Exercises High Performance Data Networking

Niels van Adrichem Chenxing Ji Fernando Kuipers[1] Jorik Oostenbrink
Belma Turkovic Adrian Zapletal

[1]Responsible lecturer.

Chapter 1

Lecture 1: Introduction to Mininet

Solution

The solution files are available here: https://surfdrive.surf.nl/files/index.php/s/v9x9Fn9bWPQ5naL

1.1 Important Notice

If you run into problems while going through any of the exercises, first try reading through the exercise again and fixing the problem by yourself. If this does not help, you can ask for assistance at the Q&A sessions or via the discussion forum.

At the end of the first and second chapters, you will find a summary of useful links and commands.

Another note: we have experienced that when copying commands or code directly from the reader (this document) into terminal, the terminal sometimes does not recognize certain special characters (e.g., `'` or `-`).

1.2 Environment Setup

As you probably do not have access to a network of OpenFlow or P4 switches, you will run your programs on emulated networks instead. For this purpose we will use Mininet[1]. Mininet emulates switches using Open vSwitch, a popular and open-source virtual switch that is used in both hardware switches sold by vendors and software switches that can be installed on generic computer hardware. In this section, we will download and set up a Virtual Machine containing a pre-installed version of Mininet and all other software required for the course exercises. We use different tools and VMs for x86- and ARM-based systems. Inside the VM, there should be no differences. If you have a Mac with M1 or M2 CPU, follow the instructions for ARM, otherwise x86. If you want to install everything yourself and not use the provided VM, you can download: https://surfdrive.surf.nl/files/index.php/s/CSDP224IId0eFpK

x86 Install the open-source VirtualBox hypervisor, found here: https://www.virtualbox.org/wiki/Downloads.

The x86 VM image can be found here:
https://drive.google.com/file/d/1oKpnLdGdJ2ETfo29UBguhDnaMaMef-5N/view?usp=sharing

The Mininet and VirtualBox websites and communities provide ample information on installation issues. *We will not provide installation support for these tools.*

After downloading the image, start VirtualBox and import the VM by executing the following steps:

1. Import the VM by opening `File -> Import Appliance...` from the menu and selecting the image

2. Start the VM. The default username / password combination is: *hpdn* / *mininet*

ARM (Mac with M* CPU) Download the open-source QEMU-based hypervisor UTM (https://github.com/utmapp/UTM) from this link: https://github.com/utmapp/UTM/releases/latest/download/UTM.dmg

The ARM VM image can be found here:
https://drive.google.com/file/d/1deICH4h2LsXIXbhyQ95BRzllEjdI1mog/view?usp=sharing

After downloading and unzipping the image, start UTM and import the VM by executing the following steps:

1. Import the VM by selecting `+` and then *Open ...*; in this menu, select the downloaded image

2. Start the VM. The default username / password combination is: *hpdn* / *mininet*

[1] http://mininet.org/

1.3 Useful information about Mininet

Open a terminal, and start Mininet by running:

```
sudo mn
```

By default, Mininet will start a virtual network with 2 hosts, h1 and h2, connected via switch s1. Run `?` to view all possible commands. Feel free to play around with them, e.g. run `pingall` to confirm connectivity between all hosts.

Each host has a separate network namespace (but hosts share access to all other system resources, such as the filesystem). Within Mininet, you can run most Linux commands directly on any of the virtual hosts by prepending the command by its hostname. For example, you can ping h2 from h1 by running `h1 ping h2` (Mininet automatically replaces the second "h2" with h2's IP address). You can cancel the ping by pressing Ctrl+C. Analogously, you can also start different programs, services, or scripts on any of the virtual hosts, simply by prepending the relevant commands by a hostname.

Tip: By appending & to a command the process will run in the background and the Mininet window isn't blocked.

You can emulate different network topologies by adding the `--topo` option to the start command for Mininet. Mininet itself comes with the following built-in topologies:

- **minimal**: The default topology of 1 switch with 2 hosts. No further parameters apply.
- **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.
- **reversed**: Equivalent to the single switch topology, except that hosts connect to the switch in reverse order (i.e. the highest host number gets the lowest switch port).
- **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.

By default, hosts are assigned randomly generated MAC addresses. By appending `--mac` to the mn command, you can make Mininet use more readable MAC and IP addresses. This can be very helpful when debugging your controller application.

You can exit Mininet by typing `exit` in the terminal.

