Software Defined Network

Sherwin Sathish, Electrical and Computer Engineering
Mentor: Dr. Tamer Omar
RSCA 2024

Introduction

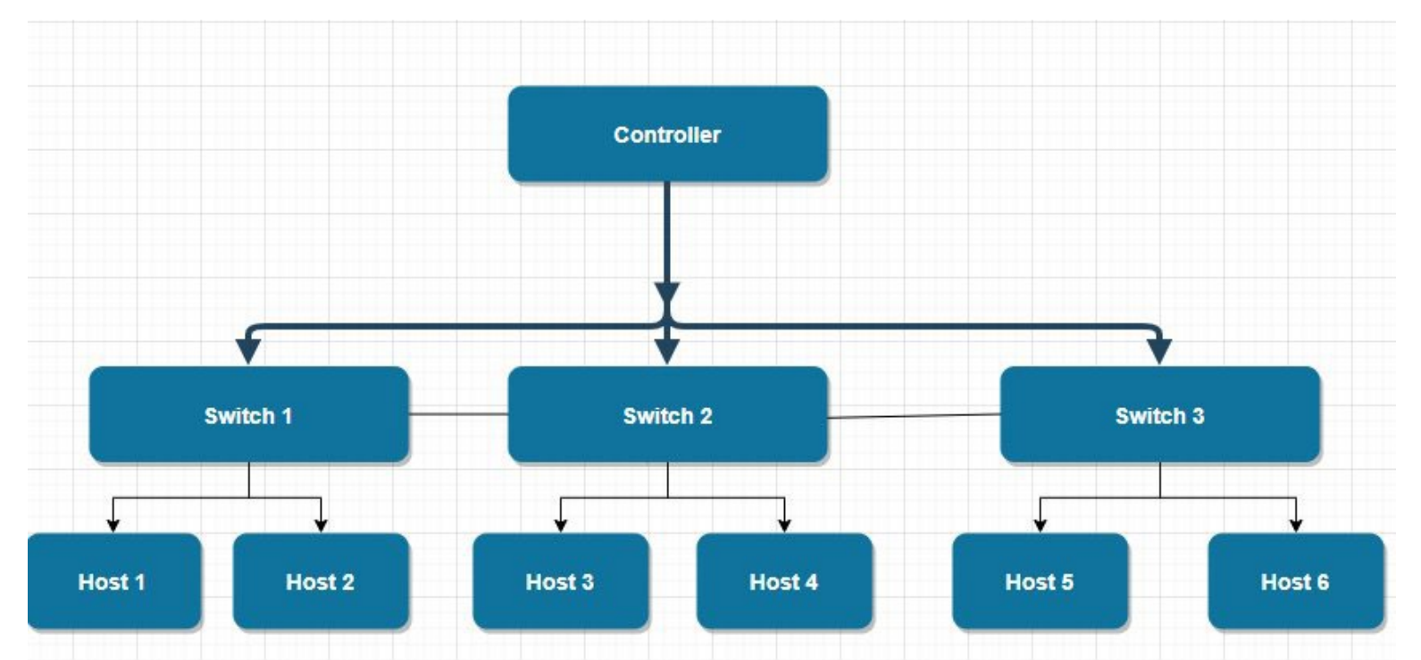
Software-Defined Networking (SDN) is an approach to networking that uses software-based controllers or application programming interfaces (APIs) to communicate with underlying hardware infrastructure and direct traffic on a network. SDN is the concept of functional separation, which divides the network into three layers: the application layer, control layer, and infrastructure layer. The application layer hosts various network services such as intrusion detection systems and load balancers, all implemented as software applications. These applications communicate with the control layer, which houses the SDN controller software responsible for centrally managing policies and traffic flows. The infrastructure layer consists of physical switches that forward network traffic based on instructions received from the controller. SDNs are gaining popularity but are at risk to Distributed Denial of Service (DDoS) Attacks. Few methods have been found for fighting against the attacks. In many methods, the primary controller will be replaced with a second or third one, but this is costly. As users in a network send messages to each other it is critical that the message content can go host to host without any interception of information. A cost-efficient solution is required to circumnavigate any potential attack to ensure safe and secure transfer of data between hosts.

