Arachne: Large Scale Data Center SDN Testing

Jamal Hadi Salim, Alexander Aring
Mojatatu Networks
Ottawa, Canada
hadi@mojatatu.com, aring@mojatatu.com

Abstract

Current state of the art in modern data center structure makes use of multi-stage switching known as a Clos Network. One advantage of a Clos network is it allows for modularity in scaling upgrades. A properly designed Clos network can be upgraded by adding more racks or PoDs (Point of Delivery) level without any need for re-arranging the existing connections.

Control of the datapath and other resources in a Clos Network is typically constructed using SDN (Software Defined Networking). A classical SDN setup includes one or more software daemons/agents that run on the control and resource path.

To test scale and robustness of a large SDN deployment requires a large investment in a lot of real switch and server equipment. Alternatively, one could reduce the amount of equipment needed by using a VM based setup such as [5]. For a scale where we need to test 10s of thousands of nodes, using VMs is still a very expensive proposition. Naturally our struggle led us to look at containers. We looked at alternative container-based approaches and concluded the complexity in adjusting them for our needs would be more work and larger maintenance effort.

This paper will describe *Arachne*, a basic SDN test tool, which uses basic Linux namespaces. A Linux bridge is created within a Linux net namespace container to emulate either a spine and leaf switch; and basic net namespaces are created to emulate end hosts residing within a data center rack.

Our focus is to test the scaling of the control-to-resource path domain and not necessarily the scale of datapath traffic.

Arachne takes as its input a Clos Network description. A dot file is generated by *Arachne* to describe the topology. The dot file is then consumed to construct a data center PoD using basic tools such as iproute2.

Keywords

Linux; Software Defined Networking; Namespaces; Containers; Data Center; Clos Networks; DOT; Graphviz; Bridge; Router; Virtual Machines; Intent Based Networking; L2; L3

Introduction

SDN is a system architecture that enables centralized control of datapath resources. Figure 1 shows a classical SDN setup. SDN control applications attach to controllers to monitor and/or impose policies on datapath resources via cluster controllers.

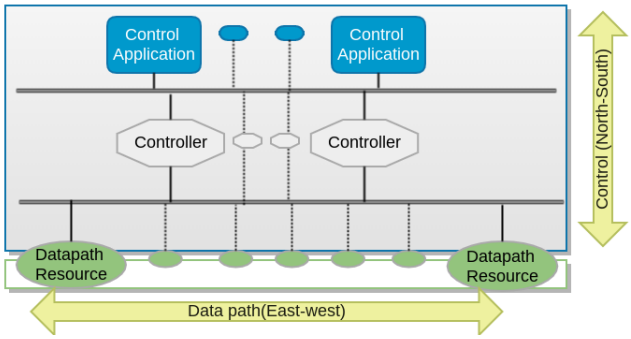

Figure 1: Classical SDN Architecture

As illustrated in figure 1, the datapath entities talk to each other on the east-to-west plane whereas the control and datapath entities talk to each other in the north-south direction.

On a large scale SDN setup, there are typically a magnitude or two more datapath entities than there are controllers; and a magnitude more applications than there are controllers. Each of the datapath and control entities run software agents which talk to each other relaying sensor information and policy actuation via a dedicated control network.

We set out to come up with a test environment where the System Under Test (SUT) is the control-to-resource infrastructure (north-south direction). We defined our goals as:

- Being able to introduce infrastructure variability so we can test with a focus on all or some of:
  - Control application testing
  - Controller testing
  - Datapath resources request-response and publish-subscribe testing
- Being able to test with setups that range from a handful to hundreds of thousands of nodes (applications, controllers and datapath)
- Being able to test scale, robustness under chaos, and basic system functionality
- Being able to achieve all the above cheaply (operational and capital costs)
- Ensuring we reuse or create open source and particularly Linux based infrastructure

- Ensuring the architecture allows for other (than our own) SDN approaches to be incorporated without a lot of effort

We are interested in the datapath only with respect to its northbound interface being a utility within the scope of our SUT. For this reason, while we care about traffic flowing between the different datapath entities (east-west) we do not strive to optimize that direction.

We named our tool *Arachne* after the figure from Greek mythology who was gifted in the art of weaving. *Arachne* is both a design and deployment tool.

This paper documents the trials and tribulations and choices made in our journey to create *Arachne*.

Deployment: Clos Network Infrastructure

SDN environments are commonly deployed on data center architectures; and currently the most popular data center deployment is based on Clos networks [4]. For this reason we chose to use the Clos network architecture as the infrastructure for meeting our goals.

Both Clos networks and SDN are a result of generations of work in the telephony world.

Clos networks were an influence of the telephony world which connected phone systems back in the 50s: if you imagine a host as a phone and the Ethernet switches as voice switches, then you can visualize how phone calls were switched from the caller to the callee.

The telephone world also greatly influenced SDN. Telephone networks introduced the concept of separating control signaling (SS7 [9]) from datapath voice as two separate networks back in the 70s.

With *Arachne* we are going to use switches and (deployment-dependent) hosts as datapath nodes. All data center equipment we have come across so far (switches or hosts) carry one or more management Ethernet port(s). We are going to use these management ports to deploy the control infrastructure.

A Clos network provides a multi-stage switching setup which is attractive for our tooling for two reasons:

- Its wide use in data centers implies we are closer to the reality of our real-world deployment
- The ability to scale the infrastructure from a small build to something extremely large was very appealing. More nodes can be added without disrupting a running Clos network setup. This means we can incrementally test different SUT scale levels without undoing a running test setup

Clos networks can be generalized to any odd number of switch stages; 3 and 5 stages are common. In our setup, in order to meet our scale goals we are also going to allow for a 7 stage Clos in the future.

