UNIVERSITY OF CALIFORNIA, SAN DIEGO

The Design and Implementation of a Managed Network Fabric

A thesis submitted in partial satisfaction of the
requirements for the degree
Master of Science

in

Computer Science

by

Nelson Huang

Committee in charge:

Professor Amin Vahdat, Chair
Professor Keith Marzullo
Professor Geoffrey M. Voelker

2011

EPIGRAPH

When in doubt, use brute force.

—Butler Lampson

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to thank my advisor, Professor Amin Vahdat, for the opportunity and pleasure of working on DCSwitch. Without his support, vision, and humor, the milestones accomplished with the DCSwitch project would simply not have been possible. Secondly, I would also like to thank my fellow lab members. It was truly a pleasure to work with each member of the lab to build DCSwitch. I would also like to thank Professor Geoffrey Voelker and Professor Keith Marzullo for serving on my thesis committee. Their classroom teachings were invaluable in the design and implementation of DCSwitch. A good part of the DCSwitch code can be summarized by the words, condition variable.

Several chapters in this thesis are extended versions of content from the following papers, "Portland: A Scalable Fault-Tolerant Layer 2 Data Center Network Fabric" and "Hedera: Dynamic Flow Scheduling for Data Center Networks" that I helped co-author with Radhika Niranjan Mysore, Andreas Pamboris, Nathan Farrington, Nelson Huang, Pardis Miri, Sivasankar Radhakrishnan, Vikram Subramanya, Mohammed Al-Fares, Barath Raghavan, and Professor Amin Vahdat. Portland appears in the proceedings of the 2009 ACM SIGCOMM conference and Hedera appears in the 2010 ACM NSDI conference.

Chapters 1, 2, 3, and 4 are an adaption of sections 1, 2, and 3 of Portland. Section 2.2 is an adaption of multipathing as found in section 2.2 of Hedera. The section describing the Flow Scheduler design in Chapter 5 is an adaption of section 3.2 of the Hedera paper. The remaining content of Chapter 2 and the contents of Chapter 5 are the products of my research. The evaluation in Chapter 6 is the result of joint work with the team members listed earlier.

VITA

1995	B. S. in Applied Mathematics, Columbia University, New York, NY
2006-2011	M. S. in Computer Science, University of California, San Diego

PUBLICATIONS

Radhika Niranjan Mysore, Andreas Pamboris, Nathan Farrington, Nelson Huang, Pardis Miri, Sivasankar Radhakrishnan, Vikram Subramanya, and Amin Vahdat. "PortLand: A Scalable, Fault-Tolerant Layer 2 Data Center Network Fabric", In *Proceedings of ACM SIGCOMM 2009.*

Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. "Hedera: Dynamic Flow Scheduling for Data Center Networks", In *Proceedings of NSDI 2010.*

ABSTRACT OF THE THESIS

The Design and Implementation of a Managed Network Fabric

by

Nelson Huang

Master of Science in Computer Science

University of California, San Diego, 2011

Professor Amin Vahdat, Chair

The rapid growth of Internet applications and services such as search, social networking, and cloud computing services in conjunction with the adoption of commoditized hardware has led to the emergence of data centers consisting of tens to hundreds of thousands of computers. A common characteristic of applications running in these data centers is their use of parallelism in their design to enable system performance to grow by scaling out as opposed to scaling up. User interactions with these services often involve forwarding requests to thousands of computers in parallel. A user search request may involve accessing an inverted index stored on thousands of computers. In addition to the network traffic generated from users of front-end facing systems, there is also a significant amounts of intra-data center network traffic generated by backend applications such as MapReduce, Hadoop, and Dryad. These backend applications typically perform analytical analysis on petabyte-sized datasets. Constructing a network fabric that can meet the performance requirements of large scale data centers in an economical manner is an immense challenge. In this thesis, we describe the implementation of a managed network fabric intended for large scale data centers. Our system incorporates the best features of layer 2 and layer 3 networks while seeking to avoid the shortcomings of each protocol. Using commoditized hardware and software primitives, we construct a functioning prototype that treats the network fabric as a single logical unit and enables traffic to be dynamically provisioned in a fault tolerant manner. Our prototype demonstrates the feasibility of building such a system and that high performance can be achieved.

Chapter 1

Introduction

Economics, convenience, and technological advantages are spurring the growth of megasized data centers consisting of thousands to hundreds of thousands of computers. These data centers are often located in supportive environments that have low cost electricity, high speed fiber optic networks, and optimal climates [Rab]. As of 2006, Google had an estimated total of 450,000 servers [Hof08] in their data centers while Yahoo! and Microsoft also operate hundreds of thousands of servers [Car]. One trend that is spurring the growth of these megasized data centers is the use of low cost computer servers. By leveraging commoditized parts and bulk volume pricing, companies are discovering that it is cheaper and easier to build, deploy, and administer a large number of servers concentrated into a smaller set of data centers. Services and applications running in such an environment are commonly designed with the expectation that failure is the norm and that increased application performance comes from scaling out as opposed to scaling up. For applications such as search, social networking, and web commerce, a user request is often processed by hundreds to thousands of computers in parallel [Sha08]. These applications are often built on top of a large scale distributed file systems [GGL] [Bor] which replicate data blocks for fault tolerance. Data blocks in the systems typically range in size from 64 megabytes to 100 megabytes and are used to store petabyte sized datasets. In addition to external traffic, there is also significant intra-data center traffic from the use of parallel computing frameworks such as MapReduce [DG], Hadoop [had], and Dryad [IBY+]. These frameworks are used to perform backend analysis on data stored in the distributed file systems and exhibit significant network bandwidth demands. High oversubscription ratios in the network often cause MapReduce type jobs to become network bound. Another factor in the growth of data centers is the increasing use of virtualization for both server hosting and the implementation of cloud services. In a virtualized environment, multiple virtual servers can be hosted on a single server to increase machine utilization and can be dynamically migrated based on machine load and end user locality. Although it enables a data center center to maximize service agility, virtual machine migration is also network bandwidth demanding.

