SDN Statistics Collection and Monitoring Using OpenFlow

January 10, 2025

University of Antwerp - Department of Computer Science
This report is part of the final project for the Future Internet class at the University of Antwerp by prof. Juan Felipe Botero Vega. It was written by Sam Roggeman and Maksim Karnaukh.

1 Introduction

Software-Defined Networking (SDN) revolutionizes network management. It accomplishes this by decoupling the control and data planes, enabling centralized control and dynamic adjustment of network traffic. Therefore, monitoring and analyzing network statistics is important for performance optimization, troubleshooting, and traffic engineering in SDN environments.

As such, this project aims to implement an SDN environment capable of collecting detailed flow, port, and other statistics using OpenFlow. The report details the steps taken to design and implement the statistics collection system. The full code for this project can be found here.

2 Implementation

2.1 Setting Up the Environment

To begin, ensure that POX is correctly installed and that you have an environment where OpenFlow switches are operational. In this case, we will work in a mininet VM.

2.2 Writing the Statistics Collection Module

The core functionality of this project lies in the custom POX component, `StatsCollector`. Below, we break down its implementation into manageable sections. All separately shown methods in the code blocks below belong to the `StatsCollector` class, unless explicitly mentioned otherwise.

2.2.1 Initialization and Timer Setup

The StatsCollector class is responsible for managing statistics requests and processing responses from switches. We set up a timer to periodically request statistics.

```
from pox.core import core
import pox.openflow.libopenflow_01 as of

from pox.lib.revent import *
from pox.lib.util import dpid_to_str
from pox.lib.util import dpidToStr
from pox.lib.recoco import Timer
from pox.openflow.of_json import *
import os
from datetime import datetime

log = core.getLogger()

class StatsCollector(EventMixin):

    def __init__(self, timer_interval=5):
        self.listenTo(core.openflow)
        self.stats = {} # store statistics per switch
        self.paths = {} # store paths per flow

        # remove txt statistics files
        ...

        self.interval = timer_interval # timer interval in seconds
        Timer(self.interval, self._timer_func, recurring=True) # library timer function
        log.info("StatsCollector initialized with timer interval %s seconds", self.
    interval)

def launch():
    core.registerNew(StatsCollector)
```

2.2.2 Requesting Statistics from Switches

The `_timer_func` method, which we use in the `Timer()` function, periodically sends requests for flow, port and possibly other statistics to all connected switches. The exact way to send a request is shown below. Namely, if we have a switch connection, we can send an `of.ofp_stats_request()` which has several arguments, of which we only care about the body argument. In the body, we can define the type of statistics we want (e.g., `ofp_port_stats_request`). For the documentation on this, one could look at the pox docs and the OpenFlow 1.0.0 spec (section of 5.3.5, pages 31-35).

```
def _timer_func(self):
    for connection in core.openflow._connections.values(): # iter over connected switches
        self.request_stats(connection)

def request_stats(self, connection):
    connection.send(of.ofp_stats_request(body=of.ofp_flow_stats_request()))
    connection.send(of.ofp_stats_request(body=of.ofp_port_stats_request()))

    # other type of statistics requests:
    connection.send(of.ofp_stats_request(
        body=of.ofp_aggregate_stats_request(
            match=of.ofp_match(), # match criteria (empty = match all flows)
            table_id=0xff, # table ID (0xff = all tables)
            out_port=of.OFPP_NONE # output port (OFPP_NONE = no specific port)
        )
    ))
    connection.send(of.ofp_stats_request(body=of.ofp_table_stats_request()))
    connection.send(of.ofp_stats_request(body=of.ofp_queue_stats_request()))
```

2.2.3 Handling Responses from Switches

The (statistics) event handlers process responses and store the statistics for further analysis. Below is an example of handling received port statistics:

```
def _handle_PortStatsReceived(self, event):
    switch_identifier = dpid_to_str(event.connection.dpid)
    log.info("PortStats received from %s", switch_identifier)

    stats_data = flow_stats_to_list(event.stats)
    if event.connection.dpid not in self.stats:
        self.stats[event.connection.dpid] = {}
    self.stats[event.connection.dpid]['port_stats'] = stats_data

    filename = "port_stats_" + switch_identifier + ".txt"
    self.write_stats_to_output(stats_data, filename, switch_identifier, stats_type='Port'
    )
```

