Routing Path Visualization Documentation for ETH Mini-Internet

1 Overview

This document describes the implementation of a traceroute-based AS path visualization feature for the ETH Mini-Internet platform. The extension allows students to launch traceroutes through the web interface, visualize the resulting AS paths and detect policy violations based on business relationships between ASes.

The implementation simplifies the process of initiating and interpreting traceroute measurements.

2 Topology Data Extraction

2.1 LaTeX-Based Topology Generation

The ETH Mini-Internet platform includes a LaTeX-based topology generator used for official documentation. This LaTeX file was extended with macros to log additional metadata (e.g., node coordinates, AS types, inter-AS links and business relationships) during compilation. The data is written to a structured file `topology.txt`. This file needs to be placed in the source directory of the web server's Docker container.

2.2 JSON Conversion

On web server startup, a Python script parses `topology.txt` and transforms it into a JSON format (`topology.json`) that is compatible with the `vis.js` visualization library. This file contains the node and edge definitions for the AS-level graph.

```
{
        "nodes": [
        {"id": 80, "label": "80", "type": "ixp", "x": 0, "y": 0},
        {"id": 1, "label": "1", "type": "tier1", "x": 85.358, "y": 32.309}
        ],
        "edges": [
        {"from": 1, "to": 80, "type": "peer"}
        ]
}
```

Listing 1: Excerpt from topology.json used by vis.js

This approach ensures visual consistency between documentation and the frontend, and allows the graph to be updated simply by recompiling the LaTeX file and replacing `topology.txt`.

3 Backend Data Derivation

To support traceroute configuration and enrich visualization, a second internal data structure is generated. It combines:

- Public AS link definitions from `aslevel_links_students.txt`
- Router definitions from `l3_routers.txt` and `l3_links.txt`

The script `subnet_config.sh`, which was already part of the platform, is reused to systematically derive the IP addresses of all router interfaces. This data structure is exposed to the frontend via an API and includes:

- Available routers per AS
- Border routers with external interfaces
- Inter-AS link information (including IPs and routers)


```
        "3": {
                "public_links": [
                {
                        "ip": "179.1.3.3",
                        "peer_asn": 1,
                        "peer_ip": "179.1.3.1",
                        "peer_role": "Provider",
                        "peer_router": "ZURI",
                        "role": "Customer",
                        "router": "MUNI",
                        "subnet": "179.1.3.0/24"
                }
                ],
                "routers": {
                        "BASE": {
                                "container": "3_BASErouter",
                                "interfaces": [{"ip": "179.2.3.3", "type": "
                                    external"}],
                                "is_border": true
                        }
                }
        }
```


Listing 2: Excerpt from derived router information base

4 Traceroute Execution Backend

The backend is implemented in Flask and provides endpoints to launch and retrieve traceroute jobs asynchronously. Traceroutes are executed using the Docker CLI inside the corresponding router container.

Workflow

1. User selects origin router and target IP in frontend
2. POST request to `/launch-traceroute`
3. Origin container is verified using a predefined whitelist
4. Target IP address is validated to ensure it is a syntactically correct IP
5. Backend starts traceroute thread, and returns `job_id`
6. Frontend polls `/get-traceroute-result?job_id=...`

Traceroutes are executed using a Docker command that launches the traceroute inside the selected router container. The command used is shown in Listing 3.

```
docker exec 3_ZURIrouter traceroute -w 1 -q 1 -m 10 3.0.5.2
```

Listing 3: Example traceroute execution

This command is configured for speed. It uses only a single probe per hop (`-q 1`) and waits just one second per probe (`-w 1`). The maximum number of hops is limited to 10 (`-m 10`) to avoid unnecessary delays.

The `max hops` value of 10 is a default parameter and should be adapted to the size of the configured topology to ensure full reachability in cases where paths span more than 10 hops.

Results are parsed using the `jc` library, which converts the traceroute output into structured JSON format and are stored temporarily in memory for retrieval. A background cleanup function periodically removes entries that have exceeded a configured expiration time (defaulting to 600 seconds).

5 Policy Violation Detection

The frontend includes logic to detect violations of the valley-free routing model. This is based on:

- AS path returned by the traceroute
- Business relationships from the backend (`/routers` API)

Violations are shown in red in the graph and with a warning in the traceroute panel.

```
Policy Violation Detected:  Invalid OVER → UP transition at AS14 → AS12
```

This logic is implemented entirely in JavaScript.

6 Repository

All implementation code is available at:
`https://github.com/FuhrmannMartin/mini_internet_project`