Routing

Introducing Roto: A Compiled Scripting Language for Rust

Team NLnet Labs
21 May 2025 • 3 min read

Photo by Joseph Barrientos / Unsplash

By Terts Diepraam

We are working on an embedded scripting language for Rust. This language, called Roto, aims to be a simple yet fast and reliable scripting language for Rust applications.

The need for Roto comes from Rotonda, our BGP engine written in Rust. Mature BGP applications usually feature some way to filter incoming route announcements. The complexity of these filters often exceed the capabilities of configuration languages. With Rotonda, we want to allow our users to write more complex filters with ease. So we decided to give them the power of a full scripting language.

We have some hard requirements for this language. First, we need these filters to be fast. Second, Rotonda is critical infrastructure and so runtime crashes are unacceptable. This rules out dynamically typed languages, of which there are plenty in the Rust community. [1] We want a statically typed language which can give us more type safety and speed. Finally, we want a language that is easy to pick up; it should feel like a statically typed version of scripting languages you're used to.

Roto fills this niche for us. In short, it's a statically typed, JIT compiled, hot-reloadable, embedded scripting language. To get good performance, Roto scripts are compiled to machine code at runtime with the cranelift compiler backend.

Below is a small sample of a Roto script. In this script, we define a `filtermap`, which results in either `accept` or `reject`. In this case, we `accept` when the IP address is within the given range.

```
filtermap within_range(range: AddrRange, ip: IpAddr) {
    if range.contains(ip) {
        accept ip
    } else {
        reject
    }
}
```

Instead of a `filtermap`, we could instead write a more conventional `function`, which can simply `return` a value. The `filtermap` is a construct that Roto supports

to make writing filters easier.

The Roto code there might look quite simple, but there's a twist: `AddrRange` is not a built-in type. Instead, it is added to Roto by the host application (e.g. Rotonda), making it available for use in the script.[2] Similarly, the `contains` method on `AddrRange` is provided by the host application as well. The full code necessary to run the script above is listed below. This example is also available on our GitHub repository.

```
use std::net::IpAddr;
use std::path::Path;
use roto::{roto_method, FileTree, Runtime, Val, Verdict};

#[derive(Clone)]
struct AddrRange {
    min: IpAddr,
    max: IpAddr,
}

fn run_script(path: &Path) {
    // Create a runtime
    let mut runtime = Runtime::new();

    // Register the AddrRange type into the runtime with a docstring
    runtime
        .register_clone_type::<AddrRange>("A range of IP addresses")
        .unwrap();

    // Register the contains method on AddrRange
    #[roto_method(runtime, AddrRange)]
    fn contains(range: &AddrRange, addr: &IpAddr) -> bool {
        range.min <= addr && addr <= range.max
    }

    // Compile the program
    let program =
        FileTree::read(path).compile(runtime).unwrap();

    // Extract the Roto filtermap, which is accessed as a function
    let function = program
        .get_function::<(), (Val<AddrRange>, IpAddr), Verdict<IpAddr, ()>
          "within_range"
```

```
        )
        .unwrap();

    // Run the filtermap
    let range = AddrRange {
        min: "10.10.10.10".parse().unwrap(),
        max: "10.10.10.12".parse().unwrap(),
    };

    let in_range = "10.10.10.11".parse().unwrap();
    println!("{:?}", function.call(&mut (), range, in_range));

    let out_of_range = "10.10.11.10".parse().unwrap();
    println!("{:?}", function.call(&mut (), range, out_of_range));
}
```

Note that nothing in the script is run automatically when the script is loaded, as happens in many other scripting language. The host application decides which functions and filtermaps it extracts from the script and when to run them.

Roto is very tightly integrated with Rust. Many Rust types[3], methods and functions can be registered directly for use in Roto. These types can be passed to Roto at negligible cost; there is no serialization between Roto and Rust. For Rotonda, this means that Roto can operate on raw BGP messages without costly conversion procedures.

The registration mechanism also ensures that Roto is not limited to Rotonda and could easily be used outside that context. It is designed as a general scripting or plug-in language.

We have many planned features on the roadmap for Roto and will continue to improve this language. This also means that the language should not be considered stable, though we'd love to hear feedback if you experiment with it. If you're interested, check out the documentation, repository and examples.

1. E.g. Rhai, Rune, Mlua, Deno, PyO3, Dyon & Koto. ↩
2. Note that the registering of types is not hindered by Rust's orphan rule, because it doesn't require any specific traits apart from `Clone`. This makes it possible to expose types from external libraries to Roto. ↩
3. Specifically, types implementing `Clone` or `Copy`. Types that don't implement these traits can be wrapped in an `Rc` or `Arc` to be passed to Roto. ↩

Overhauling Domain

By Arya K. Previously, we discussed the massive development our domain library underwent over…

21 May 2025 5 min read

Prometheus Metrics in NSD 4.12.0

By Jannik Peters We finally implemented a Prometheus metrics endpoint, providing the…

24 Apr 2025 1 min read