Автономная некоммерческая организация высшего образования
«Университет Иннополис»

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА
(БАКАЛАВРСКАЯ РАБОТА)
по направлению подготовки
09.03.01 - «Информатика и вычислительная техника»

GRADUATION THESIS
(BACHELOR'S GRADUATION THESIS)
Field of Study
09.03.01 – «Computer Science»

Направленность (профиль) образовательной программы
«Информатика и вычислительная техника»
Area of Specialization / Academic Program Title:
«Computer Science»

Тема / Topic	**Производительный фреймворк для обратных прокси и сетевых шлюзов / Performant reverse proxy/gateway framework**

Работу выполнил / Thesis is executed by	**Макаров Вадим Дмитриевич / Vadim Makarov**	подпись / signature
Руководитель выпускной квалификационной работы / Supervisor of Graduation Thesis	**Кудасов Николай Дмитриевич / Nikolay Kudasov**	подпись / signature

Иннополис, Innopolis, 2024

Contents

List of Tables

List of Figures

Abstract

Lots of IT businesses make use of reverse proxy and gateway applications to manage ingress traffic. However, most existing solutions have to compromise on either flexibility or performance. Certain cases may require more flexibility than what the majority of established proxy applications are capable of without compromising on the performance.

This research and development aims to achieve both the performance and benefits of a dynamic gateway. The goal is to offer full control of routing and other specifics to the developer at the runtime of their proxy application.

The main goal is development of a reverse proxy framework in Rust, one of the most performance focused high-level programming languages as of the time of writing. The framework handles the proxy logic with implementations provided out of the box and provides the general structure of the application. Developers can then configure the proxy dynamically during the runtime of their application.

Applications developed with the framework show performance comparable to well-established performant solutions such as nginx with similar configuration, all while introducing zero downtime.

The results of this research and development are satisfying and show that theres still room for growth in the field of reverse proxy and gateway applications. However, the developed solution is still far from being complete and production-ready. As such, future work will focus on making it reach feature-completeness and reliability.

Chapter 1

Introduction

Nowadays, many web services make use of reverse proxy and gateway solutions to manage incoming traffic. Such solutions help distribute the load, route the traffic to a specific service, improve security for the clients by acting as an SSL termination point among many other use cases. Although there already exist many reverse proxy applications, they all have their features and limitations. For example: some cannot be configured dynamically (Nginx, Apache HTTP Server, Caddy), while others are intended to work only with a very specific technological stack (Fabio, Traefik).

Often the existing solutions are perfectly fine to use and are covering the required features. However, this is not always the case and there are times when a custom solution is going to be simply easier to maintain. Examples include using something other than Kubernetes for container orchestration. Since Kubernetes is the de-facto standard for orchestration nowadays, a lot of proxy solutions may rely on Kubernetes API or other things that in turn rely on it. Other situations where using a custom solution may seem beneficial are cases where there's a lot of functionality at the gateway level, such as authorization, SSL termination

with dynamic certificates, routing based on specific HTTP headers or a complex combination of routing rules and etc.

This project is about development of a reverse proxy framework, as opposed to creating another end application. The goal is not to come up with another end application, but rather help drive the development of lots of such end applications, each tailored to specific needs. The framework's intention is to help bootstrap the development of such custom solutions and make it easier, more structured and reliable. It is, however, out of the scope of this project to implement lower-level networking details such as TCP socket polling, HTTP implementation, implementation of load balancing algorithms and so on. This project mainly focuses on providing the architecture and utilities that may prove useful when developing a reverse proxy application. That means I'm going to make use of already existing Rust projects for concurrency, networking and other necessities whenever it is reasonable to do so.

For the goals of this project, I chose to work with the Rust programming language, as it is a fairly high-level programming language focused on performance that has been gaining more and more attention lately. Rust is a compiled language that does not have a garbage collector and does not a require a virtual machine for its runtime, which already lets it achieve better performance compared to languages such as Python, Java or Go. Rust also already has safe to rely upon networking solutions, which is also very convenient.

With a framework, as opposed to a standard proxy application, it is trivial to come up with any gateway-level logic you might require as it becomes as straightforward as just writing code, which for many developers is something they do on a daily basis. Even for simple use-cases it might be more beneficial to make your own custom proxy application with a framework in the long run,

as it will allow for more flexibility in the future and generally provides a more maintainable way of configuration when compared to something like a nginx configuration templating solution.

Chapter 2

Literature review

This chapter aims to investigate what approaches could be taken when developing a performant reverse proxy framework in the Rust programming language. The main focus will be put on the existing asynchronous runtimes developed and maintained by the community and what using such a solution would entail. The chapter has the following structure:

- Section 2.1 reviews known asynchronous runtimes.
- Section 2.2 reviews existing similar solutions.
- Section 2.3 delves into the considerations one should take when choosing a runtime.
- Section 2.4 outlines the polling techniques and interfaces that asynchronous runtimes make use of in their reactors.
- Section 2.5 draws a conclusion to this chapter.

2.1 Rust asynchronous runtimes

An asynchronous runtime (later referred to as async runtime) for Rust is a library used for executing asynchronous applications [1]. In this section, I introduce some of the more known asynchronous runtimes for the Rust language that focus on networking operations.

Tokio is the most widely used async runtime in Rust as of right now, offering the biggest ecosystem. Tokio takes upon itself the thread management for the most part and makes use of epoll (when running on Linux) to achieve the asynchronous part of networking. Tokio also has work stealing for better load distribution across threads under its management [2].

Monoio, a recently introduced promising async runtime, makes use of the thread-per-core model with no work stealing, while also providing the ability to use the recent io_uring feature of the Linux Kernel [3] that can offer significant benefits to the networking applications, compared to other approaches such as epoll.

Glommio is a runtime that shares similarities with the previously introduced Monoio which was released before the former. It also uses a thread-per-core model and utilizes io_uring in its reactor [4]. Although boasting the same key differences from the conventional Tokio runtime as Monoio, according to the performance comparison posted by the Monoio team it still falls behind Monoio in terms of performance [5]. On top of that, Glommio does not seem to get active development in recent days, making it the least appealing choice of the three.

2.2 Related work

During my work on this project, Cloudflare has publicly released and open-sourced a similar in its goals framework for proxy development called Pingora [6]. It is also written in Rust and it makes use of the Tokio asynchronous runtime. It has a lot of differences from this project, however the most notable one is probably their decision to not rely on any existing HTTP solutions such as Hyper. They implemented all of the protocol lower-level details themselves, as well as load balancing algorithms and even hashing, which is what primarily sets apart Pingora from this project. Needless to say, it shouldn't be expected that I can compete with Pingora in feature-completeness or production readiness, as Pingora is far more mature and has been already used by Cloudflare before its public release [7].

