Intent Based Application

Group Project Report for Advanced Network Architectures And Wireless Systems

Francesco Iemma
Yuri Mazzuoli
Giovanni Menghini

M.Sc. in Computer Engineering

Academic Year 2021/22

Contents

Introduction

This project is aimed to design and develop an intent based application. The scenario is the following: *"Consider a set of clients that can communicate through a redundant network (e.g., based on a spine-leaf topology.) An external application can request to install paths between host pairs, by just specifying the endpoint identifiers, we refer to this as an host-2-host intent. The network has to allow communications only among the specified host pairs. Moreover, the network has to automatically reconfigure in case of link failures. Design and realize a system that allows users to request and withdraw host-to-host intents, and configures the underlying network accordingly."*
The objectives of the projects are:

1. *"Implement a Floodlight module that exposes a RESTful interface allowing clients to create/delete host-to-host instents. The module will then dynamically install and update flow rules in the network to allow the communication among specified pairs. Possible path switches must occur transparently to clients."*

2. *"Test the overall system using Mininet and Floodlight, devising proper scenarios to demonstrate the above functionalities."*

Chapter 1

Implementation and Design

The objective of the intent based application is mainly to expose a REST interface to allow the request of an **intent**. An **intent** is a request, done by an host, to have a link with another host, this link must tolerate link failures. Hence the controller must implement a module that is in charge of:

- Computing the best path between hostA and hostB
- Installing in the switches of the network the proper rules to establish this path
- Being responsive in case of a link failures and establishing a new path to maintain the link alive

In the implementation of the application we have re-used some Floodlight modules extending some classes where needed. In the following sections we will analyze the choices done and the classes extended in order to implement the features shown previously.

1.1 Floodlight Forwarding

The forwarding in Floodlight is implemented providing the abstract class ForwardingBase[1] which is in charge of providing a base class for implementing a forwarding module. The forwarding module "is responsible for programming flows to a switch in response to a policy decision". The implementation of the abstract class must implement the following abstract method:

```
    public abstract Command processPacketInMessage(IOFSwitch sw, OFPacketIn pi,
    IRoutingDecision decision, FloodlightContext cntx);
```

The Floodlight standard implementation of this abstract class is the class Forwarding[2].

1.2 Floodlight Extended Forwarding Module

The main module of our package is IntentForwarding. This module is an extension of the floodlight forwarding module[3] and we will call it also extended forwarding module. In short the added functionalities are:

- An array list that remembers the list of active intents (intents database).
- The extension of the method *ProcessPacketInMessage* in order to check if the packet that triggered the packetIn is from two host that are allowed to communicate (an intent between the two exists) or not.

[1] net.floodlightcontroller.routing.ForwardingBase
[2] net.floodlightcontroller.forwarding.Forwarding
[3] net.floodlightcontroller.forwarding.Forwarding

1.2.1 Intent Life Cycle

An intent is created, is active and then, when the time indicated in the intent establishment request finishes, expires.
When an intent is created the following operations are done:

- It is checked if an intent with the same IPs already exists. If yes no other operations are performed.
- A new timeout task is created and scheduled in order to be executed in the amount of time indicated by the intent timeout
- The intent is added to the array list

Notice that the creation of the timer is needed in order to implement the intent expiration and to perform some operations when this event takes place.
The operations that must be done when the intent expires are:

- Install rules in certain switches in order to deny the communications between the hosts of the expired intent
- Delete the intent from the intent database

Thus the main problem is to decide which are the switches where we have to install the rules.
Let assume that we have two host: *host1* and *host2* and an intent between them exists and it is going to expire. In order to prevent the two hosts to communicate after the expiry and in order to minimize the remaining packets in the network from hosts of an expired intent, the rules are installed in the first switch encountered by the *host1* when sends the message to *host2* (we will call this switch *switch1*) and in the first switch encountered when *host2* sends a message to *host1* (we will call this switch *switch2*). In practice we install the rules in the access switch that allow an host to communicate with the rest of the network.
The rules installed on each of the two switches are:

- Deny IPv4 from *host1* to *host2*
- Deny IPv4 from *host2* to *host1*

Both rules are valid for 5 seconds by default. See figure 1.1 for a graphical representation.

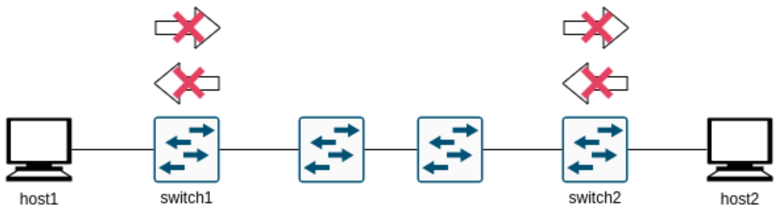


Figure 1.1: Graphical representation of where the rules for denying the communications between two hosts are installed

Thanks to this rules no other packets are allowed to pass from *switch1* and *switch2*. It is possible that some packets are in the network because they have passed the access switches before the intent expiry, in that case there are no problems because they are discarded in any case when they arrive in one of the two access switches. Those packets are a few (possibly 0) and so they doesn't affect the performance heavily, the alternative would be installing rules on each switch of the network in order to discard immediately these packets but this would be more costly.
When the rules are no longer valid for each packet arriving to *switch1* or *switch2* a packetIn is generated and thus the *processPacketInMessage* method will check if that packet is allowed (an intent has been re-established) or not (the old intent has expired and no other intent have been established).
Notice that due to the fact that the rules installed in the switches are valid for a certain time, no intent can be established until that rules are no longer valid. In the default case the rules are valid for 5 seconds and so a new intent regarding one of the two hosts can be established 5

seconds after the expiry of the old intent. That time can be changed by the network manager through REST.
The last problems that need a solution is how we can know which is the identifier of *switch1* and *switch2*. When the timeout task (the task that will be performed when the intent expires) is created we pass to the constructor:

- A reference to the object IntentForwarding, this is necessary in order to call the function who is in charge to install rules in the switches
- A reference to the object representing the intent associated to that timeout task. That object is **HostPair** and it is composed by:
  - IPv4Address host1IP
  - IOFSwitch sw1
  - IPv4Address host2IP
  - IOFSwitch sw2
  - long timeout

  That reference is necessary in order to retrieve the identifier of *switch1* and *switch2*. That identifiers are set when the first packetIn coming from the first switch encountered by the packet sent by an host is handled by the extended forwarding module i.e. IntentForwarding (for more on this see 1.2.2).

