

Configuração e Customização HAPROXY

- [2 Ways to Enable Sticky Sessions in HAProxy \(Guide\)](#)
- [HAPROXY com persistência de URL](#)
- [Building a High Availability Cluster with HAProxy, Keepalived, and Docker: A Step-by-Step Guide](#)
- [Configuração de URL Path no HAPROXY](#)
- [Configuração de logs no HAPROXY](#)
- [Reconfiguração automática do haproxy.cfg do HAPROXY](#)
- [Reconfiguração automática do haproxy.cfg do HAPROXY - 2](#)

2 Ways to Enable Sticky Sessions in HAProxy (Guide)

Link: <https://www.haproxy.com/blog/enable-sticky-sessions-in-haproxy>

HyperText Transfer Protocol (HTTP), the protocol that defines the language browsers use to communicate with web servers, is stateless, meaning that after you make a web request and a server sends back a response, no memory of that interaction remains. Websites need other ways to remember previous user interactions to make anything more sophisticated than a static web page work.

These days, Javascript frameworks like Vue.js, React.js, and others let developers create single-page applications (SPAs) that provide statefulness to the otherwise stateless web. Because they're implemented as Javascript applications running in the user's browser, they can keep track of what the user has done and render the app in a way that accounts for that shared history. However, the user's browser

Nick creates technical content for HAProxy Technologies ranging from documentation and blog posts to Wikipedia articles, GitHub READMEs and Stack Overflow answers. With a background in web development and DevOps, he has fun digging into product features and discovering the optimal path for a new blog tutorial.

[Twitter](#) [LinkedIn](#) [GitHub](#)

Related Posts

[How to Enable Health Checks in HAProxy \(Guide\)](#)

September 14th, 2021

[How to Enable Health Checks in HAProxy \(Guide\)](#)

HAProxy provides three types of health checks: active health checks, passive health checks and agent health checks. In this blog, we will learn about each one.

[2 Ways to Implement a Circuit Breaker in HAProxy \(Guide\)](#)

January 5th, 2021

[2 Ways to Implement a Circuit Breaker in HAProxy \(Guide\)](#)

In this blog post, you'll learn how to implement a circuit breaker with HAProxy in a simple and more complex way, that allows greater customization.

[Load Balancing, Affinity, Persistence & Sticky Sessions](#)

March 29th, 2012

[Load Balancing, Affinity, Persistence & Sticky Sessions](#)

This blog post shows why and how to use a load balancer, the differences between Affinity, Persistence, and Sticky Sessions, and how to enable them in HAProxy.

[Preserve Source IP Address Despite Reverse Proxies \(Guide\)](#)

June 5th, 2012

[Preserve Source IP Address Despite Reverse Proxies \(Guide\)](#)

In this blog post, we aim to demonstrate how to preserve source IP addresses when using the reverse-proxy mode in HAProxy.

HAPROXY com persistência de URL

Informações geradas pelo ChatGPT.

Sim, é possível configurar o HAProxy para atender tanto **HTTP** (porta 80) quanto **HTTPS** (porta 443) ao mesmo tempo. Isso pode ser feito configurando dois **frontends separados** — um para HTTP e outro para HTTPS — e ambos encaminhando as requisições para o mesmo backend, de modo que o balanceamento de carga e a persistência por URL sejam aplicados de maneira uniforme para ambos os protocolos.

Exemplo de Configuração para HTTP e HTTPS ao Mesmo Tempo

Aqui está um exemplo de configuração onde o HAProxy aceita tráfego HTTP e HTTPS e roteia as requisições para o mesmo backend, mantendo a persistência baseada na URL para ambos os protocolos:

```
haproxy
```

```
global
    log /dev/log local0
    log /dev/log local1
    notice
    maxconn 2000
```

```
defaults
    log global
    option httplog
    option dontlognull
    timeout connect 5000ms
    timeout client 50000ms
    timeout server 50000ms
```

```
# Frontend HTTP (porta 80)
frontend http_front
    bind *:80
    acl is_http_request path_reg ^/.* # Captura qualquer URL
    use_backend default_backend if is_http_request
```

```
# Frontend HTTPS (porta 443)
frontend https_front
```

```
bind *:443 ssl crt /etc/ssl/certs/meucertificado.pem # Certificado SSL
```

```
acl is_https_request path_reg ^/.* # Captura qualquer URL
```

```
use_backend default_backend if is_https_request
```

```
# Backend
```

```
backend default_backend
```

```
balance roundrobin
```

```
stick-table type string size 200k expire 30m store gpc0
```

```
stick on url_param(url) # Persistência baseada na URL
```

```
cookie SERVERID insert indirect nocache
```

```
server server1 192.168.1.1:80 check # Servidor HTTP
```

```
server server2 192.168.1.2:80 check # Servidor HTTP
```

```
server server3 192.168.1.3:80 check # Servidor HTTP
```

Explicação da Configuração:

1. Frontend HTTP:

- `bind *:80`: O frontend escuta na porta 80 para tráfego HTTP.
- `acl is_http_request path_reg ^/.*`: A ACL captura qualquer URL para redirecioná-la para o backend.
- `use_backend default_backend if is_http_request`: Se a requisição for HTTP, ela é direcionada para o `default_backend`.

2. Frontend HTTPS:

- `bind *:443 ssl crt /etc/ssl/certs/meucertificado.pem`: Este frontend escuta na porta 443 para tráfego HTTPS e usa o certificado SSL fornecido no arquivo `/etc/ssl/certs/meucertificado.pem`.
- `acl is_https_request path_reg ^/.*`: A ACL captura qualquer URL, assim como no frontend HTTP.
- `use_backend default_backend if is_https_request`: Se a requisição for HTTPS, ela é direcionada para o `default_backend`.

3. Backend:

- O **backend** é compartilhado entre ambos os frontends HTTP e HTTPS, com **persistência baseada na URL** usando o `stick-table` para garantir que as requisições para a mesma URL sejam sempre roteadas para o **mesmo servidor**.
- `cookie SERVERID insert indirect nocache`: Garante a persistência usando cookies.
- Os servidores backend continuam na porta **80** (HTTP), porque o HAProxy está fazendo a **terminação SSL**.

Como Funciona:

1. Requisição HTTP:

- O cliente acessa a URL usando `http://meusite.com/paginal`.

- O HAProxy recebe a requisição na porta **80** e a encaminha para um dos servidores backend (`server1`, `server2`, `server3`), com base no balanceamento de carga `roundrobin`.
- O servidor backend responde, e o HAProxy registra a URL `/pagina1` e o servidor que a atendeu.

2. Requisição HTTPS:

- O cliente acessa a URL usando `https://meusite.com/pagina1`.
- O HAProxy recebe a requisição na porta **443**, faz a descryptografia SSL e encaminha para o mesmo backend (porta 80), utilizando o mesmo servidor que atendeu a primeira requisição HTTP para aquela URL.
- O HAProxy verifica a tabela de stickiness e direciona a requisição para o **mesmo servidor** que atendeu a primeira requisição, mantendo a persistência.

3. Persistência para Todos os Usuários:

- A persistência é mantida independentemente de ser uma requisição HTTP ou HTTPS, porque a persistência está baseada na URL (`stick on url_param(url)`), não em cookies ou IP do cliente.
- Todos os usuários que acessarem a mesma URL (seja por HTTP ou HTTPS) serão direcionados para o **mesmo servidor backend** que atendeu a primeira requisição para aquela URL.

Certificado SSL:

O caminho `/etc/ssl/certs/meucertificado.pem` deve apontar para o **certificado SSL** e a **chave privada** do seu domínio. Se você estiver usando o Let's Encrypt ou outra autoridade certificadora (CA), basta colocar o arquivo `.pem` correto que contenha o **certificado público** e a **chave privada**.

Caso você precise gerar um certificado SSL autoassinado, pode usar o seguinte comando `openssl`:

bash

```
openssl req -x509 -newkey rsa:2048 -keyout /etc/ssl/private/meuchave.key -out /etc/ssl/certs/meucertificado.pem -days 365
```

Esse comando gera um certificado **autoassinado**, o que é útil para testes ou ambientes internos, mas para produção, recomenda-se o uso de um certificado de uma autoridade certificadora (CA) confiável.

Conclusão:

- **Sim, o HAProxy pode atender tanto HTTP quanto HTTPS ao mesmo tempo**, com a mesma configuração de persistência por URL para ambos os protocolos.

- A configuração utiliza dois **frontends** separados (um para HTTP e outro para HTTPS), mas ambos encaminham as requisições para o **mesmo backend**, garantindo que a persistência de URL seja aplicada de forma consistente, independentemente do protocolo.
- A **terminação SSL** acontece no HAProxy, o que significa que ele descriptografa as requisições HTTPS e as envia como HTTP para os servidores backend.

Se você deseja configurar o HAProxy para rotear tráfego em **portas diferentes de 80 (HTTP) e 443 (HTTPS)**, o procedimento é semelhante, mas você deve ajustar a configuração de **frontends** e **backends** para escutar nas portas específicas que deseja utilizar.

A lógica de balanceamento de carga e persistência por URL será a mesma, mas em vez de usar as portas padrão (80 e 443), você configurará o HAProxy para escutar nas portas que você escolher.

Exemplo de Configuração para Portas Diferentes:

Vamos imaginar que você deseja usar as portas **8080** para HTTP e **8443** para HTTPS, em vez das portas padrão (80 e 443). A configuração seria:

```
global
    log /dev/log local0
    log /dev/log local1 notice
    maxconn 2000

defaults
    log global
    option httplog
    option dontlognull
    timeout connect 5000ms
    timeout client 50000ms
    timeout server 50000ms

# Frontend para HTTP (porta 8080)
frontend http_front
    bind *:8080 # Escutando na porta 8080
    acl is_http_request path_reg ^/.* # Captura qualquer URL
    use_backend default_backend if is_http_request

# Frontend para HTTPS (porta 8443)
frontend https_front
    bind *:8443 ssl crt /etc/ssl/certs/meucertificado.pem # Certificado SSL
    acl is_https_request path_reg ^/.* # Captura qualquer URL
    use_backend default_backend if is_https_request
```

```
# Backend
backend default_backend
    balance roundrobin
    stick-table type string size 200k expire 30m store gpc0
    stick on url_param(url) # Persistência baseada na URL
    cookie SERVERID insert indirect nocache
    server server1 192.168.1.1:80 check # Servidor HTTP
    server server2 192.168.1.2:80 check # Servidor HTTP
    server server3 192.168.1.3:80 check # Servidor HTTP
```

Explicação das Modificações:

1. Frontend HTTP:

- `bind *:8080`: O HAProxy agora escuta na porta `8080` para tráfego HTTP, em vez de `80`.
- `acl is_http_request path_reg ^/.*`: A regra ACL captura qualquer URL que for solicitada e a redireciona para o backend.