Creating and supporting a large scale network fabric for the data center environment has created a number of issues. From a topology standpoint, there is the challenge of simply being able to create a network interconnect that can scale to support hundreds of thousands of hosts. Because of cost and limited port densities, large size data center networks have been constructed by interconnecting large numbers of switches to provide the logical abstraction of a single large switch. In addition to providing connectivity, the network topology also needs to support high bandwidth communication between hosts and provide fault tolerance. Traditional topologies such as the hierarchical tree grow by scaling up as opposed to scaling out and suffer from bandwidth bottlenecks the higher network traffic travels in the tree.

Similar challenges also exist at the network protocol level. Numerous challenges and problems exist from deploying network protocols originally designed for LANs in a data center environment where hosts can number in the hundred of thousands. Ethernet's flat address space and plug-and-play nature allow for seamless virtual machine migration and low administrative overhead. However, Ethernet's reliance on broadcast for discovering end hosts and its use of spanning tree protocol limits its scalability and prevents the use of multiple paths within data center networks. The flat address naming scheme also leads to significant forwarding state being stored in network switches. Although IP routing mitigates many of Ethernet's shortcomings, it has also introduced a new set of problems in the data center such as high administrative overhead and complicated virtual machine migration. Although IP based multipathing techniques greatly improve overall bisection bandwidth, they are unable to meet the bandwidth demands exhibited by data center traffic patterns and make it difficult for hosts to communicate at their full interface speeds.

In this thesis, we describe the design, implementation, and evaluation of a managed network fabric tailored for data center networks. We combined the work described described in PortLand[MPF$^+$], and Hedera [AFRR$^+$] to create a scalable layer 2 network fabric. The managed network fabric preserves the positive characteristics of layer 2 Ethernet, but avoids its reliance on broadcast. In addition, the managed network fabric supports fault tolerance and multipathing for increasing overall bisection bandwidth. We define a set of principles that guide our design based on our analysis of challenges and issues facing existing data center design and implementations. By adopting the use of a centralized fabric manager, our system treats the network as a single logical unit and enables network traffic to be dynamically provisioned onto the available paths in order to maximize network link utilization. Our approach requires no modifications to end hosts and is targeted for commoditized switches. The rest of the thesis is organized as follows. In chapter 2, we describe how data centers are currently implemented and the problem that they face. We also outline the requirements and goals for designing and constructing a solution. In chapter 3, we describe an overview of relevant related work. Chapter 4 and 5 describes the design and implementation of the managed network fabric. Chapter 6 describes the evaluation performed on the system and lastly chapter 6 summarizes our conclusions and identifies future work. Appendix A includes relevant pictures, diagrams, and code listings related to the work.

Chapter 2

Background

This chapter provides an overview of how data centers are currently implemented and describes the challenges that they currently face when scaling out. Afterwards, we define a set of goals for our system to address those challenges.

The prevalent use of commoditized components along with the massive growth in cloud computing has made it advantageous and economical to construct large scale data centers consisting of thousands to hundreds of thousands of computers. These computers provide compute and storage facilities for services that are designed for high scalability, parallelization, and location transparency. Because of cost and port count limitations on a single switch, servers in data centers networks are arranged in racks and rows that are logically arranged into a single or multi-rooted hierarchical tree topology as illustrated in Figure 2.1 for scalability and fault redundancy. The tree provides the abstraction of a single large switch that interconnects all the servers. Each rack generally holds between 20-40 servers. As of 2010, all the servers in the rack are interconnected using a Top of Rack (ToR) switch that enables non-blocking bandwidth at 1 Gbps between all the servers. The ToRs themselves have 10Gbps uplinks that are connected to End of Row (EoR) switches. The EoR switches serve to interconnect a set of server racks. EoR switches are themselves interconnected to a set of Core switches. The core switches are responsible for carrying traffic between all the rows as well as traffic into and out of the data center.

Servers in a data centers can be classified into 3 types, Bare Metal, Virtu-

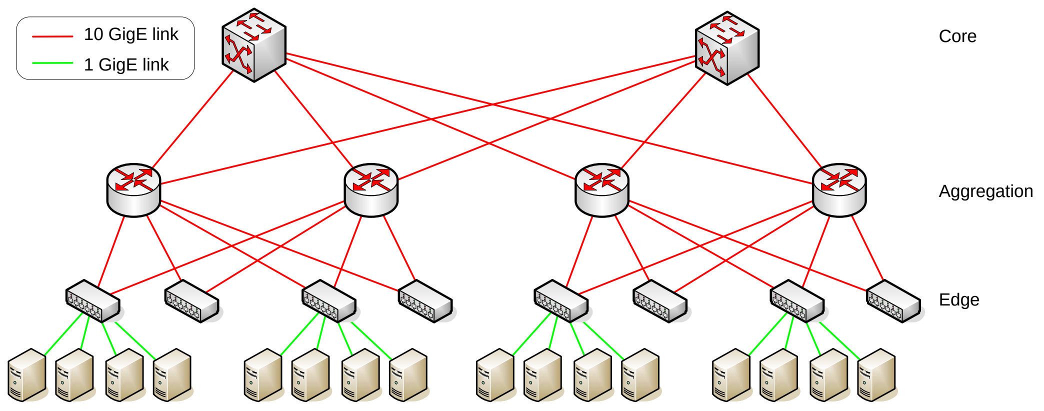


Figure 2.1: Multi-rooted Hierarchical Tree topology.

alized, and Multi-tenant. For Bare Metal, each server is directly installed with a single operating system that is configured for a particular service or application. Example applications include search processing, web server, MapReduce, and distributed storage system. For a virtualized server, a single physical server is used to host multiple virtual machines. Each virtual machine runs an installed operating system which in turn hosts a server applications. Multiple virtual machines running on a single machine enables consolidation and higher utilization of physical resources while still preserving fault isolation. A multi-tenant server is very similar to a virtualized server except that a physical server is hosting virtual machine belonging to different customers.

In the last few years, cloud computing services such as Amazon EC2, Microsoft Azure, and Google AppEngine have taken traction as an appealing platform to implement scalable, location transparent applications. Virtualization has emerged as a key technology for constructing cloud computing services by enabling the dynamic provisioning of services. A single physical server can be used to host multiple independent instances of virtual machines. Virtual machine migration is frequently used to dynamically adjust system load for cloud based services based on user demand. When service demand is high, additional virtual machines can be brought online and assigned to under utilized physical servers in order to increase system scalability. Virtual machines on highly utilized physical servers may also be migrated to less utilized and closer proximity machines to minimize network latency and avoid network congestion. During off-peak hours, virtual machines can also be deactivated and consolidated onto a smaller set of machines.

