Softwarized and Virtualized Mobile Networks

On-demand SDN slicing

Project Description

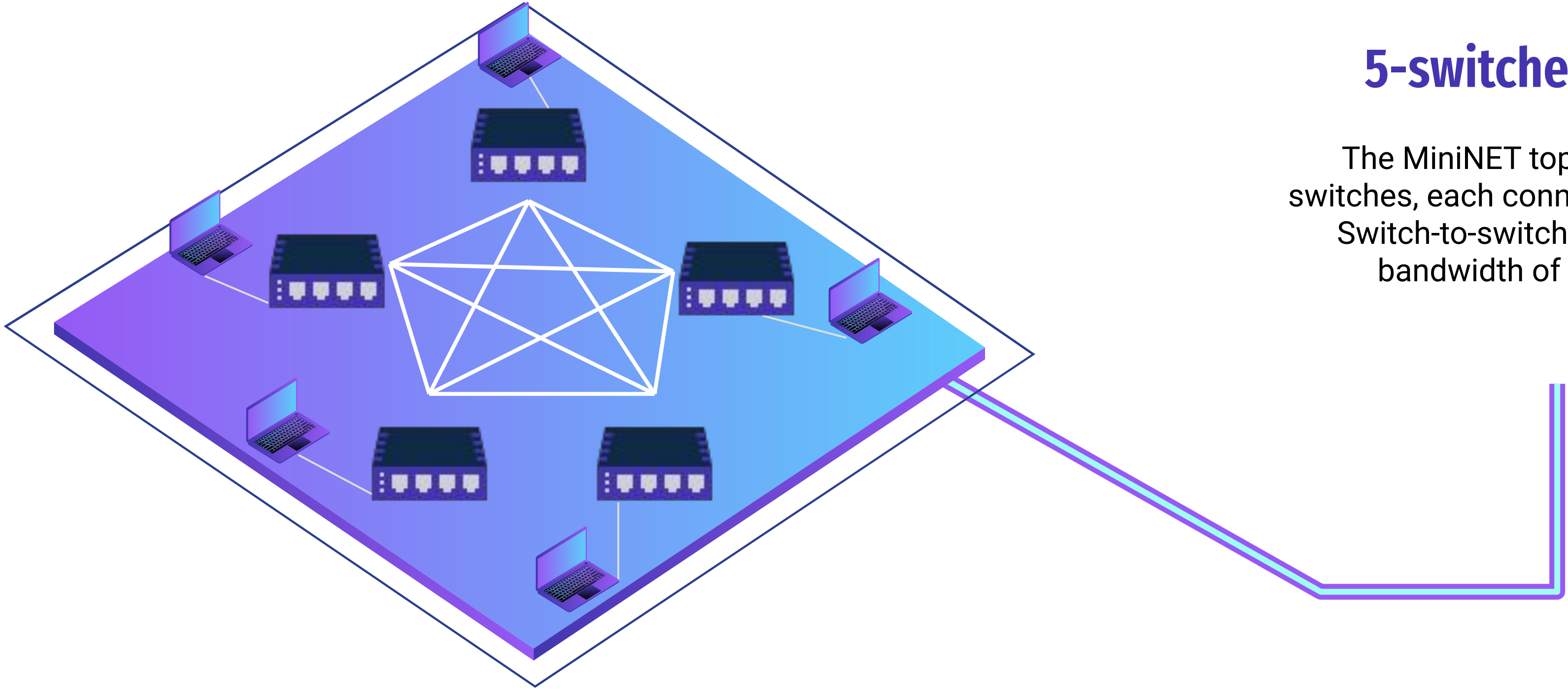
Approach

Implement in ComNetsEmu
an application enabling
on-demand handling of
network slicing

