Data Center Load Balancing

ECE 573: Internet Protocols

23 April 2020

Group 5
Jacob Carpenter
Michael Scherban
Shashank Shekhar
Siddharth Sheth

Individual Student Contributions

Equal work was done by all group members.

Data Center Load Balancing

Introduction

The explosion of services available through the internet has subsequently led to an increase in the size and complexity of data centers. The traditional enterprise network as seen in [1] of a hierarchical design of access, distribution, and core layers was also used in the traditional three-tier data center design [2]. However, this design relied upon the Spanning Tree Protocol to prevent forwarding loops and broadcast storms at the expense of blocked ports and reduced bandwidth. The cost of implementing a three-tier design does not scale economically nor does the performance remain adequate when up to 70% of network communication takes place within the data center [3].

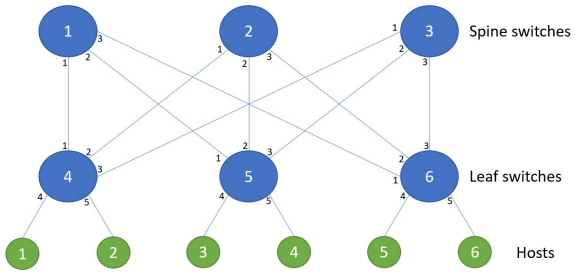
The limitation in scalability, combined with the rise of virtualization, led to an increased need to extend the Layer 2 domain across the data center to pool resources and move virtual machines [2]. This has led to the development of the spine-and-leaf architecture based on Clos networks. While there are many methods to implement this architecture, the basic concept is to create a Layer 3 underlay network and then overlay the Layer 2 network on top. This design allows for maximum flexibility in that the overlay network can be adjusted to accommodate new platforms and technologies while the underlay network remains relatively simple with point to point connections.

Just as new architectures were required to keep pace with the technical requirements of virtualization, new techniques of load balancing will also be required to make use of these architectures. Our original intent was to recreate the Presto load balancing scheme from [4] and compare it to a traditional Equal Cost Multipath (ECMP) architecture. We were unable to meet this goal due to an inability to acquire the source code from the original authors and a lack of time and resources to recreate the effort. We still emulated a traditional ECMP architecture in GNS3 while also exploring additional load balancing techniques through mininet simulation.

Design

Mininet Network

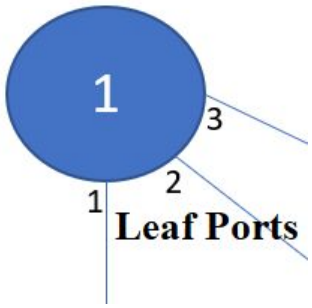
The Mininet design will allow a variable number of spine and leaf switches, as well as hosts, to be configured as a complete topology. A topology with 3 spines, 3 leaves, and 2 hosts/leaf resembles the following:



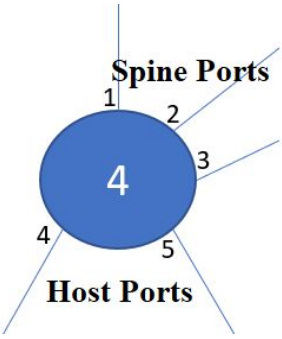
Since this network holds multiple loops it will cause a torrent of replicated packets. The switches must have intelligence to create a dynamic spanning tree, but without disabling ports. This is where the Pox controller can help the network make decisions on packet routing.

Each switch has a connection to the Pox controller, which is triggered each time there is a flow miss to make a decision on where the causing packet shall be

routed. Using this, the following rules will be implemented depending on the type of switch:



Spine Switches will behave as standard L2 learning switches. They will flood when the destination is unknown. If the destination has been seen before, it will set a flow to automatically route all future traffic to the destination port.



Leaf Switches will partially behave as L2 learning switches and are also responsible for balancing. The leaf node will be aware of the topology to know which port goes to the spine or host. For known destination packets from a host, it can only flood to other hosts and 1 spine. For unknown destination packets from the spine, it shall flood to only host ports. For destinations that are known to a host a flow will be set up. For destinations that are known over the spine, a single spine will be selected for a temporary flow.

