CS 331 Computer Networks
Modifying Southbound Communication Protocol in SDN: TCP to UDP for Ryu–OVS

Anaparthi Venkata Subrahmanya Manoj
23110025
Computer Science and Engineering
Indian Institute of Technology Gandhinagar
manoj.anaparthi@iitgn.ac.in

Abstract

We present the design and implementation of a UDP-based transport for SDN southbound communication between the Ryu controller and Open vSwitch (OVS). The goal is to enable direct UDP communication, replacing the default TCP channel to reduce control-plane latency and connection overhead. We modified the Ryu framework's `controller.py` with a `DatapathUDP` class implementing sequence number tracking, retransmission with timeout, and duplicate detection via a sliding window. OVS 3.1.0 was compiled with custom `stream-udp.c` and `vconn-udp.c` implementations. The prototype demonstrates complete OpenFlow handshake, correct handling of `SET_CONFIG`, PACKET_IN/OUT processing, and bidirectional ECHO keepalives over UDP. End-to-end validation using Mininet confirmed successful host connectivity, while benchmarking (50 samples) showed comparable latency between TCP and UDP in localhost conditions. The work validates UDP as a feasible southbound transport and provides documented reference code for further research.

1 Introduction

Software-Defined Networking (SDN) separates the control plane from the data plane, allowing centralized software controllers to program forwarding behaviour in switches. Controllers and switches typically exchange control messages using the OpenFlow protocol over TCP, which guarantees ordered, reliable delivery but incurs overhead from connection setup, retransmissions, and congestion control [5].

For frequent, small control messages (e.g., `PACKET_IN`), TCP's stream semantics and connection management can add measurable latency and head-of-line blocking. This project investigates replacing TCP with the connectionless User Datagram Protocol (UDP) for OpenFlow 1.3 southbound channels. UDP removes the transport-layer connection handshake and per-connection state, simplifying framing (one datagram per OpenFlow message) and enabling immediate transmission of control messages.

Our objectives are:

- Analyze Ryu and OVS internals to identify integration points for a UDP transport
- Implement UDP OpenFlow 1.3 transport in both controller and switch
- Design a lightweight reliability layer with XID-based retransmission
- Validate full OpenFlow message exchange over UDP via Mininet
- Benchmark TCP vs UDP latency to assess performance characteristics

The implementation includes:

- Modified Ryu framework (`ryu/controller/controller.py`) with `DatapathUDP` class
- Compiled OVS 3.1.0 with `stream-udp.c` and `vconn-udp.c`
- XID-based reliability layer (retransmission, duplicate suppression)
- End-to-end Mininet validation (`e2e_tests/mininet_ryu_udp.py`)
- TCP vs UDP benchmarking suite (`e2e_tests/benchmark_tcp_udp.py`)

2 Background and Related Work

2.1 Software-Defined Networking Architecture

Software-Defined Networking (SDN) fundamentally restructures traditional network architectures by decoupling the control plane (network intelligence) from the data plane (packet forwarding). This separation enables:

- **Centralized Control**: Single controller manages multiple switches
- **Programmability**: Network behavior defined in software
- **Dynamic Reconfiguration**: Flow rules updated in real-time
- **Vendor Independence**: Standard protocols replace proprietary solutions

2.2 OpenFlow Protocol

OpenFlow, standardized by the Open Networking Foundation (ONF), defines the communication protocol between SDN controllers and switches [6]. The protocol operates over TCP port 6653 and consists of three message types:

1. **Controller-to-Switch**: Configuration and flow management (FEATURES_REQUEST, FLOW_MOD, PACKET_OUT)
2. **Asynchronous**: Event notifications from switch (PACKET_IN, PORT_STATUS, FLOW_REMOVED)
3. **Symmetric**: Bidirectional messages (HELLO, ECHO, ERROR)

OpenFlow 1.3, used in this work, introduces multiple flow tables, group tables, and enhanced metering capabilities. Message sizes range from 8 bytes (HELLO) to several kilobytes (PACKET_IN with payload), well within UDP's 65,507-byte limit.

