CESE4045

High Performance Data Networking

2025

Assignment 1

Software-Defined Networking

Teaching Team

Instructors

Dr. Nitinder Mohan   Prof. Dr. Fernando Kuipers

Teaching Assistants

Chenxing Ji   Adrian Zapletal   Shatha Abbas   Lorenzo Theunissen

Contents

Introduction

In this assignment, you will gain hands-on experience with tools and techniques for experimenting with and managing computer networks. You will begin by using the network emulator **Mininet** to design and deploy custom network topologies. Once your network is running, you will use **Wireshark** to diagnose problems with your network.

Building on this foundation, you will then integrate an **SDN** (Software-Defined Networking) **controller** with your Mininet topology. You will first leverage the controller to install forwarding rules and to resolve the previously identified problem, and later to perform traffic engineering, collect statistics on active flows, and to explore fast-failover mechanisms that automatically reroute traffic when a link fails.

Prerequisites

Before starting this assignment, ensure you have completed the VM setup following the documentation provided along with this assignment.

- Ubuntu 24.04.3 VM installed and running
- Mininet installed successfully
- Ryu SDN controller installed with Python 3.9.19 virtual environment successfully

Assignment Deliverables

1. **Mininet script** for parking lot topology (Exercise 1.1)
2. **Mininet script** for extended parking lot topology (Exercise 1.2)
3. **Mininet script** for complex topology (Exercise 1.3)
4. A short **report** (**max. 1 page**) on ping results, the diagnosis of network problems in Exercise 1.2, and usage of the SDN controller in Exercise 1.4
5. **Ryu script** that performs traffic management (Exercise 1.5)
6. **Ryu script** that performs fast-failover (Exercise 1.6)
7. Please upload your scripts on BrightSpace with requested naming scheme and `"studentname_report.pdf"` in a .zip format with name `"studentname_studentid.zip"`

Mininet Basics

What is Mininet?

Mininet is a widely used network emulator that creates a realistic virtual network on a single machine. It allows you to define custom network topologies consisting of hosts, switches, links, and controllers.

You can start Mininet from the command line using the `mn` command:

```
sudo mn
```

By default, this launches a simple topology with two hosts connected by a single switch.

Predefined Topologies

You can start different topologies using the `-topo TOPO` option. Mininet provides several predefined topologies, such as:

- **single**: A star topology with a single switch and h hosts. This topology can be used by appending `-topo single,h` to the mn command, where h refers to the number of hosts.

- **linear**: h switches connect in a line, and there is one host connected to each switch. To use this topology, append `-topo linear,h` to the mn command.
- **tree**: A binary tree topology of depth d. To use this topology, append `-topo tree,d` to the mn command.

After starting Mininet, you will be placed in its command-line interface. From there, you can run commands on hosts (e.g., `h1 ping h2`) to test connectivity and run additional functions.

Cleaning Up

If you encounter problems or want to reset your environment, you can clean up all leftover Mininet processes and network state with:

```
sudo mn -c
```

Hint

For a step-by-step introduction to Mininet, follow the official walkthrough at http://mininet.org/walkthrough/. The Mininet website provides additional resources, including the API documentation at http://mininet.org/api/annotated.html. The mn command also has a man page.

Creating Custom Topologies

One of the main strengths of Mininet is the ability to define custom network topologies. You can write Python scripts that describe exactly how hosts, switches, and links are arranged. This gives you full flexibility to experiment with arbitrary network designs.

Basic Structure

Custom topologies are implemented by subclassing `Topo` from `mininet.topo`. Inside your class, you define how hosts and switches are added and how they are connected. A simple example is shown below:

```python
#!/usr/bin/env python3

from mininet.topo import Topo

class SimpleTopo(Topo):
    def __init__(self):
        # Initialize topology
        Topo.__init__(self)

        # Add hosts and a switch
        h1 = self.addHost('h1')
        h2 = self.addHost('h2')
        s1 = self.addSwitch('s1')

        # Connect hosts to the switch
        self.addLink(h1, s1)
        self.addLink(h2, s1)

# Expose topology to mn
topos = {'simpletopo': SimpleTopo}
```

If you store this code in `custom.py`, you can start Mininet with your defined topology with:

```
sudo mn --custom custom.py --topo simpletopo
```

Run this script and try `h1 ping h2` to test connectivity.

Adding Parameters

You can make your topology configurable by adding parameters to your script. For instance, you might want the number of switches to be flexible and configurable via a command line parameter. You can do so by adding a parameter to the `__init__()` function:

```
def __init__(self, n=4):
    ...
```

This adds `n` (i.e. number of switches) as parameter, which defaults to 4 if not specified. To specify the parameter, call the script using, e.g., `-topo simpletopo,3`.