2.3 Considerations

To choose an async runtime, it is important to understand what features are key for the specific application, in this case, the application being proxying network traffic.

In order to make the decision, I decided to look for articles and blog posts talking about technical decisions for other proxy applications, which is when I came across a blog post by Cloudflare about their Pingora proxy, which has also been developed in Rust. They state they have chosen Tokio as the async runtime for their proxy, as it fits their needs well. Pingora development team deemed work stealing [8] a necessity for their proxy with the goal of avoiding certain performance issues in their specific workloads [7], such as CPU-intensive work.

Although the Pingora team intentionally chose an async runtime that im-

plements work stealing, that does not mean all proxy applications would require the same approach. On the contrary, Nginx, which has been the staple reverse proxy application for many years, has a completely different threading model. The Nginx team states, that work stealing (or context switching, as they call it) can become a performance degradation factor under high loads [9].

2.4 Polling techniques

To better understand the advantages of the async runtimes available, it is important to understand how their reactors work. For non-blocking networking in Linux, the default approach is polling using epoll, which is what the Tokio reactor uses when running on Linux. Monoio, however, uses io_uring, which is a new Linux I/O interface that provides low latency and feature-rich asynchronous I/O operations [10]. Research shows that io_uring performs 10% better when the system call batch is large [11], which tends to be the case in proxy applications, although it can perform worse than epoll in other cases.

2.5 Conclusion

To briefly sum up what has been discussed above, there is a vast number of approaches and decisions that can be taken when developing a reverse proxy application. However, for a generic reverse proxy, it would seem fit to make use of the thread-per-core threading model, rather than a different one implementing work stealing. As such, Monoio is the best option out of the three discussed. Not only does it take the thread-per-core approach to thread management, it can also optionally use the io_uring interface, which could also potentially be beneficial

in the context of a proxy application.

Chapter 3

Design and methodology

In this chapter, we will be going through used technologies and decisions that led to said technologies being chosen.

3.1 Approach based on Tokio

In the early stages of this project, we conducted a small comparison for the sake of research on which asynchronous runtime would suit the needs of a proxy application best. In the said comparison we were mainly considering three options: Glommio, Monoio, and Tokio. We have already talked about those runtimes and their specifics in greater detail in the Literature review chapter. even before the actual performance comparison, it became clear that Glommio was not the optimal choice for the project, primarily because there were not any distinguishing features setting it apart from Monoio, all while it performed worse than Monoio according to the provided benchmarks. That left us with only Monoio and Tokio for consideration. To test and compare the two we built two proof-of-concept proxy applications that were similarly configured and measured

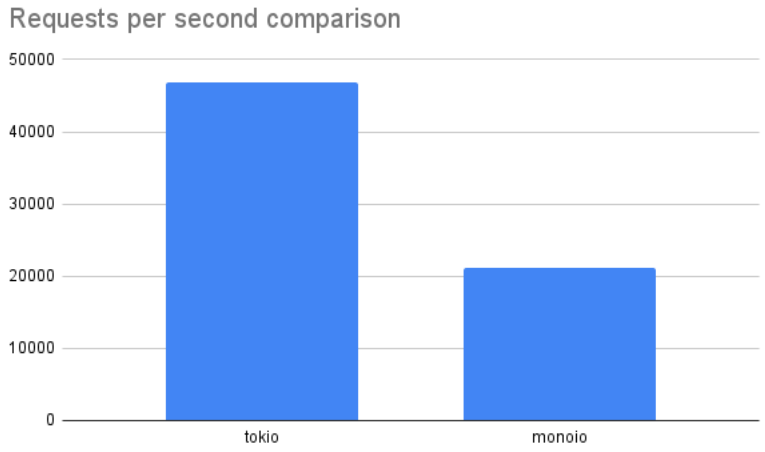

Fig. 3.1. Throughput comparison between Monoio and Tokio

their throughput and latency.

As can be seen in Figure 1 out of the two applications we built Tokio came in first with a considerable lead. At this point, a fairly large amount of time was already spent trying to improve the performance of the Monoio-based application, which led us to believe that Tokio must be the better choice out of the two for the goals of this project. Of course, we do not deny the possibility of the sample application being faulty, which could have led to such poor performance, but this is out of the scope of this project. On top of Tokio performing better in the held comparison, it is also far more established and has a lot more community-backed utilities and tools available for usage, which also played a part in the end choice.

For all of the reasons described above, I have decided to use Tokio as the asynchronous runtime for the framework.

3.2 Approach based on Tower

Since the idea of the project is to create a framework, it is a given that it should be made as extensible as possible. But the framework being extensible is

one thing, the other issue that should be considered is how hard is it for the end user of the framework to implement basic functionality. It is not reasonable to expect the user to implement all of the utilities such as rate limiting by themselves if all they are trying to do is make a trivial proxy application. That is where Tower [12] comes in. To achieve extensibility backed by already existing utilities, I have decided to make heavy use of existing tower utilities and abstractions it relies on.

Many modern server frameworks for Rust (Axum [13] for HTTP and Tonic [14] for gRPC to provide a few examples) make use of Tower, so it is not unreasonable to expect Tower concepts to already be well known to a developer familiar with server applications development in Rust. Even in the case, they are not, Tower provides excellent documentation and it is well-adopted by the community making looking for practical examples close to effortless.

To name a few utilities Tower provides with its crate features that could prove to be useful in proxy applications: load balancing, service metrics, rate limiting, concurrency limiting, timeouts, retries and reconnects.

Those are not all the tower crate itself provides, moreover, other crates provide useful functionality.

Taking all of the above in mind, it was only reasonable to rely on Tower in our project. Specifically, users primarily configure the framework's IO loop with Tower services. Tower services are used for the actual proxy functionality and routing, which the framework provides default implementations for. These being Tower services makes it trivial and straightforward to extend their behavior for things such as rate limiting implementation. Internally, the IO loop also utilizes Tower's "load" feature flag to implement Power of 2 random choices [15] load balancing between services of the same kind.

3.3 Approach based on Hyper

Reverse proxies often work with HTTP or HTTPS protocols, as such, it is a must for any reverse proxy to offer straightforward support for them. In the case of this project, that means the framework has to provide an HTTP(s) listener (request emitter) by default. I have decided to use the Hyper HTTP server to accept the incoming HTTP traffic for several reasons, that I would like to go through in this section.