As seen above, the statistics can be accessed through `event.stats`. The function `flow_stats_to_list()` (imported from `pox.openflow.of_json`) can be used to get the statistics in a list structure. What is inside of the list depends on the type of statistics:

(Individual) Flow Statistics (list with a dictionary per active flow):

```
[
    {
        match (Description of matching fields.),
        duration_sec (Time flow has been alive in seconds.),
        duration_nsec (Time flow has been alive in nanoseconds beyond duration_sec.),
        idle_timeout (Number of seconds idle before expiration.),
        hard_timeout (Number of seconds before expiration.),
        packet_count (Number of packets in flow.),
        byte_count (Number of bytes in flow.),
        ...
    },
    ...
]
```

Port Statistics (list with a dictionary per switch port):

```
[
    {
        port_no (port number)
        rx_packets (Number of received packets.),
        tx_packets (Number of transmitted packets.),
        rx_bytes (Number of received bytes.),
        tx_bytes (Number of transmitted bytes.),
        rx_dropped (Number of packets dropped by RX.),
        tx_dropped (Number of packets dropped by TX.),
        rx_errors (Number of receive errors.),
        tx_errors (Number of transmit errors.),
        rx_frame_err (Number of frame alignment errors.),
        rx_over_err (Number of packets with RX overrun.),
        rx_crc_err (Number of CRC errors.),
        collisions (Number of collisions.)
    },
    ...
]
```

The above detailed statistics are the most important to us: information about individual flows and information about physical ports. We only mentioned the important fields that are of interest to us.

One small thing that we can do is to sum up the port statistic values over all ports to get the total port statistics, which will also be present in the port statistics output.

2.2.4 Writing Statistics to Files

To make the collected data persistent, we write it to text files in a structured format. Below is an example of how the output for the different statistics looks like. Since the output is rather large and difficult to organize in the document, we will put it on a separate page each time.

The first example shows flow-level statistics for s2 at two timeframes whilst the second one shows port-level statistics for the same switch at the same timeframes. These examples are part of **flow2** and **port2**.

Flow statistics
This statistic includes information about the active network flows within a particular switch. We can see the number of packets and bytes and their averages transferred for a particular flow both as a whole and since the last timeframe (if such a timeframe exists). We can also see the duration that the flow has been active for. At the first timestamp, there are 3 active flows, all of which are new. This is logical as that is the first time any flows are openned. At the second timestamp, there are 5 active flows, with 2 new flows added and none removed from the previous set. We also see the used protocols which define a flow, for example ARP and IP are present here.

