SDN Slice Setup Optimization

Networking Mod. 2

OpenFlow

Project Goals

Mininet

- **Emulate** a SDN using Mininet and Ryu Controller
- **Deploy** containerized services
- QoS **monitor and enhancement**
- QoS **degradation reaction**

Architecture

Service Layer (Docker)

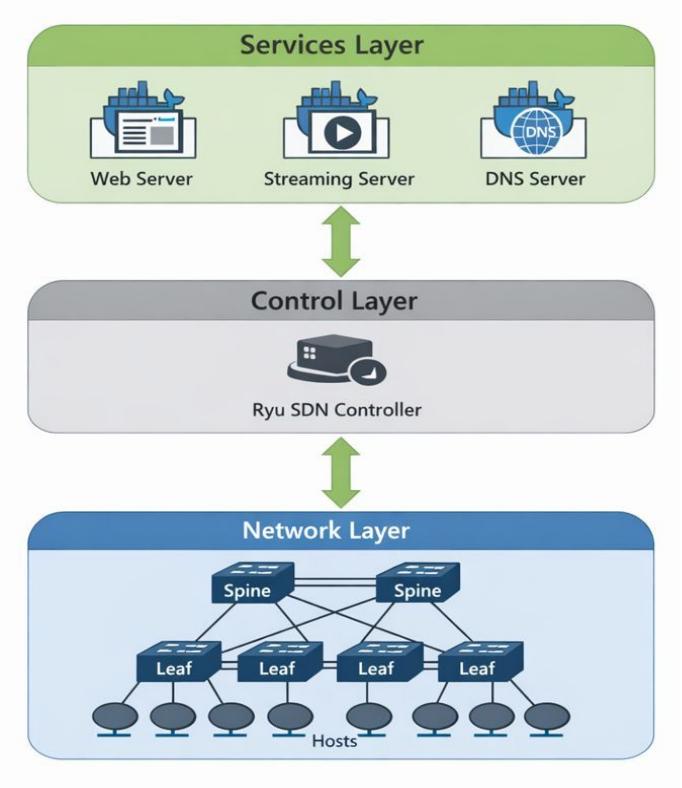
- Web service (custom Nginx)
- Streaming service (custom Nginx)
- DNS service (Technitium)

Control Layer (Ryu Controller)

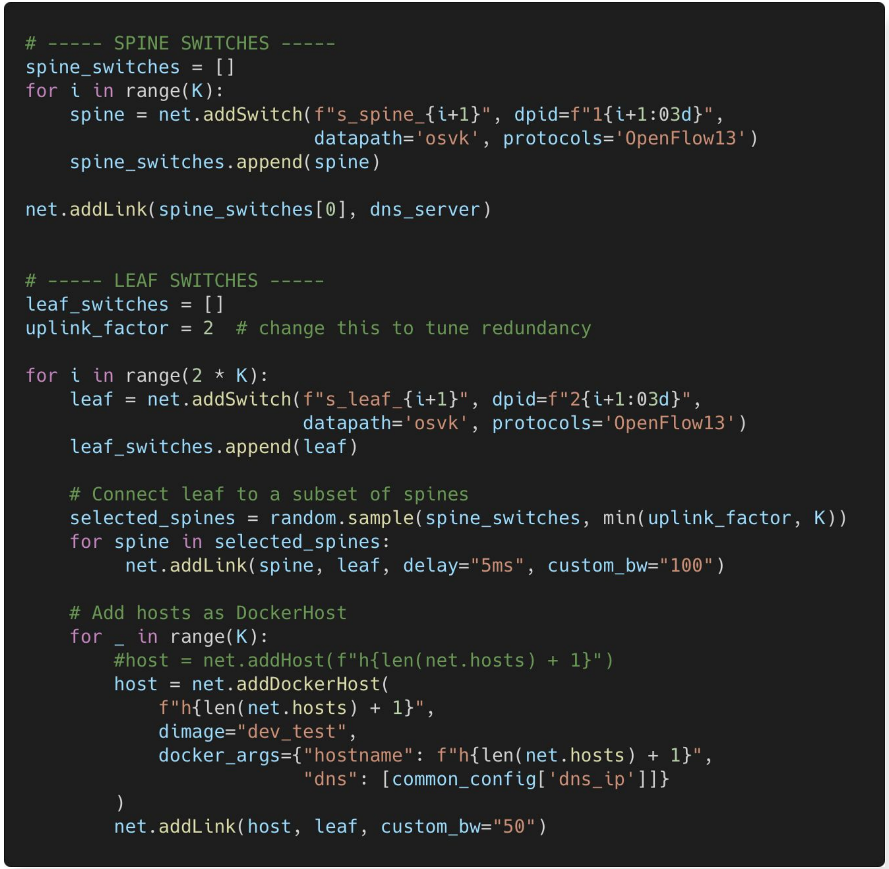
- SDN traffic management
- Service monitoring
- Service migration decision logic

