SDN-BASED QOS TRAFFIC CLASSIFICATION & LOAD BALANCING FOR IOT GATEWAY

MINOR PROJECT REPORT

SUBMITTED BY

JOSHUA FELIX (SCT22EC074)

MUNEERA S (SCT22EC088)

SIDDHARTH S KRISHNAN (SCT22EC110)

VIBHU V (SCT22EC123)

to

the APJ Abdul Kalam Technological University in partial fulfillment of requirements for the award of Degree of Bachelor of Technology in Electronics and Communication Engineering with minor in Computer Science and Engineering (Networking)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
SREE CHITRA THIRUNAL COLLEGE OF ENGINEERING,

THIRUVANANTHAPURAM - 695018

OCTOBER 2025

DEPT. OF COMPUTER SCIENCE AND ENGINEERING
SREE CHITRA THIRUNAL COLLEGE OF ENGINEERING TRIVANDRUM
2025 - 26

CERTIFICATE

This is to certify that the report Entitled **SDN-Based QoS Traffic Classification & Load Balancing for IOT Gateway** submitted by **Joshua Felix** (SCT22EC074), **Muneera S** (SCT22EC088), **Siddharth S Krishnan** (SCT22EC110), **Vibhu V** (SCT22EC123) to the APJ Abdul Kalam Technological University in partial fulfillment of the B.Tech Degree in Electronics and Communication Engineering with Minor in Computer Science and Engineering (Networking) the minor project work carried out under our guidance and supervision. This report in any form has not been submitted to any other University or Institute for any purpose.

Smt Sreepriya L
(Project Guide)
Assistant Professor
Dept.of CSE
SCTCE
Trivandrum

Smt Merrin J
(Project Coordinator)
Assistant Professor
Dept.of CSE
SCTCE
Trivandrum

Dr. Soniya B
Professor and Head
Dept.of CSE
SCTCE
Trivandrum

DECLARATION

We hereby declare that the minor project report **SDN-Based QoS Traffic Classification & Load Balancing for IOT Gateway**, submitted in partial fulfillment of the requirements for the award of the degree of Bachelor of Technology of the APJ Abdul Kalam Technological University, Kerala, is a bonafide work done by us under the supervision of Smt. Merrin J.

This submission represents our ideas in our own words and where ideas or words of others have been included, we have adequately and accurately cited and referenced the original sources.

We also declare that we have adhered to the ethics of academic honesty and integrity and have not misrepresented or fabricated any data, idea, fact, or source in our submission. We understand that any violation of the above will be a cause for disciplinary action by the institute and/or the University and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been obtained. This report has not previously formed the basis for the award of any degree, diploma, or similar title of any other University.

Trivandrum

23-10-2025

Joshua Felix (SCT22EC074)

Muneera S (SCT22EC088)

Siddharth S Krishnan (SCT22EC110)

Vibhu V (SCT22EC123)

ACKNOWLEDGEMENT

We take this opportunity to express our deepest sense of gratitude and sincere thanks to everyone who helped us to complete this work successfully. We express our sincere thanks to Dr. Soniya B, Head of Department, Computer Science and Engineering, Sree Chitra Thirunal College of Engineering for providing us with all the necessary facilities and support.

We would like to express our sincere gratitude to the Smt Sreepriya L, Assistant Professor, department of Computer Science and Engineering, Sree Chitra Thirunal College of Engineering Trivandrum for the support and co-operation.

We would like to place on record our sincere gratitude to our Minor Project coordinator, Smt Merrin , Assistant Professor, Computer Science and Engineering, Sree Chitra Thirunal College of Engineering for the guidance and mentorship throughout this work.

Finally we thank our family, and friends who contributed to the successful fulfillment of this project work.

Joshua Felix (SCT22EC074)

Muneera S (SCT22EC088)

Siddharth S Krishnan (SCT22EC110)

Vibhu V (SCT22EC123)

ABSTRACT

This project presents a Software Defined Networking (SDN) based Internet of Things (IoT) gateway system designed to enhance network traffic classification, load balancing, and security in IoT environments. By leveraging SDN principles and OpenFlow-enabled switches, the gateway dynamically manages and prioritizes IoT device traffic, ensuring Quality of Service (QoS) and responsiveness. The architecture includes rule-based traffic classification modules and QoS-aware forwarding, integrated with real-time network performance monitoring. The system was implemented using Mininet for network emulation, Ryu as the SDN controller framework, and tested with real traffic patterns to demonstrate improved flow management and network efficiency in IoT scenarios. This approach contributes towards scalable, programmable, and secure IoT networking infrastructures, addressing the challenges of heterogeneous device communication and dynamic QoS requirements effectively.

Contents

List of Figures

Chapter 1
Introduction

1.1 Background

The Internet of Things (IoT) connects myriad devices: sensors, actuators, embedded systems and consumer gadgets into networks that generate a continuous stream of data. Devices and applications differ widely in their QoS needs: emergency sensor events require minimal latency; telemetry uploads can tolerate delays. Traditional distributed networking struggles to meet these mixed demands at scale because configuration is static and policy enforcement is fragmented.

Software-Defined Networking (SDN) separates the control plane from the data plane, enabling centralized, programmatic control of forwarding devices. A logically centralized controller has a global view of the network and can install fine-grained flow rules via a southbound protocol such as OpenFlow. This programmability enables dynamic traffic engineering, network-wide QoS policy enforcement, and automation: all important for modern IoT gateways that must prioritize critical traffic while handling massive device scale.

