The design and implementation of a BGP speaker

Rayhaan Jaufeerally
AS210036

https://rayhaan.ch

About me

+ Been at SwiNOG since #34
+ Running a personal AS 210036
  + PoPs in Zurich and Fremont, CA
  + Peering on SwissIX, DE-CIX, France-IX, NL-IX
+ Started writing some tooling around BGP feeds in 2018, which eventually turned into a prototype daemon, and with the founding of AS210036, it became a full project
+ The presentation today is the third attempt, written in the Rust programming language.

Why another implementation?

- Fast evolution of networking infrastructure
  - The landscape looks different
  - More complex, more network elements, centralized control planes
  - No longer scalable or safe to manually use a CLI to configure devices
- Legacy APIs like using Zebra to program the FIB on a host do not cover modern use cases
- Using a DSL in a config file to specify routing policies adds an unnecessary layer of constraint and abstraction

What if the BGP speaker was built from the ground up to fit in well with network automation systems and the modern interconnection landscape?

What are the requirements of a modern BGP speaker?

- **Configuration is centrally managed, applied by automation**
  - Prevent outages by checking invariants, and performing a safe update of the configuration.
  - Be built for API integration / control.
- **Expose state for debugging and monitoring**
  - What the user cannot observe leads to confusion when debugging, make things observable.
- **Be extensible and modular**
  - Protocols change and new functionality is added, it should be easy to add new functionality.

The implementation

Built using the Rust programming language

Chosen after some experiments in C++ and Go.

Maturity of the Rust package ecosystem and tooling makes it a suitable choice for networked systems.

The main selling is memory and concurrency safety.

What does it actually do?

- Can be confusing since a lot of existing “BGP” implementations are also a suite of routing protocols with tight integration between components
  - i.e. can’t programmatically mutate routes between peers for example

The server currently:

- Implements an interface to connect to BGP peers,
- Receives routes from peers, keeps them in a data structure, sends them to the RIB manager
- The RIB manager aggregates routes per prefix, and exposes it as a “path set”
- Exports the RIB via gRPC to clients, both as a dump, and as a live stream

Initial design

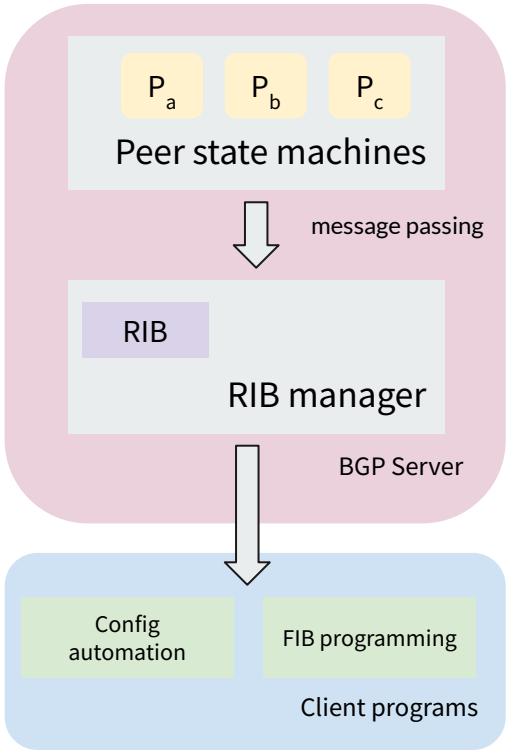
Modular parser

+ Parsing of BGP messages is isolated from the server logic
+ Easy integration of new protocol elements, e.g. new path attributes, BGP messages, address families, etc.
+ Testing and fuzzing of the parser is possible without relying on other implementation details.

gRPC interface

+ Providing a view and stream of the RIB
+ Clients use this to program the FIB
+ Less complicated and brittle logic in the code that handles route updates
+ Client of the RIB API can be restarted without affecting BGP sessions, and vice versa

