PortLand: A Scalable Fault-Tolerant Layer 2 Data Center Network Fabric

Radhika Niranjan Mysore, Andreas Pamboris, Nathan Farrington, Nelson Huang, Pardis Miri, Sivasankar Radhakrishnan, Vikram Subramanya, and Amin Vahdat
Department of Computer Science and Engineering
University of California San Diego
{radhika, apambori, farrington, nhuang, smiri, sivasankar, vikram.s3, vahdat}@cs.ucsd.edu

ABSTRACT

This paper considers the requirements for a scalable, easily manageable, fault-tolerant, and efficient data center network fabric. Trends in multi-core processors, end-host virtualization, and commodities of scale are pointing to future single-site data centers with millions of virtual end points. Existing layer 2 and layer 3 network protocols face some combination of limitations in such a setting: lack of scalability, difficult management, inflexible communication, or limited support for virtual machine migration. To some extent, these limitations may be inherent for Ethernet/IP style protocols when trying to support arbitrary topologies. We observe that data center networks are often managed as a single logical network fabric with a known baseline topology and growth model. We leverage this observation in the design and implementation of PortLand, a scalable, fault tolerant layer 2 routing and forwarding protocol for data center environments. Through our implementation and evaluation, we show that PortLand holds promise for supporting a "plug-and-play" large-scale, data center network.

Categories and Subject Descriptors

C.2.1 [**Network Architecture and Design**]: Network communications; C.2.2 [**Network Protocols**]: Routing protocols

General Terms

Algorithms, Design, Performance, Management, Reliability

Keywords

Data center network fabric, Layer 2 routing in data centers

1. INTRODUCTION

There is an increasing trend toward migrating applications, computation and storage into data centers spread across the Internet. Benefits from commodities of scale are leading to the emergence of "mega data centers" hosting applications running on tens of thousands of servers [3]. For instance, a web search request may access an inverted index spread across 1,000+ servers, and data storage and analysis applications may interactively process petabytes of information stored on thousands of machines. There are significant application networking requirements across all these cases.

In the future, a substantial portion of Internet communication will take place within data center networks. These networks tend to be highly engineered, with a number of common design elements. And yet, the routing, forwarding, and management protocols that we run in data centers were designed for the general LAN setting and are proving inadequate along a number of dimensions. Consider a data center with 100,000 servers, each hosting 32 virtual machines (VMs). This translates to a total of three million IP and MAC addresses in the data center. Assuming one switch is required for every 25 physical hosts and accounting for interior nodes, the topology would consist of 8,000 switches.

Current network protocols impose significant management overhead at this scale. For example, an end host's IP address may be determined by its directly-connected physical switch and appropriately synchronized with replicated DHCP servers. VLANs may provide some naming flexibility across switch boundaries but introduce their own configuration and resource allocation overheads. Ideally, data center network architects and administrators would have "plug-and-play" deployment for switches. Consider some of the requirements for such a future scenario:

- **R1.** Any VM may migrate to any physical machine. Migrating VMs should not have to change their IP addresses as doing so will break pre-existing TCP connections and application-level state.
- **R2.** An administrator should not need to configure any switch before deployment.
- **R3.** Any end host should be able to efficiently communicate with any other end host in the data center along any of the available physical communication paths.
- **R4.** There should be no forwarding loops.
- **R5.** Failures will be common at scale, so failure detection should be rapid and efficient. Existing unicast and multicast sessions should proceed unaffected to the extent allowed by underlying physical connectivity.

Let us now map these requirements to implications for the underlying network protocols. R1 and R2 essentially require supporting a single layer 2 fabric for the entire data center. A layer 3 fabric would require configuring each switch with its subnet information and synchronizing DHCP servers to distribute IP addresses based on the host's subnet. Worse, transparent VM migration is not possible at layer 3 (save through techniques designed for IP mobility) because VMs must switch their IP addresses if they migrate to a host on a different subnet. Unfortunately, layer 2 fabrics face scalability and efficiency challenges because of the need to support broadcast. Further, R3 at layer 2 requires MAC forwarding tables with potentially hundreds of thousands or even millions of entries, impractical with today's switch hardware. R4 is difficult for either layer 2 or layer 3 because forwarding loops are possible during routing convergence. A layer 2 protocol may avoid such loops by employing a single spanning tree (inefficient) or tolerate them by introducing an additional header with a TTL (incompatible). R5 requires efficient routing protocols that can disseminate topology changes quickly to all points of interest. Unfortunately, existing layer 2 and layer 3 routing protocols, e.g., ISIS and OSPF, are broadcast based, with every switch update sent to all switches. On the efficiency side, the broadcast overhead of such protocols would likely require configuring the equivalent of routing areas [5], contrary to R2.

Hence, the current assumption is that the vision of a unified plug-and-play large-scale network fabric is unachievable, leaving data center network architects to adopt ad hoc partitioning and configuration to support large-scale deployments. Recent work in SEATTLE [10] makes dramatic advances toward a plug-and-play Ethernet-compatible protocol. However, in SEATTLE, switch state grows with the number of hosts in the data center, forwarding loops remain possible, and routing requires all-to-all broadcast, violating R3, R4, and R5. Section 3.7 presents a detailed discussion of both SEATTLE and TRILL [17].

In this paper, we present PortLand, a set of Ethernet-compatible routing, forwarding, and address resolution protocols with the goal of meeting R1-R5 above. The principal observation behind our work is that data center networks are often physically inter-connected as a multi-rooted tree [1]. Using this observation, PortLand employs a lightweight protocol to enable switches to discover their position in the topology. PortLand further assigns internal Pseudo MAC (PMAC) addresses to all end hosts to encode their position in the topology. PMAC addresses enable efficient, provably loop-free forwarding with small switch state.

We have a complete implementation of PortLand. We provide native fault-tolerant support for ARP, network-layer multicast, and broadcast. PortLand imposes little requirements on the underlying switch software and hardware. We hope that PortLand enables a move towards more flexible, efficient and fault-tolerant data centers where applications may flexibly be mapped to different hosts, i.e. where the data center network may be treated as one unified fabric.

2. BACKGROUND

2.1 Data Center Networks

Topology.

Current data centers consist of thousands to tens of thousands of computers with emerging mega data centers hosting 100,000+ compute nodes. As one example, consider our interpretation of current best practices [1] for the layout of a 11,520-port data center network. Machines are organized into racks and rows, with a logical hierarchical network tree overlaid on top of the machines. In this example, the data center consists of 24 rows, each with 12 racks. Each rack contains 40 machines interconnected by a top of rack (ToR) switch that delivers non-blocking bandwidth among directly connected hosts. Today, a standard ToR switch contains 48 GigE ports and up to 4 available 10 GigE uplinks.

ToR switches connect to end of row (EoR) switches via 1-4 of the available 10 GigE uplinks. To tolerate individual switch failures, ToR switches may be connected to EoR switches in different rows. An EoR switch is typically a modular 10 GigE switch with a number of ports corresponding to the desired aggregate bandwidth. For maximum bandwidth, each of the 12 ToR switches would connect all 4 available 10 GigE uplinks to a modular 10 GigE switch with up to 96 ports. 48 of these ports would face downward towards the ToR switches and the remainder of the ports would face upward to a *core* switch layer. Achieving maximum bandwidth for inter-row communication in this example requires connecting 48 upward facing ports from each of 24 EoR switches to a core switching layer consisting of 12 96-port 10 GigE switches.

Forwarding.

There are a number of available data forwarding techniques in data center networks. The high-level dichotomy is between creating a Layer 2 network or a Layer 3 network, each with associated tradeoffs. A Layer 3 approach assigns IP addresses to hosts hierarchically based on their directly connected switch. In the example topology above, hosts connected to the same ToR could be assigned the same /26 prefix and hosts in the same row may have a /22 prefix. Such careful assignment will enable relatively small forwarding tables across all data center switches.