Topology

On creation the Mininet topology accepts three arguments: the number of spine switches, leaf switches, and hosts per leaf. If no inputs are provided it will default to the 3x3x2 topology as pictured earlier in this section.

Initially, hosts are created and stored within a List. Following that, the spine switches are created so they are given lower switch number assignments and stored in a List. Finally, the leaf switches are created and stored in a List.

The switch nodes are connected in an intentional manner to assign spine switches to lower port numbers on the leaf switches. Host nodes are connected to leaf switches, but do not have any requirement for their assignment.

Pox Controller

Pox communicates with each switch using OpenFlow and has the capability to instruct switches on how to route packets. It is used extensively in this project to match a packets metadata and create forwarding rules for its flow.

On invocation Pox accepts just one argument: the number of spine switches in the network, if not provided it will default to 3. Each switch has a connection to the controller and will handshake with it as it is brought up in the network, triggering the event _handle_ConnectionUp, providing its ID.

As each switch is registered for the first time in Pox an object is created for it and stored in a Dictionary structure, switches{SwitchId:Switch}, for future reference. The switch ID is used as the Dictionary Key, the value of which is the order that it was created in. Recalling the Topology script, spine switches are created first to assign lower switch IDs so the Pox controller can distinguish spine and host ports at a leaf switch. Each switch holds its own MAC-To-Port Dictionary, mac_to_port{MAC:Port}, to keep track of which MAC addresses have been

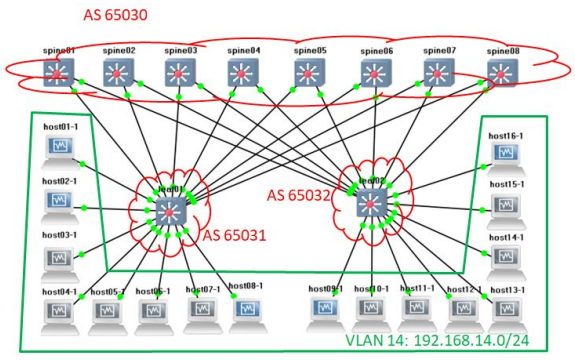
seen on which port. This structure enables L2 learning for each switch.

Pox is invoked when there is an incoming packet at a switch with no rule on how to handle it, triggering the event _handle_PacketIn. The switch provides its ID and metadata about the packet to Pox. Using the switch ID the type of switch and its object is fetched and forwarded to either a spine or leaf routine.

During the **spine routine** a standard L2 learning switch is implemented. It will log that the source MAC of the packet came from the triggering port in its dictionary. If the destination is found in the dictionary it will set up a rule to flow this packet and future ones like it to that port. If the destination is not found in the dictionary it will flood the packet to every port except the triggering one.

ECMP Architecture

The design for the External Border Gateway Protocol (eBGP) ECMP network architecture followed the design framework laid out in [5] and seen below.

