End-to-End SDN Management: Reactive Flow Control, Topology Discovery, and Real-Time Visualization with PHP and Ryu Controllers

Shweta M. K.*[1], Zahira G.*[2],

*Student, Computer Engineering Department, San Jose State University
San Jose, USA

Abstract— **Software-Defined Networking (SDN) enables centralized control and programmability of modern networks, allowing for enhanced monitoring, management, and security. This project presents the design and implementation of a real-time SDN controller application that provides full network visibility. This paper introduces a SDN controller application designed to provide real-time network visibility, topology discovery, and flow monitoring. The project implements two controllers: a Python-based controller using the Ryu framework and a PHP-based OpenFlow 1.3 controller. The Ryu controller leverages LLDP for dynamic topology discovery, collects live flow statistics, and exposes RESTful APIs for visualization and flow tracing, enabling detection of anomalies such as abnormal bandwidth usage. The PHP controller demonstrates protocol-level topology discovery by injecting and parsing LLDP packets, exporting the network graph for visualization. Together, these implementations offer a comprehensive platform for SDN monitoring, management, and research, facilitating efficient network troubleshooting and operational insight. Project source code can be found at: https://github.com/ghaz-zahira/cmpe210-NetworkVisualization for both implementations.**

***Keywords*— Software-Defined Networking (SDN); OpenFlow; Ryu; network visibility; topology discovery; LLDP; flow statistics; traffic monitoring; flow tracing; REST API; network visualization; programmable networks.**

I. Introduction

Software-Defined Networking (SDN) has emerged as a transformative paradigm in modern networking, enabling centralized, programmable control over network infrastructure through the decoupling of the control and data planes. This architectural shift facilitates dynamic network management, rapid innovation, and enhanced visibility into network operations. However, realizing the full potential of SDN in operational environments requires robust controller applications that can provide real-time insights into network topology, traffic flows, and potential anomalies.

This project addresses these needs by developing SDN controller applications that deliver comprehensive, real-time network visibility. The controllers are designed to automatically discover the network topology using Link Layer Discovery Protocol (LLDP), monitor live flow and port statistics, and enable flow tracing between hosts. By exposing this information through RESTful APIs and exporting topology data for visualization, the controllers empower network operators and researchers to monitor, troubleshoot, and optimize SDN environments effectively.

Two distinct controller implementations are presented: a PHP-based OpenFlow 1.3 controller focused on protocol-level topology discovery and a Python-based controller leveraging the Ryu framework. The PHP controller demonstrates the core mechanisms of OpenFlow-based topology discovery and visualization while the Ryu controller integrates features such as live traffic monitoring, and flow tracing. Together, these implementations provide real-time visibility into a network.

II. Technical Approaches - PHP Controller

A. *Socket-Based Server Core*

The controller runs a TCP listener on port 6653, the IANA-assigned OpenFlow port. For each connecting switch, it establishes a non-blocking socket and tracks per-switch state including:

- Connection phase (HELLO - FEATURES_REQUEST - READY)
- DPID (Datapath Identifier)
- Port list
- Timestamp of last LLDP transmission

This design avoids thread creation, relying instead on stream_select() for lightweight multiplexing.

B. *OpenFlow Message Pipeline*

The controller implements a minimal set of OpenFlow 1.3 messages required for topology discovery

Purpose	Message

Protocol negotiation	HELLO
Capability discovery	FEATURES_REQUEST/FEATURES_REPLY
Port enumeration	MULTIPART_PORT_DESC request/reply
Topology packet delivery	PACKET_OUT (LLDP frames)
Event reception	PACKET_IN
Connectivity	ECHO REQUEST/ ECHO REPLY
Error handling	ERROR messages

The controller constructs and parses messages manually using PHP's binary pack() and unpack() functions, showcasing direct protocol-level control.

C. LLDP as the Topology Discovery Manager

LLDP is the de facto method for SDN controllers to infer physical topology. When LLDP frames travel between adjacent switches, the controller learns the mapping:

(src_dpid : src_port) -> dst_dpid

The controller periodically emits LLDP frames from every physical port. When a switch receives an LLDP frame, it forwards the frame to the controller as a PACKET_IN message, allowing the controller to deduce adjacency.