If Mininet crashes or you terminated it without using the `exit` command (for example with Ctrl+C), you will need to clean the Mininet environment with the following command:

```
sudo mn –c
```

For more information on how to create a Mininet network, you can run the following command to view the manual of Mininet:

```
man mn
```

1.4 Creating custom topologies

One of the advantages of Mininet is that it enables you to create complex network topologies and run experiments on them without the need to physically create those networks. You can specify your own custom network topologies in Python using a combination of the *addHost*, *addSwitch*, and *addLink* methods of the class *Topo*. To emulate your topology within Mininet, simply use the `--custom` command to load your custom Python script and `--topo` to select the (custom) topology:

```
sudo mn ––custom /path_to_your_topology/topo_file.py ––topo topology_name
```

For example, the following Python script creates a topology of 2 connected switches, adding 1 host to each switch, while the last line maps the topology classes to topology names. These names can then be used with the `--topo` option. The script is provided in the VM in `/home/hpdn/HPDN_Exercises/week_1/custom_topo.py`.

```
from mininet.topo import Topo

class MyTopo( Topo ):
    "Simple topology example."

    def __init__( self ):
        "Create custom topo."
```

```
        # Initialize topology
        Topo.__init__( self )

        # Add hosts and switches
        leftHost = self.addHost( 'h1' )
        rightHost = self.addHost( 'h2' )
        leftSwitch = self.addSwitch( 's1' )
        rightSwitch = self.addSwitch( 's2' )

        # Add links
        self.addLink( leftHost, leftSwitch )
        self.addLink( leftSwitch, rightSwitch )
        self.addLink( rightSwitch, rightHost )


topos = { 'mytopo': MyTopo }
```

It is also possible to map multiple topology classes to names in the same Python file, e.g.

```
topos = { 'mytopo': MyTopo, 'othertopo' : OtherTopo }
```

To start a network with the custom topology in this script, use:

```
sudo mn --custom /home/hpdn/HPDN_Exercises/week_1/custom_topo.py --topo mytopo
```

1.4.1 Exercise 1. - Complete graph & Square Lattice

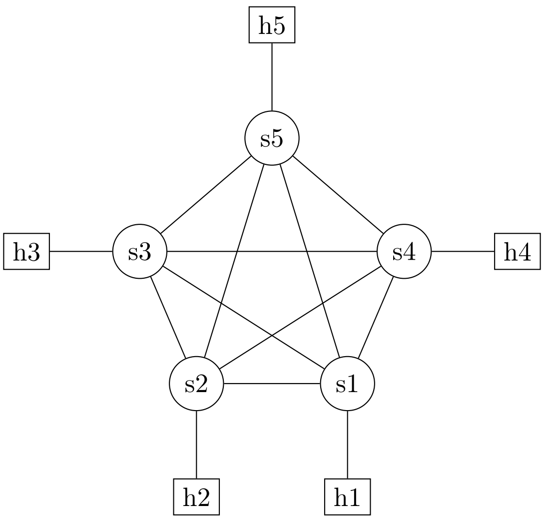


Figure 1.1: Complete graph ($n = 5$).

The following script creates a complete topology of n switches, each with a single host (as in Figure 1.1):

```
from mininet.topo import Topo

class Complete(Topo):
    #switches form a complete graph
    #One host connected to each switch

    #We set a default value for n,
    #so Mininet will not crash if someone creates this topology without specifying n
    def __init__(self, n = 5):
        Topo.__init__(self)

        switches = []
        for i in range(1,n+1):
```

```
            switch = self.addSwitch('s' + str(i))
            host = self.addHost('h' + str(i))
            self.addLink(switch, host)

            for s in switches:
                self.addLink(switch, s)

            switches.append(switch)

topos = {'complete': Complete}
```

The script is included in the VM: `/home/hpdn/HPDN_Exercises/week_1/custom_complete_topo.py`
For example, you can start a complete topology of 7 switches as follows:

```
sudo mn --custom /home/hpdn/HPDN_Exercises/week_1/custom_complete_topo.py --topo complete,7
```

Any parameters you provide for a custom topology (in this case 7) are passed on to `__init__`.

Note: Ping will not work on both the complete graph, as well as the square lattice topology you will construct yourself. (You will study this further in exercise 5 (Sec 1.8.2))!

EXERCISE

Create a custom script that can construct square lattice topologies of arbitrary size w x w. In a square lattice topology, all switches are aligned on a square grid and connected to their (up to 4) neighbors. Only add hosts to the four corner switches. See Figure 1.2 for an example of a square lattice topology of size 3x3.

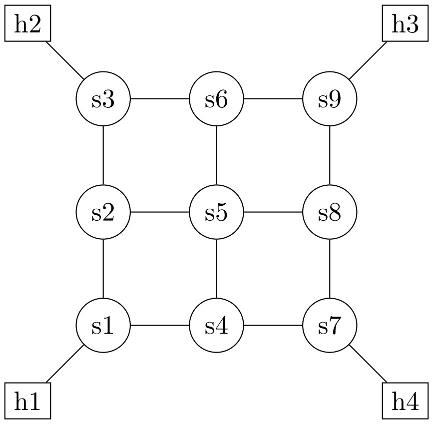


Figure 1.2: Square lattice graph ($w = 3$).