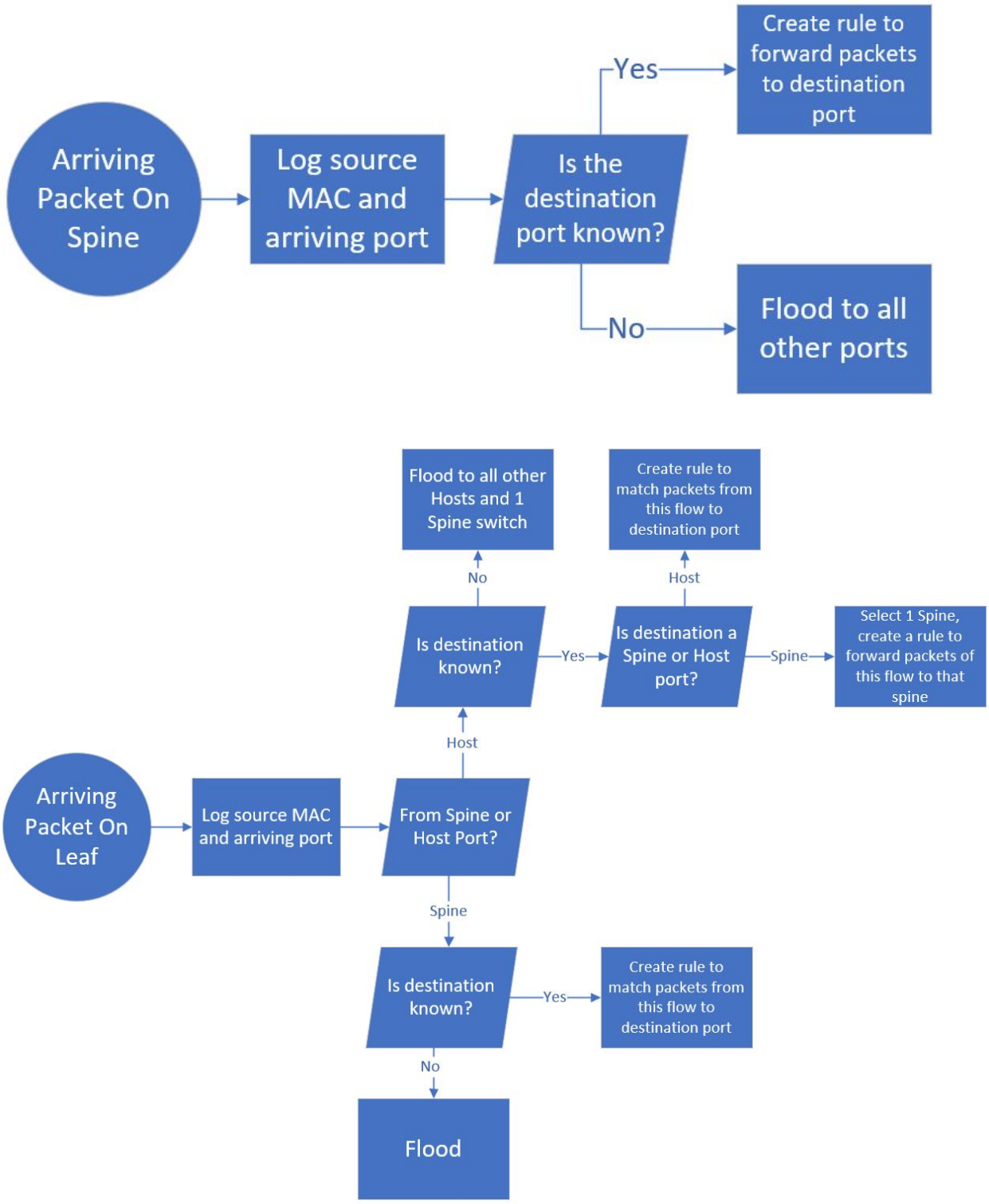


All of the spine switches were placed in autonomous system 65030 and each leaf was placed in its own autonomous system, 65031 for leaf01 and 65032 for leaf02. The underlay network consists of point to point links between each leaf to each spine with the links to and from leaf01 using subnets within 172.16.1.0/24 while the leaf02 connections utilize subnets within 172.16.2.0/24. All hosts reside on VLAN 14 which uses a network address space of 192.168.14.0/24 and both switches share the same Switch VLAN Interface of 192.168.14.1/24.

The switches are able to share this same IP address by using Virtual Extensible LAN (VXLAN) as the overlay network protocol. VXLAN encapsulates the Ethernet frames and forwards them between Virtual Tunnel Endpoints (VTEPs) [6]. VXLAN encapsulates the original frames in a UDP (port 4789) frame along with a VXLAN header that includes a 3 Byte VXLAN Network Identifier (VNI) to distinguish between individual VXLANs. In total, VXLAN adds an additional 50 Bytes of overhead: 14 Bytes for Ethernet, 20 Bytes for IP, 8 Bytes for UDP, and 8 Bytes for VXLAN headers. This additional overhead has resulted in the best practice of increasing the MTU on links in the data center [5,6].

