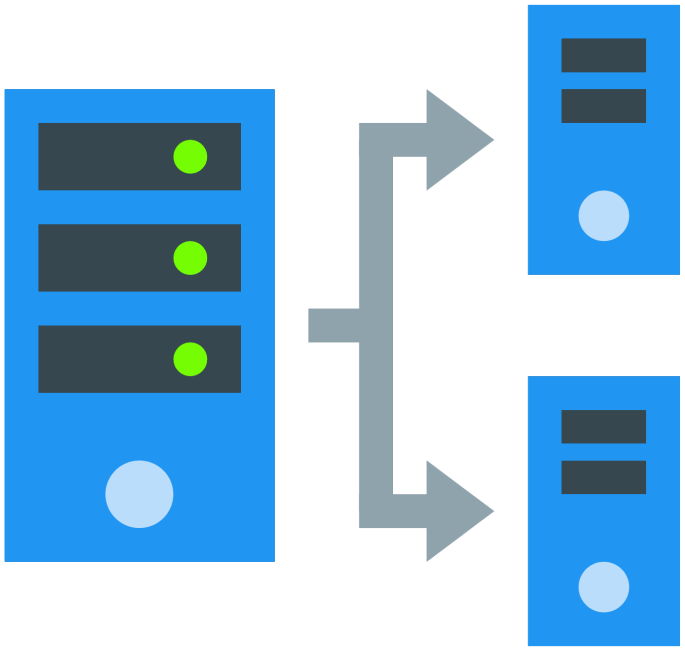
Université Moulay Ismail

Ecole Supérieure de Technologie – Meknes

Filière : LP DSIC

Mini project

Author: Mohammed Salih



Theme: Development of a load balancer

Table of Contents

I. Introduction

1. Definition

Load balancing is a key component for highly-available infrastructure commonly used to increase both performance and reliability of web apps, databases and many other services by distributing the workload across multiples servers hosting the same service.

2. Goal

The goal of this project is to develop a load balancer, which can be deployed with different types of services to balance traffic between multiple backend servers.

3. Difference between normal and load balanced infrastructure

Supposing we have the following website infrastructure with no load balancing:

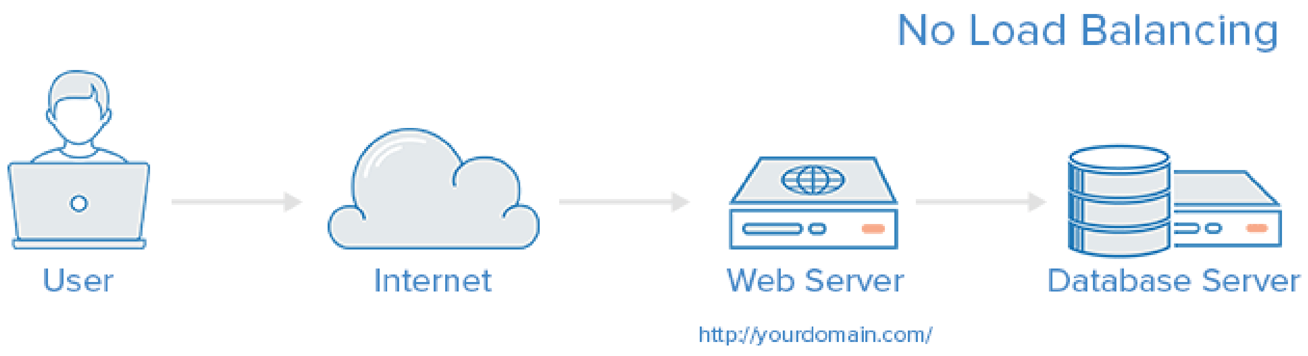


Figure 1: Simple website hosting infrastructure

The user in this instance accesses our only webserver at *yourdomain.com*. If anything happens to the webserver, it becomes a single point of failure for our solution. This downtime may prevent users from accessing our website until it is restarted, additionally if our webserver can't handle huge load of traffic, this may lead to slow response times, or may ultimately crash.

This single point of failure may be mitigated by introducing a load balancer, and another webserver hosting the same website. Typically, all webservers should host the same application so identical content can be supplied to the users no matter the instance responding to their queries.

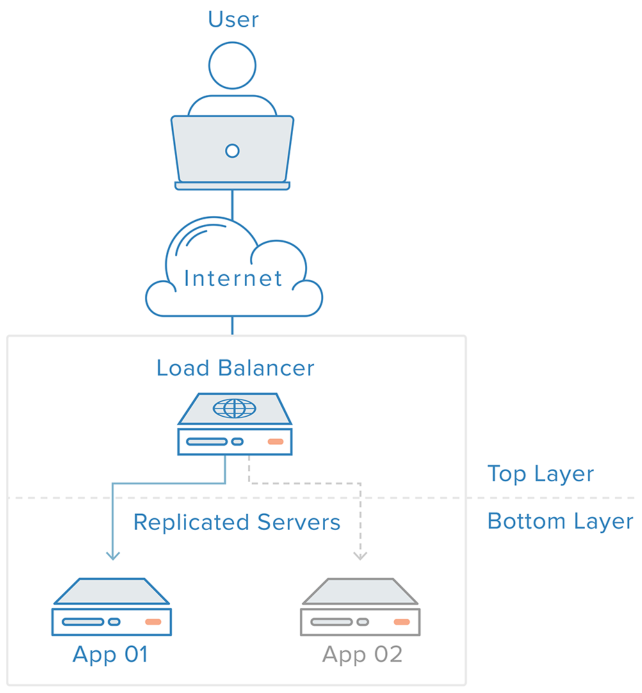


Figure 2: Website hosting infrastructure with load balancer

With the addition of the load balancer acting as a gateway for users, when a new query is made to our website, the load balancer forwards this query to one of the backends, that directly responds to the user. The load balancer becomes now the single point of failure of our infrastructure, but can be mitigated with the introducing multiple load balancers.

A load balancer uses algorithms to determine the path of a query, and may support different types of protocoles not just web applications.

4. Type of traffic handled by a load balancer

A load balancer can support different types of traffic from which there are four main ones:

- HTTP: standard balancing for HTTP directs requests based on standard HTTP mechanisms, load balancer appends some headers to give information to backend.
- HTTPS: works same way as HTTP, with addition of encryption, it has two scenarios: either with SSL passthrough, which maintains encryption all the way to the backend, or with SSL termination which places the responsibility of the decryption on the load balancer but sends the plain text traffic to the backend.
- TCP: Some applications use different protocol than HTTP, they all can be supported by TCP mode.
- UDP: More recently, some load balancers, added UDP to support core internet protocols like DNS.

In our project, we will focus on two types of traffic: HTTP (HTTPS included) and TCP.

5. How a load balancer chooses a backend

Load balancers chooses which backend to forward a request to on combination of two key factors ensuring that chosen backend is responding appropriately to requests and use a pre-configured rule to choose a backend from that healthy pool of servers.

5.1. Health check

Load balancers should only forward traffic to *healthy* backend servers. To monitor the health of every backend, health checks are regularly done by attempting to connect to the backend server by the address and port specified in the forwarding rules to ensure that the service is still listening. If a backend server fails the health check it is removed from the healthy pool, and further requests won't be forwarded to it until it responds to future health checks.

5.2. Load balancing algorithm

A load balancing algorithm determines which backend server of the healthy pool a request will be forwarded to. Common algorithms are:

- Round Robin: backend servers are selected sequentially. The load balancer will select the first backend server for the first request, moving down the list, and return to top when the list ends.
- Least connections: Load balancer will choose server with least number of connections, it is the recommended load balancing algorithm for long lived connection protocols.
- Source: In source algorithm, the load balancer will select which backend server to use based on a hash of the source IP of the request. This method ensures that same client is always connected to the same backend server. Useful for applications that require users to be always to the same backend server.