Here we can see the code executed when we add an intent.

```java
public boolean addNewIntent(HostPair newPair) {
  System.out.print("AddNewIntent Called\n");
  if(intentsDB.contains(newPair)) {
    log.info(" Intent already present in Intents List");
    return false;
  }

  long timeout = newPair.getTimeout();
  Timer timer = new Timer();
  TimerTask task = new TimeoutTask(newPair, this);
  timer.schedule(task, timeout);

  intentsDB.add(newPair);
  return true;
}
```

1.2.2 IPv4 Management

When a packetIn regarding an IPv4 packet arrives the operations done are:

- The intents database is searched:
  - If exists an intent between the sender and the receiver (or viceversa) then
    - If the sender is *host1* and *switch1* (in the object HostPair representing the intent) is null then *switch1* is set equal to the identifier of the switch that has sent the packetIn that has triggered this execution of *processPacketInMessage*. Same thing is done for *host2* and *switch2*.
    - Then the method of the super class (Forwarding.java) is invoked
  - Otherwise two rules are installed on the switch that sent the packetIn in order to deny the communication between the two hosts for a certain amount of time (as we have seen the default timeout is 5 seconds but that value can be customized through REST API).

  Here a section of the method *processPacketInMessage*. Notice that when setting the switch id, is the method *setSw(IOFSwitch switch)* that checks if the switch is already set or not and set the switch only if it is not already set.

```
    HostPair hp = null;
    int hpIndex = intentsDB.indexOf(new HostPair(sourceIP, destinIP));
    if (hpIndex != -1)
    hp = intentsDB.get(hpIndex);

    if(hp != null && intentsDB.contains(hp)) {
      System.out.printf("allowing: %s - %s on switch %s \n",
      sourceIP.toString(), destinIP.toString(), sw.getId());
      if(hp.getHost1IP().equals(sourceIP))
      hp.setSw1(sw);
      if(hp.getHost2IP().equals(sourceIP))
      hp.setSw2(sw);
      return super.processPacketInMessage(sw, pi, decision, cntx);
    }
    denyRoute(sw, sourceIP, destinIP);
    denyRoute(sw, destinIP,sourceIP);
    return Command.CONTINUE;
```

1.2.3 ARP Management

The ARP protocol is used to retrieve the layer 2 address of an host given its IP address. When an ARP packet arrives to a switch a packetIn is sent to the controller and so the processPacketInMessagge is invoked, the method understands that the packet that has triggered the packetIn is an ARP packet and so it must be handled differently.
The first operation done is checking if an intent between the sender and the receiver exists using their IPs. If an intent exists then the method of the super class is invoked otherwise the ARP packet is blocked.
The rules to avoid communication between the MACs addresses carried inside the ARP packet are installed only if that packet is an ARP response, in that case the target MAC address is not 00:00:00:00:00:00. In fact installing a rule that deny to forward an ARP messagge from a certain mac X to 00:00:00:00:00:00 means deny X to runs the ARP protocol because for each ARP request the target mac is 00:00:00:00:00:00 i.e. unspecified mac. Viceversa deny an ARP from 00:00:00:00:00:00 to any mac has no sense.

1.3 REST API

To establishing new intent between hosts and in general to manage them we have defined some APIs. All paths of these APIs are defined in the IntentWebRoutable class and all of them use method from IntentForwarding.

1.3.1 Add new intent

Thanks to "/addNewIntent/json" API we can insert a new intent to allow communications between two hosts. This API use the AddNewIntent class that gets from a JSON file two IPs (representing source and destination) and a timeout and store it in a new HostPair object. Then use the addNewIntent function of the IntentForwarding class that, first check if new intent is already presents, then in case it isn't put it in intentDb, that is an ArrayList of HostPair. Moreover, this function also set the timeout time for the pair expiration.

1.3.2 Delete an intent

Usingn"/delIntent/json" API we can delete an existing intent. This API use the DelIntent class that takes from a JSON file two IPs and create an HostPair object. Since we need only the Ips representing hosts, HostPair is created without pass the time of the intent (so is set automatically to 0 by HostPair). Then it uses the delIntent function of IntentForwarding, that iterates the intentDB list and removes the intent with IPs passed through JSON. Moreover, this function uses also the denyRoute (explained before) to install routes both in switch1 and switch2 to deny communications between the two hosts.

1.3.3 Get intents

"/getIntents/json" API is used to retrieve all intents in a certain moment. It uses the GetIntents class that thanks to the getIntent function of IntentForwarding, it gets a list of HostPair (out intents) and return a list of JSON object representing them.

1.3.4 Manage timeout of deny

"/ManageTimeout/json" API allow us to get or change (according to the fact that we are doing a get or a post request) the time that a rule to deny communication between hosts is valid. This use the ManageTimeout class that use getDenyTimeout and setDenyTimeout functions of IntentForwarding to get or change the timeout time of rules respectively.

1.4 Responsiveness To Link Failures & Topology Changes

The responsiveness of the topology is reached thanks to the ITopologyListener inteface. In fact in the Floodligh documentation for the TopologyService it is written: *"All the information about the current topology is stored in an immutable data structure called the topology instance. If there is any change in the topology, a new instance is created and the topology changed notification message is called. If other modules want to listen for changes in topology they can implement the ITopologyListener interface."*[4].
According to this we know that Floodlight handle automatically link failures and also topology changes. To do this it exploits two main techniques:

- Every time arrives a packet of which its path is unknown, a new path is computed and is taken up for 5 seconds. If in this time no packets follow this path, this rule is deleted.
- Every time that a link failure is happened, a new path is computed.

1.5 UML Class Diagram

In figure 1.2 we can see the UML class diagram of the main classes of the implemented module. For simplicity only the most relevant methods and attribute are present in the diagram.

