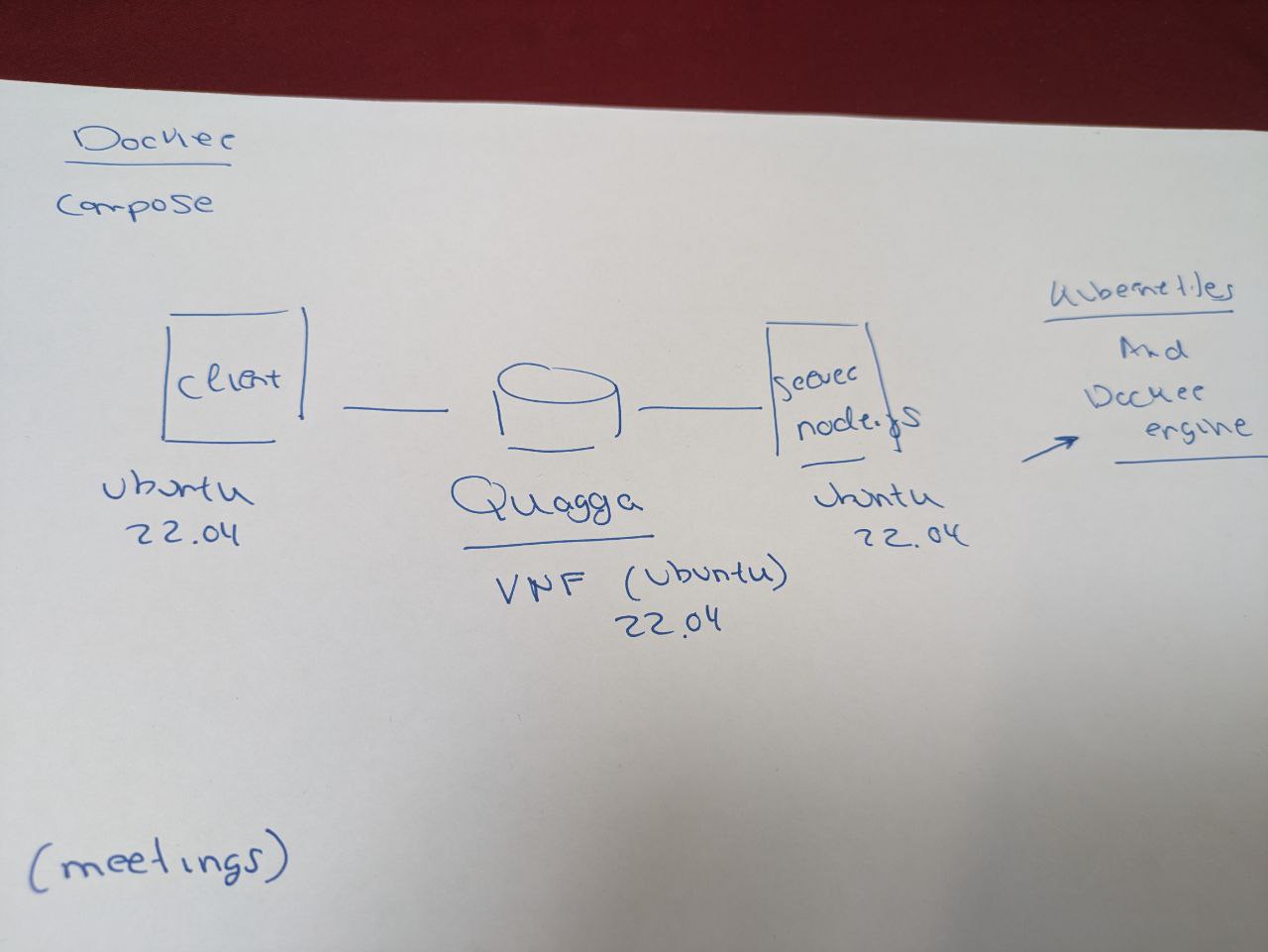
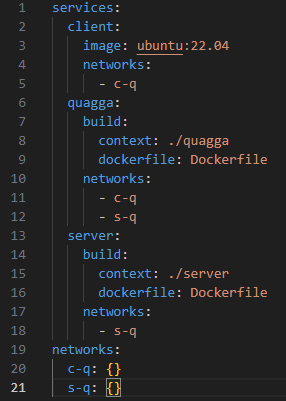
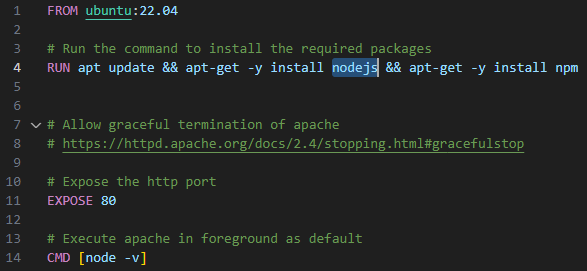
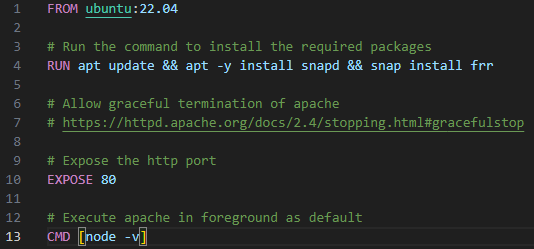
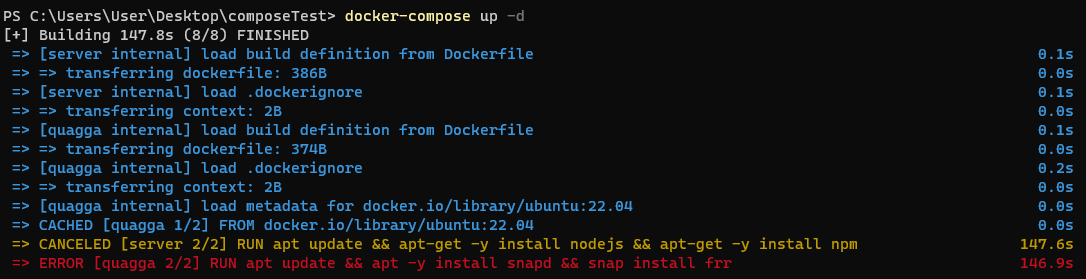
The aim is creating a virtual environment to create a monitoring system or a way to analyse the state of the system.





