Università di Pisa

Scuola di Ingegneria

Dipartimento di Ingegneria dell'Informazione

Laurea triennale in Ingegneria Informatica

Sviluppo di un software di network management per reti OSPF emulate con CONTAINERlab

Relatori:

Prof. Enzo Mingozzi

Prof. Antonio Virdis

Candidato:

Lorenzo Vezzani

Anno accademico 2024/25

Sommario

Capitolo 1

Introduzione

1.1 Importanza del network management

Con la crescita esponenziale del World Wide Web e di Internet, le reti informatiche sono diventate sempre più estese e complesse, rendendone la gestione manuale impraticabile. Per far fronte a questa sfida, sono stati sviluppati software specifici per la gestione delle reti, con l'obiettivo di semplificare e ottimizzare il controllo di infrastrutture sempre più articolate. Questi strumenti raccolgono dati sui dispositivi di rete e forniscono agli amministratori un quadro chiaro e dettagliato dello stato della rete. In questo modo, è possibile individuare rapidamente eventuali anomalie, come guasti, colli di bottiglia nelle prestazioni o problemi di conformità.

1.2 Panomarica del software sviluppato

Il software sviluppato ha come obiettivo la ricostruzione della topologia di rete partendo dalle informazioni presenti all'interno del database **OSPF** di uno specifico router della rete.
Per garantire una ricostruzione completa della topologia, sono stati utilizzati tutti i tipi di Link State Advertisement caratterizzanti OSPFv2 (LSA di tipo 1–5, come definiti nell'RFC 1247). Questo approccio consente di includere non solo informazioni intra-area, ma anche quelle **inter-area** e **inter-autonomous-system**, offrendo così una visione globale e dettagliata della rete.
L'applicativo software è progettato per essere eseguito su un end-device all'interno della rete emulata, al quale è possibile accedere direttamente tramite CLI utilizzando comandi **Docker** o **SSH**. Per rendere l'applicazione più completa e fruibile, oltre alla possibilità di interagire con l'applicazione tramite comandi dalla shell dell'end-device selezionato, è stata implementata anche un'interfaccia grafica web-based che consente una visualizzazione diretta della rete. In particolare, il backend dell'applicazione è sviluppato utilizzando **Flask**, mentre il frontend sfrutta la libreria **vis.js** per la rappresentazione grafica interattiva della topologia di rete.

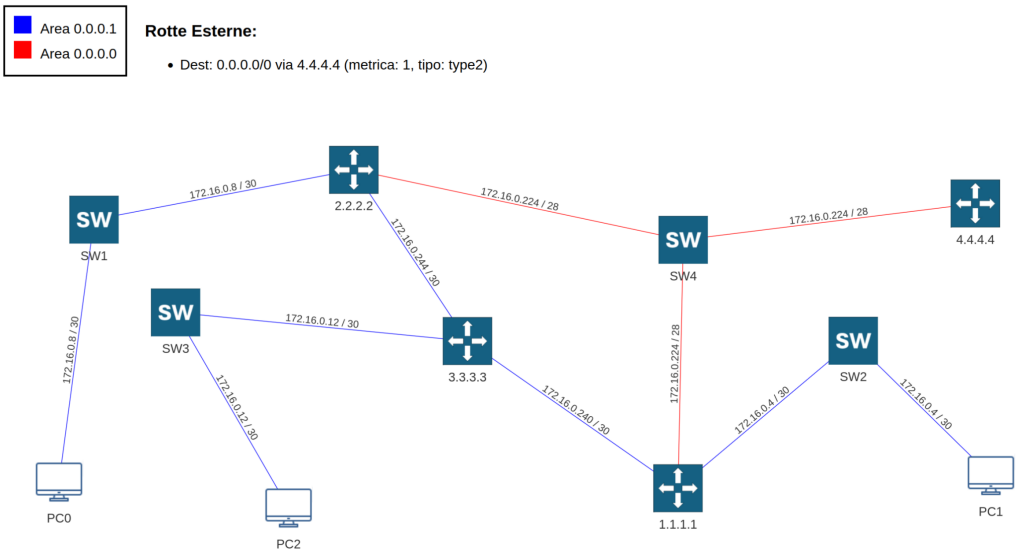

Figure 1: Snapshot dell'applicazione

Capitolo 2

CONTAINERlab

2.1 Cos'è CONTAINERlab?

CONTAINERlab è un tool open-source per l'emulazione di topologie di rete basate su container. In particolare, CONTAINERlab utilizza i container **Docker**: essi permettono una gestione flessibile ed efficiente della rete emulata, senza necessità di hypervisor o hardware dedicato.

2.2 Caratteristiche principali

Le principali feature che rendono CONTAINERlab un tool affidabile e immediato sono:

- Approccio "Lab as Code" (IaC): consente di definire i laboratori in modo dichiarativo tramite file di configurazione .clab.yaml.
- Orchestrazione avanzata dei laboratori: Oltre all'avvio e all'interconnessione dei container, offre funzionalità avanzate per la gestione del ciclo di vita dei laboratori, tra cui deploy, destroy, save, inspect e operazioni grafiche.
- Compatibile con nodi basati su macchine virtuali: grazie all'integrazione con vrnetlab, è possibile combinare nodi containerizzati e virtualizzati, mantenendo lo stesso flusso di lavoro e approccio infrastrutturale.
- Supporto per immagini di rete: Compatibile con container di vendor come Arista, Cisco, Nokia, FRRouting, ma anche con immagini generiche basate su Linux.
- Portabilità elevata: Permette di creare laboratori basati su container in modo estremamente rapido su qualsiasi sistema Linux con Docker.
- Documentazione chiara e completa: CONTAINERlab pone grande attenzione alla qualità della documentazione, offrendo guide dettagliate per evitare incertezze nell'utilizzo dello strumento.

Figure 2: Logo di CONTAINERlab

2.3 Deploy e gestione di laboratori con CONRAINERlab

La descrizione della topologia di rete da emulare si trova all'interno di un file con estensione **.clab.yaml**. Esso descrive la topologia attraverso una lista di nodi ed una lista di link.
In particolare, per ogni nodo è necessario specificare l'attributo *kind*, che definisce il tipo di nodo e il suo ruolo nella rete emulata, e l'attributo *image*, che indica la posizione dell'immagine Docker utilizzata, sia essa locale o remota. I collegamenti tra i nodi, invece, richiedono esclusivamente la specifica delle interfacce che fungono da endpoint.

```
name: srlceos01

topology:
  nodes:
    srl:
      kind: nokia_srlinux
      image: ghcr.io/nokia/srlinux:24.3.3
    ceos:
      kind: arista_ceos
      image: ceos:4.32.0F

  links:
    - endpoints: ["srl:e1-1", "ceos:eth1"]
```

Figure 3: Un semplice file .clab.yaml

Il comando da eseguire sulla macchina host per avviare il deploy della rete è:

`containerlab [--t topology.clab.yaml]`[1] `deploy`

Una volta completato il deploy, sarà possibile accedere ai singoli nodi della rete utilizzando SSH o i comandi Docker:

```
ssh username@target-node
```

```
docker exec -it target-node mode
```

All'interno di un nodo è possibile eseguire i comandi tipici di un qualsiasi NetOS che permettono di configurare le interfacce di rete, gestire il routing, eseguire il debugging dei pacchetti, visualizzare le tabelle di routing ecc.
A tal proposito, il comando *save* consente di salvare eventuali modifiche apportate ai nodi:

`containerlab [--t topology.clab.yaml] save [--node-filters node]`[2]

Infine, è ovviamente possibile terminare l'emulazione utilizzando il comando:

```
containerlab [--t topology.clab.yaml] destroy
```

Nel file `topology.clab.yaml` è possibile definire ultriore attributi relativi ai nodi che compongono la topologia.

- **mgmt-ipv4 / mgmt-ipv6**: indirizzo IP di gestione assegnato al nodo;
- **env**: variabili d'ambiente da passare al container;

[1] [–t topology.clab.yaml] è da utilizzare se `topology.clab.yaml` non si trova nella directory corrente.
[2] [–node-filters node] deve essere utilizzato qualora si desideri salvare le modifiche su un singolo nodo.