The massive number of physical hosts found in data centers along with the hosting of cloud based services and virtual machines has created many challenges for implementing large scale data center networks. For networks based on single or multi-rooted hierarchical trees, aggregate bandwidth diminishes at each higher level. For this reason, links at the aggregation and core layers of the data centers tend to be typically oversubscribed between 1:5 and 1:20. As illustrated in VL2 [GJK$^+$], data center traffic does not exhibit predictable patterns that would make static provisioning practical. The use of virtual machines to dynamically provision work among nodes in a data center can also lead to a fragmentation of services among the racks and cancels out any attempt to localize services together. This can lead to significant intra-rack traffic being exchanged that diminishes overall aggregate network bisection bandwidth.

Multi-rooted tree networks provides hosts with multiple equivalent paths to communicate and improve the potential for increased aggregate bandwidth. However, the use of existing routing and forwarding protocols within data centers limits the effectiveness of increased path parallelism. Legacy Layer 2 and layer 3 forwarding and routing protocols were designed for arbitrary topology network environments that generally have little path parallelism. For layer 2 networks, hosts on the network exist in a flat address space. This enables plug and plug connectivity and allows for minimal administrative overhead. However, since there is no TTL in the Ethernet header to prevent loops, the spanning tree protocol is used to create a single path network and eliminates any path parallelism. For large networks found in data centers, switches will also need to store significant numbers of MAC entries in their forwarding tables. The problem is exacerbated further when multiple virtual machines are deployed onto each physical machines. Perhaps the biggest problem with deploying layer 2 networks within data centers is its use of broadcast for control messages. For example, ARP performs host address resolution using broadcast queries that are in turn flooded throughout the network by all connected switches and bridges. In a data center environment, broadcast traffic can amount to a considerable amount of bandwidth being consumed. One attempt at mitigating broadcast is the use of VLANs within layer 2 networks. Although the broadcast domain is constrainted, each VLAN still uses spanning tree which results in only a single path. VLANs also fragment the network since inter-VLAN traffic will require routing which in turn can lead to bandwidth bottlenecks. Lastly, each switch that supports multiple VLANs must still store MAC forwarding addresses for hosts in all VLANs. Considering that typical switches can store between 16,000 to 32,000 MAC address entries, a multi-tenant data center consisting of a few thousand physical machines running virtual machines can quickly exceed the MAC address capacity on the switches. Since not every MAC address can be stored in a switch, there will be significant broadcast traffic when hosts ARP.

For layer 3 multi-rooted networks, Equal Cost Multipath (ECMP) is frequently used to stripe flows across available paths in order to better exploit path parallelsim. However, ECMP only statically assigns flows onto paths and does not account for current network utilization or flow size. Static assignment of flows can lead to collisions in the network which in turn leads to oversubscription and lower aggregation bandwidth. In contrast to the plug and play nature of layer 2, layer 3 networks require a significant administrative overhead for assigning IP addresses to hosts and configuring subnets for each router. A layer 3 network also complicates the migration of virtual machines between end hosts. The reason for the complication is that the IP address of the VM is tied to a particular subnet and represent both the identifier and location. Moving a virtual machine between subnets will cause existing network connections to the guest OS to be terminated. No such complication occurs for layer 2 fabrics because of Ethernet's flat address space, however broadcast scalability and the lack of multipathing is a concern when considering the large number of hosts in a data center.

Based on the shortcomings described for layer 2 and layer 3 data center implementations, we describe the following principles that our system will follow in order to realize an improved data center network fabric.

- **Single logical view of the network:** Currently in layer 3 based data centers, routers using ECMP can only make static decisions for distributing outbound traffic on parallel links. While ECMP offers improved network bandwidth, congestion can still occur when traffic from different routers are hashed onto the same path despite the existence of other free links. Ideally, we would like to have a fabric where network traffic can be dynamically placed onto links based on a single global view of the entire network instead of a limited, local view. In addition, having a single logical view enables fast fault recovery in the presence of link failures.

- **Eliminating Link Oversubscription:** Because of congestion at the aggregation and core layers of the data center, over-subscription is frequently employed. However, in order to maximize available bandwidth for demanding applications such as MapReduce, we would like to realize a fabric where oversubscription is not needed and that hosts are able to communicate with each other at full interface speeds. Avoiding oversubscription also makes it easier to exploit path redundancy for dynamically provisioning links. The traditional hierarchical tree topology suffers from decreasing bandwidth the higher traffic travels in the tree.

- **Separation of Location and Identification:** IP addresses describe both the location and unique identifier for a host. However, seamless VM migration is complicated by the existence of different subnets. Services residing in different subnets also leads to fragmentation of resources and the exchange of significant inter-rack traffic. Although Ethernet's flat addressing scheme is friendly towards VM migration, its reliance on broadcast and spanning tree limits its ability to scale for large size data centers. Because of this, we would like a naming scheme in our fabric that separates identification from location. Separating the two will enable our fabric to leverage the best of both layer 2 and layer 3 and enable service agility within the fabric.

- **Plug and Play:** Because it enables minimal administration for configuration, we would like our fabric to be based on layer 2 in order to facilitate plug and play functionality when adding new hosts. We also prefer that no modifications be made to the end hosts in order to simplify deployment. Although Layer 3 subnets can be used to minimize forwarding state stored in routers, there is a significant overhead in administering and maintaining this information for a large size data center.

- **High Scalability:** Because of the large number of hosts (physical and virtual), we would like to minimize the amount of forwarding state stored in the switches. Large size TCAMs are costly and very power hungry. As already mentioned, Ethernet's reliance on broadcasts makes it difficult to scale to data centers. Ideally, our fabric eliminates the overhead associated with broadcasts and minimizes the amount of state stored in the switches.

- **Fault Tolerant:** The system should gracefully degrade in the face of link failures. As long as there is connectivity between sender and receiver, the fabric should still be able to dynamically provision traffic based on the available paths.

