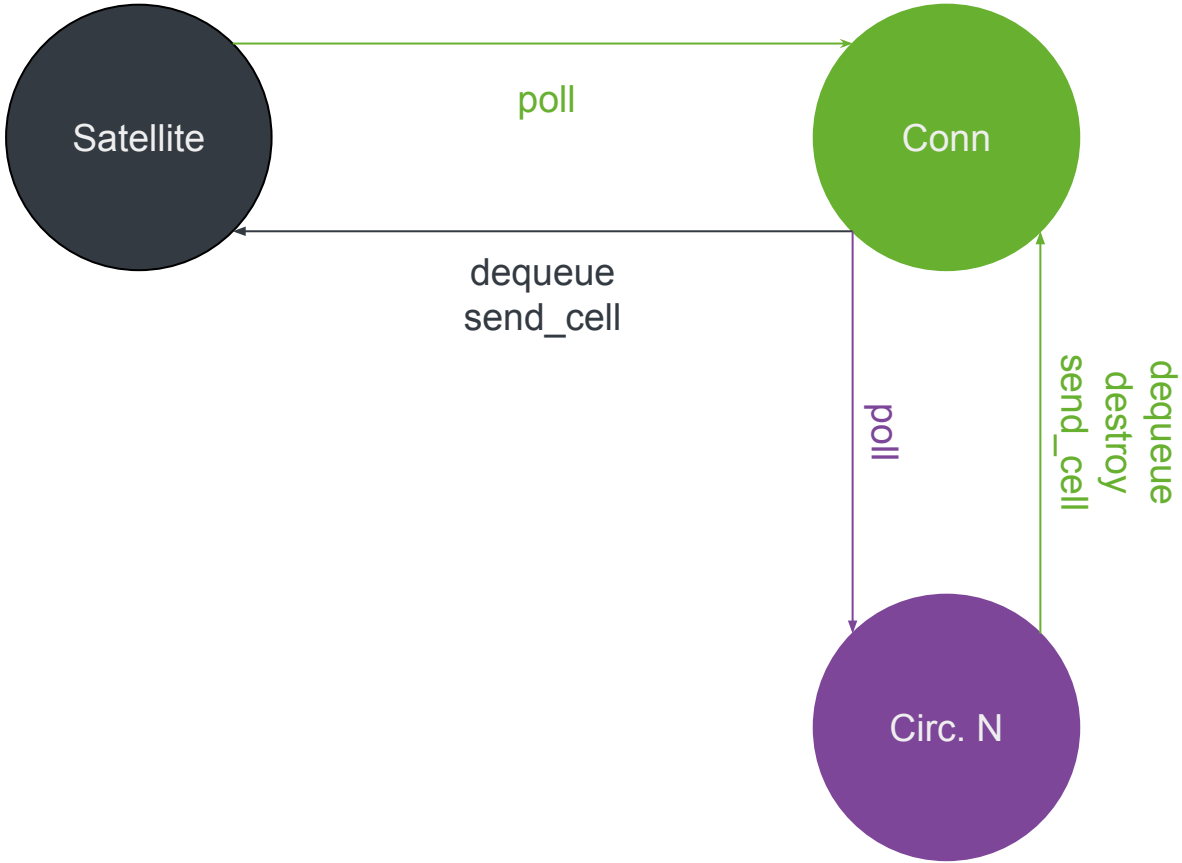
`tor_proto` – PidFifos

- Problem is: The connections and circuits need to queue the incoming cells for the appropriate circuits and stream respectively
- Each connection and circuit has several circuits and streams respectively
- The PidFifos data structure is map with the keys being PIDs and the values being ordinary FIFOS

`tor_proto` – The connection process

- Manages a connection/channel of the Tor protocol
- Spawns the circuit processes (create call)
- Spawns a single “satellite” process, that handles the raw TLS socket
- Gets polled by the satellite process, every time a new cell arrives
  - Those cells are either processed or enqueued for the appropriate circuits
- Polls the appropriate circuit if a cell is received
- State:
  - circuits (Map of all circuit IDs with their PIDs)
  - fifos (The enqueued cells for the circuits)
  - router (The onion router we are connected to)
  - satellite (The PID of the satellite process)

tor_proto – The actual Tor implementation



`tor_proto` – The satellite process

- Creates and manages the raw TLS socket
- Works on cell level
  - The encoding/decoding takes place here
- Polls the parent connection process, whenever a cell has been decoded and enqueued successfully
- State:
  - buf (The remaining data that cannot be parsed yet)
  - connection (The PID of the connection process)
  - fifos (The enqueued cells for connection)
  - socket (The actual TLS socket)
  - virginity (Boolean determining the circuit ID length)

`tor_proto` – Why is there a satellite?