- **cmd**: comando da eseguire all'avvio del container;
- **binds**: permette di montare volumi (meccanismo di condivisione di dati tra container) tra host e container;
- **volumes**: permette di definire volumi Docker permanenti;
- **interfaces**: definisce le interfacce del nodo specificando name, alias, mac, ifindex, mtu, type e state;
- **startup-config**: permette di specificare un file .cfg di configurazione iniziale;

CONTAINERlab prevede anche la creazione automatica di una **management network** al deploy della rete. Essa è usata per gestire i nodi della topologia, permettendo l'accesso SSH, API o altre interfacce di controllo[3]. Essa non è coinvolta nel traffico di forwarding dei pacchetti di rete simulati tra i dispositivi e, di default, prevede l'utilizzo del blocco di indirizzi `170.20.20.0/24`: in particolare, la macchina host acquisirà l'indirizzo `170.20.20.1`.

2.4 Utilizzo di CONTAINERlab nell'applicativo software

Le uniche considerazioni da fare riguardo all'utilizzo di CONTAINERlab riguardano due aspetti principali. In primo luogo, si utilizzano sempre nodi **Arista** per rappresentare i router, ciò che è dovuto all'impiego della libreria `pyeapi`, come descritto nel capitolo successivo. Inoltre, i nodi host sono di una tipologia ridefinita specificamente per questo progetto. Partendo dal **kind** *alpine*, è stata sviluppata una tipologia di container che include nel proprio sistema operativo tutte le librerie necessarie al funzionamento dell'applicativo software, il quale, come ricordato, è stato progettato per essere eseguito su un host della rete emulata.

[3]Sulla macchina host, al deploy della rete, è prevista la creazione automatica delle interfacce destinate alla rete di management e il conseguente aggiornamento delle tabelle di routing.

Capitolo 3

Sviluppo dell'applicazione

3.1 Python Client for eAPI

Come detto nel Capitolo 1, l'obiettivo dell'applicazione è la ricostruzione della topologia di una rete utilizzando il database OSPF di uno specifico router facente parte di essa. In particolare, l'accesso alle informazioni del router avviene per mezzo di **pyeapi**, una libreria Python open-source sviluppata da **Arista Networks** che permette di interagire con dispositivi di rete che eseguono Extendable Operating System (**EOS**): il sistema operativo caratterizzante i router e gli switch Arista.
Pyeapi è stata scelta perché permette di eseguire operazioni EOS in ambiente programmatico, consentendo quindi di automatizzare le operazioni da effettuare nei nodi necessarie all'applicazione. Inoltre, la libreria utilizza **JSON-RPC** per comunicare con i dispositivi di rete, il che rende pyeapi uno strumento flessibile e user-friendly.

3.1.1 Funzione connect()

Per stabilire una connessione con un nodo della rete la libreria pyeapi prevede l'utilizzo della funzione `connect()`. Per funzionare, è necessario specificare indirizzo IP di un'interfaccia del nodo, il protocollo di connessione (generalmente HTTP/HTTPS) e le credenziali di autenticazione. Per poter utilizzare la funzione con successo, è necessario abilitare la gestione dell'API tramite HTTP sui nodi interessati.

3.1.2 Funzione enable()

Una volta stabilita la connessione è possibile iniziare ad eseguire comandi con privilegi elevati utilizzando la funzione `enable()`. Essa richiede soltanto di specificare il comando desiderato e ritorna una risposta in formato JSON.

```
1  import pyeapi
2
3  # comandi da eseguire sul nodo target con privilegi amministrativi
4  # >management api http-commands
5  # >no shutdown
6
7  node = pyeapi.connect(
8      transport='https',
9      host='192.168.1.1',
10     username='admin',
11     password='password'
12 )
13
14 response = node.enable("show version")
```

Figure 4: Struttura della classe Node

3.2 Struttura del codice

La struttura delle directory del progetto è la seguente[4]:

```
progetto/
├── src/
│   ├── main.py
│   ├── utilities.py
│   ├── hostname/
│   │   └── get_hostname.py
│   ├── interfaces/
│   │   └── get_interfaces.py
│   ├── protocol/
│   │   └── protocol_info.py
│   ├── route_table/
│   │   └── get_route_table.py
│   ├── lsa_1/
│   │   └── router_lsa.py
│   ├── lsa_2/
│   │   └── network_lsa.py
│   ├── lsa_3/
│   │   └── summary_lsa.py
│   ├── lsa_4/
│   │   └── asbr_summary_lsa.py
│   ├── lsa_5/
│   │   └── external_lsa.py
│   └── gui/
│       ├── gui.py
│       ├── static/
│       │   ├── css/
│       │   │   └── vis-network.min.css
│       │   ├── js/
│       │   │   └── vis-network.min.js
│       │   └── images/
│       │       ├── router.png
│       │       ├── switch.png
│       │       └── monitor.png
│       └── templates/
│           └── index.html
├── ospf_mgmt_app.sh
├── venv/
└── README.md
```

3.3 Classi definite in utilities.py

All'interno del file `utilities.py` sono state definite le classi utili allo sviluppo dell'applicativo software.[5]

[4]Sono omessi i dettagli dell'ambiente virtuale

[5]Anche se non mostrato, ogni classe è fornita di un metodo __str__(self) ed un metodo toJSON(self). Inoltre, è dotata di un insieme di metodi per l'aggiunta di elementi ai rispettivi campi.

3.3.1 Classe Node

La classe `Node` (Fig. 5) rappresenta un nodo della rete. Esso, agli scopi dell'applicazione, è caratterizzato da un hostname, un OSPF-id, un insieme di interfacce, una tabella di routing e un elenco di neighbor OSPF.

```
class Node:
    def __init__(self, router_id, hostname, interface_list=None,
                 neighbor_list=None, route_table=None):
        self.hostname = hostname
        self.router_id = router_id
        self.interfaces = interface_list if interface_list else []
        self.neighbors = neighbor_list if neighbor_list else []
        self.route_table = route_table if route_table else []

    def add_interface(self, id, ip, masklen, interface_status,
       line_protocol_status):
        self.interfaces.append({
            "id": id,
            "ip": ip,
            "masklen": masklen,
            "interface_status": interface_status,
            "line_protocol_status": line_protocol_status
        })

    def add_neighbor(self, interface_id, neighbor_router_id,
       neighbor_ip_addr,
                     adjacency_state, designated_router,
                        backup_designated_router):
        self.neighbors.append({
            "interface_id": interface_id,
            "router_id": neighbor_router_id,
            "neighbor_ip_addr": neighbor_ip_addr,
            "adjacency_state": adjacency_state,
            "designated_router": designated_router,
            "backup_designated_router": backup_designated_router
        })

    def add_route(self, ip, masklen, via, interface, protocol):
        self.route_table.append({
            "ip": ip,
            "masklen": masklen,
            "via": via,
            "interface": interface,
            "protocol": protocol
        })
```

Figure 5: Struttura della classe Node

3.3.2 Classe Link

La classe `Link` (Fig. 6) modella una sottorete facente parte della topologia. Come è lecito aspettarsi, è caratterizzata da un indirizzo IP e la relativa maschera, oltre che da un insieme di endpoints. Inoltre, se il link è ad **accesso multiplo**, sono specificati anche Designated Router e Backup Designated Router.

```
class Link:
    def __init__(self, id, type, options, metric, mask=None,
                 endpoints=None, dr=None, bdr=None):
        self.id = id
        self.type = type
        self.options = options
        self.metric = metric
        self.endpoints = endpoints
        self.mask = mask
        self.dr = dr
        self.bdr = bdr
```

Figure 6: Struttura della classe Link

3.3.3 Classe Area

La classe `Area` (Fig. 7) rappresenta un'area OSPF interna alla topologia. È composta da un insieme di nodi e link, rappresentati rispettivamente dagli oggetti delle classi `Node` e `Link`. Inoltre, include un elenco di istanze delle classi `Route` e `Path_To_ASBR`, utilizzato per memorizzare informazioni **inter-area**. [6]