Network Layer (Mininet)

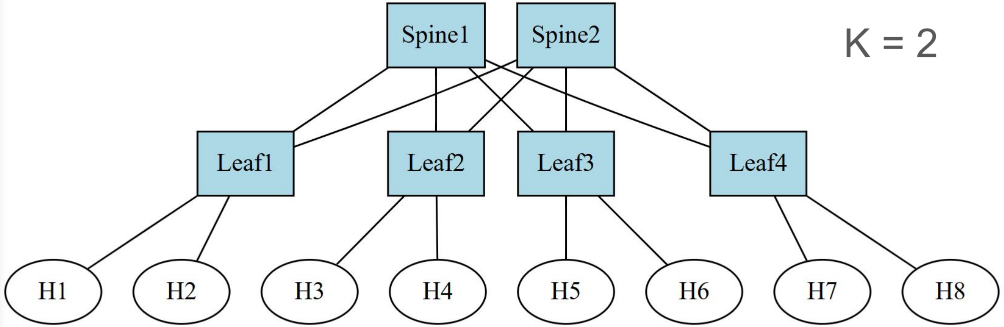
- Spine-Leaf topology
- Emulation of clients and servers



Spine-Leaf Topology



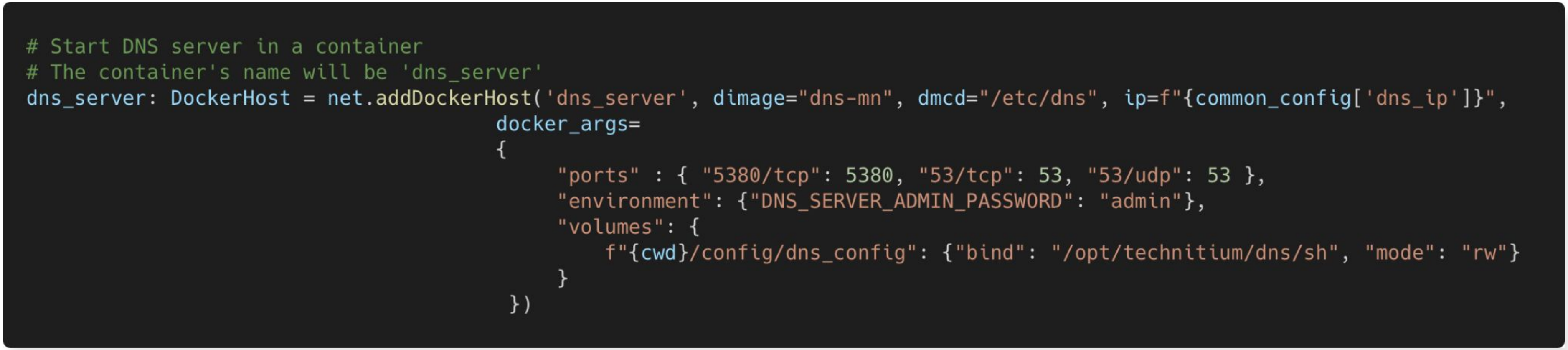
```
# ----- SPINE SWITCHES -----
spine_switches = []
for i in range(K):
    spine = net.addSwitch(f"s_spine_{i+1}", dpid=f"1{i+1:03d}",
                          datapath='osvk', protocols='OpenFlow13')
    spine_switches.append(spine)

net.addLink(spine_switches[0], dns_server)


# ----- LEAF SWITCHES -----
leaf_switches = []
uplink_factor = 2  # change this to tune redundancy

for i in range(2 * K):
    leaf = net.addSwitch(f"s_leaf_{i+1}", dpid=f"2{i+1:03d}",
                         datapath='osvk', protocols='OpenFlow13')
    leaf_switches.append(leaf)

    # Connect leaf to a subset of spines
    selected_spines = random.sample(spine_switches, min(uplink_factor, K))
    for spine in selected_spines:
         net.addLink(spine, leaf, delay="5ms", custom_bw="100")

    # Add hosts as DockerHost
    for _ in range(K):
        #host = net.addHost(f"h{len(net.hosts) + 1}")
        host = net.addDockerHost(
            f"h{len(net.hosts) + 1}",
            dimage="dev_test",
            docker_args={"hostname": f"h{len(net.hosts) + 1}",
                         "dns": [common_config['dns_ip']]}
        )
        net.addLink(host, leaf, custom_bw="50")
```