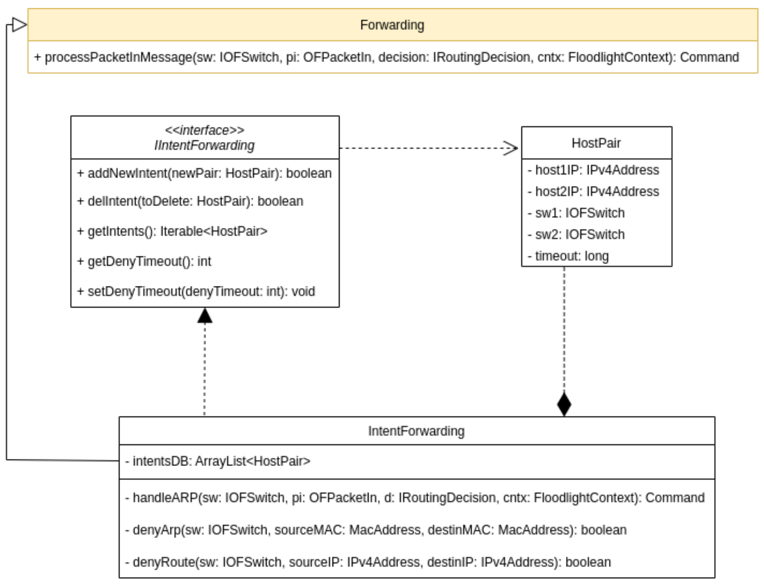

Forwarding
+ processPacketInMessage(sw: IOFSwitch, pi: OFPacketIn, decision: IRoutingDecision, cntx: FloodlightContext): Command
<<interface>>
IIntentForwarding
+ addNewIntent(newPair: HostPair): boolean
+ delIntent(toDelete: HostPair): boolean
+ getIntents(): Iterable<HostPair>
+ getDenyTimeout(): int
+ setDenyTimeout(denyTimeout: int): void
HostPair
- host1IP: IPv4Address
- host2IP: IPv4Address
- sw1: IOFSwitch
- sw2: IOFSwitch
- timeout: long
IntentForwarding
- intentsDB: ArrayList<HostPair>
- handleARP(sw: IOFSwitch, pi: OFPacketIn, d: IRoutingDecision, cntx: FloodlightContext): Command
- denyArp(sw: IOFSwitch, sourceMAC: MacAddress, destinMAC: MacAddress): boolean
- denyRoute(sw: IOFSwitch, sourceIP: IPv4Address, destinIP: IPv4Address): boolean

Figure 1.2: UML Class Diagram

[4]https://floodlight.atlassian.net/wiki/spaces/floodlightcontroller/pages/1343623/TopologyService+Dev

Chapter 2

Testing

Languages and Frameworks

In order to test our system, we need to use a tool to create a *virtual network* inside our machine: we used `mininet`[1], exploiting the `python2` APIs. To simulate input from an external application we also used a python2 library, called `requests`[2], which is able to act like an HTTP client. Given that our developement environment is inside a virtual machine, we will keep the number of virtual devices relatively small, in order to not overload the VM resources.

Scenario

We want to simulate a typical data center scenario with a single LAN, implemented in a *Spine-Leaf* topology. This configuration is widely used thanks to its easy scalability and sufficient redundancy.

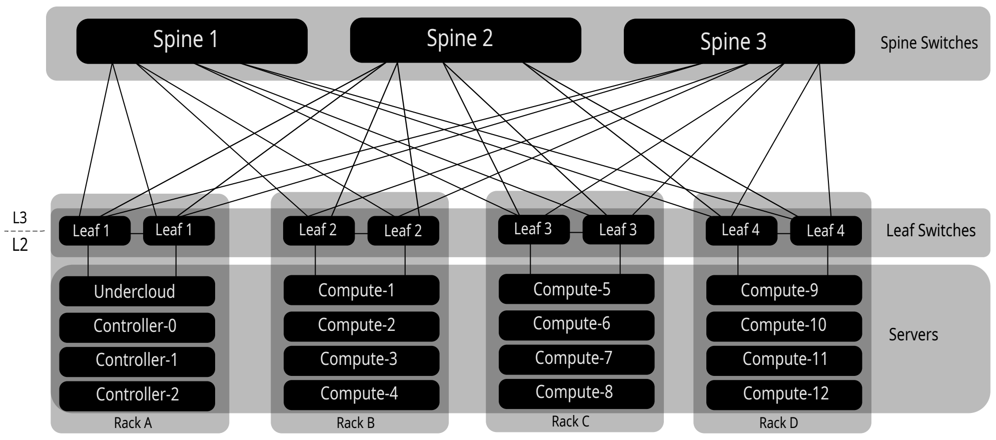
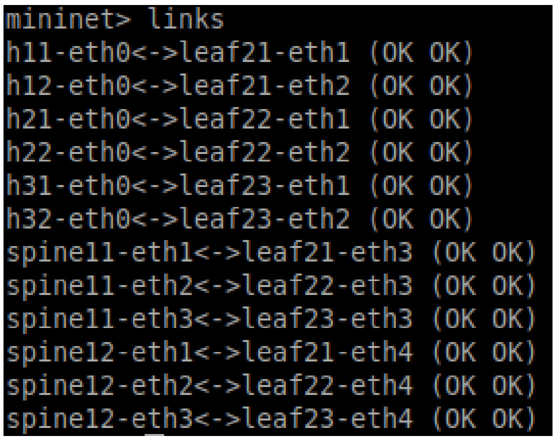


Figure 2.1: example of *spine-leaf* topology

2.1 Link Failure Test

Using `mininet`, we can also simulate an episode of link failure, in order to test the system behaviour in this specific case. The system is able to recompute a functional path between two host and use it to forward packets.
In order to perform this test we use a spine leaf network with 2 spines and 3 leaf, each leaf has two hosts connected to it. The links of the network can be seen in figure 2.2