For the purposes of this project, both round robin and least connections algorithms are implemented.

II. Project: Noxy

1. Overview

Due to the networking nature of *noxy (formal name of the project)*, having an asynchronous model for handling requests is key to process big volumes of traffic, and as a consequent, the asynchronous networking library *netty* will be used in this, project.

Noxy is split into three core packages:

- Configuration: initial configuration processing before start of load balancing.
- Server: *Noxy*'s critical section, receiving incoming connection and forwarding them to backend servers according to predefined rules in configuration.
- Balancer: balance loading algorithms, used to decide which backend server request will be forwarded to.

2. Configuration file

Configuration file component understanding is essential, because any modification of behavior is put in it.

A simple configuration file may look like:

```
global:
  maxconn: 10

defaults:
  timeout_connect: 10000 # 10 seconds in ms
  timeout_client: 30000 # 30 seconds in ms
  timeout_server: 30000
  maxconn: 1
  mode: tcp

frontend:
  - name: http_front
    mode: http
    bind:
      - addr_port: 127.0.0.1:3030
    use_backend: http_backend

backend:
  - name: http_backend
    balance: roundrobin
    servers:
      - addr_port: 127.0.0.1:8000
      - addr_port: 127.0.0.1:8001
```

Text 1: A simple configuration file

As you may have noticed, configuration files are YAML based.

Every configuration files must contain four global fields:

- `global`: Contains global settings that, `maxconn: 10` in this case signifies the max number of connection this load balancer will ever accept at once.

- `frontend`: A single frontend is a collection of listening addresses (IP address and port) in which all rules applied on this collection, apply on all listeners. Multiple frontends can be declared at once. For instance we have here a single frontend, `name: http_front` says we have a frontend with that given name, `mode: http` specifies the traffic we accept. `bind` is an array for all addresses we want to listen on, finally `use_backend: http_backend` to specify which backend we want to forward to.

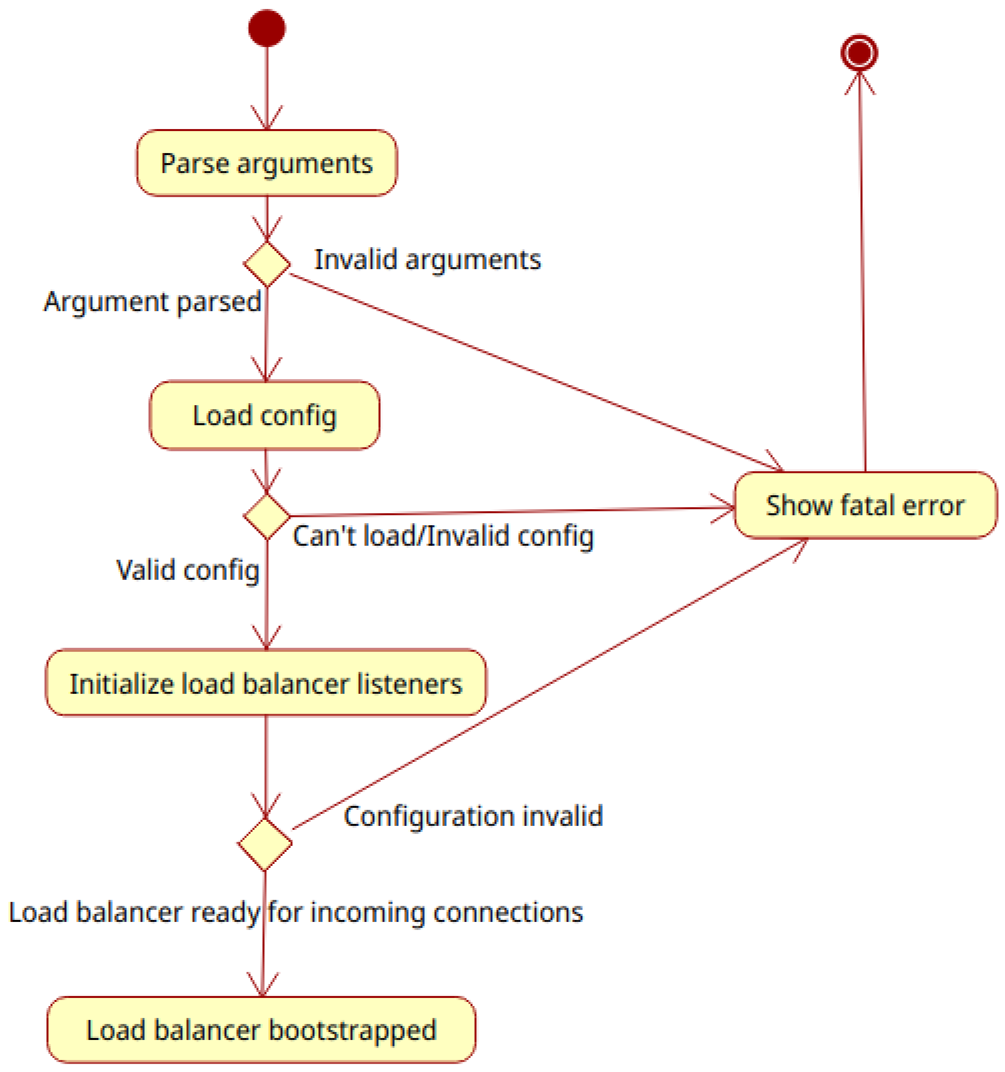
- `backend`: A single backend is a collection of reachable addresses, that should host same service. Here we do only have a single backend, `name: http_backend` is to the specify the name of this backend. `balance: roundrobin` to specify what load balancing algorithm to use for this backend. `Servers` is a list of all the servers of this backend.

- `defaults`: Multiple frontends may be defined at once in a single configuration, defaults are shared values between all these frontends, for example `timeout_connect: 10000` means that connections to backend server a timed out 10 seconds if we can't establish connection to backend, `timeout_client: 30000` we wait 30 seconds before we timeout connection with client if there is nothing to read, same for `timeout_server: 30000` but it in this case it is for server. `maxconn: 1` specifies that a frontend may only accept one connection in a given time, essentially eliminating concurrent access. `mode: tcp` specifies the traffic we accept, in this case we transparently forward traffic to backend without knowing what application protocol is used. Default values can be overridden in a frontend by defining same value.