2.3 TCP in SDN: Benefits and Limitations

TCP provides critical features for SDN control channels:

- **Reliability**: Guaranteed delivery with retransmission
- **Ordering**: Sequential message delivery
- **Flow Control**: Congestion management
- **Error Detection**: Checksum validation

However, TCP introduces performance overhead:

- **Connection Setup**: Three-way handshake adds 1–1.5 RTT delay
- **Head-of-Line Blocking**: Lost packets stall entire stream
- **Acknowledgment Overhead**: Every segment requires ACK
- **State Management**: Per-connection buffers consume memory

For high-frequency messages like `PACKET_IN` (triggered on every table miss), these overheads accumulate significantly. In data-center environments with sub-millisecond RTTs, TCP's connection management becomes proportionally expensive.

2.4 Related Work

Recent research by Kumar & Dezfouli [8] demonstrated that UDP-based transports like QUIC can reduce message latency by up to 45% and communication overhead by up to 40% in SDN environments. Their QuicSDN architecture replaces TCP with QUIC (UDP + reliability layer), showing:

- Reduced handshake latency (1-RTT vs 2-RTT for TCP)
- Per-stream multiplexing (eliminates head-of-line blocking)
- Improved flow control mechanisms

While QuicSDN adds a reliability layer atop UDP, few works have explored using plain UDP for OpenFlow communication. Our work investigates this fundamental question: *Can OpenFlow operate over UDP with a lightweight application-layer reliability mechanism?*

3 Methodology and Implementation

Our approach was executed in multiple phases, targeting both controller and switch components while maintaining OpenFlow 1.3 protocol compatibility.

3.1 Project Structure

Listing 1: Repository Structure

```
.
        openvswitch-3.1.0/
              lib/
                  stream-udp.c
                  vconn-udp.c
        ryu/
              ryu/controller/
                  controller.py
        e2e_tests/
            mininet_ryu_udp.py
            benchmark_tcp_udp.py
            ofp_message_test.py
            ofp_message_test_app.py
            artifacts/
                benchmark_summary.csv
                latency_boxplot.png
                latency_comparison.png
```

3.2 Experimental Environment

Table 1: Environment Configuration

Component	Version
OS	Ubuntu 22.04 LTS
Python	3.10
OVS	3.1.0 (compiled with UDP)
Mininet	2.3.0
OpenFlow	1.3
Test Port	6653 (RFC 7047)

3.3 Architecture Analysis

Detailed code analysis identified key modification points:
Controller Side (Ryu Framework):

- `ryu/controller/controller.py`: Uses `StreamServer` from eventlet, defaults to TCP
- Event-driven architecture based on eventlet greenlets
- Hub abstraction layer for socket operations

Switch Side (Open vSwitch):

- Stream abstraction layer: `lib/stream-tcp.c`, `lib/stream.c`
- Virtual connection layer: `lib/vconn-stream.c`, `lib/vconn.c`
- Connection manager: `ofproto/connmgr.c`

3.3.1 OVS Layered Architecture

OVS implements a clean abstraction hierarchy for network connections:

Listing 2: OVS Connection Abstraction

```
Application Layer (vswitchd)
          |
     vconn Layer (lib/vconn.c)
          |
   vconn-stream Layer (lib/vconn-stream.c)
          |
     Stream Layer (lib/stream-tcp.c)
          |
    Socket Layer (OS)
```

This architecture means UDP support requires only two new files (`stream-udp.c`, `vconn-udp.c`), leaving higher layers unchanged.

3.3.2 Message Framing Analysis

TCP's stream-based nature requires application-layer message framing. OpenFlow messages begin with an 8-byte header containing version, type, length, and transaction ID (XID). TCP implementations must read the header, parse the length, then read remaining bytes, handling partial reads across multiple `recv()` calls.

UDP's message-based nature naturally maps to OpenFlow message boundaries—each `recvfrom()` returns exactly one complete OpenFlow message, eliminating framing complexity.