K = 2

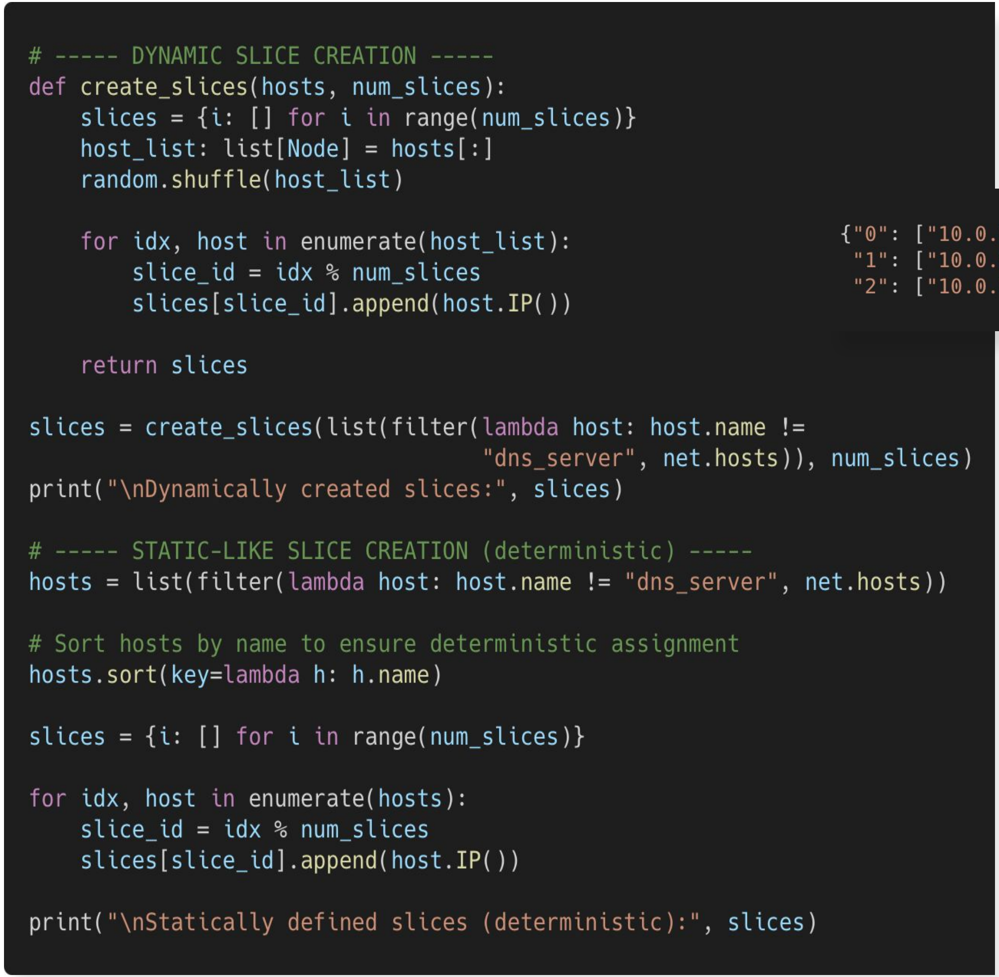
DNS

- Provides a **stable service identifier**
- Service migration is handled by **updating DNS records only**
- Enables **transparent and seamless service continuity**