2. Frontend HTTPS:

- `bind *:8443 ssl crt /etc/ssl/certs/meucertificado.pem`: O HAProxy escuta na porta `8443` para tráfego HTTPS e usa o certificado SSL localizado no caminho `/etc/ssl/certs/meucertificado.pem`. O certificado SSL deve ser válido, seja autoassinado ou de uma autoridade certificadora.

3. Backend:

- A configuração do **backend** permanece a mesma, com balanceamento de carga baseado em `roundrobin`, persistência por URL usando `stick-table`, e a inserção do cookie `SERVERID` para garantir a persistência.

4. Servidores Backend:

- `server server1 192.168.1.1:80 check`: Os servidores backend continuam sendo configurados na porta `80`, porque o HAProxy irá fazer a **terminação SSL** para o tráfego HTTPS e, em seguida, encaminhá-lo para os servidores backend na porta `80`.

Fluxo de Funcionamento:

1. Requisição HTTP (porta 8080):

- O cliente acessa a URL `http://meusite.com:8080/pagina1`.
- O HAProxy recebe a requisição na porta `8080`, e com base no balanceamento de carga (`roundrobin`), encaminha a requisição para um dos servidores backend.
- O servidor que responder à primeira requisição para `/pagina1` será registrado no **stick-table** com a URL `/pagina1`, garantindo que futuras requisições para essa URL sejam roteadas para o **mesmo servidor**.

2. Requisição HTTPS (porta 8443):

- O cliente acessa a URL `https://meusite.com:8443/pagina1`.
- O HAProxy recebe a requisição na porta `8443`, faz a **terminação SSL** (descriptografa o tráfego HTTPS) e, em seguida, redireciona a requisição para os servidores backend na porta `80`.
- O HAProxy verifica na **stick-table** qual servidor foi associado à URL `/pagina1` e direciona a requisição para o **mesmo servidor** que atendeu a primeira requisição.

3. Persistência para Todos os Usuários:

- Independentemente de ser uma requisição HTTP ou HTTPS, a persistência será mantida para todos os usuários com base na URL, garantindo que a URL `/pagina1` seja sempre tratada pelo **mesmo servidor**.

Caso Você Queira Usar Portas Diferentes para Cada Backend:

Se você quiser que o tráfego seja roteado para diferentes portas nos servidores backend, você pode ajustar a configuração para cada servidor backend. Por exemplo, se você tiver servidores que escutam em portas diferentes, a configuração seria algo como:

```
backend default_backend
    balance roundrobin
    stick-table type string size 200k expire 30m store gpc0
    stick on url_param(url)
    cookie SERVERID insert indirect nocache
    server server1 192.168.1.1:8081 check # Servidor backend escutando na porta 8081
    server server2 192.168.1.2:8082 check # Servidor backend escutando na porta 8082
    server server3 192.168.1.3:8083 check # Servidor backend escutando na porta 8083
```

Essa configuração faria com que cada servidor backend estivesse escutando em portas diferentes, mas o HAProxy ainda aplicaria a persistência e o balanceamento de carga.

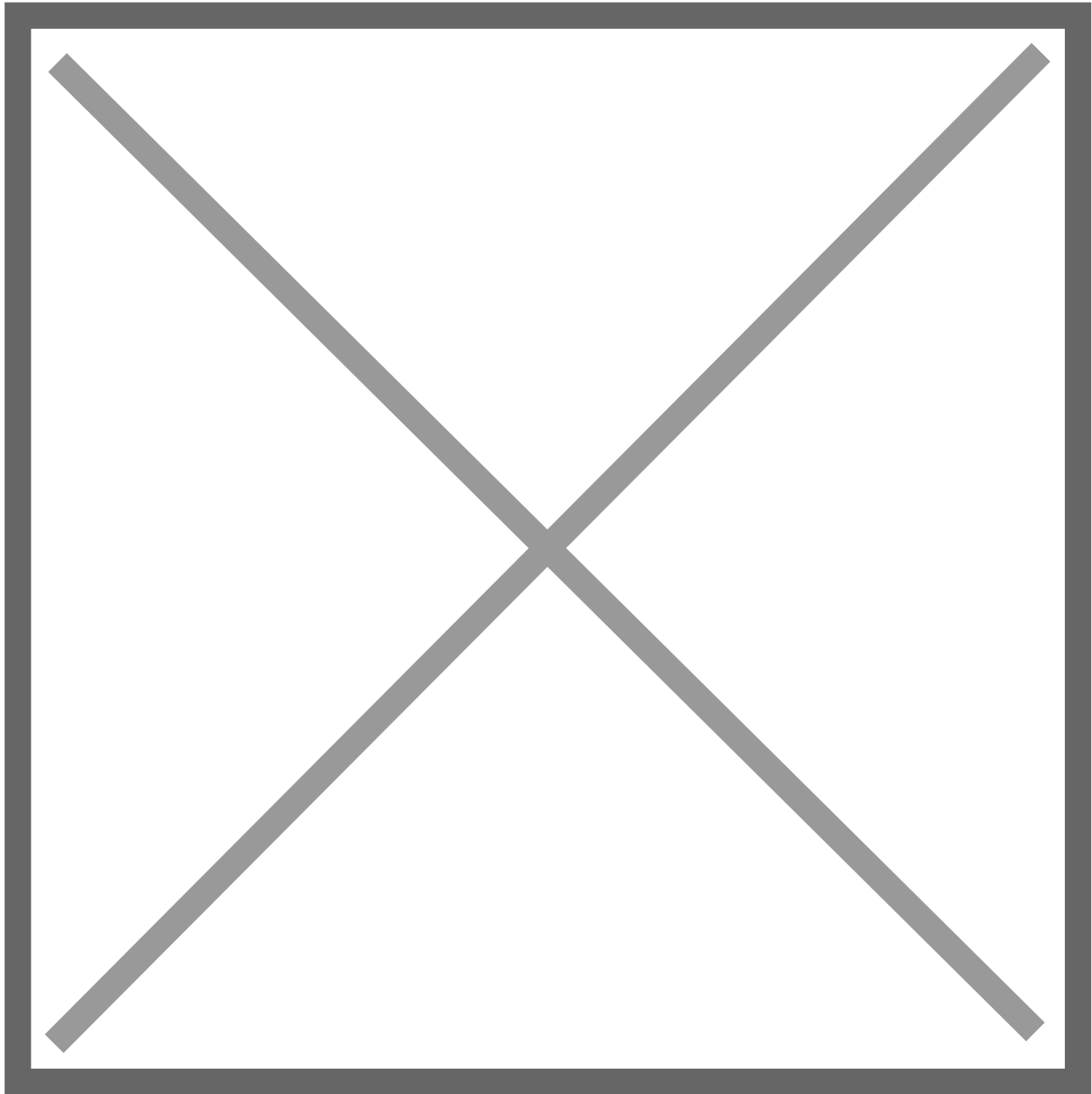
Resumo:

- **Sim, é possível configurar o HAProxy para escutar em portas diferentes de 80 e 443** (por exemplo, 8080 para HTTP e 8443 para HTTPS).
- A persistência de URL e o balanceamento de carga continuam funcionando da mesma forma, **independentemente da porta** que está sendo utilizada.
- A configuração de **terminação SSL** no HAProxy permite que o tráfego HTTPS seja manipulado na porta `8443` e redirecionado para os servidores backend na porta `80` (ou qualquer porta configurada).

Building a High Availability Cluster with HAProxy, Keepalived, and Docker: A Step-by-Step Guide

Link: <https://medium.com/@yahyasghiouri1998/building-a-high-availability-cluster-with-haproxy-keepalived-and-docker-a-step-by-step-guide-9325f4ac8aa7>

·Aug 30, 2024



High availability

High availability (HA) is essential for ensuring that web applications remain accessible, even in the face of hardware or software failures. An HA architecture distributes traffic across multiple servers, preventing any single point of failure from disrupting service. In this article, we'll explore how to build a high availability cluster using HAProxy and Keepalived, two powerful tools for load balancing and failover.

To enhance flexibility and simplify deployment, we'll leverage Docker to containerize HAProxy, Keepalived, and our web applications. Docker allows us to create a portable and easily manageable HA setup that can be deployed across different environments, whether on-premises or in the cloud.

Throughout this guide, we'll walk you through the entire process — from setting up Docker networks and building Dockerfiles to configuring HAProxy and Keepalived for seamless failover. By the end, you'll have a fully functional HA cluster that ensures your web applications are always available, all within a Dockerized environment.

I-/ General Concepts

Before diving into the different configurations, it's helpful to understand the core components of the architecture.