The HTTP protocol is based on TCP, so one could naturally think that if you implement a TCP listener - most of the groundwork for an HTTP listener is already laid. This is partially true, rust already has available crates to parse HTTP such as `httparse` [16]. However, the ways different versions of HTTP operate vary significantly, to the point where it would require a completely different approach and implementation for an HTTP version 2 listener when compared to an HTTP version 1.x listener. Hyper's server includes support for both HTTP ver. 1.x and HTTP ver. 2 [17] already, which tremendously simplified the listener implementation.

However, using hyper not only simplifies the listener implementation, it also advocates for its reliability. Hyper is a widely adopted relatively low-level HTTP library, used by lots of other projects, that implement servers based on the HTTP protocol e.g. axum, and tonic. Hyper is a reliable HTTP library with a proven track record and also is actively maintained. While in some edge cases, it might make sense to implement HTTP yourself, I did not deem it necessary for this project. Rather than trying to reinvent the wheel for no particular reason, I chose to rely on hyper for the HTTP implementation.

To add to everything mentioned above, hyper's server also integrates with

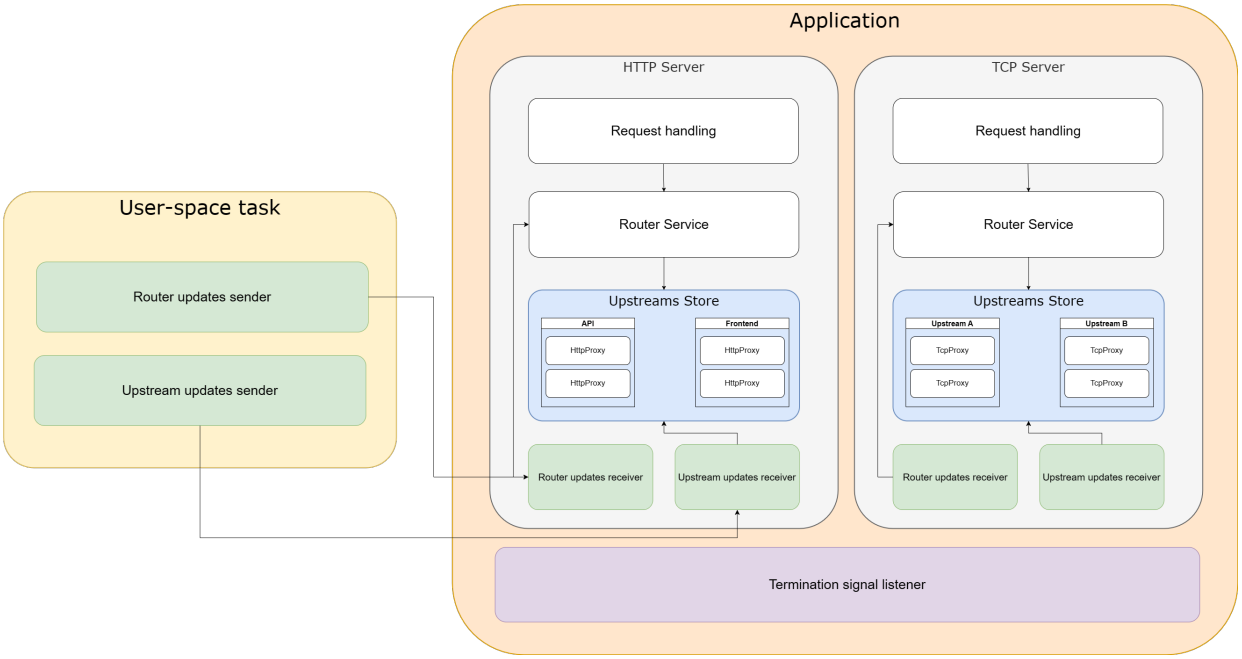

Fig. 3.2. Diagram representing architecture of the framework

the tower's Service trait very nicely and also provides HTTP client functionality, which is useful to forward the incoming request to the respective upstream.

3.4 Framework API and internal design

Overview of the architecture powering the framework and abstraction's relations to each other are presnted in figure 3.2. Later in this section each part is described in greater detail.

The overall interface provided by the framework revolves around the Server trait. A Server defines the IO loop of the public-facing endpoint, maintains upstream store and load balancing for them, and just generally acts as a server.

Server decides which upstream should be handling the incoming request using a Router service, that can be changed during the runtime through a `tokio::sync::watch` channel. A router implements the tower's Service trait, for

which the Request type parameter is the lessened information about the incoming request: for TCP it is just the peer address, and for HTTP it is the request's header map. The response of a router is just the upstream's unique identifier.

A Server operates dynamically provided by the user Proxy services (meaning they implement the tower's Service trait) through a configuration channel. First, it puts a received Service in the upstream store under the load balancer for the specified upstream id. A Proxy service as part of its call, takes the whole incoming request, handles forwarding it to the end destination, and returns the response if that is relevant (e.g. for TCP the response is not at all relevant and will be discarded by the IO loop). During the IO loop, when the Router has determined what upstream the incoming request relates to, the Server's implementation calls the upstream's load balancer, which in turn calls one of the Proxy services it operates on.

One notable constraint on the Proxy service is that it has to implement Load, which is a trait used for metrics collection. Satisfying this trait is fairly trivial when using services and layers provided by the tower's load feature. In case none of the available mechanisms are fitting, users may implement Load themselves if they choose to do so.

The framework provides implementations for HTTP and TCP Servers and Proxies, which then users can choose to either extend upon using Tower or disregard completely in favor of a self-written implementation.

Servers themselves can be combined using an Application. It serves the purpose of providing a common graceful shutdown handler for all of its underlying Servers and ease of combining them to await on a single handler.

For the user's convenience, the framework's default implementations also provide the ability to construct and configure them using the Builder pattern.

Application and all Servers and Proxies allow for their configuration in this manner.

Chapter 4

Implementation and results

This chapter will delve into the details of the implementation of different parts of the project: such as Application, Server, and Proxy abstractions that the framework relies on, as well as show the applicaion examples and comparison to other proxy solutions.

4.1 Application

The application does not really implement much of anything, but there still are interesting things to talk about. Using a builder-like pattern, the application can receive an arbitrary number of objects implementing the Server. Must be mentioned that all of these objects are NOT required to be of the same type. This essentially means that one Application can handle TCP, HTTP, or any other protocol for that matter all in one instance. It requires boxing those objects, but that performance cost is negligible considering the convenience it provides and I would argue is unavoidable. When the application is run, it awaits on all those servers' serve futures which should implement the serving logic of their respective

protocol. An example of that is shown in figure 4.1.

Because there can be a number of those Servers, I use `FuturesUnordered` from the `futures` crate for polling those futures, since that will make sure all of the futures are polled after being pushed in as well as provide guarantees regarding their required lifetimes. It also will not cancel the other futures after just one completes (in contrast to the simple `tokio::select!` macro) and also supports an arbitrary number of futures, so it made perfect sense to use this here.