 server







It seemed that quagga was not available for Ubuntu 22.04 and the installation process it is very time expensive. For this reason, it is taken as hypothesis to switch the image of Ubuntu with other which are lighter.

Comparison between quagga, frr and a new base image.

Quagga is a routing software package that provides TCP/IP based routing services, using protocols such as RIP, OSPF, IS-IS and BGP-4. It also supports IPv6 routing protocols. A system with Quagga installed acts as a dedicated router. With Quagga, your machine exchanges routing information with other routers using routing protocols. Quagga uses this information to update the kernel routing table so that the right data goes to the right place.

It is made from a collection of several daemons that work together to build the routing table. There may be several protocol-specific routing daemons and zebra the kernel routing manager.

Each protocol has its own daemon in charge of managing that routing table. Finally, one more daemon that manages the kernel routing table. It is easy to add a new routing protocol daemons to the entire routing system without affecting any other software. You need to run only the protocol daemon associated with routing protocols in use. Thus, user may run a specific daemon and send routing reports to a central routing console. Each daemon has it’s own configuration file and terminal interface. When you configure a static route, it must be done in zebra configuration file.

FRRouting is a free and open source IRP suite for Linux and Unix platform with its roots planted in Quagga. It offers some improvement for each protocol offered by quagga

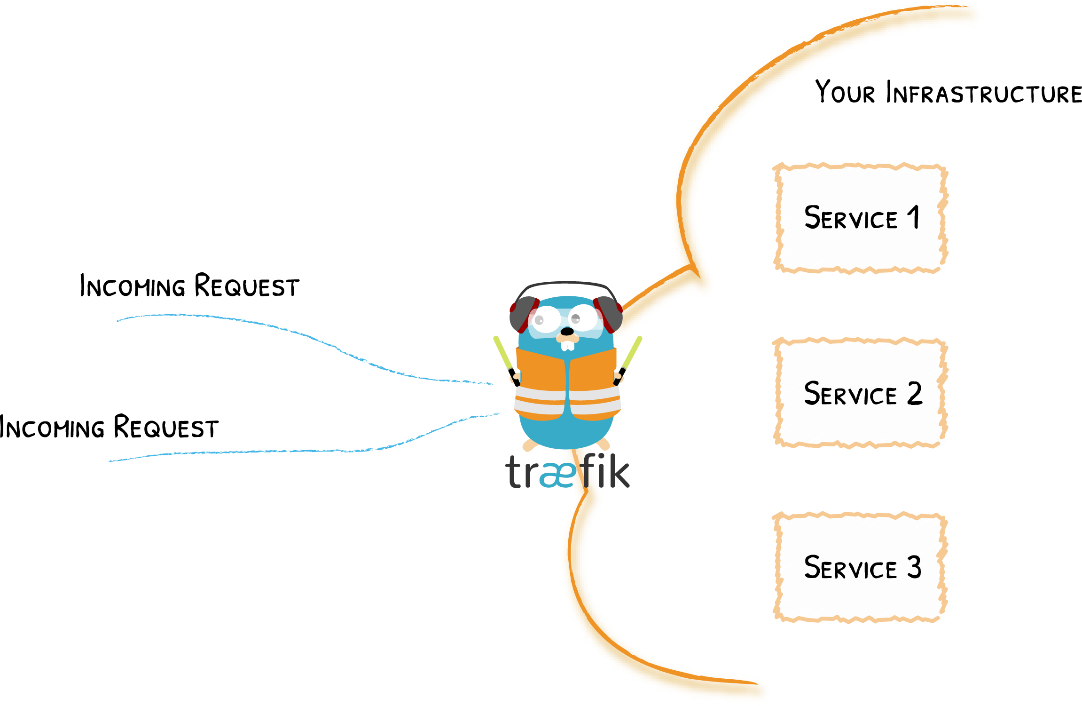
Traefik is an [open-source](https://github.com/traefik/traefik) Edge Router that makes publishing your services a fun and easy experience. It receives requests on behalf of your system and finds out which components are responsible for handling them. With Traefik, there is no need to maintain and synchronize a separate configuration file: everything happens automatically, in real time (no restarts, no connection interruptions).

Traefik is based on the concept of EntryPoints, Routers, Middlewares and Services.

The main features include dynamic configuration, automatic service discovery, and support for multiple backends and protocols.

1. [EntryPoints](https://doc.traefik.io/traefik/routing/entrypoints/): EntryPoints are the network entry points into Traefik. They define the port which will receive the packets, and whether to listen for TCP or UDP.
2. [Routers](https://doc.traefik.io/traefik/routing/routers/): A router is in charge of connecting incoming requests to the services that can handle them.
3. [Middlewares](https://doc.traefik.io/traefik/middlewares/overview/): Attached to the routers, middlewares can modify the requests or responses before they are sent to your service
4. [Services](https://doc.traefik.io/traefik/routing/services/): Services are responsible for configuring how to reach the actual services that will eventually handle the incoming requests.

Traefik is an Edge Router (or reverse Proxy), it means that it's the door to your platform, and that it intercepts and routes every incoming request: it knows all the logic and every [rule](https://doc.traefik.io/traefik/routing/routers/#rule) that determine which services handle which requests



when a service is deployed, Traefik detects it immediately and updates the routing rules in real time. Similarly, when a service is removed from the infrastructure, the corresponding route is deleted accordingly.

Trying to set Traefik I had trouble trying to ping from the client to a server. But this could be related to the fact that the router was not set as forwarding node (to be checked).

FRR seemed not to work. Once configured as show in the documentation and started the service, it required a lot of time to complete the command and once checked if the service was actually working (service frr status), the answer was:

\* Status of watchfrr: running

 \* Status of zebra: FAILED

 \* Status of bgpd: FAILED

 \* Status of ospfd: FAILED

 \* Status of staticd: running

Which means that the daemons failed and are not running. To solve this problem I had to set higher privileges.