I-1/HAProxy

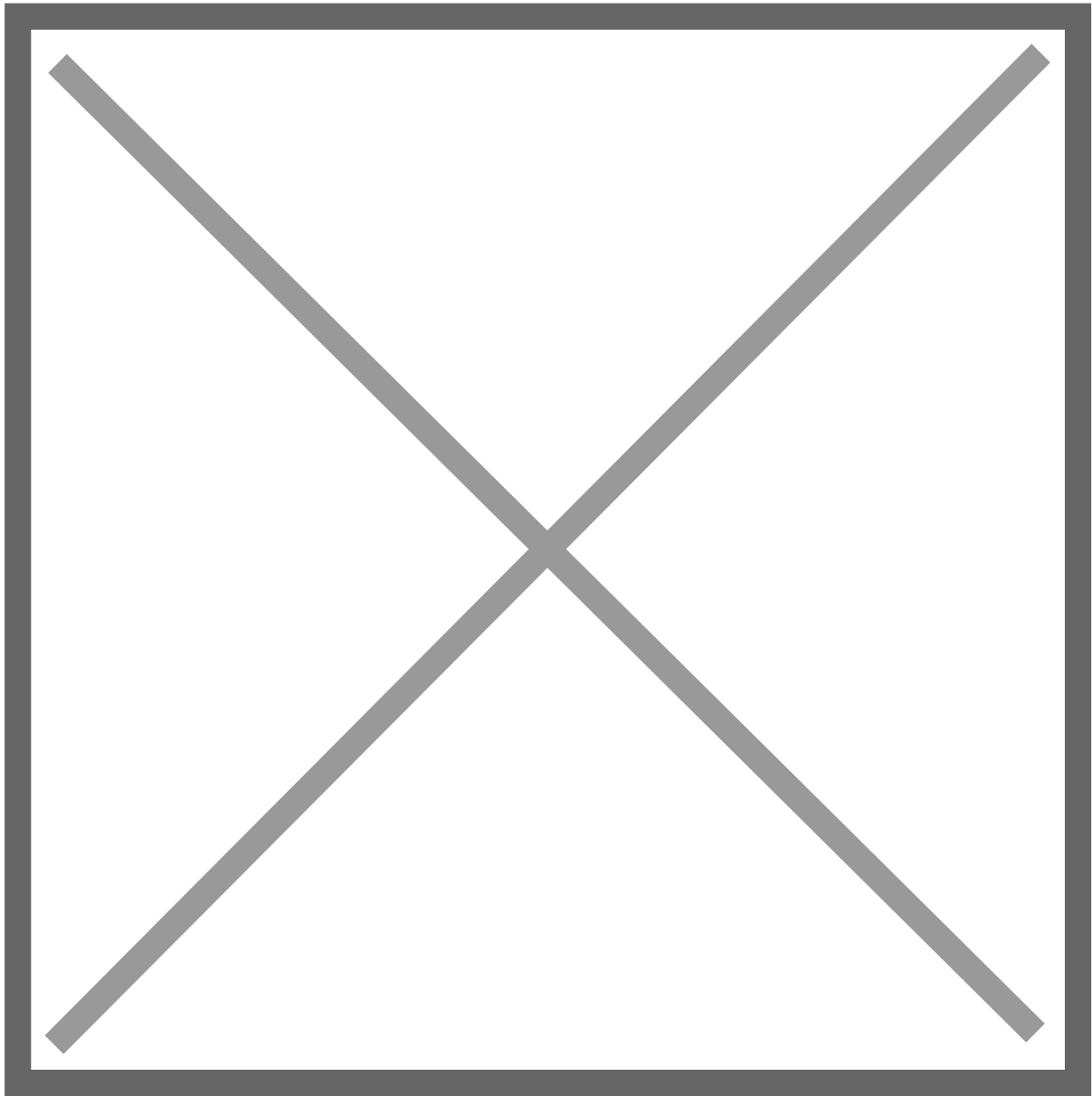
HAProxy, as its name suggests, stands for High Availability Proxy. It is a robust and versatile tool designed to provide high availability and efficient load balancing for network traffic.

HAProxy is a widely used tool for distributing incoming requests across multiple backend servers to enhance both reliability and performance. It performs continuous health checks on these servers to ensure that traffic is routed only to those that are healthy and responsive.

The tool employs various algorithms, such as round-robin and least connections, to effectively balance the load. Supporting both TCP and HTTP traffic, HAProxy operates at Layer 4 (the transport layer) and Layer 7 (the application layer) of the OSI model.

There are two main components when configuring HAProxy, a frontend and a backend section.

- **Frontend section:** It is the entry point for incoming client requests. It defines how HAProxy listens for incoming traffic and how it should handle these requests, here we specify the address and port on which HAProxy should listen, as well as any rules or conditions for routing the traffic to the appropriate backend by inspecting the incoming packets.
- **Backend section:** it represents the servers that will handle the requests forwarded by the frontend. we define how HAProxy should route traffic to the backend servers and how it should manage these servers basically we specify also load balancing algorithms, and health checks. In summary we control how requests will be distributed among the servers and how we are going to handle server failures or maintenance.



HA proxy load balancer

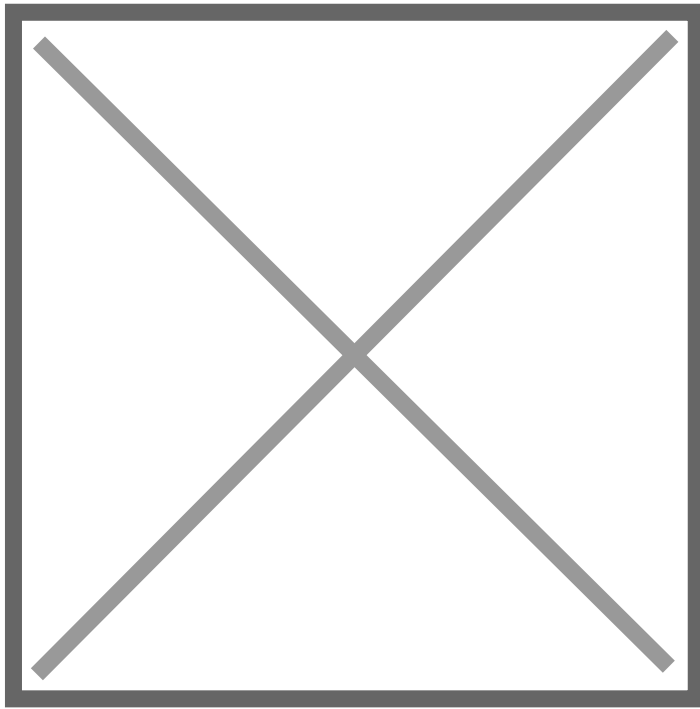
I-2/Keepalived & VRRP

VRRP (Virtual Router Redundancy Protocol) is designed to create a virtual router that represents a group of physical routers, allowing them to work together to present a single virtual IP address (VIP) to the network. This VIP is used as the default gateway by clients.

In a VRRP setup, one router is elected as the master. The master router handles traffic directed to the VIP, while the other routers in the group act as backups and monitor the master router's health. If the master router fails, one of the backup routers takes over as the new master, ensuring the continuity of service.

Keepalived is a widely used implementation of VRRP with additional features. It assigns a priority to each node in the group, and based on these priorities, it elects a new master if a failure occurs.

Keepalived enhances VRRP with advanced health checks and failover capabilities, making it ideal for high-availability setups.



Virtual IP assignment

II-/ Deployment architecture

After understanding the foundational concepts of HAProxy and Keepalived, it's crucial to see how these components come together to form a high availability cluster.

Deployment Architecture
Deployment architecture

The architecture I've implemented leverages Docker to create a resilient and scalable environment, ensuring continuous service availability. The visual representation above illustrates how traffic is routed through HAProxy instances and managed by Keepalived to provide redundancy and failover capabilities.

To interconnect all components, I set up a Docker bridge network, which ensures seamless communication between the HAProxy instances, Keepalived, and the backend servers. This network allows the HAProxy instances to effectively distribute incoming traffic across multiple backend servers while monitoring their health and performance.

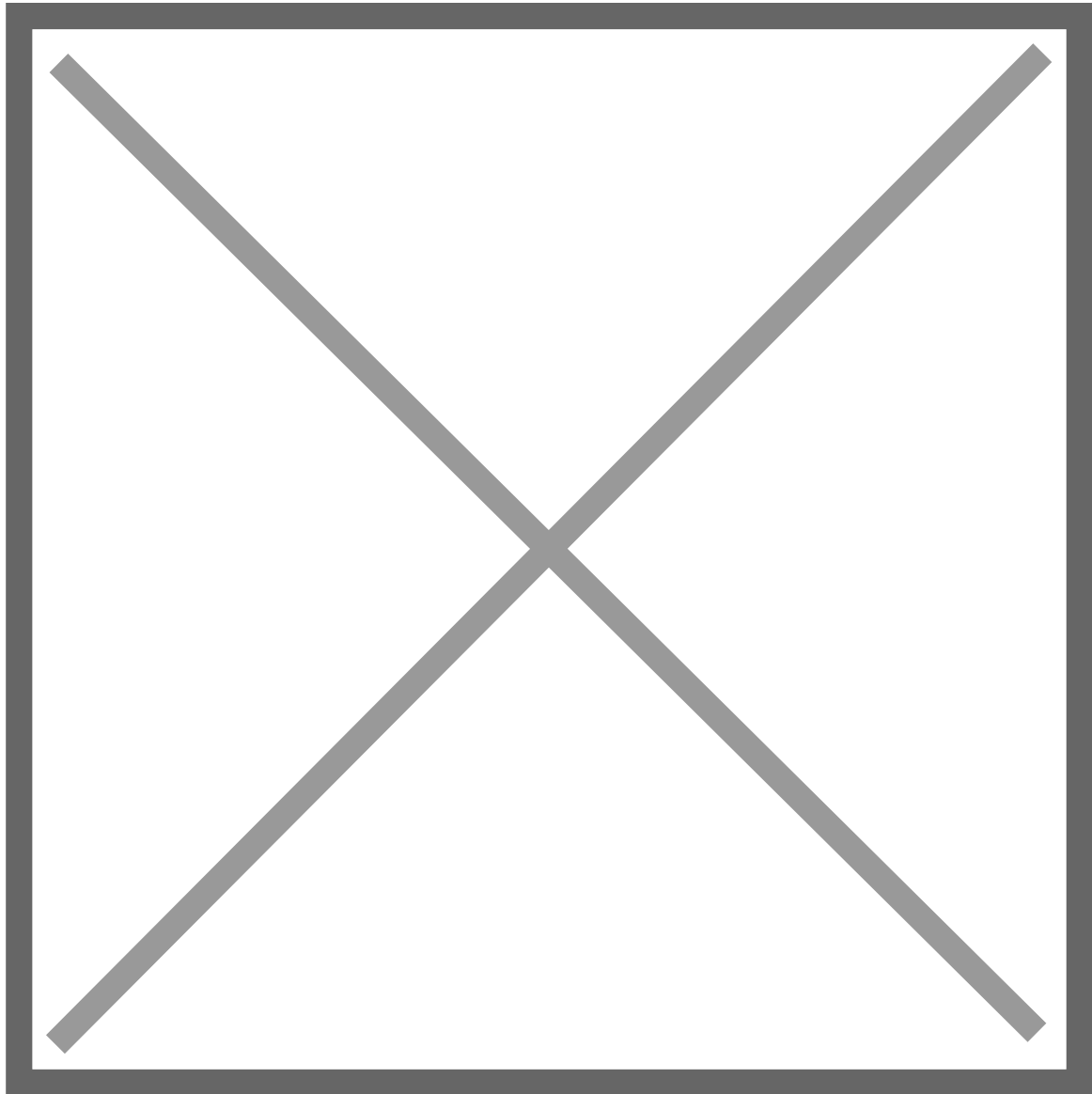
In this setup, there is a primary HAProxy instance (the master) and a secondary instance (the backup) ready to take over if the master fails. Keepalived, installed on both machines, manages the virtual IP (VIP) that clients connect to. This VIP ensures that, even in the event of a failure,

traffic is automatically redirected to the backup HAProxy instance, maintaining service availability without interruption.