One of the reasons to use Application for the framework user would be out-of-box support for graceful shutdown. Graceful shutdown is implemented using the `tokio-util` crate's `CancellationToken` [18]. It can be cloned very cheaply and can split into child tokens as well, which makes it more convenient to use than just pure `tokio`'s `sync::oneshot` channel. The idea is that in the cancellable parts, we await not just the routine future itself, we also await on the token's `.cancelled()` future using `tokio::select!` macro. We just have to make sure that the other future is safely cancellable (such as the proxy server routine).

The Server trait requires implementing `fn cancel_token(&self)` which returns `CancellationToken` so that the Application can access it to cancel externally from the `Server`'s `.serve(&mut self)` implementation. With that all the `Application` has to do is spawn a task that will poll SIGINT and SIGTERM signals using `tokio::signal::unix` and call `.cancel()` on all of the Servers' tokens.

4.2 Server

The `Server` trait itself is fairly trivial and can be seen in figure 4.2

There is not much to reason about in and of itself; rather, it is more interesting

```
Application::new()
        .server(srv1)
        .server(srv2)
        .serve_all()
        .await;
```

Fig. 4.1. Example using Application for multiple Servers

```
pub trait Server {
    type Error;

    async fn serve(&mut self) -> Result<(), Self::Error>;

    fn cancel_token(&self) -> CancellationToken;
}
```

Fig. 4.2. Definition of the Server trait

to talk about implementations provided by the framework for TCP and HTTPv1 (later referred to as simply H1) protocols.

Both TCP and H1 need to maintain a few common things: a `Router` service, an upstream store containing load-balanced Proxies, a `ConnectionTracker` from the `tokio-util` crate, as well as two tasks run concurrently that poll router and upstream change channels for dynamic configuration updates.

For the router service polling and upstream changes polling, I put those two tasks separately from the end implementations into a `util` module, as well as the full type definition for the upstream store, since those are very similar in both implementations.

The router store is basically a concurrency-friendly `HashMap` implementation, provided by the crate `scalable-concurrent-containers` [19]. It uses the `UpstreamId` as the key and stores a `tower::load::p2c`'s Balance instance, that

operates over the related to this id Proxy Services received through the upstream change channel. The updates are handled using another `tokio` channel internally, since Balance takes in `Discover` and later on maintains its own set of services. After taking an `UpstreamChange` from the channel provided by the framework user, the upstream ID is extracted from it and the rest of the received data is converted to `tower::discover::Change`, which is later passed on to the internal channel whose receiving part is consumed by the Balance instance. It should also be mentioned that Balance takes in a `Stream` trait implementation, which is not implemented by the `tokio` channels, so to get that functionality for a tokio channel `tokio_stream::wrappers::UnboundedReceiverStream` [20] is used.

For the router Service things are a lot simpler, since it is received over a watch channel and the value is just the Router service itself. To be able to safely update the Router value during the runtime, it is put behind `tokio::sync`'s `RwLock`. Since it is assumed that the `Router` itself should not hold any mutable state, it is required for the Router Service to implement `Clone` as well, so it can be safely cloned in the connection accepting the Server's routine with acquiring just the read lock. The write lock is only ever acquired by the Router channel polling task upon detecting a Router update.

Aside from already mentioned HTTP and TCP servers, there is also `AxumServer` provided by default. Axum is an HTTP framework for building web applications in Rust. Although it is not directly related to proxies, I believe it still is something that is going to be of use often, for such things as: providing a health check endpoint for the proxy application itself, receiving configuration updates over HTTP and exposing internal proxy data to other services. As Axum integrates with the Tower ecosystem which the framework also relies upon, it only made sense for Axum to be the web application framework to provide basic support

for. Internally, `AxumServer` does not do much, aside from calling `axum::serve` and providing graceful shutdown for it. Since `AxumServer` implements `Server` as well, it can be combined with other `Servers` using Application.

4.3 Proxy

Proxy that has been extensively referred to in this paper is neither a trait like Server nor a specific struct such as Application. A proxy is anything that implements tower's Service trait. The restrictions to that apply only in the specific Server implementation.

Figure 4.3 shows us, what specific traits must apply to a Service for it to be treated as a viable HTTP Proxy. The constraints for Send and Sync on all types under Proxy including the Proxy parameter itself arise from the internal `tokio` tasks spawning. Since `tokio`'s tasks are not guaranteed to run on the same thread throughout their lifetime, we must apply those `Send + Sync` restrictions. `'static` lifetime constraint is required for the same reason, since we cannot guarantee spawned tasks to not outlive the context they were spawned from, we cannot use a tighter lifetime.

The Proxy also must implement Load, which is a requirement to implement load balancing. To be more specific, this requirement arises from the use of tower's `p2c::Balance`. Proxies provided by the package by default (HttpProxy and TcpProxy) do not implement Load themselves, but rather leave it up to the user to decide, how should the load be measured. For most use-cases the Load can be easily implemented using utilities provided by `tower::load`, as shown in figure 4.4. In the example the Load's metric will represent the amount of currently pending requests, which is in turn going to act as the primary deciding factor for

```
impl<
        Address: ToSocketAddrs + Display + Send + Sync,
        UpstreamId: Send + Sync + Hash + Eq + Debug
            + 'static,
        UpstreamKey: Eq + Hash + Debug + Clone
            + Send + Sync + 'static,
        Router: Service<Request<Incoming>> + Send
            + Sync + Clone + 'static,
        Proxy: Service<
            Request<Incoming>,
            Response = Response<
                BoxBody<Bytes, hyper::Error>
            >
        >
            + Load
            + Send
            + Sync
            + 'static,
    > Server for HttpServer<
        Address, UpstreamId, UpstreamKey, Proxy, Router
    >
where
    Proxy::Error: Send + Sync + Error
        + Into<BoxError> + Debug,
    Proxy::Response: Send + Sync,
    Proxy::Future: Send + Sync,
    <Proxy as Load>::Metric: Debug,
    Router::Error: Send + Sync + Error,
    Router::Future: Send + Sync,
    Router::Response: Into<UpstreamId>,
```

Fig. 4.3. Type parameter constraints for the H1 Server

```
PendingRequests::new(
    HttpProxy::new("127.0.0.1:8000".parse().unwrap()),
    CompleteOnResponse::default(),
);
```

Fig. 4.4. Example of a Proxy service with a Load implementation

```
pub struct TcpProxy {
    pub target_addr: SocketAddr,
}

impl TcpProxy {
    pub fn new(target_addr: SocketAddr) -> Self {
        TcpProxy { target_addr }
    }
}
```

Fig. 4.5. TcpProxy struct definition

the load balancing algorithm.