3. Load balancer life cycle

3.1. Bootstrapping

When *noxy* is started with a given configuration files, it must follow a certain process for it to be able to start listening for incoming requests:



Parse arguments
Invalid arguments
Argument parsed
Load config
Show fatal error
Can't load/Invalid config
Valid config
Initialize load balancer listeners
Configuration invalid
Load balancer ready for incoming connections
Load balancer bootstrapped

Figure 3: Activity diagram of noxy bootstrapping

3.2. Listener

After successfully loading the configuration file and server is listening for new connections. Every new connection made to the load balancer follows the next procedure:

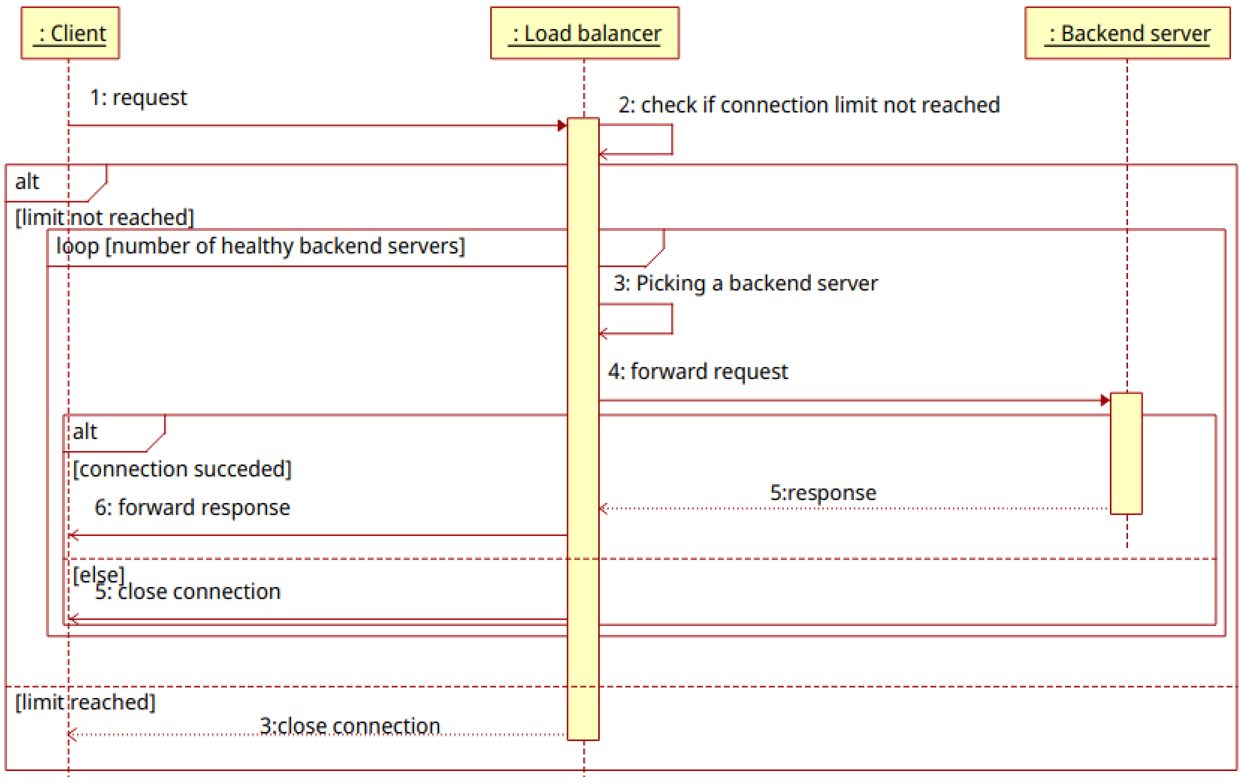


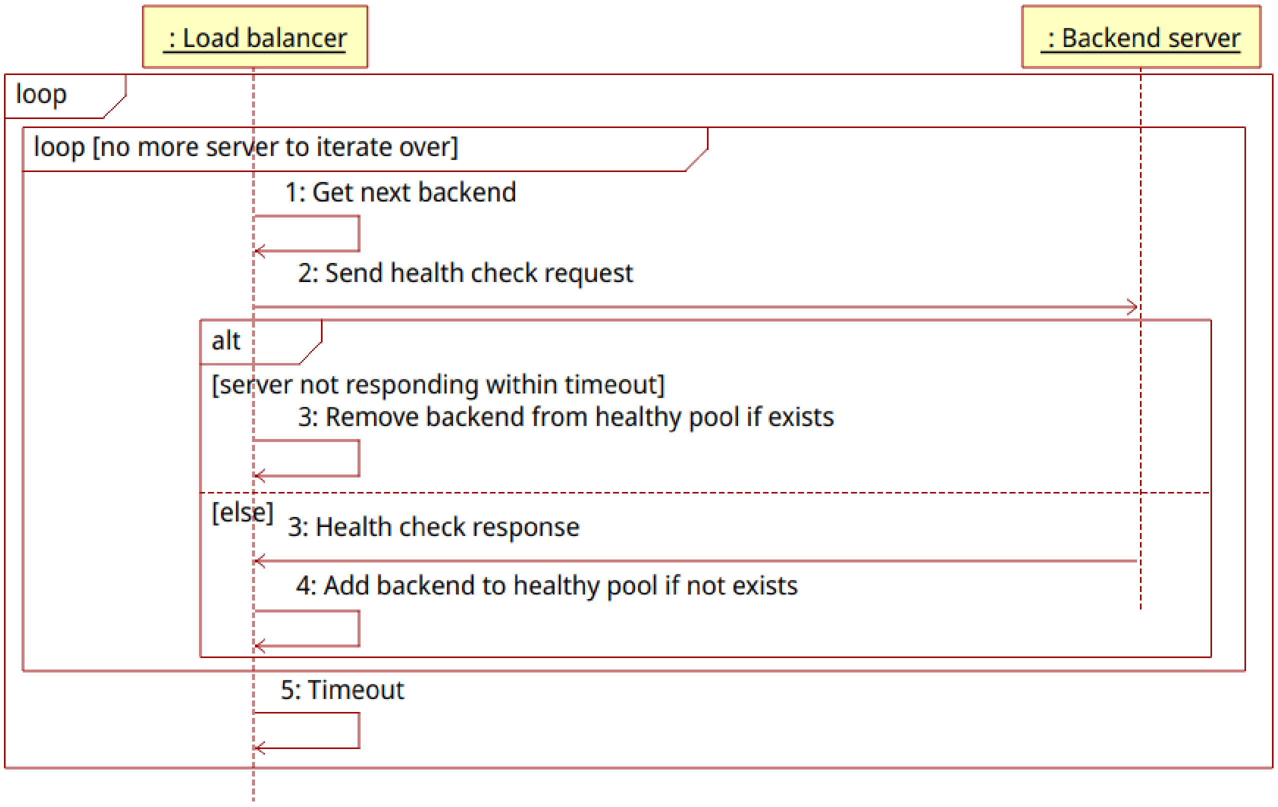
Figure 4: Sequence diagram of accepting new connections

3.3. Backend picking

Whenever a new connection that respects predefined rules in the configuration is made, the load balancer must choose a backend which the request will be forwarded to. The load balancer has two criteria to pick a backend as discussed before.

3.3.1. Health check

A periodical ping sent to every backend, in which only healthy backends that reply with good responses are kept in a list called a healthy pool, the procedure can be summarized with the following diagram:

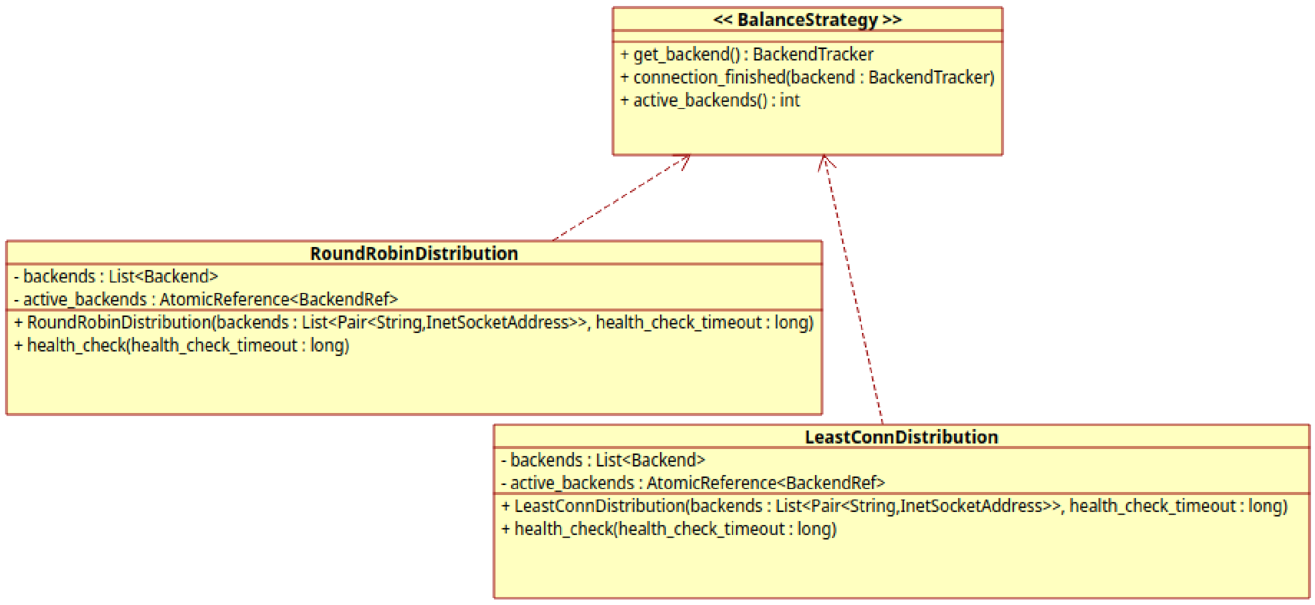


: Load balancer
: Backend server
loop
loop [no more server to iterate over]
1: Get next backend
2: Send health check request
alt
[server not responding within timeout]
3: Remove backend from healthy pool if exists
[else]
3: Health check response
4: Add backend to healthy pool if not exists
5: Timeout

Figure 5: Sequence diagram of health check process

3.3.2. Load balancing algorithm

To decide which backend will handle a request, a load balancing algorithm must be used. To have a uniform way of interacting with these load balancing algorithms (currently, there are two different algorithms: round robin and least connection), they all have the same interface:



<< BalanceStrategy >>
+ get_backend() : BackendTracker
+ connection_finished(backend : BackendTracker)
+ active_backends() : int
RoundRobinDistribution
- backends : List<Backend>
- active_backends : AtomicReference<BackendRef>
+ RoundRobinDistribution(backends : List<Pair<String,InetSocketAddress>>, health_check_timeout : long)
+ health_check(health_check_timeout : long)
LeastConnDistribution
- backends : List<Backend>
- active_backends : AtomicReference<BackendRef>
+ LeastConnDistribution(backends : List<Pair<String,InetSocketAddress>>, health_check_timeout : long)
+ health_check(health_check_timeout : long)

Figure 6: Load balancing algorithms interface

To knew more about how a load balancing algorithm works, the following diagram explains the procedure taken by the round robin algorithm to choose a backend:

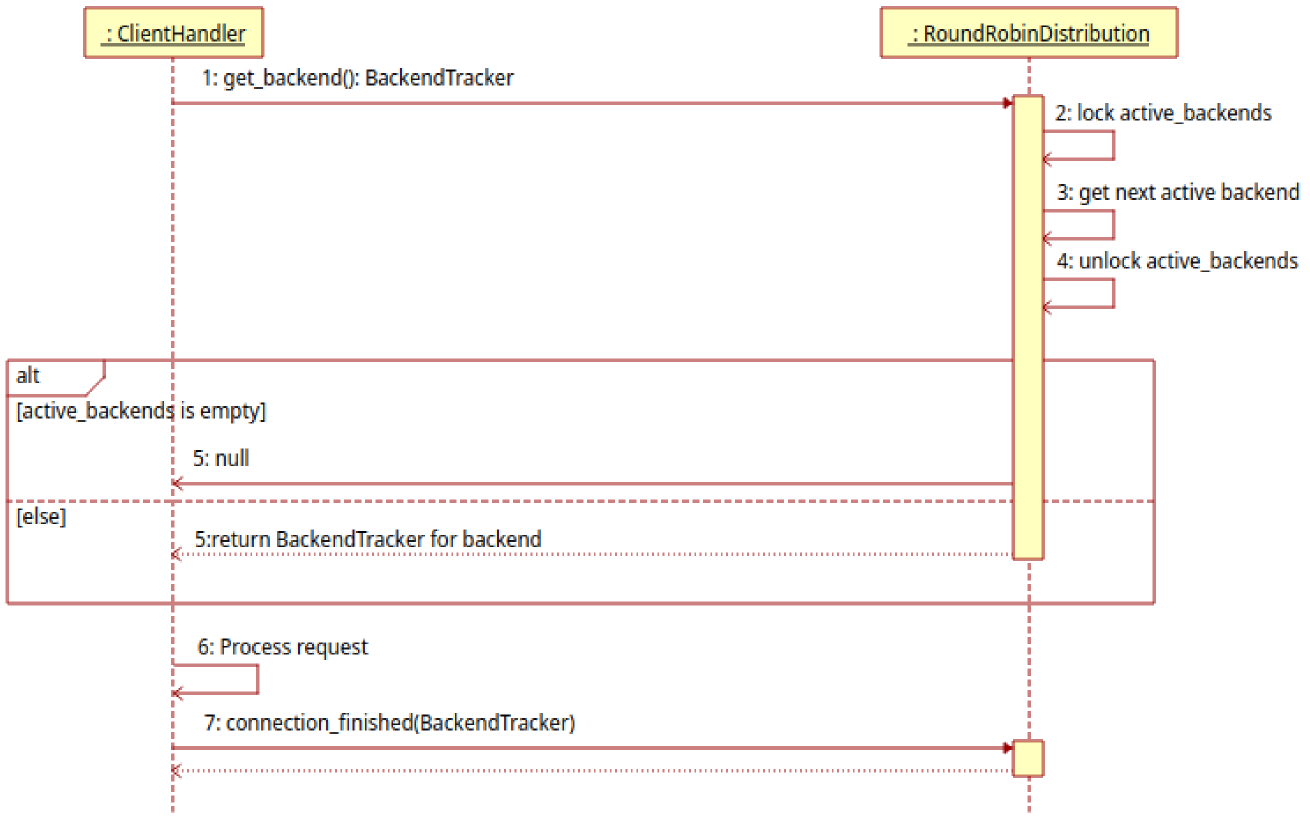

: ClientHandler
: RoundRobinDistribution
1: get_backend(): BackendTracker
2: lock active_backends
3: get next active backend
4: unlock active_backends
alt
[active_backends is empty]
5: null
[else]
5:return BackendTracker for backend
6: Process request
7: connection_finished(BackendTracker)

Figure 7: Sequence diagram of round robin load balancing algorithm

III. Deployment

Now that we’ve seen the internals of *noxy*, it’s time to take a look at some scenarios on how *noxy* might be used.

For the purposes of this demonstration we will be using *docker* along with *docker-compose* so every scenario can be fully reproducible on different machines.

All scenarios can be found in ‘*./deployment/*’ folder that comes with this report.

Compiled executables used for this demonstration can be found in ‘./bin/’.

1. Simple load balancing scenario

Supposing that we have the following scenario:

- We have multiple backend servers that serve the same web application.
- Our backends don’t support *https*, but we want to have it (SSL termination).
- Redirect all incoming requests from *http* to *https*.

1.1. Setup

For this scenario, we have two backends that are two instances of *nginx* webserver, one load balancer that hosts *noxy*, all these instances are on the same network, and the only reachable services are the ones hosted on the load balancer (port 80 for *http* and 443 for *https*).