The backend comprises three cloned instances of a server running a simple Flask application that serves static content. This setup is an example of a stateless application deployment, where each instance operates independently without relying on session persistence or shared state. In the case of stateful applications, additional architectural considerations would be necessary, such as implementing shared storage, session replication, sticky sessions, or database clustering to ensure consistency and availability. Following best practices in system design is crucial to address these challenges and optimize the architecture based on the application's specific requirements and context.

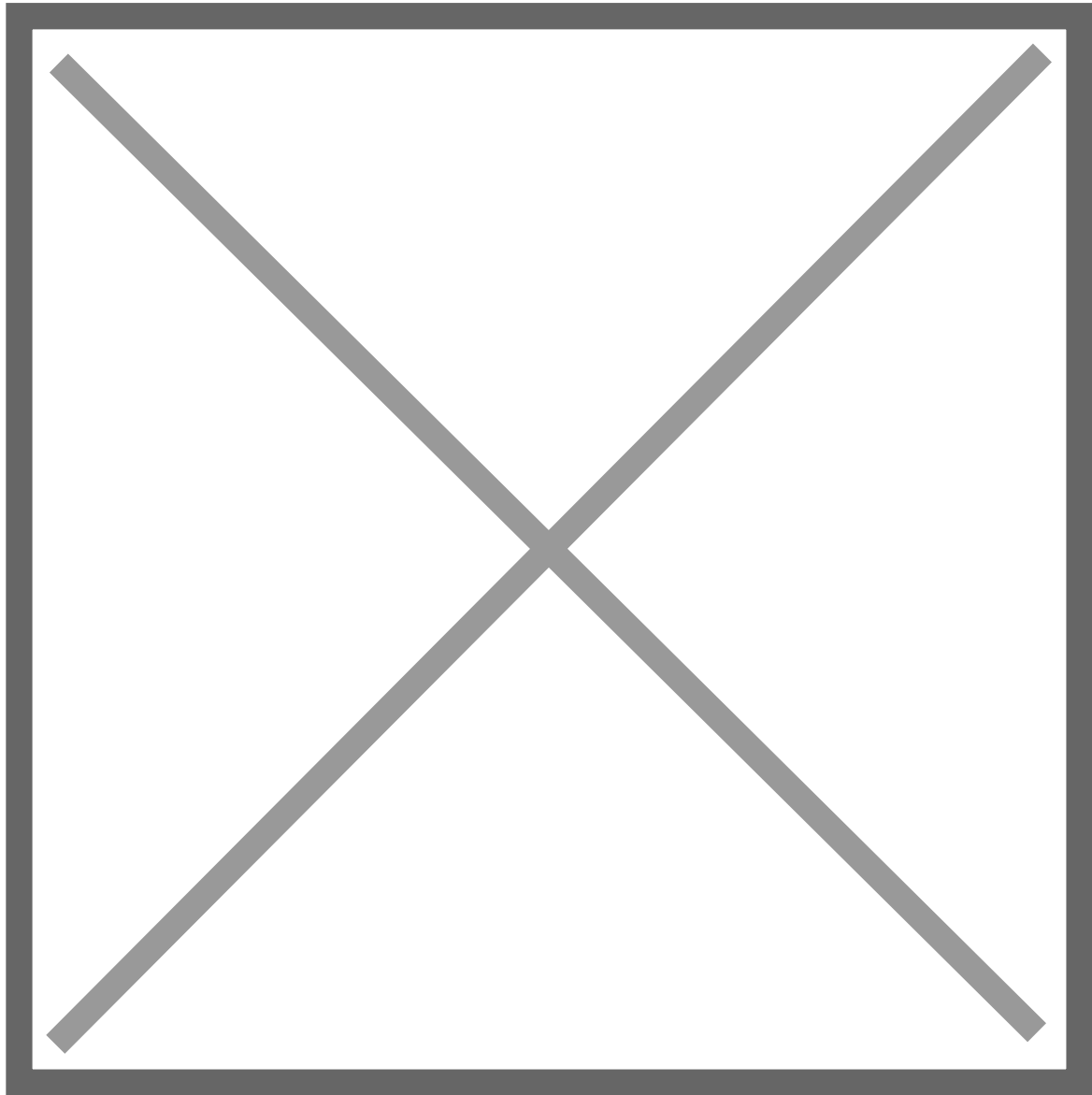
III-/Step by step guide

- The first thing we're going to do is create our stateless app a simple Python application that doesn't store any session information. To get started, we need to create a virtual environment, so make sure you have Python installed on your machine.



Virtual environment creation

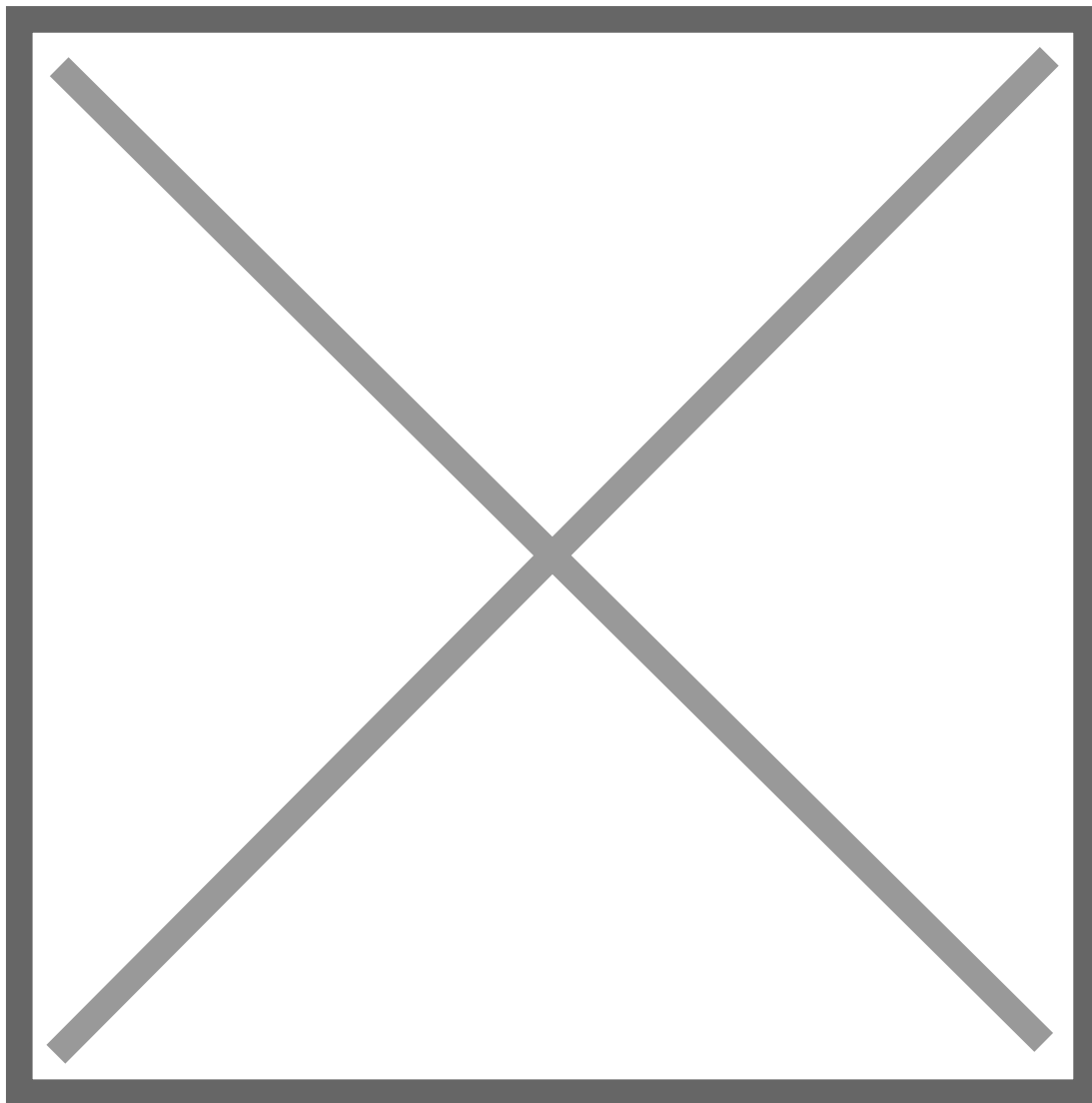
- Activate the virtual environment, and then install Flask.