Port statistics
This output showcases port-level monitoring in an SDN environment. Over the five-second interval: Ports 1 and 2 experienced significant traffic growth. Port 3 showed limited activity, likely indicating its role in specific, low-traffic tasks. The statistics highlight critical performance metrics (e.g., no collisions or frame errors), ensuring smooth operations and identifying potential bottlenecks or dropped packets.

```
================================================================================================================================================
====================================Flow-Level Statistics for Switch 00-00-00-00-00-02 at 2025-01-08 19:31:20=====================
================================================================================================================================================
3 active flows with 3 new flows and 0 out of the previous 0 removed

Statistics for each flow (sorted by byte rate):
        Flow matching (protocol: IP) source: 10.0.0.2, 00:00:00:00:00:02 and destination: 10.0.0.1, 00:00:00:00:00:01
                Statistics since start:
                        Number of packets: 1, averaging 1.085 per second
                        Number of bytes: 42, averaging 45.553 per second
                        Duration: 0.922 seconds

        Flow matching (protocol: IP) source: 10.0.0.1, 00:00:00:00:00:01, port: 8 and destination: 10.0.0.2, 00:00:00:00:00:02
                Statistics since start:
                        Number of packets: 1, averaging 1.078 per second
                        Number of bytes: 42, averaging 45.259 per second
                        Duration: 0.928 seconds

        Flow matching (protocol: ARP) source: 10.0.0.2, 00:00:00:00:00:02 and destination: 10.0.0.1, 00:00:00:00:00:01
                Statistics since start:
                        Number of packets: 1, averaging 1.073 per second
                        Number of bytes: 42, averaging 45.064 per second
                        Duration: 0.932 seconds

================================================================================================================================================
====================================Flow-Level Statistics for Switch 00-00-00-00-00-02 at 2025-01-08 19:31:25=====================
================================================================================================================================================
5 active flows with 2 new flows and 0 out of the previous 3 removed

Statistics for each flow (sorted by byte rate):
        Flow matching (protocol: ARP) source: 10.0.0.1, 00:00:00:00:00:01 and destination: 10.0.0.2, 00:00:00:00:00:02
                Statistics since start:
                        Number of packets: 1, averaging 1.096 per second
                        Number of bytes: 42, averaging 46.053 per second
                        Duration: 0.912 seconds

        Flow matching (protocol: ARP) source: 10.0.0.2, 00:00:00:00:00:02 and destination: 10.0.0.1, 00:00:00:00:00:01
                Statistics since start:
                        Number of packets: 1, averaging 1.093 per second
                        Number of bytes: 42, averaging 45.902 per second
                        Duration: 0.915 seconds

        Flow matching (protocol: IP) source: 10.0.0.1, 00:00:00:00:00:01, port: 8 and destination: 10.0.0.2, 00:00:00:00:00:02
                Statistics since start:
                        Number of packets: 6, averaging 1.012 per second
                        Number of bytes: 252, averaging 42.489 per second
                        Duration: 5.931 seconds
                Statistics since last request:
                        Number of packets: 5, averaging 0.999 per second
                        Number of bytes: 210, averaging 41.975 per second
                        Duration: 5.003 seconds

        Flow matching (protocol: IP) source: 10.0.0.2, 00:00:00:00:00:02 and destination: 10.0.0.1, 00:00:00:00:00:01
                Statistics since start:
                        Number of packets: 6, averaging 1.013 per second
                        Number of bytes: 252, averaging 42.532 per second
                        Duration: 5.925 seconds
                Statistics since last request:
                        Number of packets: 5, averaging 0.999 per second
                        Number of bytes: 210, averaging 41.975 per second
                        Duration: 5.003 seconds

        Flow matching (protocol: ARP) source: 10.0.0.2, 00:00:00:00:00:02 and destination: 10.0.0.1, 00:00:00:00:00:01
                Statistics since start:
                        Number of packets: 1, averaging 0.168 per second
                        Number of bytes: 42, averaging 7.077 per second
                        Duration: 5.935 seconds
                Statistics since last request:
                        Number of packets: 0, averaging 0.0 per second
                        Number of bytes: 0, averaging 0.0 per second
                        Duration: 5.003 seconds
```

Port-Level Statistics for Switch 00-00-00-00-00-02 at 2025-01-08 19:31:20

rx_over_err	rx_packets	tx_errors	rx_dropped	tx_bytes	collisions	rx_frame_err	tx_packets	port_no	rx_bytes	tx_dropped	rx_crc_err	rx_errors
0	0	0	1	0	0	0	0	65534	0	0	0	0
0	2	0	0	84	0	0	2	1	84	0	0	0
0	2	0	0	84	0	0	2	2	84	0	0	0
0	0	0	0	42	0	0	1	3	0	0	0	0
0	4	0	1	210	0	0	5	Total	168	0	0	0

Port-Level Statistics for Switch 00-00-00-00-00-02 at 2025-01-08 19:31:25

rx_over_err	rx_packets	tx_errors	rx_dropped	tx_bytes	collisions	rx_frame_err	tx_packets	port_no	rx_bytes	tx_dropped	rx_crc_err	rx_errors
0	0	0	1	0	0	0	0	65534	0	0	0	0
0	8	0	0	336	0	0	8	1	336	0	0	0
0	8	0	0	336	0	0	8	2	336	0	0	0
0	0	0	0	42	0	0	1	3	0	0	0	0
0	16	0	1	714	0	0	17	Total	672	0	0	0

2.2.5 Updating Paths and Identifying Top Talkers

The system maintains a record of paths between source and destination addresses. It can also identify the top talkers for these paths based on bytes or packets.

An `update_paths` function processes flow statistics collected from switches to determine paths between source and destination nodes. For each flow, the function evaluates the `nw_src` (source network address) and `nw_dst` (destination network address) fields in the flow's match criteria, along with their associated traffic statistics, such as byte and packet counts. These paths and their statistics are stored in a dictionary structure, where each key is a tuple of (`source_address`, `destination_address`, `protocol`) and the value holds cumulative traffic statistics across flows.