For the container hosting *noxy*, we bind port 80 and 443 to the same port on localhost, we expose the configuration file inside the container, and we also expose both the TLS certificate and private key.

For both the two backends, each of them are given a directory which hosts a static web applications, they are completely identical, except we have a string that we can distinguish with who replies to a request when we send one.

The following compose script does what we described above to setup our services:

```yaml
version: "3"
services:
  noxy:
    build:
      context: .
      dockerfile: NoxyDockerfile
    container_name: noxy
    ports:
      - "80:80"
      - "443:443"
    volumes:
      - type: bind
        source: ./simple.yaml
        target: /app/conf.yaml
      - type: bind
        source: ./ssl_cert/cert.pem
        target: /app/cert.pem
      - type: bind
        source: ./ssl_cert/key.pem
        target: /app/key.pem
    environment:
      - "org.apache.logging.log4j.level=INFO"
    networks:
      - back_network

  webserver1:
    image: docker.io/nginx
    container_name: webserver1
    volumes:
      - ./content_1/:/usr/share/nginx/html
    networks:
      - back_network

  webserver2:
    image: docker.io/nginx
    container_name: webserver2
    volumes:
      - ./content_2/:/usr/share/nginx/html
    networks:
      - back_network

networks:
  back_network: {}
```

Text 2: Docker compose configuration file for a simple scenario

1.2. Configuration file

Noxy will have a single frontend that listens on two port (80 and 443), for *https* port we will specify TLS certificate. And we will have a condition that redirects all requests from *http* to *https*. All along with some other parameters like limiting number of concurrent connections so that we can test this later on.

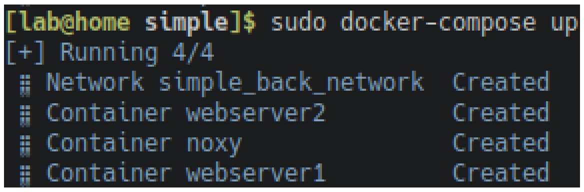
To achieve what was said before, the following configuration was used:

```
global:
  maxconn: 20 # limit max connecions to 20
  threads: max # use all cpu threads availables
  health_check: 5000 # do a health check on backends every 5 seconds

defaults:
  timeout_connect: 10000 # 10 seconds in ms
  timeout_client: 30000 # 30 seconds in ms
  timeout_server: 30000
  maxconn: 20 # We only have a single frontend, so global maxconn must match
frontend's maxconn
  mode: http

frontend:
  - name: http_front
    bind: # all addr:port we want to listen on
      - addr_port: 0.0.0.0:80
      - addr_port: 0.0.0.0:443
        ssl: # we load a certificat for this particular addr:port
          cert: /app/cert.pem
          key: /app/key.pem
          versions: # tls versions allowed
            - TLSv1.3
          alpn: # http version negotiation accepted in tls handshake
            - h2
            - http/1.1
    use_backend: http_backend
    http_condition:
      - when: "scheme == 'http'" # redirect condition, activated when scheme is
http
        redirect: "sprintf('https://127.0.0.1%s', uri)" # url to redirect to,
usually this should be a domain and discouraged to be an ip address

backend:
  - name: http_backend
    balance: roundrobin
    servers: # list of backends, docker auto resolves domain to corresponding ip
address
      - addr_port: webserver1:80
      - addr_port: webserver2:80
```

Text 3: Noxy configuration file for a simple scenario

1.3. Testing

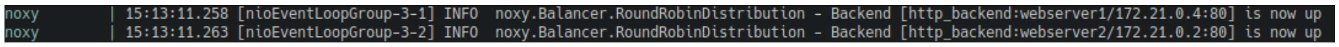
Now that we have configured all the necessary configurations files, we may start our containers, like the following:



```
[lab@home simple]$ sudo docker-compose up
[+] Running 4/4
 ⠿ Network simple_back_network  Created
 ⠿ Container webserver2         Created
 ⠿ Container noxy               Created
 ⠿ Container webserver1         Created
```

Figure 8: Starting simple scenario docker containers using docker-compose

Once we are fully bootstrapped, we can notice that both of our backends are up and running, and that *noxy* did already add them to the healthy pool:

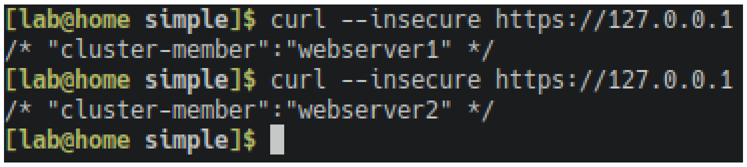


```
noxy        | 15:13:11.258 [nioEventLoopGroup-3-1] INFO  noxy.Balancer.RoundRobinDistribution - Backend [http_backend:webserver1/172.21.0.4:80] is now up
noxy        | 15:13:11.263 [nioEventLoopGroup-3-2] INFO  noxy.Balancer.RoundRobinDistribution - Backend [http_backend:webserver2/172.21.0.2:80] is now up
```

Figure 9: Checking status of backends from logs

1.3.1. Availability

To check that all our backends are handling requests, we can now send a request:



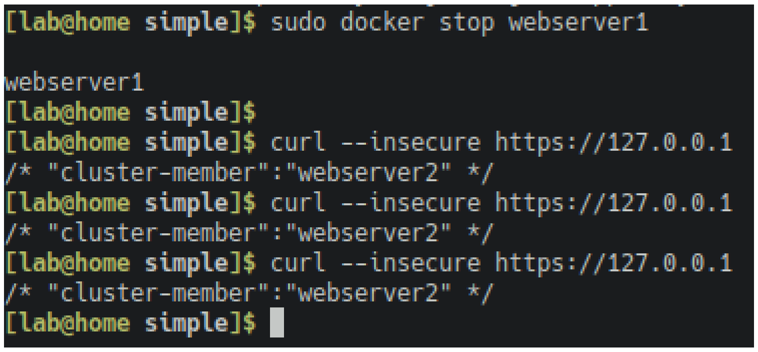
```
[lab@home simple]$ curl --insecure https://127.0.0.1
/* "cluster-member":"webserver1" */
[lab@home simple]$ curl --insecure https://127.0.0.1
/* "cluster-member":"webserver2" */
[lab@home simple]$
```

Figure 10: Sending requests to the load balancer

Ah! We can see that both backends are responding, and in sequence because we are using the round robin algorithm algorithm.

Now comes the hard part, verify that we are still getting all responses successfully to our requests when one of these backends goes down.

We quickly take down one of the two backends, and try to send multiple requests:

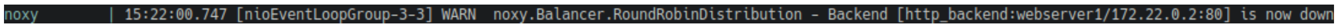


```
[lab@home simple]$ sudo docker stop webserver1

webserver1
[lab@home simple]$
[lab@home simple]$ curl --insecure https://127.0.0.1
/* "cluster-member":"webserver2" */
[lab@home simple]$ curl --insecure https://127.0.0.1
/* "cluster-member":"webserver2" */
[lab@home simple]$ curl --insecure https://127.0.0.1
/* "cluster-member":"webserver2" */
[lab@home simple]$ 
```

Figure 11: Verifying failing recovery capabilities of the load balancer

We can see that when we took *webserver1*, now only *webserver2* is responding.