Once done this and set the router as forwarding node (to check if it is necessary) the frr daemons started working.

During the studying, I also tried to run GNS3 and it ended in bringing compatibility problems related to WSL.

Comparison between Ubuntu with node.js, alpine+node and apache.

I have to agree with the others, I always use official images if I can and I always select the Alpine version if available. Why use Alpine? Two reasons:

1. Much small images. Ubuntu is 188MB alone. Then you add your app on top of that probably exceeding 200MB. Alpine Linux is only 4MB! After adding my Python runtime and code most of my images are only 52MB. Compare that will almost 200MB of Ubuntu. Smaller images are smaller upload/download and take up less disk space.
2. Smaller attack surface! When you start from Ubuntu, you are adding lots of other services that may be running that are not needed and could be exploited. Alpine Linux has very little to attack so it is inherently more secure.

You don’t need an entire generic OS… all you need is a runtime for your app and nothing more. I always start with Alpine Linux when I’m building my own containers from scratch.

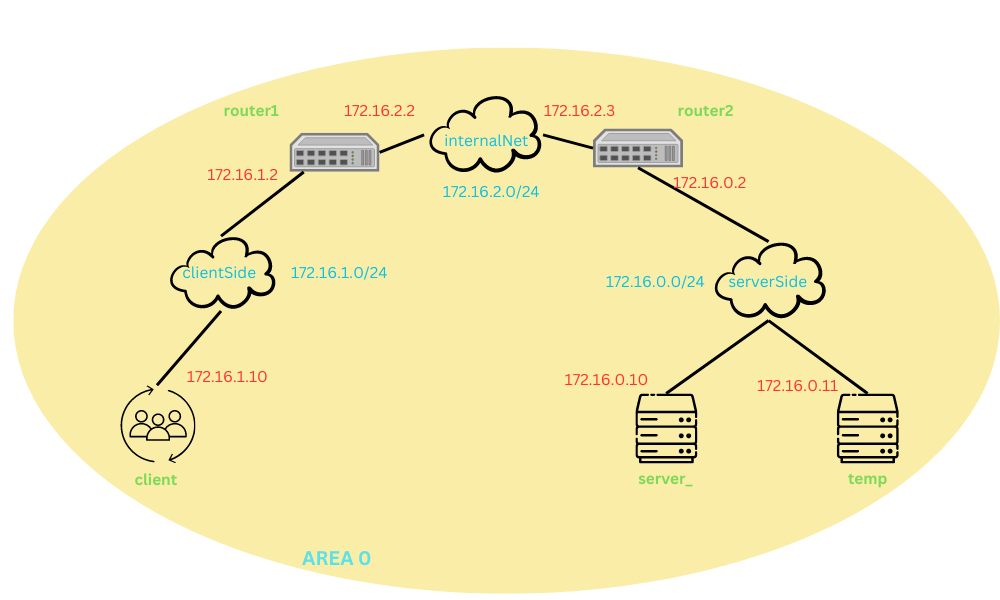
How to use node:

<https://github.com/optimized/docker-node/blob/master/Dockerfile>

https://github.com/nodejs/docker-node/blob/main/README.md#how-to-use-this-image

<https://github.com/nodejs/docker-node>

Since the protocol I decided to use is ospf and a routing protocol make sense if there are more than one router, I decided to add one more but to make it simple by putting everything inside just one area (area 0).



**# Intelligent-forensics-for-the-automatic-anomaly-detection-in-distributed-infrastructures README.md**

The system is composed by a client (ubuntu 22.04 with iputils-ping and iproute2), 2 server (server\_: ubuntu 22.04; temp: ubuntu 22.04 with iputils-ping, iproute2, nodejs, npm and other packet that still have their usefullness to be checked) and 2 routers (ubuntu 22.04 with iputils-ping, iproute2, coreutils, iptables, frr).

The routers have been set t0 work with the protocol ospf. For sake of semplicity, in this phase, the entire architecture have been inserted in a single area (area 0).