Slicing

Allow to identify flows,
topology and QoS for each
slice

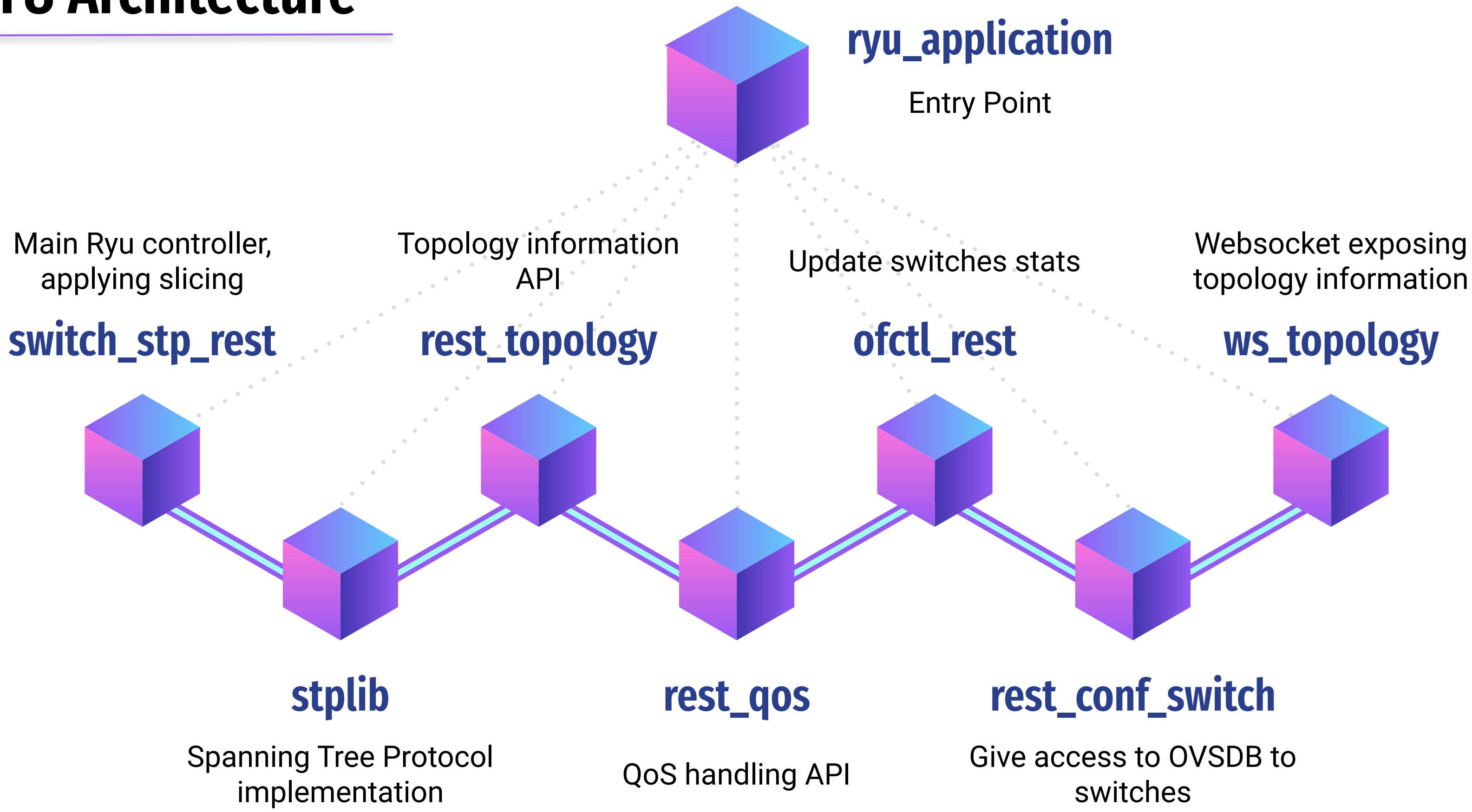
MiniNET starting topology



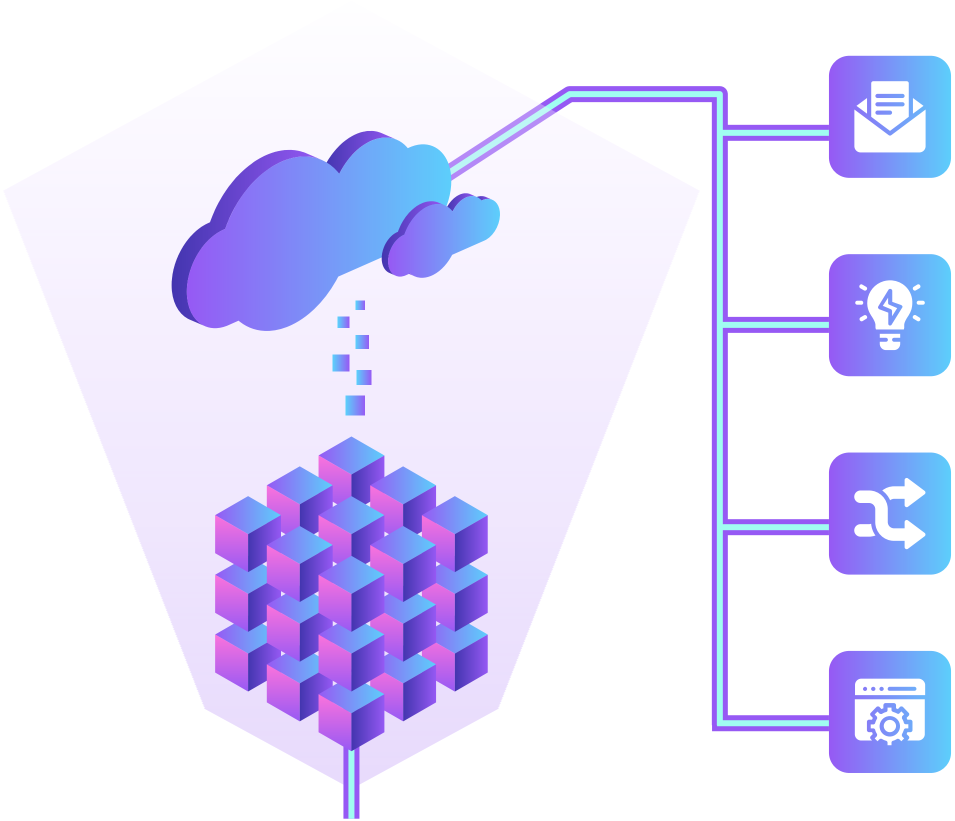
5-switches Mesh

The MiniNET topology has 5 switches, each connected to 1 host. Switch-to-switch links have a bandwidth of 10 Mbps.

RYU Architecture



switch_stp_rest



@packet-in

Handle packets by applying the requested slice template

Spanning Tree Protocol

Apply the STP protocol to handle loops within the architecture

Custom Events

Send specially-crafted events when applying architectural changes

Rest API

Expose an API providing information and methods to handle slices

switch_stp_rest - STP

- Prevent bridge loops by building a loop-free logical topology
- The result is a spanning tree
- IEEE 802.1D