3-Stage Clos: PoD

A basic 3-stage Clos network encapsulation is known as a PoD (Point of Deployment).

Within a PoD:

- A rack of hosts is connected to a leaf switch (Interchangeably referred to as a Top Of Rack (TOR) switch)
- Each leaf is connected to every spine in that PoD

Traffic sourced in a host in one rack needs to get through three switch hops/stages (hence the 3-stage convention) to get to a host on another rack within the PoD:

1. Local leaf to a spine
2. Spine to remote leaf
3. Remote leaf to remote host

Figure 2 shows a very basic PoD setup with 4 racks with each rack carrying a dozen hosts and a leaf switch.

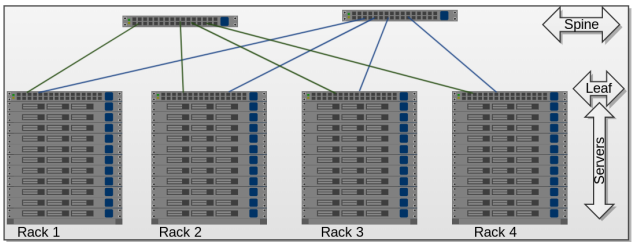

Figure 2: 3-Stage Clos Point of Delivery

From a scale/modularity and automation perspective, one can imagine wheeling in a new preconfigured rack of hosts with its leaf switch; connect the leaf switch to all spines in the PoD and power it on for immediate use.

5-stage Clos: Zone

A Clos network can further be scaled by inter-connecting PoDs; this would require a layer of switches above the spines to connect the PoDs to each other. We will call such a setup a *zone*. It should be noted that inter-PoD switches have a lot of other naming conventions in the industry such as *fabric*, *super-spine*, *core* etc. In our naming convention we will refer to these inter-PoD switches as *zone switches*.

In a 5-stage Clos network packets emanating from a host in one PoD have to take 5 hops to get to a host on a different PoD.

A typical setup of how zones are connected via the zone switches is similar to leaf to spine connections, i.e each spine switch on every PoD connects to all zone switches. Such a setup is illustrated in figure 3.

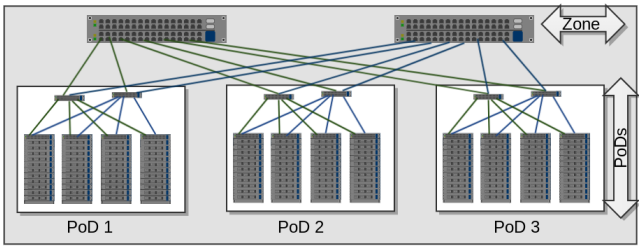

Figure 3: 5-Stage Clos

From a scale-up/modularity and automation perspective, one can imagine trucking in a brand new pre-configured PoD from the factory and connect the PoDs spine switches to zone switches, power up things and voila! Things just work.

Clos Design Constraints

How does one decide how many hosts sit on a rack? Or how many racks per PoD? Or how many PoDs per zone? Or whether they need a 7-stage Clos network?

The selection of number of hosts within a rack, number of racks connecting to a set of spines or the number of spines within a PoD are decided based on many factors. We list some of the prominent ones we have come across:

- Space available to house the equipment
- Power consumption required to operate the equipment. A 10 rack setup requires 1-4 MW according to SGI [17]
- Cooling needs of the equipment and capacity of temperature and airflow controls
- Limitations of available costed hardware. For example if the leaf switch can only do 48x10G + 4x40G ports, you are restricted to a maximum of 48 hosts and 4 spines that can connect to such a rack
- Limitations of available bandwidth per port on a host or switch defines what switches get used for leaf or spine

To re-iterate: our primary goal is to test the control-datapath plane; however, being able to exercise constraints of different real-world setups when designing is a pragmatic secondary goal which would allow us for example to quickly churn and emulate LinkedIn's Altair data center design [12] and test our SDN solution on it. We plan to consider this constraint setting feature for upcoming *Arachne* releases.

Deployment Approach Selection

The immediate thought that comes to mind when contemplating a SUT deployment is to build a physical setup. Given our desire to build 10s-100s of thousands of datapath entities, we very quickly figured out that the operational and capital costs were beyond our means. We are not going to be able to afford a LinkedIn or Facebook layout. So we crossed out this possibility.

Naturally this leads one to an option to use VMs to simulate all the components. An excellent fit for a VM emulating a Linux switch is the Cumulus VX [5].

While the Cumulus VX worked very well, we soon ran into challenges. On a small x86 device (i5, NUC) with 8G of RAM we were unable to run more than 12 VMs effectively (single PoD with 2 spines and 4 racks each with one host). For our requirement of achieving 10s-100s of thousands of nodes, the cost would have been too high for us to bear (in terms of hosts we would have to acquire). For this reason we also abandoned this VM approach.

It was clear we had to go to the path of using containers to achieve a large scale setup.

Our immediate thought was to use Docker [6] since it is integrated into our CI build and has a rich set of tooling.

We quickly reached a conclusion it was too heavyweight for our goals. All we needed was to be able to create emulation switches and hosts and interconnect them; Docker required a lot of extra infrastructure of daemons and setups that would consume more of our limited compute resources.

We next looked at Mininet [14] which looked interesting because it was lightweight. The major obstacle was Mininet's heavy bias towards OVS and OpenFlow which contradicts our intention to allow other SDN approaches to be easily added (and ours in particular). Mininet is also a lot more complex than we needed because it is designed to be multi-purpose; e.g. it allows arbitrary topologies, while we were only interested in Clos.

In the end we decided we needed to create our own tool and so *Arachne* was born.