There are many options for providing a control plane to leaf and spine architecture, such as flood-and-learn which is similar to traditional Layer 2 technologies [2], but we chose to implement Ethernet Virtual Private Network (EVPN) over BGP. This technology reduces ARP broadcasts by transmitting the MAC address/IP address mapping directly to all VTEPs in the same VNI over BGP [5].

In our topology, the VTEPs are loopback interfaces on each leaf switch with leaf01 utilizing 10.0.3.1/32 and leaf02 utilizing 10.0.3.2/32. Because the overlay network will stay the same regardless of where a host is connected, it is possible to move hosts to any leaf switch in the same VNI and not have to adjust any IP or MAC settings.


Arriving Packet On Spine
Log source MAC and arriving port
Is the destination port known?
Yes
Create rule to forward packets to destination port
No
Flood to all other ports
Arriving Packet On Leaf
Log source MAC and arriving port
From Spine or Host Port?
Host
Is destination known?
Yes
Is destination a Spine or Host port?
Spine
Select 1 Spine, create a rule to forward packets of this flow to that spine
Host
Create rule to match packets from this flow to destination port
No
Flood to all other Hosts and 1 Spine switch
Spine
Is destination known?
Yes
Create rule to match packets from this flow to destination port
No
Flood

Implementation

Several utilities are used in the development of this project:

VirtualBox: For hosting the Mininet virtual machine image as well as various images used by GNS3.

Mininet: A tool to create a network of virtual hosts and OpenFlow switches. Each host has the capability to run built-in Linux utilities such as iperf.

Pox: Installed by default with the Mininet virtual machine image. Pox is an Openflow compatible software controller that supports the switches created by Mininet.

GNS3: A network emulation tool used to create network topologies using images of real network devices. Enables the design to be easily replicated in production networks.

CumulusVX: A free version of Cumulus Linux that is designed to familiarize users with the Cumulus Linux OS. Facilitates learning Linux networking as well as testing network designs before implementation. Version 3.7.10 was used here.

TinyCore: the hosts used in the GNS3 network were modified from the Linux Core virtual machine created by Radovan Brezula available on [8]. The Virtual Box network host 4.7.7 was used.

Source Code

topo.py: Written in Python, it is executed by Mininet to create the hosts and switches of the topology. This script takes 3 arguments: number of spine switches, number of leaf switches and number of hosts per leaf switch. The default values passed are 3, 3, and 2 respectively.

proj-pox.py: Written in Python, it is utilized by Pox to implement the network functionality described in the Design section of this document. This script takes just one argument, the number of spine switches. The default argument is 3.

Testing

test_1.py: The simplest test of all, this script just sends an ICMP message from the first host to every fourth host in the network. The aim of this test is just to check the number of flows generated and how they are stored in the switches.

test_2.py: This test script runs through all the nodes in the topology and checks connectivity of a client node with its respective server using the iperf utility. It generates files containing flows at each switch at two stages of the test: about halfway through the test and at the end of the test. It also generates files containing test results for each node.

test_3.py: This test script runs through all the nodes and runs a client script on a client node and a server script if a node is supposed to run as a server. It generates a file for each node containing output of the script that node ran. Similar to the previous test, flow details are also checked at two stages of the test.

client.py: This script is run on a client node of the second test. It opens a socket on the host node, connects with its respective server node and sends 500 packets of 1024 bytes each.

server.py: This script is run on a server node of the second test. It opens a listening socket on the node and every time a new connection is received, it spawns off a new thread to service that connection. This thread acknowledges each packet received from the client and shuts down once it stops receiving packets. The listening socket shuts down after a period of 10 s with no activity.

Results and Discussion

Only the scalability test was conducted on the ECMP architecture. This was because the design ran into hash collisions as mentioned in [4]. With multiple routes installed, Cumulus Linux hashes on a per flow basis on IP protocol, ingress interface, source IPv4 or IPv6 address, destination IPv4 or IPv6 address, and

source/destination TCP/UDP port [7]. Since there were only two leaf switches, all traffic between leaves would hash to the same VTEP destination IP address and would utilize the same link. Thus, even if all 8 hosts on a leaf were communicating to 8 different hosts on the other leaf, the traffic would only flow over 2 spine switches regardless of the amount of traffic. While this behaviour would be minimized in a large data center with many leaves, it is a well known problem that was replicated here.