Aside from the constraints we have already covered, the only ones left are some of the constraints on `Proxy::Error`. The `Into<BoxError>` constraint is yet again required by the `p2c::Balance`, and Debug is required for the Server to be able to log errors.

There are two default Proxy implementations provided by the package: HttpProxy and TcpProxy, each corresponding to a default implementation of a Server.

In figure 4.5 it can be seen that the only thing required to create a TcpProxy is the address of the target socket, to which the traffic should be proxied.

Figure 4.6 shows the full definition of the `HttpProxy` struct. In contrast to `TcpProxy` for `HttpProxy` there's also the HTTP client instance to be mindful about, so there currently are two functions with which a new HttpProxy instance can be created: `new` and `with_client`. `new` will instantiate an HTTP client for

```rust
pub struct HttpProxy {
    target_host: String,
    client: HttpClient,
}

impl HttpProxy {
    pub fn new(target_host: String) -> Self {
        let client = Client::builder(TokioExecutor::new())
            .pool_idle_timeout(Duration::from_secs(30))
            .pool_timer(TokioTimer::new())
            .build_http();

        HttpProxy {
            target_host,
            client,
        }
    }

    pub fn with_client(
        target_host: String,
        client: HttpClient
    ) -> Self {
        HttpProxy {
            target_host,
            client,
        }
    }
}
```

Fig. 4.6. HttpProxy struct definition

the user with a default configuration, while `with_client` can be used whenever more control over the client is desired, as it lets you provide the pre-configured client instance yourself.

4.4 Application examples

4.4.1 Simple load balancing

Figure 4.7 shows how to load balance a single upstream of 3 instances. There is a vector of hostnames for these instances, which are brought up using Docker Compose. Then, using a for loop we create a Proxy instance for each of the server instances and add them to the `DefaultUpstream`.

4.4.2 Several upstreams and routing

Figure 4.8 demonstrates how to create more than just one upstream and how to route incoming traffic to a specific one. Let's say we have two upstreams: API and frontend. For both these upstreams we define a constructor in the `UpstreamId` enum. Then we define a custom Service for the Router using `tower::service_fn`, that will route all of the incoming requests to the frontend upstream, unless the value of the Host header is equal to "api.localhost" in which case the request will be directed to the API upstream instead. After that, we create a `HttpServer` using `HttpServerBuilder`, create `HttpProxy` instances for both the frontend and API upstreams and pass them to the configuration channel.

```
#[derive(PartialEq, Eq, Hash, Clone, Copy, Debug)]
enum UpstreamId {
    DefaultUpstream,
}

#[tokio::main]
async fn main() {
    let (tx, rx) = tokio::sync::mpsc::unbounded_channel();
    let router_svc = RouteAll::new(UpstreamId::DefaultUpstream);
    let (router_tx, router_rx) = watch::channel(router_svc);
    let srv = HttpServerBuilder::default()
        .bind("0.0.0.0:8080")
        .router_channel(router_rx)
        .upstreams_channel(rx)
        .build()
        .unwrap();
    let hostnames = vec![
        "hello-world-1".to_string(),
        "hello-world-2".to_string(),
        "hello-world-3".to_string()
    ];

    for hostname in hostnames {
        let proxy = PendingRequests::new(
            HttpProxy::new(format!("{hostname}:8000")),
            CompleteOnResponse::default(),
        );
        tx.send(UpstreamChange::Add(
            UpstreamId::DefaultUpstream,
            hostname,
            proxy,
        )).unwrap();
    }

    Application::new().server(srv).serve_all().await;
}
```

Fig. 4.7. Load balancing application example

```
#[derive(PartialEq, Eq, Hash, Clone, Copy, Debug)]
enum UpstreamId {
    FrontendUpstream,
    ApiUpstream
}

#[tokio::main]
async fn main() {
    let (tx, rx) = tokio::sync::mpsc::unbounded_channel();
    let api_host = "api.localhost"
        .parse::<HeaderValue>().unwrap();
    let router_svc = service_fn(|req_info: Parts| {
        if let Some(host) = req_info.headers.get("Host") {
            if *host == api_host {
                return future::ready(
                    Ok::<_, Infallible>(UpstreamId::ApiUpstream)
                );
            }
        }

        future::ready(Ok::<_, Infallible>(
            UpstreamId::FrontendUpstream
        ))
    });
    let (router_tx, router_rx) = watch::channel(router_svc);
    let srv = HttpServerBuilder::default()
        .bind("0.0.0.0:8080")
        .router_channel(router_rx)
        .upstreams_channel(rx)
        .build()
        .unwrap();

    let api_proxy = PendingRequests::new(
        HttpProxy::new("api:8080".to_string()),
        CompleteOnResponse::default(),
    );
```

Fig. 4.8. Multiple upstreams example application

```
    let frontend_proxy = PendingRequests::new(
        HttpProxy::new("frontend:3000".to_string()),
        CompleteOnResponse::default(),
    );

    tx.send(UpstreamChange::Add(
        UpstreamId::ApiUpstream,
        "api-1",
        api_proxy,
    ))
    .unwrap();

    tx.send(UpstreamChange::Add(
        UpstreamId::ApiUpstream,
        "frontend-1",
        frontend_proxy,
    ))
    .unwrap();

    Application::new().server(srv).serve_all().await;
}
```

Fig. 4.8. Multiple upstreams example application (cont.)

4.4.3 External dynamic configuration

Figure 4.9 demonstrates an application that runs an Axum HTTP server alongside the HTTP proxy server to receive dynamic upstream configuration updates from the outside. For the sake of simplicity, we are only going to define a single upstream `DefaultUpstream`.

To perform configuration updates from an Axum handler, we need to be able to access the configuration send channel. For that, we can make use of `axum::Router`'s `.with_state()`, which will let us have state accessible from within all of the handlers using the `State` extractor. Our state is represented with the `AppState` struct, whose only field is the configuration send channel. We also define structs `AddUpstream` and `RemoveUpstream`, that will be the request data for addition and removal handlers. Then we define the handlers themselves, that using the incoming data and the configuration channel will be issuing upstream configuration updates to the proxy server.

Inside the `main` function, as usual we first instantiate our `HttpServer`. Then, we create our axum application using `axum::Router` where we specify routes and methods for our earlier defined handlers and pass the state containing the configuration channel. Having created our Axum's `Router`, we now instantiate the `AxumServer` on a different port, that will be running alongside our `HttpServer`.