This mechanism allows the system to track end-to-end traffic across the network, even if it traverses multiple switches. The trick is here to designate one switch as the collecting switch for a path, and ignore all other switches for that path. The code for this can be found in the python file.

The output of these written statistics look like the following:

```
Top 20 Talkers (sorted by bytes):
Source: 10.0.0.1, Destination: 10.0.0.3, Protocol: ALL,
    Path: 00-00-00-00-00-02 -> 00-00-00-00-00-01 -> 00-00-00-00-00-03: Total Bytes: 962, Total Packets: 11
Source: 10.0.0.3, Destination: 10.0.0.1, Protocol: ALL,
    Path: 00-00-00-00-00-02 -> 00-00-00-00-00-01 -> 00-00-00-00-00-03: Total Bytes: 962, Total Packets: 11
Source: 10.0.0.2, Destination: 10.0.0.1, Protocol: ALL,
    Path: 00-00-00-00-00-02: Total Bytes: 672, Total Packets: 16
Source: 10.0.0.1, Destination: 10.0.0.2, Protocol: ALL,
    Path: 00-00-00-00-00-02: Total Bytes: 672, Total Packets: 16
```

2.3 Running the Controller and the Topology

In order to use the statistics file, we will place it in the `misc/` directory. We will also be running the `forwarding.l2_learning`, since our statistics component doesn't handle any forwarding.

To start the custom POX module, launch POX with the `StatsCollector` component:

```
sudo ./pox.py log.level --DEBUG forwarding.l2_learning misc.sdn_statistics
```

In a separate terminal, we can then run a simple 2-depth tree mininet topology:

```
sudo mn --mac --controller=remote,port=6633 --topo=tree,depth=2 --link tc
```

The topology is shown in the image below:

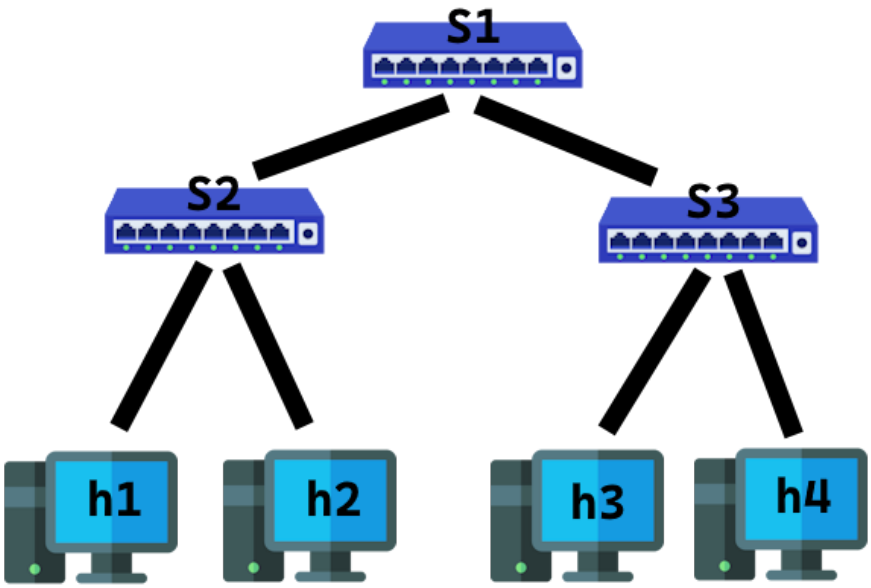

Figure 1: Layout of the network

If we then send some pings in the mininet console:

```
mininet> h1 ping h2 -c 15 -s 0
mininet> h1 ping h3 -c 10 -s 50
```

We will see that our output txt files contain the flow, port and top talker statistics.

3 Results

The results demonstrate the successful implementation of a statistics collection system in an SDN environment. Key findings include:

1. Real-time flow and port statistics were collected and logged.
2. Persistent records of statistics were stored in text files for further analysis.
3. Top talkers in the network were identified based on traffic volume.

This implementation showcases the capabilities of the POX SDN controller in monitoring network statistics with OpenFlow. The modular approach allows for easy extension and integration with other modules. The collected data can provide valuable insights for network optimization and troubleshooting.