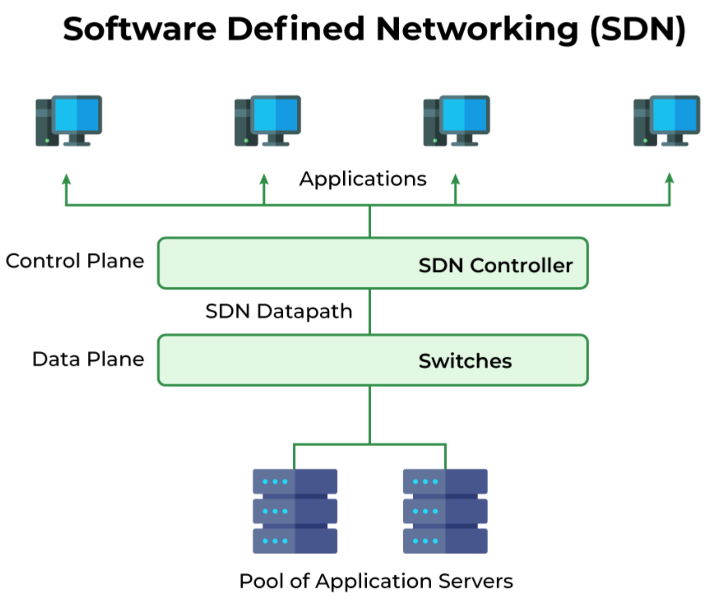

Figure 1.1: General Software-Defined Networking (SDN) architecture

1.2 Motivation

A central controller in SDN brings flexibility but also a single point of intensive computation. As the number of switches and flow requests rises, a controller's CPU and I/O can become a bottleneck, causing increased flow-setup latency and degraded QoS for critical flows. Multi-controller deployments mitigate scale limits but introduce the problem of uneven load distribution: some controllers become overloaded while others remain underutilized. The challenge is to manage controller assignment of switches dynamically such that QoS and throughput are preserved while avoiding controller saturation.

1.3 Goals

The goals of this project are:

1. Design a practical SDN architecture for IoT gateways that performs real-time QoS-aware traffic classification.
2. Implement a Decision Controller that monitors controller load and executes adaptive, minimum-cost switch migration using a dynamic thresholding mechanism.
3. Evaluate the system in emulation (Mininet/Ryu) and demonstrate improvements in average throughput and latency compared to static-threshold baselines.

1.4 Contributions

This project contributes:

- A dual-controller design separating responsibilities: Work Controller and Decision Controller (global load balancing and threshold control).
- An implementation blueprint using Ryu, Open vSwitch and Mininet, with instrumentation for Packet-In counting and lightweight telemetry.
- An adaptive threshold algorithm inspired by CLBDT (dynamic threshold regulation), ensuring stable migration decisions and preventing oscillatory behavior.
- A Streamlit-based dashboard.

Chapter 2
Literature Review

2.1 SDN for IoT and QoS

SDN has been used widely to introduce dynamic QoS control in IoT and wireless deployments. Works such as Sounni et al. (2021) demonstrate how SDN can centralize AP association decisions in Wi-Fi networks to balance load across access points, improving per-AP throughput and reducing jitter. QoS-aware path selection and queueing strategies (e.g., multipath routing and bandwidth reservations) have been shown to lower latency for delay-sensitive flows.

2.2 Controller Load Balancing

Large SDN deployments prefer multiple controllers to meet scale and reliability requirements. However, uncoordinated partitioning can produce imbalanced controller loads. Fixed-threshold migration schemes were among first attempts to address this, but they suffer when traffic fluctuates rapidly. More recent approaches, notably CLBDT (Controller Load Balancing based on Dynamic Threshold, Li et al., 2023), use dynamic threshold regulation to adapt migration aggressiveness based on control-plane load statistics. CLBDT includes rules for threshold reduction (when network is underutilized) and threshold increase (when migration would overload the target controller), and selects minimal sets of switches whose migration reduces the source controller load below the threshold — reducing migration overhead.

2.3 IoT-specific Considerations

IoT traffic is heterogeneous: short periodic telemetry, sporadic bursts, firmware updates, and event-driven alerts coexist. Approaches that integrate traffic classification at the controller (using port/application signatures such as MQTT, CoAP, HTTP) allow early prioritization. Combining classifier outputs with controller load metrics provides a more informed basis for migration decisions: for instance, switches carrying a high proportion of high-priority flows may be avoided for migration unless necessary, or migration can be scheduled to minimize disruption to real-time flows.

2.4 Summary of Gaps

From the literature:

- QoS classification and controller balancing are often addressed independently.
- Fixed thresholds are brittle; dynamic thresholds improve stability but require careful tuning.
- Practical implementations must minimize migration cost (number of switches moved, duplicated flow entries) to avoid harming QoS during reconfiguration.

This project integrates these lessons into a single, modular framework targeted at IoT gateways.

Chapter 3
Problem Statement and Objectives

3.1 Problem Statement

IoT gateways must handle diverse flows while preserving low latency and high reliability. Without adaptive control, some SDN controllers get overloaded by Packet-In bursts from traffic-heavy switches. Overload increases flow installation latency and may starve critical flows, degrading QoS. The central problem is designing a load balancing mechanism that:

- Reacts to controller overloads promptly,
- Minimizes migration cost and QoS disruption,
- Adapts thresholds and policies in changing traffic conditions.

3.2 Objectives

1. Implement rule-based traffic classification to prioritize IoT flows at the Work Controller.
2. Build a Decision Controller that continuously collects per-switch and per-controller telemetry and computes normalized loads.
3. Implement an adaptive threshold system that decides when and which switches to migrate.
4. Validate improvements in throughput and latency in Mininet emulation with Ryu controllers and Open vSwitch.

Chapter 4
System Design

4.1 Architecture

The architecture uses a logically centralized control plane split into two cooperating controllers:

- Work Controller: installs flow rules, performs packet inspection for classification, assigns queues (high/medium/low), and reports local per-switch telemetry (Packet-In counts, queue occupancy).
- Decision Controller: aggregates telemetry from multiple Work Controllers, computes normalized controller loads, runs the dynamic threshold logic, selects migration candidates, and coordinates migration steps.

Both controllers communicate over a secure control-channel (e.g., REST API or message bus) for coordination. The data plane comprises Open vSwitch instances controlled via OpenFlow 1.3.