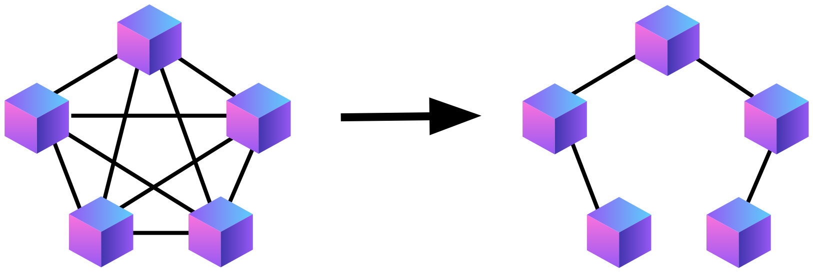


Diagram 1 - Example of a STP result

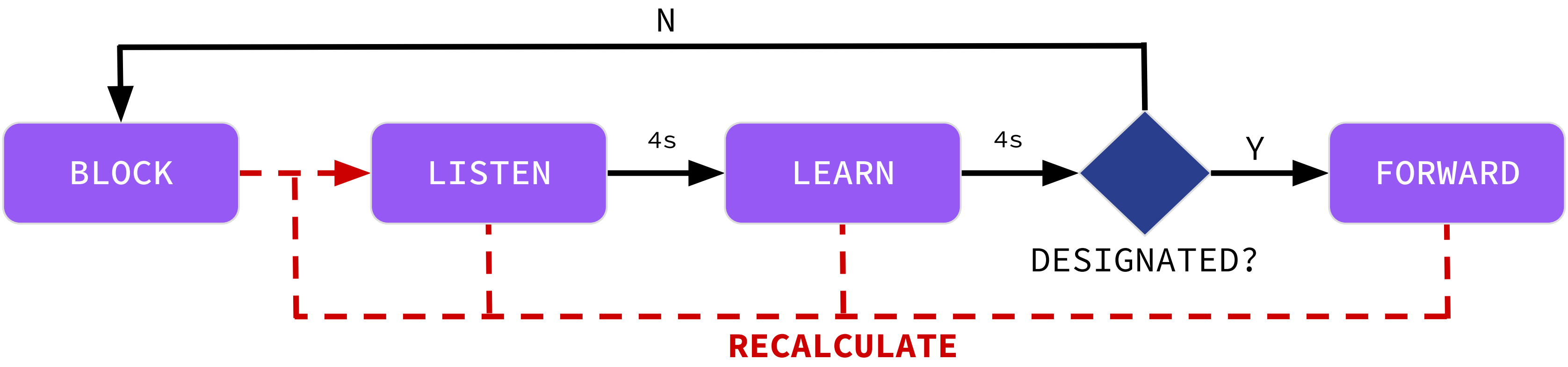
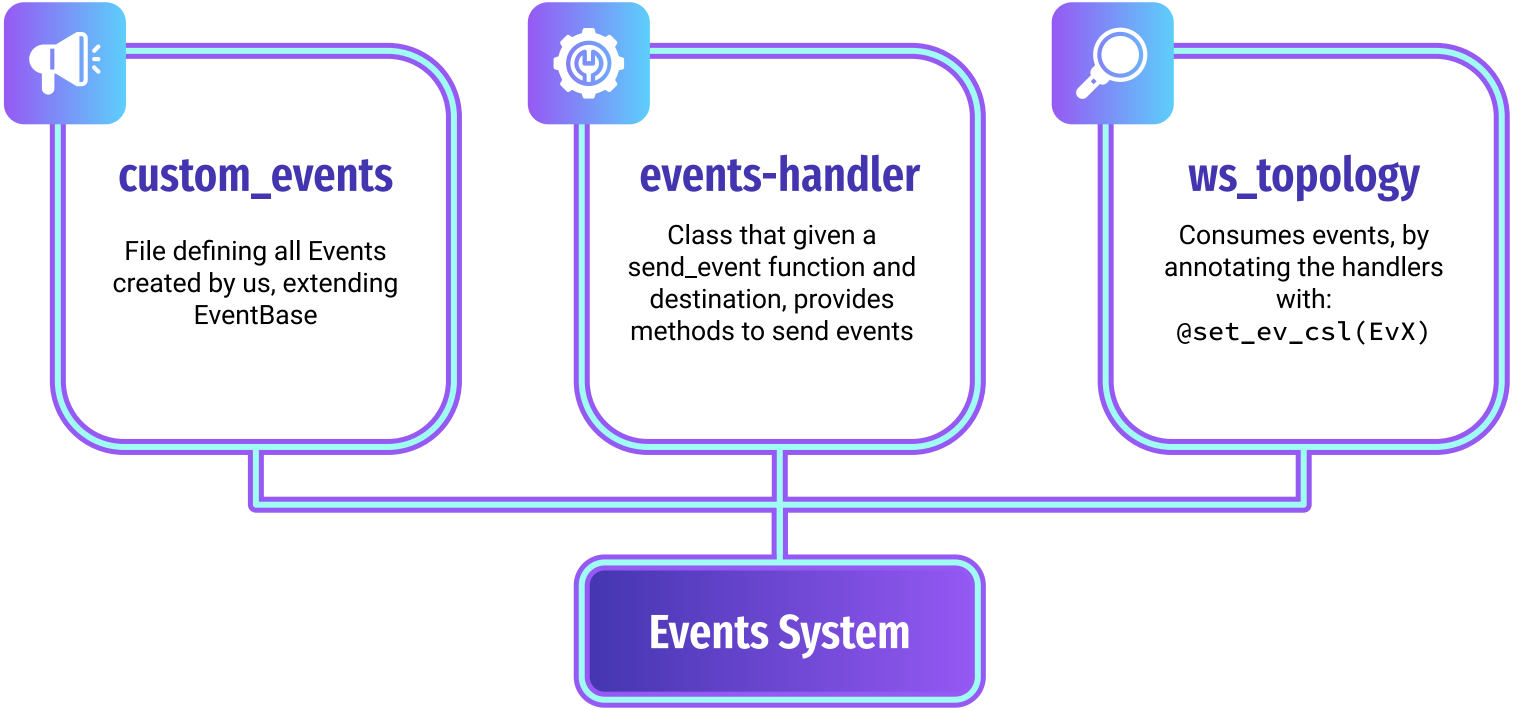