Standard intra-domain routing protocols such as OSPF [22] may be employed among switches to find shortest paths among hosts. Failures in large-scale network topologies will be commonplace. OSPF can detect such failures and then broadcast the information to all switches to avoid failed links or switches. Transient loops with layer 3 forwarding is less of an issue because the IP-layer TTL limits per-packet resource consumption while forwarding tables are being asynchronously updated.

Unfortunately, Layer 3 forwarding does impose administrative burden as discussed above. In general, the process of adding a new switch requires manual administrator configuration and oversight, an error prone process. Worse, improperly synchronized state between system components, such as a DHCP server and a configured switch subnet identifier can lead to unreachable hosts and difficult to diagnose errors. Finally, the growing importance of end host virtualization makes Layer 3 solutions less desirable as described below.

For these reasons, certain data centers deploy a layer 2 network where forwarding is performed based on flat MAC addresses. A layer 2 fabric imposes less administrative overhead. Layer 2 fabrics have their own challenges of course. Standard Ethernet bridging [24] does not scale to networks with tens of thousands of hosts because of the need to support broadcast across the entire fabric. Worse, the presence of a single forwarding spanning tree (even if optimally designed) would severely limit performance in topologies that consist of multiple available equal cost paths.

A middle ground between a Layer 2 and Layer 3 fabric consists of employing VLANs to allow a single logical Layer 2 fabric to cross multiple switch boundaries. While feasible for smaller-scale topologies, VLANs also suffer from a number of drawbacks. For instance, they require bandwidth resources to be explicitly assigned to each VLAN at each participating switch, limiting flexibility for dynamically changing communication patterns. Next, each switch must maintain state for all hosts in each VLAN that they participate in, limiting scalability. Finally, VLANs also use a single forwarding spanning tree, limiting performance.

End Host Virtualization.

The increasing popularity of end host virtualization in the data center imposes a number of requirements on the underlying network. Commercially available virtual machine monitors allow tens of VMs to run on each physical machine in the data center[1], each with their own fixed IP and MAC addresses. In data centers with hundreds of thousands of hosts, this translates to the need for scalable addressing and forwarding for millions of unique end points. While individual applications may not (yet) run at this scale, application designers and data center administrators alike would still benefit from the ability to arbitrarily map individual applications to an arbitrary subset of available physical resources.

Virtualization also allows the entire VM state to be transmitted across the network to migrate a VM from one physical machine to another [11]. Such migration might take place for a variety of reasons. A cloud computing hosting service may migrate VMs for statistical multiplexing, packing VMs on the smallest physical footprint possible while still maintaining performance guarantees. Further, variable bandwidth to remote nodes in the data center could warrant migration based on dynamically changing communication patterns to achieve high bandwidth for tightly-coupled hosts. Finally, variable heat distribution and power availability in the data center (in steady state or as a result of component cooling or power failure) may necessitate VM migration to avoid hardware failures.

Such an environment currently presents challenges both for Layer 2 and Layer 3 data center networks. In a Layer 3 setting, the IP address of a virtual machine is set by its directly-connected switch subnet number. Migrating the VM to a different switch would require assigning a new IP address based on the subnet number of the new first-hop switch, an operation that would break all open TCP connections to the host and invalidate any session state maintained across the data center, etc. A Layer 2 fabric is agnostic to the IP address of a VM. However, scaling ARP and performing routing/forwarding on millions of flat MAC addresses introduces a separate set of challenges.

2.2 Fat Tree Networks

Recently proposed work [6, 14, 15] suggest alternate topologies for scalable data center networks. In this paper, we consider designing a scalable fault tolerant layer 2 domain over one such topology, a fat tree. As will become evident, the fat tree is simply an instance of the traditional data center multi-rooted tree topology (Section 2.1). Hence, the techniques described in this paper generalize to existing data center topologies. We present the fat tree because our available hardware/software evaluation platform (Section 4) is built as a fat tree.

Figure 1 depicts a 16-port switch built as a multi-stage topology from constituent 4-port switches. In general, a three-stage fat tree built from k-port switches can support non-blocking communication among $k^3/4$ end hosts using $5k^2/4$ individual k-port switches. We split the fat tree into three layers, labeled edge, aggregation and core as in Figure 1. The fat tree as a whole is split into k individual *pods*, with each pod supporting non-blocking operation among $k^2/4$ hosts. Non-blocking operation requires careful scheduling of packets among all available paths, a challenging problem. While a number of heuristics are possible, for the purposes of this work we assume ECMP-style hashing of flows [16] among the $k^2/4$ available paths between a given source and destination. While current techniques are less than ideal, we consider the flow scheduling problem to be beyond the scope of this paper.

2.3 Related Work

Recently, there have been a number of proposals for network architectures specifically targeting the data center. Two recent proposals [14, 6] suggest topologies based on fat trees [19]. As discussed earlier, fat trees are a form of multi-rooted trees that already form the basis for many existing data center topologies. As such, they are fully compatible with our work and in fact our implementation runs on top of a small-scale fat tree. DCell [15] also recently proposed a specialized topology for the data center environment. While not strictly a multi-rooted tree, there is implicit hierarchy in the DCell topology, which should make it compatible with our techniques.

Others have also recently recognized the need for more scalable layer 2 networks. SmartBridge [26] extended the original pioneering work on learning bridges [24] to move beyond single spanning tree networks while maintaining the loop free property of extended LANs. However, SmartBridge still suffers from the scalability challenges characteristic of Ethernet networks. Contemporaneous to our work, MOOSE [28] also suggests the use of hierarchical Ethernet addresses and header rewriting to address some of Ethernet's scalability limitations.

RBridges and TRILL [25], its IETF standardization effort, address some of the routing challenges in Ethernet. RBridges run a layer 2 routing protocol among switches. Essentially switches broadcast information about their local connectivity along with the identity of all directly connected end hosts. Thus, all switches learn the switch topology and the location of all hosts. To limit forwarding table size, ingress switches map destination MAC addresses to the appropriate egress switch (based on global knowledge) and encapsulate the packet in an outer MAC header with the egress switch identifier. In addition, RBridges add a secondary header with a TTL field to protect against loops. We also take inspiration from CMU Ethernet [23], which also proposed maintaining a distributed directory of all host information. Relative to both approaches, PortLand is able to achieve improved fault tolerance and efficiency by lever-

[1]One rule of thumb for the degree of VM-multiplexing allocates one VM per thread in the underlying processor hardware. x86 machines today have 2 sockets, 4 cores/processor, and 2 threads/core. Quad socket, eight core machines will be available shortly.

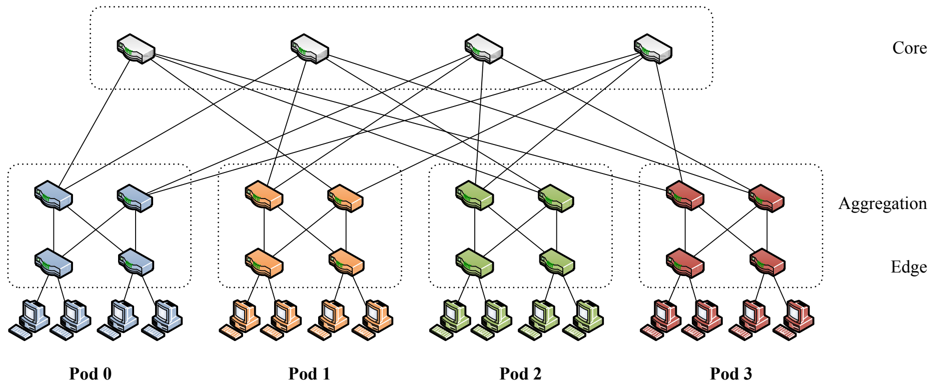


Figure 1: Sample fat tree topology.

aging knowledge about the baseline topology and avoiding broadcast-based routing protocols altogether.

Failure Carrying Packets (FCP) [18] shows the benefits of assuming some knowledge of baseline topology in routing protocols. Packets are marked with the identity of all failed links encountered between source and destination, enabling routers to calculate new forwarding paths based on the failures encountered thus far. Similar to PortLand, FCP shows the benefits of assuming knowledge of baseline topology to improve scalability and fault tolerance. For example, FCP demonstrates improved routing convergence with fewer network messages and lesser state.

