CSCD58 Final Project Report

1. Team members
   - Phong (Paul) Ngoc Hoang Nguyen, 1010579161
   - Ning Qi (Paul) Sun, 1010294156
   - Jeremy Janella, 1010412215
2. Project Description

   We implemented a layer 4 load balancer on the two-arm architecture that can proxy TCP traffic that uses dynamically updating based on the resource load algorithm described in this paper. The load balancer is deployed between two separate networks in the Docker environment, forwarding traffic from client network to server network. On the server side, it runs software that monitors server's resources (effective network bandwidth, IO, memory, CPU utilization rate) and sends an update to the load balancer every 10 seconds for the algorithm to adjust load for each server. The load balancer is also configurable through a YAML file; it lets system administrators tune behavior without changing the code by selecting algorithms and per-rule parameters at both startup and during runtime.

   We also implemented other commonly known algorithms like round-robin or IP hashing to benchmark and compare the efficiency of the Dynamic Adjustment of Weight algorithm.

   Summary of project features:

   1. **Load Balancing Algorithms:**
      Round-robin, source IP hashing, and adaptive weight.
   2. **Health Monitoring:**
      Real-time backend health updates and integration with load balancers.
   3. **Dynamic Configuration:**
      Hot-reloading of configuration without restarting the application.
   4. **TCP Proxying:**
      Efficient and concurrent proxying of client connections to backends.
   5. **Extensibility:**

Pluggable design for creating custom routing rules: create different cluster servers, priority management, adding new or tune parameters of existing load balancing algorithms.

6. **Reliability:**
   Server attempts to reconnect to the load balancer when connection is lost. When the server terminates the connection, the load balancer handles it gracefully.
7. **Testing Infrastructure:**
   Simulated load balancer, client and backend servers for integration testing.

3. Project goals

The goal of this project is to implement and examine the efficiency of the algorithm described in the paper, as well as implementing a load balancer that supports multiple load balancing algorithms. We picked Rust as the language for the load balancer (and the server heartbeat) because we wanted to experiment with the expressiveness it provides in building abstractions, the memory safety, and the powerful tools it provides for concurrent programming.

4. Project contribution

Phong (Paul) Ngoc Hoang Nguyen:

- Set up the Docker images and Docker network topology for clients, load balancers and servers
- Implemented resource monitoring (health check) software running on the server, and load balancer
- Implemented and write unit tests for Dynamic Adjustment of Weight and IP hashing load balancing algorithm
- Contributed to process and handle server metrics on the load balancer, set up iperf server to measure maximum network bandwidth

Ning Qi (Paul) Sun

- Implemented matching and routing rules for incoming connections:

        - Inbound connections get grouped by ports, and for each port, there is a separate routing table that performs longest prefix match to find a matching set of backends and algorithm to use.
    - Implemented connection proxying logic
        - Mostly metadata and logging.
    - Implemented config module for the load balancer:
        - Generates data structures used for load balancing: initializes Balancer structs, creates RoutingTables
    - Implemented round robin algorithm for load balancing

Jeremy Janella

- Research proxy layer implementations
    - developed experimental data link, network, transport, and application layer proxies
- Implemented async parallel tcp-proxy using tokio
- Review and implementation of load balancer test cases

5. Implementation details

We made this program in Rust and used Docker network for serving the infrastructure.

Folder structure of src:

- main.rs - Entry point of load balancer. Reads the config file initially, sets up the file change watcher to auto reload config later. Starts the listeners for incoming connections, and performs the rule matching logic.
- balancer module - Defines some load balancer algorithms
    - balancer/mod.rs - Defines the Balancer trait that all the balance structs must implement (for choose_backend() method)
    - balancer/adaptive_weight.rs - Implement algorithm on the mentioned paper (see project description) for choose_backend() on Balancer trait, as well as defining a structure to keep track of required coefficient and weights for the algorithm + unit tests.
    - balancer/ip_hashing.rs - Implement IP hashing algorithm for choose_backend() method on Balancer trait.

    - balancer/round_robin.rs - Implement round-robin algorithm for choose_backend() method on Balance trait.
- backend module - Management and maintenance of backend pools
    - backend/mod.rs - Defines Backend structure which represents a single backend from the config file, and BackendPool structure which represents a cluster of Backends.
    - health.rs - A structure to keep track and update of server's physical health statistics, listen to newly updated server metrics and start an independent iperf server.
- config module - Configuration of the load balancer
    - config/mod.rs - Defines structs for the YAML configuration
    - loader.rs - Takes a parsed configuration, creates the rules and routing tables
- proxy module - Proxy logic for different protocols (only TCP at the moment)
    - proxy/mod.rs - Defines metadata and logging struct ConnectionContext.
    - tcp.rs - Connection proxying for TCP connections.