3.3.4 Classe Network

Infine, l'intera della topologia della rete è modellata utilizzando la classe `Network` (Fig. 10). Essa è formata da un insieme di aree (oggetti dell classe `Area`) e da un elenco di rotte **inter-AS**, con lo scopo di memorizzare le informazioni per raggiungere Autonomous System esterni.

```
class Area:
    def __init__(self, area_id):
        self.area_id = area_id
        self.nodes = set()
        self.links = set()
        self.ospf_inter_area_routes = set()
        self.paths_to_asbrs = set()
```

Figure 7: Struttura della classe Area

3.4 Funzioni getter

All'interno della struttura del codice si osservano numerose directory contenenti un unico file `.py` (ad esempio, `get_interfaces.py`, `protocol_info.py`, `router_lsa.py`, ecc.), ognuno dei

[6] La classe Route rappresenta una rotta in una tabella di routing, definita da un indirizzo IP di destinazione, una maschera, un next-hop, un valore di metrica e un tipo di metrica. La classe Path_To_ASBR rappresenta un percorso verso un AS Border Router, specificando l'ASBR di destinazione, il next-hop e la metrica associata.

```
class Network:
    def __init__(self):
        self.areas = set()
        self.external_routes = set()
```

Figure 8: Struttura della classe Network

quali implementa una singola funzione dedicata all'estrazione delle informazioni corrispondenti al nome del file. Data la frequenza delle chiamate alle `pyeapi`, questa suddivisione si è rivelata particolarmente conveniente, migliorando la leggibilità del codice e mantenendo separata l'elaborazione dell'oggetto JSON restituito.

3.5 Il file principale: main.py

Inizialmente, viene stabilita una connessione con il nodo target. Successivamente, si procede con la costruzione dell'oggetto `Node`, estraendo le informazioni dal nodo stesso. Tali informazioni, come illustrato nella Sezione 3.3.1, comprendono le interfacce di rete, la tabella di routing e i dati relativi ai vicini OSPF. Succcessivamente, viene instanziato un oggetto della classe `Network`, inizialmente vuoto.

```
target_node = pyeapi.client.connect(
    transport='https',
    host=input_node,
    username='admin',
    password='admin',
    return_node=True
)
hostname = get_hostname(target_node)
interfaces = get_interfaces(target_node)
route_table = get_route_table(target_node)
protocol_info = get_protocol_info(target_node)
neighbors = get_neighbors(target_node)

router = Node(protocol_info['Router ID'], hostname, interfaces,
   neighbors, route_table)
```

Figure 9: Sezione iniziale di main.py

3.5.1 Router LSA

Per la ricostruzione della topologia, il primo passo consiste nell'estrazione degli **LSA-1**, ossia i router LSA. L'oggetto JSON ricevuto dal router è strutturato come un dizionario annidato con una lista interna[7], i cui elementi sono identificato primariamente dall'**id** di una specifica **area OSPF**. Di conseguenza, per ogni iterazione del ciclo `for` che analizza l'oggetto, viene istanziato un'`Area`. Internamente, utilizzando la chiave **areaDatabase** è possibile accedere ad una lista di LSA, in cui ogni elemento è strutturato a dizionario. Il **link-state-id** di ciascun LSA, che identifica il nodo creatore dell'annuncio, viene utilizzato per aggiungere un nuovo nodo all'oggetto `Area` appena creato. Successivamente, è necessario identificare le **sotto-reti**

[7]Poiché l'oggetto JSON ricevuto dal router ha una struttura identica per ogni tipo di LSA, la sua descrizione sarà omessa nei paragrafi dedicati all'ottenimento degli LSA successivi.

annunciate nel LSA. Per tale scopo, si verifica se il collegamento è già memorizzato all'interno dell'oggetto `Area`: nel caso in cui il link non sia presente, viene istanziato un nuovo oggetto della classe `Link` e aggiunto all'oggetto `Area`; in caso contrario, l'attributo **endpoints** dell'oggetto `Link` viene aggiornato includendo il nuovo `link-state-id`. Infine, qualora la sotto-area annunciata sia una **stub network**, il campo `link-data` viene utilizzato per assegnare al link l'attributo **mask**. (Fig. 10)

3.5.2 Network LSA

Al fine di compilare tutti i campi previsti dalle strutture dati relative alle sottoreti interne alle aree, è necessario richiedere al router target anche gli **LSA-2**. L'oggetto JSON, strutturato come nel caso precedente, contiene le informazioni insieme di aree OSPF. Per ognuna di esse è presente una lista di LSA. All'interno di questi LSA sono presenti informazioni riguardo alla **mask** e agli **attached router**, oltre che al **designated router** e al **backup designated router**. (Fig. 11)

```
router_lsa_1 = get_router_lsa_info(target_node)

for area_data in router_lsa_1:
    new_area = Area(area_data)

    for area_db_entry in router_lsa_1[area_data]['areaDatabase']:
        for lsa_entry in area_db_entry['areaLsas']:
            link_state_id = lsa_entry['linkStateId']
            advertising_router = lsa_entry['advertisingRouter']
            new_area.add_node(link_state_id)

            for router_link in lsa_entry['ospfRouterLsa']['
               routerLsaLinks']:
                link_id = router_link['linkId']
                link_type = router_link['linkType']
                metric = router_link['metric']

                existing_link = None
                for link in new_area.links:
                    if link.id == link_id and link.type == link_type:
                        existing_link = link
                        break

                if existing_link:
                    existing_link.add_endpoint(link_state_id)
                else:
                    new_link = Link(link_id, link_type, None, metric,
                       link_state_id)
                    if(link_type == "stubNetwork"):
                        new_link.set_mask(router_link['linkData'])

                    new_area.add_link(new_link)

    network_topology.add_area(new_area))
```

Figure 10: Recupero informazioni LSA-1

```
network_lsa_2 = get_network_lsa_info(target_node)

for area_id, area_info in network_lsa_2.items():
    target_area = network_topology.find_target_area(area_data)
    if not target_area:
        continue

    for area_db_entry in area_info['areaDatabase']:
        for lsa_entry in area_db_entry['areaLsas']:
            link_state_id = lsa_entry['linkStateId']
            network_mask = lsa_entry['ospfNetworkLsa']['networkMask']
            dr = lsa_entry['advertisingRouter']
            attached_routers = lsa_entry['ospfNetworkLsa']['
               attachedRouters']
            bdr = attached_routers[1] if len(attached_routers) > 1
               else None

            link = next((l for l in target_area.links if l.id ==
               link_state_id), None)
            if link:
                link.set_mask(network_mask)
                link.set_dr_bdr(dr, bdr)
```

Figure 11: Recupero informazioni LSA-2

3.5.3 Summary LSA

Le informazioni sulle rotte **inter-area** sono contenute negli **LSA di tipo 3**. Per ciascuna area a cui appartiene il nodo target, sono presenti diversi LSA che descrivono le rotte verso sottoreti esterne all'area stessa. Ogni LSA include le seguenti informazioni come **indirizzo IP** della sottorete di destinazione, **mask** associata, **router annunciante** (via) e **metrica** del percorso. (Fig. 12)

```
summary_lsa_3 = get_summary_lsa_info(target_node)

for area_data in summary_lsa_3:
    target_area = network_topology.find_target_area(area_data)
    if not target_area:
        continue

    for area_db_entry in summary_lsa_3[area_data]['areaDatabase']:
        for lsa_entry in area_db_entry['areaLsas']:
                ip = lsa_entry['linkStateId']
                mask = lsa_entry['ospfSummaryLsa']['networkMask']
                via = lsa_entry['advertisingRouter']
                metric = lsa_entry['ospfSummaryLsa']['metric']

                route = Route(ip, mask, via, metric)

                target_area.add_inter_area_route(route)
```

Figure 12: Recupero informazioni LSA-3

3.5.4 ASBR summary LSA

I percorsi verso router **ASBR** costiuiscono un elemento fondamentale nella ricostruzione della topologia di rete, per ottenere informazioni su di essi si utilizzano **LSA-4**. La struttura del codice è simile al caso degli LSA-3: l'unica differenza consiste nell'assenza di una maschera, dato che con gli ASBR Summary LSA viene pubbblicizzato uno specifico nodo e non una sottorete. (Fig. 13)