To reduce the state and communication overhead associated with routing in large-scale networks, recent work [8, 9, 10] explores using DHTs to perform forwarding on flat labels. We achieve similar benefits in per-switch state overhead with lower network overhead and the potential for improved fault tolerance and efficiency, both in forwarding and routing, by once again leveraging knowledge of the baseline topology.

3. DESIGN

The goal of PortLand is to deliver scalable layer 2 routing, forwarding, and addressing for data center network environments. We leverage the observation that in data center environments, the baseline multi-rooted network topology is known and relatively fixed. Building and maintaining data centers with tens of thousands of compute elements requires modularity, advance planning, and minimal human interaction. Thus, the baseline data center topology is unlikely to evolve quickly. When expansion does occur to the network, it typically involves adding more "leaves" (e.g., rows of servers) to the multi-rooted tree topology described in Section 2.1.

3.1 Fabric Manager

PortLand employs a logically centralized *fabric manager* that maintains soft state about network configuration information such as topology. The fabric manager is a user process running on a dedicated machine responsible for assisting with ARP resolution, fault tolerance, and multicast as further described below. The fabric manager may simply be a redundantly-connected host in the larger topology or it may run on a separate control network.

There is an inherent trade off between protocol simplicity and system robustness when considering a distributed versus centralized realization for particular functionality. In PortLand, we restrict the amount of centralized knowledge and limit it to soft state. In this manner, we eliminate the need for any administrator configuration of the fabric manager (e.g., number of switches, their location, their identifier). In deployment, we expect the fabric manager to be replicated with a primary asynchronously updating state on one or more backups. Strict consistency among replicas is not necessary as the fabric manager maintains no hard state.

Our approach takes inspiration from other recent large-scale infrastructure deployments. For example, modern storage [13] and data processing systems [12] employ a centralized controller at the scale of tens of thousands of machines. In another setting, the Route Control Platform [7] considers centralized routing in ISP deployments. All the same, the protocols described in this paper are amenable to distributed realizations if the tradeoffs in a particular deployment environment tip against a central fabric manager.

3.2 Positional Pseudo MAC Addresses

The basis for efficient forwarding and routing as well as VM migration in our design is hierarchical Pseudo MAC (PMAC) addresses. PortLand assigns a unique PMAC address to each end host. The PMAC encodes the location of an end host in the topology. For example, all end points in the same pod will have the same prefix in their assigned PMAC. The end hosts remain unmodified, believing that they maintain their actual MAC (AMAC) addresses. Hosts performing ARP requests receive the PMAC of the destination host. All packet forwarding proceeds based on PMAC addresses, enabling very small forwarding tables. Egress switches perform PMAC to AMAC header rewriting to maintain the illusion of unmodified MAC addresses at the destination host.

PortLand edge switches learn a unique pod number and a unique position number within each pod. We employ the Location Discovery Protocol (Section 3.4) to assign these values. For all directly connected hosts, edge switches assign a 48-bit PMAC of the form *pod.position.port.vmid* to all directly connected hosts, where *pod* (16 bits) reflects the pod number of the edge switch, *position* (8 bits) is its position in the pod, and *port* (8 bits) is the switch-local view of the port number the host is connected to. We use *vmid* (16 bits) to multiplex multiple virtual machines on the same physical machine (or physical hosts on the other side of a bridge). Edge switches assign monotonically increasing *vmid*'s to each subsequent new MAC address observed on a given port. PortLand times out *vmid*'s without any traffic and reuses them.

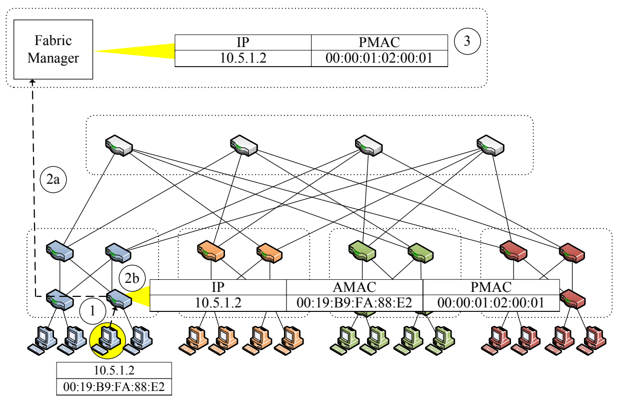


Figure 2: Actual MAC to Pseudo MAC mapping.

When an ingress switch sees a source MAC address never observed before, the packet is vectored to the switch software. The software creates an entry in a local PMAC table mapping the host's AMAC and IP address to its PMAC. The switch constructs the PMAC as described above and communicates this mapping to the fabric manager as depicted in Figure 2. The fabric manager uses this state to respond to ARP requests (Section 3.3). The switch also creates the appropriate flow table entry to rewrite the PMAC destination address to the AMAC for any traffic destined to the host.

In essence, we separate host location from host identifier [21] in a manner that is transparent to end hosts and compatible with existing commodity switch hardware. Importantly, we do not introduce additional protocol headers. From the underlying hardware, we require flow table entries to perform deterministic PMAC $\leftrightarrow$ AMAC rewriting as directed by the switch software. We also populate switch forwarding entries based on longest prefix match against a destination PMAC address. OpenFlow [4] supports both operations and native hardware support is also available in commodity switches [2].

3.3 Proxy-based ARP

Ethernet by default broadcasts ARPs to all hosts in the same layer 2 domain. We leverage the fabric manager to reduce broadcast overhead in the common case, as depicted in Figure 3. In step 1, an edge switch intercepts an ARP request for an IP to MAC address mapping and forwards the request to the fabric manager in step 2. The fabric manager consults its PMAC table to see if an entry is available for the target IP address. If so, it returns the PMAC in step 3 to the edge switch. The edge switch creates an ARP reply in step 4 and returns it to the original host.

It is possible that the fabric manager does not have the IP to PMAC mapping available, for example after failure. In

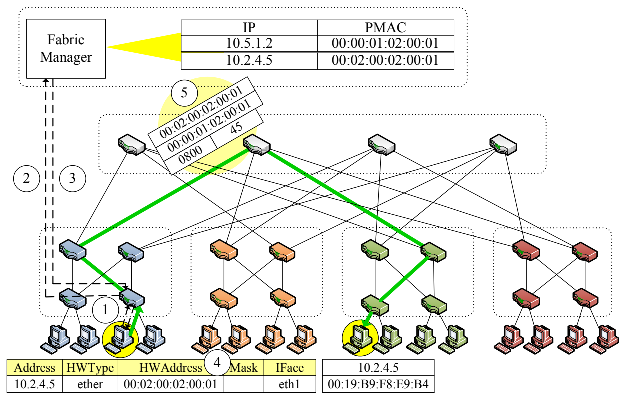


Figure 3: Proxy ARP.

this case, the fabric manager will fall back to broadcast to all end hosts to retrieve the mapping. Efficient broadcast is straightforward in the failure-free case (fault-tolerance extensions are described below): the ARP is transmitted to any core switch, which in turn distributes it to all pods and finally all edge switches. The target host will reply with its AMAC, which will be rewritten by the ingress switch to the appropriate PMAC before forwarding to both the querying host and the fabric manager.

Note that end hosts receive PMACs in response to an ARP request and that all packet forwarding proceeds based on the hierarchical PMAC. The egress switch performs PMAC to AMAC rewriting only on the last hop to the destination host. In the baseline, forwarding in each switch requires just $O(k)$ state using hierarchical PMAC addresses. This required state compares favorably to standard layer 2 switches that require an entry for every flat MAC address in the network, i.e., tens or even hundreds of thousands in large deployments. Additional forwarding state may be required to perform per-flow load balancing across multiple paths [6].