The Mininet topology was tested against two types of data flows. In the first case, we consider a heavier flow generated by the iperf utility while testing a route's capacity. In the latter case we test the topology under a client/server paradigm where a client sends a large number of relatively small frames to a server.

a) In the first example, the script test_2.py builds a network in Mininet as per the topo.py script. test_2.py takes as arguments the number of servers in addition to the arguments of topo.py. It evenly intersperses servers among the clients. Each server host runs the iperf utility waiting for incoming requests. Each client tries to check the throughput capacity to its allotted server which generates traffic on the network.

b) In the second case, the script test_3.py builds a similar topology as above. However, each server runs a server.py script where it opens a listening socket and waits for incoming requests. Each client sends a number of fixed size packets to its associated server. Similar to the previous case, servers are evenly spaced amid the clients.

In all cases it was observed that the load balancing algorithm worked as intended. Each flow is identified by source and destination IP address, and source and destination port number. ARP messages comprise a separate flow. In test_1.py it is seen that 3 flows are generated by a single ICMP message and each flow is routed through a different spine switch. In the other tests, the flows are distributed roughly equally among all spine switches. It can also be seen that no flows are generated in the spine switches when all communication is between hosts connected to the same leaf switch.

The key takeaways from this project are an understanding of how SDN switches operate and how they can be managed by programming a POX controller. An appreciation of how the process of learning and forwarding in OpenFlow enabled switches is significantly different from regular L2/L3 switches was also developed. This implementation and simulation may be very different from our original aim and intention, but considering the extenuating circumstances and limited resources at our disposal, it is an effort to acquaint ourselves with the SDN paradigm for the future.

Related Work and References

[1] Cisco (2018). Cisco Validated Design: Campus LAN and Wireless LAN Design Guide [White paper]. Retrieved April 10, 2020 from Cisco: https://www.cisco.com/c/en/us/solutions/enterprise/design-zone-borderless-networks/design-guide-listing.html.

[2] Cisco (2020). Cisco Data Center Spine-and-Leaf Architecture: Design Overview [White paper]. Retrieved March 20, 2020 from Cisco: https://www.cisco.com/c/en/us/products/collateral/switches/nexus-7000-series-switches/white-paper-c11-737022.html.

[3] K. Bilal, S. U. Khan, J. Kolodziej, L. Zhang, K. Hayat, S. A. Madani, N. Min-Allah, L. Wang, and D. Chen, "A Comparative Study of Data Center Network Architectures," in *26th European Conference on Modeling and Simulation (ECMS)*, Koblenz, Germany, May 2012, pp. 526-532.

[4] Keqiang He, Eric Rozner, Kanak Agarwal, Wes Felter, John B. Carter, Aditya Akella: Presto: Edge-based Load Balancing for Fast Datacenter Networks. Computer Communication Review 45(5): 465-478 (2015).

[5] Narváez, Pablo. Cumulus Networks Layer-3 Leaf-Spine Fabric with EVPN as a Control Plane for VXLAN. Retrieved March 18, 2020 from the-bitmask: https://the-bitmask.com/2017/09/26/cumulusvx-layer-3-leaf-spine-fabric-with-vxlan-evpn/.

[6] Juniper (2019). EVPN User Guide [White paper]. Retrieved April 10, 2020 from Juniper: https://www.juniper.net/documentation/en_US/junos/information-products/pathway-pages/junos-sdn/evpn-vxlan.html.

[7] Cumulus. Cumulus Linux 4.1 User Guide: Equal Cost Multipath Load Sharing - Hardware ECMP [White paper]. Retrieved April 10, 2020 from Cumulus: https://docs.cumulusnetworks.com/cumulus-linux-41/Layer-3/Equal-Cost-Multipath-Load-Sharing-Hardware-ECMP/.

[8] Brezula, Radovan. Linux Core Virtual Appliances. Retrieved March 20, 2020 from Brezular: https://brezular.com/2013/09/17/linux-core-appliances-download/.