3.5.5 External LSA

Infine è necessario ricavare le rotte esterne. Per farlo, si utilizzano gli **LSA-5**. La struttura del codice è nuovamente simile a dei casi precedenti, ma senza che sia necessario effettuare una discriminazione iniziale dell'oggetto JSON in basealle aree, dato che gli LSA-5 sono unici all'interno dell'autnomous system. (Fig. 13)

```
# recupero LSA tipo 4
asbr_summary_lsa_4 = get_asbr_summary_lsa_info(target_node)

for area_data in asbr_summary_lsa_4:
    target_area = network_topology.find_target_area(area_data)

    if not target_area:
        continue

    for area_db_entry in asbr_summary_lsa_4[area_data]['areaDatabase']:
        for lsa_entry in area_db_entry['areaLsas']:
                asbr = lsa_entry['linkStateId']
                via = lsa_entry['advertisingRouter']
                metric = lsa_entry['ospfSummaryLsa']['metric']

                path = Path_To_ASBR(asbr, via, metric)

                target_area.add_path_to_asbr(path)

# recupero LSA di tipo 5
external_lsa_5 = get_external_lsa_info(target_node)

for external_data in external_lsa_5:
    for lsa in external_data['externalLsas']:
        ip = lsa['linkStateId']
        mask = lsa['ospfExternalLsa']['networkMask']
        via = lsa['advertisingRouter']
        metric = lsa['ospfExternalLsa']['metric']
        metric_type = lsa['ospfExternalLsa']['metricType']

        route = Route(ip, mask, via, metric, metric_type)

        network_topology.add_external_network(route)
```

Figure 13: Recupero informazioni LSA-4 e LSA-5

3.5.6 Completare la lista dei nodi

Per completare la topologia, è possibile sfruttare la libreria `pyeapi` per accedere anche agli altri nodi della rete. Fino a questo momento, infatti, le uniche informazioni disponibili sui nodi della rete, ad eccezione del nodo target, sono l'**id OSPF** e gli indirizzi IP delle interfacce appartenenti alle sottoreti di cui fa parte anche il nodo target. Questi indirizzi IP possono essere utilizzati per accedere ai router attraverso `pyeapi` e ottenere informazioni più dettagliate. In particolare, vengono richieste ai router le seguenti informazioni: interfacce, tabelle di routing e **vicini OSPF**. Sfruttando quest'ultima, è possibile ottenere informazioni su nodi che non sono direttamente vicini al router target, completando così la lista dei nodi della topologia. Il codice relativo a questo processo è mostrato nella Figura 14. Come si può osservare, la lista dei nodi è inizialmente composta solo dal router target. All'interno del ciclo `while`, il primo router della coda (nella prima iterazione il target) viene prelevato e la lista dei vicini viene esaminata: se un neighbor trovato non è già presente tra i nodi conosciuti, viene creato un nuovo nodo che viene inserito nella coda, il cui insieme di vicini verrà analizzato nelle iterazioni successive.

```python
def discover_router(ip_addr):
    node = pyeapi.client.connect(
        transport='https',
        host=ip_addr,
        username='admin',
        password='admin',
        return_node=True
    )

    hostname = get_hostname(node)
    interfaces = get_interfaces(node)
    route_table = get_route_table(node)
    protocol_info = get_protocol_info(node)
    neighbors = get_neighbors(node)

    router_id = protocol_info['Router ID']

    return Node(router_id, hostname, interfaces, neighbors,
       route_table), neighbors

# Mappa router_id -> Node
network_routers = {router.router_id: router}
# Mappa router_id -> IP dei vicini
discovered_ips = {router.router_id: router.neighbors}

queue = Queue()
queue.put(router)

while not queue.empty():
    current_router = queue.get()

    for nghb in current_router.neighbors:
        nghb_id = nghb['router_id']
        nghb_ip = nghb['neighbor_ip_addr']

        if nghb_id not in network_routers:
            new_router, new_neighbors = discover_router(nghb_ip)
            network_routers[nghb_id] = new_router
            queue.put(new_router)
```

Figure 14: Completamento della lista dei nodi

Capitolo 4

Interfaccia grafica

4.1 Flask

Per l'implementazione dell'interfaccia grafica è stato adottato un approccio **web-based**, scelto per la flessibilità offerta dal sistema di gestione degli eventi nativo di **JavaScript** e per la vasta disponibilità di librerie open-source, che rendono lo sviluppo più efficiente e versatile. Il lato **server** dell'applicativo web è gestito utilizzando `Flask`, un microframework per lo sviluppo di applicazioni web in Python. Flask si confa alla perfezione alle esigenze necessarie allo sviluppo dell'applicazione. Esso infatti offre le funzionali essenziali per sviluppare un server-web senza imporre vincoli rigidi al programmatore, rendendosi dunque adatto sia a progetti minimali come il software di network management descritto all'interno di questa trattazione o, data la sua nativa estensibilità, ad applicativi più complessi. Inoltre, supportando **Jinja2**[8] per la creazione dinamica delle pagine, Flask si combina perfettamente anche con tecnologie frontend.

Figure 15: Logo di Flask

4.1.1 Sviluppo di gui.py

Il codice contenuto in `gui.py` definisce una semplice applicazione web Flask.
In particolare, dopo la creazione dell'istanza Flask `app` (che sarà il server-web vero e proprio) viene definita la route principale (Fig. 16). Inoltre, vengono recuperate tre variabili dal campo `config` della variabile `app` e li passa al template `index.hmtl` utilizzando la funzione **render_template()**. Esse, nel dettaglio, consistono in:

- **target**: nodo target della rete;
- **data**: ricostruzione della topologia attraverso LSA.
- **network_routers**: informazioni dettagliate sui nodi.

La funzione **render_template()** serve a renderizzare un template HTML dinamicamente, sostituendo i **placeholder** con i valori che vengono passati. I placeholder sono racchiusi tra doppie parentesi graffe {{ }} per le variabili, e tra {% %} per le strutture di controllo come i cicli o le condizioni.
All'interno del file `main.py` il server flask viene avviato su richiesta dell'utente (maggiori dettagli saranno forniti nel capitolo successivo) secondo le modalità illustrate nella Figura 17.

[8] Jinja2 è un motore di template scritto in Python, utilizzato per generare contenuti dinamici in applicazioni web.

```python
from flask import Flask, render_template
import json

app = Flask(__name__)

@app.route('/')
def index():
    target = app.config.get('TARGET')
    data = app.config.get('ENC_DATA')
    network_routers = app.config.get('NETWORK_ROUTERS_JSON')

    return render_template('index.html', target, data,
        network_routers)
```

Figure 16: Struttura di gui.py

Nel dettaglio, il server web viene eseguito su un **thread separato** ascoltando l'indirizzo IP 0.0.0.0 e la **porta 5000**. Esso è quindi raggiungibile tramire l'indirizzo IP di una qualsiasi interfaccia presente sulla macchina in cui è eseguito l'applicativo di management. Nello specifico, la macchina host su cui è emulata la rete portà usufruire dell'interfaccia grafica utilizzando l'indirizzo IP relativo alla rete di management creata da CONTAINERlab.

```python
from gui.gui import app
def run_flask():
    app.run(host='0.0.0.0', port=5000, debug=True, use_reloader=False)

# [...]

network_routers_json = {router_id: json.loads(router.toJSON()) for
   router_id, router in network_routers.items()}
data = network_topology.toJSON()

app.config['NETWORK_ROUTERS_JSON'] = network_routers_json
app.config['DATA'] = data
app.config['TARGET'] = input_node

flask_thread = Thread(target=run_flask, daemon=True)
flask_thread.start()
```

Figure 17: Avvio del server Flask in main.py

4.2 Libreria vis.js

Per lo sviluppo **frontend** dell'applicazione è stata utilizzata la liberia `vis.js`. Essa è una libreria JavaScript open-source utilizzata per la visualizzazione di dati dinamici, come **grafi**, **reti**, timeline e grafici interattivi. La libreria è composta dai componenti *DataSet*, *Timeline*, *Network*, *Graph2d* e *Graph3d*. La componente centrale nello sviluppo dell'interfaccia grafica è stata *Network*.