And if we go back to *noxy*’s logs, we can verify that *noxy* detects that *webserver1* is down and takes it down from the healthy pool:

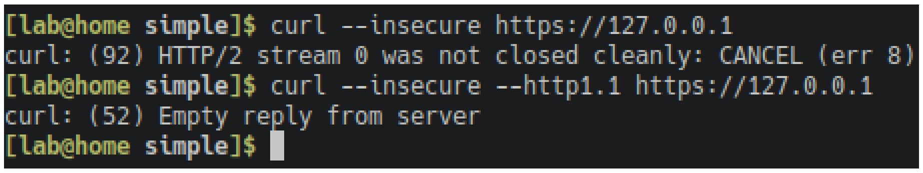


```
noxy        | 15:22:00.747 [nioEventLoopGroup-3-3] WARN  noxy.Balancer.RoundRobinDistribution - Backend [http_backend:webserver1/172.22.0.2:80] is now down
```

Figure 12: Verifying backend status from logs

The question now might be what happens when all backends are down, sadly at that point, *noxy* at that point will close all incoming connections until one of the backends returns online.

We verify that by taking webserver2 down too:



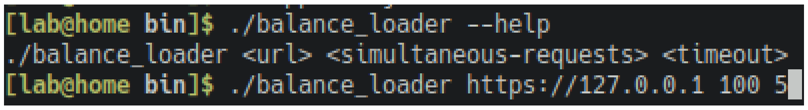
```
[lab@home simple]$ curl --insecure https://127.0.0.1
curl: (92) HTTP/2 stream 0 was not closed cleanly: CANCEL (err 8)
[lab@home simple]$ curl --insecure --http1.1 https://127.0.0.1
curl: (52) Empty reply from server
[lab@home simple]$ 
```

Figure 13: Verifying case when all backends are down

1.3.2. Load balancing

The most important part in a load balancer, is how well it can load balance traffic!

To be able to verify *noxy*'s load balancing capabilities, we'll rely on another program located in '*./bin/balance_loader*', it sends periodical asynchronous requests, and show backends who handled every request and shows them in a simple bar chart.

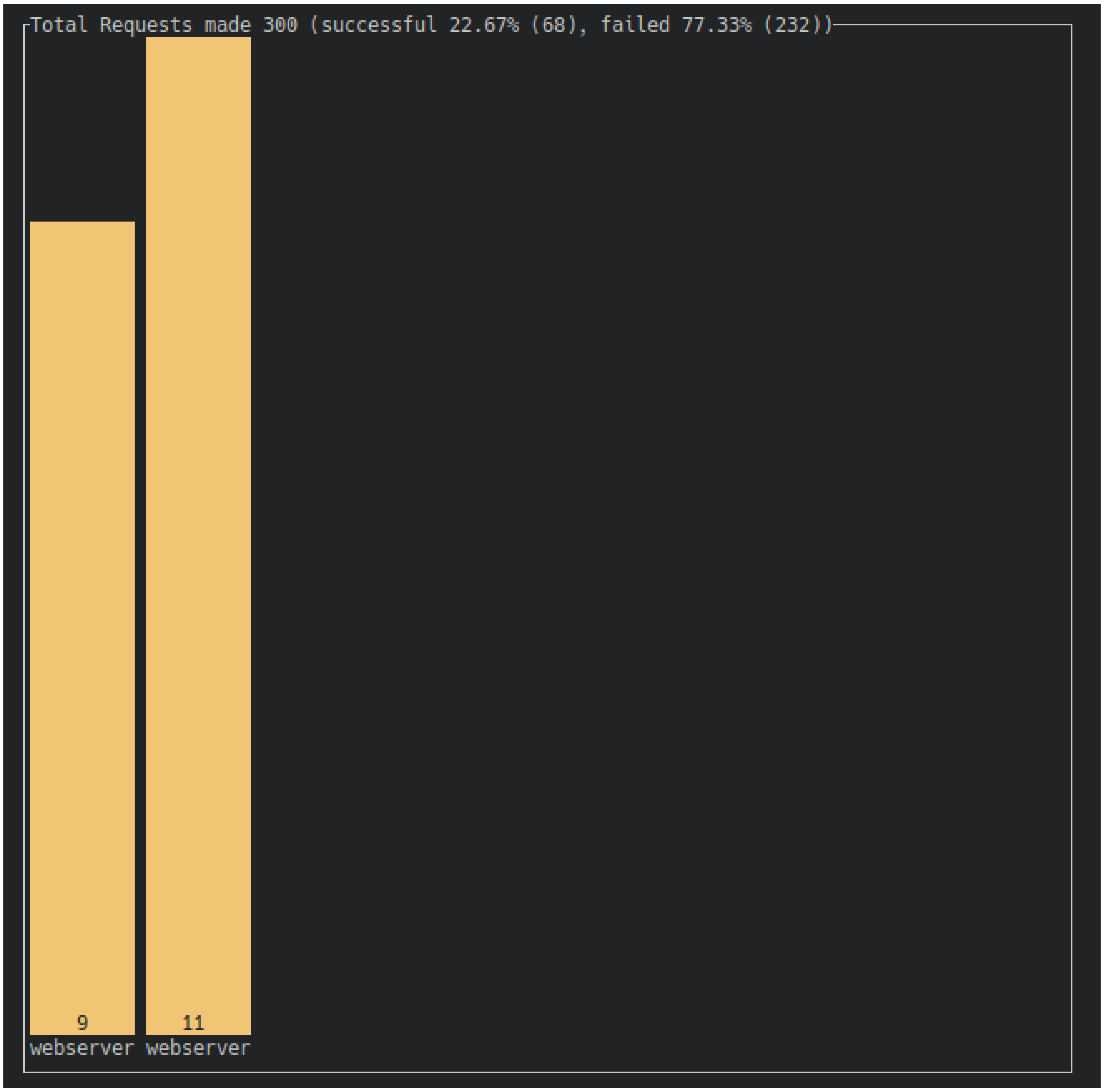


```
[lab@home bin]$ ./balance_loader --help
./balance_loader <url> <simultaneous-requests> <timeout>
[lab@home bin]$ ./balance_loader https://127.0.0.1 100 5
```

Figure 14: Assigning balance_loader options

To test *noxy*, we give it the URL it's listening on, how many requests we want to send, and delay between each attempt to send requests.

When we execute it with the given parameters we get the following bar chart that updates every time timeout:



```
Total Requests made 300 (successful 22.67% (68), failed 77.33% (232))
```

Figure 15: load_balancer bar chart with stats about requests sent to backends

We can see that it managed to detect both backend instances (*webserver1* and *webserver2*), total successful requests in that given moment was 20, it worth knowing that although we set max number of concurrent connections to 20, we may sometimes see a number higher than 20 in this tool, because although all requests are started at same time, not all of them connect to the server in same instant.

The total successful requests made was 68 out of 300, that is because we are sending a 100 request in a given moment, while *noxy* only accepts 20 concurrent connection according to our configuration.

2. Complex load balancing scenario

Now that our web application became bigger, we decided to use the micro-services architecture. Here is the scenario:

- We have two public web applications (website, api) that have different subdomains.
- Another service for administration on a different port, that does SSL termination and requires TLS user authentication (May not necessarily be a web application).
- All backends only host a single webapp.
- Redirect *http* to *https* website and api.

1.1. Setup

For this scenario, we have two webservers for the website, another two webservers for the api, and a telnet console (Only for demonstration purposes, use ssh for production). Most of the settings are like what we did in the simple scenario, except now, a separate network will host every application separately.