D. Topology Data Model

The topology is stored as an in-memory associative array keyed by:

(src_dpid : src_port) -> dst_dpid

Each discovery event updates a timestamp, enabling pruning of stale links. A structured JSON file is generated containing:

- Nodes (unique switch DPIDs)
- Edges (directed LLDP adjacency)
- Metadata (port labels, timestamp)

This JSON representation is compatible with widely used visualization libraries such as vis.js.

III. Implementation Details - PHP Controller

A. Connection Lifecycle

1. HELLO Exchange

Upon accepting a new TCP connection, the controller sends:

OFPT_HELLO

When the switch responds with HELLO, the controller proceeds to request capabilities

2. Feature Discovery

The controller then emits:

OFPT_FEATURES_REQUEST

The switch replies with a FEATURES_REPLY containing the Datapath ID (DPID). This is extracted using a custom parser which reads the first 8 bytes of the payload and interprets them as a 64-bit big-endian integer.

the DPID becomes the unique identifier for that switch.

3. Port Description Retrieval

OpenFlow 1.3 no longer embeds port information in the FEATURES_REPLY. Instead, the controller requests:

OFPT_MULTIPART_REQUEST

The reply contains an array of ofp_port structures (each 64 bytes), from which the controller extracts the valid port numbers, excluding reserved ports such as:

- OFPP_CONTROLLER
- OFPP_LOCAL
- Special port constants > 0xffffff00

B. LLDP Packet Construction

The controller manually constructs a valid LLDP Ethernet frame:

1. Destination MAC: LLDP multicast
2. Source MAC: Derived from the switch DPIDs lower 6 bytes
3. Ethertype: 0x88cc (LLDP)
4. LLDP TLVs:
    - Chassis ID (subtype: MAC address)
    - Port ID (subtype: local port number)
    - TTL
    - End of LLDPDU

To maintain correctness, TLVs use the format:

type (7 bits) | length (9 bits)

This guarantees interoperability with real OpenFlow switches (OVS, hardware switches, Mininet).

C. PACKET_OUT Transmission

LLDP payloads are encapsulated into PACKET_OUT messages using:

buffer_id = OFP_NO_BUFFER

out_port = actual physical port

Each switch receives LLDP frames every 5 seconds. This periodicity prevents high LLDP traffic while ensuring timely topology updates.

D. PACKET_IN Parsing and LLDP Interpretation

Whenever LLDP frames are received from a switch:

1. The controller extracts the embedded Ethernet frame from the PACKET_IN payload.
2. It verifies the Ethernet (0x88cc).
3. TLVs are parsed using a sequential decoder that extracts Source DPID (from MAC) and Source Port ID
4. The destination DPID is known from the socket that received the PACKET_IN.

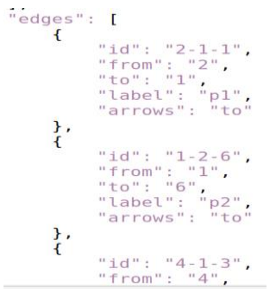
If a new adjacency is found, the controller updates its topology map and regenerates the JSON topology file.

E. Topology Pruning

To prevent outdated links from persisting, the controller removes any LLDP edges older than 20 seconds. This ensures the topology model reflects real-time network state.

E. JSON Topology Export

The controller writes the JSON file using the structure:



```
"edges": [
    {
        "id": "2-1-1",
        "from": "2",
        "to": "1",
        "label": "p1",
        "arrows": "to"
    },
    {
        "id": "1-2-6",
        "from": "1",
        "to": "6",
        "label": "p2",
        "arrows": "to"
    },
    {
        "id": "4-1-3",
        "from": "4",
```

The file is written to */var/www/html/topology.json.* This allows immediate integration with front-end dashboards.

IV. Technical Approaches - Ryu Controller & Dashboard

A. System Architecture

The system architecture consists of four major components:

1. *Ryu SDN Controller* – Runs the custom Python-based application.
2. *OpenFlow Switches* – Forward packets based on flow rules installed by the controller.
3. *Mininet Emulator* – Emulates the SDN network topology.
4. *Web-Based Dashboard* – Retrieves data from the controller via REST APIs and displays network state

The controller communicates with switches using OpenFlow v1.3, while the dashboard communicates with the controller using HTTP-based REST endpoints. The architecture supports bi-directional interaction: switches report events to the controller, and the controller sends flow rules and packet injections to the switches.