There is one additional detail for supporting VM migration. Upon completing migration from one physical machine to another, the VM sends a gratuitous ARP with its new IP to MAC address mapping. This ARP is forwarded to the fabric manager in the normal manner. Unfortunately, any hosts communicating with the migrated VM will maintain that host's previous PMAC in their ARP cache and will be unable to continue communication until their ARP cache entry times out. To address this limitation, the fabric manager forwards an invalidation message to the migrated VM's previous switch. This message sets up a flow table entry to trap handling of subsequent packets destined to the invalidated PMAC to the switch software. The switch software transmits a unicast gratuitous ARP back to any transmitting host to set the new PMAC address in that host's ARP cache. The invalidating switch may optionally transmit the packet to the actual destination to prevent packet loss.

3.4 Distributed Location Discovery

PortLand switches use their position in the global topology to perform more efficient forwarding and routing using only pairwise communication. Switch position may be set manually with administrator intervention, violating some of our original goals. Since position values should be slow to

Algorithm 1 *LDP_listener_thread*()

```
1:  **While** (true)
2:      **For each** $tp$ **in** $tentative\_pos$
3:          **If** $(curr\_time - tp.time) > timeout$
4:              $tentative\_pos \leftarrow tentative\_pos - \{tp\}$;
5:      ▷ **Case 1:** On receipt of LDM $P$
6:      $Neighbors \leftarrow Neighbors \bigcup \{switch\ that\ sent\ P\}$
7:      **If** $(curr\_time - start\_time > T$ **and** $|Neighbors| \leq \frac{k}{2})$
8:          $my\_level \leftarrow 0$; $incoming\_port \leftarrow$ up;
9:          $Acquire\_position\_thread()$;
10:     **If** $(P.level = 0$ **and** $P.dir =$ up$)$
11:         $my\_level \leftarrow 1$; $incoming\_port \leftarrow$ down;
12:     **Else If** $(P.dir =$ down$)$
13:         $incoming\_port \leftarrow$ up;
14:     **If** $(my\_level = -1$ **and** $|Neighbors| = k)$
15:         $is\_core \leftarrow$ true;
16:         **For each** $switch$ **in** $Neighbors$
17:             **If** $(switch.level \neq 1$ **or** $switch.dir \neq -1)$
18:                 $is\_core \leftarrow$ false; **break**;
19:         **If** $(is\_core =$ true$)$
20:             $my\_level \leftarrow 2$; Set $dir$ of all ports to down;
21:     **If** $(P.pos \neq -1$ **and** $P.pos \nsubseteq Pos\_used)$
22:         $Pos\_used \leftarrow Pos\_used \bigcup \{P.pos\}$;
23:     **If** $(P.pod \neq -1$ **and** $my\_level \neq 2)$
24:         $my\_pod \leftarrow P.pod$;
25:
26:     ▷ **Case 2:** On receipt of position proposal $P$
27:     **If** $(P.proposal \nsubseteq (Pos\_used \bigcup tentative\_pos))$
28:         $reply \leftarrow$ {"Yes"};
29:         $tentative\_pos \leftarrow tentative\_pos \bigcup \{P.proposal\}$;
30:     **Else**
31:         $reply \leftarrow$ {"No", $Pos\_used$, $tentative\_pos$};
```

Algorithm 2 *Acquire_position_thread*()

```
1:  $taken\_pos = \{\}$;
2:  **While** $(my\_pos = -1)$
3:      $proposal \leftarrow random()\%\frac{k}{2}$, s.t. $proposal \nsubseteq taken\_pos$
4:      Send proposal on all upward facing ports
5:      Sleep($T$);
6:      **If** (more than $\frac{k}{4} + 1$ switches confirm $proposal$)
7:          $my\_pos = proposal$;
8:          **If**$(my\_pos = 0)$
9:              $my\_pod =$ Request from $Fabric\ Manager$;
10:     Update $taken\_pos$ according to replies;
```

change, this may still be a viable option. However, to explore the limits to which PortLand switches may be entirely plug-and-play, we also present a location discovery protocol (LDP) that requires no administrator configuration. PortLand switches do not begin packet forwarding until their location is established.

PortLand switches periodically send a Location Discovery Message (LDM) out all of their ports both, to set their positions and to monitor liveness in steady state. LDMs contain the following information:

- *Switch identifier (switch_id)*: a globally unique identifier for each switch, e.g., the lowest MAC address of all local ports.

- *Pod number (pod)*: a number shared by all switches in the same pod (see Figure 1). Switches in different pods will have different pod numbers. This value is never set for core switches.

- *Position (pos)*: a number assigned to each edge switch, unique within each pod.

- *Tree level (level)*: 0, 1, or 2 depending on whether the switch is an edge, aggregation, or core switch. Our approach generalizes to deeper hierarchies.

- *Up/down (dir)*: Up/down is a bit which indicates whether a switch port is facing downward or upward in the multi-rooted tree.

Initially, all values other than the switch identifier and port number are unknown and we assume the fat tree topology depicted in Figure 1. However, LDP also generalizes to multi-rooted trees as well as partially connected fat trees. We assume all switch ports are in one of three states: disconnected, connected to an end host, or connected to another switch.

The key insight behind LDP is that edge switches receive LDMs only on the ports connected to aggregation switches (end hosts do not generate LDMs). We use this observation to bootstrap level assignment in LDP. Edge switches learn their level by determining that some fraction of their ports are host connected. Level assignment then flows up the tree. Aggregations switches set their level once they learn that some of their ports are connected to edge switches. Finally, core switches learn their levels once they confirm that all ports are connected to aggregation switches.

Algorithm 1 presents the processing performed by each switch in response to LDMs. Lines 2-4 are concerned with position assignment and will be described below. In line 6, the switch updates the set of switch neighbors that it has heard from. In lines 7-8, if a switch is not connected to more than $k/2$ neighbor switches for sufficiently long, it concludes that it is an edge switch. The premise for this conclusion is that edge switches have at least half of their ports connected to end hosts. Once a switch comes to this conclusion, on any subsequent LDM it receives, it infers that the corresponding incoming port is an upward facing one. While not shown for simplicity, a switch can further confirm its notion of position by sending pings on all ports. Hosts will reply to such pings but will not transmit LDMs. Other PortLand switches will both reply to the pings and transmit LDMs.

In lines 10-11, a switch receiving an LDM from an edge switch on an upward facing port concludes that it must be an aggregation switch and that the corresponding incoming port is a downward facing port. Lines 12-13 handle the case where core/aggregation switches transmit LDMs on downward facing ports to aggregation/edge switches that have not yet set the direction of some of their ports.

Determining the level for core switches is somewhat more complex, as addressed by lines 14-20. A switch that has not yet established its level first verifies that all of its active ports are connected to other PortLand switches (line 14). It then verifies in lines 15-18 that all neighbors are aggregation switches that have not yet set the direction of their links (aggregation switch ports connected to edge switches would have already been determined to be downward facing). If these conditions hold, the switch can conclude that it is a core switch and set all its ports to be downward facing (line 20).

Edge switches must acquire a unique position number in each pod in the range of $0..\frac{k}{2} - 1$. This process is depicted in Algorithm 2. Intuitively, each edge switch proposes a randomly chosen number in the appropriate range to all aggregation switches in the same pod. If the proposal is verified by a majority of these switches as unused and not tentatively reserved, the proposal is finalized and this value will be included in future LDMs from the edge switch. As shown in lines 2-4 and 29 of Algorithm 1, aggregation switches will hold a proposed position number for some period of time before timing it out in the case of multiple simultaneous proposals for the same position number.

LDP leverages the fabric manager to assign unique pod numbers to all switches in the same pod. In lines 8-9 of Algorithm 2, the edge switch that adopts position 0 requests a pod number from the fabric manager. This pod number spreads to the rest of the pod in lines 21-22 of Algorithm 1.