4.2.1 Componente Network

Questa parte della libreria è progettata per la **visualizzazione** e l'**interazione** con reti di nodi e archi. Per creare una rete è necessario definire due oggetti: `nodes` e `edges`, rispettivamente un array di nodi ed un array di archi. Ogni nodo può essere rappresentato come un oggetto con varie proprietà come **id**, **label**, **color**, **shape**, **image**, ecc., mentre ogni arco avrà proprietà come **from**, **to**, **label**, **length**, ecc. Oltre alle proprietà riportare, è possibile definre delle opzioni relative alla rete nella sua totalità per configuararne il comportamento visivo e interattivo.
La libreria offre infine vari eventi per interagire con la rete, come **click**, **doubleClick**, **selectNode**, **selectEdge**, e molti altri. Questi eventi saranno usati per fornire all'utente maggiore dinamcità e interattività nell'utilizzo dell'interfaccia grafica.
Lo sviluppo di una semplice rete sfruttando le strutture messe a disposizione da vis.js è riportato nella Figura 18.

```
let nodes = new vis.DataSet([
  { id: 1, label: 'Nodo 1' },
  { id: 2, label: 'Nodo 2' },
  { id: 3, label: 'Nodo 3' }
]);

let edges = new vis.DataSet([
  { from: 1, to: 2 },
  { from: 2, to: 3 }
]);

let container = document.getElementById("network-container");
let data = { nodes: nodes, edges: edges };
let options = {
  physics: {
    enabled: true,
    barnesHut: {
      gravitationalConstant: -2000,
      springLength: 100
    }
  },
  edges: {
    color: { inherit: false, color: 'blue' },
    width: 2
  },
  interaction: {
    hover: true,
    tooltipDelay: 1000
  }
};

let network = new vis.Network(container, data, options);
network.on('click', (event) => {
  alert('Nodi: ' + event.nodes + ' Archi: ' + event.edges);
});
```

Figure 18: Utilizzo di vis-network

4.3 Sviluppo del template: index.html

All'interno del template `index.hmtl`, la libreria vis.js è utilizzata per la visualizzare ed interagire con la topologia di rete ricostruirita secondo le modalità descritte nel capitolo 4.

4.3.1 Body HTML

La struttura del corpo della pagina è piuttosto semplice e si compone esclusivamente di quattro `<div>` che svolgono la funzione di container: uno per la **topologia di rete**, uno per la **legenda delle aree**, uno per le **rotte esterne** e uno per le informazioni generali che vengono aggiornate dinamicamente in base agli eventi `onclick()` su nodi e archi.

4.3.2 Script di index.html

All'interno dell'elemento `script`, le variabili vengono recuperate utilizzando i placeholder con la sintassi prevista da Jinja2 (Fig. 19)[9].

```
1 let router = "{{ target|safe }}";
2 let networkData = {{ data|safe }};
3 let networkRouters = {{ network_routers|safe }};
```

Figure 19: Recupero delle variabili passate con `render_template()`

Inoltre, all'inizio della porzione di script vengono inizializzate le seguenti variabili e strutture dati (Fig. 20):

- **nodes**: un array inizialmente vuoto, destinato a contenere i nodi per vis.js;
- **edges**: un array inizialmente vuoto, destinato a contenere gli archi per vis.js;
- **colors**: un array di 8 elementi, ciascuno rappresentante un colore utilizzato per differenziare i link delle varie aree OSPF;
- **routerImage**: il percorso dell'immagine di un router;
- **switchImage**: il percorso dell'immagine di uno switch;
- **monitorImage**: il percorso dell'immagine di un monitor.

La prima variabile analizzata è `networkData`, che contiene un oggetto JSON ottenuto applicando il metodo `toJSON()` all'oggetto `network_topology` presente nel file `main.py`. Poiché tale oggetto è suddiviso in diverse aree, l'analisi viene effettuata separatamente per ciascuna di esse. Durante ogni iterazione, il primo passo consiste nell'aggiornare la **legenda** delle aree, rappresentata da un elemento `<div>` posizionato nella parte superiore della pagina. Questo elemento contiene un elenco che associa gli ID delle aree ai rispettivi colori di rappresentazione. (Fig. 21) Su questi elementi `<div>` è inoltre definito un evento `onclick()`, al quale è associata una routine che aggiorna il container delle informazioni generali. In particolare, tale routine permette di visualizzare le rotte **inter-area** e i percorsi verso gli **ASBR** (Risultato visivo in Fig. 22, codice in Fig. 23).
Ogni area è caratterizzata da una lista di **nodi** e una lista di **link**. Scorrendo l'elenco dei nodi, si creano semplicemente nuovi oggetti `node` di vis.js, utilizzando gli attributi specificati nella Figura 24. L'elaborazione della lista dei link è più articolata e prevede tre casi distinti:

[9]Utilizzando il filtro `|safe`, Jinja2 interpreta la variabile come HTML o JavaScript, anziché come semplice testo.

```
        const colors = [
            "#0000FF",  "#FF0000", // blu, rosso
            "#00FF00", "#FFFF00", // verde, giallo
            "#00FFFF", "#FF00FF", // ciano, magenta
            "#FFA500", "#800080"  // arancione, viola
        ];

        let routerImage = "../static/images/router.png";
        let switchImage = "../static/images/switch.png";
        let monitorImage = "../static/images/monitor.png";

        let nodes = [];
        let edges = [];
```

Figure 20: Variabili e strutture dati utili nello script di index.html

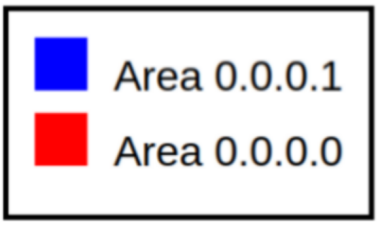

Figure 21: Legenda aree

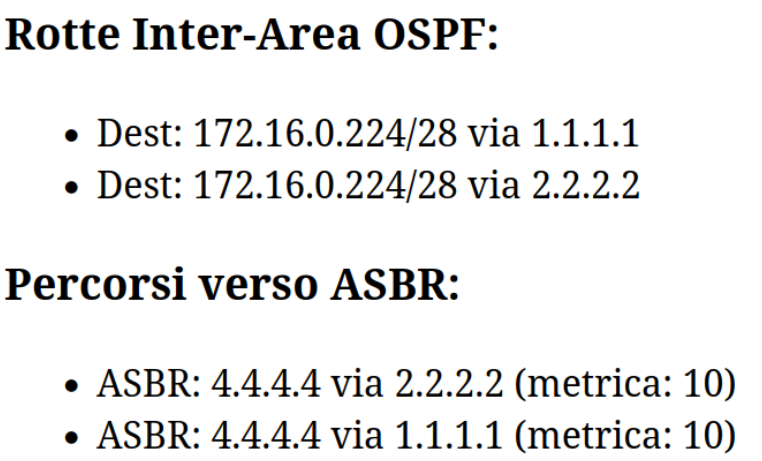

Figure 22: Risultato visivo del click su un elemento della legenda

- **Singolo endpoint** (stub network): si crea un nodo `pc` e un nodo `switch`, insieme a due oggetti `link`, che collegano rispettivamente il pc allo switch e lo switch al nodo indicato come endpoint;
- **Doppio endpoint**: viene generato un oggetto `link` che collega direttamente i due endpoint, senza la necessità di creare ulteriori nodi;
- **Tre o più endpoint**: si introduce un nodo `switch`, collegato agli endpoint tramite oggetti `link`, per riflettere la presenza di un link ad accesso multiplo nella rete.