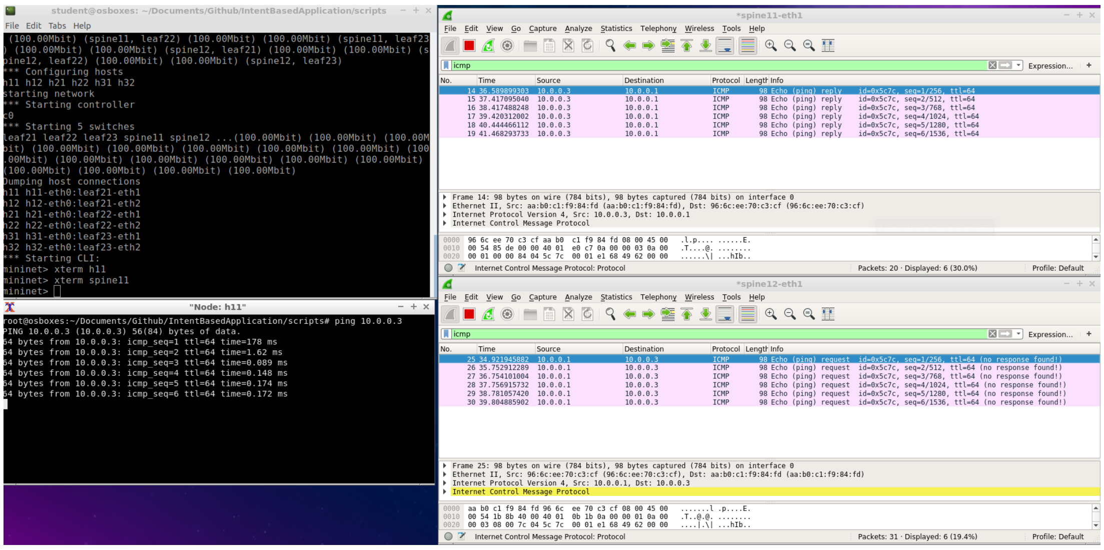
[1]http://mininet.org/
[2]https://docs.python-requests.org/en/latest/



```
mininet> links
h11-eth0<->leaf21-eth1 (OK OK)
h12-eth0<->leaf21-eth2 (OK OK)
h21-eth0<->leaf22-eth1 (OK OK)
h22-eth0<->leaf22-eth2 (OK OK)
h31-eth0<->leaf23-eth1 (OK OK)
h32-eth0<->leaf23-eth2 (OK OK)
spine11-eth1<->leaf21-eth3 (OK OK)
spine11-eth2<->leaf22-eth3 (OK OK)
spine11-eth3<->leaf23-eth3 (OK OK)
spine12-eth1<->leaf21-eth4 (OK OK)
spine12-eth2<->leaf22-eth4 (OK OK)
spine12-eth3<->leaf23-eth4 (OK OK)
```

Figure 2.2: Links of the spine leaf network

To prove that the system is capable of handling a link failure we will establish an intent, use wireshark to sniff the traffic on the interfaces of the spines and see what happens when a link failure takes place and how the traffic is redirected.
The first thing to do is obviously to start the floodlight controller and the network[3], then we establish an intent between the host11 (connected to leaf21) and the host21 (connected to leaf22). In order to check the connectivity we use wireshark on spine11-eth1 (the interface that connects leaf21 with spine11) and spine12-eth1 (the interface that connects leaf21 with spine12) and we perform a ping between the two hosts to see what happens on these interfaces.



```
student@osboxes: ~/Documents/Github/IntentBasedApplication/scripts
 (100.00Mbit) (spine11, leaf22) (100.00Mbit) (100.00Mbit) (spine11, leaf23
) (100.00Mbit) (100.00Mbit) (spine12, leaf21) (100.00Mbit) (100.00Mbit) (s
pine12, leaf22) (100.00Mbit) (100.00Mbit) (spine12, leaf23)
*** Configuring hosts
h11 h12 h21 h22 h31 h32
starting network
*** Starting controller
c0
*** Starting 5 switches
leaf21 leaf22 leaf23 spine11 spine12 ...(100.00Mbit) (100.00Mbit) (100.00M
bit) (100.00Mbit) (100.00Mbit) (100.00Mbit) (100.00Mbit) (100.00Mbit) (100
.00Mbit) (100.00Mbit) (100.00Mbit) (100.00Mbit) (100.00Mbit) (100.00Mbit)
(100.00Mbit) (100.00Mbit) (100.00Mbit) (100.00Mbit)
Dumping host connections
h11 h11-eth0:leaf21-eth1
h12 h12-eth0:leaf21-eth2
h21 h21-eth0:leaf22-eth1
h22 h22-eth0:leaf22-eth2
h31 h31-eth0:leaf23-eth1
h32 h32-eth0:leaf23-eth2
*** Starting CLI:
mininet> xterm h11
mininet> xterm spine11
mininet> 
```

```
"Node: h11"
root@osboxes:~/Documents/Github/IntentBasedApplication/scripts# ping 10.0.0.3
PING 10.0.0.3 (10.0.0.3) 56(84) bytes of data.
64 bytes from 10.0.0.3: icmp_seq=1 ttl=64 time=178 ms
64 bytes from 10.0.0.3: icmp_seq=2 ttl=64 time=1.62 ms
64 bytes from 10.0.0.3: icmp_seq=3 ttl=64 time=0.089 ms
64 bytes from 10.0.0.3: icmp_seq=4 ttl=64 time=0.148 ms
64 bytes from 10.0.0.3: icmp_seq=5 ttl=64 time=0.174 ms
64 bytes from 10.0.0.3: icmp_seq=6 ttl=64 time=0.172 ms
```

```
*spine11-eth1
icmp
No.  Time          Source    Destination Protocol Length Info
  14 36.589899303  10.0.0.3  10.0.0.1    ICMP     98 Echo (ping) reply    id=0x5c7c, seq=1/256, ttl=64
  15 37.417095040  10.0.0.3  10.0.0.1    ICMP     98 Echo (ping) reply    id=0x5c7c, seq=2/512, ttl=64
  16 38.417488248  10.0.0.3  10.0.0.1    ICMP     98 Echo (ping) reply    id=0x5c7c, seq=3/768, ttl=64
  17 39.420312002  10.0.0.3  10.0.0.1    ICMP     98 Echo (ping) reply    id=0x5c7c, seq=4/1024, ttl=64
  18 40.444466112  10.0.0.3  10.0.0.1    ICMP     98 Echo (ping) reply    id=0x5c7c, seq=5/1280, ttl=64
  19 41.468293733  10.0.0.3  10.0.0.1    ICMP     98 Echo (ping) reply    id=0x5c7c, seq=6/1536, ttl=64

Frame 14: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on interface 0
Ethernet II, Src: aa:b0:c1:f9:84:fd (aa:b0:c1:f9:84:fd), Dst: 96:6c:ee:70:c3:cf (96:6c:ee:70:c3:cf)
Internet Protocol Version 4, Src: 10.0.0.3, Dst: 10.0.0.1
Internet Control Message Protocol

0000  96 6c ee 70 c3 cf aa b0  c1 f9 84 fd 08 00 45 00   .l.p.... ......E.
0010  00 54 85 de 00 00 40 01  e0 c7 0a 00 00 03 0a 00   .T....@. ........
0020  00 01 00 00 84 04 5c 7c  00 01 e1 68 49 62 00 00   ......\| ...hIb..
Internet Control Message Protocol: Protocol        Packets: 20 · Displayed: 6 (30.0%)        Profile: Default
```