For space constraints, we leave a description of the entire algorithm accounting for a variety of failure and partial connectivity conditions to separate work. We do note one of the interesting failure conditions, miswiring. Even in a data center environment, it may still be possible that two host facing ports inadvertently become bridged. For example, someone may inadvertently plug an Ethernet cable between two outward facing ports, introducing a loop and breaking some of the important PortLand forwarding properties. LDP protects against this case as follows. If an uninitialized switch begins receiving LDMs from an edge switch on one of its ports, it must be an aggregation switch or there is an error condition. It can conclude there is an error condition if it receives LDMs from aggregation switches on other ports or if most of its active ports are host-connected (and hence receive no LDMs). In an error condition, the switch disables the suspicious port and signals an administrator exception.

3.5 Provably Loop Free Forwarding

Once switches establish their local positions using LDP, they employ updates from their neighbors to populate their forwarding tables. For instance, core switches learn the pod number of directly-connected aggregation switches. When forwarding a packet, the core switch simply inspects the bits corresponding to the pod number in the PMAC destination address to determine the appropriate output port.

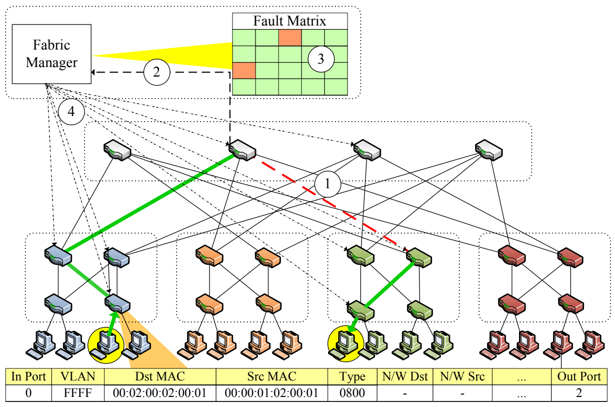
Similarly, aggregation switches learn the position number of all directly connected edge switches. Aggregation switches must determine whether a packet is destined for a host in the same or different pod by inspecting the PMAC. If in the same pod, the packet must be forwarded to an output port corresponding to the *position* entry in the PMAC.

If in a different pod, the packet may be forwarded along any of the aggregation switch's links to the core layer in the fault-free case. For load balancing, switches may employ any number of techniques to choose an appropriate output port. The fabric manager would employ additional flow table entries to override the default forwarding behavior for individual flows. However, this decision is orthogonal to this work, and so we assume a standard technique such as flow hashing in ECMP [16].

PortLand maps multicast groups to a core switch using a deterministic hash function. PortLand switches forward all multicast packets towards this core, e.g., using flow hashing to pick among available paths. With simple hardware support, the hash function may be performed in hardware with no additional state in the fault-free case (exceptions for failures could be encoded in switch SRAM). Without hardware support, there would be one entry per multicast group. Edge switches forward IGMP join requests to the fabric manager using the PMAC address of the joining host. The fabric manager then installs forwarding state in all core and aggregation switches necessary to ensure multicast packet delivery to edge switches with at least one interested host.

Our forwarding protocol is provably loop free by observing up-down semantics [27] in the forwarding process as explained in Appendix A. Packets will always be forwarded up to either an aggregation or core switch and then down toward their ultimate destination. We protect against transient loops and broadcast storms by ensuring that once a packet begins to travel down, it is not possible for it to travel back up the topology. There are certain rare simultaneous failure conditions where packets may only be delivered by, essentially, detouring back down to an aggregation switch to get to a core switch capable of reaching a given destination. We err on the side of safety and prefer to lose connectivity in these failure conditions rather than admit the possibility of loops.

3.6 Fault Tolerant Routing

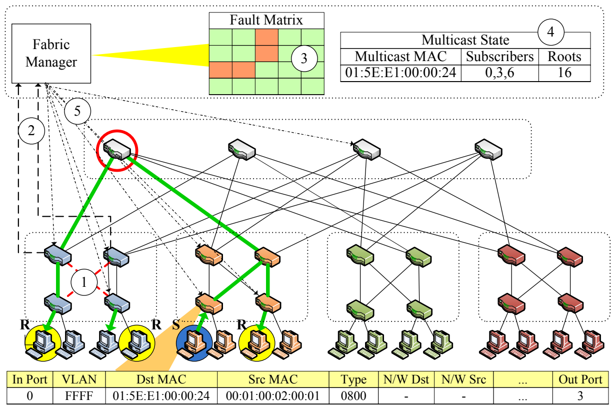


In Port	VLAN	Dst MAC	Src MAC	Type	N/W Dst	N/W Src	...	Out Port
0	FFFF	00:02:00:02:00:01	00:00:01:02:00:01	0800	-	-	...	2

Figure 4: Unicast: Fault detection and action.

Given a largely fixed baseline topology and the ability to forward based on PMACs, PortLand's routing protocol is largely concerned with detecting switch and link failure/recovery. LDP exchanges (Section 3.4) also serve the dual purpose of acting as liveness monitoring sessions. We describe our failure recovery process using an example, as depicted in Figure 4. Upon not receiving an LDM (also referred to as a *keepalive* in this context) for some configurable period of time, a switch assumes a link failure in step 1. The detecting switch informs the fabric manager about the failure in step 2. The fabric manager maintains a logical fault matrix with per-link connectivity information for the entire topology and updates it with the new information in step 3. Finally, in step 4, the fabric manager informs all affected switches of the failure, which then individually recalculate their forwarding tables based on the new version of the topology. Required state for network connectivity is modest, growing with $k^3/2$ for a fully-configured fat tree built from k-port switches.

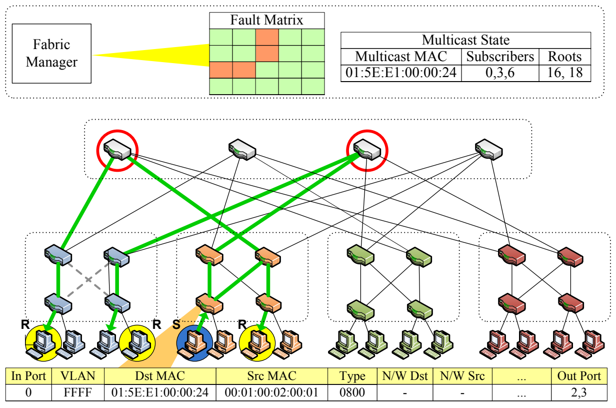
Traditional routing protocols require all-to-all communication among n switches with $O(n^2)$ network messages and associated processing overhead. PortLand requires $O(n)$ communication and processing, one message from the switch detecting failure to the fabric manager and, in the worst case, n messages from the fabric manager to affected switches.



Multicast State		
Multicast MAC	Subscribers	Roots
01:5E:E1:00:00:24	0,3,6	16

In Port	VLAN	Dst MAC	Src MAC	Type	N/W Dst	N/W Src	...	Out Port
0	FFFF	01:5E:E1:00:00:24	00:01:00:02:00:01	0800	-	-	...	3

Figure 5: Multicast: Fault detection and action.



Multicast State		
Multicast MAC	Subscribers	Roots
01:5E:E1:00:00:24	0,3,6	16, 18

In Port	VLAN	Dst MAC	Src MAC	Type	N/W Dst	N/W Src	...	Out Port
0	FFFF	01:5E:E1:00:00:24	00:01:00:02:00:01	0800	-	-	...	2,3

Figure 6: Multicast: After fault recovery.

We now consider fault tolerance for the multicast and broadcast case. Relative to existing protocols, we consider failure scenarios where there is no single spanning tree rooted at a core switch able to cover all receivers for a multicast group or broadcast session. Consider the example in Figure 5. Here, we have a multicast group mapped to the left-most core switch. There are three receivers, spread across pods 0 and 1. A sender forwards packets to the designated core, which in turn distributes the packets to the receivers. In step 1, two highlighted links in pod 0 simultaneously fail. Two aggregation switches detect the failure in step 2 and notify the fabric manager, which in turn updates its fault matrix in step 3. The fabric manager calculates forwarding entries for all affected multicast groups in step 4.