3.4 Ryu Controller Modifications

We modified `ryu/controller/controller.py` to add UDP transport support:

- Added `-ofp-listen-transport udp` command-line option
- Created `DatapathUDP` class for UDP-based datapath handling
- Implemented reliability layer with retransmission and duplicate detection

3.4.1 DatapathUDP Class

The `DatapathUDP` class handles UDP communication with sequence-based reliability:

Listing 3: DatapathUDP Implementation

```
class DatapathUDP(ofproto_protocol.ProtocolDesc):
    """UDP-backed␣OpenFlow␣datapath␣with
␣␣␣␣␣␣␣sequence-based␣reliability."""

    RETRANSMIT_TIMEOUT = 1.0  # seconds
    MAX_RETRANSMITS = 3
    SEQ_WINDOW_SIZE = 1000    # Duplicate detection
        window

    def __init__(self, socket, address, ...):
        self._server_socket = socket
        self.address = address
        # Pending messages: XID -> (buf, time,
            count, seq)
        self._pending_msgs = {}

        # Sequence numbers for reliability
        self._send_seq = 0           # Next seq to
            send
        self._recv_seq_expected = 0  # Expected seq
        self._recv_seq_seen = set()  # Sliding
            window
        self._seq_stats = {
            'sent': 0, 'received': 0,
            'duplicates': 0, 'out_of_order': 0,
            'retransmits': 0
        }
```

3.4.2 Reliability Layer

The reliability layer provides sequence-based tracking:

1. **Sequence Numbers**: Each message assigned monotonically increasing sequence number; enables ordering detection
2. **Retransmission**: Background thread checks `_pending_msgs` every 100ms; retransmits after 1.0s timeout (max 3 attempts)
3. **Duplicate Detection**: Sliding window of 1000 sequence numbers (`_recv_seq_seen`); duplicates logged and dropped
4. **Out-of-Order Detection**: Compares received sequence to `_recv_seq_expected`; logs out-of-order arrivals
5. **Statistics Tracking**: Maintains counters for sent, received, duplicates, out-of-order, and retransmits

Listing 4: Retransmission Loop

```
def _retransmit_loop(self):
    while self.is_active:
        now = _time.time()
        with self._pending_lock:
            for xid, (buf, send_time, count, seq) \
                in list(self._pending_msgs.items())
                    :
                if now - send_time > self.
                    RETRANSMIT_TIMEOUT:
                    if count < self.MAX_RETRANSMITS
                        :
                        self._udp_socket.sendto(buf
                            , self._udp_addr)
                        self._pending_msgs[xid] = (
                            buf, now, count+1, seq)
                    else:
                        del self._pending_msgs[xid]
        hub.sleep(0.1)
```

3.5 OVS UDP Transport

We added two files to OVS 3.1.0:

3.5.1 stream-udp.c

Implements the OVS stream interface for UDP:

Listing 5: UDP Stream Structure

```
struct udp_stream {
    struct stream stream;
    int fd;
    struct sockaddr_storage remote;
    socklen_t remote_len;
    bool connected;
};
```

Key functions:

- `udp_open()`: Creates UDP socket via `inet_open_active(SOCK_DGRAM, ...)`
- `udp_recv()`: Receives datagrams via `recvfrom()`
- `udp_send()`: Sends datagrams via `sendto()`

3.5.2 vconn-udp.c

Wraps stream-udp for OpenFlow virtual connections. Registers `udp_vconn_class` enabling:

```
ovs-vsctl set-controller br0 udp:127.0.0.1:6653
```

3.5.3 Build Process

Listing 6: Building OVS with UDP

```
cd openvswitch-3.1.0
./boot.sh
./configure
make -j$(nproc)
sudo make install
```

4 Results

4.1 Functional Validation

The Mininet test (`e2e_tests/mininet_ryu_udp.py`) validates:

- Ryu controller listening on UDP port 6653
- OVS connecting via `udp:127.0.0.1:6653`
- OpenFlow handshake (HELLO, FEATURES_REQUEST/REPLY, SET_CONFIG)
- PACKET_IN triggering flow installation
- Successful ping between hosts through OVS

Listing 7: Running the Demo

```
sudo python3 e2e_tests/mininet_ryu_udp.py
```

4.2 Benchmarking

The benchmark (`e2e_tests/benchmark_tcp_udp.py`) measures latency for Echo RTT, Flow-Mod, and Stats Request (50 samples each).