High Level Design Overview

We decided early on that it was imperative to make the *Arachne* interface friendly to humans. To address this challenge, we provided two user facing features:

1. A simple user configuration to describe the Clos network. Anyone with a basic knowledge of Clos networks should be able to describe what they intend their network to look like
2. A visual output of the created network so the user can validate that it was in fact the infrastructure layout they intended

Figure 4 illustrates the workflow an *Arachne* user will have to go through.

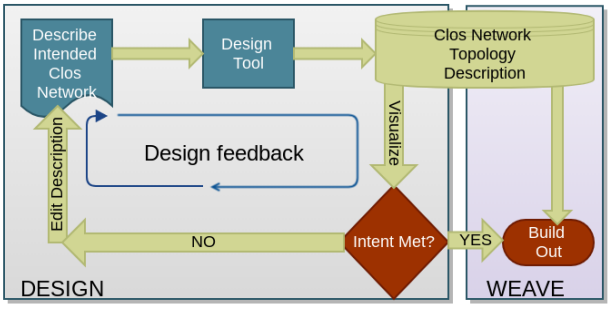

Figure 4: Arachne Workflow

There are two phases to the *Arachne* workflow: A **design phase** and a **weave phase**.

Design Phase

In the design phase the user creates the Clos network of choice.

1. The user defines, via the *Arachne* console interface, what they wish their Clos network to be. *Arachne* takes the defined user description and crunches out the Clos network graph (which the user can save)
2. The user requests, via the *Arachne* console interface, a visualization of the resulting network. If the user is unhappy with the results they can edit the description and redo the design

This feedback loop can go on as many times as the user wishes.

Network Description Semantics As mentioned earlier, we wanted to have a simple way for a human to describe their Clos networks without needing to understand any of the involved intricacies.

We introduce five parameters that a user needs to provide to *Arachne* to describe their Clos network. All parameters are optional if the user chooses to use the default prescribed design.

Number of zones Needed if more than one PoD is present. At the moment of writing only one zone is supported by *Arachne*

Number of PoDs By default 1 PoD is assumed

Number of spines in a PoD By default 2 spines are assumed

Number of racks per PoD By default 4 racks are assumed

Number of hosts per rack By default one host per rack is assumed. At the moment of writing, *Arachne* supports a single leaf switch per rack

Most use cases assume all PoDs are created equal (same number of racks and hosts per rack) and so the above parameters are sufficient; however, such a setup may be insufficient for other multi-PoD setups. As an example, a compute PoD may have 40 hosts per rack but a storage PoD may have 16 hosts per rack for the same rack real estate. The *Arachne* design tool allows the user to be more specific at a per-PoD level.

Topology Description Instead of inventing our own topology language, or using Mininet's variant we decided to use the Dot graph language [7]. Our choice was inspired from our experience with Cumulus VX which uses the Dot language. Dot is open and widely used and has a lot of tooling around it including many programming language bindings.

Weave Phase

Once the user is satisfied with the results they instruct *Arachne* (using the console interface) to go ahead and build/deploy the setup.

Addressing Convention We decided to adopt another telephony feature in creating the addresses of the nodes loosely based on the numbering plan described in E.164 [10].

Arachne uses this geographical addressing scheme for both MAC and IP addresses.

Based on E.164, as most phone user would recognize, telephone numbers take the form:

Country code	Area code	Subscriber number

Figure 5: E.164 Telephone Number Scheme

We map:

Country code An ID we refer as **zone ID**

Area code An ID we refer to as **PoD ID**

Subscriber number An number which depends on the type of node (spine or leaf switch, host etc) we give it a semantic as described

Why Geographic Addressing? As mentioned earlier, we are striving for our setup to handle hundreds of thousands of nodes. This means there will be a large number of packets flying all over the ether: imagine running tcpdump on some arbitrary port. If it is immediately apparent what the geographical source or destination of a packet is, it is immensely helpful for debugging; security features such as checking whether a packet should be seen at certain locations become much easier to implement.

It should be noted also that IPv6 has adopted hierarchical addressing and routing which allows identifying nodes by geographic location.

We believe that a geographical addressing setup facilitates simplification in infrastructure management automation (as we hope to convince you in this paper). We can choose to either go with a basic L2 or L3 setup and pre-compute the setup.

For L3, although we allow the *Arachne* user to run dynamic routing protocols, we feel it is sufficient to setup static routing and static provisioned policy. Moreover, we feel geographic addressing allows us to reduce the number of routes and ACL policies. There are cases where dynamic routing features may be needed in particular in hardware based deployments; as an example when an upstream link becomes defective resulting in a path/route that a downstream node uses to become unavailable. Our view is we can achieve **much faster convergence** with an SDN approach that is aware of the topology than by using a dynamic routing protocol. While this is an opinion we hold, we are not dogmatic about whether the user wishes to use dynamic routing, SDN or neither. *Arachne* will support all those approaches.

MAC Addressing By default we construct the MAC address according a special pattern.

10	ZID	PID	R	D	0

Figure 6: EUI-48 OUI Scheme

The first 3 bytes of the EUI-48 MAC address (known as OUI) is constructed as shown in figure 6. The bits has the following meaning, we use "10" as first octet to signal a locally administrated address. All other fields has the following meanings:

10 8 bits for the constant "10", meaning "locally administered address"

ZID 3 bits for zone ID implying we can have a maximum of 8 zones

PID 6 bits for PoD ID implying we can have a maximum of 64 PoDs

R 2 bits for role identity type. Further explanation follows in this section

D 2 bits for direction. This bit describes the direction in orientation of the Clos Network, if the link goes north or south

0 3 bits for the constant "0"

Role ID number	Port ID

Figure 7: EUI-48 NIC Scheme

The last 3 bytes are constructed according to figure 7. Each field is part of the geographic address scheme and means the following:

RID 12 bits for role identity number. This indicates the ID number which belongs to it

Port ID 12 bits for port ID. This means a given node can have a maximum of 4096 ports

The Role ID type identifies whether the address is located on a host, spine switch, leaf switch or zone switch.