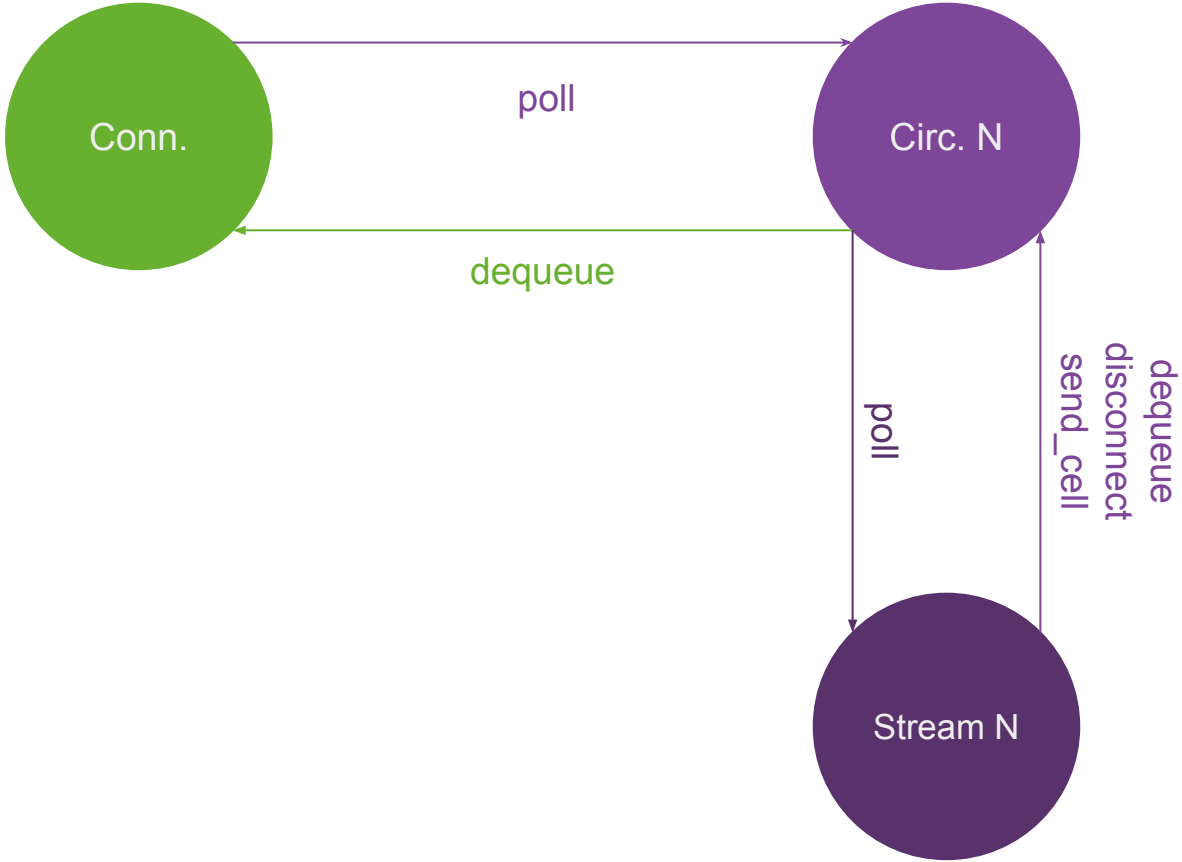
- The idea of a separate process to handle the TLS socket may sound not intuitive
- Why not perform that directly in the connection process?

- We need to receive data, while already processing a request
- We cannot call something like a recv(2), instead we get the data in the process mailbox
- Especially during the initialization, we may not call receive, as it’s discouraged with GenServer

tor_proto – The circuit process

- Manages a circuit of the Tor protocol
- **The only process that ever holds the crypto keys!**
- Spawns the stream processes (connect cell)
- Gets polled by the connection process, every time a cell from the circuit arrives
- Relay cells are decrypted and encrypted here
- Polls the appropriate stream process if such a relay cell has been received
- State:
  - circ_id (The circuit ID assigned by the connection)
  - connection (The PID of the connection)
  - fifos (The enqueued cells for the streams)
  - hops (The hops our circuit is made of [and the keys])
  - streams (Map of all stream IDs with their PIDs)

tor_proto – The actual Tor implementation



`tor_proto` – The stream process

- Manages a stream of the Tor protocol
- During creation, it accepts a closure that handles received data
- Gets polled by the circuit if an appropriate stream relay cell arrived
- No encryption/decryption performed here, all done by the circuit
- Accepts data from external processes using the client API

`tor_proto` – The client API

1. Create connection
2. Create circuit
3. Extend circuit
4. Extend circuit again
5. Create stream with closure
6. Send data
7. Terminate the stream
8. Terminate the circuit
9. Terminate the connection

Upcoming features

- Currently, we have a priority on the directory and path protocols
- Hidden services would also be nice (but probably hard as well)
- Relay support is not planned right now, but I am not against this
- I’m going to university soon though, I do not know about my free time then

Development

- Our Fossil repository is here: https://dev.emux.org/funion
- There is a GitHub mirror: https://github.com/emilengler/funion
- Pull requests and issues are more than welcome. 😍

Greatest Pitfalls

- Circuit IDs need to have an MSB of 1
- Circuit extensions have to be done in `RELAY_EARLY`, not `RELAY`
- Thinking too complicated for the digest verification
- Not realizing Tor uses a symmetric **stream cipher** for onion skins

- You're welcome future Tor implementor

Demo

Thank You!