Observations:

Table 2: TCP vs UDP Latency (Localhost, 50 samples)

Metric	TCP (ms)	UDP (ms)	TCP StdDev	UDP StdDev
Echo RTT	7.53	8.21	3.82	6.36
Flow-Mod	13.19	13.34	1.53	1.20
Stats	4.47	5.07	1.68	1.36

- TCP showed slightly lower mean latency in localhost (optimized kernel stack)
- UDP exhibited lower standard deviation for Flow-Mod (1.20 vs 1.53 ms)
- Performance was nearly identical for the critical Flow-Mod operation
- Localhost testing minimizes UDP's advantages (no real network latency/loss)

4.3 Generated Artifacts

Benchmarks output to `e2e_tests/artifacts/`:

- `benchmark_summary.csv`: Raw latency data
- `latency_boxplot.png`: Distribution comparison
- `latency_comparison.png`: Bar chart with error bars

5 Discussion

5.1 Architecture Validation

Our implementation aligns with recent industry research. Comparison with QuicSDN [8]:

Table 3: Architecture Comparison: This Work vs QuicSDN

Layer	This Work	QuicSDN
Application	OpenFlow 1.3	OpenFlow 1.3
Reliability	Sequence + XID	QUIC protocol
Transport	UDP (DGRAM)	UDP (DGRAM)
Ordering	Detected (logged)	Guaranteed
Encryption	None	TLS 1.3
Handshake	0-RTT	1-RTT

Both architectures use UDP as base transport, validating our approach. We implement a lightweight sequence-based reliability layer with duplicate detection and out-of-order logging, rather than full QUIC, trading features for simplicity and lower overhead.

5.2 Performance Implications

5.2.1 Head-of-Line Blocking Elimination

TCP's sequential delivery creates head-of-line (HOL) blocking. Consider a scenario where a switch sends a large PACKET_IN (1,500 bytes) followed by a time-sensitive ECHO_REQUEST (8 bytes). With TCP, if the PACKET_IN segment is lost,

the ECHO_REQUEST blocks until retransmission completes, potentially causing keepalive timeout. With UDP, each message is independent—PACKET_IN loss doesn't affect ECHO_REQUEST delivery.

5.2.2 Resource Overhead

Table 4: Resource Overhead Comparison (Theoretical)

Resource	TCP	UDP
Per-connection memory	87 KB (Linux)	Minimal
Send/Recv buffers	16–87 KB each	Per-message
Retransmission queue	Yes	App-layer
Congestion window	Yes	No
ACK processing	Required	None

For controllers managing hundreds of switches, memory savings can be substantial.

5.3 Why Localhost Results Favor TCP

In localhost testing:

- Network latency is 0ms (kernel loopback)
- Packet loss is 0% (no network)
- TCP handshake overhead is negligible (0.01ms)
- Linux kernel TCP stack is highly optimized

In real networks, UDP's benefits would be more pronounced:

- No 1.5 RTT connection setup
- No head-of-line blocking
- No congestion control delays

5.4 Reliability Layer Trade-offs

Our implementation provides sequence-based reliability without TCP's complexity:

Table 5: Reliability Comparison

Feature	TCP	Our UDP
Retransmission	Automatic	Timeout-based (1s, max 3)
Duplicate Detection	Sequence-based	Seq sliding window (1000)
Ordering Detection	Guaranteed	Logged (not enforced)
Sequence Numbers	Yes	Yes (`_send_seq`)
Congestion Control	Yes	No
Statistics	Kernel-level	App-level tracking