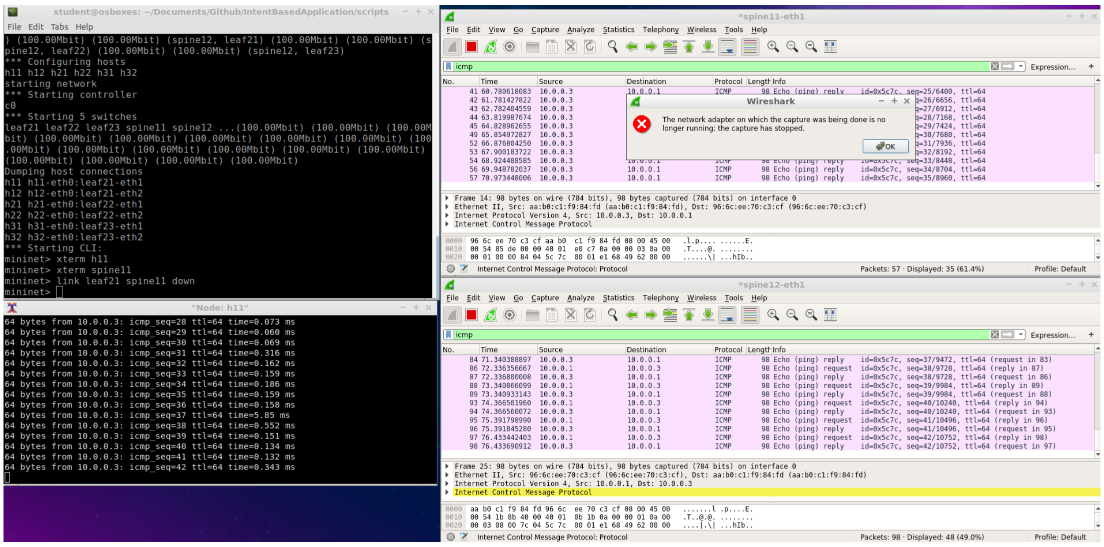
```
*spine12-eth1
icmp
No.  Time          Source    Destination Protocol Length Info
  25 34.921945882  10.0.0.1  10.0.0.3    ICMP     98 Echo (ping) request  id=0x5c7c, seq=1/256, ttl=64 (no response found!)
  26 35.752912289  10.0.0.1  10.0.0.3    ICMP     98 Echo (ping) request  id=0x5c7c, seq=2/512, ttl=64 (no response found!)
  27 36.754101004  10.0.0.1  10.0.0.3    ICMP     98 Echo (ping) request  id=0x5c7c, seq=3/768, ttl=64 (no response found!)
  28 37.756915732  10.0.0.1  10.0.0.3    ICMP     98 Echo (ping) request  id=0x5c7c, seq=4/1024, ttl=64 (no response found!)
  29 38.781057420  10.0.0.1  10.0.0.3    ICMP     98 Echo (ping) request  id=0x5c7c, seq=5/1280, ttl=64 (no response found!)
  30 39.804885902  10.0.0.1  10.0.0.3    ICMP     98 Echo (ping) request  id=0x5c7c, seq=6/1536, ttl=64 (no response found!)

Frame 25: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on interface 0
Ethernet II, Src: 96:6c:ee:70:c3:cf (96:6c:ee:70:c3:cf), Dst: aa:b0:c1:f9:84:fd (aa:b0:c1:f9:84:fd)
Internet Protocol Version 4, Src: 10.0.0.1, Dst: 10.0.0.3
Internet Control Message Protocol

0000  aa b0 c1 f9 84 fd 96 6c  ee 70 c3 cf 08 00 45 00   .......l .p....E.
0010  00 54 1b 8b 40 00 40 01  0b 1b 0a 00 00 01 0a 00   .T..@.@. ........
0020  00 03 08 00 7c 04 5c 7c  00 01 e1 68 49 62 00 00   ....|.\| ...hIb..
Internet Control Message Protocol: Protocol        Packets: 31 · Displayed: 6 (19.4%)        Profile: Default
```

File Edit Tabs Help
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Expression...

Figure 2.3: Ping between host11 and host21

As we can see in figure 2.3 the ping is executed successfully and both interfaces are used, one for sending packets from h11 to h21 and the other for the reverse path.
Now we put down the link between leaf21 and spine11, so the interface spine11-eth1 goes down (see figure 2.4). Nevertheless we can see that the ping continues to be performed successfully because now only the interface spine12-eth1 is used for both the direct and the reverse path (figure 2.4).
The only indication that a link failure has took place is the fact that one ping needs more time to be completed (see figure 2.5) because of the overhead caused by the reconfiguration of the path, but the connectivity is ensured in any case (in fact in figure 2.5 we can see that no icmp packets have been lost because there is no gap in icmp_seq) and the delay affects only one packet. Hence at the end we can say that the intent continues to work even if a link fails, thus the system is able to cope with a link failure.

