The Tor Network Status API

About Me

Name: Mattia Righetti
University: Politecnico di Milano, IT | Master of Science in Computer Science (Expected graduation: Sept 2023)
Contacts: mattiarighetti@outlook.com
Website: https://mattrighetti.com
GitHub: https://github.com/mattrighetti
Timezone: GMT+1 Rome
Resume: https://mattrighetti.com/resume.pdf

Hey there! My name is Mattia and I am a software engineer with a passion for backend engineering and distributed systems. I am about to complete my thesis for my MSc in computer science at Politecnico di Milano, Italy.

Throughout my career, I have primarily worked as a backend engineer and have gained extensive experience in this area. I am particularly interested in exploring the world of distributed systems and am constantly looking for ways to challenge myself in this field.

In addition to my professional work, I am also a great user of Tor browser, which is why I want to collaborate on this open source project in the first place. I believe in the power of collaborative work and enjoy contributing to open source projects whenever I can, especially to get to know more about the organization and its efforts to give privacy to users.

Finally, I am excited to learn more about Rust and its backend ecosystem. I believe that Rust has a lot of potential in the world of backend engineering, and I am eager to explore this area further.

Thank you for considering me, and I look forward to the opportunity to contribute my skills and experience to the Tor project.

Code Contribution

- Fixes #40034 (Merged): Very first contribution to the sub-org project **onionoo**. The PR mainly contains code refactoring. Commits mostly change dead/unused code that was marked for deletion a long time ago but was still present in the codebase.

Project Information

Sub-Org: Tor Network Health

Project Abstract

In an effort to improve current resource utilisation and optimisation, the Tor team is developing a new version of their pipeline (v2.0). This update involves transferring much of the data related to Tor nodes and bridges from files stored on a single server's disk to two separate databases: Postgres and Victoria Metrics.

The main objective of this project is to design a RESTful API service using the `actix_web` framework that is going to be integrated in the new pipeline v2.0 to support data retrival from the two databases. In particular, the focus will be on designing the new APIs, its requests, and response formats. The project will also involve defining appropriate endpoints and data models, ensuring scalability, performance, and security. The final goal is to achieve a web service that is going to extend/replace the current onionoo protocol used by stakeholders interested in the status of the Tor network and its individual nodes.

Detailed Description

Currently, the Tor network status informations are retrived from a service called onionoo which is a web protocol that provides data about running Tor relays and bridges to other applications and websites.

Onionoo as of March 2023 does three things:

1. Reads the current tor network descriptors
2. Writes statuses in txt files to disk
3. Produce output for each relay and bridge from all those status files

As stated in the pipeline's wiki, these three tasks create two different kind of problems:

- Descriptors are processed multiple times in multiple places
- Tasks are heavy on disk and network I/O ops, which slows down the service and metrics processing

To solve the problems listed above, the Tor team is currently working on a pipeline 2.0 which makes use of two databases to store and archive all of this data, in particular, a Postgres database and an instance of Victoria Metrics for timeseries data. By querying the database, most of the disk I/O operations that the service had to run are gone and responsiveness should be improved, potentially by a lot. Also, server load should drop significanlty because there are no more processess running those I/O operations described above. The Victoria Metrics instance, on the other hand, is going to enable historical data querying which is a feature that the Tor Network teams is looking forward to and that is not offered by onionoo.

With the introduction of this new pipeline 2.0 and the adoption of Postgres and Victoria Metrics, it comes the need of a web service that does the heavy lifting task of serving results data to interested stakeholders. Since the service is going to slowly take over/extend the onionoo protocol, `actix_web` is the framework proposed to develop the RESTful APIs, which is a well estabilished web framework written in Rust with a huge community behind it and it promises peak performance and reliability.

Backward compatibility is a valuable asset in this particular case especially if we want the service to either extend or replace onionoo. For the service to be ready-to-use and ready-to-test I propose to keep the same endpoints that onionoo offers at the moment:

- `GET https://onionoo.torproject.org/summary`: returns a summary document
- `GET https://onionoo.torproject.org/details`: returns a details document
- `GET https://onionoo.torproject.org/bandwidth`: returns a bandwidth document
- `GET https://onionoo.torproject.org/weights`: returns a weights document
- `GET https://onionoo.torproject.org/clients`: returns clients document
- `GET https://onionoo.torproject.org/uptime`: returns an uptime document

Same goes for the query parameters:

- `type`
- `running`
- `search`
- `lookup`
- `country`
- `as`
- `as_name`
- `flag`
- `first_seen_days`
- `last_seen_days`
- `first_seen_since`
- `last_seen_since`
- `contact`
- `family`
- `version`
- `os`
- `host_name`
- `recommended_version`

- `fields`
- `order`
- `offset`
- `limit`

Response structure should also be kept the same in order to not break current clients, even though the onionoo protocol page explicitly warns

> *Clients should be able to handle all valid JSON responses, ignoring unknown fields and not breaking horribly in case of missing fields.*

All of this should be the basis and a good starting point for the project.