The following Role IDs are defined:

0 Reserved for future use

1 Leaf node identification

2 Spine node identification

3 Zone node identification

Host IPv4 Addressing By default, the host IPv4 addressing is constructed by the following scheme:

10	ZID	PID	RID	HID

Figure 8: Host IPv4 Addressing Scheme

10 A prefix-octet with value "10" occupies the first byte

ZID The zone ID occupies the next 3 bits

PID The PoD ID occupies the next 6 bits

RID A 4 bit rack ID identifies a rack within a PoD. This implies we can have a maximum of 15 racks per PoD

HID The remainder 11 bits identify the host on a rack. This implies up to 2045 hosts can be supported within a rack

Leaf Switch IPv4 Addressing We only give IP addresses to leaf nodes when *Arachne* runs in L3 mode and operates as router. More later when we discuss L3 mode.

One IP address is attached to the loopback ("lo") interface. This leaf address is used by a neighbor (host/spine) as a next hop IP address for L3 forwarding. Given Linux IPv4 addressing is based on the weak host model (RFC 1122 [1]), the IPv4 address attached to lo is seen by any ARP request from any of the leaf ports (spine or host attached).

Leaf routers use IPv4 Link-Local Addresses (RFC 3927 [3]) for their addresses.

169.254	1	PID	RID	L-ID	1

Figure 9: Leaf IPv4 Link-Local Address Scheme

169.254 The first 16 bits are occupied by octets "169.254"

1 2 bits are occupied by the Role identification. A leaf always has value 1

PID 6 bits identify the PoD

RID A 4 bit rack ID identifies a rack within a PoD. This implies we can have a maximum of 15 racks per PoD

LID 2 bits represent the leaf ID. This means we can have up to a maximum of 4 leaf routers per rack. At the moment *Arachne* supports only one leaf so this number is hard-coded to 0

1 2 bits are used for An IP address ID (that is attached to "lo"). At the moment this value is hard coded to 1

Spine Switch IPv4 Addressing We only give IP addresses to spine routers when *Arachne* runs in L3 mode. One spine IP address is attached to the loopback ("lo"). A spine address is used by a neighbor (leaf/spine) as a next hop IP address for L3 forwarding. More below when we discuss L3 mode.

Spine routers use IPv4 Link-Local Addresses (RFC 3927) for their addresses.

169.254	2	PID	SID	1

Figure 10: Spine IPv4 Link-Local Address Scheme

169.254 The first 16 bits are occupied by octets "169.254"

2 2 bits are occupied by the Role identification. A spine always has value 2

PID 6 bits identify the PoD

SID A 4 bit spine ID identifies a spine router within a PoD. This implies we can have a maximum of 15 spines per PoD

1 4 bits are used for An IP address ID. At the moment, a value of "1" is hard coded

Zone Switch IPv4 Addressing We only give IP addresses to zone routers when *Arachne* runs in L3 mode.

One zone IP address is attached to the loopback ("lo"). A zone address is used by a neighbor (a spine within a PoD or a core router/switch) as a next hop IP address for L3 forwarding. More below when we discuss L3 mode.

Zone switches use IPv4 Link-Local Addresses (RFC 3927) for their addresses.

169.254	3	ZID	ZSID	1

Figure 11: Zone IPv4 Link-Local Address Scheme

169.254 The first 16 bits are occupied by octets "169.254"

3 2 bits are occupied by the Role identification. A zone router always has value 3

ZID 3 bits are occupied by the zone ID

ZSID A 6 bit zone router ID identifies a zone router within a zone. This implies we can have a maximum of 64 zone routers per zone

1 5 bits are used for an IP address ID. At the moment, a value of "1" is hard coded

Naming Convention *Arachne* uses geographic encoded names for all nodes. These names are used to assign a name to each iproute2 net namespace.

```
H<HID>_R<RID>_P<PID>_Z<ZID>
```

Listing 1: Host Node Name Convention

Listing 1 shows the naming convention for host namespaces. The following table show the replacement parts as described in the naming convention which is also follows the geographic addressing scheme.

HID Host/Subscriber ID (Range 3..2044)

RID Rack ID (Range 1..15)

PID PoD ID (Range 1..63)

ZID Zone ID (Range 1..7)

Leaf node names follow the convention:

```
L<LID>_R<RID>_P<PID>_Z<ZID>
```

Listing 2: Leaf Node Name Convention

LID Leaf ID (Range 1..3)

RID Rack ID (Range 1..15)

PID PoD ID (Range 1..63)

ZID Zone ID (Range 1..7)

The spine node names are constructed as the following scheme:

```
S<SID>_P<PID>_Z<ZID>
```

Listing 3: Spine Node Name Convention

SID Spine ID (Range 1..15)

PID PoD ID (Range 1..63)

ZID Zone ID (Range 1..7)

The zone node names are constructed as the following scheme:

```
ZS<SID>_Z<ZID>
```

Listing 4: Zone Node Name Convention

ZS Zone switch/router ID (Range 1..63)

ZID Zone ID (Range 1..7)

Network Build Approach *Arachne* consumes the generated dot files from the design phase and proceeds to weave the network.

Each node is built as a network container. Figure 12 shows how a single PoD with 3 racks and one spine switch would be constructed.