Key software architectural decisions



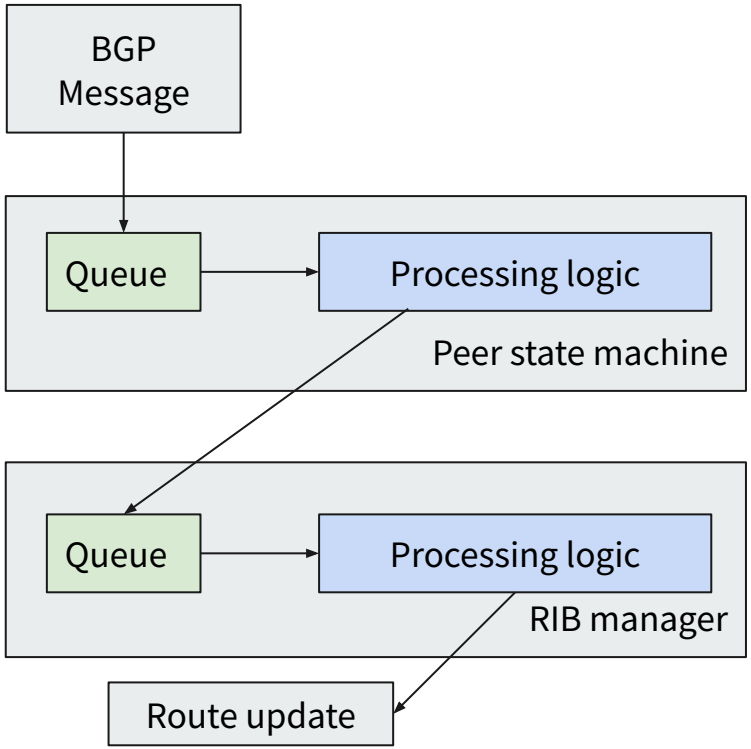
Modular parser, self contained → testable and extendable without modifying the server code

Peer state machine → Model the peer as a state machine that has well defined transitions.

Message passing → Structure of the program is in logical tasks which pass chunks of work around.

RPC interface → streaming full RIB to clients (for e.g. programming into the FIB).

Why message passing



- Allows decoupling code on logical boundaries
- Scaling the program up whilst maintaining ordering of messages where required
- Scheduler takes care of allocating workers to tasks, so CPU resources are assigned to where they're needed
- Data locality during processing
  - Message in the queue can remain in CPU cache whilst the code that's running is changed (scheduler optimization)

Threading models

- **One thread per connection**
  - Conceptually very simple
  - Does not scale well with larger numbers of peers
- **Dispatcher thread**
  - One thread handles events on multiple / all connections and hands off work to worker threads for processing
  - Uses select / epoll for watching many sockets for updates
- **Runtime**
  - Use a library to manage task scheduling in userspace
  - Allows for more than just connection dispatch, but rather dispatch on any event
  - Allows the application to be more modularly split across logical boundaries (e..g parsing packets, sending updates, notifying peers, etc)

Tokio runtime

- Implements a scheduler for tasks in userspace
  - Implements “green threads” like goroutines in Go
  - New tasks can be cheaply spawned on a runtime, and scheduling is done cooperatively by tasks yielding control back to the runtime
- Provides implementations of structures that allow for asynchronous operations
  - TCPStream — to communicate with an endpoint over TCP
  - MPSC channel — Multi producer single consumer channels for message passing
- Synchronization primitives, such as locks, timers, notifications

https://github.com/tokio-rs/tokio

Nom: Parser combinator framework

- A parser combinator library that supports binary and text formats
- Makes handling input and advancing the parser state trivial
  - Consume input rather than keeping a pointer into the buffer
  - Return errors with the current series of bytes being parsed for context
- The parser is structured as a series of functions that recognize small parts of the input and chain together to recognize complex inputs
- Removes repetitive boilerplate code that makes parsers brittle
- Allows building partial parsers for the part of the input that's currently relevant, and omitting unnecessary logic