4.1.1 Block Diagram

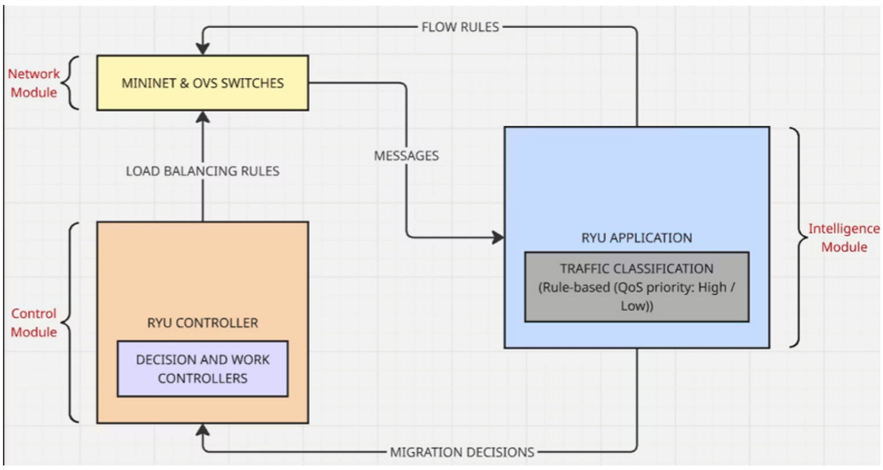

Figure 4.1: Block diagram

4.1.2 Network Diagram

The emulated network mimics an IoT gateway scenario: several hosts (sensors), edge switches, and wired/wireless APs represented by Open vSwitch instances. Each switch maintains per-queue configuration to support prioritized forwarding.

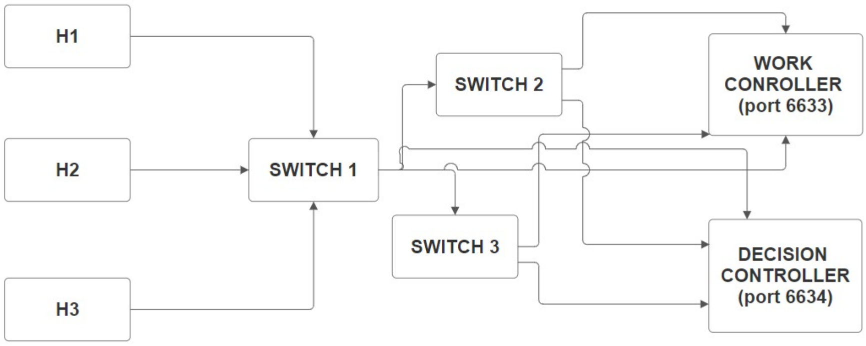

Figure 4.2: Mininet topology

4.2 Modules

4.2.1 Work Controller

Responsibilities:

- Packet-In handling: examine packet headers when a packet misses a flow entry.
- Traffic classification: assign flow to a QoS class using protocol/port rules (examples: MQTT (1883) and CoAP (5683) as high priority; HTTP telemetry as medium; OTA updates as low).
- Flow installation: install prioritized flow entries and configure switch queues accordingly.
- Local telemetry: measure packet-in arrival rates per switch (λ_s), queue occupancy, and send periodic reports to Decision Controller.

4.2.2 Decision Controller

Responsibilities:

- Global monitoring: collect controller load metrics and per-switch report summaries.
- Load computation: compute controller load using total Packet-In counts per switch, $L_{ctrl} = \sum_{s \in S_{ctrl}} \lambda_s$, where λ_s is the measured Packet-In rate from switch s. No normalization factor (C_{max}) is applied.
- Threshold management: adapt the threshold θ based on network balance indicators.
- Migration decision: select a minimal set of switches whose migration reduces the overloaded controller's load below θ and ensures target controller load remains below θ after migration.
- Orchestrate migration: coordinate with the Work Controller to perform safe handover and flow-table updates.

4.3 Load Migration Workflow

The migration process proceeds as follows:

1. Decision Controller identifies an overloaded controller ($L_{src} > \theta$).
2. Candidate selection: find switches s with load l_s such that moving them reduces L_{src} below θ and keeps $L_{tgt} + l_s \leq \theta$ for the chosen target controller.
3. If no candidate set exists, invoke threshold increase routine (to avoid endless failed migrations).
4. Coordinate migration: instruct Work Controller to push overlapping flow entries to both controllers temporarily and then switch mastership (OpenFlow role change or reassign via controller mapping).
5. Cleanup: remove duplicate entries once the target confirms stable operation.

This workflow minimizes packet loss by briefly allowing both controllers to manage flows during handover and by choosing migration candidates that reduce the number of switches moved. In practice, migration was handled through controlled reassignment of switch–controller mappings to minimize disruption.

Chapter 5
Methodology

5.1 Telemetry and Load Measurement

Accurate telemetry drives the Decision Controller. Key metrics collected at the Work Controller include:

- Packet-In arrival rate per switch (λ_s).
- Flow-table size and miss rates.
- CPU utilization and response latency of controller (heartbeat/RPC round-trip).
- Per-queue occupancy and packet drop rates.

These are summarized and sent every monitoring interval (configurable, e.g., 1–5 seconds) to the Decision Controller.

5.2 Controller Load Calculation

Each Work Controller periodically measures the number of Packet-In messages received from every switch it manages. This rate (denoted as λ_s) represents the load contributed by switch s. The Decision Controller then aggregates these values per controller to determine the total controller load:

$$L_{ctrl} = \sum_{s \in S_{ctrl}} \lambda_s$$

where λ_s is the measured Packet-In rate or message count from switch s. No normalization or capacity factor (C_{max}) is used in this implementation. The raw Packet-In counts are compared directly between controllers to identify load imbalances.

5.3 Threshold-based Load Detection

The Decision Controller implements a CLBDT-inspired adaptive threshold mechanism that adjusts based on observed load variations. In this simplified heuristic form, the controller directly compares total Packet-In counts between controllers to detect overload.

Each controller's load is compared against a fixed dynamic threshold value (expressed as an integer or relative percentage). If a controller's load exceeds the threshold while another controller is lightly loaded, migration is triggered.

Unlike the original CLBDT model, this implementation does not compute ratios such as γ or perform continuous threshold updates. Thresholds are manually adjusted after observation or testing to maintain system stability under different traffic levels.