Keeping the same structure of the onionoo service makes integration easier with current solutions adopted in the Tor org and it could be useful in the foreseeable future when we are going to slowly transition network traffic and load from onionoo to the new Tor Network Status APIs.

Moving on to the new historical data feature enabled by Vicotria Metrics, I propose two different solutions that we may want to explore further:

1. Introduce a new endpoint by prepending `/history` to the current endpoints + range filters
   1. `/history/summary`
   2. `/history/clients`
2. Work with new range filters only

The first proposed solution will result in a cleaner codebase because the history endpoint (i.e. `/history/summary`) is going to have a completely separate logic from the default endpoint (i.e. `/summary`). This is useful since the historical endpoint needs to contact Victoria Metrics while the non-historical endpoint is going to query Postgres. Also, the historical endpoint logic will need to use a web-client to query Victoria Metrics + `PromQL` (Victoria Metrics HTTP API, Victoria Metrics Query data), while the default endpoint will just make use of a default connection to Postgres + plain `SQL` queries.

If we want to keep the same endpoints both for historical and non-historical data and just work with range filters as URL query parameters we will need to encapsulate both logics under a single endpoint which could be harder to read, test and maintain.

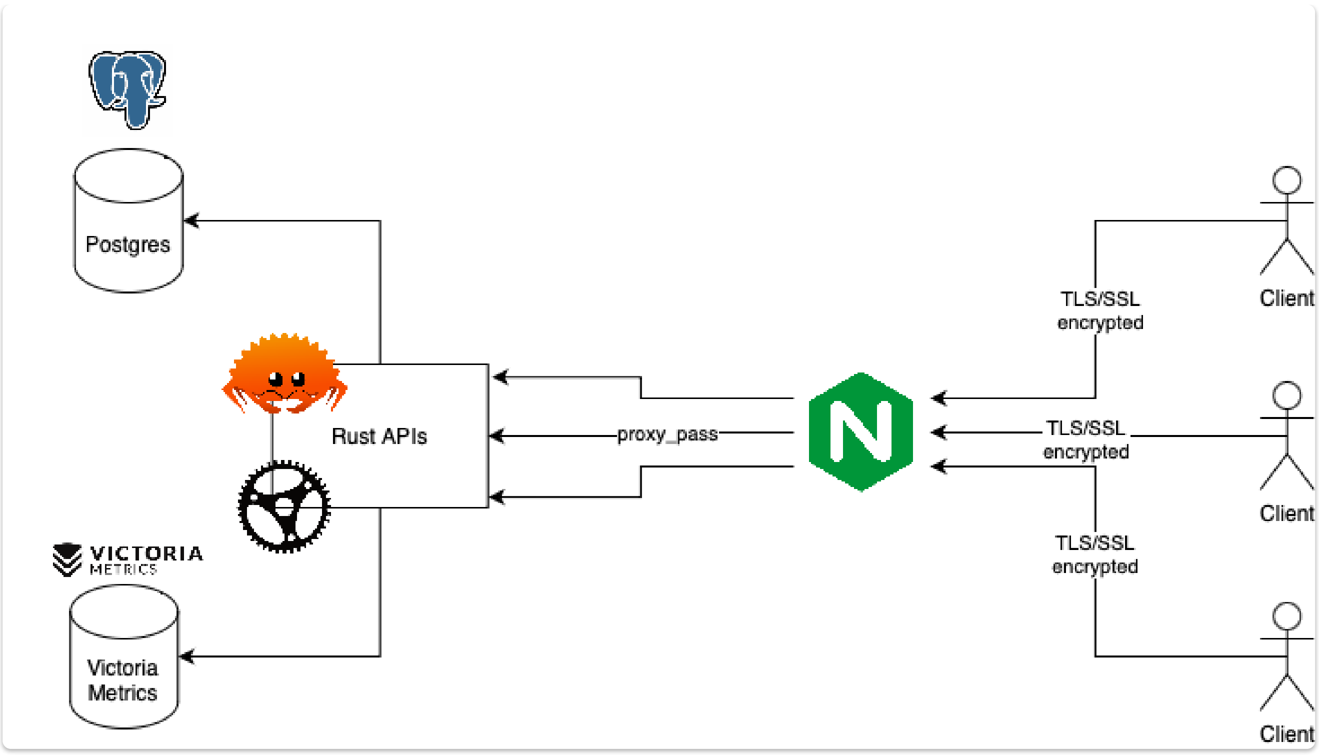
Either way, both solutions will provide date range filters to select dates of interest. I propose to use URL query parameters that take an initial date `start_date` of interest and an end date `end_date`, both of them should take a date in ISO 8601, `YYYY-mm-dd` could be enough for this use case but that is up to discussion. An example of historical query filters would be `start_date=2023-01-10&end_date=2023-01-15`.