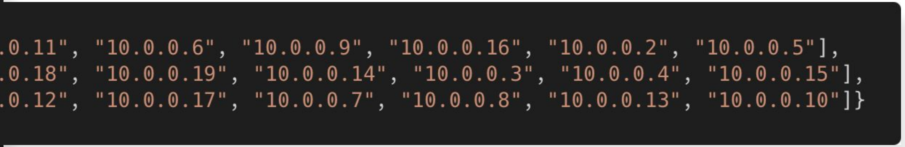


```python
# Start DNS server in a container
# The container's name will be 'dns_server'
dns_server: DockerHost = net.addDockerHost('dns_server', dimage="dns-mn", dmcd="/etc/dns", ip=f"{common_config['dns_ip']}",
                                       docker_args=
                                       {
                                            "ports" : { "5380/tcp": 5380, "53/tcp": 53, "53/udp": 53 },
                                            "environment": {"DNS_SERVER_ADMIN_PASSWORD": "admin"},
                                            "volumes": {
                                                f"{cwd}/config/dns_config": {"bind": "/opt/technitium/dns/sh", "mode": "rw"}
                                            }
                                        })
```

Slice Creation



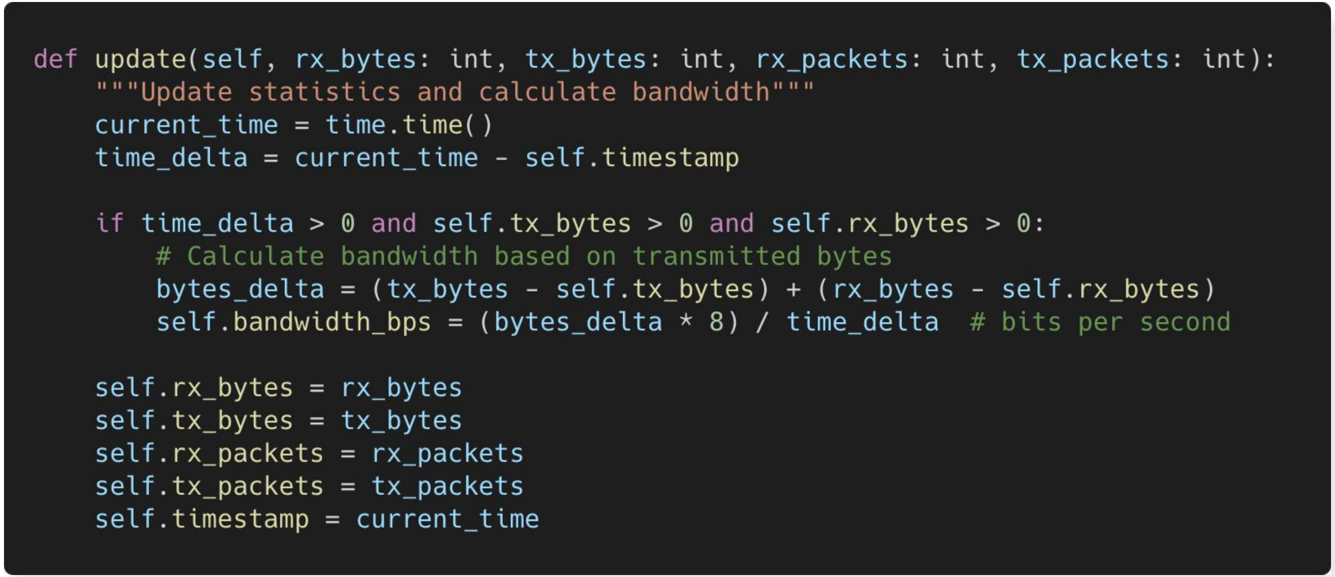
```
# ----- DYNAMIC SLICE CREATION -----
def create_slices(hosts, num_slices):
    slices = {i: [] for i in range(num_slices)}
    host_list: list[Node] = hosts[:]
    random.shuffle(host_list)

    for idx, host in enumerate(host_list):
        slice_id = idx % num_slices
        slices[slice_id].append(host.IP())

    return slices

slices = create_slices(list(filter(lambda host: host.name !=
                                    "dns_server", net.hosts)), num_slices)
print("\nDynamically created slices:", slices)

# ----- STATIC-LIKE SLICE CREATION (deterministic) -----
hosts = list(filter(lambda host: host.name != "dns_server", net.hosts))

# Sort hosts by name to ensure deterministic assignment
hosts.sort(key=lambda h: h.name)

slices = {i: [] for i in range(num_slices)}

for idx, host in enumerate(hosts):
    slice_id = idx % num_slices
    slices[slice_id].append(host.IP())

print("\nStatically defined slices (deterministic):", slices)
```



```
{"0": ["10.0.0.11", "10.0.0.6", "10.0.0.9", "10.0.0.16", "10.0.0.2", "10.0.0.5"],
 "1": ["10.0.0.18", "10.0.0.19", "10.0.0.14", "10.0.0.3", "10.0.0.4", "10.0.0.15"],
 "2": ["10.0.0.12", "10.0.0.17", "10.0.0.7", "10.0.0.8", "10.0.0.13", "10.0.0.10"]}
```