Configuring Link Properties

In Mininet, you can configure link properties such as bandwidth, delay, loss rate, and queue size. When adding a link, configure it as `TCLink` via `self.addLink(..., cls=TCLink)`. This enables `tc` (traffic control) on the link, which can be used to configure the aforementioned properties. For example, to configure a bandwidth of 25 Mbps, delay of 15 ms, 1% packet loss, and a buffer size of 80 packets:

```
from mininet.link import TCLink
...
self.addLink(s1, s2, bw=25, delay='15ms', loss=1,
             max_queue_size=80, cls=TCLink)
```

Note that the default unit for `bw` is Mbps, and the default unit for delay is μs.

Testing Your Network

You can test the delay using ping, e.g., `h1 ping h2`, and you can test the bandwidth using iperf, e.g.:

```
h2 iperf -s &
h1 iperf -c h2
```

Exercise 1.1: Parking Lot Topology [5 points]

In this exercise, you are required to write a Mininet script that creates a parking lot topology, where switches are lined up and each switch has two hosts connected to it. Figure 1 depicts an exemplary parking lot topology with 4 switches. In your submission, the number of switches should be configurable.

Tasks

1. The topology must support a configurable number of switches, n (use parameter `n` in the topology constructor), with $n \geq 1$.
2. The links between switches should have a bandwidth of 10 Mbps and a delay of 5 ms.
3. The links connecting hosts to switches must be left unchanged (i.e., use default link properties).

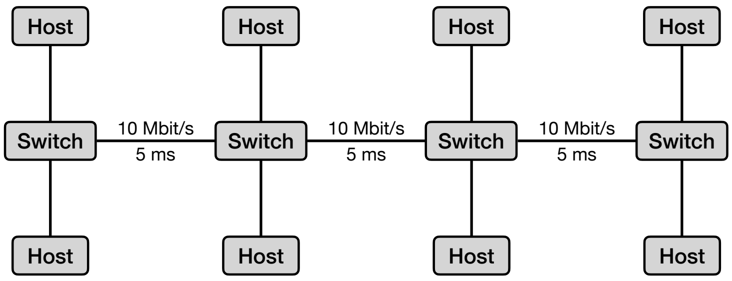

Figure 1: An exemplary parking lot topology where $n = 4$.

4. Each switch must have exactly two hosts attached to it.
5. The topology should allow pinging between any pair of hosts.
6. After implementing your topology, send a ping from a host connected to the first switch to a host connected to the last switch. Use a large n (e.g., 10) and observe the results.

Submission

1. A Mininet script named `parkinglot.py` that defines the class `ParkingLotTopo`. The topology key inside the `topos` dictionary must be named `parkinglottopo`.
2. A note in your report on the results of pinging across the topology. Mention your n and explain the ping results. A short paragraph suffices.

Exercise 1.2: Diagnosing Network Issues [4 points]

In this exercise, you will slightly extend your parking lot topology, creating a problematic topology. You will debug what the issue is using Wireshark, and then solve the problem in the topology without SDN (later, you will solve it with SDN).

Tasks

1. Make a copy of your parking lot topology script and change it to create one additional link between the first switch and the last switch. Try to ping between any pair of hosts. What do you see?
2. Use Wireshark to analyze the packets being sent in the network and find out why ping does not work:
   (a) Start your extended parking lot topology.
   (b) Choose any pair of hosts to test (e.g., h1 and h3).
   (c) On the sending host (e.g., h1), start Wireshark (or use `tcpdump` to write a packet trace to a file and then open it in Wireshark):

```
h1 wireshark &
```

Note: Mininet starts processes as superuser, which might cause a warning from Wireshark. You can safely ignore this warning.

(d) Ping the destination host (e.g., h3) from the sending host.

(e) In Wireshark, filter for ARP by using `arp` as filter. What can you see?

(f) Start Wireshark on the destination host (e.g., h3). Repeat the process and check whether the destination receives the ARP requests and sends out ARP replies.

(g) Finally, use Wireshark to investigate the switch that is attached to your sending host (e.g., s1).

Hint: Mininet switches do not run on their own separate network namespace. The Wireshark client you start on one switch can also access the interfaces of all other switches. When capturing multiple interfaces, you can add a column to indicate the interface by selecting any packet, right-clicking on "interface id" in the first collapsed section, and selecting "Apply as Column" (cf. `https://osqa-ask.wireshark.org/questions/30636/traces-from-multiple-interface/`).

3. After finding out what causes the problem, solve the problem *in your Mininet script* without SDN. The goal is that hosts in the topology are able to ping each other reliably, for arbitrary values of n, and you must not attach an SDN controller; the solution must be implemented only by modifying the Mininet script that creates the topology.