Objective

This project explores the implementation of a Software-Defined Networking (SDN) topology using Mininet to demonstrate the principles of centralized network control and programmability. The OpenFlow protocol is used, where switches within the network are controlled by a central SDN controller. The topology incorporates switches, hosts, and a controller, creating an environment for efficient data packet forwarding. With the threat of a DDOS attack viable cost-efficient solutions of defense must be explored as well. Once the network is set up it is necessary to incorporate a detection and mitigation system into the existing network infrastructure.

Materials and Methods

The diagram below is the structure of the virtual network that I created:



Methods:

Install Mininet: I began by installing Mininet on a virtual machine using VirtualBox and Ubuntu. This involves updating the package repository, installing necessary dependencies, cloning the Mininet repository, and running the installation script.

Set Up the Topology: After installing Mininet, I proceeded to set up the network topology by creating Python script. This script utilizes the Mininet library to define the topology, including switches, hosts, and their connections. The topology script creates a network with three switches and six hosts interconnected in a predefined manner.

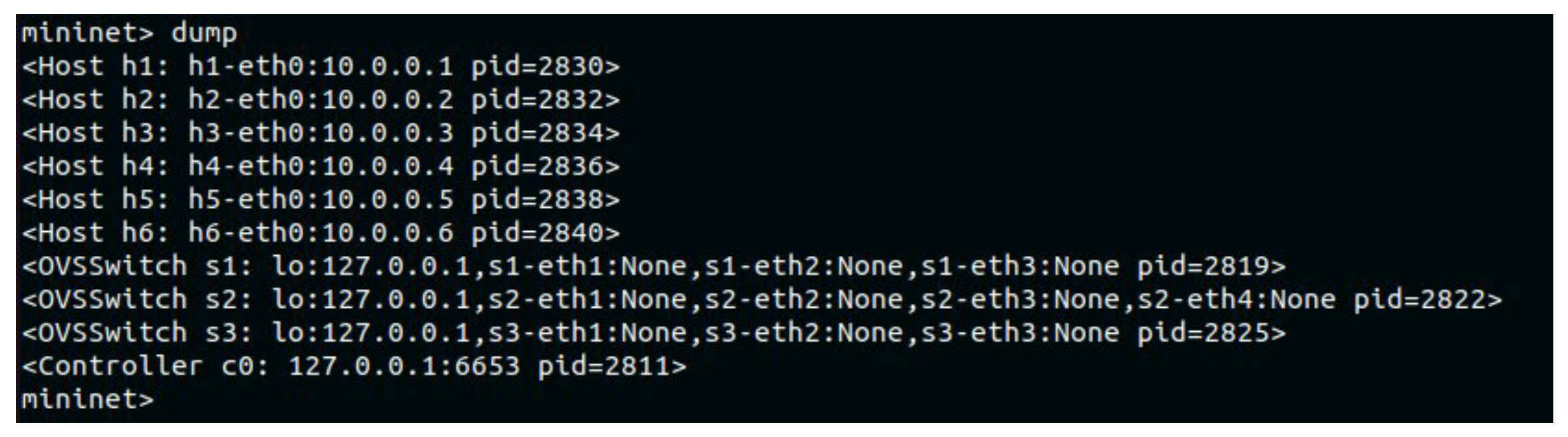
Test Data Transmission: With the network topology configured, I conducted tests to ensure proper data transmission within the network. I utilized the Mininet command-line interface (CLI) to ping between hosts, send traffic, measure bandwidth using iperf, and display network information using various commands such as net, dump, nodes, and links.

Wireshark Setup: I installed and configured Wireshark to capture and analyze packet transfer within the SDN network. I installed Wireshark on the virtual machine using the Linux terminal. I started Wireshark and selected the appropriate network interface for packet capture and began capturing packets while running the Mininet script to simulate network traffic.