- Partition hosts into **logical slices** to isolate services and traffic
- Number of slices configurable via num_slices
- Each slice contains a list of host IPs
- Slice information is saved locally (slices.json)
- Slices are sent to the controller via REST API (/api/v0/slice)

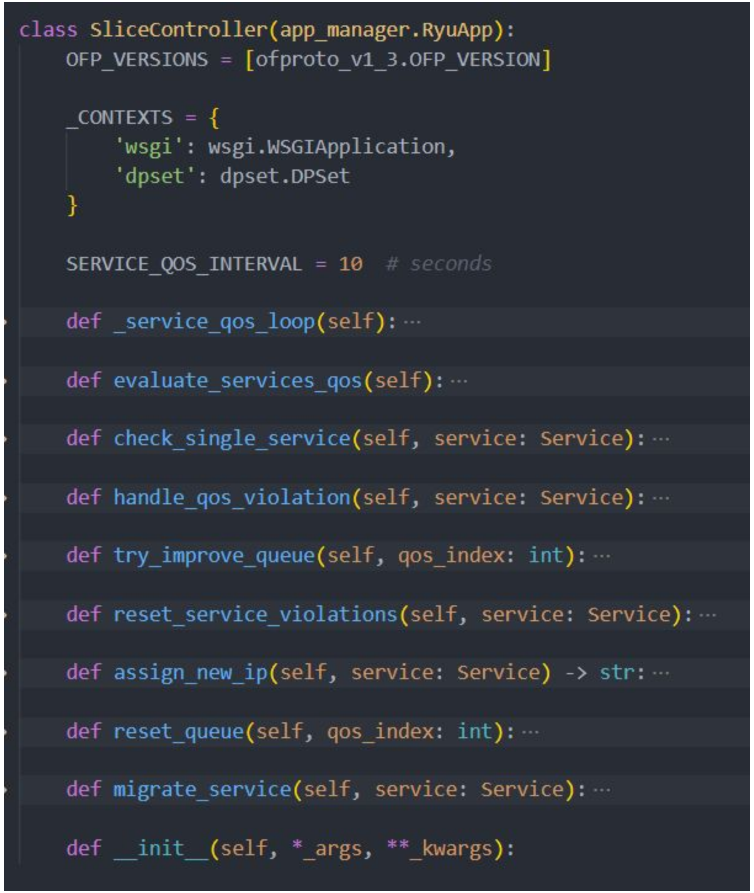
Link Statistics Monitoring



```python
def update(self, rx_bytes: int, tx_bytes: int, rx_packets: int, tx_packets: int):
    """Update statistics and calculate bandwidth"""
    current_time = time.time()
    time_delta = current_time - self.timestamp

    if time_delta > 0 and self.tx_bytes > 0 and self.rx_bytes > 0:
        # Calculate bandwidth based on transmitted bytes
        bytes_delta = (tx_bytes - self.tx_bytes) + (rx_bytes - self.rx_bytes)
        self.bandwidth_bps = (bytes_delta * 8) / time_delta  # bits per second

    self.rx_bytes = rx_bytes
    self.tx_bytes = tx_bytes
    self.rx_packets = rx_packets
    self.tx_packets = tx_packets
    self.timestamp = current_time
```