Running this application, we are able to configure our only upstream by performing HTTP requests to the Axum server. For instance, sending a PUT request to `http://localhost:8999/upstreams` with a JSON body `{"target": "hello-world-1:8000", "key": "hello-world-1"}` will add a proxy to the load balancer of our upstream with key `hello-world-1`. If we later decide that this proxy needs to be removed, we can do so by sending a DELETE request to

the same endpoint with a JSON body `{ "key": "hello-world-1" }`.

4.5 Performance comparison

I have tested the performance of an application written using the framework against nginx with the same configuration proxying traffic to 3 instances of a very simple HTTP service returning "Hello world", as shown in figure 4.10. Nginx was configured to use the same Power of Two Random Choices load balancing algorithm. All of the tests were performed on the same machine in the same environment with access logs turned off. For the framework application the optimized release build was used.

For the application representing the framework, the code presented in figure 4.7 was used.

For nginx the configuration used is presented in figure 4.11. I have defined a `hello_world` upstream containing the same three addresses. The `random two;` line at the end indicates that the Power of Two Random Choices algorithm should be used for load balancing. In the `server` block I turned off the access logs using `access_log off;` so that it does not disrupt the performance.

All of the testing was performed on a machine with the following specifications: OS: Ubuntu 22.04.2 LTS on WSL2, CPU: AMD Ryzen 9 7900X, RAM: 32GB DDR5.

The benchmarking was performed using Hey HTTP load generator [21] configured to perform a total of 100000 requests with the maximum of 250 concurrent requests. I have performed 20 runs for both Nginx and the framework application and calculated the requests per second value for every run. Mean RPS across all 20 runs for both solutions with standard deviation is presented as a

```
#[derive(PartialEq, Eq, Hash, Clone, Copy, Debug)]
enum UpstreamId {
    DefaultUpstream,
}

#[derive(Clone)]
struct AppState {
    upstream_tx: UnboundedSender<
        UpstreamChange<
            UpstreamId,
            String,
            PendingRequests<HttpProxy>
        >
    >,
}

#[derive(Serialize, Deserialize)]
struct AddUpstream {
    target: String,
    key: String,
}

#[derive(Serialize, Deserialize)]
struct RemoveUpstream {
    key: String,
}

async fn add_upstream_route(
    State(state): State<Arc<AppState>>,
    Json(payload): Json<AddUpstream>,
) -> StatusCode {
    let proxy = PendingRequests::new(
        HttpProxy::new(payload.target),
        CompleteOnResponse::default(),
    );
```

Fig. 4.9. Example externally configured application: Part 1

```
    let res = state.upstream_tx.send(UpstreamChange::Add(
        UpstreamId::DefaultUpstream,
        payload.key,
        proxy,
    ));

    match res {
        Ok(_) => StatusCode::CREATED,
        Err(_) => StatusCode::INTERNAL_SERVER_ERROR,
    }
}

async fn remove_upstream_route(
    State(state): State<Arc<AppState>>,
    Json(payload): Json<RemoveUpstream>,
) -> StatusCode {
    let res = state.upstream_tx.send(UpstreamChange::Remove(
        UpstreamId::DefaultUpstream,
        payload.key,
    ));

    match res {
        Ok(_) => StatusCode::OK,
        Err(_) => StatusCode::INTERNAL_SERVER_ERROR,
    }
}

#[tokio::main]
async fn main() {
    let (tx, rx) = tokio::sync::mpsc::unbounded_channel();
    let router_svc = RouteAll::new(UpstreamId::DefaultUpstream);
```

Fig. 4.9. Example externally configured application: Part 2

```
    let (router_tx, router_rx) = watch::channel(router_svc);
    let srv = HttpServerBuilder::default()
        .bind("0.0.0.0:8080")
        .router_channel(router_rx)
        .upstreams_channel(rx)
        .build()
        .unwrap();

    let axum_state = Arc::new(AppState { upstream_tx: tx });
    let axum_app = Router::new()
        .route("/upstreams", put(add_upstream_route))
        .route("/upstreams", delete(remove_upstream_route))
        .with_state(axum_state);
    let axum_srv = AxumServerBuilder::default()
        .bind("0.0.0.0:8999")
        .router(axum_app)
        .build()
        .unwrap();

    Application::new()
        .server(axum_srv)
        .server(srv)
        .serve_all()
        .await;
}
```

Fig. 4.9. Example externally configured application: Part 3

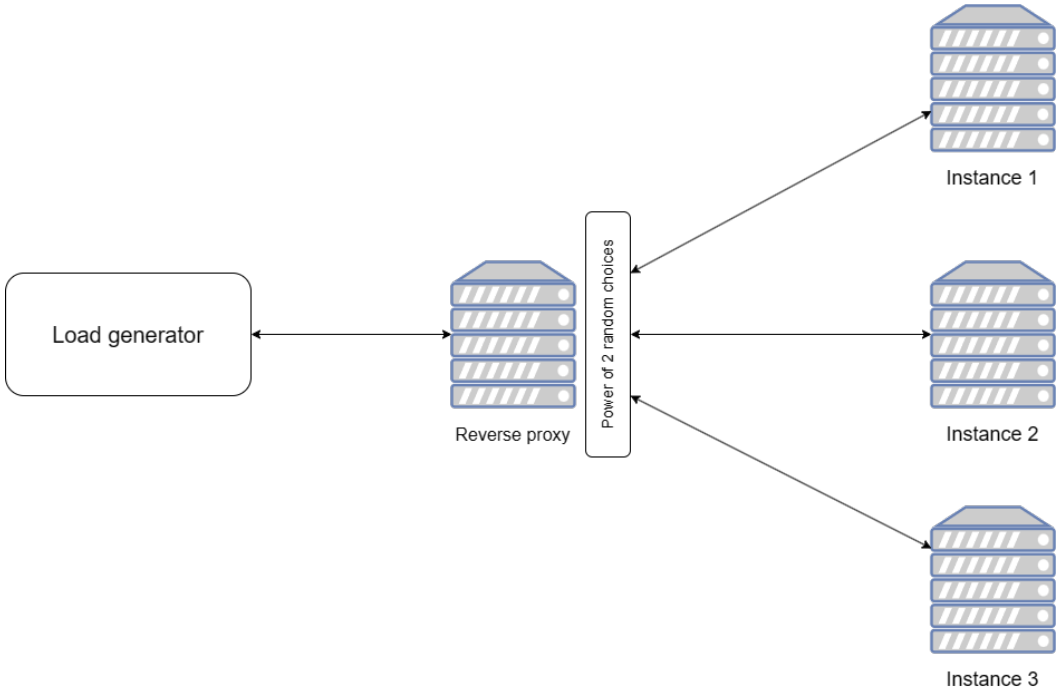

Fig. 4.10. Service setup used for benchmarking

```
upstream hello_world {
    server hello-world-1:8000;
    server hello-world-2:8000;
    server hello-world-3:8000;

    random two;
}

server {
    listen 80;
    access_log off;

    location / {
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;

        proxy_pass http://hello_world/;
    }
}
```