Solution

```
from mininet.topo import Topo

class SquareLattice(Topo):
    # Switches form a square lattice
    # One host at each corner

    # Note that we give a default value for w,
    # so mininet will not crash if someone creates this topology without specifying w
    def __init__(self, w = 3):
        Topo.__init__(self)

        switches = []
        for i in range(w):
            for j in range(w):
                switch_id = i * w + j + 1
                # create switch with id 's' + str(sid)
                switch = self.addSwitch('s' + str(switch_id))
```

```
                switches.append(switch)

                if j > 0:
                    # link current switch with switch below
                    self.addLink(switch, switches[-2])
                if i > 0:
                    # link current switch with switch to the left
                    self.addLink(switch, switches[-1 - w])

        # add the hosts
        host1 = self.addHost('h1')
        host2 = self.addHost('h2')
        host3 = self.addHost('h3')
        host4 = self.addHost('h4')

        # and link them to their switches
        self.addLink(switches[0], host1)
        self.addLink(switches[w-1], host2)
        self.addLink(switches[-1], host3)
        self.addLink(switches[-w], host4)

# create the mapping of custom topology names to topologies
topos = {'lattice': SquareLattice}
```

1.5 Introducing link properties

Mininet also has the capability to emulate network link parameters, such as bandwidth, delay, jitter, and loss. For example, if you want to set the bandwidth of all links in the network to 40 Mbps and their delay to 15 ms, you can run a command like the following. The default bandwidth unit is Mbps, and the default delay unit is μs:

```
sudo mn --link tc,bw=40,delay=15ms
```

It is possible to add these parameters in the custom Python files as well by specifying these options in the *addLink* function and using `TCLink` as shown below, and this allows us to set parameters of each link separately:

```
from mininet.link import TCLink
...
self.addLink(s1, s2, delay='5ms', bw=20, cls=TCLink)
```

or

```
from mininet.link import TCLink
...
self.addLink(s1, s2, delay=5000, bw=20, cls=TCLink)
```

Instead of importing `TCLink` and using `cls=TCLink`, you can also add the `--link tc` option when running mininet. `tc` stands for traffic control.

1.5.1 Exercise 2.1 - Add link properties

Create a simple loop-free topology using a topology script. Make sure there is a path from h1 to h2 in your topology. You can use a topology like the initial example in Section 1.4 where 2 hosts are connected via 2 switches (`custom_topo.py`). Modify your topology to set the delay of each link to a random value between 0 ms and 10 ms. In addition, set all link bandwidths to 10 Mbps.

Confirm your results using iperf and ping, for example:

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

Keep in mind that the resulting bandwidth and delay values might not be entirely precise because Mininet itself can be a little imprecise and we are running it inside a VM, causing more loss of precision.

Solution

```
from mininet.topo import Topo
from mininet.link import TCLink
from random import random

class SimpleTCTopo( Topo ):
    "Simple topology example."

    def __init__( self ):
        "Create custom topo."

        # Initialize topology
        Topo.__init__( self )

        # Add hosts and switches
        leftHost = self.addHost( 'h1' )
        rightHost = self.addHost( 'h2' )
        leftSwitch = self.addSwitch( 's1' )
        rightSwitch = self.addSwitch( 's2' )

        # Add links
        self._add_custom_link( leftHost, leftSwitch )
        self._add_custom_link( leftSwitch, rightSwitch )
        self._add_custom_link( rightSwitch, rightHost )

    def _add_custom_link(self, n1, n2):
        delay = random()*10
        self.addLink(n1, n2, delay=str(delay) + 'ms', bw=10, cls=TCLink)

topos = { 'tctopo': SimpleTCTopo }
```

Note that the delay is set to a new random value for each individual link.

When starting the topology, Mininet displays the bandwidth and delay of each link (in both directions) under "Adding links:". For example, `(10.00Mbit 9.77986975089ms delay) (10.00Mbit 9.77986975089ms delay) (h1, s1)`

If you start a ping between hosts `h1` and `h2`, the time should fluctuate around the sum of the link latencies between `h1` and `h2`. Similarly, the results of iperf should give a bandwidth of around 10 Mbps. (There can be some fluctuation here because you are running the simulation inside a VM.)

1.5.2 Exercise 2.2 - Congestion Control

In this exercise, we will use Mininet's capability to set link properties to experiment with congestion control. For this, you first need to create a bottleneck, i.e., a link that has lower capacity than other links. This bottleneck is where congestion will happen. If you use `custom_topo.py`, you can create a bottleneck between the two switches. Set the bandwidth of the link between the two switches to 10 Mbps. Set the bandwidth of the links between the hosts and the switches to 100 Mbps. Fix the delay on every link to 5 ms. Additionally, set the buffer size at the bottleneck link to 25 packets using the parameter `max_queue_size=25` in the addLink() function.

Start the network and set the congestion control algorithm on one host (in this case h1) to Reno using

```
h1 ip route change 10.0.0.0/8 dev h1-eth0 congctl reno
```

Run iperf to send a data stream from this host to another host. The bandwidth you see should be close to 10 Mbps, which is the bottleneck link speed.

Now change the buffer size to 5 packets and restart the network. As before, set the congestion control algorithm to Reno and run iperf. What can you observe?

Finally, change the congestion control algorithm to BBR using

```
h1 ip route change 10.0.0.0/8 dev h1-eth0 congctl bbr
```

Run iperf again. What happens? Can you explain why?