In this example, recovering from the failure requires forwarding through two separate aggregation switches in pod 0. However, there is no single core switch with simultaneous connectivity to both aggregation switches. Hence, a relatively simple failure scenario would result in a case where no single core-rooted tree can cover all interested receivers. The implications are worse for broadcast. We deal with this scenario by calculating a greedy set cover for the set of receivers associated with each multicast group. This may result in more than one designated core switch associated with a multicast or broadcast group. The fabric manager inserts the required forwarding state into the appropriate tables in step 5 of Figure 5.

Finally, Figure 6 depicts the forwarding state for the sender after the failure recovery actions. The multicast sender's edge switch now forwards two copies of each packet to two separate cores that split the responsibility for transmitting the multicast packet to the receivers.

3.7 Discussion

Given an understanding of the PortLand architecture, we now compare our approach to two previous techniques with similar goals, TRILL [25] and SEATTLE [10]. Table 1 summarizes the similarities and differences along a number of dimensions. The primary difference between the approaches is that TRILL and SEATTLE are applicable to general topologies. PortLand on the other hand achieves its simplicity and efficiency gains by assuming a multi-rooted tree topology such as those typically found in data center settings.

For forwarding, both TRILL and SEATTLE must in the worst case maintain entries for every host in the data center because they forward on flat MAC addresses. While in some enterprise deployment scenarios the number of popular destination hosts is limited, many data center applications perform all-to-all communication (consider search or MapReduce) where every host talks to virtually all hosts in the data center over relatively small time periods. PortLand forwards using hierarchical PMACs resulting in small forwarding state. TRILL employs MAC-in-MAC encapsulation to limit forwarding table size to the total number of switches, but must still maintain a rewriting table with entries for every global host at ingress switches.

Both TRILL and SEATTLE employ a broadcast-based link state protocol to discover the network topology. PortLand leverages knowledge of a baseline multi-rooted tree to allow each switch to establish its topological position based on local message exchange. We further leverage a logically centralized fabric manager to distribute failure information.

TRILL handles ARP locally since all switches maintain global topology knowledge. In TRILL, the link state protocol further broadcasts information about all hosts connected to each switch. This can add substantial overhead, especially when considering virtual machine multiplexing. SEATTLE distributes ARP state among switches using a one-hop DHT. All switches register the IP address to MAC mapping for their local hosts to a designated resolver. ARPs for an IP address may then be forwarded to the resolver rather than broadcast throughout the network.

While decentralized and scalable, this approach does admit unavailability of otherwise reachable hosts during the recovery period (i.e., several seconds) after a resolver switch fails. Worse, simultaneous loss of soft state in both the resolving switch and a host's ingress switch may leave certain hosts unreachable for an extended period of time. PortLand protects against these failure conditions by falling back to broadcast ARPs in the case where a mapping is unavailable in the fabric manager and associated state is lost. We are able to do so because the PortLand broadcast protocol is efficient, fault tolerant, and provably loop free.

To protect against forwarding loops, TRILL adds a secondary TRILL header to each packet with a TTL field. Unfortunately, this means that switches must both decrement

System	Topology	Forwarding		Routing	ARP	Loops	Multicast
		Switch State	Addressing				
TRILL	General	*O(number of global hosts)*	Flat; MAC-in-MAC encapsulation	Switch broadcast	All switches map MAC address to remote switch	TRILL header with TTL	ISIS extensions based on MOSPF
SEATTLE	General	*O(number of global hosts)*	Flat	Switch broadcast	One-hop DHT	Unicast loops possible	New construct: groups
PortLand	Multi-rooted tree	*O(number of local ports)*	Hierarchical	Location Discovery Protocol; Fabric manager for faults	Fabric manager	Provably loop free; no additional header	Broadcast-free routing; multi-rooted spanning trees

Table 1: System comparison

the TTL and recalculate the CRC for every frame, adding complexity to the common case. SEATTLE admits routing loops for unicast traffic. It proposes a new "group" construct for broadcast/multicast traffic. Groups run over a single spanning tree, eliminating the possibility of loops for such traffic. PortLand's forwarding is provably loop free with no additional headers. It further provides native support for multicast and network-wide broadcast using an efficient fault-tolerance mechanism.

4. IMPLEMENTATION

4.1 Testbed Description

Our evaluation platform closely matches the layout in Figure 1. Our testbed consists of 20 4-port NetFPGA PCI card switches [20]. Each switch contains 4 GigE ports along with Xilinx FPGA for hardware extensions. We house the NetFPGAs in 1U dual-core 3.2 GHz Intel Xeon machines with 3GB RAM. The network interconnects 16 end hosts, 1U quad-core 2.13GHz Intel Xeon machines with 3GB of RAM. All machines run Linux 2.6.18-92.1.18el5.

The switches run OpenFlow v0.8.9r2 [4], which provides the means to control switch forwarding tables. One benefit of employing OpenFlow is that it has already been ported to run on a variety of hardware platforms, including switches from Cisco, Hewlett Packard, and Juniper. This gives us some confidence that our techniques may be extended to commercial platforms using existing software interfaces and hardware functionality. Each switch has a 32-entry TCAM and a 32K entry SRAM for flow table entries. Each incoming packet's header is matched against 10 fields in the Ethernet, IP and TCP/UDP headers for a match in the two hardware flow tables. Each TCAM and SRAM entry is associated with an action, e.g., forward the packet along an output port or to the switch software. TCAM entries may contain "don't care" bits while SRAM matches must be exact.

4.2 System Architecture

PortLand intercepts all ARP requests and IGMP join requests at the edge switch and forwards them to the local switch software module running separately on the PC hosting each NetFPGA. The local switch module interacts with the OpenFlow fabric manager to resolve ARP requests and to manage forwarding tables for multicast sessions. The first few packets for new flows will miss in hardware flow tables and will be forwarded to the local switch module as a result. The switch module uses ECMP style hashing to choose among available forwarding paths in the switch and inserts a new flow table entry matching the flow. On receiving failure and recovery notifications from the fabric manager, each switch recalculates global connectivity and modifies the appropriate forwarding entries for the affected flows through the switch.

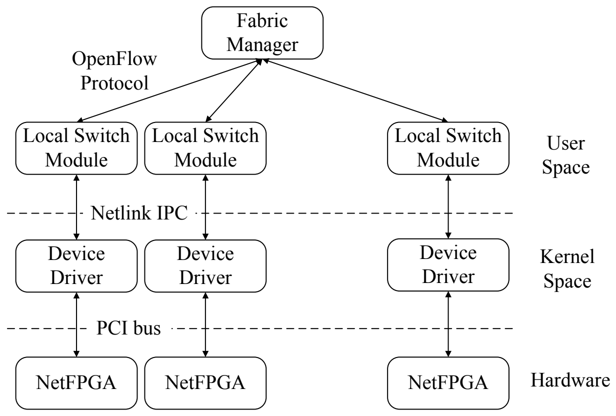


Figure 7: System architecture.

The OpenFlow fabric manager monitors connectivity with each switch module and reacts to the liveness information by updating its fault matrix. Switches also send keepalives to their immediate neighbors every 10ms. If no keepalive is received after 50ms, they assume link failure and update the fabric manager appropriately.

Figure 7 shows the system architecture. OpenFlow switch modules run locally on each switch. The fabric manager transmits control updates using OpenFlow messages to each switch. In our testbed, a separate control network supports communication between the fabric manager and local switch modules. It is of course possible to run the fabric manager simply as a separate host on the data plane and to communicate inband. The cost and wiring for a separate lower-speed control network will actually be modest. Consider a control network for a 2,880-switch data center for the $k = 48$ case. Less than 100 low-cost, low-speed switches should suffice to provide control plane functionality. The real question is whether the benefits of such a dedicated network will justify the additional complexity and management overhead.

Table 2 summarizes the state maintained locally at each switch as well as the fabric manager. Here
$k = Number\ of\ ports\ on\ the\ switches$,
$m = Number\ of\ local\ multicast\ groups$,
$p = Number\ of\ multicast\ groups\ active\ in\ the\ system$.