Installing Flask

- Now Flask is installed in our virtual environment, we are going to create a simple Flask App with `hello world!` content, for me I'll use nano editor, you can use whatever editor you want for that purpose.

```
nano app.py
```



Flask webapp

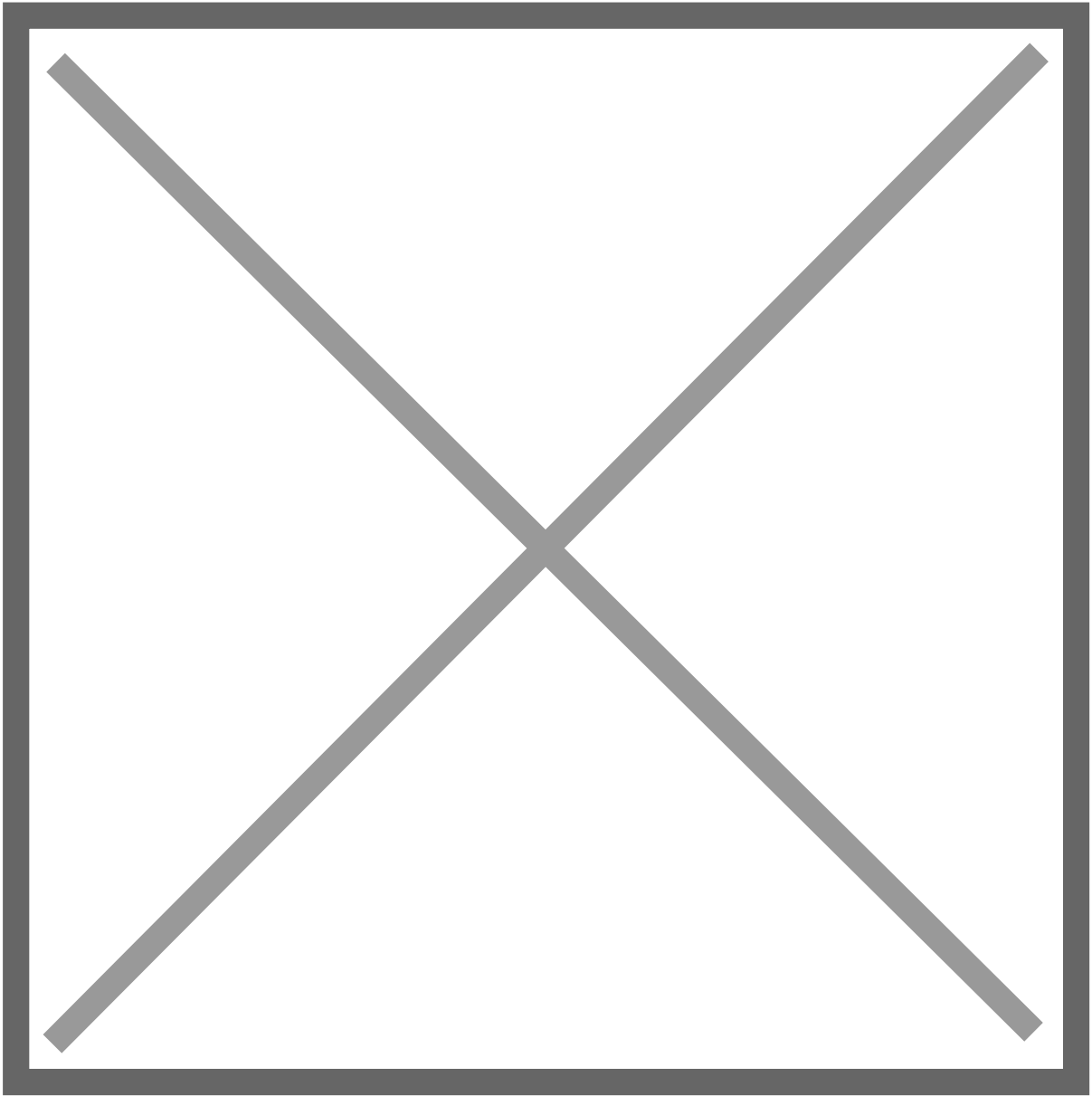
Copy paste the following content or create your own:

```
from flask import Flask
app = Flask(__name__)

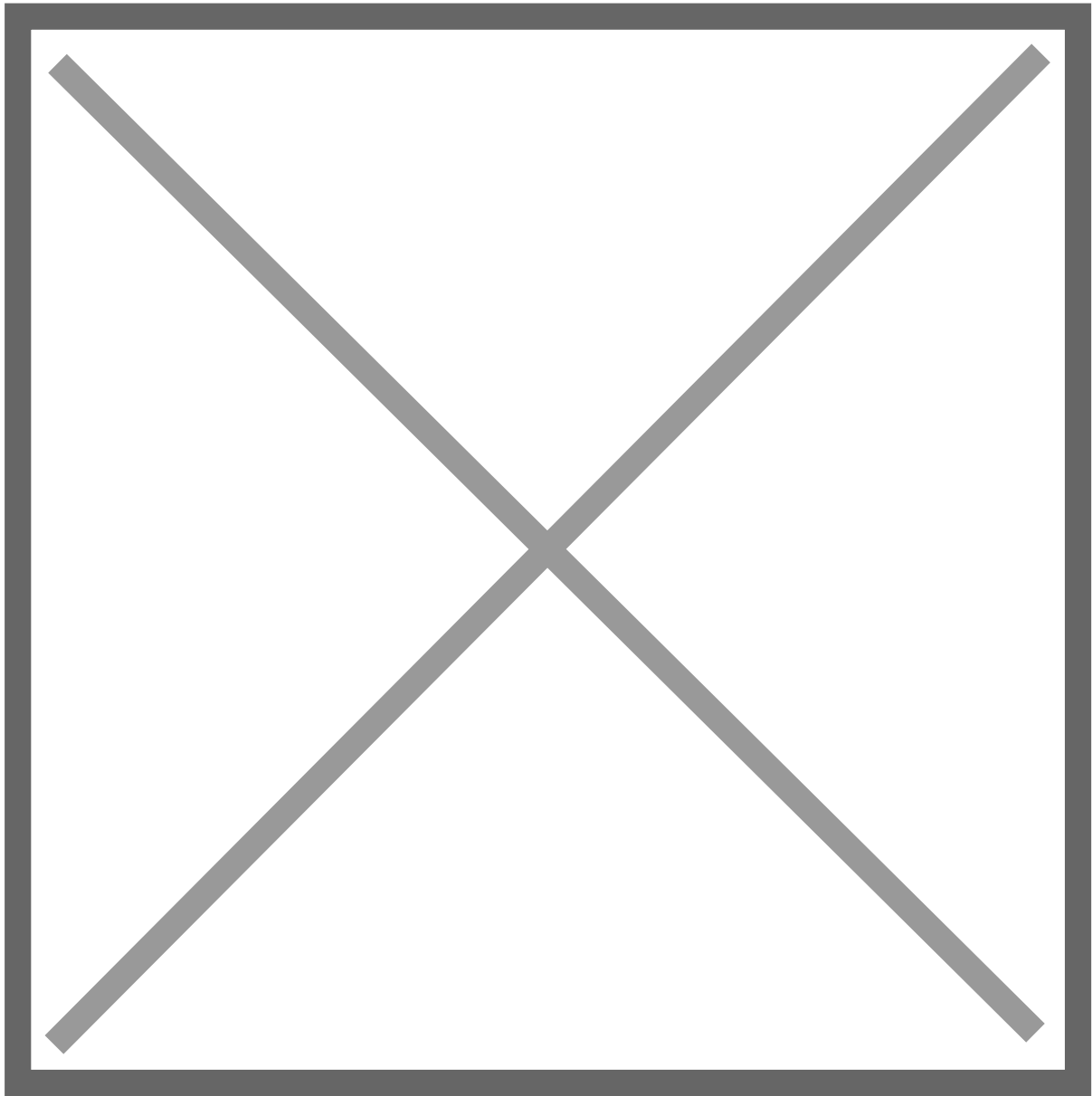
@app.route('/')
def hello_world():
    return 'Hello, World!'

if __name__ == '__main__':
    app.run(debug=False)
```

Flask apps run by default in port 5000, you can test the webapp by running: `python app.py`

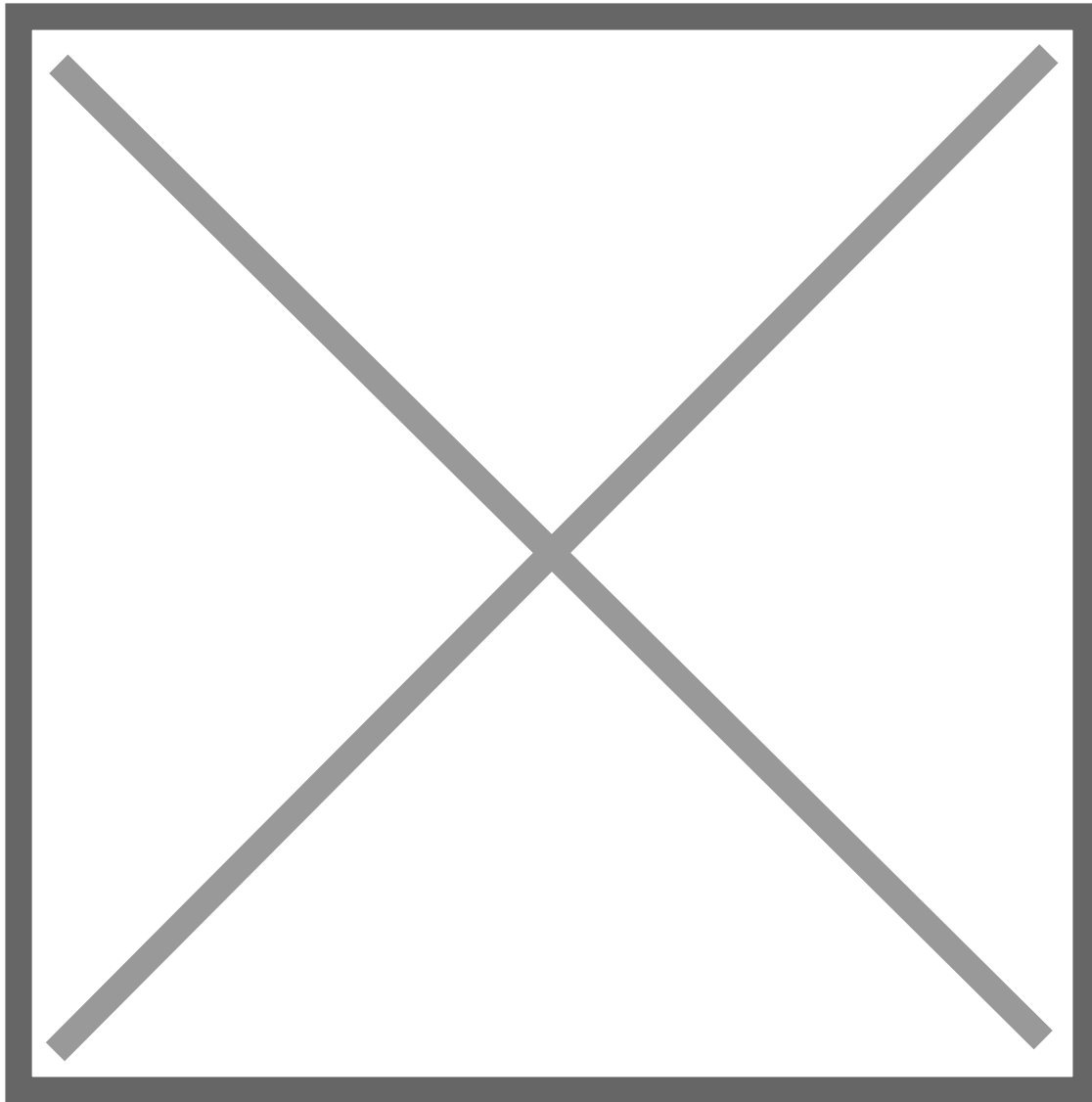


Running Flask app



Accessing web app on the browser

The app is up and running Now we should get the dependencies used in our app, we will save them on requirements.txt .