Fig. 4.11. Nginx configuration used in the benchmark setup

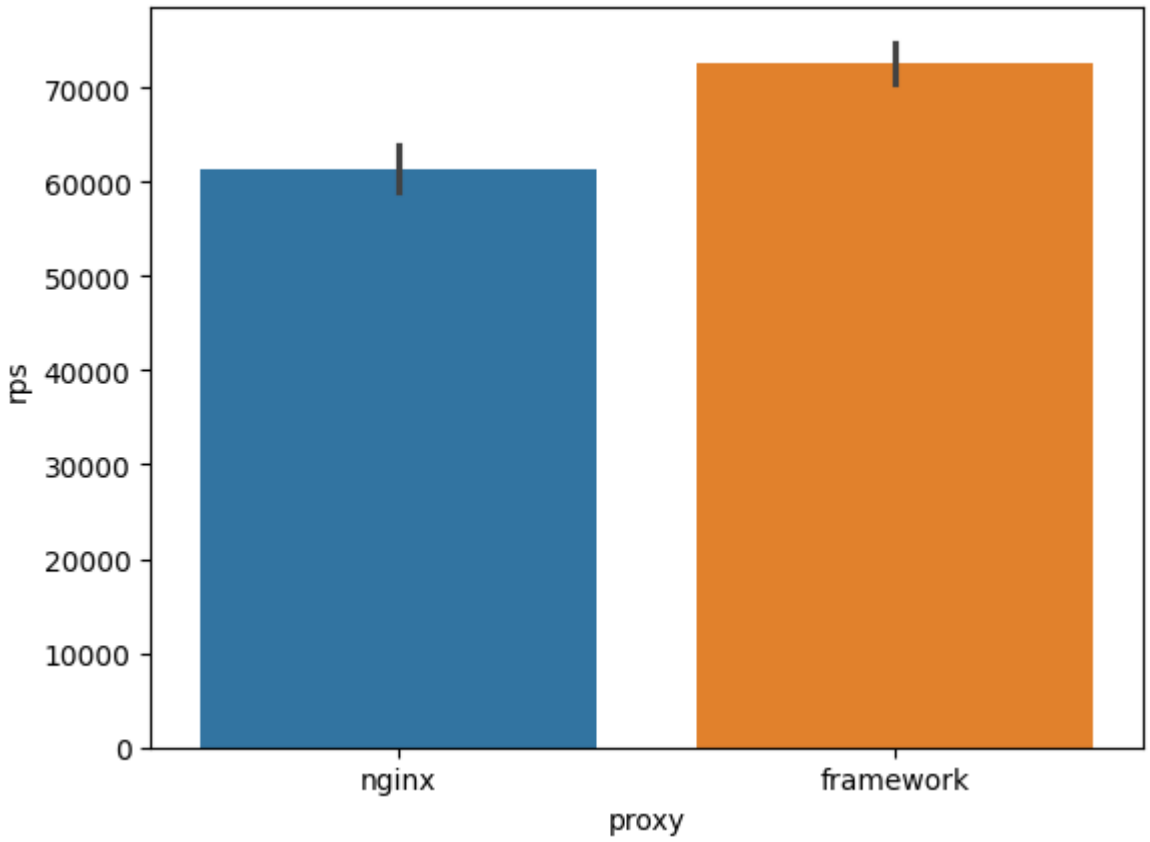

Fig. 4.12. Throughput comparison between framework and nginx

diagram in figure 4.12. Exact Mean RPS and deviation values are shown in table I. Full data on RPS for each run is available in table II. I will later be referring to these results in the Analysis and discussion chapter.

Proxy	Mean RPS	Standard deviation
Framework	72650	2064
Nginx	61418	2424

TABLE I
Benchmark results

Run	Nginx RPS	Framework RPS
1	58052	70378
2	62042	71485
3	61267	73373
4	64579	68861
5	65172	72854
6	61170	73502
7	57854	73265
8	56847	73169
9	60720	71372
10	64292	74217
11	62755	74538
12	63115	73271
13	60890	73303
14	62224	69290
15	63267	72448
16	61908	74593
17	64185	69774
18	57475	76728
19	61732	70726
20	58817	75850

TABLE II
RPS data for every single benchmark run

Chapter 5

Analysis and discussion

5.1 Performance analysis

Initially, one of the project's goals was to be performant relative to competition such as Nginx. Nginx should be making for a good comparison point, as it is one of the most used reverse proxies and held highly for its performance among many other things.

I would have considered it a success if the framework's performance came within 10% behind Nginx. As benchmarking showed, not only the framework's performance is close to Nginx it even slightly outperforms the latter. This is beyond my expectations and a very pleasant finding, although I believe is going to be insignificant in the real world applications. Nonetheless, I believe this proves enough that the framework provides sufficient performance for the end application.

However pleasant the results may be, the reason behind them is not entirely clear to me, as I am not familiar with Nginx internals enough to be able to draw any definitive conclusion. If I had to speculate about the reason, it may be that the

locking cost is cheaper for the framework or that the default workers number for Nginx was not adjusted correctly to the machine specifications which is not ever going to be a problem for the framework, since it spawns short-lived concurrent tasks on demand. In any case, specific conclusions regarding the results of this performance comparison will require additional investigation, but I believe it to be fair to at the very least say that framework's performance is comparable if not on par with that of Nginx.

5.2 Framework API overview

Aside from performance, the goals of this project included a concise yet extensible interface for the developer. Although there are unfortunate extensibility limitations that I will go through in more detail later in this chapter, all in all I believe the API turned out to be decent and allows for extensibility by the user both in terms of protocols and proxy implementation through usage of Tower and provision of Server trait alongside its default implementations.

From the comparison with similarly configured Nginx and application code examples it is seen that it is not much more work to configure the framework runtime than to write an Nginx configuration, so I believe that the API is concise enough and does not require much repetitive code from the developer.

To summarize, I consider the goals for the framework API mostly achieved and therefore the general API design more successful than not.

5.3 Limitations and future work

As of right now, the developed framework does not provide a user with a good way of modifying the incoming request in any way, which is something I believe is necessary for feature-completeness. Therefore, as part of the future work I would like to focus more on improving the Router API to allow for request data modification as a part of the Router service user-defined logic.