2.1 Fat Tree Networks

Recent work such as A Scalable Commodity Data Center [AFLV], DCell [GWT+], Towards a Commodity Data Center, BCube [GLL+], & VL2 [GJK+] have proposed alternate topologies for a scalable data center network. In this thesis, we have constructed our data center network using a multi-staged fat tree topology. Fat trees are a type of CLOS network and have the property of being rearrangeably non-blocking. In a non-blocking CLOS switch network, there is a dedicated path from an input port on an ingress switch to a certain output port on an egress switch that doesn't conflict with another path. A rearrangeably non-blocking CLOS network differs from a non-blocking one in that establishing a non-conflicting path between two hosts may require rearranging other existing host-to-host paths. Because of its rearrangeably non-blocking property, multi-staged fat tree networks enables all to all communication between end hosts and eliminates the need for link oversubscription. The many paths available in a Fat Tree also provides path redundancy from a fault tolerance standpoint. The Fat

Tree also represents a fundamental building block that can be replicated for scaling out the network.

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. The fat tree is composed of three types of switches, core, aggregation, and edge. The fat tree itself is partitioned into pods. Each pod consists of aggregation and edge switches that enable non-blocking communication between $k^2/4$ hosts. The pod represents a building increment block that can be added to a Fat Tree for scaling out the fabric. Each core switch is connected to every pod. Figure 2.2 illustrates a k=4 fat tree network that contains 20 switches interconnecting 16 end hosts. A fat tree built from 48-port switches would consist of 48 pods and would support 27,648 hosts.

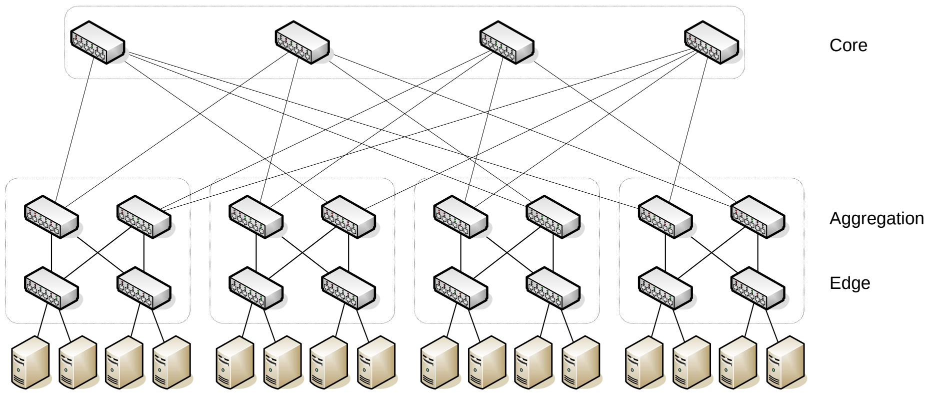


Figure 2.2: K=4 Fat Tree that supports 16 hosts.

2.2 Multipathing

Multipathing is a routing technique that leverages the existence of multiple alternate paths within a network for the purposes of improving fault tolerance and overall increased bisection bandwidth. Layer 3 networks typically use shortest path routing to forward packets between hosts. Although multiple paths may exist between senders and receivers, multipath routing is typically not used in general networks because of possible differences in MTU between routers and fluctuating link latency which could lead to fragmentation, out of order packet delivery, and packet retransmissions. In contrast, because data center networks are characterized by a fixed topology exhibiting multiple equivalent weight paths between hosts for redundancy and scalability, multipathing can dramatically increase the overall bisection bandwidth of the network.

2.2.1 Equal Cost Multipath

One popular routing method for exploiting the parallelism in the network is Equal Cost Multipath Routing (ECMP) [cis]. ECMP is a per-hop routing decision made by the router to load balance a packet onto one of many equal weight paths to a destination. In order to avoid the problem of out of order packet delivery, ECMP is typically implemented by hashing the tuples of a packet header so that packets are bucketed by flows. Each flow is then deterministically placed onto one of many multiple paths. This technique guarantees that packets within a flow arrive in order.

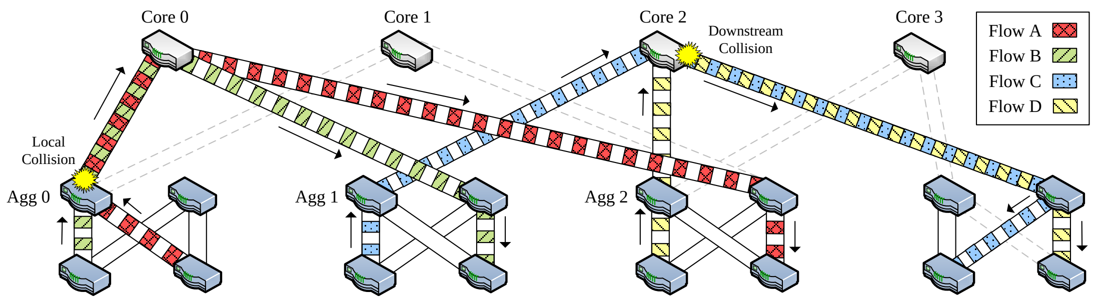


Figure 2.3: Example of ECMP collisions.

Because ECMP statically hashes flows to an egress port, it is possible for flows from different sources to intersect at a downstream router despite the existence of alternate collision free paths. In Figure 2.3, Flows A and B hash to the same aggregation switch, Agg0. Despite the existence of other free paths to their destinations, flows A and B are capped by the outbound link capacity to Core0 from Agg0 and will only transmit at half of their available interface bandwidth. Flow C and D demonstrate a case where downstream ECMP hashing results in a flow collision that was not visible from the perspective of the initial edge switches.

Flows C and D converge onto Core 2 which results in half the available interface bandwidth being transmitted by the senders. In all cases, full interface transmit bandwidth from the sending hosts could have been used if the flows were placed onto non-colliding paths.

2.2.2 Valiant Load Balancing

Another multipath routing method is Valiant Load Balancing (VLB) [GJK$^+$]. In VLB, packets are randomly assigned to one of many equal weight paths between sender and receiver. The random selection of paths ensures that packets are uniformly spread among the equivalent links. However, to avoid out of order packet delivery that would complicate protocols such as TCP, VLB can also be implemented by randomly placing flows instead of individual packets on to equivalent links.