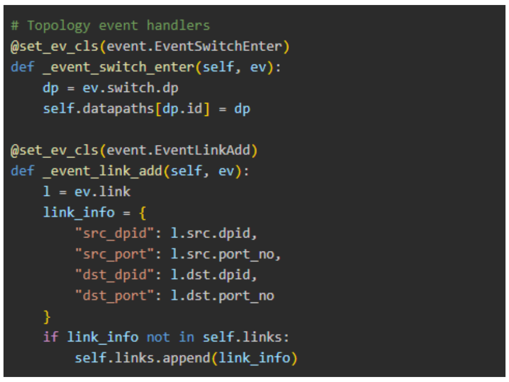
B. Controller Design

The controller was implemented using the Ryu framework and leverages OpenFlow 1.3 for switch communication. The controller performs topology discovery by listening to LLDP-based events (EventSwitchEnter, EventLinkAdd, EventLinkDelete) and maintains a dynamic view of switches and inter-switch links. Host discovery is achieved by learning MAC-to-port mappings from ARP and IPv4 packets, while IP-to-MAC associations are also tracked for flow tracing.

Flow installation is handled reactively: upon receiving a PacketIn event, the controller learns host locations and installs flows to optimize forwarding. All installed flows are recorded, deduplicated, and exposed via a REST API for visualization. The controller periodically requests flow statistics from all datapaths, normalizes the results, and makes them available for dashboard queries. Anomalies such as abnormal bandwidth usage can be detected by analyzing these statistics.

C. Topology Discovery

We leverage Ryu's topology events and LLDP packets to discover switches and links. The controller listens for EventSwitchEnter, EventLinkAdd, and EventLinkDelete to dynamically update its view of the network. Host discovery is achieved by learning MAC-to-port mappings from ARP and IPv4 packets.



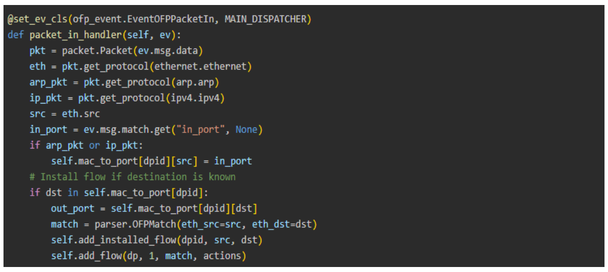
```
# Topology event handlers
@set_ev_cls(event.EventSwitchEnter)
def _event_switch_enter(self, ev):
    dp = ev.switch.dp
    self.datapaths[dp.id] = dp

@set_ev_cls(event.EventLinkAdd)
def _event_link_add(self, ev):
    l = ev.link
    link_info = {
        "src_dpid": l.src.dpid,
        "src_port": l.src.port_no,
        "dst_dpid": l.dst.dpid,
        "dst_port": l.dst.port_no
    }
    if link_info not in self.links:
        self.links.append(link_info)
```

Fig. 1 Switch and link discovery using Ryu topology events.

D. Host Learning and Flow Installation

The controller learns host locations by inspecting ARP and IPv4 packets. When a packet arrives, it updates the MAC-to-port table and installs flows to optimize forwarding. Installed flows are recorded and deduplicated for dashboard visualization.



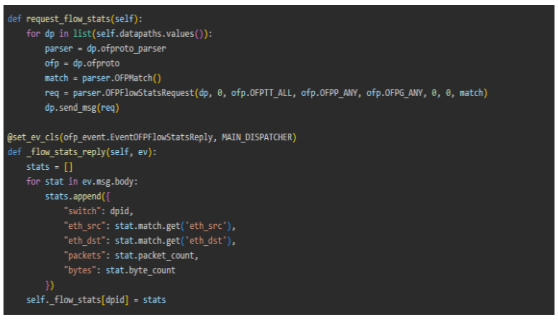
```
@set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
def packet_in_handler(self, ev):
    pkt = packet.Packet(ev.msg.data)
    eth = pkt.get_protocol(ethernet.ethernet)
    arp_pkt = pkt.get_protocol(arp.arp)
    ip_pkt = pkt.get_protocol(ipv4.ipv4)
    src = eth.src
    in_port = ev.msg.match.get("in_port", None)
    if arp_pkt or ip_pkt:
        self.mac_to_port[dpid][src] = in_port
    # Install flow if destination is known
    if dst in self.mac_to_port[dpid]:
        out_port = self.mac_to_port[dpid][dst]
        match = parser.OFPMatch(eth_src=src, eth_dst=dst)
        self.add_installed_flow(dpid, src, dst)
        self.add_flow(dp, 1, match, actions)
```

Fig. 2 Host learning and flow installation logic.