Inoltre, vengono mantenuti due contatori distinti, uno per i PC e uno per gli switch, al fine di assegnare a ciascun nodo un identificativo univoco. In particolare, l'`id` di ogni PC viene generato concatenando la stringa *pc* al valore del rispettivo contatore, mentre per gli switch viene utilizzata la stringa *switch* seguita dal relativo contatore (Fig. 25 e Fig. 26).
Successivamente, vengono gestite le informazioni relative alle rotte esterne e visualizzate all'interno di un elemento `<div>` con attributo `id` impostato su *external-routes*.
Infine, viene istanziato l'oggetto `network`, associandogli il container di riferimento, i nodi e gli archi precedentemente costruiti, oltre alle relative opzioni[10].

[10]Le opzioni scelte mirano a ridurre al minimo la mobilità del grafo, garantendo leggibilità e immediatezza.

```
networkData.areas.forEach(function(area, index) {
    const areaColor = colors[index % colors.length];

    const legendDiv = document.getElementById('legend');
    const areaLegend = document.createElement('div');
    areaLegend.className = 'areaLegend';

    const colorBox = document.createElement('span');
    colorBox.className = 'colorBox';
    colorBox.style.backgroundColor = areaColor;

    const areaName = document.createElement('span');
    areaName.textContent = "Area " + area.area_id;

    areaLegend.appendChild(colorBox);
    areaLegend.appendChild(areaName);

    areaLegend.addEventListener('click', () => {
        const infoDiv = document.getElementById('info');
        infoDiv.innerHTML = "";

        if (area.ospf_inter_area_routes.length > 0) {
            const interAreaTitle = document.createElement('h3');
            interAreaTitle.textContent = 'Rotte Inter-Area OSPF:';
            infoDiv.appendChild(interAreaTitle);

            const interAreaList = document.createElement('ul');
            area.ospf_inter_area_routes.forEach(route => {
                const listItem = document.createElement('li');
                listItem.textContent = `Dest: ${route.ip}/${
                    find_subnet_length(route.mask)} via ${route.via}`;
                interAreaList.appendChild(listItem);
            });
            infoDiv.appendChild(interAreaList);
        }

        if (area.paths_to_asbrs.length > 0) {
            const asbrTitle = document.createElement('h3');
            asbrTitle.textContent = "Percorsi verso ASBR:";
            infoDiv.appendChild(asbrTitle);

            const asbrList = document.createElement('ul');
            area.paths_to_asbrs.forEach(path => {
                const listItem = document.createElement('li');
                listItem.textContent = `ASBR: ${path.asbr} via ${path.
                    via} (metrica: ${path.metric})`;
                asbrList.appendChild(listItem);
            });
            infoDiv.appendChild(asbrList);
        }
    });
    legendDiv.appendChild(areaLegend);
```

Figure 23: Codice relativo alla Fig. 21 e Fig. 22

```
area.nodes.forEach(function(node) {
    if (!nodes.some(n => n.id === node)) {
        nodes.push({
            id: node,
            label: node,
            image: routerImage
        });
    }
});
```

Figure 24: Creazione degli oggetti `node`

```
area.links.forEach(function(link) {
    let endpoints = link.endpoints;

    if (endpoints.length >= 3) {
        let switchId = "switch_" + link.id;

        if (!nodes.some(n => n.id === switchId)) {
            nodes.push({
                id: switchId,
                label: "SW" + swCounter,
                image: switchImage
            });
        }
        swCounter++;

        for (let i = 0; i < endpoints.length; i++) {
            edges.push({
                from: switchId,
                to: endpoints[i],
                label: link.id + " / "+ find_subnet_length(link.mask),
                title: link.id,
                color: { color: areaColor }
            });
        }
    } else {
        let endpoint1 = endpoints[0];
        let endpoint2 = endpoints[1];

        edges.push({
            from: endpoint1,
            to: endpoint2,
            label: link.id + " / " + find_subnet_length(link.mask),
            title: link.id,
            arrows: "none",
            color: { color: areaColor }
        });
    }
    // Continua nella Fig. 25
```

Figure 25: Creazione degli oggetti `link`: caso singolo endpoint

```
    else if (endpoints.length === 1){
        let new_node;

        new_node = 'PC' + pcCounter;
        pcCounter++;

        nodes.push({
            id: new_node,
            label: new_node,
            image: monitorImage
        });

        let switchId = "switch_" + link.id;
        nodes.push({
            id: switchId,
            label: "SW" + swCounter,
            image: switchImage
        });
        swCounter++;

        edges.push({
            from: new_node,
            to: switchId,
            label: link.id + " / " + find_subnet_length(link.mask),
            title: link.id,
            arrows: 'none',
            color: { color: areaColor }
        });

        edges.push({
            from: switchId,
            to: endpoints[0],
            label: link.id + " / " + find_subnet_length(link.mask),
            title: link.id,
            arrows: 'none',
            color: { color: areaColor }
        });
    }
```

Figure 26: Creazione degli oggetti `link`: caso due e tre, o più, endpoint

4.3.3 Eventi definiti su nodi ed archi

Come detto in 4.2.1, è possibile definire eventi per permettere una maggiore interazione dell'utente con la topologia. In particolare, sono definiti eventi `hoverNode` e `blurNode`, `hoverEdge` e `blurEdge`, per modificare il cursore, impostandolo in modalità *pointer* esclusivamente quando si trova sopra un nodo o un arco. (Fig. 27)
Inoltre, vengono definiti eventi `onclick` sul grafo. Se l'elemento cliccato è un arco, nel `div` dedicato alle informazioni generiche vengono mostrate le caratteristiche del link selezionato (Fig, 28). Se invece viene cliccato un nodo, nello stesso elemento vengono visualizzate le informazioni relative ad esso (Fig, 29).

```
network.on("hoverNode", () => { container.style.cursor = "pointer"; })

network.on("blurNode", () => { container.style.cursor = "default"; })

network.on("hoverEdge", () => { container.style.cursor = "pointer"; })

network.on("blurEdge", () => { container.style.cursor = "default"; })
```

Figure 27: Gestione del cursore

```
network.on("click", (params) => {
    if (params.edges.length > 0 && params.nodes.length === 0) {
        let edgeId = params.edges[0];
        let targetEdge = edges.find(e => e.id === edgeId);

        let edge = null;
        for (let area of networkData.areas) {
            edge = area.links.find(link => link.id === targetEdge.
               label.split(' /')[0]);
            if (edge) break;
        }

        if (!edge) {
            console.warn("Nessun link trovato con: ",
                targetEdge.label);
            return;
        }

        let infoDiv = document.getElementById("info");
        infoDiv.innerHTML = `
            <h3>Dettagli Link</h3>
            <p><strong>IP:</strong> ${edge.id}</p>
            <p><strong>Mask:</strong> ${edge.mask}</p>
            <p><strong>Tipo:</strong> ${edge.type}</p>
            <p><strong>Metric:</strong> ${edge.metric}</p>
            ${edge.dr ? `
                <p><strong>Designated Router:</strong>
                ${edge.dr}</p>` : ""}
            ${edge.bdr ?
                `<p><strong>Backup Designated Router:</strong>
                ${edge.bdr}</p>` : ""}
            <p>
                <strong>Endpoints:</strong>
                ${edge.endpoints.join(", ")}
            </p>
        `;
    }
// Continua nella figura 28
```

Figure 28: Eventi on click sugli archi

```
else if (params.nodes.length > 0) {
    let nodeId = params.nodes[0];
    let targetNode = nodes.find(n => n.id === nodeId);
    let node = networkRouters[targetNode.id];

    if (node === undefined) {
        console.warn("I nodi non router non possono essere scelti");
        return;
    }

    let infoDiv = document.getElementById('info');

    infoDiv.innerHTML = `
        <h3>Hostname: ${node.hostname}</h3>
        <p><strong>Router ID:</strong> ${node.router_id}</p>

        <h4>Interfaces:</h4>
        <ul>
            ${node.interfaces.map(iface => `
                <li>
                    <strong>${iface.name}</strong><br>
                    <strong>IP:</strong> ${iface.ip}<br>
                    <strong>Masklen:</strong> ${iface.masklen}<br>
                    <strong>Status:</strong>
                        ${iface.interface_status}<br>
                    <strong>Line Protocol:</strong>
                        ${iface.line_protocol_status}
                </li>
            `).join('')}
        </ul>

        <h4>Neighbors:</h4>
        <ul>
            ${node.neighbors.map(neighbor => `
                <li>
                    <strong>Router ID:</strong> ${neighbor.router_id}
                        <br>
                    <strong>Interface:</strong> ${neighbor.interface}
                        <br>
                    <strong>Neighbor IP:</strong>
                        ${neighbor.neighbor_ip_addr}<br>
                    <strong>Adjacency State:</strong>
                        ${neighbor.adjacency_state}
                </li>
            `).join('')}
        </ul>

        [...]`

// la medesima logica e' usata per le informazioni della route table
}
```