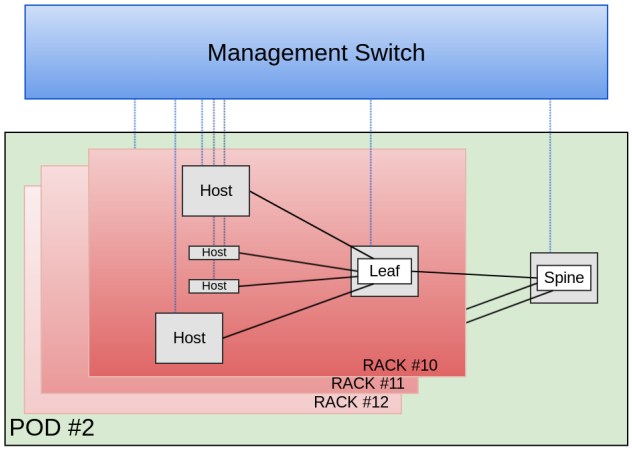

Figure 12: Arachne Single Container Build

Each node, regardless of its type, is created as a container with unshared network and UTS namespaces using iproute2. On a single host all containers share the host filesystem.

All ports are of type *veth*. Veth is ideal because of its dual port nature. It can be used as a virtual *cable* between pairs of switches or between switches and hosts, with each end of the veth being placed in a different network namespace.

Both leaf and spine switch nodes contain a Linux bridge interface.

- All nodes have a management veth endpoint inside their respective namespace; the other end of the veth is attached to a bridge in the container parent which is used as a management switch. By default, *Arachne* has a DHCP daemon connected to this switch; all containers get their management IP addresses via this daemon
- Host to leaf switch connections are constructed with a veth cable with one end residing inside the host container and the other inside the leaf container attached to the resident Linux bridge (the leaf switch is the master of the veth endpoint)
- Leaf to spine connections are likewise veth based. The leaf side is attached to the Linux bridge residing in the leaf container and the spine side is attached to the Linux bridge in the spine container

Arachne has two components in its weaving approach: A front end on which the user interacts and a back-end which does the actual network creation.

Figure 13 shows how these components come together.

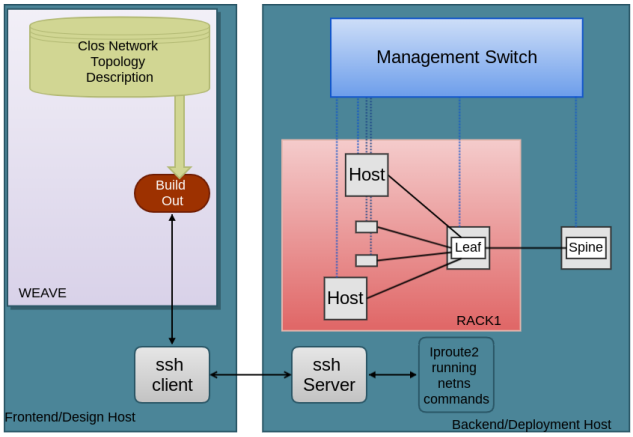

Figure 13: Arachne Weaving

The front- and back-ends communicate over ssh. This means that the network design can be deployed on any remote node which has an ssh host.

Another approach to realize the deployment at the back-end side is to use Ansible [16]. We tried Ansible and came across issues. There was no easy way to allow for big changes dynamically. Ansible depends on creating provisioning steps using playbooks. The playbook inventories are static in nature and *Arachne's* intended use is more dynamic therefore needing more flexibility with inventories. Debuggability of the playbooks was a bit of a challenge as well. Given what we needed was easily implemented using only ssh whereas Ansible added more dependencies, we gave up on Ansible.

Back-End Commands

Arachne provides a console interface to the user to manipulate data center parameters as shown in the section "Network Description Semantics". The weave phase initiates back-end commands when the user types **weave**. In the background *Arachne* generates a bash script which will be executed by the back-end machine.

```
...
ip link add swp5 type veth peer name _swp2
ip link set netns S2_P2_Z1 dev swp5
ip link set _swp2 name swp2
ip link set netns L1_R3_P2_Z1 dev swp2
...
```

Listing 5: Arachne Back-End Weave

Listing 5 shows part of the generated script to create a veth cable connection from interface swp5 at S2_P2_Z1 to interface swp2 at L1_R3_P2_Z1. The command **unweave** tries to cleanup every created net namespace.

Listing 6 shows how *Arachne* deletes all generated networking namespaces.

```
...
ip netns pids L1_R4_P1_Z1 | xargs kill
ip netns del L1_R4_P1_Z1
...
```

Listing 6: Arachne Back-End Unweave

Network Infrastructure Choices

There are several Clos network connectivity modes that *Arachne* supports. We discuss them in this section.

Layer 2 Network

A layer 2 network is very simple to setup and automate. *Arachne* defaults to such a setup. Only hosts are assigned IP addresses. So the whole thing looks like one Big Freaking Switch (BFS) i.e a single broadcast domain with hosts hanging off it.

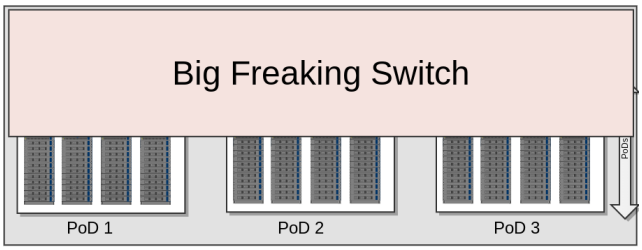

Figure 14: Arachne L2 Mode

A layer 2 network has drawbacks.