Diagram 2 - Port State Machine of the STP

switch_stp_rest - Events

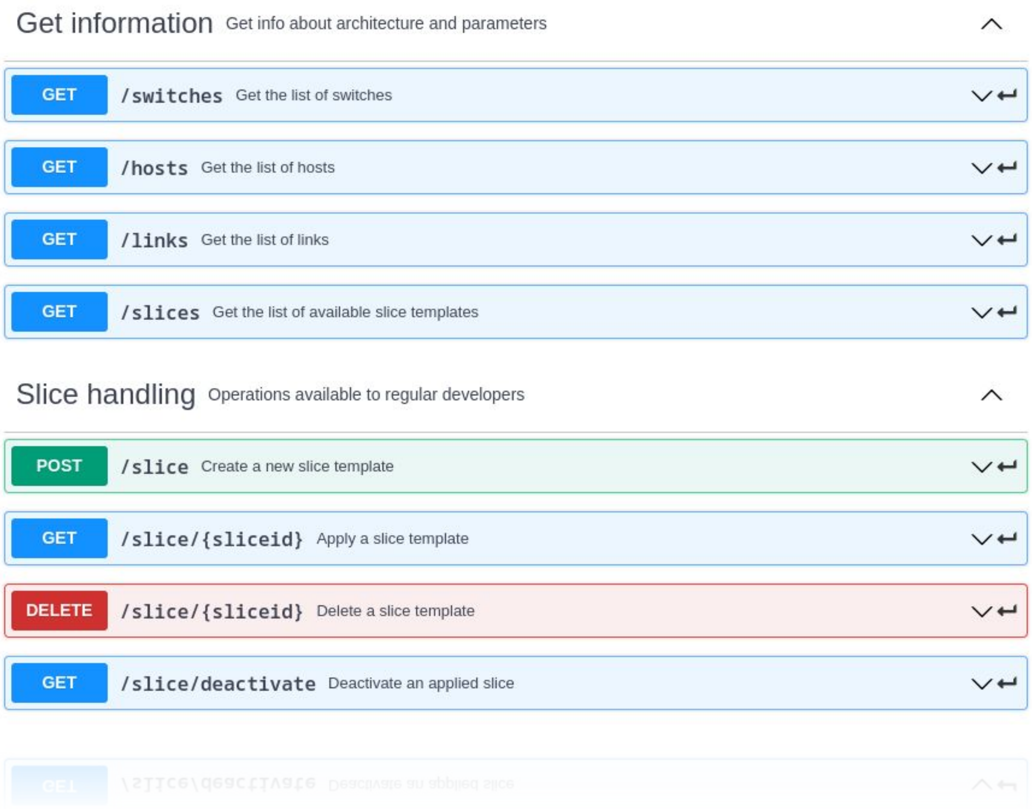


switch_stp_rest - Rest API

- OpenAPI 3.0.0 specification
- Documentation available at:
  - /docs/index.html
  - [SwaggerHUB]()

The YAML file containing the source is available under:

- `resources/docs.yaml`



switch_stp_rest - Slices Handling

01

Template Definition

Each slice is represented by two templates:

- Topology slicing template
- QoS template

```
{
  "1": {"5": [1,3], "1": [5], "3": [5]},
  "2": {"1": [5], "5": [1]},
  "3": {},
  "4": {"1": [5], "5": [1]},
  "5": {}
}
```

Snippet 1 - Example of a topology slicing template

```
[{
  "switch_id": 1,
  "port_name": "s1-eth5"
  "match": [{
    "nw_dst": 10.0.0.2,
    "nw_src": 10.0.0.1
  }, …],
  "queues": [{
    "queue": "0",
    "max_rate": "500000"
  }, …],
}, …]
```

Snippet 2 - Example of a QoS template

switch_stp_rest - Slices Handling

```
for switch in self.get_switches():
  switch_id = dpid_lib.str_to_dpid(switch["dpid"])
  bridge = self.stp.bridge_list[switch_id]
  for port in switch["ports"]:
    port_id = self.str_to_port_no(port["port_no"]])
    port = self.dpset.get_port(switch_id, port_id)
    if **port not in slice template**:
      bridge.link_down(port)
    else:
      bridge.link_up(port)
```

Snippet 3 - Example of a topology slicing template

02

Topology Slicing activation

After the template activation:

- all links not identified by the template are disabled by using the STP APIs
- the STP recalculation process is then initiated

switch_stp_rest - Slices Handling

03

QoS settings application

When a slice is applied, the corresponding QoS configuration is set up:

- give access to OVS_DB to the switch
- define the virtual queue
- add the rule

```
requests.put(
    'http://localhost:8080/v1.0/conf/switches/{switch_id}/ovsdb_addr',
    data='"tcp:127.0.0.1:6632"'
)

requests.post('http://localhost:8080/qos/queue/{switch_id}', json.dumps({
    "port_name": qos_configuration["port_name"],
    "type": "linux-htb",
    "max_rate": "10000000000",
    "queues": [{"max_rate": qos_configuration['max_rate']}]
}))

requests.post('http://localhost:8080/qos/rules/{switch_id}', json.dumps({
    "match": {
        "nw_dst": qos_configuration["nw_dst"],
        "nw_src": qos_configuration["nw_src"],
    },
    "actions": {
        "queue": qos_configuration["queue"]
    }
}))
```

Snippet 4 - QoS queue and rule creation

switch_stp_rest - Slices Handling

```
if str(in_port) in self.slice_to_port[str(dpid)]:
  # learn a mac address to avoid FLOOD next time.
  self.mac_to_port[dpid][src] = in_port
  if dst in self.mac_to_port[dpid] and **dst port in slice**:
    out_port = [self.mac_to_port[dpid][dst]]
  else:
    out_port = self.slice_to_port[str(dpid)][str(in_port)]
  actions = [parser.OFPActionOutput(int(out)) for out in out_port]
else:
  # Can't communicate due to slice restrictions
  ...
```