E. RESTful API for Real-Time Data

The controller exposes a RESTful API using Ryu's WSGI module. Endpoints provide real-time topology (/topology), installed flows (/flows), and flow statistics (/flow_stats). Additional endpoints allow for active probing: /pingall injects ARP and ICMP packets to trigger host responses and update the controller's view. For instance, the /flow_stats endpoint periodically requests flow statistics from all switches, normalizes the results, and exposes them. This enables the dashboard to visualize live traffic metrics. A table depicting the intended results of the endpoints can also be found below:

Endpoint	**Method**	**About**	**Output**
`/flows`	GET	Returns learned MAC address tables and all installed flow entries for each switch.	JSON object containing switches, learned MAC tables, and installed flows
`/topology`	GET	Returns real-time network topology including switches, host attachments, and inter-switch links.	JSON object containing switch list, host mappings, and discovered links
`/flow_stats`	GET	Requests and returns live OpenFlow flow statistics from all connected switches.	JSON array containing per-flow packet count, byte count, duration, and match fields
`/pingall`	POST	Inject ICMP echo requests between all discovered host pairs to verify end-to-end connectivity.	JSON object with status and number of packets sent



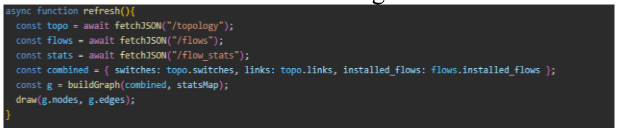
```python
def request_flow_stats(self):
    for dp in list(self.datapaths.values()):
        parser = dp.ofproto_parser
        ofp = dp.ofproto
        match = parser.OFPMatch()
        req = parser.OFPFlowStatsRequest(dp, 0, ofp.OFPTT_ALL, ofp.OFPP_ANY, ofp.OFPG_ANY, 0, 0, match)
        dp.send_msg(req)

@set_ev_cls(ofp_event.EventOFPFlowStatsReply, MAIN_DISPATCHER)
def _flow_stats_reply(self, ev):
    stats = []
    for stat in ev.msg.body:
        stats.append({
            "switch": dpid,
            "eth_src": stat.match.get('eth_src'),
            "eth_dst": stat.match.get('eth_dst'),
            "packets": stat.packet_count,
            "bytes": stat.byte_count
        })
    self._flow_stats[dpid] = stats
```

Fig. 3 Flow statistics request and reply handling.

F. *Visualization Dashboard*

The dashboard is implemented as a web application using Vis.js. It fetches topology and flow data from the controller, rendering switches, hosts, and flows as an interactive graph. Users can inspect per-switch flow tables, trace flows between hosts, and potentially identify anomalies such as abnormal bandwidth usage.



```javascript
async function refresh(){
  const topo = await fetchJSON("/topology");
  const flows = await fetchJSON("/flows");
  const stats = await fetchJSON("/flow_stats");
  const combined = { switches: topo.switches, links: topo.links, installed_flows: flows.installed_flows };
  const g = buildGraph(combined, statsMap);
  draw(g.nodes, g.edges);
}
```

Fig. 4 Dashboard fetching and rendering logic.

V. DISCUSSION

A. *Advantages of a PHP-Based Controller*

Although PHP is not commonly used in SDN development, this implementation demonstrates several advantages:

- Extremely lightweight runtime
- No external dependencies
- High portability (runs anywhere PHP CLI is available)
- Easy debugging with printable hex dumps.
- Ideal for educational and experimental environments.

B. *PHP-Based Controller- Limitations*

- Not intended for large scale production environments.
- Missing advanced controller features (routing, ACLs, QoS, group tables)
- Single-threaded architecture limits scalability

C. *PHP-Based Controller- Extensibility*

- Flow-rule installation for packet-forwarding applications
- Switch statistics polling
- Custom LLDP variants
- Event-driven northbound APIs.