Sections of this chapter are an adaption of the material that appears in “PortLand: A Scalable, Fault-Tolerant Layer 2 Data Center Network Fabric”. Niranjan Mysore, Radhika; Pamboris, Andreas; Farrington, Nathan Farrington; Huang, Nelson; Miri, Pardis; Radhakrishnan, Sivasankar; Subramanya, Vikram; Vahdat, Amin. In *Proceedings of ACM SIGCOMM 2009.* Sections of this chapter are also adapted from the material that appears in “Hedera: Dynamic Flow Scheduling for Data Center Networks”. Mohammad Al-Fares, Mohammad; Radhakrishnan, Sivasankar; Raghavan, Barath; Huang, Nelson; Vahdat, Amin. In *Proceedings of NSDI 2010.* The thesis author was a co-author of the listed papers.

Chapter 3

Related work

In the past few years, there has been a significant number of papers and work in the data center network space. In SEATTLE [KCR], the authors proposes a scalable layer 2 fabric for general network topologies without the overhead of broadcast. Switches in SEATTLE use a link-state protocol to learn about the topology and and form a DHT that is used for storing MAC/IP lookup information. Like this thesis, SEATTLE enables a fabric that is conducive to low administration and host mobility while avoiding the overhead associated with layer 2 broadcast protocols. However, SEATTLE does not consider multipathing.

Trill [PED$^+$09] and its implementation, RBridges, is a IETF protocol that enables a link state based routing protocol to be used at layer 2. Once connected, an RBridge learns all the hosts that are directly connected to it. RBridges in turn are interconnected with other RBridges and discover each other using a link state protocol. Layer 2 packets are encapsulated with a Trill Header that allows for shortest path packet routing. Because the Trill header features a TTL field, there is no need to prune links using Spanning Tree protocol within a RBridge network. Since equivalent shortest paths are not pruned, Trill can use ECMP based hashing techniques to multipath packets in order to increase aggregate bandwidth.

In DCell [GWT$^+$], the authors propose a specialized network structure for building a scalable data center fabric that provides fault tolerance and high bandwidth capacity. The DCell is a recursively defined network building block consisting of a set of servers interconnected with a low end switch. By connecting each server in a DCell to a different server in a different DCell, a fully connected DCell graph can be constructed. This structure in turn can be recursively repeated to construct larger networks of DCells while still exhibiting fault tolerance and path parallelism. However, because of it's unique structure, DCell requires new routing and broadcast protocols to be used. Like this thesis, DCell explores the use of a basic network building block for building a scalable network fabric, however it does not venture into techniques for exploiting the path parallelism found in the network.

In A Scalable Commodity Data Center Switch [AFLV], the authors proposed the use of a Fat tree based topology for enabling non-blocking communication between end hosts using commoditized switches. Our work in this thesis leverages the topology proposed in the paper and builds on its foundation.

In Monsoon [GLM$^+$], the authors propose a data center fabric where servers and their corresponding load balancers are interconnected using a layer-2 mesh topology. To realize a large scale, multipathed layer-2 fabric between servers, Monsoon utilizes host based valiant load balancing, a centralized MAC/IP directory, and layer-2 encapsulation features found in commodity switches.

In VL2, the authors describe their implementation for a data center fabric that is implemented using a CLOS network. Like this thesis, VL2 provides the abstraction of a single large network switch and implements a naming scheme for hosts that decouples the location from the identifier. Each hosts IP address is mapped to a location based address determined by its network edge router. By using this technique, seamless virtual machine migration is possible since the packet forwarding network is a single subnet that is decoupled from the hosts IP address identifier. A CLOS network interconnects the aggregation and core switches. Flows are uniformly spread across aggregation and core links using Valiant Load Balancing (VLB). To avoid ARP broadcasts, a end host network shim layer is introduced which forwards ARP requests to a centralized ARP directory that is replicated for redunancy and efficiency.

In Moose [SMC], the authors propose an hierarchical naming scheme that enables scalability in Ethernet networks. Like this thesis, Moose uses the idea of separating identification from location for a hosts address. In Moose, each hosts MAC address just serves as an identifier. Host packets are intercepted by the ingress Moose switch and are rewritten to use a location specific MAC address. Because the location specific mac address is a hierarchical naming scheme, shortest path routing and multipathing can be applied. Moose also requires a centralized directory for handling ARP requests intercepted at the Moose switch.

Ethane [CFP$^+$] proposes a fabric where a centralized controller is used to simplify the application of network policy to all connected switches. Switches in the Ethane fabric expose a programmatic, flow based interface that a centralized controller can use for applying policy. Packets that don't match flow entries at the switch level are forwarded to the centralized controller for classification. The central controller is omniscient and enables the fabric to be treated as a single logical unit for the purposes of applying policy. Other papers that explore the use of a centralized fabric manager for applying policy or enforcing routing include 4D [GHM$^+$], Tessaract [YMN$^+$], and RCP [CCF$^+$]. In addition, works such as the Google File System [GGL] and MapReduce [DG] also demonstrate that centralized managers are feasible and scalable for large scale systems.

Sections of this chapter are an adaption of the material that appears in "PortLand: A Scalable, Fault-Tolerant Layer 2 Data Center Network Fabric". Niranjan Mysore, Radhika; Pamboris, Andreas; Farrington, Nathan Farrington; Huang, Nelson; Miri, Pardis; Radhakrishnan, Sivasankar; Subramanya, Vikram; Vahdat, Amin. In *Proceedings of ACM SIGCOMM 2009*. The thesis author was a co-author of the listed paper.

Chapter 4

Design

In this section, we describe the design for the managed network fabric that addresses the goals described earlier. Our design builds on the work previously performed in PortLand[MPF$^+$], and Hedera [AFRR$^+$]. In conjunction with a Fat Tree network and a centralized fabric manager, we strive for a fabric where dynamically changing end host traffic can be load balanced onto multiple redundant paths while maximizing aggregate bandwidth in a fault tolerant manner.

4.1 Single Logical View

The key benefit of being able to treat the network fabric as a single logical unit is that resource allocation can be made based on a global picture. As already mentioned, switches and routers currently make packet forwarding decision based on a local view that is unaware of downstream conditions. In the case of flow placement, a global view of the fabric links can be leveraged to avoid the ECMP collisions described earlier in Figure 2.3. To realize a single logical view of the network, we use a centralized fabric manager that is connected to all the switches. Because the fabric manager can receive local switch link state information, it is able to maintain a global view of the network and can push applicable forwarding updates back to all the switches. Internally, the fabric manager maintains a matrix that describes the graph of the network and the corresponding state of each link in the network. Switches report local link failures to the fabric manager which in turn updates other connected switches. All switches maintain a local copy of the entire network. By collecting and analyzing individual flow statistics from each switch, the fabric manager can determine which links or paths are heavily utilized and can push flow path updates to the switches to direct traffic onto alternate paths.