Snippet 5 - Code portion applying the slice

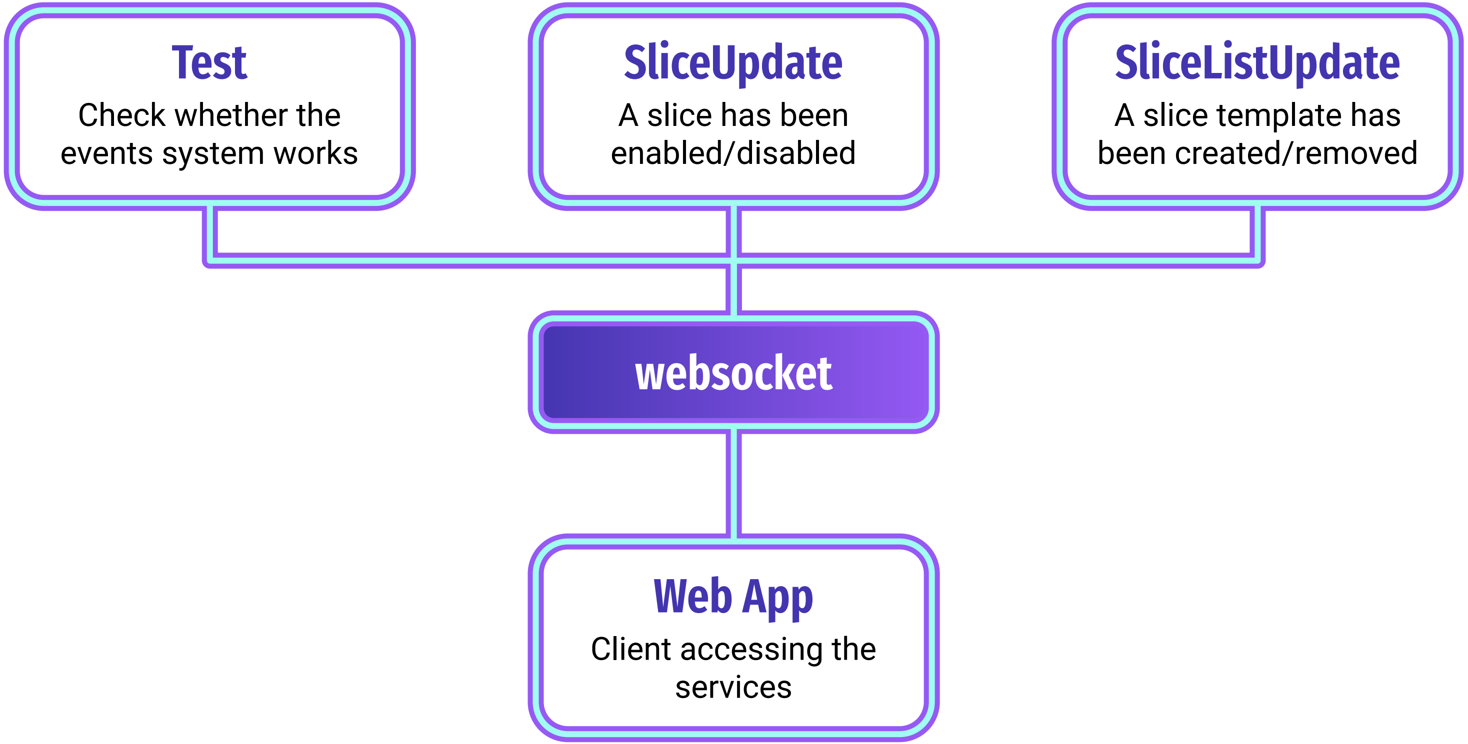
04

Packet in Handler

After the slice activation, the packet in handler:

- checks if the slice allows the communication
- sends the message to the destination or performs flood, if the mac is not known

Frontend communication



Overall View

Frontend

- Listens to events
- Uses the Rest API to manage the network

Backend

- Processes requests
- Emits events
- Sends events to WS

Overall View

Thanks to this architecture, multiple clients can concurrently access the web application: updates will automatically be pushed to anyone subscribing to the websocket