5.4 Switch Migration Logic

When an overload condition is detected, the Decision Controller identifies the switches managed by the overloaded controller. It then performs migration sequentially — one switch at a time — by reassigning them to the lighter controller until the load difference drops below the threshold.

The reassignment is executed using direct controller update commands:

```
ovs-vsctl set-controller sX tcp:<target-IP>:<port>
```

or through OpenFlow role reconfiguration. No optimization or batch selection is applied; migration decisions are sequential and condition-based.

5.5 Traffic Classification Rules

Traffic classification is rule-based and executed on Packet-In:

- High priority: MQTT (1883), CoAP (5683), VoIP RTP, emergency alerts. Assigned to low-latency queue.
- Medium priority: HTTP telemetry, periodic sensor sync. Assigned to best-effort queue with moderate weight.

- Low priority: Bulk transfers, OTA updates, background synchronization. Assigned to low-priority/background queue.

The Work Controller may optionally use simple DPI signatures or IP-subnet rules for better accuracy.

5.6 Migration Safety and QoS Preservation

To keep QoS during migration:

- Brief overlap: flow entries are temporarily installed by both controllers for a short consistent period.
- Priority-preserving moves: avoid migrating switches that currently carry a high proportion of high-priority flows unless unavoidable.
- Graceful rollback: if migration fails (increased packet loss or flow errors), the Decision Controller can revert ownership. In practice, migration was handled through controlled reassignment of switch–controller mappings to minimize disruption. No dual-controller overlap was implemented; instead, the target controller directly assumed ownership once reassignment was completed.

Chapter 6
Implementation and Results

6.1 Technology Stack and Environment

- Ubuntu 20.04 (host OS)
- Mininet (or Mininet-WiFi for wireless-like scenarios) to emulate network topologies
- Ryu SDN Controller (two instances: Work Controller on port 6633 and Decision Controller on port 6634)
- Open vSwitch (OvS) as programmable switches
- Python 3.9 and virtual environment for controller applications
- iperf3 and ping for throughput/latency measurements
- Streamlit for a real-time dashboard showing per-controller load, migration logs, and per-AP throughput

6.1.1 Implementation Details

Work Controller implementation:

- Packet-In handler inspects Ethernet/IP/TCP/UDP headers and applies classification rules.
- Flow installation sets match fields and assigns priority (e.g., using OpenFlow meter/queue configuration).
- Periodic telemetry thread aggregates per-switch Packet-In counts and queue stats, sends compact JSON to Decision Controller.

Decision Controller implementation:

- REST endpoint or message bus consumes telemetry.

- Load calculator computes controller loads from received Packet-In counts and runs threshold logic periodically.
- Migration module issues reassignment commands to update the switch–controller mapping (reassigning mastership or modifying controller IP lists used by OvS).
- Safety module ensures overlapping flow entries and orchestrates cleanup.

6.2 Topology used for Experiments

A typical testbed used:

- 3 Open vSwitch switches (s1, s2, s3)
- 3 hosts (h1, h2, h3) simulating various IoT devices and traffic patterns
- Two controller instances: Work Controller, Decision Controller (monitor and sometimes primary for s3 after migration)

6.3 Results

Experiments compare two configurations:

1. Static Threshold: baseline with a fixed controller load threshold.
2. Dynamic (Proposed): CLBDT-inspired dynamic threshold adjustment + minimal-cost migration.

Metric	Static Threshold	Dynamic (Proposed)
Average Throughput (per AP)	20 Mbps	35 Mbps
Average Latency	20 ms	6 ms

Table 6.1: Performance comparison (static vs dynamic threshold).

6.3.1 Observations

- Throughput improvements stem from fewer Packet-In backlogs and faster flow setup for high-priority flows.

- Latency reductions occur because the Decision Controller migrates hot switches to less loaded controllers before severe degradation.

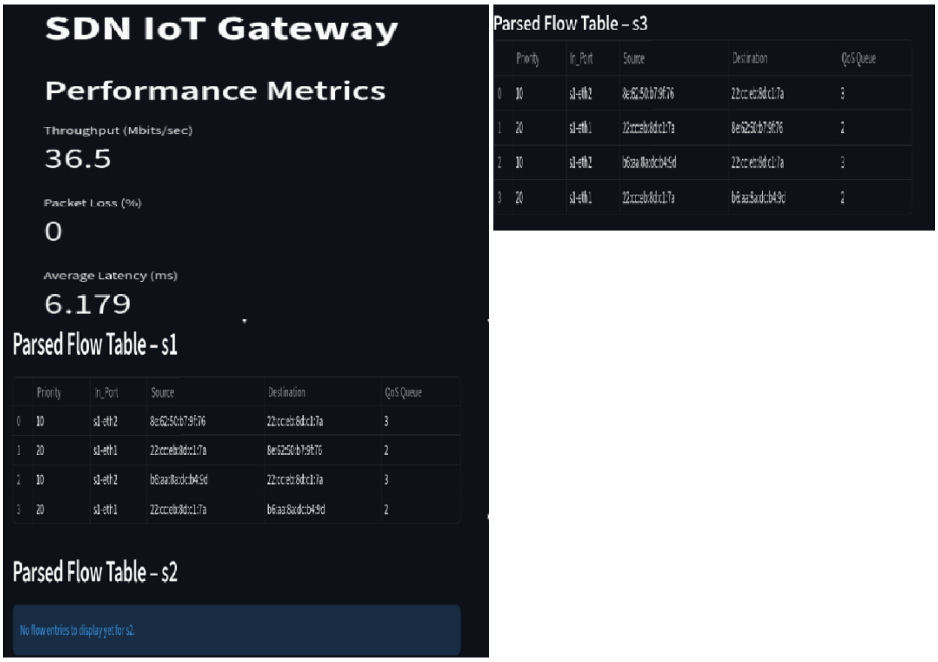

Figure 6.1: Streamlit UI

Chapter 7
Conclusion and Future Work

7.1 Conclusion

This project demonstrates a practical approach for combining QoS-aware traffic classification with adaptive controller load balancing in SDN-based IoT gateways. The dual-controller design: Work Controller for classification and Decision Controller for adaptive balancing effectively reduces controller overload scenarios while preserving QoS for critical flows. Emulation results show improved throughput and lower latency compared to a static threshold baseline.