Solution

```
from mininet.topo import Topo
from mininet.link import TCLink
from random import random
```

```
class CongestionTopo( Topo ):
    "Simple topology example."

    def __init__( self ):
        "Create custom topo."

        # Initialize topology
        Topo.__init__( self )

        # Add hosts and switches
        leftHost = self.addHost( 'h1' )
        rightHost = self.addHost( 'h2' )
        leftSwitch = self.addSwitch( 's1' )
        rightSwitch = self.addSwitch( 's2' )

        # Add links
        self._add_custom_link( leftHost, leftSwitch, 100 )
        self._add_custom_link( leftSwitch, rightSwitch, 10, 25 )
        self._add_custom_link( rightSwitch, rightHost, 100 )

    def _add_custom_link(self, n1, n2, bw, bufsize=None):
        delay = 5
        if bufsize:
            self.addLink(n1, n2, delay=str(delay) + 'ms',
                        bw=bw, max_queue_size=bufsize, cls=TCLink)
        else:
            self.addLink(n1, n2, delay=str(delay) + 'ms',
                        bw=bw, cls=TCLink)

topos = { 'cctopo': CongestionTopo }
```

When running iperf with Reno and a buffer size of 25 packets, iperf will report a bandwidth close to 10 Mbps (e.g., 9.5 Mbps). When you lower the buffer size to 5 packets, Reno can only achieve a lower bandwidth (e.g., 7.5 Mbps). With BBR, on the other hand, you should see a higher bandwidth even when the buffer is small (e.g., 9.3 Mbps).

This happens because upon packet loss, Reno backs off (Multiplicative Decrease) and then slowly raises its sending rate again (Additive Increase). If the buffer is too small, Reno backs off so often that its Additive Increase can never reach maximum throughput before there is another packet loss.

A well-known rule of thumb is that the ideal buffer size when using Reno is 1 Bandwidth-Delay Product (BDP). With 10 Mbps bandwidth and 30 ms delay (6 links with 5 ms delay each), one BDP is 37500 bytes, which is 25 packets (a typical IP packet is 1500 Bytes). When there is packet loss and Reno backs off, the buffer is still filled with 25 packets, which get forwarded while Reno sends at a lower rate. By the time the buffer is empty, Reno's Additive Increase has reached maximum throughput again. This effectively keeps up the throughput because packets are sent from the buffer even though Reno has reduced its sending rate. However, if the buffer is too small, it does not contain enough packets to keep up the effective throughput. BBR takes a fundamentally different approach to congestion control and therefore does not suffer from the same throughput reduction as Reno.

1.6 Automating Tasks

It is possible to automate certain tasks in Mininet, such as adding routes to the hosts, executing Python scripts, or tearing down links. To do so, simply create a file and put a single command on each line, just like in a shell script. These commands can then be executed in Mininet with `source file`, where `file` is the path to your automation script.

For example, the following script first adds multicast routes to hosts `h1`, `h2`, and `h3` and prints "routes setup". Next, it starts two multicast iperf servers on hosts `h1` and `h2` and connects them with a client on host `h3`. After `h3` has finished sending its multicast packets, the two servers are shut down.

```
h1 route add -net 224.0.0.0 netmask 240.0.0.0 dev h1-eth0
h2 route add -net 224.0.0.0 netmask 240.0.0.0 dev h2-eth0
h3 route add -net 224.0.0.0 netmask 240.0.0.0 dev h3-eth0
py "routes setup"
h1 iperf -s -B 224.0.0.14 -u -i 1 &
h2 iperf -s -B 224.0.0.14 -u -i 1 &
```

```
h3 iperf -c 224.0.0.14 -u -t 5
h1 kill %iperf
h1 kill %iperf
h2 kill %iperf
h2 kill %iperf
```

This script exists in `HPDN_Exercises/week_1/example_script`. You can create a single Mininet topology with 3 switches and run the script with:

```
sudo mn --topo single,3
```

```
mininet> source /home/hpdn/HPDN_Exercises/week_1/example_script
```

Important note: We put & after commands to start the process in the background. This way, the next command will execute immediately and would not have to wait until the earlier one is finished. In the case of iperf, we could alternatively use the `-D` option to run it as a daemon.

1.7 Custom Mininet commands

It is possible to create custom Mininet commands. For example, we can add the command "sleep" to Mininet by creating the following Python file and loading it with `--custom`:

```python
from mininet.cli import CLI

from time import sleep

def custom_sleep(self, time):
    "custom sleep function"
    sleep(int(time))

CLI.do_sleep = custom_sleep
```

By specifying "CLI.do_*foo* = *bar*", you create the custom command *foo* that executes the function *bar*.

If we load this custom file we can use `sleep 10` to let the Mininet command line interface sleep for 10 seconds.

You can have multiple custom files. For example, if we have a topology file "custom_topo.py" and a commands file "custom_command.py", we can load both of them with:

```
--custom custom_topo.py,custom_command.py
```

1.7.1 Exercise 3. - Creating a Mininet script

Start Mininet with the linear topology with 4 switches (using `--topo=linear,4`) and the "custom_command.py" file created earlier, which contains the custom sleep command. Then, create an automation script that does the following:

- start a ping between h1 and h4
- sleep for 2 seconds
- tear down the link between switches s2 and s3 (`link s2 s3 down`)
- sleep for 60 seconds
- bring the link back up
- sleep for 2 seconds
- kill the ping process (`h1 kill %ping`)

Execute the script. What happens? What changes if you change the middle sleep command to 6 seconds instead of 60? Can you still deduce that the link was down from the ping messages?