[3]through the script scripts/start_only_spine_topo.py



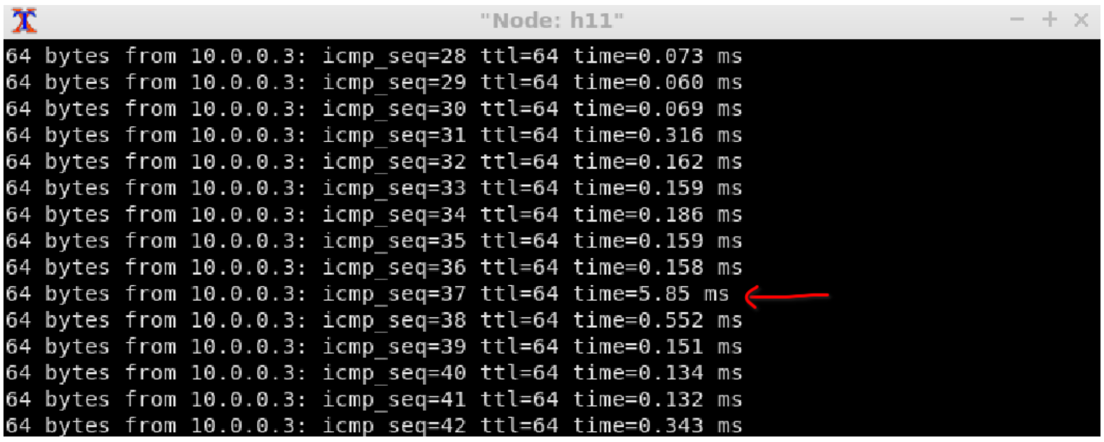
```
student@osboxes: ~/Documents/Github/IntentBasedApplication/scripts
) (100.00Mbit) (100.00Mbit) (spine12, leaf21) (100.00Mbit) (100.00Mbit) (s
pine12, leaf22) (100.00Mbit) (100.00Mbit) (spine12, leaf23)
*** Configuring hosts
h11 h12 h21 h22 h31 h32
starting network
*** Starting controller
c0
*** Starting 5 switches
leaf21 leaf22 leaf23 spine11 spine12 ...(100.00Mbit) (100.00Mbit) (100.00M
bit) (100.00Mbit) (100.00Mbit) (100.00Mbit) (100.00Mbit) (100.00Mbit) (100
.00Mbit) (100.00Mbit) (100.00Mbit) (100.00Mbit) (100.00Mbit) (100.00Mbit)
(100.00Mbit) (100.00Mbit) (100.00Mbit) (100.00Mbit)
Dumping host connections
h11 h11-eth0:leaf21-eth1
h12 h12-eth0:leaf21-eth2
h21 h21-eth0:leaf22-eth1
h22 h22-eth0:leaf22-eth2
h31 h31-eth0:leaf23-eth1
h32 h32-eth0:leaf23-eth2
*** Starting CLI:
mininet> xterm h11
mininet> xterm spine11
mininet> link leaf21 spine11 down
mininet> 
```

```
"Node: h11"
64 bytes from 10.0.0.3: icmp_seq=28 ttl=64 time=0.073 ms
64 bytes from 10.0.0.3: icmp_seq=29 ttl=64 time=0.060 ms
64 bytes from 10.0.0.3: icmp_seq=30 ttl=64 time=0.069 ms
64 bytes from 10.0.0.3: icmp_seq=31 ttl=64 time=0.316 ms
64 bytes from 10.0.0.3: icmp_seq=32 ttl=64 time=0.162 ms
64 bytes from 10.0.0.3: icmp_seq=33 ttl=64 time=0.159 ms
64 bytes from 10.0.0.3: icmp_seq=34 ttl=64 time=0.186 ms
64 bytes from 10.0.0.3: icmp_seq=35 ttl=64 time=0.159 ms
64 bytes from 10.0.0.3: icmp_seq=36 ttl=64 time=0.158 ms
64 bytes from 10.0.0.3: icmp_seq=37 ttl=64 time=5.85 ms
64 bytes from 10.0.0.3: icmp_seq=38 ttl=64 time=0.552 ms
64 bytes from 10.0.0.3: icmp_seq=39 ttl=64 time=0.151 ms
64 bytes from 10.0.0.3: icmp_seq=40 ttl=64 time=0.134 ms
64 bytes from 10.0.0.3: icmp_seq=41 ttl=64 time=0.132 ms
64 bytes from 10.0.0.3: icmp_seq=42 ttl=64 time=0.343 ms
```

```
*spine11-eth1
icmp
No.  Time          Source     Destination  Protocol Length Info
  41 60.780618083  10.0.0.3   10.0.0.1     ICMP       98 Echo (ping) reply    id=0x5c7c, seq=25/6400, ttl=64
  42 61.781427822  10.0.0.3                                                   q=26/6656, ttl=64
  43 62.782404559  10.0.0.3                                                   q=27/6912, ttl=64
  44 63.819987674  10.0.0.3                                                   q=28/7168, ttl=64
  45 64.828962655  10.0.0.3                                                   q=29/7424, ttl=64
  49 65.854972827  10.0.0.3                                                   q=30/7680, ttl=64
  52 66.876804250  10.0.0.3                                                   q=31/7936, ttl=64
  53 67.900183722  10.0.0.3                                                   q=32/8192, ttl=64
  54 68.924488585  10.0.0.3   10.0.0.1     ICMP       98 Echo (ping) reply    id=0x5c7c, seq=33/8448, ttl=64
  56 69.948782037  10.0.0.3   10.0.0.1     ICMP       98 Echo (ping) reply    id=0x5c7c, seq=34/8704, ttl=64
  57 70.973448006  10.0.0.3   10.0.0.1     ICMP       98 Echo (ping) reply    id=0x5c7c, seq=35/8960, ttl=64

Wireshark
The network adapter on which the capture was being done is no
longer running; the capture has stopped.
OK

Frame 14: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on interface 0
Ethernet II, Src: aa:b0:c1:f9:84:fd (aa:b0:c1:f9:84:fd), Dst: 96:6c:ee:70:c3:cf (96:6c:ee:70:c3:cf)
Internet Protocol Version 4, Src: 10.0.0.3, Dst: 10.0.0.1
Internet Control Message Protocol

0000  96 6c ee 70 c3 cf aa b0  c1 f9 84 fd 08 00 45 00   .l.p.... ......E.
0010  00 54 85 de 00 00 40 01  e0 c7 0a 00 00 03 0a 00   .T....@. ........
0020  00 01 00 00 84 04 5c 7c  00 01 e1 68 49 62 00 00   ......\| ...hIb..

Internet Control Message Protocol: Protocol     Packets: 57 · Displayed: 35 (61.4%)     Profile: Default
```

```
*spine12-eth1
icmp
No.  Time          Source     Destination  Protocol Length Info
  84 71.340388897  10.0.0.3   10.0.0.1     ICMP       98 Echo (ping) reply    id=0x5c7c, seq=37/9472, ttl=64 (request in 83)
  86 72.336356667  10.0.0.1   10.0.0.3     ICMP       98 Echo (ping) request  id=0x5c7c, seq=38/9728, ttl=64 (reply in 87)
  87 72.336800008  10.0.0.3   10.0.0.1     ICMP       98 Echo (ping) reply    id=0x5c7c, seq=38/9728, ttl=64 (request in 86)
  88 73.340866099  10.0.0.1   10.0.0.3     ICMP       98 Echo (ping) request  id=0x5c7c, seq=39/9984, ttl=64 (reply in 89)
  89 73.340933143  10.0.0.3   10.0.0.1     ICMP       98 Echo (ping) reply    id=0x5c7c, seq=39/9984, ttl=64 (request in 88)
  93 74.366501960  10.0.0.1   10.0.0.3     ICMP       98 Echo (ping) request  id=0x5c7c, seq=40/10240, ttl=64 (reply in 94)
  94 74.366560072  10.0.0.3   10.0.0.1     ICMP       98 Echo (ping) reply    id=0x5c7c, seq=40/10240, ttl=64 (request in 93)
  95 75.391798990  10.0.0.1   10.0.0.3     ICMP       98 Echo (ping) request  id=0x5c7c, seq=41/10496, ttl=64 (reply in 96)
  96 75.391845280  10.0.0.3   10.0.0.1     ICMP       98 Echo (ping) reply    id=0x5c7c, seq=41/10496, ttl=64 (request in 95)
  97 76.433442403  10.0.0.1   10.0.0.3     ICMP       98 Echo (ping) request  id=0x5c7c, seq=42/10752, ttl=64 (reply in 98)
  98 76.433690912  10.0.0.3   10.0.0.1     ICMP       98 Echo (ping) reply    id=0x5c7c, seq=42/10752, ttl=64 (request in 97)

Frame 25: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on interface 0
Ethernet II, Src: 96:6c:ee:70:c3:cf (96:6c:ee:70:c3:cf), Dst: aa:b0:c1:f9:84:fd (aa:b0:c1:f9:84:fd)
Internet Protocol Version 4, Src: 10.0.0.1, Dst: 10.0.0.3
Internet Control Message Protocol

0000  aa b0 c1 f9 84 fd 96 6c  ee 70 c3 cf 08 00 45 00   .......l .p....E.
0010  00 54 1b 8b 40 00 40 01  0b 1b 0a 00 00 01 0a 00   .T..@.@. ........
0020  00 03 08 00 7c 04 5c 7c  00 01 e1 68 49 62 00 00   ....|.\| ...hIb..

Internet Control Message Protocol: Protocol     Packets: 98 · Displayed: 48 (49.0%)     Profile: Default
```