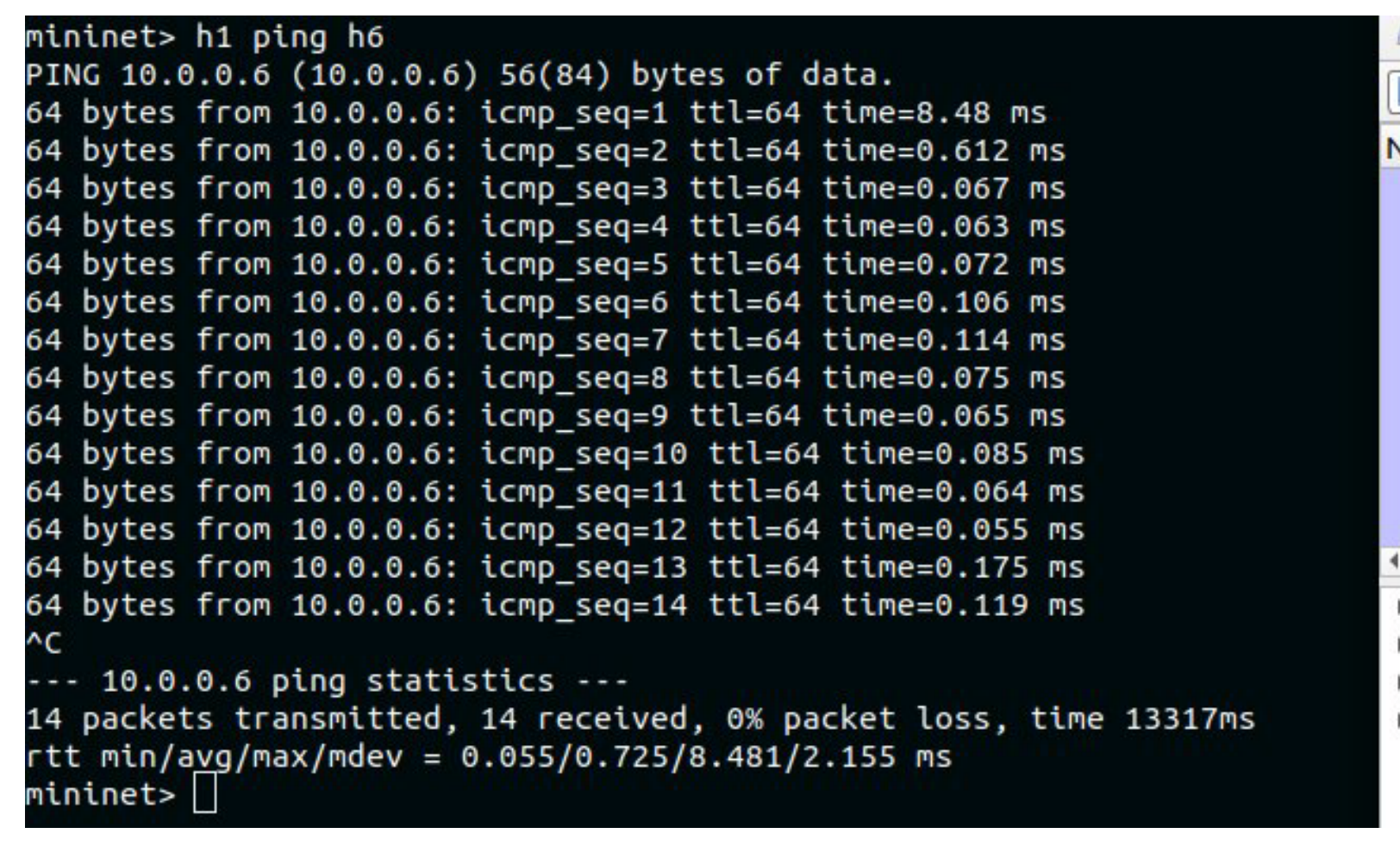
Analysis with Wireshark: I analyzed captured packets in Wireshark to gain insights into network behavior, traffic patterns, and protocol usage. I also interpreted the results to evaluate network performance and identify any anomalies or issues.

Code Explanation: I provided an explanation of the Python script used to define the network topology. The explanation breaks down the script into its components, including importing required modules, defining the topology creation function, adding controllers, switches, and hosts, establishing links between them, starting the network and CLI, and stopping the network.