Figure 29: Eventi on click sugli archi

Hostname: RA

Router ID: 1.1.1.1

Interfaces:

- **Ethernet1 IP:** 172.16.0.225 **Masklen:** 28 **Status:** connected **Line Protocol:** up
- **Ethernet2 IP:** 172.16.0.241 **Masklen:** 30 **Status:** connected **Line Protocol:** up
- **Ethernet3 IP:** 172.16.0.5 **Masklen:** 30 **Status:** connected **Line Protocol:** up
- **Loopback0 IP:** 1.1.1.1 **Masklen:** 32 **Status:** connected **Line Protocol:** up
- **Management0 IP:** 172.20.20.5 **Masklen:** 24 **Status:** connected **Line Protocol:** up

Neighbors:

- **Router ID:** 2.2.2.2 **Interface:** Ethernet1 **Neighbor IP:** 172.16.0.226 **Adjacency State:** full
- **Router ID:** 4.4.4.4 **Interface:** Ethernet1 **Neighbor IP:** 172.16.0.227 **Adjacency State:** full
- **Router ID:** 3.3.3.3 **Interface:** Ethernet2 **Neighbor IP:** 172.16.0.242 **Adjacency State:** full

Routing Table:

- **IP:** 0.0.0.0 **Masklen:** 0 **Via:** 172.16.0.227 **Interface:** Ethernet1 **Protocol:** ospfExternalType2
- **IP:** 1.1.1.1 **Masklen:** 32 **Via:** Directly Connected **Interface:** Loopback0 **Protocol:** connected
- **IP:** 172.16.0.4 **Masklen:** 30 **Via:** Directly Connected **Interface:** Ethernet3 **Protocol:** connected
- **IP:** 172.16.0.8 **Masklen:** 30 **Via:** 172.16.0.242 **Interface:** Ethernet2 **Protocol:** OSPF
- **IP:** 172.16.0.12 **Masklen:** 30 **Via:** 172.16.0.242 **Interface:** Ethernet2 **Protocol:** OSPF
- **IP:** 172.16.0.224 **Masklen:** 28 **Via:** Directly Connected **Interface:** Ethernet1 **Protocol:** connected
- **IP:** 172.16.0.240 **Masklen:** 30 **Via:** Directly Connected **Interface:** Ethernet2 **Protocol:** connected
- **IP:** 172.16.0.244 **Masklen:** 30 **Via:** 172.16.0.242 **Interface:** Ethernet2 **Protocol:** OSPF
- **IP:** 172.20.20.0 **Masklen:** 24 **Via:** Directly Connected **Interface:** Management0 **Protocol:** connected