The framework also does not provide the user with the ability to configure the load balancing algorithm for the default implementation of the Server trait, which I also consider important for feature completeness. In the future, I would like to look into the possibility of separating load balancing logic into a standalone abstraction and provide the user with the ability to either choose from a variety of provided by default load balancing algorithms or implement their own.

One of other important for reverse proxy features that I have not yet implemented is SSL termination. At the time of writing, there's no provided by default TLS support for either TcpServer or HttpServer. So, I would also like to focus on that in the future as well, although whether it will be a Server API change or a whole another Server implementation is yet unclear.

Speaking from the production readiness perspective, not only the framework is currently lacking in mentioned above features, it also lacks extensive testing. I do not have even a clear testing methodology in mind yet, so in the future when trying to achieve production-readiness I would like to look into this in greater detail and perform more testing.

I also believe the framework in its current state lacks observability, precisely runtime metrics collection. I believe this to be a necessity for production readiness, as such I would like to look into ways metrics collection and exporting can be

introduced into the framework without hindering the performance too much.

Putting aside both feature completeness and production readiness, for future work I would also like to look into the accessibility of the framework for developers coming from backgrounds other than Rust. There are several ways to get closer to that I have in mind at the moment. The first one is trying to incorporate WebAssembly into the existing framework structure and the second one is allowing JavaScript for things such as routing and service discovery using the V8 JavaScript engine. WebAssembly would allow to write code in languages that can compile to it such as C++ and Go, and JavaScript is one of the most popular langauges in the world, so JavaScript support would definitely make the framework more approachable.

Chapter 6

Conclusion

This project was focused on development of a performant dynamic reverse proxy in the Rust programming language. The framework was developed and does cover a lot of expected from a reverse proxy features, such as load balancing, routing and implementation of both TCP and HTTPv1 protocols. There's definitely work still to be done and the framework is by no means production ready just yet, but all in all I would consider the development mostly successful, as well as the achieved performance results.

I believe that this work proves that there are still approaches to proxying incoming traffic for web services that are barely touched on and are worth exploring, as they allow for more flexibility for the user and do not compromise on performance.

The current limitations in terms of feature completeness include the inability to easily modify the incoming request data and use a different load balancing algorithm. The future work should focus on bringing this features to the framework, as well as performing more extensive testing and accessibility to the developers from non-Rust background.

Bibliography cited

[1] T. Cramer. “Asynchronous Programming in Rust.” (2018), [Online]. Available: https://rust-lang.github.io/async-book/ (visited on 05/01/2024).

[2] “Tokio - An asynchronous Rust runtime: Tutorial.” (n.d.), [Online]. Available: https://tokio.rs/tokio/tutorial (visited on 05/01/2024).

[3] C. R. Team. “Introducing Monoio: A high-performance Rust Runtime based on io-uring.” (2023), [Online]. Available: https://www.cloudwego.io/blog/2023/04/17/introducing-monoio-a-high-performance-rust-runtime-based-on-io-uring/ (visited on 05/01/2024).

[4] G. Costa. “Introducing Glommio, a Thread-per-Core Crate for Rust & Linux.” (2020), [Online]. Available: https://www.datadoghq.com/blog/engineering/introducing-glommio/ (visited on 05/01/2024).

[5] “Monoio performance test data and comparison.” (n.d.), [Online]. Available: https://github.com/bytedance/monoio/blob/master/docs/en/benchmark.md (visited on 05/01/2024).

[6] “Pingora - A library for building fast, reliable and evolvable network services.” (n.d.), [Online]. Available: https://github.com/cloudflare/pingora (visited on 05/01/2024).

[7] A. H. Yuchen Wu. "How we built Pingora, the proxy that connects Cloudflare to the Internet." (2022), [Online]. Available: https://blog.cloudflare.com/how-we-built-pingora-the-proxy-that-connects-cloudflare-to-the-internet/ (visited on 05/01/2024).

[8] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *Journal of the ACM*, vol. 46, no. 5, pp. 720–748, Sep. 1999, ISSN: 1557-735X. DOI: 10.1145/324133.324234. [Online]. Available: http://dx.doi.org/10.1145/324133.324234.

[9] O. Garret. "Inside NGINX: How We Designed for Performance & Scale." (2015), [Online]. Available: https://www.nginx.com/blog/inside-nginx-how-we-designed-for-performance-scale/ (visited on 05/01/2024).

[10] Sergio. "An Introduction to the io_uring Asynchronous I/O Framework." (2020), [Online]. Available: https://medium.com/oracledevs/an-introduction-to-the-io-uring-asynchronous-i-o-framework-fad002d7dfc1 (visited on 05/01/2024).

[11] W. Xiaoguang. "io_uring vs. epoll Which Is Better in Network Programming?" (2022), [Online]. Available: https://www.alibabacloud.com/blog/io-uring-vs--epoll-which-is-better-in-network-programming_599544 (visited on 05/01/2024).

[12] "Tower is a library of modular and reusable components for building robust networking clients and servers." (n.d.), [Online]. Available: https://docs.rs/tower/latest/tower/ (visited on 05/01/2024).

[13] "Axum is a web application framework that focuses on ergonomics and modularity." (n.d.), [Online]. Available: https://docs.rs/axum/latest/axum/ (visited on 05/01/2024).

[14] "Tonic - A native gRPC client & server implementation with async/await support." (n.d.), [Online]. Available: https://github.com/hyperium/tonic (visited on 05/01/2024).

[15] A. Richa, M. Mitzenmacher, and R. Sitaraman, "The power of two random choices: A survey of techniques and results," Oct. 2000. [Online]. Available: https://www.eecs.harvard.edu/~michaelm/postscripts/handbook2001.pdf.

[16] "Httparse - a push parser for the http 1.x protocol." (n.d.), [Online]. Available: https://docs.rs/httparse/latest/httparse/ (visited on 05/01/2024).

[17] "Hyper - an http library for rust." (n.d.), [Online]. Available: https://docs.rs/hyper/latest/hyper/ (visited on 05/01/2024).

[18] "Tokio_util - utilities for working with tokio." (n.d.), [Online]. Available: https://docs.rs/tokio-util/latest/tokio_util/ (visited on 05/01/2024).

[19] "Scalable concurrent containers - a collection of high performance containers and utilities for concurrent and asynchronous programming." (n.d.), [Online]. Available: https://github.com/wvwwvwwv/scalable-concurrent-containers (visited on 05/01/2024).

[20] "Tokio_stream - stream utilities for tokio." (n.d.), [Online]. Available: https://docs.rs/tokio-stream/latest/tokio_stream/ (visited on 05/01/2024).

[21] "HTTP load generator, ApacheBench (ab) replacement." (n.d.), [Online]. Available: https://github.com/rakyll/hey (visited on 05/01/2024).