ad-hoc Caching solutions (i.e. Redis) shouldn't be necessary at the very beginning and should be implemented when there's real need for it because it's going to introduce significant complexity in the design phase. It's also important to point out that before ad-hoc caching solutions we should leverage Postgres caching and query optimizations which in most cases make the former not necessary. Indeed, a proper Postgres setup is going to cache indexes, query plans and table data in RAM, so it should be pretty fast. Exploring current performances of queries could hint if there's real room for improvements in caching, i.e. looking at the ratio of data coming from disk I/O or Postgres shared_buffer.

Finally, I propose to place an NGINX instance as a reverse proxy in front of the service for multiple reasons:

1. Improved security: By putting NGINX in front of our service, we can offload the task of handling incoming requests and enforcing security policies to a dedicated proxy server. This reduces the exploitable surface, making it less vulnerable to common attacks.
2. Load balancing and scalability: NGINX can act as a load balancer, distributing incoming requests across multiple instances of our service. This improves the scalability and availability of the service by handling more requests and reducing the impact of failures or downtime.
3. Caching and performance optimization: NGINX can cache frequently requested API responses. It can also compress responses and perform other optimizations to improve the performance of our APIs.
4. Simplified API management: central point of control for managing APIs, including authentication and authorization, rate limiting, and access control. This simplifies the management and configuration of APIs, making it easier to enforce policies and monitor usage.
5. SSL/TLS encryption: useful to encrypt traffic between clients and the proxy server.

This is a high level picture of the final architecture of the service



Development Details

In this section I would like to elaborate more on the proposed Rust crates to use in the project and explain why they are a good fit:

- `actix_web` is the proposed framework for this project, both for performances and reliability. To maximise system resources usage, everything will be programmed to run and serve requests concurrently.
- `sqlx` to interface with Postgres, it provides concurrent connections to Pg and is used and maintained by a large community. `diesel` is another famous and well adopted crate, but it's harder to setup and could present a significant obstacle for newcomers that would like to contribute to the project in the future.
- `reqwest` is going to be the HTTP client to query Victoria Metrics APIs, but I could also see it used in API tests to impersonate a real user querying the service. If we need a more fine-grained control of requests we can opt for `hyper`.
- `serde` for data serialization and de-serialization, it's the de-facto standard in Rust and it is also deeply integrated with `actix_web`.
- `config` is the proposed crate to setup project configuration files.

Development workflow steps include:

- APIs implementation
- APIs testing
- APIs documentation

API testing will be automated with GitLab CI/CD pipelines. Each endpoint is going to be tested with an HTTP client and each test will be run with expected and unexpected data/filters to check that everything behaves as expected.

Timeline

Week 1:

1. Community bonding

Week 2:

1. Current APIs/protocol review
2. Design new API for historical data (new filters, endpoints)
3. Discuss possible improvements in re-desining some of the existing APIs/protocol
4. Layout initial project structure (actix_web)
5. GitLab CI/CD setup

Week 3

1. Initial API implementation
    1. `/summary`
    2. `/details`
2. API Documentation
3. API Testing

Week 4

1. API implementation
    1. `/bandwidth`
    2. `/weights`
2. API Documentation
3. API Testing

Week 5

1. API implementation
    1. `/clients`
    2. `/uptime`
2. API Documentation
3. API Testing

Week 6

1. Code refactoring
2. Performance tests and evaluation

3. (Beta deployment and test)

Week 7

1. Historic data API implementation
2. API Documentation
3. API Testing

Week 8

1. Historic data API implementation
2. API Documentation
3. API Testing

Week 9

1. Historic data API implementation
2. API Documentation
3. API Testing
4. (dev deployment and test)

Week 10

1. Code refactoring
2. Performance tests and evaluation

Week 11

1. Code refactoring
2. Performance tests and evaluation
3. NGINX test deployment

Week 12

1. NGINX test deployment
2. Performance tests and evaluation

Other Commitments

- Dad 60 year birthday vacation (6-12 July)