![architecture](./images/architecture\_temp2.png)

To run the system:

1. run the docker engine (tested: by running docker desktop)

2. launch the command "docker-compose up -d"

3. access the server temp "docker exec -it intelligent-forensics-temp-1 bash". Add the route to the client network "ip route add 172.16.1.0/24 via 172.16.0.2", in which 172.16.0.2 is the interface of the router connected to the server (router2).

4. in another terminal, access the client "docker exec -it intelligent-forensics-client-1 bash". Add the route to the server network "ip route add 172.16.0.0/24 via 172.16.1.2", in which 172.16.1.2 is the interface of the router connected to the client (router1). Test the connection "ping 172.16.0.11" by pinging the server temp: it should fail:

```console

PING 172.16.0.11 (172.16.0.11) 56(84) bytes of data.

From 172.16.1.2 icmp\_seq=2 Redirect Host(New nexthop: 172.16.1.1)

From 172.16.1.2 icmp\_seq=3 Redirect Host(New nexthop: 172.16.1.1)

From 172.16.1.2 icmp\_seq=4 Redirect Host(New nexthop: 172.16.1.1)

```

5. access the router1 "docker exec -it intelligent-forensics-router1-1 bash". Start frr "service frr start" and check its status "service frr status" is:

```console

 \* Status of watchfrr: running

 \* Status of zebra: running

 \* Status of bgpd: running

 \* Status of ospfd: running

 \* Status of staticd: running

```

which means all the frr daemons chosen are running.

6. access the router2 "docker exec -it intelligent-forensics-router2-1 bash". Start frr "service frr start" and check its status "service frr status" is:

```console

 \* Status of watchfrr: running

 \* Status of zebra: running

 \* Status of bgpd: running

 \* Status of ospfd: running

 \* Status of staticd: running

```

which means all the frr daemons chosen are running.

7. Now it's necessary to configure the protocol for both the router. Enter the vty bash with the command "vtysh". Enter the configuration with "conf t" and then the conf of the router "router ospf". Finally, insert the network connected to the router into the same area with "network <xx.xx.xx.xx/xx> area <x>".

     - router1: "network 172.16.2.0/24 area 0" "network 172.16.1.0/24 area 0"

     - router2: "network 172.16.2.0/24 area 0" "network 172.16.0.0/24 area 0"

     Finish this operation with "end"

After this operation, thanks to the daemon of ospf, the routers are going to exchange their routing tables. Verify it with the command "show ip route" that should give:

- router1:

```console

    K>\* 0.0.0.0/0 [0/0] via 172.16.1.1, eth0, 00:02:52

    O>\* 172.16.0.0/24 [110/20] via 172.16.2.3, eth1, weight 1, 00:00:54

    O   172.16.1.0/24 [110/10] is directly connected, eth0, weight 1, 00:01:29

    C>\* 172.16.1.0/24 is directly connected, eth0, 00:02:52

    O   172.16.2.0/24 [110/10] is directly connected, eth1, weight 1, 00:01:37

    C>\* 172.16.2.0/24 is directly connected, eth1, 00:02:52

```

- router2

```console

    K>\* 0.0.0.0/0 [0/0] via 172.16.2.1, eth0, 00:02:41

    O   172.16.0.0/24 [110/10] is directly connected, eth1, weight 1, 00:01:00

    C>\* 172.16.0.0/24 is directly connected, eth1, 00:02:41

    O>\* 172.16.1.0/24 [110/20] via 172.16.2.2, eth0, weight 1, 00:00:55

    O   172.16.2.0/24 [110/10] is directly connected, eth0, weight 1, 00:01:02

    C>\* 172.16.2.0/24 is directly connected, eth0, 00:02:41

```

8. Exit the vtysh with "exit" and check the route of the host with "ip route show":

- router1:

```console

    default via 172.16.1.1 dev eth0

    172.16.0.0/24 nhid 12 via 172.16.2.3 dev eth1 proto ospf metric 20

    172.16.1.0/24 dev eth0 proto kernel scope link src 172.16.1.2

    172.16.2.0/24 dev eth1 proto kernel scope link src 172.16.2.2

```

- router2:

```console

   default via 172.16.2.1 dev eth0

    172.16.0.0/24 dev eth1 proto kernel scope link src 172.16.0.2

    172.16.1.0/24 nhid 14 via 172.16.2.2 dev eth0 proto ospf metric 20

    172.16.2.0/24 dev eth0 proto kernel scope link src 172.16.2.3

```

9. Now verify that the client can actually ping the server. Enter the client bash and "ping 172.16.0.11" to ping the server temp.