Saving requirements in a file

```
pip freeze > requirements.txt
```

- We will need these requirements so we can create the Dockerfile of our webapp, then run the following command to create a Dockerfile of the webapp.

```
nano Dockerfile.webapp
```

copy and paste the following:

```
FROM python:3.12.5
Expose 5000
ENV FLASK_app=app.py
WORKDIR /app
COPY ./app.py .
COPY ./requirements.txt .
RUN pip install -r requirements.txt
RUN pip install gunicorn
CMD gunicorn -w 4 -b :5000 app:app
```

Note that we added Gunicorn in our Dockerfile to run our Python web app because using development servers is not suitable for production environments.

Let's create our first image , it'll be saved locally.

```
docker build -f Dockerfile.webapp -t webapp_test .
```

After finalizing the Dockerfile for our web app, it's time to create our Docker Compose YAML file, which will define our entire architecture. We'll break it down step by step.

Here is the docker compose file.

```
version: '3.8'

services:
  haproxy1:
    build:
      context: .
      dockerfile: Dockerfile
    container_name: haproxy1
    networks:
      yahya_prive:
        ipv4_address: 10.0.0.150
    cap_add:
      - NET_ADMIN
    ports:
      - "8888:80"
      - "8404:8404"
    volumes:
      - C:\Users\John macmillan\Desktop\python_project\haproxy.cfg:/usr/local/etc/haproxy/haproxy.cfg
      - C:\Users\John macmillan\Desktop\python_project\keepalived_primary.conf:/etc/keepalived/keepalived.conf
    depends_on:
      - web1
      - web2
      - web3
    entrypoint: ["/bin/sh", "-c", "keepalived -D -f /etc/keepalived/keepalived.conf && haproxy -D -f /usr/local/etc/haproxy/haproxy.cfg"]

  haproxy2:
    build:
      context: .
      dockerfile: Dockerfile
    container_name: haproxy2
    networks:
      yahya_prive:
        ipv4_address: 10.0.0.155
    cap_add:
      - NET_ADMIN
    ports:
      - "8800:80"
      - "8405:8404"
    volumes:
      - C:\Users\John macmillan\Desktop\python_project\haproxy.cfg:/usr/local/etc/haproxy/haproxy.cfg
      - C:\Users\John macmillan\Desktop\python_project\keepalived_bck.conf:/etc/keepalived/keepalived.conf
```

```

depends_on:
  - web1
  - web2
  - web3
entrypoint: ["/bin/sh", "-c", "keepalived -D -f /etc/keepalived/keepalived.conf && haproxy

web1:
  image: webapp_test
  container_name: web1
  networks:
    - yahya_prive

web2:
  image: webapp_test
  container_name: web2
  networks:
    - yahya_prive

web3:
  image: webapp_test
  container_name: web3
  networks:
    - yahya_prive

networks:
  yahya_prive:
    driver: bridge
    #specify the driver
    ipam:
      config :
        - subnet: 10.0.0.0/24
          gateway: 10.0.0.1

```

let's break this down:

First thing we did is to create a network bridge . This Network bridge is called `yahya_prive` then we specified the CIDR notation: `10.0.0.0/24` with the following gateway: `10.0.0.1`

it'll be our private network in which all the containers will be assigned an ip address from the ip address range 10.0.0.0/24 .

```

networks:
  yahya_prive:
    driver: bridge
    #specify the driver
    ipam:
      config :
        - subnet: 10.0.0.0/24
          gateway: 10.0.0.1

```

It's advisable to use a custom network rather than the default network provided by Docker Compose. This approach enhances security and allows you to use domain names instead of IP addresses in configurations.

Then there is a section for our webapps, they're called respectively: web1, web2 and web3, all three of them is now attached to `yahya_prive` network.

Each one was provided a name, and the base image was created previously, and it was saved locally which is `webapp_test`

```
web1:
  image: webapp_test
  container_name: web1
  networks:
    - yahya_prive

web2:
  image: webapp_test
  container_name: web2
  networks:
    - yahya_prive

web3:
  image: webapp_test
  container_name: web3
  networks:
    - yahya_prive
```

After this we've created the service for our HAProxy load balancer.

```
haproxy1:
  build:
    context: .
    dockerfile: Dockerfile
  container_name: haproxy1
  networks:
    yahya_prive:
      ipv4_address: 10.0.0.150
  cap_add:
    - NET_ADMIN
  ports:
    - "8888:80"
    - "8404:8404"
  volumes:
    - C:\Users\John macmillan\Desktop\python_project\haproxy.cfg:/usr/local/etc/haproxy/haproxy.cfg
    - C:\Users\John macmillan\Desktop\python_project\keepalived_primary.conf:/etc/keepalived/keepalived.conf
  depends_on:
    - web1
    - web2
    - web3
  entrypoint: ["/bin/sh", "-c", "keepalived -D -f /etc/keepalived/keepalived.conf && haproxy -
```

The Master HAProxy is called haproxy1, and it was assigned the ip address:

`10.0.0.150` from the private network yahya_prive.

To enable Keepalived, which uses VRRP for failover, the container requires additional network capabilities. Therefore, we grant the container `NET_ADMIN` privileges, allowing it to manage network settings necessary for VRRP operations.

The necessary configuration files for HAProxy and Keepalived are mounted as read-only volumes from the host machine to ensure that both services are properly configured.

The `entrypoint` directive ensures that Keepalived starts in the background and monitors the HAProxy service, providing the high availability setup.

And obviously, there is the configuration `Dockerfile` that contains our Keepalived and HAProxy.

```
FROM ubuntu:22.04

# Install Keepalived and HAProxy
RUN apt-get update && apt-get install -y \
    nano \
    net-tools \
    keepalived \
    haproxy

EXPOSE 80
EXPOSE 8444
```

The configuration for both load balancers is exactly the same (`haproxy.cfg`)

```
global
    stats socket /var/run/api.sock user haproxy group haproxy mode 660 level admin expose-fd listen
    log stdout format raw local0 info

defaults
    mode http
    timeout client 10s
    timeout connect 5s
    timeout server 10s
    timeout http-request 10s
    log global

frontend stats
    bind *:8404
    stats enable
    stats uri /
    stats refresh 10s

frontend myfrontend
    bind :80
    default_backend webservers

backend webservers
    server s1 web1:5000 check
    server s2 web2:5000 check
    server s3 web3:5000 check
```

What we've done here is configure our load balancer to listen for requests on port 80. We've also defined our web servers in the backend webserver section, allowing us to route traffic to them effectively.

On the other hand, we had different configurations of Keepalived daemons for both nodes.

here is the configuration for the master node (`keepalived_primary.conf`):

```
vrrp_instance VI_1 {
    state MASTER
    interface eth0
    virtual_router_id 33
    priority 255
    advert_int 1
    unicast_src_ip 10.0.0.50

    authentication {
        auth_type PASS
        auth_pass letmein
    }

    virtual_ipaddress {
        10.0.0.50/24 dev eth0
    }
}
```

This is a basic configuration where we've assigned a priority of 255, which is higher than the priority set on the backup machine. As a result, this machine will be assigned the virtual IP address first. We've defined the state as `MASTER` for this machine and specified `eth0` as the interface, as it's connected to the private network through this interface.

The most critical part of this configuration is the `virtual_ipaddress` section, where we define the virtual IP address (`10.0.0.50/24`) that will be managed by the VRRP protocol. This IP address will be assigned to the master machine, ensuring that it handles traffic as long as it remains in the master state. The `authentication` section provides basic security by requiring a password for VRRP communications, adding an extra layer of protection to the setup.

The configuration for the backup node will be different a little bit (`keepalived_bck.conf`):

```
vrrp_instance VI_1 {
    state BACKUP
    interface eth0
    virtual_router_id 33
    priority 150
    advert_int 1
    unicast_src_ip 10.0.0.50

    authentication {
        auth_type PASS
        auth_pass letmein
    }
}
```

```
virtual_ipaddress {  
    10.0.0.50/24 dev eth0  
}  
}
```

We've set a priority of 150, which is lower than that of the master, and designated the state as `BACKUP`. Both the master and backup nodes share the same virtual IP address (VIP). In the event of the master node failing, the backup node will detect this through ARP checks and automatically take over the VIP, ensuring continued service availability.

Note that the Docker Compose file includes a `depends_on` section that specifies the order in which containers are started. In this case:

```
depends_on:  
  - web1  
  - web2  
  - web3
```

This configuration ensures that the web applications (`web1`, `web2`, and `web3`) start before the load balancers are initialized. This order is crucial to ensure that the web servers are up and running before the load balancers begin routing traffic to them.