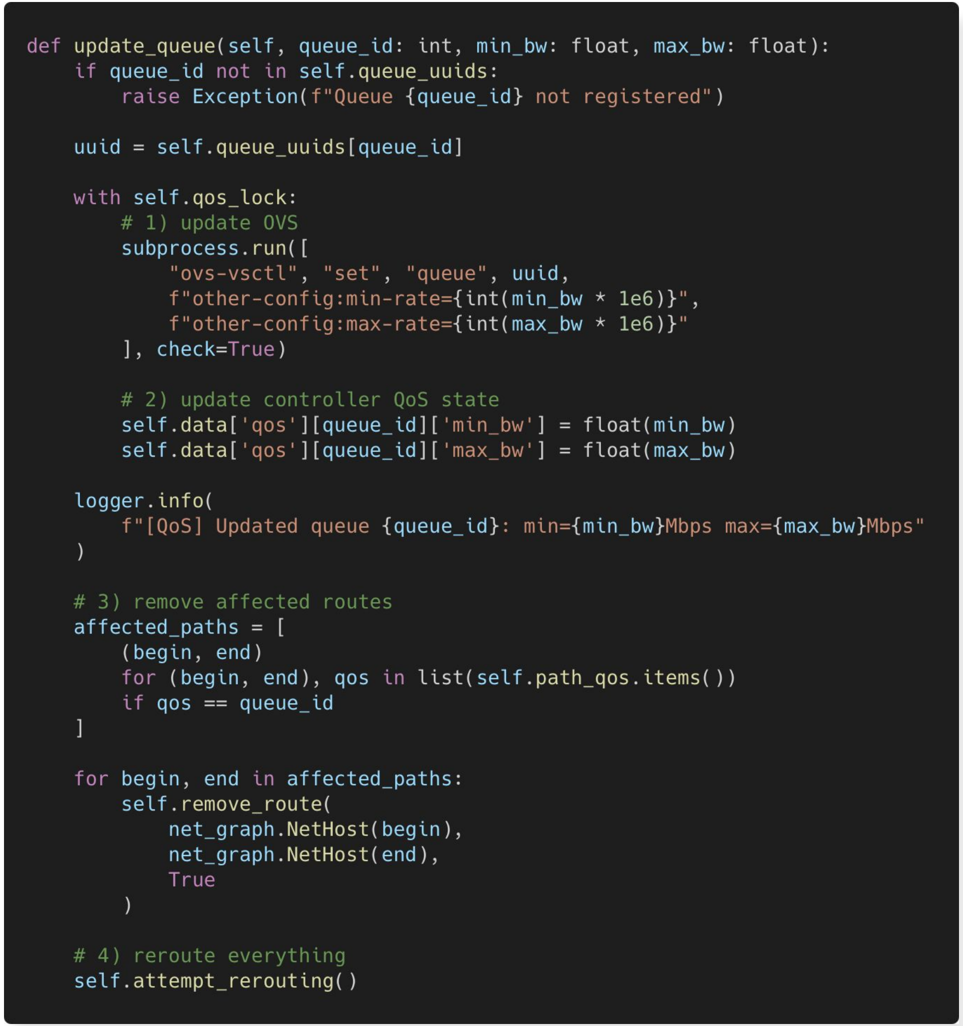
- **Tracks per-link metrics:** RX/TX bytes for each port of the switches
- **Periodic polling** via a dedicated monitoring thread
- **Handles OpenFlow port stats replies** from switches
- **Calculates bandwidth** from byte deltas over time