D. *Advantages of a Ryu-Based Controller*

- Standard choices for SDN to use Python
- Rich networking libraries with built in support for packet parsing (Ethernet, ARP, IPv4, ICMP) and OpenFlow 1.3 simplifies controller logic and flow management
- High portability (runs on any system with Python and Ryu installed)
- Extensive logging and exception handling make it straightforward to trace network events and controller actions
- Ideal for educational and experimental environments

E. *Ryu-Based Controller Limitations*

- Not intended for large scale production environments
- Missing advanced controller features (routing ACLs, QoS, group tables)
- Single threaded architecture limits scalability
- Polling based statistics

F. *Ryu-Based Controller- Extensibility*

- Custom flow rule installation
- Enhanced topology discovery allowing for advanced link detection and richer topology mapping
- Built in support for flow and port statics polling can be expanded for deeper traffic analysis

- Controller-driven ARP and ICMP injection enables active probing, flow tracing, and stress testing of network paths
- REST endpoints are easily expandable for integration with northbound APIs
- Event-driven extensions

VI. Conclusions

This project demonstrates the feasibility and benefits of real-time network visualization and monitoring using SDN controller applications. By implementing both a PHP-based OpenFlow controller and a Python-based Ryu controller, we highlight the flexibility and extensibility of SDN for educational and research purposes. The controllers provide dynamic topology discovery, live flow statistics, and RESTful APIs for visualization, enabling efficient network troubleshooting and operational insight. While both implementations are limited in scalability and advanced features, they serve as effective platforms for learning, experimentation, and further development.

A. Potential Extensions

- More interactive display:
  - Enable direct manipulation of the topology through the dashboard (e.g., removing hosts or switches).
  - Support saving snapshots of network state at different moments, which is valuable for security and auditing in real-world applications.
  - Provide a live view for continuous monitoring and immediate feedback.
- Scalability improvements:
  - Expand probe IP ranges and adjust timeouts for larger topologies.
  - Enhance traffic generation to better emulate production environments, moving beyond synthetic controller-injected traffic.
  - Future work can extend these controllers with richer analytics, advanced flow management, and integration with production-scale SDN environments.

References

[1] Project Source Code: https://github.com/ghaz-zahira/cmpe210-NetworkVisualization

[2] Demo Video: https://youtu.be/MgC9URY9Uos

[3] Open Network Foundation, *OpenFlow Switch Specification Version 1.3.5*, ONF, March 2015.

[4] IEEE Association, *IEEE Standard 802.1AB-2016: Station and Media Access Control Connectivity Discovery*, IEEE, 2016.

[5] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar et al, "*OpenFlow: Enabling Innovation in Campus Networks*" ACM SIGCOMM Computer Communication Review, vol. 38, no. 2, pp. 69-74, 2008.

[6] D. Kreutz, F. M. Ramos, P. Verissimo, C. Esteve, S. Azodolmolky, S. Uhlig, "*Software-Defined Networking: A Comprehensive Survey*," Proceedings of the IEEE, vol. 103, no. 1, pp. 14-76, 2015.

[7] A. Lara, A. Kolasani, B. Ramamurthy, "*Network Innovation Using OpenFlow: A Survey*," IEEE Communication Surveys & Tutorials, vol. 16, no. 1, pp. 493-512, 2014.

[8] M. Jammal, T. Singh, A. Shami, R. Asal, Y. Li, "*Software Defined Networking: State of the Art and Research Challenges*," Computer Networks, vol. 72, pp. 74-98, 2014.

[9] R. Sherwood, G. Gibb, K. K. Yapp, M. Casado, N. McKeown, G. Parulkar, "*FlowVisor: A Network Virtualization Layer*," OpenFlow Switch Consortium, Tech. Rep., 2009.

[10] R. Enns, M. Bjorklund, J. Schoenwaelder, A. Biermann, "*Network Configuration Protocol (NETCONF)*," RFC 6241, IETF, June 2011.

[11] B. Pfaff et al, "*The Design and Implementation of Open vSwitch*," in USENIX NSDI, 2015.

[12] S. Jain et al, "*B4: Experience with a Globally Deployed Software Defined WAN*," ACM SIGCOMM, 2013.