The following compose configuration was used:

```yaml
version: "3"
services:
  noxy:
    build:
      context: .
      dockerfile: NoxyDockerfile
    container_name: noxy
    ports:
      - "80:80"
      - "443:443"
      - "49383:49383"
    volumes:
      - type: bind
        source: ./complex.yaml
        target: /app/conf.yaml
      - type: bind
        source: ./ssl_cert/cert.pem
        target: /app/cert.pem
      - type: bind
        source: ./ssl_cert/key.pem
        target: /app/key.pem
      - type: bind
        source: ./ssl_cert/ca-cert.pem
        target: /app/ca-cert.pem
    environment:
      - "org.apache.logging.log4j.level=DEBUG"
    networks:
      - back_network
      - api_network
      - admin_network

  webserver1:
    image: docker.io/nginx
    container_name: webserver1
    volumes:
      - ./content_1/:/usr/share/nginx/html
    networks:
      - back_network

  webserver2:
    image: docker.io/nginx
    container_name: webserver2
    volumes:
      - ./content_2/:/usr/share/nginx/html
    networks:
      - back_network
```

Text 4: Docker compose configuration file for a complexe scenario (Part 1)

```
apiserver1:
    image: docker.io/nginx
    container_name: apiserver1
    volumes:
      - ./api_1/:/usr/share/nginx/html
      - type: bind
        source: nginx.conf
        target: /etc/nginx/conf.d/default.conf
    networks:
      - api_network

  apiserver2:
    image: docker.io/nginx
    container_name: apiserver2
    volumes:
      - ./api_2/:/usr/share/nginx/html
      - type: bind
        source: nginx.conf
        target: /etc/nginx/conf.d/default.conf
    networks:
     - api_network
telnetadmin:
    build:
      context: .
      dockerfile: TelnetDockerfile
    container_name: telnetadmin
    networks:
      - admin_network

networks:
  back_network: {}
  api_network: {}
  admin_network: {}
```

Text 5: Docker compose configuration file for a complexe scenario (Part 2)

The only big difference between this scenario and the previous, is that we isolated every service to a separate network, but *noxy*'s container is connected to all of them.

1.2. Configuration file

Noxy will have a two frontends:

- One that listens to web ports (80 and 443), for *https* port we will specify TLS certificate. And we will have a condition that redirects all requests from *http* to *https*.
- The second would be used for administration (let’s pick 49383 for this scenario), this only allows client with certificate authentication.

And there are three backends:

- Website: backends that host our website
- Api: backends that host our api
- Administration: the telnet console

This is all along with some other parameters we discussed before.

To achieve what was said before, the following configuration was used:

```
global:
  maxconn: 200
  threads: max # use all cpu threads availables
  health_check: 5000 # do a health check on backends every 5 seconds

defaults:
  timeout_connect: 10000 # 10 seconds in ms
  timeout_client: 30000 # 30 seconds in ms
  timeout_server: 30000
  maxconn: 200
  mode: http

frontend:
  - name: http_front
    bind: # all addr:port we want to listen on
      - addr_port: 0.0.0.0:80
      - addr_port: 0.0.0.0:443
        ssl: # we load a certificat for this particular addr:port
          cert: /app/cert.pem
          key: /app/key.pem
          versions: # tls versions allowed
            - TLSv1.3
          alpn: # http version negotiation accepted in tls handshake
            - h2
            - http/1.1
    use_backend: web_backend
    http_condition:
      - when: "scheme == 'http'" # redirect condition, activated when scheme is
http
        redirect: "sprintf('https://%s%s', host, uri)"
      - when: "host == 'api.noxy.rev'" # switch backend if api is wanted
        use_backend: api_backend

  - name: admin_front
    mode: tcp
    maxconn: 2 # don't allow more than 2 connections, only admin needs to connect
    timeout_client: 500000 # 500 seconds, due to type of telnet service
    timeout_server: 500000 # we will have long lived connection with some delays
between each input
    bind:
      - addr_port: 0.0.0.0:49383
        ssl:
          cert: /app/cert.pem
          key: /app/key.pem
          ca: /app/ca-cert.pem # certificate authority, used to do client
authentication
          versions: # tls versions allowed
            - TLSv1.3
    use_backend: admin_backend
```

Text 6: Noxy configuration file for a complexe scenario (Part 1)

```
backend:
  - name: web_backend
    balance: roundrobin
    servers: # list of backends, docker auto resolves domain to corresponding ip
address
      - addr_port: webserver1:80
      - addr_port: webserver2:80

  - name: api_backend
    balance: roundrobin
    servers:
      - addr_port: apiserver1:80
      - addr_port: apiserver2:80

  - name: admin_backend
    balance: roundrobin
    servers:
      - addr_port: telnetadmin:23
```

Text 7: Noxy configuration file for a complexe scenario (Part 2)

Noxy can have multiple conditions chained together, that's how we can have multiple condition like redirect and changing backends.

We also specified a certificate authority in *admin_front* so that only administrator with valid signed certificate can access telnet service.

1.3.1. Availability

Now we verify if all services configured are reachable and are load balanced.

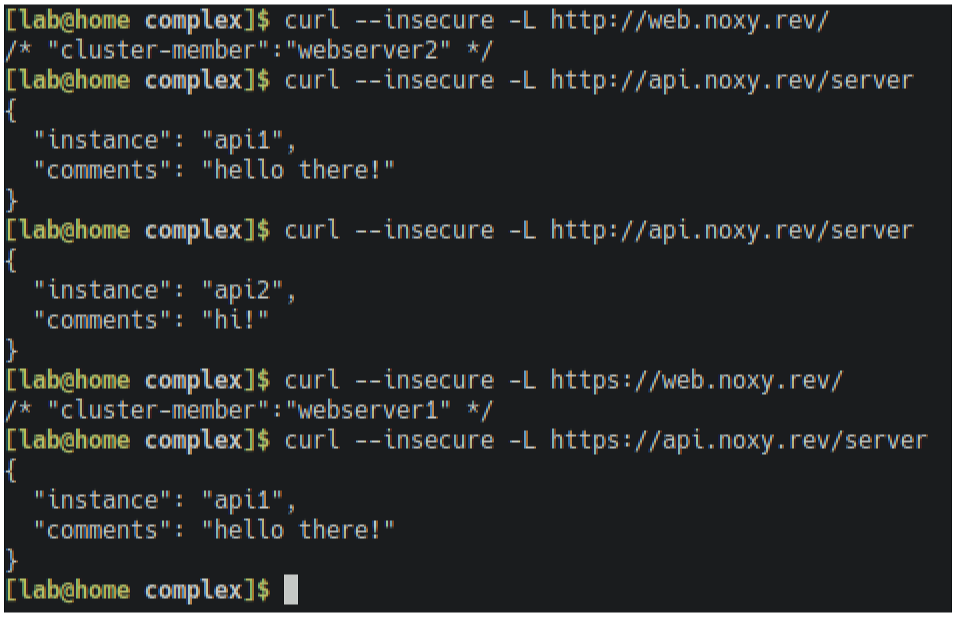
Before we do anything we need to configure services hostnames in host machines, so that we can use domains instead of ip addresses for web and api services. In '*/etc/hosts*' we append the following lines:

```
127.0.0.1       web.noxy.rev
127.0.0.1       api.noxy.rev
```

Text 8: assigning domain names to web and api ip address

This solution is only for testing purposes. A fully qualified DNS server should be used in production.