7.2 Future Work

Possible extensions:

- Integrate lightweight ML models to predict controller load trends and migrate proactively.
- Expand to multi-domain SDN and hierarchical controllers for large-scale deployments.
- Add security-aware classification (e.g., policing or quarantine for suspicious flows) integrated into migration decisions.
- Evaluate with Mininet-WiFi and mobility scenarios to replicate access-point association balancing (following Sounni et al.).

References

[1] H. Sounni, N. El Kamoun, and F. Lakrami, “A new SDN-based load balancing algorithm for IoT devices,” Indonesian Journal of Electrical Engineering and Computer Science, vol. 21, no. 2, pp. 1209–1217, Feb. 2021. DOI:10.11591/ijeecs.v21.i2.pp1209-1217.

[2] S. Li, Z. Xin, X. Xu, and Z. Zhang, “Load Balancing Algorithm of SDN Controller Based on Dynamic Threshold,” in 2023 3rd Asia-Pacific Conference on Communications Technology and Computer Science (ACCTCS), 2023. DOI:10.1109/ACCTCS58815.2023.00105.

[3] P. Kamboj, S. Pal, S. Bera, and S. Misra, “QoS-Aware Multipath Routing in Software-Defined Networks,” IEEE Transactions on Network Science and Engineering, vol. 10, no. 2, pp. 723–735, 2023.

[4] M. Shuaib et al., “An Optimized, Dynamic, and Efficient Load-Balancing Framework for Resource Management in the Internet of Things (IoT) Environment,” Sensors (MDPI), 2023.

[5] R. R. Fontes et al., “Mininet-WiFi: Emulating software-defined wireless networks,” 2015 11th Int. Conf. on Network and Service Management (CNSM), 2015.

[6] S. Asadollahi, B. Goswami, and M. Sameer, “Ryu controller’s scalability experiment on software defined networks,” 2018 IEEE International Conference on Current Trends in Advanced Computing (ICCTAC), 2018.

Appendix

1 decision_controller.py

```python
from ryu.base import app_manager
from ryu.controller import ofp_event
from ryu.controller.handler import MAIN_DISPATCHER, DEAD_DISPATCHER, set_ev_cls
from ryu.lib import hub
from ryu.ofproto import ofproto_v1_3
from ryu.lib.packet import packet, ethernet

class DecisionController(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]  # Use OpenFlow 1.3

    def __init__(self, *args, **kwargs):
        super(DecisionController, self).__init__(*args, **kwargs)
        self.controller_loads = {}  # dpid -> load
        self.threshold = 1000000  # initial migration threshold
        self.datapaths = {}
        self.switch_to_controller = {}  # switch dpid to controller dpid
        self.switch_priority = {}  # switch dpid to priority: HIGH/MEDIUM/LOW
        self.monitor_thread = hub.spawn(self._monitor)

    @set_ev_cls(ofp_event.EventOFPStateChange, [MAIN_DISPATCHER, DEAD_DISPATCHER])
    def _state_change_handler(self, ev):
        datapath = ev.datapath
        dpid = datapath.id
        if ev.state == MAIN_DISPATCHER:
            self.datapaths[dpid] = datapath
```

```
            self.logger.info("Registered datapath %s", dpid)
            self.switch_to_controller[dpid] = dpid  # Initially own controller
            self.switch_priority[dpid] = 'LOW'  # Default priority
        elif ev.state == DEAD_DISPATCHER:
            if dpid in self.datapaths:
                self.logger.info("Unregistered datapath %s", dpid)
                self.datapaths.pop(dpid)
            if dpid in self.switch_to_controller:
                self.switch_to_controller.pop(dpid)
            if dpid in self.switch_priority:
                self.switch_priority.pop(dpid)

    @set_ev_cls(ofp_event.EventOFPSwitchFeatures, MAIN_DISPATCHER)
    def switch_features_handler(self, ev):
        datapath = ev.msg.datapath
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser
        # install table-miss flow entry to send unmatched packets to controller
        match = parser.OFPMatch()
        actions = [parser.OFPActionOutput(ofproto.OFPP_CONTROLLER,
                                          ofproto.OFPCML_NO_BUFFER)]
        self.add_flow(datapath, 0, match, actions)
        self.logger.info(f"Switch {datapath.id}: Installed table-miss flow")

    def add_flow(self, datapath, priority, match, actions, buffer_id=None):
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser
        inst = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS, actions)]
        if buffer_id:
```

```
            mod = parser.OFPFlowMod(datapath=datapath, buffer_id=buffer_id,
                                    priority=priority, match=match,
                                    instructions=inst)
        else:
            mod = parser.OFPFlowMod(datapath=datapath, priority=priority,
                                    match=match, instructions=inst)
        datapath.send_msg(mod)

    @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
    def _packet_in_handler(self, ev):
        msg = ev.msg
        datapath = msg.datapath
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser

        pkt = packet.Packet(msg.data)
        eth = pkt.get_protocol(ethernet.ethernet)
        if eth.ethertype == 0x88cc:  # Ignore LLDP packets
            return

        in_port = msg.match['in_port']

        # Example: Flood all packets (replace with real logic)
        actions = [parser.OFPActionOutput(ofproto.OFPP_FLOOD)]
        data = None
        if msg.buffer_id == ofproto.OFP_NO_BUFFER:
            data = msg.data
        out = parser.OFPPacketOut(datapath=datapath, buffer_id=msg.buffer_id,
                                  in_port=in_port, actions=actions,
                                  data=data)
        datapath.send_msg(out)
        self.logger.debug(f"Flooded packet on switch {datapath.id}")
```

```
    def _monitor(self):
        while True:
            self.collect_loads()
            self.adjust_threshold()
            self.check_migration()
            hub.sleep(10)

    def collect_loads(self):
        for dp in self.datapaths.values():
            self.request_stats(dp)

    def request_stats(self, datapath):
        parser = datapath.ofproto_parser
        req = parser.OFPFlowStatsRequest(datapath)
        datapath.send_msg(req)

    @set_ev_cls(ofp_event.EventOFPFlowStatsReply)
    def flow_stats_reply_handler(self, ev):
        dpid = ev.msg.datapath.id
        total_load = 0
        for stat in ev.msg.body:
            weight = 1
            total_load += stat.packet_count * weight
        self.controller_loads[dpid] = total_load
        self.logger.info("Controller load - DPID %s: %d",
           dpid, total_load)

    def adjust_threshold(self):
        if len(self.controller_loads) == 0:
            return
        avg_load = sum(self.controller_loads.values()) /
           len(self.controller_loads)
        new_threshold = int(avg_load * 1.5)
        if new_threshold != self.threshold:
            self.logger.info("Adjusting threshold from %d to
               %d",
                                self.threshold, new_threshold)
            self.threshold = new_threshold
```

```python

    def check_migration(self):
        for dpid, load in self.controller_loads.items():
            if load > self.threshold:
                self.logger.info(f"Overload detected on controller {dpid}, migrating switches...")
                self.migrate_switches(dpid)

    def migrate_switches(self, overloaded_dpid):
        self.logger.info(f"Performing migration from overloaded controller {overloaded_dpid}")

        switches = [sw for sw, ctrl in self.switch_to_controller.items() if ctrl == overloaded_dpid]
        sorted_controllers = sorted(((d, l) for d, l in self.controller_loads.items() if d != overloaded_dpid), key=lambda x: x[1])
        if not sorted_controllers:
            self.logger.info("No other controllers to migrate to")
            return
        target_dpid = sorted_controllers[0][0]

        low_priority_switches = [sw for sw in switches if self.switch_priority.get(sw, 'LOW') == 'LOW']
        if not low_priority_switches:
            self.logger.info("No low priority switches available for migration")
            return

        for sw in low_priority_switches:
            self.logger.info(f"Migrating switch {sw} from controller {overloaded_dpid} to {target_dpid}")
            self.switch_to_controller[sw] = target_dpid
            # Notify external script or orchestration system to perform actual migration
            break  # migrate one at a time