Note that you only see the full output of ping **after** the process is killed.

Solution

```
h1 ping h4 &
sleep 2
link s2 s3 down
sleep 60
link s2 s3 up
sleep 2
h1 kill %ping
```

The output should look something like this:

```
PING 10.0.0.4 (10.0.0.4) 56(84) bytes of data.
64 bytes from 10.0.0.4: icmp_seq=1 ttl=64 time=51.9 ms
64 bytes from 10.0.0.4: icmp_seq=2 ttl=64 time=1.01 ms
64 bytes from 10.0.0.4: icmp_seq=3 ttl=64 time=0.208 ms
From 10.0.0.4 icmp_seq=40 Destination Host Unreachable
From 10.0.0.4 icmp_seq=41 Destination Host Unreachable
From 10.0.0.4 icmp_seq=42 Destination Host Unreachable
From 10.0.0.4 icmp_seq=43 Destination Host Unreachable
From 10.0.0.4 icmp_seq=44 Destination Host Unreachable
From 10.0.0.4 icmp_seq=45 Destination Host Unreachable
From 10.0.0.4 icmp_seq=46 Destination Host Unreachable
From 10.0.0.4 icmp_seq=47 Destination Host Unreachable
From 10.0.0.4 icmp_seq=48 Destination Host Unreachable
From 10.0.0.4 icmp_seq=49 Destination Host Unreachable
From 10.0.0.4 icmp_seq=50 Destination Host Unreachable
From 10.0.0.4 icmp_seq=51 Destination Host Unreachable
From 10.0.0.4 icmp_seq=52 Destination Host Unreachable
From 10.0.0.4 icmp_seq=53 Destination Host Unreachable
From 10.0.0.4 icmp_seq=54 Destination Host Unreachable
From 10.0.0.4 icmp_seq=55 Destination Host Unreachable
From 10.0.0.4 icmp_seq=56 Destination Host Unreachable
From 10.0.0.4 icmp_seq=57 Destination Host Unreachable
From 10.0.0.4 icmp_seq=58 Destination Host Unreachable
From 10.0.0.4 icmp_seq=59 Destination Host Unreachable
From 10.0.0.4 icmp_seq=60 Destination Host Unreachable
64 bytes from 10.0.0.4: icmp_seq=61 ttl=64 time=2042 ms
64 bytes from 10.0.0.4: icmp_seq=62 ttl=64 time=1042 ms
64 bytes from 10.0.0.4: icmp_seq=63 ttl=64 time=43.3 ms
64 bytes from 10.0.0.4: icmp_seq=64 ttl=64 time=0.995 ms
```

We can clearly see the link is down, as we get multiple `Destination Host Unreachable messages`. Now, if we replace the 60 seconds sleep with 6 seconds, the output will be similar to:

```
PING 10.0.0.4 (10.0.0.4) 56(84) bytes of data.
64 bytes from 10.0.0.4: icmp_seq=1 ttl=64 time=15.3 ms
64 bytes from 10.0.0.4: icmp_seq=2 ttl=64 time=11.9 ms
64 bytes from 10.0.0.4: icmp_seq=3 ttl=64 time=0.812 ms
64 bytes from 10.0.0.4: icmp_seq=10 ttl=64 time=16.2 ms
64 bytes from 10.0.0.4: icmp_seq=11 ttl=64 time=0.092 ms
```

We do not get any explicit messages that the host is unreachable anymore. It takes some time for these Destination Host Unreachable messages to appear because they only happen after the entry in the ARP cache times out, which is after 15–45 seconds. Nevertheless, even without Destination Host Unreachable messages, we can recognize that 6 ping requests did not receive a reply (corresponding to the 6 seconds the link was down) by *comparing the icmp_ seq numbers*.

1.8 Mininet packet capture using Wireshark

Wireshark is a network protocol analyzer used to capture and store network packets. Monitoring packets can be very helpful in later exercises to help you debug your application. In this section, we will use Wireshark to monitor the traffic between switches.

1.8.1 Exercise 4. - Monitoring traffic

1. Start Mininet with the linear topology with 2 switches.

2. Start Wireshark as a background process on host 1 (`h1 wireshark &`). Mininet always starts processes as superusers. This causes the initial warning you will see when starting Wireshark, and you can safely ignore this warning.

3. On the left you can select any of h1's interfaces to monitor. We are only interested in $h1-eth0$, so select this one and press start. You should not see any packets yet, as there is currently no traffic in your emulated network.

4. Generate some traffic by starting a ping from host 1 to host 2. You should be able to spot 2 types of traffic: ARP packets (so h1 and h2 can learn each others' addresses) and ICMP packets.