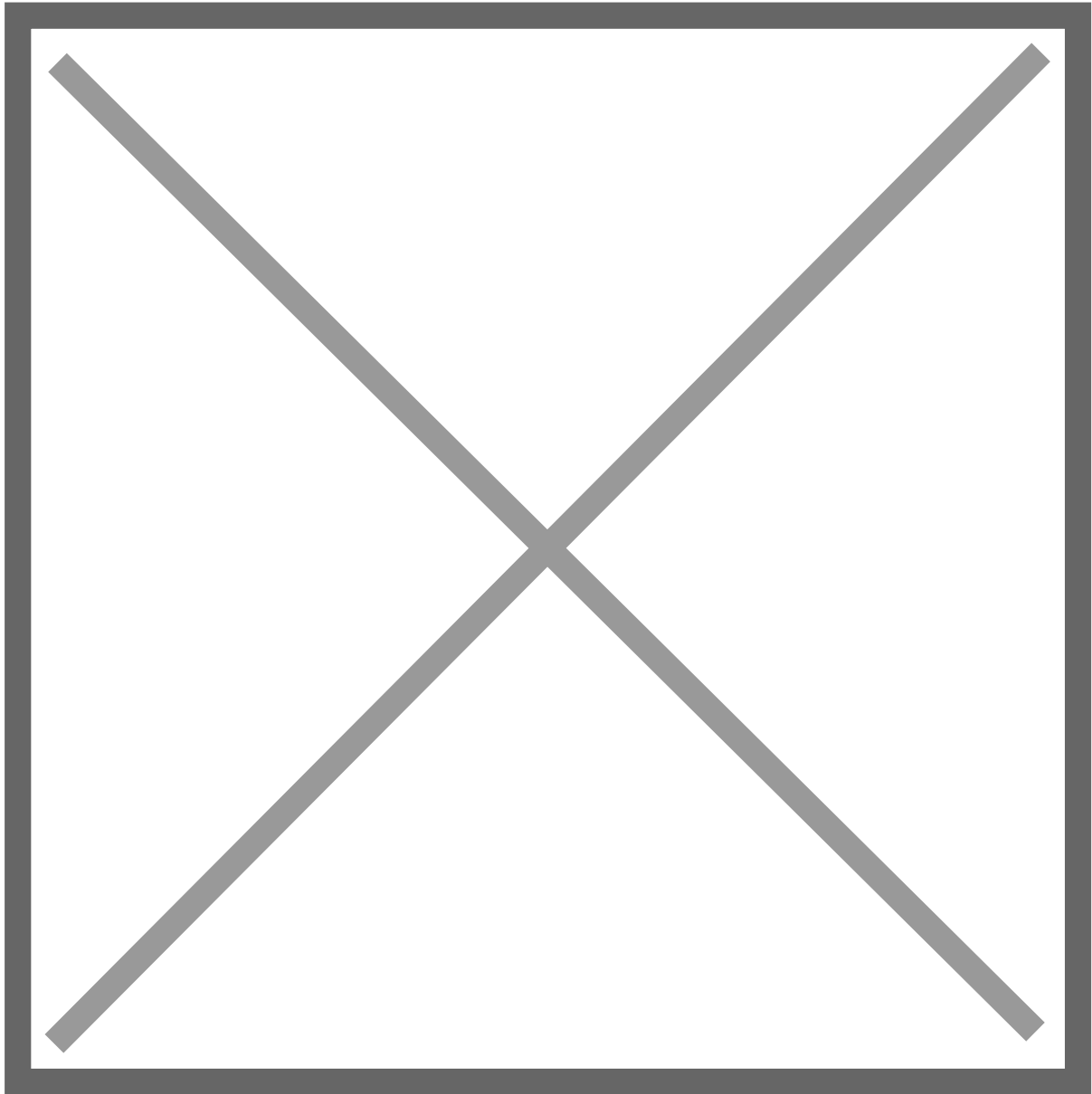
After completing the necessary configurations, you can run the entire setup with the following command:

```
docker-compose -p high_availability_cluster up -d
```

Now, your setup is up and running.

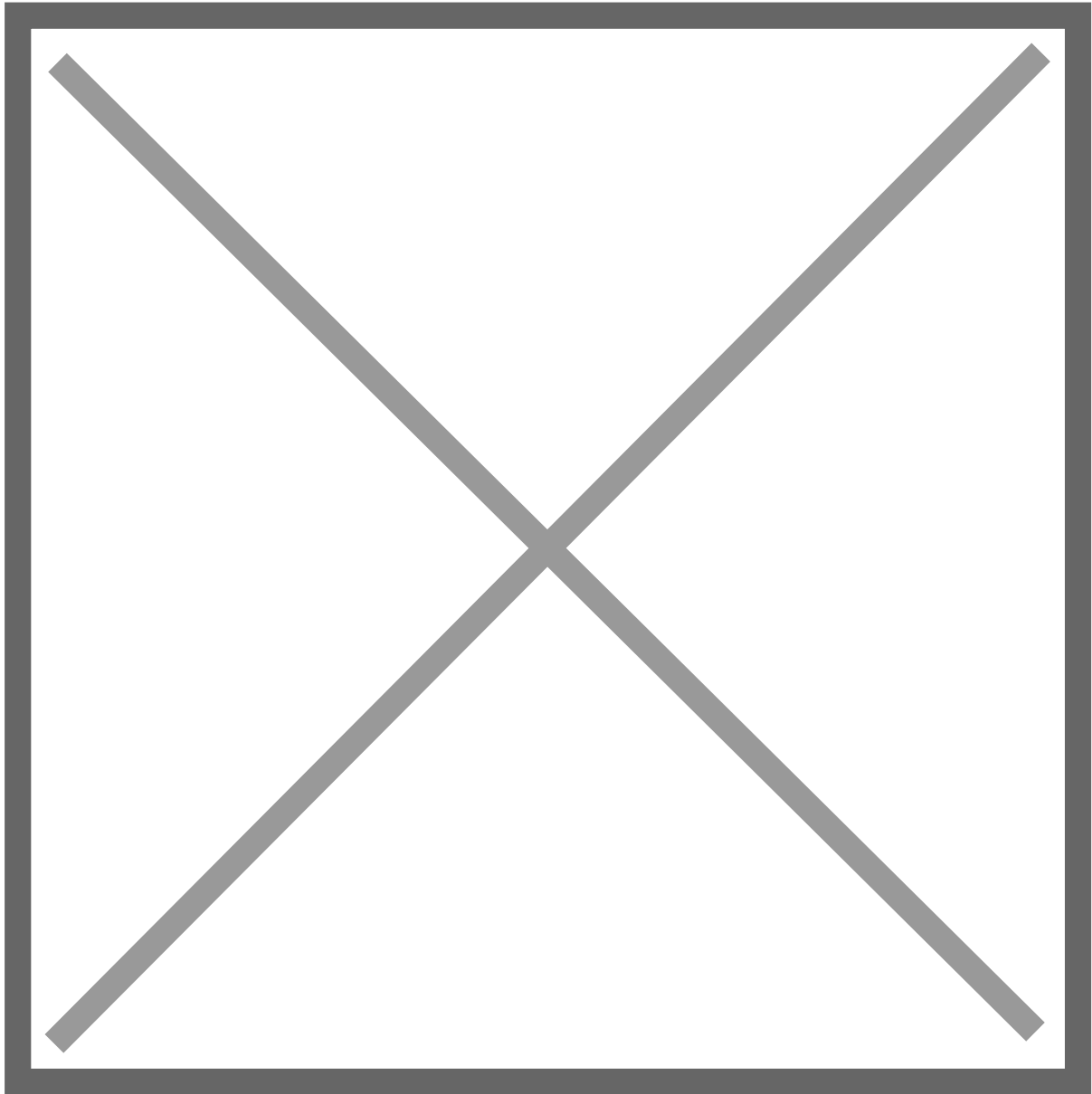
Accessing webapp
Accessing webapp

The `haproxy1` container will receive the virtual IP address on its `eth0` interface. You can verify this by running the following command inside the container: `ip a`.



Virtual Ip address assigned

We can try sending an HTTP request using the `curl` command from a node inside the `yahya_prive` network. For example, we can use `web1` for this purpose.



Getting web content from using VIP

IV-/Conclusion

In conclusion, building a high availability cluster with HAProxy, Keepalived, and Docker is an effective way to ensure continuous service availability and reliability. Through this guide, we've explored the fundamental concepts of high availability, examined how HAProxy and Keepalived work together to manage traffic and failover, and demonstrated how to set up this architecture in a Docker environment. By following these steps, you can create a resilient infrastructure that can handle disruptions and maintain service continuity, making it an essential setup for any robust and scalable application deployment.

Equipped with this understanding, you'll now be able to enhance your network infrastructure and deploy more resilient, scalable applications with confidence.

We invite you to share your experiences and insights in the comments below. We're eager to hear your feedback and thoughts. Happy networking!

References

For more detailed information :

1. [Networks top-level elements | Docker Docs](#)
2. [HAProxy version 2.9.9-41 — Starter Guide](#)
3. [Understanding Virtual Router Redundancy Protocol \(VRRP\) | FS Community](#)
4. [Setting up a Linux cluster with Keepalived: Basic configuration | Enable Sysadmin \(redhat.com\)](#)

Configuração de URL Path no HAPROXY

Arquivo exemplo haproxy.cfg

```
global
    log /dev/log local0
#   log 127.0.0.1 local0 info
    daemon
    log stdout format raw local0
    maxconn 2048
    user haproxy
    group haproxy

defaults
    log global
    mode http
    option httplog
    option dontlognull
    timeout connect 5000ms
    timeout client 50000ms
    timeout server 50000ms
    retries 3

listen stats
    bind *:1120
    mode http
    stats enable
    stats uri /stats
    stats refresh 10s
    stats auth admin:senha123

#####
# FRONTEND 1 - porta 2023
#####
frontend http_front_2023
    bind *:2023
    default_backend doc_backend_2023

#####
# FRONTEND 2 - porta 2024
```

#####

frontend http_front_2024

bind *:2024

default_backend doc_backend_2024

#####

FRONTEND 3 - porta 2025

#####

frontend http_front_2025

bind *:2025

default_backend doc_backend_2025

#####

BACKEND 1 - para frontend 2023

#####

backend doc_backend_2023

balance roundrobin

stick-table type string len 64 size 100k expire 1h store server_id

stick on path # registra somente o /teste

stick on concat(req.hdr(host), path) - entrada para registrar url e /teste

stick on concat(concat(req.hdr(host), path), concat("?", query)) - entrada para registrar url e /teste?normal (query)

server srv1_2023 192.168.0.101:2023 check

server srv2_2023 192.168.0.102:2023 check

server srv3_2023 192.168.0.103:2023 check

#####

BACKEND 2 - para frontend 2024

#####

backend doc_backend_2024

balance roundrobin

stick-table type string len 64 size 100k expire 1h store server_id

stick on path # registra somente o /teste

stick on concat(req.hdr(host), path) - entrada para registrar url e /teste

stick on concat(concat(req.hdr(host), path), concat("?", query)) - entrada para registrar url e /teste?normal (query)

server srv1_2024 192.168.0.104:2024 check

server srv2_2024 192.168.0.105:2024 check

server srv3_2024 192.168.0.106:2024 check

#####

BACKEND 3 - para frontend 2025

#####

backend doc_backend_2025

balance roundrobin

```
stick-table type string len 64 size 100k expire 1h store server_id
stick on path      # registra somente o /teste
# stick on concat(req.hdr(host), path) - entrada para registrar url e /teste
# stick on concat(concat(req.hdr(host), path), concat("?", query)) - entrada para registrar url e
/teste?normal (query)
```

```
server srv1_2025 192.168.0.107:2025 check
server srv2_2025 192.168.0.108:2025 check
server srv3_2025 192.168.0.109:2025 check
```

Configuração de logs no HAPROXY

ChatGPT:

Ótimo! Vamos configurar o **HAProxy + rsyslog** dentro (ou fora) do Docker para gerar logs em **horário local**, salvos no arquivo `/var/log/haproxy.log`.