   Hint: Check the Mininet API documentation (`http://mininet.org/api/annotated.html`) for `OVSSwitch`. What subclasses does `OVSSwitch` have, and what options are available when creating switches of that subclass? Furthermore, note that the solution might take some time after starting the topology to take effect.

4. Finally, run a ping from a host connected to the first switch to a host connected to the last switch, like you did in Exercise 1.1.

Submission

1. A Mininet script named **`parkinglot_extended.py`** that defines the extended parking lot topology and solves the issue you encountered. The script should define the class **`ExtendedParkingLotTopo`**, and the topology key must be named **`extendedparkinglottopo`**.
2. A short section in your report that explains what the problem is and how you solved it. One or two paragraphs suffice.
3. A note in your report on the results of pinging across the topology. Mention your n and explain the ping results. A short paragraph suffices.

Exercise 1.3: Multi-Tier Topology [8 points]

In this exercise, you will expand your parking lot topology to include an aggregation layer for better connectivity.

Tasks

1. Use the parking lot topology from Exercise 1.1 *without the additional link you made in Exercise 1.2*, i.e., there are n switches in a line with two hosts connected to each switch. We call these n switches *edge switches*. The links between the edge switches

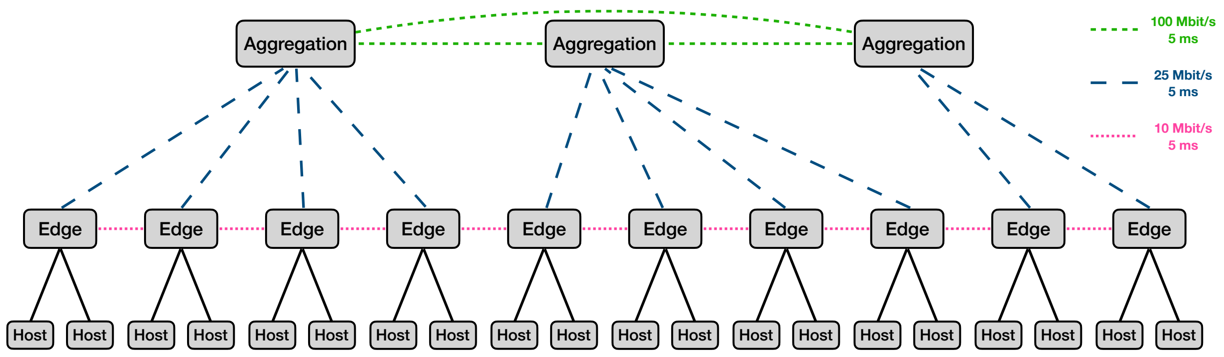

Figure 2: An example topology for Exercise 1.3 where $n = 10$ and $k = 4$.

should be configured to 10 Mbps bandwidth and 5 ms delay, like in Exercise 1.1.

2. Extend it as follows:
   (a) Add a layer of *aggregation switches*, where each aggregation switch is connected to k edge switches (k is a new parameter in this topology).
   (b) If n is not divisible by k, the last aggregation switch should only connect to $n \bmod k$ switches.
   (c) The links between edge and aggregation switches should be configured to 25 Mbps bandwidth and 5 ms delay.
   (d) All aggregation switches should be connected in a clique (fully connected mesh).
   (e) The links between aggregation switches should be configured to 100 Mbps bandwidth and 5 ms delay.
   (f) The links between edge switches and hosts do not require configuration.

   Figure 2 shows an example of the desired topology with $n = 10$ and $k = 4$.
3. Note that in this topology, you still have the problem that you discovered in Exercise 1.2. You can use the same fix here.
4. Finally, run a ping from a host connected to the first switch to a host connected to the last switch (e.g., for $n = 10$ and $k = 4$).

Submission

1. A Mininet script named `aggregationtopology.py` that defines the class `AggTopo`. The topology key inside the `topos` dictionary must be named `aggtopo`.
2. A note in your report on the results of pinging across the topology. Mention your n and k and explain the ping results. A short paragraph suffices.

Software-Defined Networking (SDN)

What is SDN?

Software-Defined Networking (SDN) separates the control plane (decision-making) from the data plane (packet forwarding). A centralized **controller** manages the network by installing forwarding rules on switches.

Ryu SDN Controller

So far, you have been experimenting with network topologies in Mininet. Next, you will use Software-Defined Networking (SDN) to attach a controller, which allows you to control the behavior of your network. You will run a Ryu controller application that installs forwarding rules on your switches.

Running Ryu with Mininet

You can start any Mininet topology and indicate that you have a controller application to attach:

```
sudo mn --custom custom.py --topo customtopo --controller=remote
```

In a separate terminal, start the Ryu controller with your controller application:

```
ryu-manager my_controller.py
```

For instance, you can use the built-in "simple switch" application `simple_switch_13.py`. This script comes with Ryu and implements basic MAC-learning switch behavior using OpenFlow 1.3.