5. Filter on the ARP protocol by writing `arp` in the filter at the top-left.

6. You can filter packets on a wide variety of properties. For example, we can filter out all packets with IP destination address 10.0.0.2 by typing in `ip.dst!=10.0.0.2`. You can test this along with other conditions like the following:

   ```
   ip.dst!=10.0.0.2 and icmp
   ```

The interfaces of switches can be monitored in the same way. For example, you can start Wireshark on switch s1 with `s1 wireshark &`. The only difference to hosts is that Mininet switches do not run on their own separate network namespace, so the Wireshark client we started on switch s1 can also access the interfaces of all other switches.

You can capture the traffic of multiple interfaces by either selecting the interfaces and pressing start, or by starting multiple Wireshark processes at the same host/switch and starting multiple captures.

Hint: When capturing multiple interfaces at the same time, you can add an additional column to indicate the interface the traffic was captured on by following these instructions: `https://osqa-ask.wireshark.org/questions/30636/traces-from-multiple-interface`

1.8.2 Exercise 5. - Diagnosing Network Issues

Start a Mininet instance with your square lattice topology (change the default width and set $w = 2$). Try to ping host h4 from host h2. Use Wireshark to find out what goes wrong and explain why this problem occurs.

Hint 1: Learning Switches. Switches start with an empty forwarding table and incrementally build it through a learning process based on the processed packets. This process works as follows: the switch maintains a table of MAC addresses that have appeared as source addresses in packets and the interface the switch has received these packets on. Thus, the switch learns and knows how to reach these addresses if one of them later shows up as a destination address. If a switch does not have an entry for a particular destination address, it defaults to flooding, i.e., it forwards the packet out on every port except the one on which it received the packet.

Hint 2: In previous exercises, we saw some network topologies where ping worked and others where it fails. What is a structural difference between the networks where ping succeeds and the ones where ping fails?

Hint 3: In Wireshark, order packets by timestamp. If your computer slows down too much during this exercise, restart the Mininet network.

Can you think of a method to fix the problem?

Solution

The ping does not seem to work, as it only outputs `Destination Host Unreachable` and gives a 100% packet loss rate. To figure out why this is happening, let's first capture the traffic on host `h2`, to see if it actually sends the ping request packets:

No.	Time	Source	Destination	Protocol	Lengtl	Info
1	0.000000000	00:00:00_00:00:02	Broadcast	ARP	42	Who has 10.0.0.4? Tell 10.0.0.2
2	0.006288000	00:00:00_00:00:02	Broadcast	ARP	42	Who has 10.0.0.4? Tell 10.0.0.2
3	0.016007000	00:00:00_00:00:02	Broadcast	ARP	42	Who has 10.0.0.4? Tell 10.0.0.2
4	0.029603000	00:00:00_00:00:02	Broadcast	ARP	42	Who has 10.0.0.4? Tell 10.0.0.2
5	0.038928000	00:00:00_00:00:02	Broadcast	ARP	42	Who has 10.0.0.4? Tell 10.0.0.2
6	0.045765000	00:00:00_00:00:02	Broadcast	ARP	42	Who has 10.0.0.4? Tell 10.0.0.2
7	0.049952000	00:00:00_00:00:02	Broadcast	ARP	42	Who has 10.0.0.4? Tell 10.0.0.2
8	0.053438000	00:00:00_00:00:02	Broadcast	ARP	42	Who has 10.0.0.4? Tell 10.0.0.2
9	0.056414000	00:00:00_00:00:02	Broadcast	ARP	42	Who has 10.0.0.4? Tell 10.0.0.2
10	0.061347000	00:00:00_00:00:02	Broadcast	ARP	42	Who has 10.0.0.4? Tell 10.0.0.2
11	0.063348000	00:00:00_00:00:02	Broadcast	ARP	42	Who has 10.0.0.4? Tell 10.0.0.2
12	0.070361000	00:00:00_00:00:02	Broadcast	ARP	42	Who has 10.0.0.4? Tell 10.0.0.2
13	0.071253000	00:00:00_00:00:02	Broadcast	ARP	42	Who has 10.0.0.4? Tell 10.0.0.2
14	0.073508000	00:00:00_00:00:02	Broadcast	ARP	42	Who has 10.0.0.4? Tell 10.0.0.2
15	0.074459000	00:00:00_00:00:02	Broadcast	ARP	42	Who has 10.0.0.4? Tell 10.0.0.2
16	0.077882000	00:00:00_00:00:02	Broadcast	ARP	42	Who has 10.0.0.4? Tell 10.0.0.2

```
Frame 1: 42 bytes on wire (336 bits), 42 bytes captured (336 bits) on interface 0
Ethernet II, Src: 00:00:00_00:00:02 (00:00:00:00:00:02), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
Address Resolution Protocol (request)
```

As we can see, `h2` is not sending out any ping requests to the network, as it does not have the Ethernet address of `h4` (10.0.0.4). To get this address `h2` continuously sends out ARP packets to the network, but it does not receive any replies. (As we will later learn, what we are seeing here are not only packets sent out by `h2`, but also some of its own packets that got sent back by the switch to `h2`. Unfortunately, we cannot separate these in Wireshark).