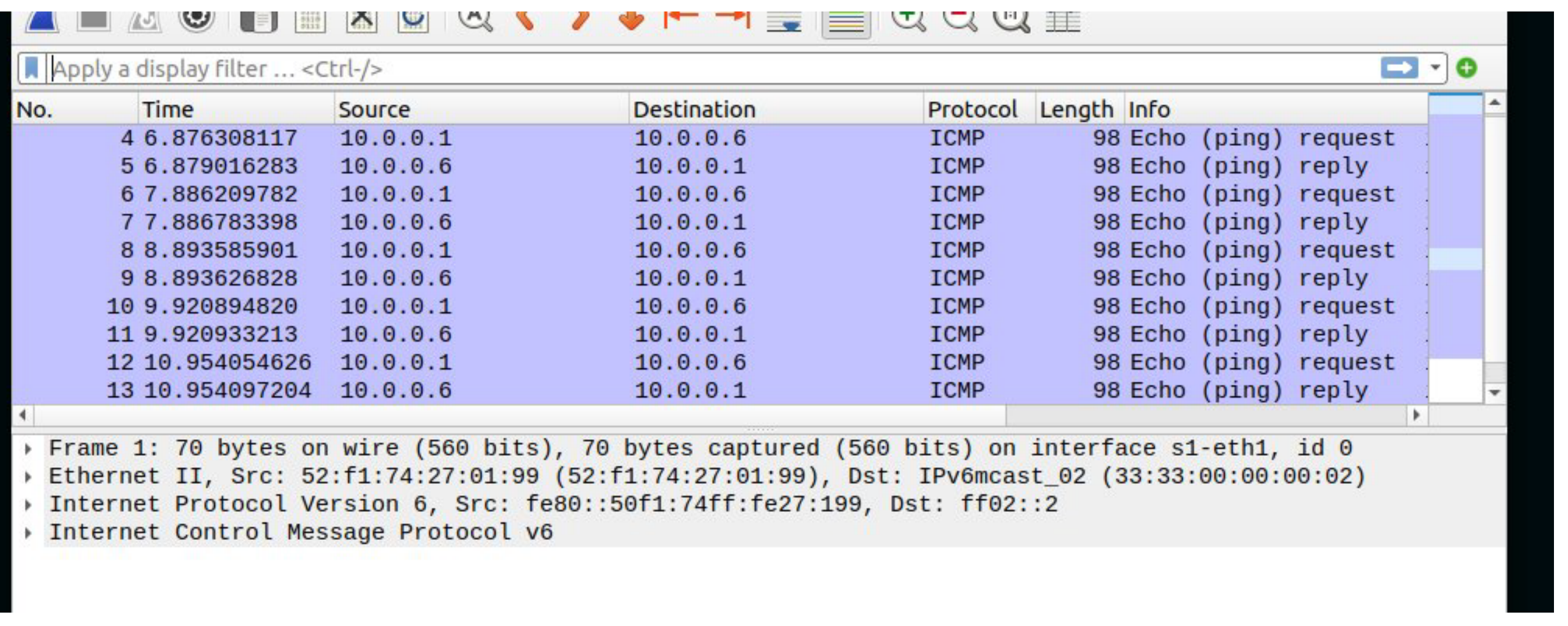
Results



```
mininet> dump
<Host h1: h1-eth0:10.0.0.1 pid=2830>
<Host h2: h2-eth0:10.0.0.2 pid=2832>
<Host h3: h3-eth0:10.0.0.3 pid=2834>
<Host h4: h4-eth0:10.0.0.4 pid=2836>
<Host h5: h5-eth0:10.0.0.5 pid=2838>
<Host h6: h6-eth0:10.0.0.6 pid=2840>
<OVSSwitch s1: lo:127.0.0.1,s1-eth1:None,s1-eth2:None,s1-eth3:None pid=2819>
<OVSSwitch s2: lo:127.0.0.1,s2-eth1:None,s2-eth2:None,s2-eth3:None,s2-eth4:None pid=2822>
<OVSSwitch s3: lo:127.0.0.1,s3-eth1:None,s3-eth2:None,s3-eth3:None pid=2825>
<Controller c0: 127.0.0.1:6653 pid=2811>
mininet>
```



```
mininet> h1 ping h6
PING 10.0.0.6 (10.0.0.6) 56(84) bytes of data.
64 bytes from 10.0.0.6: icmp_seq=1 ttl=64 time=8.48 ms
64 bytes from 10.0.0.6: icmp_seq=2 ttl=64 time=0.612 ms
64 bytes from 10.0.0.6: icmp_seq=3 ttl=64 time=0.067 ms
64 bytes from 10.0.0.6: icmp_seq=4 ttl=64 time=0.063 ms
64 bytes from 10.0.0.6: icmp_seq=5 ttl=64 time=0.072 ms
64 bytes from 10.0.0.6: icmp_seq=6 ttl=64 time=0.106 ms
64 bytes from 10.0.0.6: icmp_seq=7 ttl=64 time=0.114 ms
64 bytes from 10.0.0.6: icmp_seq=8 ttl=64 time=0.075 ms
64 bytes from 10.0.0.6: icmp_seq=9 ttl=64 time=0.065 ms
64 bytes from 10.0.0.6: icmp_seq=10 ttl=64 time=0.085 ms
64 bytes from 10.0.0.6: icmp_seq=11 ttl=64 time=0.064 ms
64 bytes from 10.0.0.6: icmp_seq=12 ttl=64 time=0.055 ms
64 bytes from 10.0.0.6: icmp_seq=13 ttl=64 time=0.175 ms