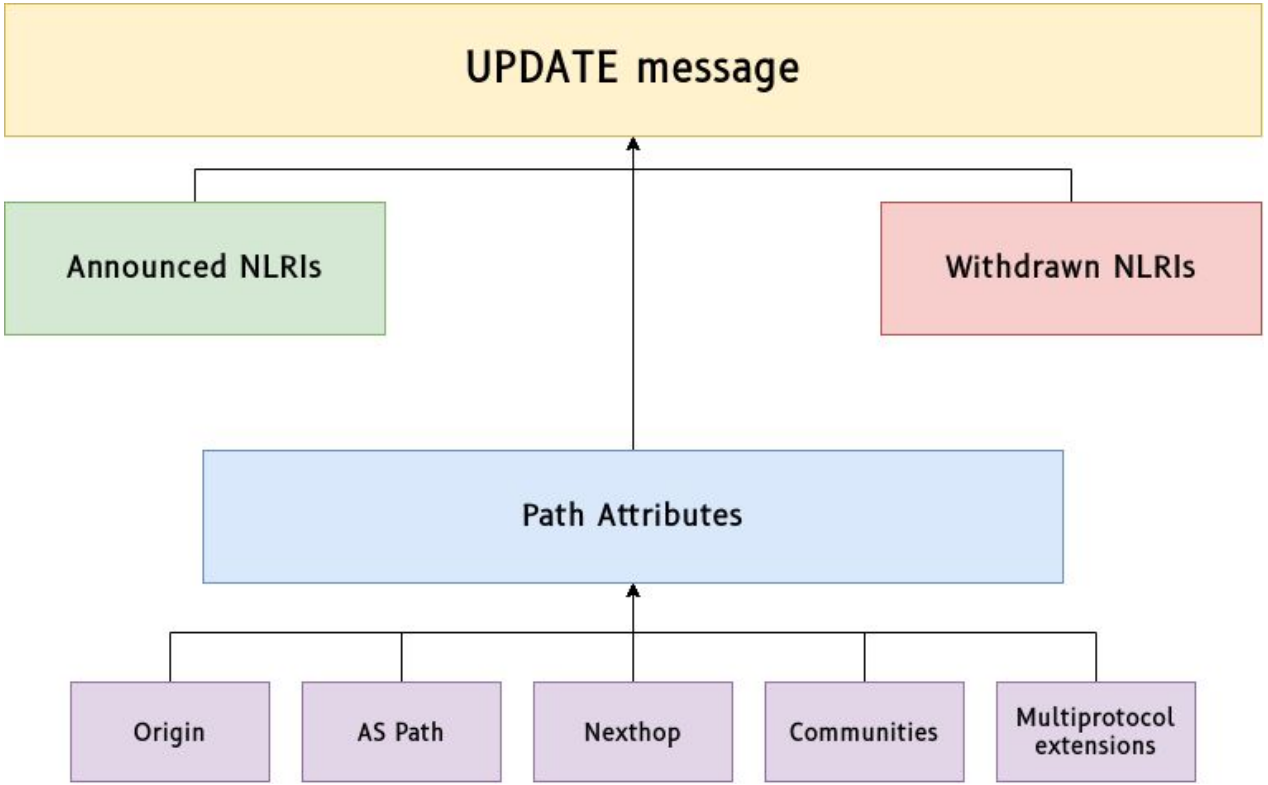
https://github.com/Geal/nom

Example Type-Length-Value parser

```
use nom::number::streaming::u8;
use nom::multi::length_value;

fn parse_tlv(s: &[u8]) -> IResult<(u8, &[u8]), &[u8]> {
    let type = u8(buf)?;
    let value: Vec<u8> = length_value(u8, buf)?;
    Ok((type, value.as_slice()))
}
```

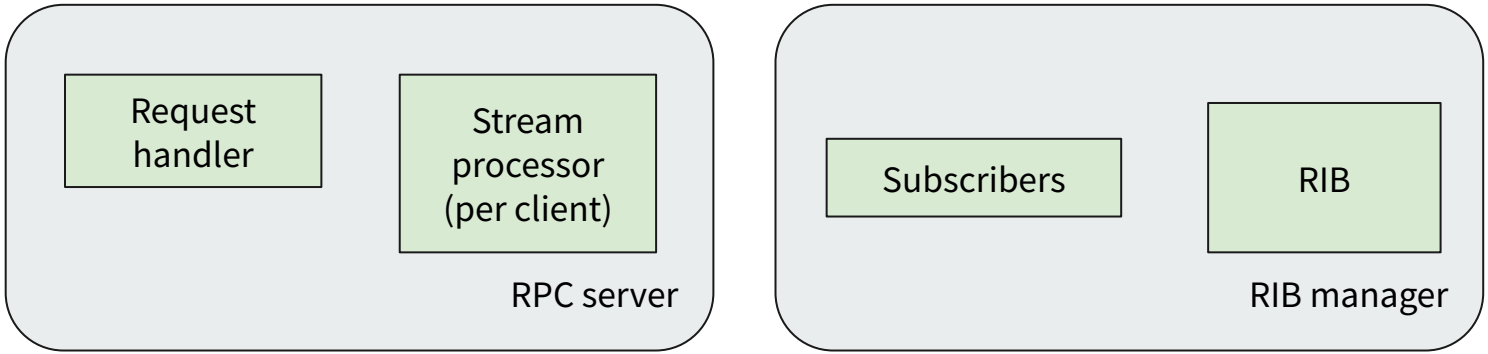
Parser architecture, e.g. BGP UPDATE



gRPC API

In Rust the third-party tonic library https://github.com/hyperium/tonic provides a gRPC client / server

+ Tonic is fully featured RPC server framework including support for streaming RPCs,
+ Streaming the RIB in real time is the key technology that allows the BGP speaker to be split up into multiple smaller components.
+ Methods and messages defined using protobufs.



Future direction

Further design goals

Configuration canarying

Automated testing of new configurations, and rolling back in case of failure.