1. It is possible to have broadcast loops
2. Even if you avoid broadcast loops, an ARP/ND from a host on one PoD will show up at all hosts on all PoDs causing unnecessary traffic
3. Given the BFS nature, every switch's FDB/MAC tables will have an entry for every host. This part is undesirable if the FDB tables are not large enough

To address issue 1, *Arachne* enables STP on all the switches. It could be argued on one hand that STP results in less of the available cluster link capacity being utilized; on the other hand, it could be argued that STP provides path resiliency. Our need for STP is mainly to deal with loops.

For small scale setup, we simply ignore issues 2 and 3; we believe that the offered convenience of simplicity is a reasonable tradeoff. Recall: our goal is to test the north-south frontier and so we are not overly focussed on an optimal east-west setup.

When deploying using real hardware switches, we do recognize L2 equipment has limited FDB/MAC table sizes - and the cost of such hardware tends to go up as these table sizes go up; moreover, once these tables fill up, packets get broadcast and consume more wire resources.

Although such a limit is a red herring as far as software based L2 is concerned, we expect *Arachne* users to have desire to emulate deployments closer to what the hardware offers. So we propose to address drawbacks 2 and 3.

Layer 3 Network

L3 overcomes all the issues mentioned as shortcomings in L2, but it comes at the cost of complexity of setup.

The most predominant solutions we have seen in data centers is to set up dynamic routing on all spines and leafs. Operators run I/EBGP or OSPF, in our opinion, mostly because they are already skilled craftsmen of such routing protocols. While *Arachne* allows for such setup, we do not believe such complexity is necessary given our geographic addressing setup.

We can get away with pre-configured static routing which takes advantage of the geographic addressing.

The reader is reminded to review the "Why Geographic Addressing?" section and consult the scheme shown in Figure 15. Everything we are describing in the next section is easy to automate and pre-provision (because we use geographical addressing).

In this section, all of the leaf/spine/zone nodes run as routers i.e no Linux bridge is configured. All packets are forwarded using FIB table.

L3 Ingress traffic IP packets destined for a specific zone are selected based on a 10.x/11 prefix as illustrated by the ingress direction (north to south) of figure 15. Essentially, the prefix selects a zone. The next hop in a specific zone's direction is selected from one amongst the zone routers within that zone. This is achieved by pre-provisioning a default ECMP route with next hops pointing to all possible zone routers (in the 169.254.x.y address space using the zone router addressing as described in figure 11. It should be noted all the zone routers in the next hop list will have a "1" in their least significant nibble.

From a specific zone router downstream, a /17 mask is used to select a PoD as well as a spine router within the PoD. Since there are multiple spines per PoD, again ECMP is used to select one of the spines within a target PoD as the next hop (in the 169.254.x.y address space as described in figure 10).

From a specific spine within a PoD, a 10.x/21 address prefixing is used to forward to a specific rack. Given we currently support a single leaf router per rack, the next hop points to that single leaf residing in the target rack. The next hop IP address is in the 169.254.x.y address space with the last nibble of value 0x1 (leaf ID of "0" and IP address ID of "1" as described earlier)

Once the packet hits the leaf router, it gets routed to the host using a host link route.

L3 Egress traffic Each host has a default gateway pointing to the leaf router. The leaf is configured so as to proxy ARP on the host side. This helps us reduce the complexity of the leaf router setup. All ARPs are responded to by the leaf.

The leaf is provisioned with:

- Host routes for all its attached hosts
- Default ECMP to point to all the spines north of it

When a host within a rack tries to communicate with another within the same rack, the leaf does the IP forwarding locally. When the host tries to communicate with a host in a different rack or PoD, the leaf forwards to one the spines within the PoD per ECMP rules.

On the spine, there are two possibilities:

- If the destination IP was within the PoD but a different rack then a /21 route will select a leaf router within the PoD (as was described in the "L3 Ingress Traffic" section)
- Otherwise, by default one of the upstream zone routers are selected based on ECMP. IOW, the spine has a default ECMP route with the next hop pointing to all of the zone routers it is connected to

Once a packet is received from a spine on the zone router, there are two possible outcomes:

- If the destination IP is within the same zone, then a destination PoD is selected and the packet is forwarded to a specific PoDs spine router based on ECMP as described earlier (caveat: this is not a default ECMP route)
- Otherwise, by default a zone is selected based on /11 mask and the next hop is programmed to be one of the core routers

Trials And Tribulations

We ran into several challenges in the process of creating *Arachne*. There is no doubt there are more to come as *Arachne* evolves. We summarize several issues that stood out for us below.

Tooling Challenges

The *iproute2* netns utility, which we use to create all containers, was our first challenge. Some user space applications (e.g. various DHCP clients/servers, LLDPd [13] etc.) use *gethostname()* and exchange hostnames in their messaging. It means we require that containers have hostnames. *Iproute2* netns only isolates network namespaces. The approach of writing hostnames within a container results in an over-write which is visible in all other containers as well as the host. To achieve unique hostnames, we patched *iproute2* to support isolation of the UTS namespace. Refer to the section "Naming Convention" on how we named the different containers.

Veth interface pairs are created in one namespace and then one endpoint must be moved to another namespace. At first we tried to create long names for the ports to reflect whether they were in a spine/leaf/host etc. But we quickly ran into the limitation of "IFNAMSIZ" being too short so we decided to use shorter names which are still reflective of the port location. For example, switch port 1 is named "swp1". We quickly ran into a namespace collision problem. Let's say in both namespaces we have the ports being connected called "swp1". It is not possible to create a veth in the initial namespace and call both endpoints "swp1". To solve this issue we used an intermediate name for the remote endpoint. Once migrated to the other namespace we could rename it. Listing 5 illustrates an example.