☐ O que vamos fazer:

1. Configurar o HAProxy para enviar logs ao syslog (`/dev/log`)
 2. Garantir que o rsyslog esteja ativo no host
 3. Mapear o socket `/dev/log` para o container Docker (ou fazer logging fora)
 4. Salvar os logs com horário local no host
-

☐ 1. Configuração do HAProxy (`haproxy.cfg`)

Altere a seção `global` do seu `haproxy.cfg` para:

```
global
    log /dev/log local0 info
    daemon

    maxconn 2048

    user haproxy

    group haproxy

defaults

    log global

    mode http
```

```
option httplog
```

```
option dontlognull
```

```
timeout connect 5000ms
```

```
timeout client 50000ms
```

```
timeout server 50000ms
```

❑ 2. Configurar o rsyslog no **host Linux**

Verifique se o rsyslog está ativo:

```
sudo systemctl status rsyslog
```

Se não estiver, ative:

```
sudo systemctl enable --now rsyslog
```

Crie o arquivo de log e a regra:

```
sudo touch /var/log/haproxy.log
```

```
sudo chown syslog:adm /var/log/haproxy.log
```

Agora crie um arquivo de configuração para o rsyslog:

```
sudo nano /etc/rsyslog.d/49-haproxy.conf
```

E adicione:

```
local0.* /var/log/haproxy.log
```

Reinicie o rsyslog:

```
sudo systemctl restart rsyslog
```

❑ 3. Rodar o container HAProxy com acesso ao `/dev/log`

Você precisa montar o socket do syslog no container:

❏ Exemplo com `docker run`:

bash

```
docker run -d \  
  --name haproxy-logs \  
  -v $(pwd)/haproxy.cfg:/usr/local/etc/haproxy/haproxy.cfg:ro \  
  -v /dev/log:/dev/log \  
  -p 2025:2025 \  
  haproxy:2.9
```

“❏ O `-v /dev/log:/dev/log` é essencial — ele conecta o HAProxy ao rsyslog do host.

❏ Exemplo com `docker-compose.yml`:

```
services:  
  
  haproxy:  
    image: haproxy:2.9  
  
  volumes:  
  
    - ./haproxy.cfg:/usr/local/etc/haproxy/haproxy.cfg:ro  
  
    - /dev/log:/dev/log  
  
  ports:  
  
    - "2025:2025"
```

❏ 4. Verificar os logs

Após acessar HAProxy (ex: `curl http://localhost:2025/teste`), veja os logs com:

```
tail -f /var/log/haproxy.log
```

O horário será o local do host, por exemplo:

```
Jul 18 10:08:27 haproxy-logs haproxy[1]: 192.168.0.10:42839 [18/Jul/2025:10:08:27.315] http_front  
doc_backend ...
```

☐☐ Dica extra: usar hostname no log

Se quiser que apareça o nome do container no log, passe `-h haproxy-logs` no `docker run`, ou use no `docker-compose.yml`:

```
hostname: haproxy-logs
```

☐☐ Pronto!

Agora o HAProxy está:

- Gerando logs com `option httplog`
- Mandando logs para o rsyslog do host via `/dev/log`
- Gravando logs com horário **local** em `/var/log/haproxy.log`

Reconfiguração automática do haproxy.cfg do HAPROXY

ChatGPT:

Perfeito! Aqui está um script cron completo e seguro para fazer o reload do haproxy.cfg em todos os containers haproxy do Swarm, sem reiniciar os containers — apenas recarregando o processo haproxy com kill -HUP.

❏ Script: reload-haproxy-containers.sh

bash

INICIO SCRIPT

```
#!/bin/bash
# Nome da imagem usada no serviço haproxy
IMAGE_NAME="haproxytech/haproxy-debian:3.3"
# Comando de reload usado dentro do container
RELOAD_CMD="haproxy -c -f /usr/local/etc/haproxy/haproxy.cfg && kill -HUP 1"
# Busca containers que usam a imagem especificada
CONTAINERS=$(docker ps --filter "ancestor=$IMAGE_NAME" --format "{{.ID}}")

if [ -z "$CONTAINERS" ]; then
    echo "[INFO] Nenhum container HAProxy em execução com imagem '$IMAGE_NAME'"
    exit 0
fi

echo "[INFO] Recarregando configuração HAProxy em $(echo "$CONTAINERS" | wc -l) container(s)..."

for cid in $CONTAINERS; do
    echo " → Validando e recarregando HAProxy no container $cid..."
    docker exec "$cid" sh -c "$RELOAD_CMD" && \
        echo " ❏ Container $cid recarregado com sucesso." || \
        echo " ❏ Erro ao recarregar o container $cid."
done
```

FIM SCRIPT

❏ Como usar

1. Salve o script

```
sudo mkdir -p /usr/local/bin
sudo nano /usr/local/bin/reload-haproxy-containers.sh
Cole o conteúdo, salve, e depois:
sudo chmod +x /usr/local/bin/reload-haproxy-containers.sh
```

2. Agende via crontab

Edite o cron para o usuário que gerencia o Swarm (geralmente root):

```
sudo crontab -e
```

Adicione a linha para executar de hora em hora:

```
0 * * * * /usr/local/bin/reload-haproxy-containers.sh >> /var/log/reload-haproxy.log 2>&1
```

❏ Segurança adicional (opcional)

Se quiser ser ainda mais seletivo, você pode filtrar por nome do serviço ou label, ex:

```
docker ps --filter "name=mystack_haproxy" --format "{{.ID}}"
```

Ou com label:

```
docker ps --filter "label=com.docker.stack.namespace=mystack" --format "{{.ID}}"
```

❏ Resultado

Com isso, seu Swarm:

- ❏ Executará reload leve e seguro do haproxy.cfg de hora em hora
- ❏ Sem precisar redeploy, sem downtime
- ❏ Sincronizado via GlusterFS automaticamente
- ❏ Sem scripts dentro dos containers

Reconfiguração automática do haproxy.cfg do HAPROXY

- 2

chatGPT

Perfeito! Abaixo está um script que monitora alterações no arquivo haproxy.cfg via inotify e executa um reload automático nos containers HAProxy quando detectar mudanças — ideal para quem está usando GlusterFS com bind mount no Docker Swarm.

□ Objetivo

Monitorar:

bash
/opt/aplicativos/haproxy/haproxy.cfg
Quando houver qualquer modificação, o script:

Valida a nova configuração com haproxy -c

Executa kill -HUP 1 no processo do HAProxy dentro de cada container ativo

□ Pré-requisitos

Instale o inotify-tools no node onde o script vai rodar:

bash
Copiar
Editar
sudo apt install inotify-tools
□ Script: watch-haproxy-reload.sh
bash
Copiar
Editar
#!/bin/bash

WATCH_FILE="/opt/aplicativos/haproxy/haproxy.cfg"
IMAGE="haproxytech/haproxy-debian:3.3"

echo "[INFO] Monitorando \$WATCH_FILE por alterações..."

```
inotifywait -m -e close_write "$WATCH_FILE" | while read -r path event file; do
    echo "[INFO] Detecção de alteração em $file — iniciando validação e reload..."
```

```
CONTAINERS=$(docker ps --filter "ancestor=$IMAGE" --format "{{.ID}}")
```

```
if [ -z "$CONTAINERS" ]; then
    echo "[WARN] Nenhum container com imagem $IMAGE em execução."
    continue
fi
```

```
for cid in $CONTAINERS; do
    echo "→ Validando config dentro do container $cid..."
    if docker exec "$cid" haproxy -c -f /usr/local/etc/haproxy/haproxy.cfg; then
        echo "✓ Configuração válida. Recarregando HAProxy no container $cid..."
        docker exec "$cid" kill -HUP 1
        echo "□ Reload concluído para $cid"
    else
        echo "□ Configuração inválida. Nenhum reload feito."
    fi
done
```

done

□ Como usar

1. Salve o script

bash

Copiar

Editar

```
sudo nano /usr/local/bin/watch-haproxy-reload.sh
```

Cole o conteúdo acima, salve e torne executável:

bash

Copiar

Editar

```
sudo chmod +x /usr/local/bin/watch-haproxy-reload.sh
```

2. Execute em background (em 1 só node basta)

bash

Copiar

Editar

```
nohup /usr/local/bin/watch-haproxy-reload.sh >> /var/log/haproxy-watch.log 2>&1 &
```

Ou adicione no /etc/rc.local ou systemd para subir automaticamente no boot.

□ Segurança e estabilidade

Use em apenas 1 node do Swarm para evitar reloads duplicados.

O reload é suave (com SIGHUP), não derruba conexões abertas.

Você pode integrar isso com o script cron também, se quiser backup/rollback.

☐ Resumo do que você tem agora:

Item Status

GlusterFS replicando haproxy.cfg ☐

Containers Docker Swarm usando bind mount ☐

Reload automático com inotify ☐

Validação de configuração antes do reload ☐