Controller Logic



```python
class SliceController(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]

    _CONTEXTS = {
        'wsgi': wsgi.WSGIApplication,
        'dpset': dpset.DPSet
    }

    SERVICE_QOS_INTERVAL = 10  # seconds

    def _service_qos_loop(self): ···

    def evaluate_services_qos(self): ···

    def check_single_service(self, service: Service): ···

    def handle_qos_violation(self, service: Service): ···

    def try_improve_queue(self, qos_index: int): ···

    def reset_service_violations(self, service: Service): ···

    def assign_new_ip(self, service: Service) -> str: ···

    def reset_queue(self, qos_index: int): ···

    def migrate_service(self, service: Service): ···

    def __init__(self, *_args, **_kwargs):
```

- The controller **enforces** network slicing to isolate traffic, QoS, and services.
- **Routing decisions** are QoS-aware and dynamically recomputed based on network state.
- Service degradation triggers **automatic QoS tuning or service migration.**

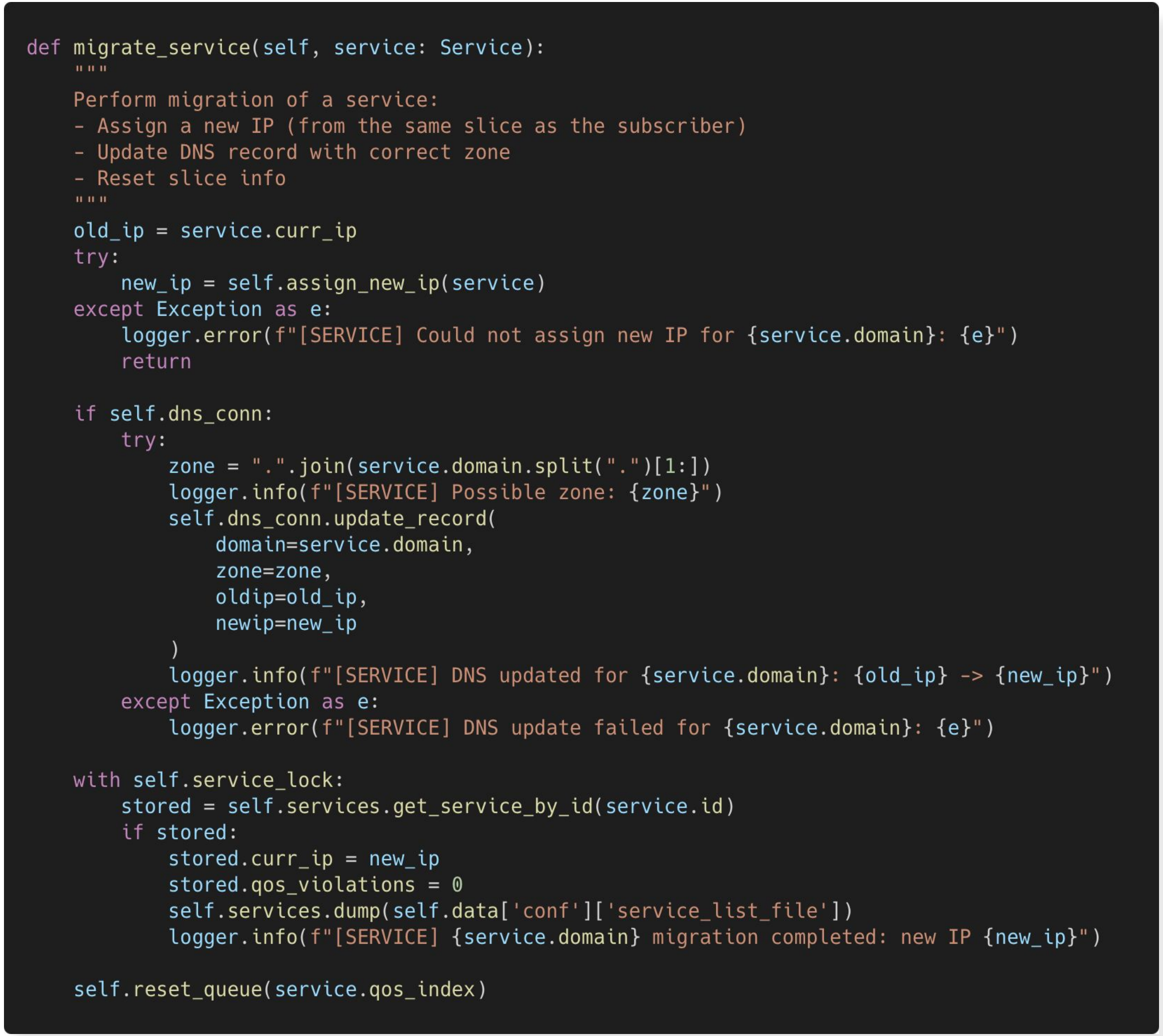


```python
def update_queue(self, queue_id: int, min_bw: float, max_bw: float):
    if queue_id not in self.queue_uuids:
        raise Exception(f"Queue {queue_id} not registered")

    uuid = self.queue_uuids[queue_id]

    with self.qos_lock:
        # 1) update OVS
        subprocess.run([
            "ovs-vsctl", "set", "queue", uuid,
            f"other-config:min-rate={int(min_bw * 1e6)}",
            f"other-config:max-rate={int(max_bw * 1e6)}"
        ], check=True)

        # 2) update controller QoS state
        self.data['qos'][queue_id]['min_bw'] = float(min_bw)
        self.data['qos'][queue_id]['max_bw'] = float(max_bw)

    logger.info(
        f"[QoS] Updated queue {queue_id}: min={min_bw}Mbps max={max_bw}Mbps"
    )

    # 3) remove affected routes
    affected_paths = [
        (begin, end)
        for (begin, end), qos in list(self.path_qos.items())
        if qos == queue_id
    ]

    for begin, end in affected_paths:
        self.remove_route(
            net_graph.NetHost(begin),
            net_graph.NetHost(end),
            True
        )

    # 4) reroute everything
    self.attempt_rerouting()
```