```

```python

if __name__ == "__main__":
    from ryu.cmd import manager
    manager.main()
```

Listing 1: Decision Controller Python code

2 enhanced_traffic_controller.py

```python
from ryu.base import app_manager
from ryu.controller import ofp_event
from ryu.controller.handler import MAIN_DISPATCHER,
   CONFIG_DISPATCHER, set_ev_cls
from ryu.ofproto import ofproto_v1_3
from ryu.lib.packet import packet, ethernet, ipv4, tcp, udp,
   icmp
from ryu.lib import hub
import time

class EnhancedTrafficController(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]
    STATS_PERIOD = 10  # seconds

    def __init__(self, *args, **kwargs):
        super(EnhancedTrafficController,
           self).__init__(*args, **kwargs)
        self.mac_to_port = {}
        self.datapaths = {}
        self.load_stats = {}  # holds weighted load per switch
        self.flow_priorities = {}  # key: (dpid, src, dst,
           in_port), value: priority
        self.monitor_thread = hub.spawn(self._monitor)

    @set_ev_cls(ofp_event.EventOFPSwitchFeatures,
       CONFIG_DISPATCHER)
    def switch_features_handler(self, ev):
        datapath = ev.msg.datapath
        ofproto = datapath.ofproto
```

```python
        parser = datapath.ofproto_parser
        match = parser.OFPMatch()
        actions = [parser.OFPActionOutput(ofproto.OFPP_CONTROLLER,
                                          ofproto.OFPCML_NO_BUFFER)]
        self.add_flow(datapath, 0, match, actions)
        self.logger.info(f"Switch {datapath.id}: Installed table-miss flow")

    def priority_value(self, priority_str):
        priorities = {'HIGH': 30, 'MEDIUM': 20, 'LOW': 10}
        return priorities.get(priority_str, 10)

    def add_flow(self, datapath, priority, match, actions,
                 buffer_id=None, idle_timeout=0, hard_timeout=0):
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser
        if isinstance(priority, str):
            priority_val = self.priority_value(priority)
        else:
            priority_val = priority

        inst = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,
                                             actions)]
        try:
            if buffer_id:
                mod = parser.OFPFlowMod(datapath=datapath, buffer_id=buffer_id,
                                        priority=priority_val, match=match,
                                        instructions=inst, idle_timeout=idle_timeout,
                                        hard_timeout=hard_timeout)
            else:
                mod = parser.OFPFlowMod(datapath=datapath, priority=priority_val,
```

```
                                        match=match, instructions=inst,
                                        idle_timeout=idle_timeout, hard_timeout=hard_timeout)
            datapath.send_msg(mod)
            dpid = datapath.id
            src = match.get('eth_src')
            dst = match.get('eth_dst')
            in_port = match.get('in_port')
            flow_key = (dpid, src, dst, in_port)
            self.flow_priorities[flow_key] = priority_val
            self.logger.info(f"Flow added on DPID {dpid} with priority {priority_val}")
        except Exception as e:
            self.logger.error(f"Failed to add flow: {e}")

    def classify_priority(self, pkt):
        ip_pkt = pkt.get_protocol(ipv4.ipv4)
        if not ip_pkt:
            return 'LOW'
        proto = ip_pkt.proto
        if proto == 6:  # TCP
            tcp_pkt = pkt.get_protocol(tcp.tcp)
            if tcp_pkt and tcp_pkt.dst_port in [80, 443]:
                return 'HIGH'
            else:
                return 'MEDIUM'
        elif proto == 17:  # UDP
            udp_pkt = pkt.get_protocol(udp.udp)
            if udp_pkt and udp_pkt.dst_port == 53:
                return 'HIGH'
            else:
                return 'LOW'
        elif proto == 1:  # ICMP
            return 'MEDIUM'
        else:
            return 'LOW'

    def priority_to_queue(self, priority):
```

```
        if priority == 'HIGH':
            return 1
        elif priority == 'MEDIUM':
            return 2
        else:
            return 3

    @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
    def packet_in_handler(self, ev):
        msg = ev.msg
        datapath = msg.datapath
        dpid = datapath.id
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser
        in_port = msg.match['in_port']

        # Log datapath ID to confirm packet_in from all
           switches including s2
        self.logger.info(f"Packet_in received from DPID
           {dpid}")

        pkt = packet.Packet(msg.data)
        eth = pkt.get_protocol(ethernet.ethernet)
        if eth.ethertype == 0x88cc:
            return
        dst = eth.dst
        src = eth.src

        # Learn MAC address per datapath
        self.mac_to_port.setdefault(dpid, {})
        self.mac_to_port[dpid][src] = in_port

        out_port = self.mac_to_port[dpid].get(dst,
           ofproto.OFPP_FLOOD)

        priority = self.classify_priority(pkt)
        queue_id = self.priority_to_queue(priority)