**# advices**

If the images gets modified by changing the dockerfiles, to be effective the image should be mounted again or simply deleted before using again "docker-compose up -d". Fast way is deleting the images.

EXTRA: check if dividing in 2 areas the architecture something change and further configuration are needed. For example it is possible that the border routers to exchange info between them need others protocol like bgp.

DOCKER LOGS: <https://stackoverflow.com/questions/41144589/how-to-redirect-docker-container-logs-to-a-single-file>

FIWARE:

Generated with chatGPT:

FIWARE is an open-source platform designed to facilitate the development and deployment of smart applications and services in various domains, particularly within the context of the Internet of Things (IoT) and smart cities. It provides a set of standardized APIs (Application Programming Interfaces) that enable seamless communication and data exchange between different components and systems.

The main goal of FIWARE is to create a framework that promotes interoperability, scalability, and reusability of smart solutions. It aims to bridge the gap between proprietary technologies and provide a common foundation for developers, businesses, and governments to build smart applications and services more efficiently.

Key features and components of FIWARE include:

1. Context Broker: The Context Broker is a central component of FIWARE that manages and stores context information about devices, sensors, and entities. It allows applications to query and update context data in real-time.

2. NGSI (Next Generation Service Interface) API: This API is used to interact with the Context Broker and access context information.

3. Data Models: FIWARE provides standard data models for various types of entities, making it easier to represent and handle different types of data in a standardized way.

4. IoT Agents: FIWARE offers IoT agents that facilitate the integration of IoT devices and sensors with the platform.

5. Application Mashups: FIWARE enables the development of applications by combining data and services from various sources, fostering a collaborative and integrated approach to development.

6. Publishing and Subscribing: FIWARE supports the publish/subscribe model, allowing applications to subscribe to real-time updates and events from context data sources.

FIWARE is not limited to a specific industry or application area, but it has found significant adoption in various domains, including smart cities, agriculture, transportation, environmental monitoring, and more.

The project is driven by the FIWARE Foundation, a non-profit organization that oversees the development and promotion of the FIWARE platform. It encourages an open and collaborative ecosystem, where developers can contribute to the platform's enhancement and share their applications with the community.

----

FIWARE is an open-source platform designed to offer a unique way of represents data to facilitate communication between different organisations and individuals. Its standardised API allows the acceleration of the development of smart solutions promoting interoperability, scalability and reusability. It deals mainly with the context of the IoT and smart cities.

Generated with chatGPT:

Yes, within the FIWARE ecosystem, there is a specific component known as "FIWARE Cygnus" that is related to Docker and is used for gaining information from Docker logs and other data sources.

FIWARE Cygnus is an open-source connector service that facilitates the integration of context data from different sources into FIWARE's Context Broker. It is commonly used for handling context data generated by IoT devices, but it can also be employed for other sources like logs from Docker containers.

Cygnus supports multiple persistence sinks, including databases (such as MySQL and MongoDB) and data brokers (such as Apache Kafka). These sinks allow Cygnus to store data from various sources in a format that is easily consumable by other FIWARE components and applications.

To collect information from Docker logs using Cygnus, you would typically set up the Docker container logs as a data source for Cygnus. Cygnus would then process these logs and forward the relevant context data to the FIWARE Context Broker, which acts as a central repository for context information.

It's important to note that the FIWARE ecosystem is dynamic and continuously evolving, so it's always a good idea to check the latest documentation and resources for specific details on how to set up and use Cygnus for Docker log integration. Additionally, other components or plugins may be available or introduced over time to handle Docker logs or other aspects of Docker within the FIWARE framework.

<https://fiware-docker-container-service.readthedocs.io/en/latest/>

<https://fiware-monitoring.readthedocs.io/en/develop/> : https://github.com/telefonicaid/fiware-monitoring