5.5 Limitations

5.5.1 Current Implementation Limitations

- **No congestion control**: Could flood network under high load
- **Fixed retransmission timeout**: 1.0s is not adaptive to network conditions
- **Ordering detection only**: Out-of-order messages are logged but not reordered; FLOW_MODs may arrive inconsistently
- **Localhost testing**: Doesn't demonstrate loss recovery benefits

5.5.2 Network Environment Considerations

The practicality of UDP transport depends on network conditions:

- **Data center** ($<$0.01% loss, $<$1ms latency): UDP works effectively
- **Campus network** (0.1–1% loss): Limited reliability mechanisms advisable
- **WAN** ($>$1% loss, variable latency): Full reliability layer required

5.6 Future Work

- Test under controlled packet loss (`tc netem`)
- Implement adaptive timeout based on RTT estimation
- Add lightweight congestion control
- Benchmark with multiple switches and higher message rates
- Evaluate message reordering impact on flow table consistency

6 Conclusion

This project successfully demonstrates the feasibility of UDP-based southbound communication in SDN environments. We implemented a complete OpenFlow 1.3 UDP transport for both the Ryu controller and Open vSwitch:

- Modified Ryu controller with `DatapathUDP` class providing UDP socket handling and sequence-based reliability (retransmission after 1.0s, max 3 attempts, duplicate detection via 1000-entry sliding window, out-of-order detection with statistics tracking)
- Compiled OVS 3.1.0 with `stream-udp.c` and `vconn-udp.c` enabling `udp:` controller specification
- Validated complete OpenFlow handshake (HELLO, FEATURES, SET_CONFIG, ECHO) over UDP
- Demonstrated end-to-end functionality via Mininet with successful host connectivity
- Benchmarked TCP vs UDP latency showing comparable performance in localhost conditions

The architecture proves that OpenFlow can operate over UDP with application-layer reliability, eliminating TCP connection overhead and head-of-line blocking. While localhost testing showed similar latency (TCP slightly faster due to optimized kernel stack), the architectural advantages of UDP—zero-RTT transmission, message independence, and reduced per-connection state—would be more pronounced in real network environments with non-zero latency and packet loss.

6.1 Repository

- GitHub

References

[1] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling Innovation in Campus Networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008. Available: https://dl.acm.org/doi/abs/10.1145/1355734.1355746

[2] D. Kreutz, F. M. Ramos, P. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-Defined Networking: A Comprehensive Survey," *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, Jan. 2015. doi:10.1109/JPROC.2014.2371999. Available: https://ieeexplore.ieee.org/abstract/document/6994333

[3] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, and A. Amidon, "The Design and Implementation of Open vSwitch," in *Proc. USENIX NSDI*, 2015. Available: https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/pfaff

[4] B. Lantz, B. Heller, and N. McKeown, "A Network in a Laptop: Rapid Prototyping for Software-Defined Networks," in *Proc. 9th ACM SIGCOMM Workshop on Hot Topics in Networks (HotNets-IX)*, 2010. Available: http://mininet.org

[5] IBM, "What is software-defined networking (SDN)?" [Online]. Available: https://www.ibm.com/think/topics/sdn

[6] Open Networking Foundation, "OpenFlow Switch Specification Version 1.3.0," 2012. [Online]. Available: https://opennetworking.org/software-defined-standards/specifications/

[7] J. Postel, "User Datagram Protocol," RFC 768, Aug. 1980. Available: https://tools.ietf.org/html/rfc768

[8] P. Kumar and B. Dezfouli, "QuicSDN: Transitioning from TCP to QUIC for Southbound Communication in SDNs," *Journal of Network and Computer Applications*, vol. 222, 2024, Art. no. 103780. Preprint: https://arxiv.org/abs/2107.08336

[9] M. R. Rahman, A. Zilberman, and E. P. G. Jasinska, "SDUDP: A Reliable UDP-Based Transmission Protocol Over SDN," in *IEEE ICC Workshops*, 2017, pp. 2552–2557. doi: 10.1109/ICCWorkshops.2017.7898398