```

```python
        self.logger.info(f"DPID {dpid} Packet from {src} to {dst} Priority={priority} Queue={queue_id}")

        actions = [parser.OFPActionSetQueue(queue_id),
                   parser.OFPActionOutput(out_port)]

        if out_port != ofproto.OFPP_FLOOD:
            match = parser.OFPMatch(in_port=in_port, eth_dst=dst, eth_src=src)
            self.add_flow(datapath, priority, match, actions,
                          buffer_id=msg.buffer_id, idle_timeout=30)
        data = None
        if msg.buffer_id == ofproto.OFP_NO_BUFFER:
            data = msg.data
        out = parser.OFPPacketOut(datapath=datapath, buffer_id=msg.buffer_id,
                                  in_port=in_port, actions=actions, data=data)
        datapath.send_msg(out)

    @set_ev_cls(ofp_event.EventOFPStateChange, [MAIN_DISPATCHER, CONFIG_DISPATCHER])
    def _state_change_handler(self, ev):
        datapath = ev.datapath
        if ev.state == MAIN_DISPATCHER:
            if datapath.id not in self.datapaths:
                self.logger.info("Register datapath: %s", datapath.id)
                self.datapaths[datapath.id] = datapath
        elif ev.state == CONFIG_DISPATCHER:
            if datapath.id in self.datapaths:
                del self.datapaths[datapath.id]

    def _monitor(self):
        while True:
            for dp in self.datapaths.values():
                self.request_stats(dp)
```

```python
            hub.sleep(self.STATS_PERIOD)

    def request_stats(self, datapath):
        self.logger.debug("Requesting stats from datapath %s", datapath.id)
        parser = datapath.ofproto_parser
        req = parser.OFPFlowStatsRequest(datapath)
        datapath.send_msg(req)

    @set_ev_cls(ofp_event.EventOFPFlowStatsReply, MAIN_DISPATCHER)
    def flow_stats_reply_handler(self, ev):
        datapath = ev.msg.datapath
        dpid = datapath.id
        total_load = 0
        for stat in ev.msg.body:
            src = stat.match.get('eth_src')
            dst = stat.match.get('eth_dst')
            in_port = stat.match.get('in_port')
            flow_key = (dpid, src, dst, in_port)
            priority = self.flow_priorities.get(flow_key, 10)
            weight = 3 if priority >= 30 else 2 if priority >= 20 else 1
            load = stat.packet_count * weight
            total_load += load
        self.load_stats[dpid] = total_load
        self.logger.info(f"DPID {dpid} Load: {total_load}")


if __name__ == "__main__":
    from ryu.cmd import manager
    manager.main()
```