Figure 30: Dettagli di un nodo

Dettagli Link

IP: 172.16.0.244

Mask: 255.255.255.252

Tipo: transitNetwork

Metric: 10

Designated Router: 3.3.3.3

Backup Designated Router: 2.2.2.2

Endpoints: 2.2.2.2, 3.3.3.3

Figure 31: Dettagli di un link transit-network

Dettagli Link

IP: 172.16.0.4

Mask: 255.255.255.252

Tipo: stubNetwork

Metric: 10

Endpoints: 1.1.1.1

Figure 32: Dettagli di un link stub-network

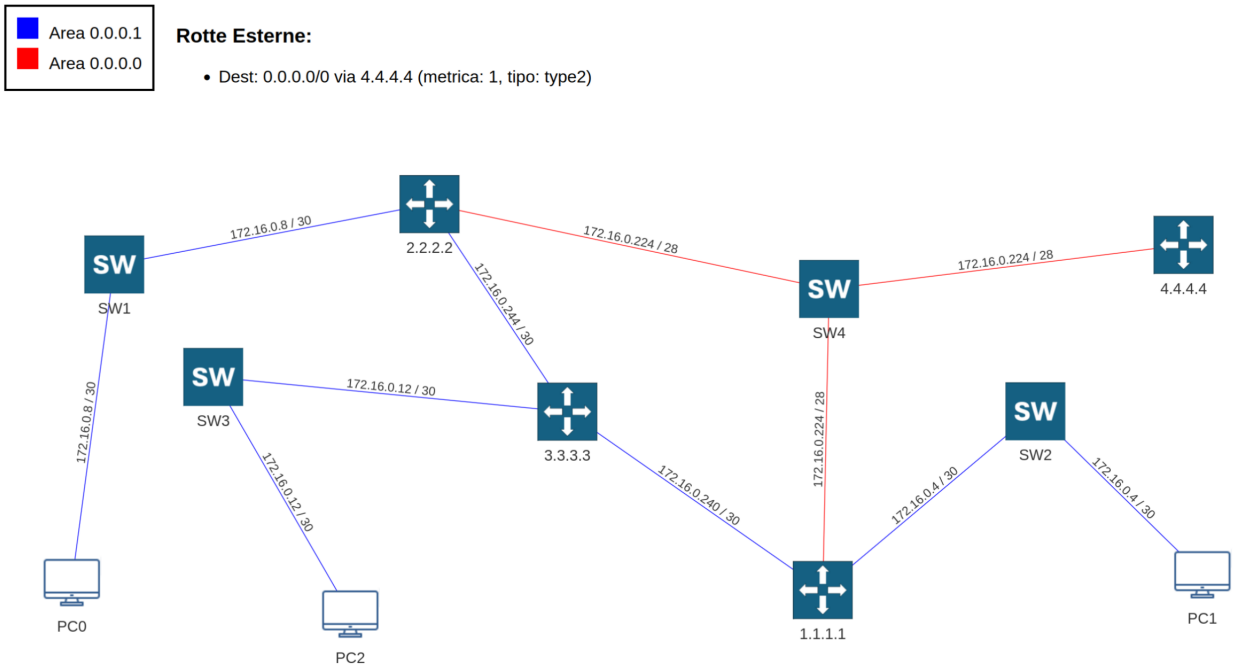

Figure 33: Interfaccia grafica completa, esempio 1

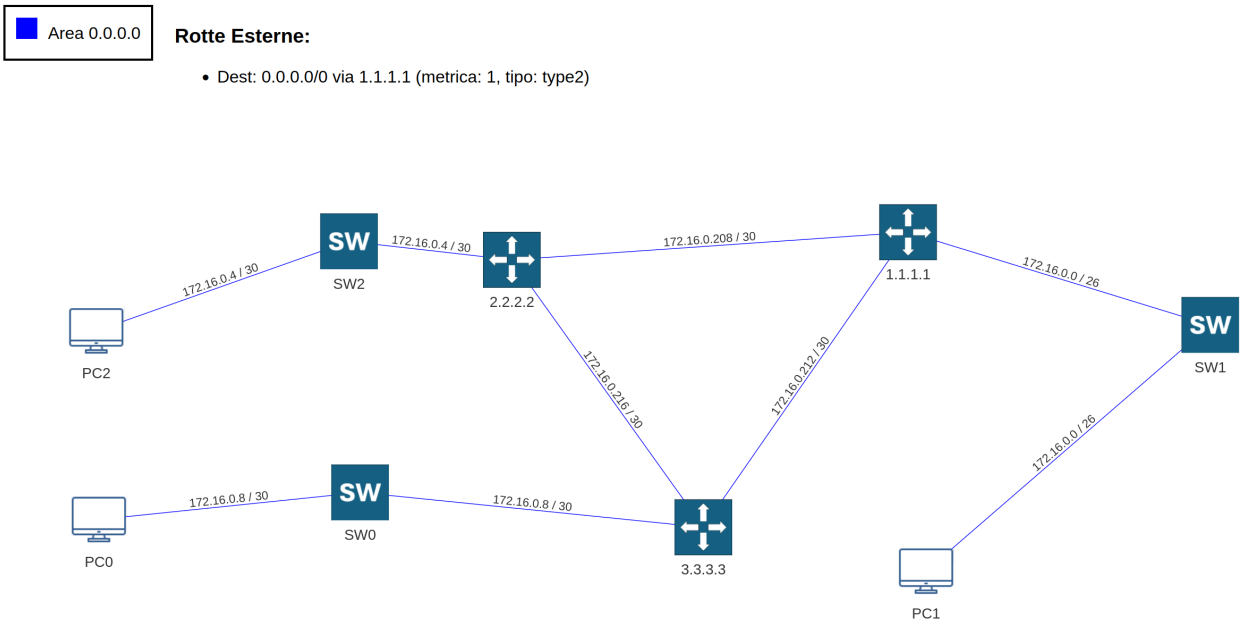

Figure 34: Interfaccia grafica completa, esempio 2

Capitolo 5

Utilizzo del software

5.1 Avvio dell'applicazione

Per avviare l'applicazione da terminale si utilizza il comando:

```
./ospf_mgmt_app.sh ip-interfaccia-target
```

Il parametro ip-interfaccia-target specifica l'indirizzo IP dell'interfaccia del nodo a partire dal quale si è interessati a ricostruire la topologia. Nel file `ospf_mgmt_app.sh`, oltre all'avvio di **main.py**, è presente anche l'attivazione di un **virtual environment**. Questo è richiesto esplicitamente da Flask, ma rappresenta comunque una scelta vantaggiosa, in quanto consente di isolare le dipendenze e gestire le numerose librerie utilizzate in modo maggiormente flessibile.

```
1 ip=$1
2
3 source venv/bin/activate
4
5 python3 src/main.py "$ip"
```

Figure 35: ospf_mgmt_app.sh

5.2 Interfaccia a linea di comando

Durante l'esecuzione `main.py` recupera le informazioni sulla topologia secondo le modalità descritte nel Capitolo 3. Inizialmente, vengono mostrate a terminale le caratteristiche del nodo target (Fig. 36).
Dopo questa fase, il programma entra in attesa di un comando. L'insieme dei comandi selezionabili dall'utente è il seguente:

- `id ospf_id`: visualizza le caratteristiche di un nodo utilizzando l'ID OSPF specificato[11];
- `nodes`: mostra le caratteristiche di tutti i nodi della rete;
- `topology`: fornisce informazioni sull'intera topologia di rete. In particolare, l'output è suddiviso in aree. Per ciascuna area, viene mostrata una lista dei **nodi**, identificati esclusivamente dal loro ID OSPF, seguita dall'insieme dei **link** che appartengono all'area stessa. Successivamente, vengono riportate le **rotte inter-area** e i **percorsi verso gli ASBR**. Infine, una volta descritte tutte le aree, vengono mostrate le **rotte esterne** all'autonomous system;
- `display`: avvia il server Flask all'indirizzo IP `0.0.0.0`, consentendo il caricamento di `index.html`, il template dell'interfaccia grafica descritta nel Capitolo 4;
- `help`: mostra una lista dei comandi disponibili;
- `exit`: termina l'esecuzione del programma.

[11] L'output delle caratteristiche di qualsiasi nodo segue lo stesso formato della Fig. 36.

```
/ospf-mgmt-app # ./ospf_mgmt.sh 172.16.0.5
**** OSPF Management APP ****


Hostname: RA
Router ID: 1.1.1.1
    Interfaces:
        ID: Ethernet1, ip: 172.16.0.225/28, Interface Status:
           connected, Line Protocol Status: up
        ID: Ethernet2, ip: 172.16.0.241/30, Interface Status:
           connected, Line Protocol Status: up
        ID: Ethernet3, ip: 172.16.0.5/30, Interface Status:
           connected, Line Protocol Status: up
        ID: Loopback0, ip: 1.1.1.1/32, Interface Status:
           connected, Line Protocol Status: up
        ID: Management0, ip: 172.20.20.4/24, Interface Status:
           connected, Line Protocol Status: up
    Neighbors:
        Interface: Ethernet1, Router ID: 2.2.2.2, Neighbor ip
           address: 172.16.0.226, Adj-State: full, DR:
           172.16.0.227, BDR: 172.16.0.226
        Interface: Ethernet1, Router ID: 4.4.4.4, Neighbor ip
           address: 172.16.0.227, Adj-State: full, DR:
           172.16.0.227, BDR: 172.16.0.226
        Interface: Ethernet2, Router ID: 3.3.3.3, Neighbor ip
           address: 172.16.0.242, Adj-State: full, DR:
           172.16.0.242, BDR: 172.16.0.241
    Route Table:
        Destination: 0.0.0.0/0, Via: 172.16.0.227, Interface:
           Ethernet1, Protocol: ospfExternalType2
        Destination: 1.1.1.1/32, Via: Directly Connected,
           Interface: Loopback0, Protocol: connected
        Destination: 172.16.0.4/30, Via: Directly Connected,
           Interface: Ethernet3, Protocol: connected
        Destination: 172.16.0.8/30, Via: 172.16.0.242, Interface:
            Ethernet2, Protocol: OSPF
        Destination: 172.16.0.12/30, Via: 172.16.0.242, Interface
           : Ethernet2, Protocol: OSPF

        // Restante tabella di routing omessa per brevita'
```

Figure 36: Caratteristiche nodo target

Capitolo 6

Conclusioni

6.1 Considerazioni finali

L'utilizzo di CONTAINERlab per emulazione di rete, unitamente a Flask per la gestione server side la libreria vis.js per l'implementazione dell'interfaccia grafica web-based e PyEAPI per la gestione delle interazioni con i dispositivi di rete, si sono rilevate scelte vincenti ai fini dello sviluppo dell'applicativo software. CONTAINERlab, in quanto strumento di emulazione di rete basato su container, ha consentito di simulare in modo altamente realistico diverse topologie di rete, senza la necessità di un'infrastruttura fisica complessa. Inoltre, CONTAINERlab e la libreria PyEAPI si sono integrate alla perfezione, permettendo una gestione fluida e automatizzata della topologia di rete emulata. Infine, la decisione di sviluppare un'interfaccia grafica web-based si è rivelata vantaggiosa, grazie alla possibilità nativa di definire eventi che consentono un'interazione semplice e diretta con l'utente su una pagina web, oltre alla grande flessibilità offerta dal framework Flask.

6.2 Possibili miglioramenti futuri

L'applicazione, sebbene sia perfettamente in grado di eseguire le operazioni per cui è stata progettata, risulta comunque abbastanza limitata. Essa, infatti, ricostruisce la topologia esclusivamente nel caso di configurazione OSPFv2: questo rappresenta sia un limite che un punto di partenza per eventuali miglioramenti futuri. In particolare, sarebbe utile estendere il supporto a ulteriori protocolli di rete, sia intra-autonomous-system (come RIP e OSPFv3) che inter-autonomous-system (ad esempio BGP). Inoltre, l'integrazione con sistemi di monitoraggio esterni, come ad esempio Zabbix o Nagios, permetterebbe un controllo ancora più completo della rete.

Capitolo 7

Bibliografia

[1] containerlab. *Documentazione ufficiale di CONTAINERlab.* https://containerlab.dev/

[2] Arista Networks. *Documentazione ufficiale di Arista EOS.* https://www.arista.com/en/firmware/eos

[3] Internet Engineering Task Force (IETF). *OSPFv2 - Open Shortest Path First versione 2.* https://tools.ietf.org/html/rfc2328

[4] Arista Networks. *Documentazione ufficiale della libreria PyEAPI.* https://github.com/arista-netdev/PyEAPI

[5] Pallets Projects. *Documentazione ufficiale del framework Flask.* https://flask.palletsprojects.com/

[6] *Documentazione ufficiale della libreria vis.js.* https://visjs.org/