Viewing Flow Rules

Back in the Mininet CLI, you can test connectivity between hosts using ping. You can observe the flow rules installed by the controller with:

```
ovs-ofctl dump-flows s1
ovs-ofctl dump-flows s2
```

Exercise 1.4: Solving Network Issues with SDN [1 points]

In Exercise 1.2, you discovered a problem with your network. Now you will solve the problem with SDN.

Take the topology you implemented in Exercise 1.2 *without* the solution to the ping problem. This time, use SDN to solve the ping problem. Explain what controller script you used and why the controller is able to solve the problem.

Tasks

1. Use your extended parking lot topology from Exercise 1.2 (with the loop) but without the fix you implemented.
2. Start it with a Ryu controller that can handle the network issue.
3. Verify that hosts can now ping each other successfully.

Submission

1. A short paragraph in your report about what controller you used and a brief explanation of why it solves the problem.

Exercise 1.5: Traffic Management [5 points]

In this exercise, you will work with a three-path topology available at https://github.com/spear-lab/HPDN-Assignments/blob/main/threepath.py, which you will use for the following two exercises. This topology connects a source host to a destination host via three parallel paths. These three paths have their bandwidth limited to 10 Mbps and delay increased to 200 ms. You will create a Ryu controller that forwards different traffic types across different paths. This will show how fine-grained OpenFlow rules can be used to program custom network behavior.

Tasks

1. Write a Ryu controller application that forwards TCP traffic from the source host to the destination host along the first path, forwards UDP traffic along the second path, and forwards ICMP traffic (e.g., ping) along the third path.
   The IP protocol number for TCP is 6, for UDP 17, and for ICMP 1. So, for instance, you can match TCP traffic using:

   ```
   match = parser.OFPMatch(eth_type=0x0800, ip_proto=6)
   ```

2. Test your implementation: TCP traffic with `iperf`, UDP traffic with `iperf -u`, and ICMP traffic with `ping`.

Submission

1. A Ryu app script that routes TCP, UDP, and ICMP traffic along different paths in the provided topology. The script should be named `trafficmanagement.py`.

Exercise 1.6: Bandwidth Measurement and Failure Handling [5 points]

In this exercise, you will extend your controller to use the OpenFlow Flow Statistics Request (OFPFlowStatsRequest) to periodically measure the bandwidth used by the TCP flow and to add resilience to the network by installing a fast-failover group.

Tasks

1. Implement bandwidth measurement for the TCP path:
   (a) Query one of the switches with OFPFlowStatsRequest periodically (e.g., every second). The switch will reply with an OFPFlowStatsReply, containing counters such as `packet_count`, `byte_count`, and `duration_sec`.
   (b) Compute the bandwidth in bits per second from this information.
   (c) Print the bandwidth using *precisely the following format*:
   `bandwidth = <BW> Mbps`
   where `<BW>` is replaced by the actual bandwidth value, e.g., `bandwidth = 9.4 Mbps`
   You can use the following line to produce this output format:

   ```
   print(f"bandwidth = {bw} Mbps")
   ```

   Note: Because you are running Mininet inside a VM, the bandwidth might be slightly inaccurate (depending on your computer and how you set up the VM). Ideally, you should see the bandwidth reach close to 10 Mbps, but do not worry if it reaches a bit less.

2. Add resilience to your network using fast-failover groups:
   (a) If the UDP path fails, UDP traffic should be rerouted automatically via the TCP path.
   (b) This is achieved with fast-failover groups in OpenFlow. Unlike normal flow rules, which require controller intervention when links fail, fast-failover groups are evaluated directly by the switch. If the primary port goes down, the switch instantly switches to the backup port without contacting the controller.
   (c) Write a controller application that installs a fast-failover group to reroute the UDP traffic via the TCP link if the UDP path fails.
   (d) Measure the bandwidth of the TCP flow when traffic gets rerouted.
3. Test your solution:
   (a) Run TCP via `iperf`, UDP via `iperf -u -b 5m` (to generate 5 Mbps UDP traffic), and ICMP via `ping`.
   (b) Break a link between switches `sX` and `sY` by running the following command from the Mininet CLI: `link sX sY down`
   (c) Verify that the failover works correctly and observe the bandwidth changes.

Hint

You can automate certain tasks by writing automation scripts, which are basically shell scripts containing the commands you want to automate, and run them using `source <myscript>`. For instance, you can write an automation script to start sending traffic, wait a few seconds, and then break the link.

Submission

1. A Ryu app script that performs the bandwidth measurement for the TCP path with the given output format and implements fast-failover from the UDP path to the TCP path. The script should be named **`failover.py`**.

End of Assignment 1