In our system, we utilize and extend the primitives provided by OpenFlow [MAB$^+$] to realize a single logical network fabric. All switches in the fabric are connected to the central OpenFlow controller. We extended the reference OpenFlow controller to function as our fabric manager. In addition to its normal role of pushing flow classification policy to the switches, the central controller has been extended to also provide ARP resolution, fault tolerance, and flow scheduling for dynamically directing traffic onto available paths.

4.2 Eliminating Link Oversubscription

To eliminate oversubscription, three approaches are used. The first is the use of the fat tree topology that enables rearrangeably non-blocking communication between end hosts. Rearrangeably non-blocking communication enables an end host to communicate at its full interface speed with another host since there will always be some path in the network that doesn't collide with other existing traffic. However, to achieve full bisection bandwidth between hosts, it may be necessary to rearrange traffic paths.

Our second approach is to exploit the link parallelism found in the network. Although our fabric is layer 2, the absence of the spanning tree protocol and the use of our forwarding rules enables all paths to be used. For our system, our unit of communication is a network flow. All flows initially are load balanced among the available outbound links using a static ECMP algorithm. ECMP is ideal for small, short-lived flows typical of RPC requests. However, as described earlier, static ECMP hashing is inadequate for all traffic communication patterns.

For large size flows that can lead to network congestion and diminished bisection bandwidth, we employ the third approach of using a dynamic flow scheduler. Through the use of the central fabric manager, we leverage the rearrangeably non-blocking property of the fabric by dynamically shifting network flows among the available paths based on the latest link state in the network. The key component within the fabric manager for dynamically provisioning network flows is the Hedera Flow Scheduler.

4.2.1 Hedera Flow Scheduler

This section describes the Hedera Flow Scheduler at a high level. Full details can be found in [AFRR$^+$]. The basic unit of scheduling is a flow. Flows are packets that share the same 10-tuple values of <source MAC, destination MAC, source IP, destination IP, EtherType, IP Protocol, TCP source port, TCP destination port, VLAN, input port>. Table 5.1 summarizes the contents of a flow. There are two types of flow categories: network-limited and host-limited. A Network-limited flow will use all bandwidth available along its assigned path and as such is limited by the amount of congestion in the network. A host-limit flow is limited by the "slower" of the source and destination hosts. For our system, the Hedera Flow Scheduler adopts a network-limited flow model.

The basic motivation for the scheduler is that large flows are more likely to lead to packet collisions in the network. To overcome this problem, the scheduler seeks to dynamically provision large flows among the multiple paths available between a source and destination host. Initially, all new flows are treated as mice flows by the switches. Mice flows are forwarded deterministically by the switches based on a ECMP style hash created from the flows 10-tuple. However, once a flow grows past a threshold size, it is classified as a large flow and processed by the Hedera Flow Scheduler. Currently, the threshold is 10% of each hosts 1GigE link.

As illustrated in Figure 4.1, the flow scheduler performs the following steps. Edge switches detect large flows. Once a large flow is identified, the scheduler estimates the demand of each identified large flow and applies a placement algorithm for determining which path to assign to a particular large flow. Once the paths are computed, the central fabric manager installs the paths by applying flow entry updates to all the applicable switches using OpenFlow messages.

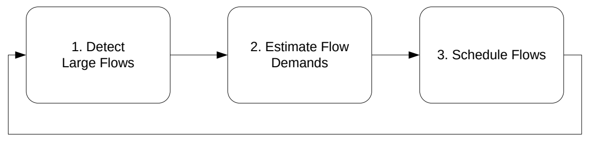


Figure 4.1: Hedera Scheduler Architecture

The input for the demand estimator is the set F of source and destination pairs for all active elephant flows. Internally, the demand estimator maintains a N x N where N is the number of hosts. For each matrix element (i,j), 3 values are stored: (1) the number of flows from host i to host j, (2) the estimated demand of each of the flows from host i to host j, and (3) a flag to indicate whether the flows between host i and host j have converged. The demand estimator performs repeated iterations of increasing the flow capacities from the sources and decreasing exceeded capacity at the receivers until the flow capacities converge. The total amount of flows sent to a receiver cannot exceed its consumption capacity. Eventually all the flows will converge. Figure 4.2 illustrates the convergence of an example demand matrix. In this example, there are 4 hosts (H0, H1, H2, and H3). H0 sends 1 flow each to H1, H2, and H3. H1 sends 2 flows to H0 and 1 flow to H2. H2 sends 1 flow to H0 and H3. H3 sends 2 flows to H1.

$$\begin{bmatrix} & (\tfrac{1}{3})_1 & (\tfrac{1}{3})_1 & (\tfrac{1}{3})_1 \\ (\tfrac{1}{3})_2 & & (\tfrac{1}{3})_1 & 0_0 \\ (\tfrac{1}{2})_1 & 0_0 & & (\tfrac{1}{2})_1 \\ 0_0 & (\tfrac{1}{2})_2 & 0_0 & \end{bmatrix} \Rightarrow \begin{bmatrix} & [\tfrac{1}{3}]_1 & (\tfrac{1}{3})_1 & (\tfrac{1}{3})_1 \\ [\tfrac{1}{3}]_2 & & (\tfrac{1}{3})_1 & 0_0 \\ [\tfrac{1}{3}]_1 & 0_0 & & (\tfrac{1}{2})_1 \\ 0_0 & [\tfrac{1}{3}]_2 & 0_0 & \end{bmatrix} \Rightarrow \begin{bmatrix} & [\tfrac{1}{3}]_1 & (\tfrac{1}{3})_1 & (\tfrac{1}{3})_1 \\ [\tfrac{1}{3}]_2 & & (\tfrac{1}{3})_1 & 0_0 \\ [\tfrac{1}{3}]_1 & 0_0 & & (\tfrac{2}{3})_1 \\ 0_0 & [\tfrac{1}{3}]_2 & 0_0 & \end{bmatrix} \Rightarrow \begin{bmatrix} & [\tfrac{1}{3}]_1 & (\tfrac{1}{3})_1 & [\tfrac{1}{3}]_1 \\ [\tfrac{1}{3}]_2 & & (\tfrac{1}{3})_1 & 0_0 \\ [\tfrac{1}{3}]_1 & 0_0 & & [\tfrac{2}{3}]_1 \\ 0_0 & [\tfrac{1}{3}]_2 & 0_0 & \end{bmatrix}$$

Figure 4.2: Demand Estimation Example

The scheduler aims to assign flows to non-conflicting paths. In addition, the scheduler also tries to place multiple flows on a link that cannot accommodate the combined bandwidth demands. Depending on the path chosen by the scheduler, the fabric manager will apply flow entries into the edge and aggregation switches of the source pod for that flow. The forwarding entries serve to direct the flow on to its new chosen path. Once a large flow ends, the flow entries applied by the fabric manager will expire after a timeout period.