Figure 2.4: We put down the link between leaf21 and spine11 but the ping continues to work.



```
"Node: h11"
64 bytes from 10.0.0.3: icmp_seq=28 ttl=64 time=0.073 ms
64 bytes from 10.0.0.3: icmp_seq=29 ttl=64 time=0.060 ms
64 bytes from 10.0.0.3: icmp_seq=30 ttl=64 time=0.069 ms
64 bytes from 10.0.0.3: icmp_seq=31 ttl=64 time=0.316 ms
64 bytes from 10.0.0.3: icmp_seq=32 ttl=64 time=0.162 ms
64 bytes from 10.0.0.3: icmp_seq=33 ttl=64 time=0.159 ms
64 bytes from 10.0.0.3: icmp_seq=34 ttl=64 time=0.186 ms
64 bytes from 10.0.0.3: icmp_seq=35 ttl=64 time=0.159 ms
64 bytes from 10.0.0.3: icmp_seq=36 ttl=64 time=0.158 ms
64 bytes from 10.0.0.3: icmp_seq=37 ttl=64 time=5.85 ms
64 bytes from 10.0.0.3: icmp_seq=38 ttl=64 time=0.552 ms
64 bytes from 10.0.0.3: icmp_seq=39 ttl=64 time=0.151 ms
64 bytes from 10.0.0.3: icmp_seq=40 ttl=64 time=0.134 ms
64 bytes from 10.0.0.3: icmp_seq=41 ttl=64 time=0.132 ms
64 bytes from 10.0.0.3: icmp_seq=42 ttl=64 time=0.343 ms
```

Figure 2.5

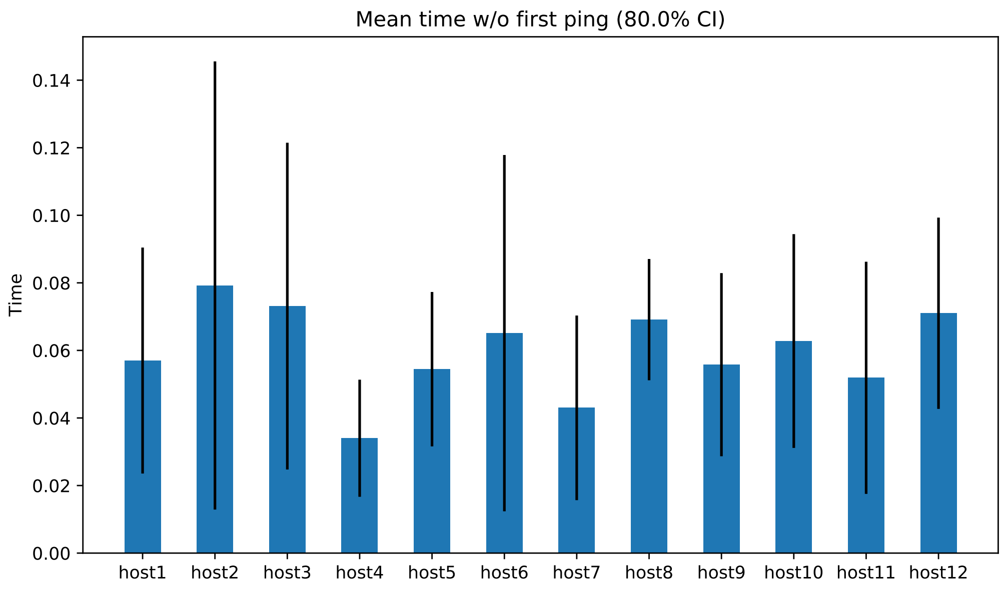
2.2 Ping Latency

In order to measure the network performance we will consider a fixed scenario with 2 spines, 3 leafs and 4 hosts for each leafs (i.e. 12 hosts in total); for every host:

- we establish an intent with another random chosen one;
- we start a ping session, measuring the `round trip time` of each packet
- the session end when 10 ping are successfully exchanged, or the last timeout elapses

First of all we monitor the number of `DESTINATION HOST UNREACHABLE` and `DUPLICATE`, those 2 parameters are equal to 0, confirming the correct behaviour of the network. The performance results (see figure 2.6) are consistent with our expectation:

- the first ping is significantly slower than the others, because it is processed by the controller, triggering the route establishment;
- subsequent pings are very fast (<1 ms), because they don't have to traverse "real" network cables and because they are processed by the virtual switches and not by the controller.