Frontend implementation

The frontend code mainly relies on:

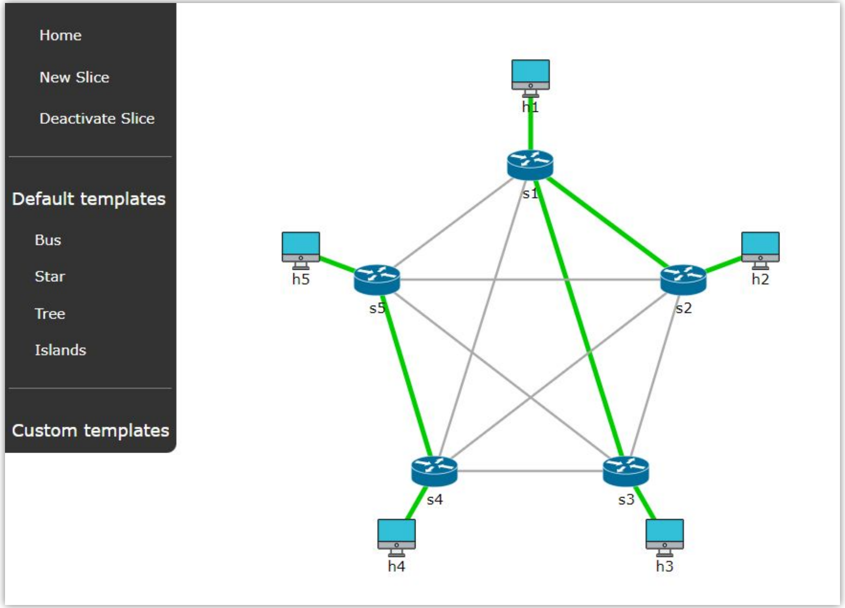
- **D3.js** - JS library for producing dynamic and interactive data visualizations
- **WebSocket** - protocol that enables full-duplex communication channels over a TCP connection

```
const ws = new WebSocket(`ws://${location.host}/v1.0/topology/ws`);
ws.onmessage = (e) => {
  const data = e.data;
  // Handle event...
}
```

Snippet 6 - Code portion creating the WebSocket

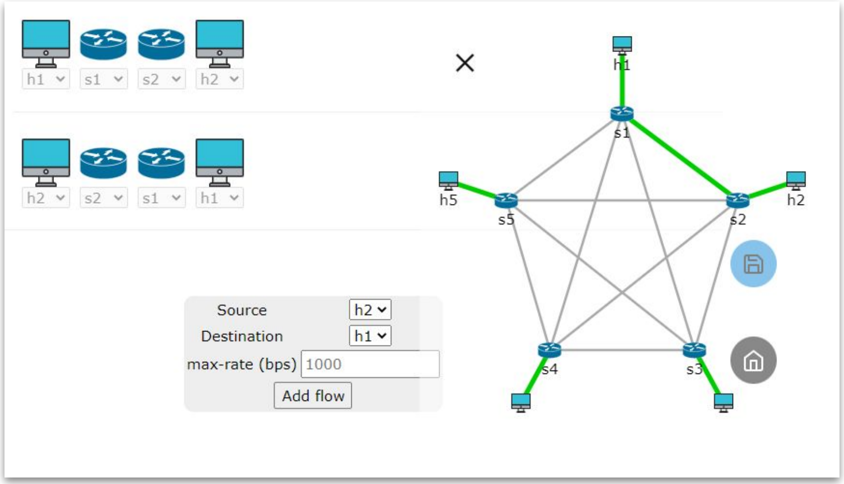
Frontend implementation - Homepage

1. View the network topology
2. Apply predefined slices
3. Apply custom slices
4. Deactivate the current slice



Frontend implementation - Custom slices

1. Create and delete new slices
2. Add QoS configurations
3. Preview the created slice
4. Name and save the configuration



Thank You!

- END OF PRESENTATION -