Two placement algorithms, Global First Fit and Simulated Annealing are supported. In Global First Fit, the scheduler searches linearly through all possible paths that can accommodate the identified large flow. The algorithm will greedily place the flow on the first path found that can accommodate it. Once placed, a flow will be not be reassigned and will not be removed until the flow ends and the flow entries time out. Global First Fit does not guarantee that all flows can be accommodated. For it's simplicity, First Fit performs relatively well.

The second algorithm, Simulated Annealing (SA), performs a probabilistic search to compute the paths for a flow. SA searches through the solution state space to find a near optimal solution. In contrast to Global First Fit, the SA algorithm will reassign large flows based on the current state of the network. An energy function E defines the energy in the current state. For each iteration, we move to a neighboring state with a certain acceptance probability P, depending on the energies in the current and neighboring state. In each iteration, the temperature T is decreased. When the temperature is zero, the algorithm stops iterating. To simplify the problem space, all flows with the same destination host are assigned the same core switch.

The Hedera Scheduler works complementary to the static nature of ECMP. Since not all network traffic consists of large flows, ECMP is simple, fast, and sufficient for handling small, short-lived flows such as request/response RPC style traffic. However, in the presence of large flows, the Hedera Scheduler can address the short-comings of ECMP and dynamically provision the traffic to avoid downstream collisions that would lower bisection bandwidth for the fabric.

4.3 Separation of Location and Identification

In layer 3 networks, IP addresses uniquely identify a end host and describe its location based on the subnet mask. This property hampers service agility that is frequently seen in virtual machine migration. Because of the dual purpose role of the IP address, VM migration becomes complicated when moving between different subnets. Ethernet's flat address naming scheme makes migration straightforward, however its reliance on broadcast prevents it from scaling adequately to data centers. To provide the illusion of a single, scalable, flat address space, our system implements PortLands naming system for hosts that decouples location from its identification.

4.3.1 PortLand Naming Scheme

In our system, hierachical Pseudo MAC addresses (PMAC) are assigned to each edge switch host facing interface. The PMAC encodes the location of the end host in the topology. All hosts in the same pod share the same PMAC prefix. The end host interfaces are not modified and maintain their original MAC address. The IP address assigned to an end host simply serves as a unique identifier and is not used to encode its location. All forwarding between switches in the fabric is based on PMACs. Instead of maintaining forwarding state for every host in the network, switches can maintain forwarding entries based on the prefix for each pod. This minimizes the amount of state stored on the switch. When a host ARPs for a destination end host, the ARP request is intercepted by the ingress switch. The ingress switch maintains a local cache containing IP / PMAC mappings. Egress switches perform MAC rewriting on the outbound packet so that destination MAC of the intended recipient is used. Using this scheme, our fabric provides the illusion of a large Ethernet fabric to the end host.

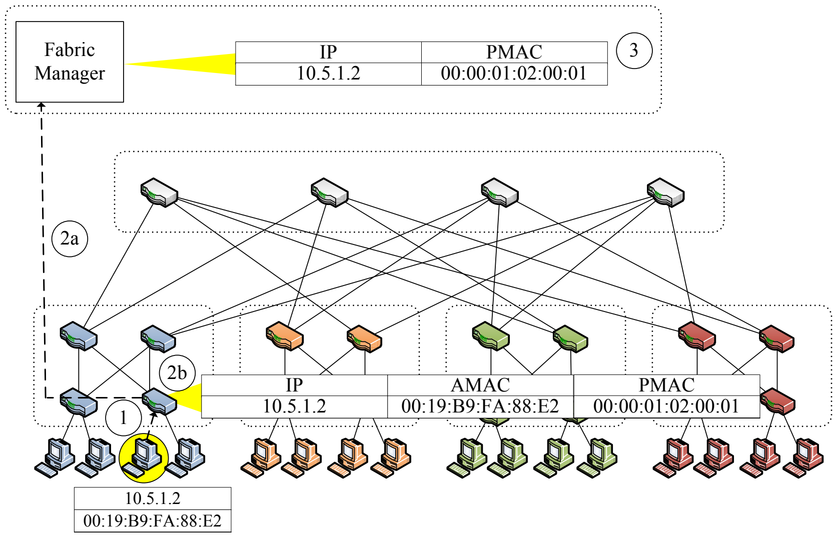


Figure 4.3: Actual MAC to Pseudo MAC Mapping.

The PMAC naming convention is defined as a 48-bit address of the following form pod.position.port.vmid. The pod (16 bits) identifies the pod that the host is located in. The position (8 bits) identifies the hosts position in the pod. The port (8 bits) matches the port number on the switch that the host is directly attached to. Lastly, vmid (16 bits) is an identifier that can be used to uniquely identify a virtual machine that is multiplexed on a single physical machine. Figure 4.3 describes an example. In the example, host 10.5.1.2 has an actual MAC (AMAC) address of 00:19:B9:FA:88:E2. However, all packets sent by 10.5.1.2 will have its source MAC address overwritten with its corresponding PMAC that describes its location in the network. In the example, the PMAC of 00:00:01:02:00:01 encodes 10.5.1.2 location as Pod 1, switch 2, switch port 0, and machine ID 1.

The benefit of this approach is that IP addresses only serve as a unique identifier for an end host. The end hosts location is identified by its corresponding PMAC address. From a host perspective, the entire fabric is treated as single flat address space. The flat address space enables virtual machine migration to be seamless and not hindered by subnet boundaries.