The next obvious step is to capture the traffic on `h4` as well, to see if it actually receives and replies to these ARP packets:

No.	Time	Source	Destination	Protocol	Lengtl	Info
1	0.000000000	00:00:00_00:00:02	Broadcast	ARP	42	Who has 10.0.0.4? Tell 10.0.0.2
2	0.000028000	00:00:00_00:00:04	00:00:00_00:00:02	ARP	42	10.0.0.4 is at 00:00:00:00:00:04
3	0.002693000	00:00:00_00:00:02	Broadcast	ARP	42	Who has 10.0.0.4? Tell 10.0.0.2
4	0.002716000	00:00:00_00:00:04	00:00:00_00:00:02	ARP	42	10.0.0.4 is at 00:00:00:00:00:04
5	0.016408000	00:00:00_00:00:02	Broadcast	ARP	42	Who has 10.0.0.4? Tell 10.0.0.2
6	0.016434000	00:00:00_00:00:04	00:00:00_00:00:02	ARP	42	10.0.0.4 is at 00:00:00:00:00:04
7	0.020371000	00:00:00_00:00:02	Broadcast	ARP	42	Who has 10.0.0.4? Tell 10.0.0.2
8	0.020402000	00:00:00_00:00:04	00:00:00_00:00:02	ARP	42	10.0.0.4 is at 00:00:00:00:00:04
9	0.034920000	00:00:00_00:00:02	Broadcast	ARP	42	Who has 10.0.0.4? Tell 10.0.0.2
10	0.034944000	00:00:00_00:00:04	00:00:00_00:00:02	ARP	42	10.0.0.4 is at 00:00:00:00:00:04
11	0.037903000	00:00:00_00:00:02	Broadcast	ARP	42	Who has 10.0.0.4? Tell 10.0.0.2
12	0.037929000	00:00:00_00:00:04	00:00:00_00:00:02	ARP	42	10.0.0.4 is at 00:00:00:00:00:04
13	0.046013000	00:00:00_00:00:02	Broadcast	ARP	42	Who has 10.0.0.4? Tell 10.0.0.2
14	0.046033000	00:00:00_00:00:04	00:00:00_00:00:02	ARP	42	10.0.0.4 is at 00:00:00:00:00:04
15	0.047535000	00:00:00_00:00:02	Broadcast	ARP	42	Who has 10.0.0.4? Tell 10.0.0.2
16	0.047555000	00:00:00_00:00:04	00:00:00_00:00:02	ARP	42	10.0.0.4 is at 00:00:00:00:00:04

```
Frame 1: 42 bytes on wire (336 bits), 42 bytes captured (336 bits) on interface 0
Ethernet II, Src: 00:00:00_00:00:02 (00:00:00:00:00:02), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
Address Resolution Protocol (request)
```

Interestingly, `h4` does receive and reply to the ARP packets, but for some reason, the network is not sending these replies to `h2`.

To discover why the replies are not being sent to `h2`, we will have to capture the traffic on a switch itself. We capture the traffic on all interfaces of switch `s2` (the switch connected to `h2`). As we are capturing the traffic on multiple interfaces, it will be useful to sort the traffic on timestamp, as by default the packets are sorted by interface first and time of arrival second (to increase clarity, we added an interface id column):