Forwarding plane heterogeneity

Decoupling of the FIB allows for easier integration with non-kernel based forwarding planes

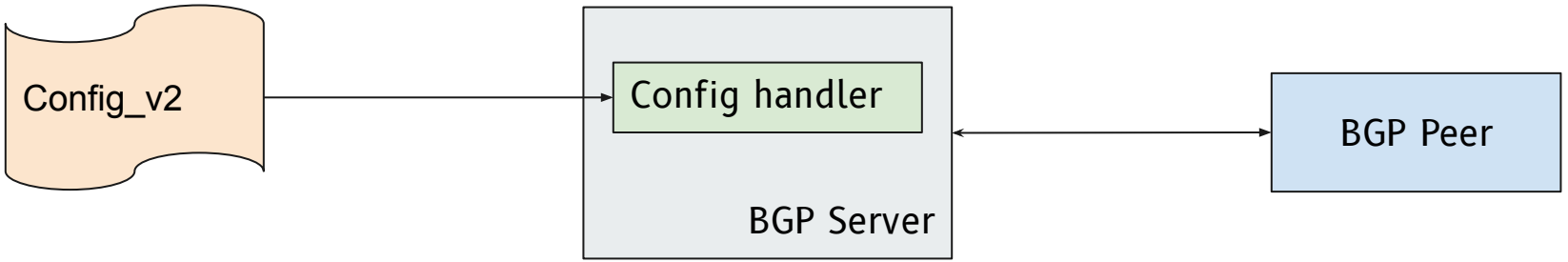
Programmability

Expose fine grained knobs at the API level to allow software to control how the server operates

Automation integration

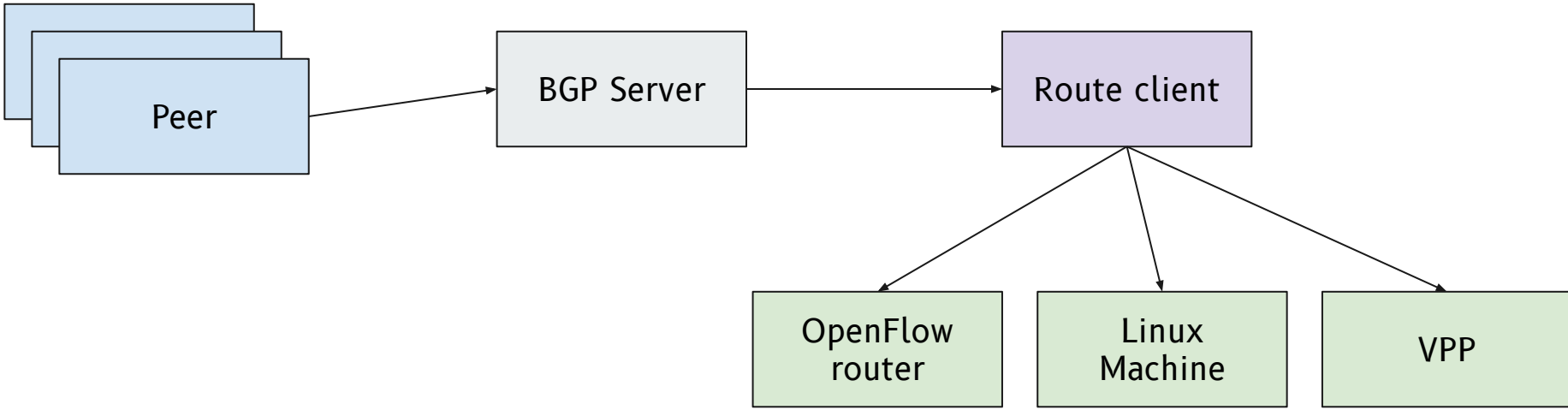
Making configuration state reliably stored, changeable and observable.

Configuration canarying



① New config is generated and pushed to the RPC endpoint on the BGP server to be applied
② Config handler makes a checkpoint and temporarily applies the new configuration
③ Observe certain parameters of the server, e.g. number of routes accepted
    ⓐ If the diff with the new config is too large, e.g. 30% change in accepted routes, reject and rollback
    ⓑ If the diff is below the defined thresholds, finish the update and commit the state

Forwarding plane heterogeneity



Programmability

Peer actions ← Announce / withdraw, update filters, inspect, send notification, get statistics

RIB actions ← Configure forwarding / route reflection,

Further: use a bytecode VM for executing filters on routes, such as eBPF.

Automation integration

Synchronizing configuration to routers is a pain point and involves a lot of glue code (e.g. RANCID) to make sure internal state is consistent with external state.

Many software applications use configuration storage in consistent storage systems like etcd (using Raft consensus)

Routing stacks that use this too can achieve resilience against issues that emerge from doing file and CLI based configurations.

Future

The current state of the code is still quite a proof of concept

Immediate roadmap:

+ Complete the FIB programming support on Linux,
+ Implement filtering logic and additional control plane RPCs
+ Conformance with base BGP spec in RFC4271
+ Additional RFC support, e.g. route refresh, graceful restart,
+ RPKI

Where's the code?

https://github.com/net-control-plane/bgp

Questions?