Now we can verify that each request is forwarded to wanted service:



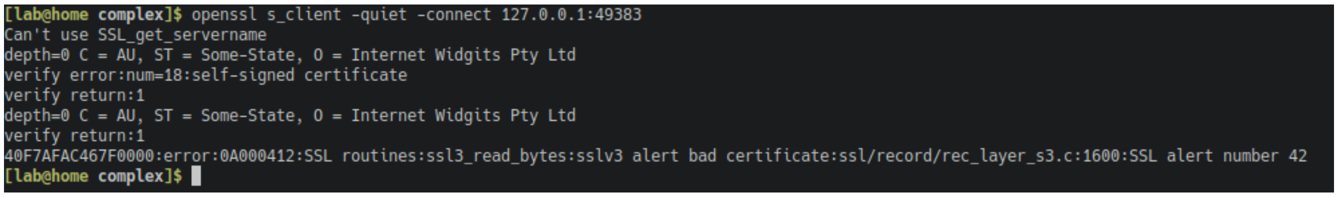
```
[lab@home complex]$ curl --insecure -L http://web.noxy.rev/
/* "cluster-member":"webserver2" */
[lab@home complex]$ curl --insecure -L http://api.noxy.rev/server
{
  "instance": "api1",
  "comments": "hello there!"
}
[lab@home complex]$ curl --insecure -L http://api.noxy.rev/server
{
  "instance": "api2",
  "comments": "hi!"
}
[lab@home complex]$ curl --insecure -L https://web.noxy.rev/
/* "cluster-member":"webserver1" */
[lab@home complex]$ curl --insecure -L https://api.noxy.rev/server
{
  "instance": "api1",
  "comments": "hello there!"
}
[lab@home complex]$ 
```

Figure 16: Verifying reachability of requests to all backends

We can verify that our requests are reaching both web and api backends, and that load balacing is indeed present.

Next is to verify that our administration telnet port is working.

We will first try without authentication:

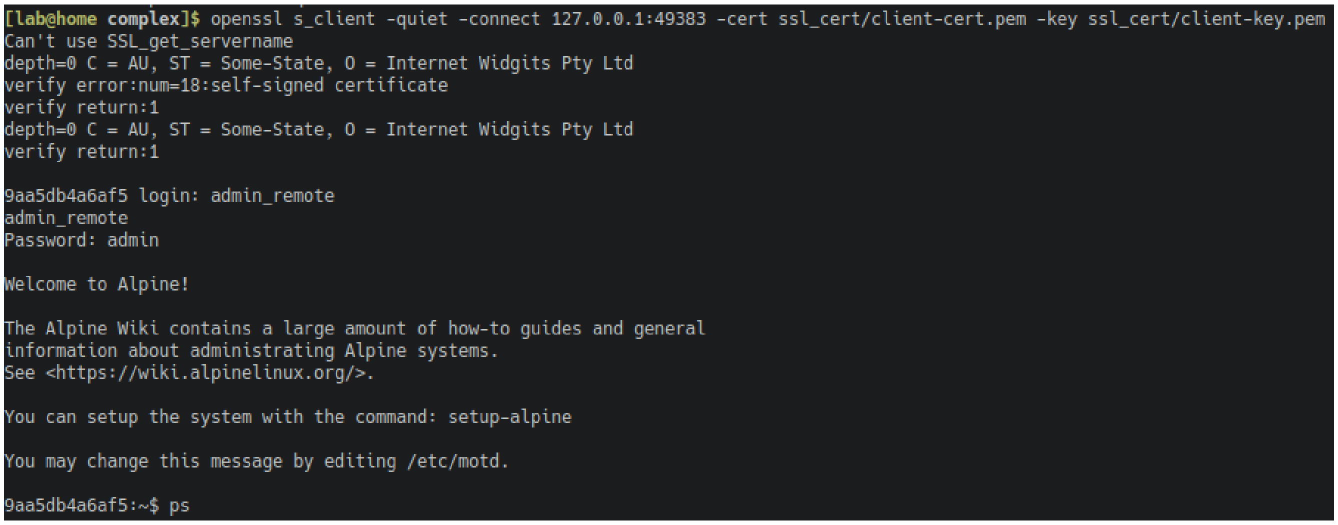


```
[lab@home complex]$ openssl s_client -quiet -connect 127.0.0.1:49383
Can't use SSL_get_servername
depth=0 C = AU, ST = Some-State, O = Internet Widgits Pty Ltd
verify error:num=18:self-signed certificate
verify return:1
depth=0 C = AU, ST = Some-State, O = Internet Widgits Pty Ltd
verify return:1
40F7AFAC467F0000:error:0A000412:SSL routines:ssl3_read_bytes:sslv3 alert bad certificate:ssl/record/rec_layer_s3.c:1600:SSL alert number 42
[lab@home complex]$ 
```

Figure 17: Openssl: Bad certificate

Oh! we get a complain that we have a bad certificate.

Let’s now try with a certificate signed by the certificate authority we gave *noxy* in the configuration:

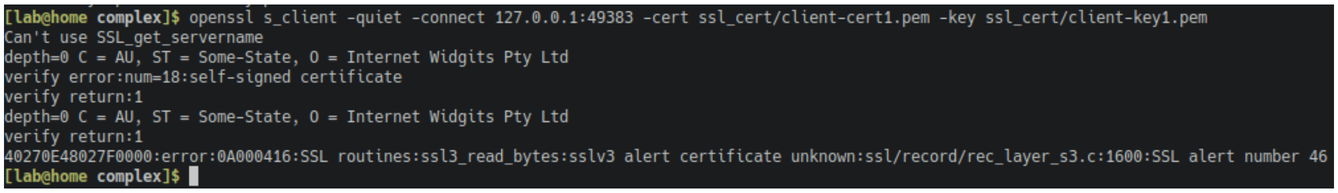


```
[lab@home complex]$ openssl s_client -quiet -connect 127.0.0.1:49383 -cert ssl_cert/client-cert.pem -key ssl_cert/client-key.pem
Can't use SSL_get_servername
depth=0 C = AU, ST = Some-State, O = Internet Widgits Pty Ltd
verify error:num=18:self-signed certificate
verify return:1
depth=0 C = AU, ST = Some-State, O = Internet Widgits Pty Ltd
verify return:1

9aa5db4a6af5 login: admin_remote
admin_remote
Password: admin

Welcome to Alpine!

The Alpine Wiki contains a large amount of how-to guides and general
information about administrating Alpine systems.
See <https://wiki.alpinelinux.org/>.

You can setup the system with the command: setup-alpine

You may change this message by editing /etc/motd.

9aa5db4a6af5:~$ ps
```

Figure 18: Openssl: Good certificate signed bad certificate authority

Now, we get the telnet prompt asking for login credentials, and we can log in like normal!

Let’s try again but now with a certificate not signed by the certificate authority we use:



```
[lab@home complex]$ openssl s_client -quiet -connect 127.0.0.1:49383 -cert ssl_cert/client-cert1.pem -key ssl_cert/client-key1.pem
Can't use SSL_get_servername
depth=0 C = AU, ST = Some-State, O = Internet Widgits Pty Ltd
verify error:num=18:self-signed certificate
verify return:1
depth=0 C = AU, ST = Some-State, O = Internet Widgits Pty Ltd
verify return:1
40270E48027F0000:error:0A000416:SSL routines:ssl3_read_bytes:sslv3 alert certificate unknown:ssl/record/rec_layer_s3.c:1600:SSL alert number 46
[lab@home complex]$ 
```

Figure 19: Openssl: Bad certificate (2)

So, that means no matter how much you try, you can’t log in without a signed certificate. This is remedy for legacy devices that does not support ssh, not a solution that should be ever used!

IV. Conclusion

This was it for load balancing! While we mainly focused on how server side load balancing works, by creating our own load balancer, it is worth noting that other techniques on the server side can be used, such as clustering, or load balancer redundancy. And for the other part we omitted in this report, client side load balancing can decrease traffic on a specific cluster by sending clients to different clusters, one of them is DNS load balancing.

When it comes to load balancing, having a mix of client side and server side balancing, will have a greater impact on reducing load on a single server or cluster, and increase throughput while lowering latency.