QoS Optimization: Queue Improve

When the **QoS is degraded**, the controller attempts to:

- **Update OVS** (Open vSwitch)
- **Update controller** QoS state
- Remove affected rows and attempt to **reroute**



```python
def migrate_service(self, service: Service):
    """
    Perform migration of a service:
    - Assign a new IP (from the same slice as the subscriber)
    - Update DNS record with correct zone
    - Reset slice info
    """
    old_ip = service.curr_ip
    try:
        new_ip = self.assign_new_ip(service)
    except Exception as e:
        logger.error(f"[SERVICE] Could not assign new IP for {service.domain}: {e}")
        return

    if self.dns_conn:
        try:
            zone = ".".join(service.domain.split(".")[1:])
            logger.info(f"[SERVICE] Possible zone: {zone}")
            self.dns_conn.update_record(
                domain=service.domain,
                zone=zone,
                oldip=old_ip,
                newip=new_ip
            )
            logger.info(f"[SERVICE] DNS updated for {service.domain}: {old_ip} -> {new_ip}")
        except Exception as e:
            logger.error(f"[SERVICE] DNS update failed for {service.domain}: {e}")

    with self.service_lock:
        stored = self.services.get_service_by_id(service.id)
        if stored:
            stored.curr_ip = new_ip
            stored.qos_violations = 0
            self.services.dump(self.data['conf']['service_list_file'])
            logger.info(f"[SERVICE] {service.domain} migration completed: new IP {new_ip}")

    self.reset_queue(service.qos_index)
```

QoS Optimization: Service Migration

In case the QoS hasn’t been **enhanced enough**, the controller triggers the migration of the service:

- **Assign** a new IP from the same slice as the subscriber
- **Update** DNS record with the correct zone
- **Reset** slice info

NB: the migration is performed only on the browsing service.