64 bytes from 10.0.0.6: icmp_seq=14 ttl=64 time=0.119 ms
^C
--- 10.0.0.6 ping statistics ---
14 packets transmitted, 14 received, 0% packet loss, time 13317ms
rtt min/avg/max/mdev = 0.055/0.725/8.481/2.155 ms
mininet> 
```



```
Apply a display filter ... <Ctrl-/>
No.   Time          Source     Destination  Protocol  Length  Info
    4 6.876308117   10.0.0.1   10.0.0.6     ICMP          98 Echo (ping) request
    5 6.879016283   10.0.0.6   10.0.0.1     ICMP          98 Echo (ping) reply
    6 7.886209782   10.0.0.1   10.0.0.6     ICMP          98 Echo (ping) request
    7 7.886783398   10.0.0.6   10.0.0.1     ICMP          98 Echo (ping) reply
    8 8.893585901   10.0.0.1   10.0.0.6     ICMP          98 Echo (ping) request
    9 8.893626828   10.0.0.6   10.0.0.1     ICMP          98 Echo (ping) reply
   10 9.920894820   10.0.0.1   10.0.0.6     ICMP          98 Echo (ping) request
   11 9.920933213   10.0.0.6   10.0.0.1     ICMP          98 Echo (ping) reply
   12 10.954054626  10.0.0.1   10.0.0.6     ICMP          98 Echo (ping) request
   13 10.954097204  10.0.0.6   10.0.0.1     ICMP          98 Echo (ping) reply

▸ Frame 1: 70 bytes on wire (560 bits), 70 bytes captured (560 bits) on interface s1-eth1, id 0
▸ Ethernet II, Src: 52:f1:74:27:01:99 (52:f1:74:27:01:99), Dst: IPv6mcast_02 (33:33:00:00:00:02)
▸ Internet Protocol Version 6, Src: fe80::50f1:74ff:fe27:199, Dst: ff02::2
▸ Internet Control Message Protocol v6
```