State	Switch	Fabric Manager
Connectivity Matrix	$O(k^3/2)$	$O(k^3/2)$
Multicast Flows	$O(m)$	$O(p)$
$IP \rightarrow PMAC$ mappings	$O(k/2)$	$O(k^3/4)$

Table 2: State requirements.

5. EVALUATION

In this section, we evaluate the efficiency and scalability of our implementation. We describe the experiments carried out on our system prototype and present measurements to characterize convergence and control overhead for both multicast and unicast communication in the presence of link failures. We ran all experiments on our testbed described in Section 4.

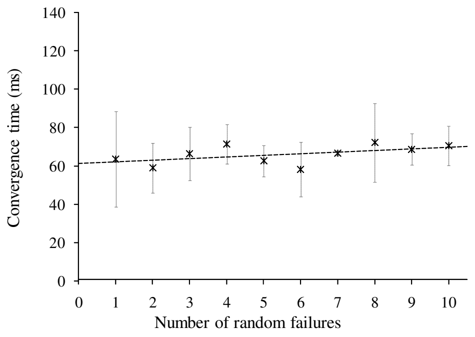


Figure 8: Convergence time with increasing faults.

Convergence Time With Increasing Faults.

We measured convergence time for a UDP flow while introducing a varying number of random link failures. A sender transmits packets at 250Mbps to a receiver in a separate pod. In the case where at least one of the failures falls on the default path between sender and receiver, we measured the total time required to re-establish communication.

Figure 8 plots the average convergence time across 20 runs as a function of the number of randomly-induced failures. Total convergence time begins at about 65ms for a single failure and increases slowly with the number of failures as a result of the additional processing time.

TCP convergence.

We repeated the same experiment for TCP communication. We monitored network activity using tcpdump at the sender while injecting a link failure along the path between sender and receiver. As illustrated in Figure 9, convergence for TCP flows takes longer than the baseline for UDP despite the fact that the same steps are taken in the underlying network. This discrepancy results because TCP loses an entire window worth of data. Thus, TCP falls back to the

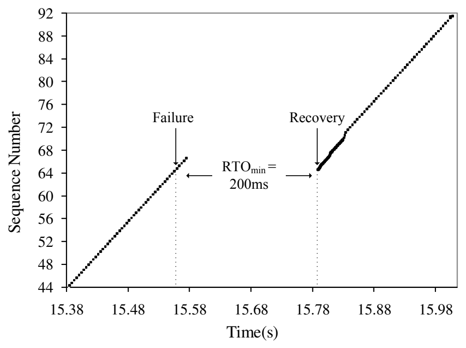


Figure 9: TCP convergence.

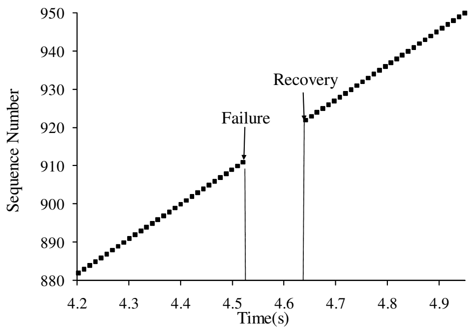


Figure 10: Multicast convergence.

retransmission timer, with TCP's RTO$_{min}$ set to 200ms in our system. By the time the first retransmission takes place, connectivity has already been re-established in the underlying network.

Multicast Convergence.

We further measured the time required to designate a new core when one of the subscribers of a multicast group loses connectivity to the current core. For this experiment, we used the same configuration as in Figure 5. In this case, the sender transmits a multicast flow to a group consisting of 3 subscribers, augmenting each packet with a sequence number. As shown in Figure 10, 4.5 seconds into the experiment we inject two failures (as depicted in Figure 5), causing one of the receivers to lose connectivity. After 110ms, connectivity is restored. In the intervening time, individual switches detect the failures and notify the fabric manager, which in turn reconfigures appropriate switch forwarding tables.

Scalability.

One concern regarding PortLand design is scalability of the fabric manager for larger topologies. Since we do not have a prototype at scale, we use measurements from our existing system to project the requirements of larger systems. Figure 11 shows the amount of ARP control traffic the fabric manager would be expected to handle as a function of overall cluster size. One question is the number of

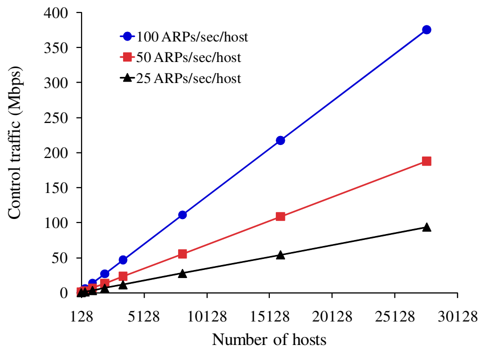


Figure 11: Fabric manager control traffic.

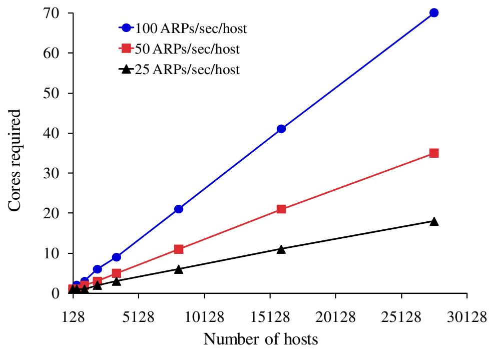


Figure 12: CPU requirements for ARP requests.

ARPs transmitted per host. Since we are interested in scalability under extreme conditions, we considered cases where each host transmitted 25, 50 and 100 ARP requests/sec to the fabric manager. Note that even 25 ARPs/sec is likely to be extreme in today's data center environments, especially considering the presence of a local ARP cache with a typical 60-second timeout. In a data center with each of the 27,648 hosts transmitting 100 ARPs per second, the fabric manager must handle a manageable 376Mbits/s of control traffic. More challenging is the CPU time required to handle each request. Our measurements indicate approximately 25 μs of time per request in our non-optimized implementation. Fortunately, the work is highly parallelizable, making it amenable to deployment on multiple cores and multiple hardware thread contexts per core. Figure 12 shows the CPU requirements for the fabric manager as a function of the number of hosts in the data center generating different numbers of ARPs/sec. For the highest levels of ARPs/sec and large data centers, the required level of parallelism to keep up with the ARP workload will be approximately 70 independent cores. This is beyond the capacity of a single modern machine, but this also represents a relatively significant number of ARP misses/second. Further, it should be possible to move the fabric manager to a small-scale cluster (e.g., four machines) if absolutely necessary when very high frequency of ARP requests is anticipated.

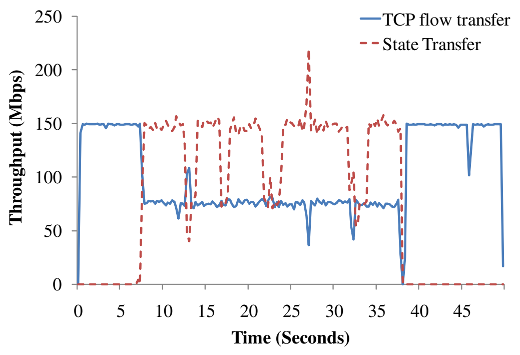


Figure 13: State and TCP application transfer during VM migration.

VM Migration.

Finally, we evaluate PortLand's ability to support virtual machine migration. In this experiment, a sender transmits data at 150 Mbps to a virtual machine (hosted on Xen) running on a physical machine in one pod. We then migrate the virtual machine to a physical machine in another pod. On migration, the host transmits a gratuitous ARP with its new MAC address, which is in turn forwarded to all hosts communicating with that VM by the previous egress switch. The communication is not at line-rate (1 Gbps) since we use software MAC layer rewriting capability provided by OpenFlow to support PMAC and AMAC translation at edge switches. This introduces additional per packet processing latency. Existing commercial switches have MAC layer rewriting support directly in hardware [2].