As mentioned earlier, all container management ports are connected to a management bridge that resides on the host.

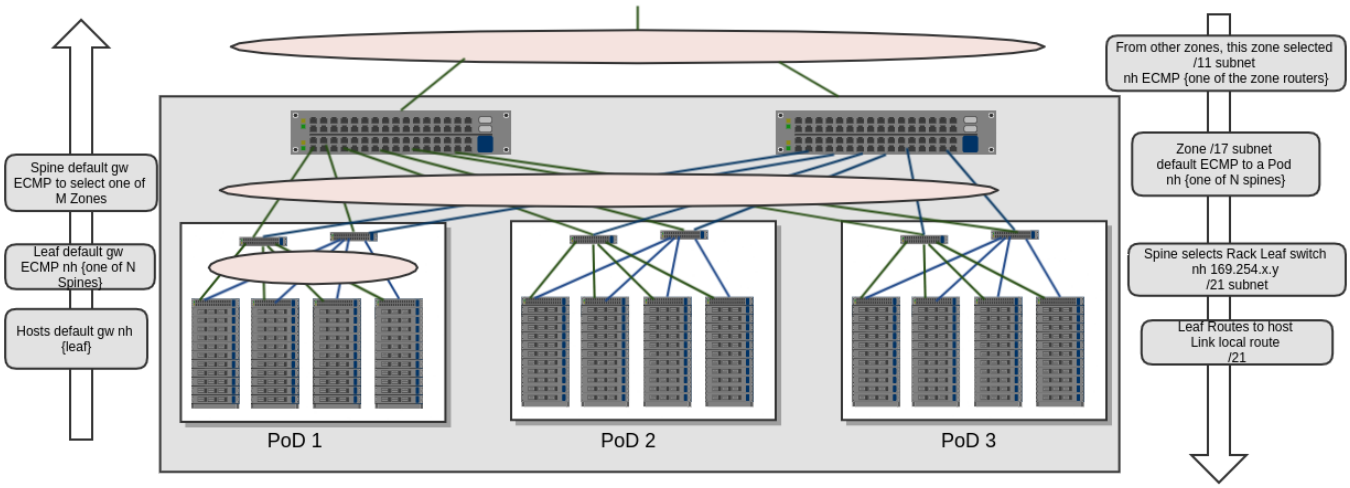

Figure 15: Arachne L3 Mode

We chose dnsmasq [18] to issue DHCP management addresses to the containers. A DHCP client [11] running inside a container will then issue a request for an IP address. This worked well until we integrated the ability to start applications (such as the SDN agent) inside the container. Because we not have a real init/systemd setup inside the containers, we ran into challenges when the application needed to bind to an IP address - but the request for the address was after the application was started. We resolved the issue by using DHCP hooks to run the application that needed to bind to an IP address.

IPv6 stateless autoconfiguration sends unsolicited ndisc messages as soon as the interfaces are brought up. Given we wanted veth to operate as a cable i.e only forwarding traffic from/to the bridge, this caused us some grief (in particular before we discovered the failure of veth to participate in STP). For now we disable IPv6 on these interfaces.

Python seems to be going through some transition from python2 to 3. Depending on which distro you use there is a lack of backward and often forward compatibility between version 3 and 2 of the system. We settled on python3. The backward/forward compatibility issue caused us problems when testing on different distros; there were times when we corrupted a machines python installation while installing dependencies. Ultimately, we ended creating python binaries using pyinstaller [15].

Bridge Issues

One of the first fun things we came across was L2 broadcast loops. Our natural reaction was to turn on STP on the bridges. To our surprise, all bridge ports using veth netdev came up in the forwarding state and yet broadcast loops persisted. Upon further investigation we found that the kernel was obstructing LLC packets going across ports that are not in init_net. Patching the kernel resolved this issue.

Another bridge issue we encountered had to do with the management bridge. We need to be able to reach the containers from the host based on the management addresses. This became a problem when trying to access containers from the host while *Arachne* was still in its weaving phase. Such a situation will occur when we are initializing SDN agents inside the container as part of the container creation. By default the Linux bridge uses the lowest MAC address of any of its enslaved interfaces for packets "sourced" via the bridge on the host towards the containers. When a newly created veth has a lower MAC, as soon as it is attached, it becomes inherited by bridge as its source MAC for any packets emanating out of the host towards the containers. The neighbor tables inside the containers communicating with the host become confused; and until those neighbor caches are re-updated (which could take many seconds) connectivity is stalled. We solved this issue by assigning the bridges MAC address when the bridge is created. The bridge then uses this MAC address as its source regardless of any other MAC addresses that might be added to it.

Scaling Issues

Amusingly, the first scale issue we ran into was in regards to the process of issuing management IP addresses to containers via DHCP. As we increased the size of the Clos network, it became very clear that the DHCP protocol exchange was slowing the process of weaving. Each DHCP address assignment required up to 4 messages to issue a management address to the container. At the point when we started creating hundreds of containers, this serial process became unbearable. We solved this problem by abandoning DHCP altogether for management and creating geographical static IP addressing in the management space (similar to the approach taken for east-west IP addresses). We ended up using a similar geographical address handling scheme. We use the address prefix 25.x.x.x which is assigned to "UK Ministry of Defence" [8]. We used this prefix because there are no other /8 prefixes available for private use and the "UK Ministry of Defence" doesn't use this address space at all. A report from Oct. 28 2015 indicates the UK wants to sell these unused addresses [2].

The second issue also was manifested by the management

bridge. The Linux bridge only allows 1024 ports to be attached to it.