For the results when on the Mininet CLI using the "dump" command, detailed information about the network components within the Software-Defined Networking (SDN) topology was retrieved, providing insights into the configuration and status of the network. The output revealed the presence of six hosts, three Open vSwitch (OVS) switches, and a controller within the network topology. Each host was assigned an IP address within the subnet 10.0.0.0/24, with specific interface information provided for each host. For instance, host details included the presence of an Ethernet interface (eth0) configured with the IP address 10.0.0.1. Additionally, a Process ID (PID) was associated with each host, indicating the unique identifier assigned to the process within the system. The output further included the configuration of the OVS switches within the network. Each switch was assigned a loopback interface (lo) with the IP address 127.0.0.1, as well as a specific interface (s1-eth1) for connectivity within the network topology. These interfaces facilitate communication between switches and hosts, enabling the forwarding of data packets based on the controller's instructions. Moreover, information pertaining to the SDN controller (c0) was provided, including its IP address (127.0.0.1) and port number (6653), which are essential for establishing communication between the controller and network devices. All this data establishes connection between the different components.

The conducted pinging test was aimed to validate the successful establishment of network connectivity within the implemented Software-Defined Networking (SDN) topology using Mininet. The primary measure used to assess the success of the network connectivity was the ability to ping between hosts, as well as the corresponding packet statistics observed during the process. Successful ping responses were observed, indicating the establishment of network connectivity. Each successful ping operation resulted in a response message with statistics, including the size of the packet (64 bytes), the source IP address (10.0.0.6), the ICMP sequence number, the time to live (TTL) value (64), and the round-trip time (RTT) in milliseconds (5.93 ms). These response messages confirmed the successful transmission of ICMP echo requests and replies between hosts, validating the integrity of the network connectivity. Furthermore, packet-level analysis using Wireshark further confirmed the successful ping operation by capturing and analyzing ICMP protocol packets exchanged between the source and destination hosts. Wireshark logs revealed the presence of ICMP echo request and reply packets, confirming bidirectional communication between hosts and providing additional evidence of successful network connectivity establishment.

Summary/Future Works

Summary:

The "dump" command in the Mininet CLI provided comprehensive insights into the configuration and status of the Software-Defined Networking (SDN) topology. Detailed information about the network components, including six hosts, three Open vSwitch (OVS) switches, and a controller, was retrieved. Each host was assigned a unique IP address within the subnet 10.0.0.0/24, along with specific interface details such as Ethernet interface (eth0) configurations and associated Process IDs (PIDs). Similarly, the OVS switches were configured with loopback interfaces (lo) and specific connectivity interfaces (e.g., s1-eth1) to facilitate communication between switches and hosts. Additionally, information regarding the SDN controller (c0), including its IP address (127.0.0.1) and port number (6653), was provided, highlighting its crucial role in establishing communication with network devices. The successful pinging test further validated the network connectivity within the topology, with each ping operation resulting in response messages containing pertinent packet statistics. Analysis using Wireshark confirmed bidirectional communication between hosts through the exchange of ICMP echo request and reply packets, thus affirming the successful establishment of network connectivity. Together, these results underscore the effectiveness and reliability of the implemented SDN topology in facilitating seamless communication between network components.

Future Works:

The future plan is to develop several virtual Controllers and created DDoS attacks to one of the controllers. Also implement an auto system such that the upcoming suspicious packets were detected and redirected to a honeypot while the secondary Controller can automatically replace the primary one without any interruption, or the interruption is limited to the least situation. This plan can be put into action by creating multiple python scripts that interface with each other.

Code Breakdown

This code is written in Python and uses the Mininet library to create a virtual network topology. Here is an explanation of each part of the code.

Importing required modules:
from mininet.net import Mininet
from mininet.node import Controller, OVSSwitch, RemoteController
from mininet.cli import CLI
This code imports necessary classes from the Mininet library. It includes the Mininet class for creating the network, Controller class for specifying the controller for the network, OVSSwitch class for creating Open vSwitch switches, and CLI for the command-line interface to interact with the network.

Defining the topology creation function:

```
def create_topology():
   net = Mininet(controller=Controller, switch=OVSSwitch)
```

This function create_topology() initializes a Mininet object net by specifying the controller and switch types. In this case, it uses the Controller and OVSSwitch classes.

Adding a controller:
c0 = net.addController('c0')
This line adds a controller to the network with the name 'c0'. The controller will be responsible for managing the switches in the network.

Adding switches and hosts:
s1 = net.addSwitch('s1')

h1 = net.addHost('h1')
This code adds three switches (s1, s2, s3) and six hosts (h1, h2, h3, h4, h5, h6) to the network. Switches are the devices that route traffic between hosts, and hosts are the end devices (e.g., computers) that generate or receive traffic.

Linking hosts to switches:
net.addLink(h1, s1)
This code establishes links between the hosts and switches. Each host is connected to one switch using the addLink() method.

Linking switches together:
net.addLink(s1, s2)
This code establishes links between the switches. The switches are interconnected to form the network topology using the same addLink() method.

Starting the network and CLI:
net.start()
CLI(net)
The net.start() method starts the Mininet network, and the CLI(net) function opens a command-line interface that allows you to interact with the network. You can use the CLI to ping hosts, test connectivity, and run other commands.

Stopping the network:
net.stop()
The net.stop() method is called to stop the Mininet network and clean up the virtual network resources.

References

RSCA
Research, Scholarship and Creative Activities