Figure 13 plots the results of the experiment with measured TCP rate for both state transfer and flow transfer (measured at the sender) on the y-axis as a function of time progressing on the x-axis. We see that 5+ seconds into the experiment, throughput of the tcp flow drops below the peak rate as the state of the VM begins to migrate to a new physical machine. During migration there are short time periods (200-600ms) during which the throughput of the flow drops to near zero (not visible in the graph due to the scale). Communication resumes with the VM at full speed after approximately 32 seconds (dominated by the time to complete VM state transfer).

6. CONCLUSIONS

The goal of this work is to explore the extent to which entire data center networks may be treated as a single plug-and-play fabric. Modern data centers may contain 100,000 hosts and employ virtual machine multiplexing that results in millions of unique addressable end hosts. Efficiency, fault tolerance, flexibility and manageability are all significant concerns with general-purpose Ethernet and IP-based protocols. In this paper, we present PortLand, a set of Ethernet-compatible routing, forwarding, and address resolution protocols specifically tailored for data center deployments. It is our hope that through protocols like PortLand, data center networks can become more flexible, efficient, and fault tolerant.

7. REFERENCES

[1] Cisco Data Center Infrastructure 2.5 Design Guide. www.cisco.com/application/pdf/en/us/guest/netsol/ns107/c649/ccmigration_09186a008073377d.pdf.

[2] Configuring IP Unicast Layer 3 Switching on Supervisor Engine 2. www.cisco.com/en/US/docs/routers/7600/ios/12.1E/configuration/guide/cef.html.

[3] Inside Microsoft's $550 Million Mega Data Centers. www.informationweek.com/news/hardware/data_centers/showArticle.jhtml?articleID=208403723.

[4] OpenFlow. www.openflowswitch.org/.

[5] OSPF Design Guide. www.ciscosystems.com/en/US/tech/tk365/technologies_white_paper09186a0080094e9e.shtml.

[6] M. Al-Fares, A. Loukissas, and A. Vahdat. A Scalable, Commodity Data Center Network Architecture. In *SIGCOMM '08: Proceedings of the ACM SIGCOMM 2008 conference on Data communication*, pages 63–74, New York, NY, USA, 2008. ACM.

[7] M. Caesar, D. Caldwell, N. Feamster, J. Rexford, A. Shaikh, and J. van der Merwe. Design and Implementation of a Routing Control Platform. In *USENIX Symposium on Networked Systems Design & Implementation*, 2005.

[8] M. Caesar, M. Castro, E. B. Nightingale, G. O, and A. Rowstron. Virtual Ring Routing: Network Routing Inspired by DHTs. In *Proceedings of ACM SIGCOMM*, 2006.

[9] M. Caesar, T. Condie, J. Kannan, K. Lakshminarayanan, I. Stoica, and S. Shenker. ROFL: Routing on Flat Labels. In *Proceedings of ACM SIGCOMM*, 2006.

[10] M. C. Changhoon Kim and J. Rexford. Floodless in SEATTLE: A Scalable Ethernet Architecture for Large Enterprises. In *SIGCOMM '08: Proceedings of the ACM SIGCOMM 2008 conference on Data communication*, 2008.

[11] C. Clark, K. Fraser, S. Hand, J. G. H. E. J. C. Limpach, I. Pratt, and A. Warfield. Live Migration of Virtual Machines. In *USENIX Symposium on Networked Systems Design & Implementation*, 2005.

[12] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI'04: Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation*, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[13] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. *ACM SIGOPS Operating Systems Review*, 37(5), 2003.

[14] A. Greenberg, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. Towards a Next Generation Data Center Architecture: Scalability and Commoditization. In *PRESTO '08: Proceedings of the ACM Workshop on Programmable Routers for Extensible Services of Tomorrow*, pages 57–62, New York, NY, USA, 2008. ACM.

[15] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, and S. Lu. DCell: A Scalable and Fault-tolerant Network Structure for Data Centers. In *Proceedings of the ACM SIGCOMM 2008 conference on Data communication*, pages 75–86, New York, NY, USA, 2008. ACM.

[16] C. Hopps. Analysis of an Equal-Cost Multi-Path Algorithm. RFC 2992, Internet Engineering Task Force, 2000.

[17] V. Kashyap. Dynamic Host Configuration Protocol (DHCP) over InfiniBand. RFC 4390 (Proposed Standard), 2006.

[18] K. Lakshminarayanan, M. Caesar, M. Rangan, T. Anderson, S. Shenker, I. Stoica, and H. Luo. Achieving Convergence-Free Routing Using Failure-Carrying Packets. In *Proceedings of ACM SIGCOMM*, 2007.

[19] C. E. Leiserson. Fat-Trees: Universal Networks for Hardware-Efficient Supercomputing. *IEEE Transactions on Computers*, 34(10):892–901, 1985.

[20] J. W. Lockwood, N. McKeown, G. Watson, G. Gibb, P. Hartke, J. Naous, R. Raghuraman, and J. Luo. NetFPGA–An Open Platform for Gigabit-Rate Network Switching and Routing. In *Proceedings of the 2007 IEEE International Conference on Microelectronic Systems Education*, pages 160–161, Washington, DC, USA, 2007. IEEE Computer Society.

[21] R. Moskowitz and P. Nikander. Host Identity Protocol (HIP) Architecture. RFC 4423 (Proposed Standard), 2006.

[22] J. Moy. OSPF Version 2. RFC 2328, Internet Engineering Task Force, 1998.

[23] A. Myers, T. S. E. Ng, and H. Zhang. Rethinking the Service Model: Scaling Ethernet to a Million Nodes. In *ACM HotNets-III*, 2004.

[24] L. S. C. of the IEEE Computer Society. IEEE Standard for Local and Metropolitan Area Networks, Common Specifications Part 3: Media Access Control (MAC), Bridges Ammendment 2: Rapid Reconfiguration, June 2001.

[25] R. Perlman, D. Eastlake, D. G. Dutt, S. Gai, and A. Ghanwani. Rbridges: Base Protocol Specification. Technical report, Internet Engineering Task Force, 2009.

[26] T. L. Rodeheffer, C. A. Thekkath, and D. C. Anderson. SmartBridge: A Scalable Bridge Architecture. In *Proceedings of ACM SIGCOMM*, 2001.

[27] M. D. Schroeder, A. D. Birrell, M. Burrows, H. Murray, R. M. Needham, T. L. Rodeheffer, E. H. Satterthwaite, and C. P. Thacker. Autonet: A High-Speed, Self-Configuring Local Area Network Using Point-to-Point Links. In *IEEE Journal On Selected Areas in Communications*, 1991.

[28] M. Scott and J. Crowcroft. MOOSE: Addressing the Scalability of Ethernet. In *EuroSys Poster session*, 2008.

Appendix A: Loop-Free Proof

A fat-tree network topology has many physical loops, which can easily lead to forwarding loops given some combination of forwarding rules present in the switches. However, physical loops in data center networks are desirable and provide many benefits such as increased network bisection bandwidth and fault tolerance. Traditional Ethernet uses a minimum spanning tree to prevent forwarding loops at the cost of decreased bisection bandwidth and fault tolerance.

Here we show that fat trees can be constrained in such a way as to prevent forwarding loops, without requiring an explicit spanning tree. This constraint is simple, stateless, local to an individual switch, and uniform across all switches in the fat tree.

Constraint 1. *A switch must never forward a packet out along an upward-facing port when the ingress port for that packet is also an upward-facing port.*

Theorem 1. *When all switches satisfy Constraint 1 (C1), a fat tree will contain no forwarding loops.*

Proof. C1 prevents traffic from changing direction more than once. It imposes the logical notion of up-packets and down-packets. Up-packets may travel only upward through the tree, whereas down-packets may travel only downward. C1 effectively allows a switch to perform a one-time conversion of an up-packet to a down-packet. There is no provision for converting a down-packet to an up-packet. In order for a switch to receive the same packet from the same ingress port more than once, this packet should change its direction at least twice while routed through the tree topology. However this is not possible since there is no mechanism for converting a down-packet to an up-packet, something that would be required for at least one of these changes in direction. □