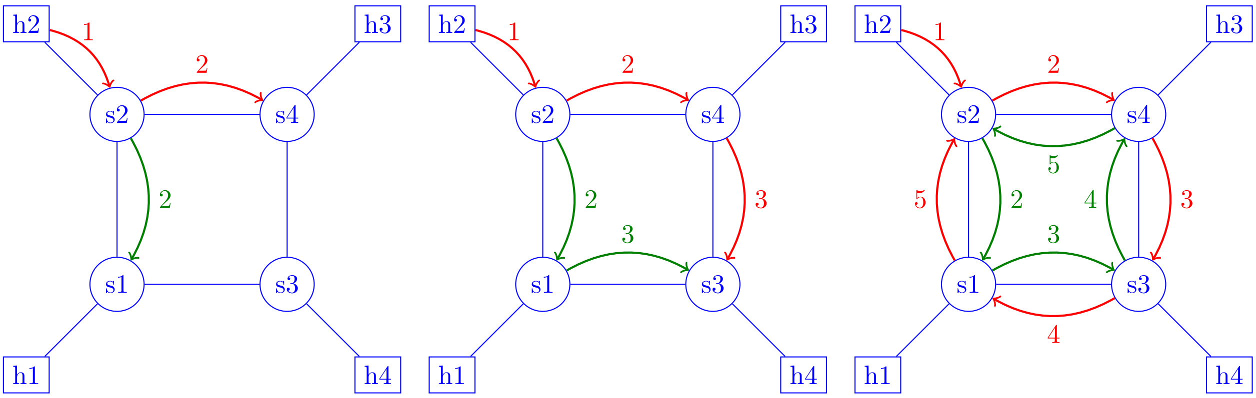
No.	Time	Source	Destination	Protocol	Length	Interface id	Info
1	0.000000000	00:00:00_00:00:02	Broadcast	ARP	42	0	Who has 10.0.0.4? Tell 10.0.0.2
22	0.000779000	00:00:00_00:00:02	Broadcast	ARP	42	1	Who has 10.0.0.4? Tell 10.0.0.2
44	0.000785000	00:00:00_00:00:02	Broadcast	ARP	42	2	Who has 10.0.0.4? Tell 10.0.0.2
23	0.003080000	00:00:00_00:00:02	Broadcast	ARP	42	1	Who has 10.0.0.4? Tell 10.0.0.2
45	0.004790000	00:00:00_00:00:02	Broadcast	ARP	42	2	Who has 10.0.0.4? Tell 10.0.0.2
2	0.004795000	00:00:00_00:00:02	Broadcast	ARP	42	0	Who has 10.0.0.4? Tell 10.0.0.2
46	0.005251000	00:00:00_00:00:02	Broadcast	ARP	42	2	Who has 10.0.0.4? Tell 10.0.0.2
24	0.006680000	00:00:00_00:00:02	Broadcast	ARP	42	1	Who has 10.0.0.4? Tell 10.0.0.2
3	0.006686000	00:00:00_00:00:02	Broadcast	ARP	42	0	Who has 10.0.0.4? Tell 10.0.0.2
25	0.011601000	00:00:00_00:00:02	Broadcast	ARP	42	1	Who has 10.0.0.4? Tell 10.0.0.2
47	0.012347000	00:00:00_00:00:02	Broadcast	ARP	42	2	Who has 10.0.0.4? Tell 10.0.0.2
48	0.012782000	00:00:00_00:00:02	Broadcast	ARP	42	2	Who has 10.0.0.4? Tell 10.0.0.2
4	0.012787000	00:00:00_00:00:02	Broadcast	ARP	42	0	Who has 10.0.0.4? Tell 10.0.0.2
26	0.014088000	00:00:00_00:00:02	Broadcast	ARP	42	1	Who has 10.0.0.4? Tell 10.0.0.2
5	0.014092000	00:00:00_00:00:02	Broadcast	ARP	42	0	Who has 10.0.0.4? Tell 10.0.0.2
27	0.014480000	00:00:00_00:00:02	Broadcast	ARP	42	1	Who has 10.0.0.4? Tell 10.0.0.2

```
Frame 1: 42 bytes on wire (336 bits), 42 bytes captured (336 bits) on interface 0
Ethernet II, Src: 00:00:00_00:00:02 (00:00:00:00:00:02), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
Address Resolution Protocol (request)
```

`s2` does not receive the ARP reply either (**NOTE:** This differs per machine. On some machines, `s2` will receive the reply and on others it will not. However, the overall problem remains the same.), but this capture does give us all the information needed to figure out the reason ping does not work on this network. As you can see, `s2` does not only receive the ARP request on interface `0` (its connection to `h2`), but on all its interfaces. That is, the packets are looping in the network. Loops such as in our lattice topology are problematic for communication networks.

By default, Mininet switches act as learning switches. That is, they output packets addressed to Ethernet address A to the last port they received a packet from A on (or just output the packet to all ports if they have not received a packet from A yet). Now, because multiple packets are looping around in our network, this output port will, with some exceptions, not be the correct output port. In fact, in our case, a switch is dropping all ARP replies, because it receives them on the same port it should send them to.

In our network, every ARP request sent to switch `s2` results in two packets looping through the network in opposite directions, as broadcast packets are output to all ports (except the one they are received on). This is illustrated below:



If we were to use a square lattice with a larger width, a single broadcast packet would be duplicated infinitely. This is called a broadcast storm.

There are multiple protocols to solve this problem. One prominent example is the Spanning Tree Protocol (STP), which makes switches flood/broadcast packets over a spanning tree instead of to all output ports, by disabling all ports that are not part of the tree.

In an OpenFlow network, loops in the network are much less of a problem. The controller has a central view of the network, so it should not have to flood packets to all output ports of switches. Nevertheless, when handling broadcast packets, care should be taken to not loop these packets through the network. To handle these kinds of packets, a similar approach to the STP can be taken by installing a spanning tree in the network and redirecting all broadcast packets to this spanning tree. Alternatively, the controller can send broadcast packets to all hosts itself. However, in many cases, this is not recommended, as the controller will potentially be flooded and overburdened with broadcast packets.

1.9 Useful commands

`dump`: Dump information about all nodes
`help`: Display all options
`device_name command`: If the first typed string is a host, switch or controller name, the command is executed on that node, e.g. to display the IP address of a host h1 type `h1 ifconfig`

`link s1 s2 down/up`: Bring links down and up
`h1 ping h2`: Test connectivity between two hosts
`pingall`: Test connectivity by having all nodes ping each other
`h1 arp`: Display the ARP table of host h1
`h1 route`: Display the routing table of host h1
`h1 python -m SimpleHTTPServer 80 &`: Run a simple web server on host h1
`h1 kill \%python`: Shut the started web server down
`h2 wget -O - h1`: Send a request from node h2 to the web server running on node h1

You have now finished the HPDN Mininet exercises. For more information on Mininet, you can look through the official walkthrough: http://mininet.org/walkthrough/