Listing 2: Enhanced Traffic Controller Python code

3 multi_controller_topo.py

```python
import re
```

```python
from mininet.net import Mininet
from mininet.node import RemoteController, OVSSwitch
from mininet.topo import Topo
from mininet.cli import CLI
from mininet.log import setLogLevel

class MultiControllerTopo(Topo):
    def build(self):
        s1 = self.addSwitch('s1', protocols='OpenFlow13')
        s2 = self.addSwitch('s2', protocols='OpenFlow13')
        s3 = self.addSwitch('s3', protocols='OpenFlow13')

        h1 = self.addHost('h1')
        h2 = self.addHost('h2')
        h3 = self.addHost('h3')

        self.addLink(h1, s1)
        self.addLink(h2, s2)
        self.addLink(h3, s3)
        self.addLink(s1, s2)
        self.addLink(s2, s3)

def parse_iperf_output(output):
    bandwidth_match = re.search(r'(\d+\.?\d*)\sMbits/sec',
        output)
    return float(bandwidth_match.group(1)) if bandwidth_match
        else None

def run_ping(net, src, dst, count=5):
    print(f"Running ping test from {src} to {dst}...")
    ping_result = net.get(src).cmd(f'ping -c {count}
        {net.get(dst).IP()}')
    loss_match = re.search(r'(\d+)% packet loss', ping_result)
    latency_match = re.search(r'rtt min/avg/max/mdev =
        ([\d\.]+)/([\d\.]+)/([\d\.]+)/([\d\.]+) ms',
        ping_result)
    loss = int(loss_match.group(1)) if loss_match else None
    latency_avg = float(latency_match.group(2)) if
        latency_match else None
```

```
    return loss, latency_avg

def print_traffic_classification(flow_table_str):
    print("\n=== Traffic Classification and QoS Queue Assignments ===")
    for line in flow_table_str.splitlines():
        if "actions=" in line:
            queue_match = re.search(r"set_queue:(\d+)", line)
            priority_match = re.search(r"priority=(\d+)", line)
            src_match = re.search(r"dl_src=([\w:]+)", line)
            dst_match = re.search(r"dl_dst=([\w:]+)", line)
            in_port_match = re.search(r"in_port=\"?([\w-]+)\"?", line)
            if queue_match and priority_match and src_match and dst_match and in_port_match:
                print(f"Flow with priority {priority_match.group(1)} on port {in_port_match.group(1)}")
                print(f"  Source MAC: {src_match.group(1)}, Destination MAC: {dst_match.group(1)}")
                print(f"  Assigned to QoS Queue: {queue_match.group(1)}\n")

if __name__ == '__main__':
    setLogLevel('info')

    net = Mininet(topo=MultiControllerTopo(), controller=None, switch=OVSSwitch)
    c1 = net.addController('c1', controller=RemoteController, ip='127.0.0.1', port=6633)
    c2 = net.addController('c2', controller=RemoteController, ip='127.0.0.1', port=6634)

    c1.start()
    c2.start()
    net.start()

    net.get('s1').start([c1])
```

```python
    net.get('s2').start([c2])
    net.get('s3').start([c1])

    print("Starting iperf server on h2...")
    net.get('h2').cmd('iperf -s &')

    print("Starting iperf client on h1...")
    iperf_output = net.get('h1').cmd('iperf -c %s -t 10' % net.get('h2').IP())
    print("Raw iperf output:\n", iperf_output)

    bandwidth = parse_iperf_output(iperf_output)
    if bandwidth:
        print(f"Measured Throughput: {bandwidth:.2f} Mbits/sec")
        if bandwidth > 30:
            print("Throughput is strong and meets design expectations for efficient data transfer.")
        else:
            print("Throughput is lower than expected; this might indicate network congestion or inefficiencies.")

    net.get('h2').cmd('pkill iperf')

    # Start iperf server on h1 to generate flows on s2
    print("Starting iperf server on h1...")
    net.get('h1').cmd('iperf -s &')

    print("Starting iperf client on h2...")
    iperf_output_h2_to_h1 = net.get('h2').cmd('iperf -c %s -t 10' % net.get('h1').IP())
    print("Raw iperf output from h2 to h1:\n", iperf_output_h2_to_h1)

    net.get('h1').cmd('pkill iperf')

    loss, latency = run_ping(net, 'h1', 'h3')
    if loss is not None and latency is not None:
```

```
        print(f"Ping Test Results from h1 to h3: Packet Loss
            = {loss}%, Average Latency = {latency} ms")
        if loss == 0:
            print("Zero packet loss indicates very reliable
                network connectivity.")
        else:
            print("Packet loss detected; reliability can be
                improved.")

    # Dump and parse flow tables for s1, s2, s3
    for sw in ['s1', 's2', 's3']:
        flows = net.get(sw).dpctl('dump-flows -O OpenFlow13')
        with open(f'flows_{sw}.txt', 'w') as f:
            f.write(flows)
        print(f"Flow table on {sw}:")
        print(flows)
        print_traffic_classification(flows)
        print(f"Dumped flows for {sw} to flows_{sw}.txt\n")

    CLI(net)
    net.stop()
```

Listing 3: Multi-Controller Mininet Topology Python script

4 sdn_dashboard.py

```
import streamlit as st
import re
import pandas as pd

def read_file(filename):
    try:
        with open(filename) as f:
            return f.read()
    except FileNotFoundError:
        return None

def parse_flow_table(flow_table_str):
```

```
    entries = []
    for line in flow_table_str.strip().splitlines():
        queue_match = re.search(r"set_queue:(\d+)", line)
        priority_match = re.search(r"priority=(\d+)", line)
        src_match = re.search(r"dl_src=([\w:]+)", line)
        dst_match = re.search(r"dl_dst=([\w:]+)", line)
        in_port_match = re.search(r'in_port="?([\w-]+)"?',
           line)
        if queue_match and priority_match and src_match and
           dst_match and in_port_match:
            entries.append({
                "Priority": priority_match.group(1),
                "In Port": in_port_match.group(1),
                "Source": src_match.group(1),
                "Destination": dst_match.group(1),
                "QoS Queue": queue_match.group(1)
            })
    return entries

# Page title and layout
st.set_page_config(page_title="SDN IoT Gateway",
   layout="wide")
st.title("SDN IoT Gateway")

# Performance Metrics Section
st.header("Performance Metrics")

# Read iperf and ping output from files
iperf_text = read_file("iperf.txt")
ping_text = read_file("ping.txt")

# Parse throughput from iperf output
throughput_line = None
if iperf_text:
    for line in iperf_text.splitlines():
        if 'Mbits/sec' in line:
            throughput_line = line.strip()
            break

```

```python
# Parse packet loss and latency via regex on ping line
packet_loss_val = None
latency_val = None
if ping_text:
    for line in ping_text.splitlines():
        if "Packet Loss =" in line:
            packet_loss_line = line.strip()
            loss_match = re.search(r"Packet Loss =
               ([\d\.]+)%", packet_loss_line)
            latency_match = re.search(r"Average Latency =
               ([\d\.]+) ms", packet_loss_line)
            if loss_match:
                packet_loss_val = loss_match.group(1)
            if latency_match:
                latency_val = latency_match.group(1)

# Display Throughput
if throughput_line:
    try:
        throughput_val = throughput_line.split()[-2]
        st.metric("Throughput (Mbits/sec)", throughput_val)
    except Exception:
        st.info("Throughput data format issue")
else:
    st.info("Throughput data not available yet.")

# Display Packet Loss
if packet_loss_val is not None:
    st.metric("Packet Loss (%)", packet_loss_val)
else:
    st.info("Packet loss data not available yet.")

# Display Average Latency
if latency_val is not None:
    st.metric("Average Latency (ms)", latency_val)
else:
    st.info("Latency data not available yet.")

# Flow Table summary section
```

```python
st.header("Flow Table Summary")
for switch in ["s1", "s2", "s3"]:
    st.subheader(f"Switch {switch.upper()}")
    flow_text = read_file(f"flows_{switch}.txt")
    if flow_text:
        flow_entries = parse_flow_table(flow_text)
        if flow_entries:
            df = pd.DataFrame(flow_entries)
            st.table(df)
            st.caption(f"Total parsed flows:
                {len(flow_entries)}")
        else:
            if switch == "s2":
                st.info("Switch s2 currently operates mostly
                    with default flow rules; no detailed flow
                    entries detected.")
            else:
                st.info(f"No detailed flow entries found for
                    switch {switch}.")
    else:
        st.info(f"Flow data for switch {switch} is not
            available yet.")
```

Listing 4: Streamlit Dashboard Python code