Routing details:

When a TCP connection is made to us, it will first be accepted by a port listener. This is because a TcpListener can only bind to one port.

Once we have it, we consult the RoutingTable for that port. In here we have a bunch of IP address ranges in CIDR notation, and these are stored as a pair with a Balancer. We perform longest prefix matching to find which one we use.

The Balancer struct has a method choose_backend() that we then call to pick a good backend, and then we proxy the connection between the client and the chosen backend.

(Balancer is a struct, because it needs information about what set of Backends its allowed to load balance around, so most of the

algorithms store a BackendPool struct internally, which is generated by the config).

Config module:

The config is two parts: parsing the definitions in the YAML file, and then actually turning it into data structures the load balancer core is able to use.

Parsing is done by defining a couple of structs with fields corresponding to what the fields in the configuration file should be. Then, we use serde crate to parse the configuration file and fill out our configuration struct for us. (I cannot claim any credit here, but the way it works is very smart: it uses procedural macros to inspect the fields in the structure definition, and automatically generates the code that reads and parses from the YAML file. So as long as you write your struct definition, you can parse the YAML easily).

Once we have the parsed file, we need to actually turn it into data that the load balancer core can use to do its job.

Recall the structure of the rules: we have a set of ip/subnet:port as the clients (we need to match a client with one of these). Then we have the target, aka the set of backends that these clients are allowed to use. But also recall the load balancer core structure: We have concurrently running threads, one for each port that we need to listen on.

So we first group up these rules by client port (if a rule has multiple different ports for clients, we just split it into two rules). Then for each port, we read all the rules, and add the CIDR IP address + the requested Balancer, into a RoutingTable. We put this port + RoutingTable into a hashmap, and once we're done with all of them, we return this to main.rs

Though handled by main.rs, the config auto reload is not too special, it just spawns a new thread that waits for the file to change, and then

when it does, we need to redo this portion. To achieve no downtime and no dropped / killed connections during the reload, we need a way to give the listener threads a new routing table. We achieved this with a MPSC channel, and modified the accept() loop to also monitor this channel (We can use tokio's select! macro, which works similarly to select() on Linux, to be able to wait for any of these two possible events to happen, in any order and frequency).

Backend module:

A backend is a single endpoint. Put simply, each IP+port from the backends list in the config will become a Backend struct.

A BackendPool is a vector of shared Backends, because Backend has a field active_connections, meaning it could belong to multiple rules (and hence multiple Balancer structs) and ports (entirely different threads).

The module also manages backend health metrics through the ServerMetrics struct, which tracks resource usage (CPU, memory, network, and I/O). These metrics are updated dynamically via the start_healthcheck_listener function, which listens for health updates from backends, processes incoming JSON metrics, and updates the shared ServerMetrics for each backend using thread-safe Arc<RwLock>. This integration ensures that load balancers have real-time data to make informed decisions.

Balancer module:

The balancer module is the core of the load balancing logic, defining the Balancer trait and implementing multiple load balancing algorithms. The Balancer trait provides a unified interface with the choose_backend method, which selects an appropriate backend for a given connection. Each algorithm is implemented as a struct that adheres to this trait, allowing for pluggable and extensible load balancing strategies. The module includes the following algorithms:

1. Round Robin (round_robin.rs):

   Implements a simple round-robin strategy, cycling through backends in order. It maintains an index to track the next backend to use, ensuring even distribution of connections.

2. Source IP Hashing (ip_hashing.rs):

   Uses a hash of the client's source IP to consistently route connections from the same client to the same backend. This ensures session stickiness and is useful for stateful applications.

3. Adaptive Weight (adaptive_weight.rs):

   A more advanced algorithm that uses backend health metrics (CPU, memory, network, and I/O) to calculate weights dynamically. It adjusts weights based on resource availability and selects backends accordingly. The implementation includes partial support for the W110 paper's algorithm, with features like threshold-based selection and randomized adjustments.

The balancer module is designed to work seamlessly with the backend module, leveraging shared Backend instances and real-time ServerMetrics for decision-making. Each algorithm is encapsulated in its own file, ensuring modularity and ease of extension. The mod.rs file acts as the entry point, exposing the Balancer trait and the implemented algorithms. This design allows the system to support diverse balancing strategies while maintaining a consistent interface.