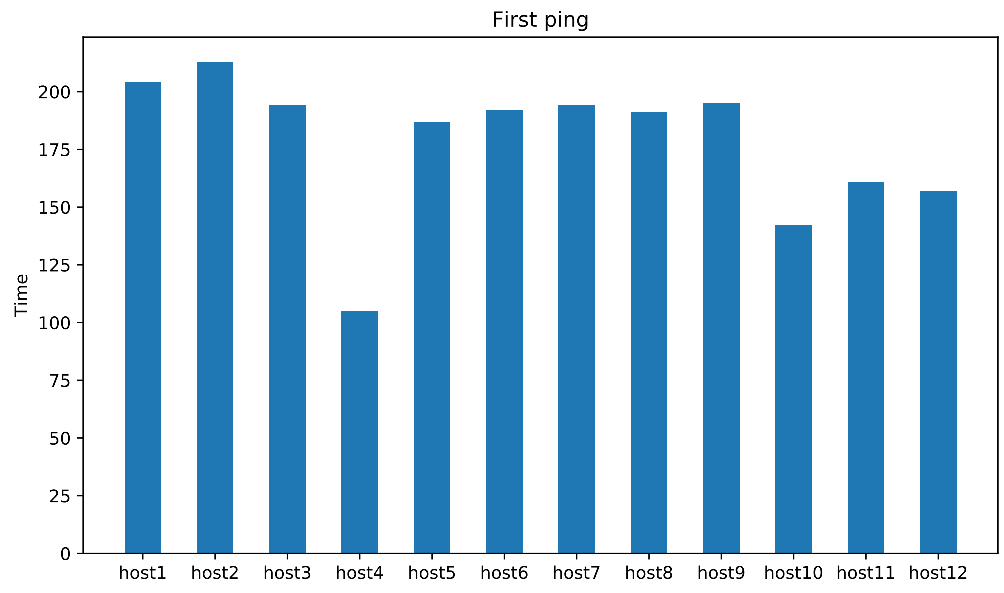


Figure 2.6: Ping times with 12 hosts

We also tried to evaluate the elasticity of the network with increasing load of traffic (i.e. increasing number of host pinging at the same time) and the results of this tests can be seen in figure 2.7.
The time needed for the first ping to complete tends to increase with the amount of traffic meanwhile the time for the others pings remains pretty stable (mind the confidence intervals). This is expected because route establishment is done by the controller in a centralized way (the requests will queue up); once the route is established, switches will handle the traffic in a distributed manner, leading to optimal forwarding of traffic.

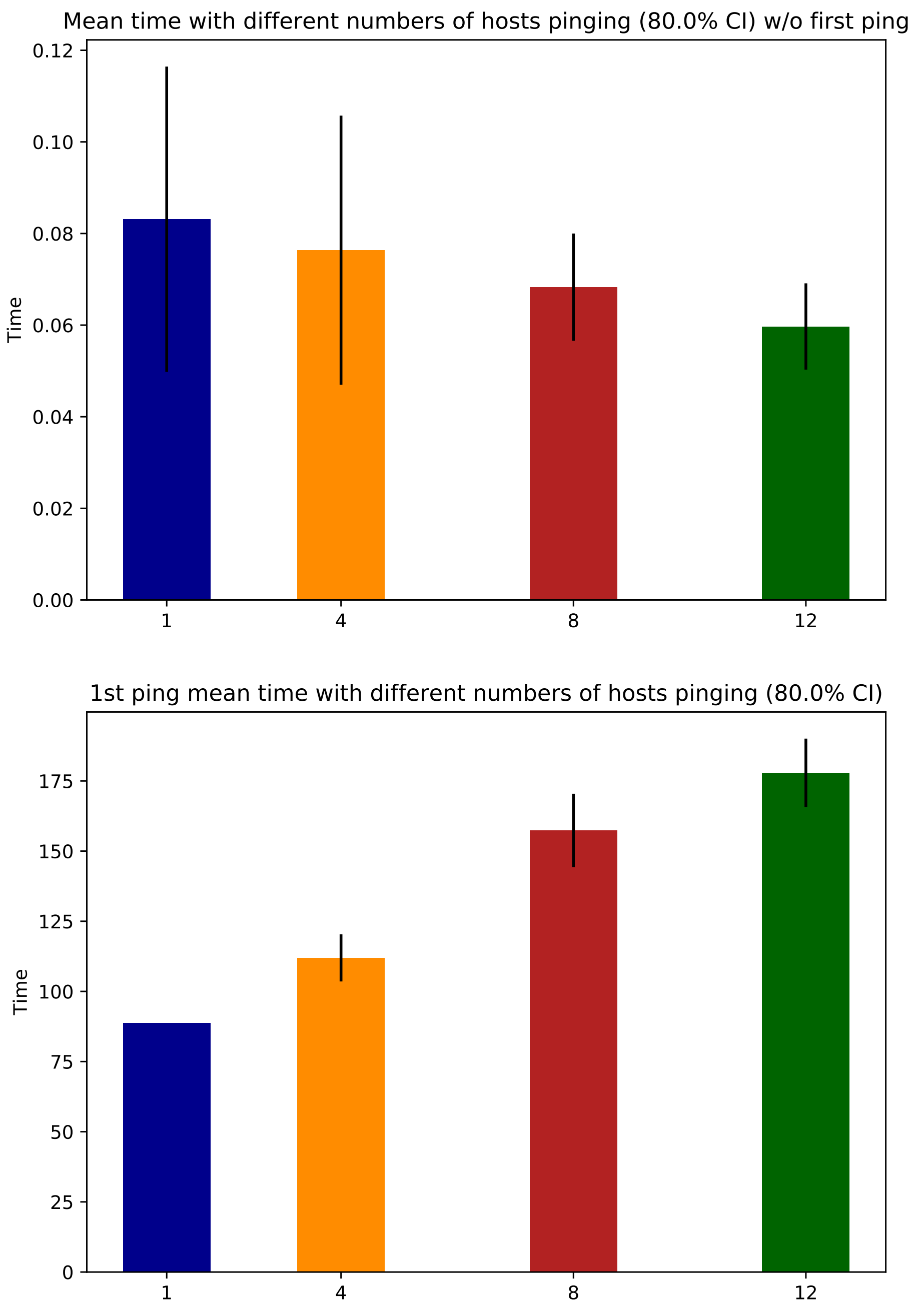


Figure 2.7: Ping times with different loads