We fixed the issue by increasing the bridge port limit hard coded in the Linux kernel. Unfortunately the Linux bridge code uses static data structures to store the attached ports. An additional module parameter or dynamically allocating a variable-sized data structure can fix this issue.

The third issue was again manifested by the management bridge. As the number of management IP addresses connecting to the host went up, given the host to container to host communication, we ran into issues of ARP table exhaustion on the host. We fixed this issue by tweaking the different ARP garbage collection parameters.

In one of our setups we ran a daemon (LLDP) inside each container that created a file descriptor. As indicated earlier, the filesystem was shared between the host and all containers. As the number of containers went up, we hit the upper limit on the number of open fds. We resolved this issue by increasing the system fd limit.

Conclusions And Future Work

In this document we described our end goals for a large SDN test infrastructure. We explained how our investigation eventually led us to conclude we needed to implement a new tool: *Arachne*. We explained the rationale behind the design decisions we took at cross roads in our journey, and some of the potholes that confronted us along the roads we took. We journeyed on by changing direction at times (eg. giving up on DHCP for example) and in some cases we patched the potholes (eg. kernel and iproute2) so we could continue our travel.

It is our belief that, on a single medium to higher end physical/server machine, we will be able to run tens of thousands of these light weight container nodes (and therefore build a very large scale network emulation). Connecting more than one of these physical machines should allows us to experiment on a very large scale network cluster. We are hoping to experiment on such large scale setups and report back on our experiences.

It should be noted that in our testing scenarios, due to the fact that we run in a software only environment, we could have settled on having a single PoD or maybe a single zone and it would have adequately served our purpose in providing a large number of nodes. But as we stated earlier we want to be closer to what a real physical network looks like. Physical switches have a limited number of ports (unlike the Linux bridge) and limited bandwidth capacity. Towards that end, we plan to:

- Add support to allow the user to design a 7-stage Clos Network
- Use switch-constraint templates i.e. allowing the network design to be able to specify constraints on the different switches. For example, a switch by vendor foo may be constrained to have 24x100G + 48x25G ports and a switch by vendor bar has a limit of 48x50G ports which can be split into 10G or 25G ports using octopus cables etc. The designer can then specify which switches to use for spines/leafs/servers and *Arachne* will do some basic sanity checking

We plan to introduce some variant of chaos monkey support. The monkey could randomly or deterministically manipulate links to bring them up/down, introduce latency, corrupt packets, kill containers, etc. We are going to focus on both east-west as well as north-south frontiers for the chaos monkey work.

At the moment our addressing scheme is purely IPv4. We plan to add support for both IPv6 and a hybrid IPv4/6.

In order to improve east-west traffic performance we plan to add support for bridge offloading on modern NICs and use real physical switches when present.

Last but not least, we plan to open source *Arachne*.

References

[1] Braden, R. 1989. RFC 1122 Requirements for Internet Hosts - Communication Layers.

[2] Business Insider UK. The Ministry of Defence is sitting on 38.5 million worth of unused IP addresses. http://uk.businessinsider.com/government-slow-to-embrace-ipv6-2015-10. [Online; accessed 18-October-2017].

[3] Cheshire, D. S. D.; Ph.D., D. B. D. A.; and Guttman, E. 2005. Dynamic Configuration of IPv4 Link-Local Addresses. RFC 3927.

[4] Clos, C. 1953. A study of non-blocking switching networks. *The Bell System Technical Journal* 32(2):406–424.

[5] Cumulus Networks. Cumulus VX. https://cumulusnetworks.com/products/cumulus-vx/. [Online; accessed 17-October-2017].

[6] Docker. Build, Ship, and Run Any App, Anywhere. https://www.docker.com/. [Online; accessed 17-October-2017].

[7] Graphviz. The DOT Language. http://www.graphviz.org/content/dot-language. [Online; accessed 17-October-2017].

[8] IANA. IPv4 Address Space Registry. https://www.iana.org/assignments/ipv4-address-space/ipv4-address-space.xhtml. [Online; accessed 18-October-2017].

[9] International Telecommunication Union. 1993. Introduction to CCITT Signalling System No. 7 - Recommendation Q.700. http://www.itu.int/rec/T-REC-Q.700-199303-I/en.

[10] International Telecommunication Union. 2010. The international public telecommunication numbering plan - Recommendation E.164. http://www.itu.int/rec/T-REC-E.164-201011-I/en.

[11] Internet Systems Consortium. Dynamic Host Configuration Protocol for connection to an IP network. https://www.isc.org/downloads/dhcp/. [Online; accessed 19-October-2017].

[12] LinkedIn. Project Altair - The Evolution of LinkedIns Data Center Network.

https://engineering.linkedin.com/blog/2016/03/project-altair–the-evolution-of-linkedins-data-center-network. [Online; accessed 17-October-2017].

[13] LLDPd. Implementation of IEEE 802.1ab (LLDP). https://vincentbernat.github.io/lldpd/. [Online; accessed 18-October-2017].

[14] Mininet. An Instant Virtual Network on your Laptop (or other PC). http://mininet.org/. [Online; accessed 17-October-2017].

[15] PyInstaller. Put Python programs into a stand-alone executables. http://www.pyinstaller.org/. [Online; accessed 18-October-2017].

[16] Red Hat. Ansible is Simple IT Automation. https://www.ansible.com/. [Online; accessed 18-October-2017].

[17] Silicon Graphics International Corp. HPE SGI 8600 System - Supercomputer Technology & Architecture. https://www.hpe.com/us/en/servers/hpc-server-sgi8600.html. [Online; accessed 17-October-2017].

[18] Simon Kelley. Dnsmasq. http://www.thekelleys.org.uk/dnsmasq/doc.html. [Online; accessed 19-October-2017].