6. Documentation

The load balancer is configured with a YAML file, and has the ability to automatically reload when the file is edited, allowing you to change rules without dropping any connections.

The configuration file consists of:

- Defining your health response address (IP + port)
- Defining your iperf server address (IP + port)

- A list of backends, which are the server IP + port that you will be load balancing with.
- A list of clusters, which each are a group alias for a set of backends.
- A list of rules, which consists of :
    - Clients: One or more IP address ranges (written in CIDR notation) + port number, which we will use to match an incoming client with.
    - Targets: One or more clusters of backends.
    - Strategy: A chosen load balancing algorithm to use. When we match an inbound connection to this rule, we use the algorithm to match the connection to one of the target backends.
        - Depending on the strategy, there may be more configuration, like the weights for the adaptive algorithm.

Here is a sample of the configuration:

```
healthcheck_addr: "10.0.1.0:9000"

iperf_addr: "10.0.1.0:5201"

backends:
  - id: "srv-1"
    ip: "10.0.1.1:8081"

  - id: "srv-2"
    ip: "10.0.1.2:8082"

  - id: "srv-3"
    ip: "10.0.1.3:8083"

  - id: "srv-4"
    ip: "10.0.1.4:8084"


clusters:
  main-api:
    - "srv-1"
    - "srv-2"
  priority-api:
    - "srv-3"
    - "srv-4"
```

```
rules:
  - clients:
      - "0.0.0.0/0:80"
    targets:
      - "main-api"
    strategy:
      type: "RoundRobin"

  - clients:
      - "10.0.0.0/24:80"
      - "10.0.0.0/24:8080"
      - "10.0.1.0/24:8080"
    targets:
      - "main-api"
      - "priority-api"
    strategy:
      type: "Adaptive"
      coefficients: [ 1.5, 1.0, 0.5, 0.1 ]
      alpha: 0.75
```

The internal network containing the servers are set to be IPs in 10.0.1.0/24. The backend servers will periodically send us information about their resources. We define two clusters, a main cluster and a priority cluster.

For our clients, we have two rules: Everyone, on port 80 only, has access to the main cluster, and are served using the round robin balancer. For anyone on 10.0.0.0/24 specifically, on port 80 and additionally 8080, as well as anyone on 10.0.1.0/24 but only port 8080, they get access to all 4 backends in the main and priority cluster. Their load balancing will use the adaptive load balancing algorithm instead of round robin.

Internally, clients are matched with rules by checking the ports, and then using longest prefix matching. So although the first rule does cover all port 80 traffic, since the second rule is more specific, it will be the one that clients on 10.0.0.0/24:80 get matched to.

7. Conclusion and lesson learn
    - Learning Rust is a challenge because it offers a new mindset of writing code. The compiler's strict rules around borrowing, error handling, and concurrency can be overwhelming but it will teach us how to write safer and more deliberate code (Anecdote from writing this project: If the code managed to compile, there's a good chance it is correct already. Such are the benefits of an expressive language, with a good type system). Additionally, Rust offers a variety of tools to manage concurrency, abstraction and crates (libraries) used for different purposes like asynchronous network socket, extracting system information, serializing and deserializing data in different formats like JSON, YAML easily. This makes programming experience with Rust both painful and less painful at the same time.
    - Building a network environment with Docker is a fun experience as Docker offers various options to configure your network, hosts, and even the host's physical configuration! Docker is also convenient as it is easy to deploy, make changes, share the environment, and ensures the software run does not produce unexpected errors when running on the other machine.
    - Deciding on which layer to implement the proxy was a trade-off between speed, complexity, and being able to support different features. Implementing a proxy at the data link layer would offer the fastest performance using cut-through forwarding, the performance gain is marginal unless implemented in dedicated hardware, and it still doesn't solve the problem of routing sessions. A network layer implementation, where we would modify IP packet destination headers, introduces the significant complexity of manually tracking user sessions to ensure persistence (all packets from one user go to the same server). Therefore, implementing the load balancer at the transport layer or application layer provides the best balance. The TCP layer already handles sessions, making it suitable for forwarding. The application layer offer would have the server relaying HTTP requests, however would limit our program to only working with HTTP. It was decided to use TCP which supports a much broader range of protocols

which are built on it (including HTTP), allowing more versatility of the balancer.