4.4 Plug and Play

One of the key benefits of a flat address network such as Ethernet is that it requires minimal administration. As Layer 3 networks become larger, administrative overhead from configuring routing forwarding tables and interface subnets becomes quite significant. As already described, layer 2 networks are difficult to scale to large networks such as those found in data centers because of its reliance on broadcast and the presence of large amounts of forwarding state in the switches. By utilizing the PortLand Naming scheme, our system maintains the ease of configuration found in layer 2 networks. When an ingress edge switch observes a source MAC that it has never seen before, it creates an ARP cache entry consisting of a PMAC, MAC, and IP mapping entry. This entry is consulted for ARP requests originating from the end hosts. The central fabric manager is also updated with this directory entry. To support end host mobility, ARP cache entries can be expired after a period of time.

4.5 High Scalability

One of the challenges with implementing a fabric for a data center is making it scale to support the large number of connected physical and virtual hosts. One key challenge that must be minimized is the need for broadcast or flooding. In the case of Layer-2 and as described in Seattle[KCR], ARP broadcasts can introduce significant overhead. Another key scaling challenge is minimizing the amount of state stored in the switch. Considering that an environment hosts could number in the hundreds of thousands of hosts and that switches typically store between 16,000 to 32,000 forwarding entries, it becomes apparent that existing layer 2 fabrics will have difficulty to support such mega sized data centers.

4.5.1 PortLand ARP Handling

As mentioned before, Ethernet relies on broadcasting for ARP requests. Given the large number of hosts in a data center, broadcasts introduce a significant overhead. Figure 4.4 illustrates an example of how ARP is handled by the system. In Step 1, the ingress switch intercepts the ARP requests and forwards the packet to the software layer of the switch. Each ingress switch maintains a local IP-PMAC lookup table and returns the PMAC if a matching entry for the requested IP address is found. In Step 2, if the local switch cache doesn't contain a matching entry, the ARP request is forwarded to the central fabric manager which maintains a global directory. Step 3 occurs if the central fabric manager matches a incoming ARP request and returns a PMAC response back to the original ingress switch. If the global directory in the fabric manager fails to resolve the ARP request, the ARP request is transmitted to a randomly selected core switch. The core switch then forwards the ARP request to all of its connected destination pods. Our system minimizes broadcast and flooding by leveraging the fabric manager for resolving ARP requests.

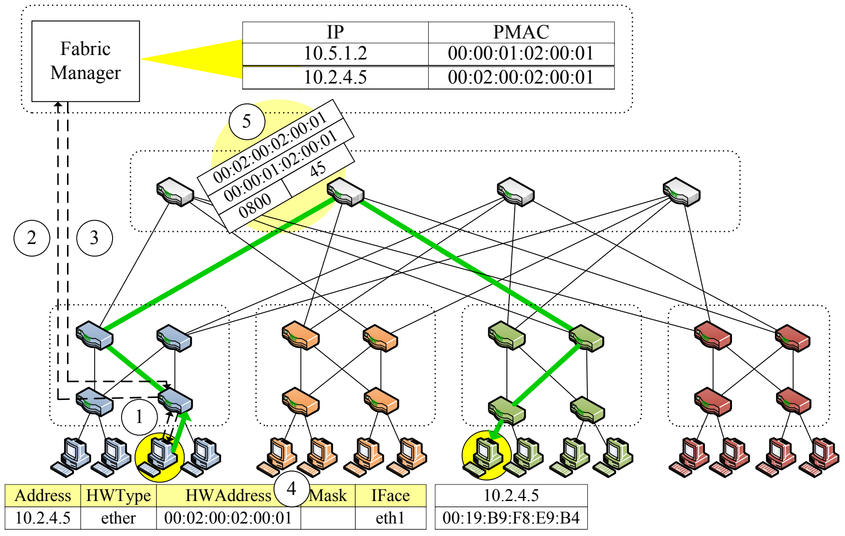


Figure 4.4: ARP Handling

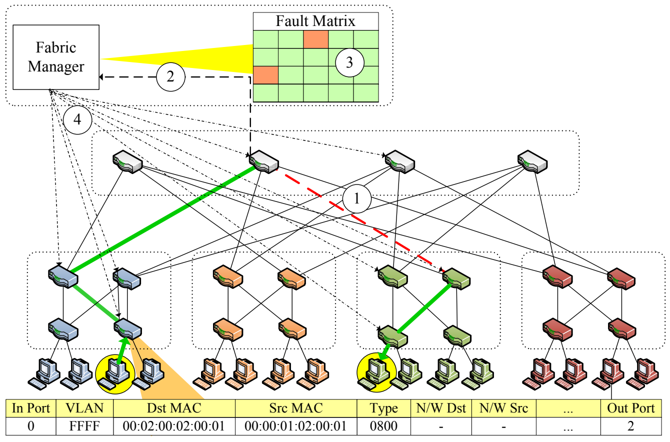
4.5.2 PortLand Forwarding

As already described, one of the scaling challenges with implementing a layer 2 fabric for a data center is coping with the large amounts of forwarding state that would need to be stored in switches for the large numbers of hosts. IP routing introduces a naming hierarchy that can reduce state, however, it also fragments the address space and complicates the delivery of agile services. As described above, this thesis utilizes the PMAC scheme to provide a separation of identification and location for end host. All forwarding entries make use of PMACs as opposed to IP based prefix / suffixes. Because the PMAC encodes location, we can program forwarding entries using the OpenFlow protocol that are hierarchical in nature.

To eliminate loops, we program the forwarding entries based on a set of rules. Packets will always be forwarded up to either an aggregation or core switch then down towards their ultimate destination. To prevent loops and broadcast storms, we ensure that once a packet begins to travel downwards towards an end host, it cannot be directed upwards anymore.

4.6 Fault Tolerant

Each switch stores a local graph representation of the entire network. Each edge represents the state of a network link between a switch. Keep-alive packets are transmitted out each switch interface on a periodic basis. If a link failure is detected, a topology update message is transmitted to the fabric manager. The fabric manager in turn will update its view of the network and unicast a topology update to all connected switches. Upon receiving a fault update, each switch will recalculate the shortest path to destination hosts and update its flow entries.



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 4.5: Fault Detection

Figure 4.5 illustrates the fault detection mechanism. In step 1, a link has failed between a core and aggregation switch. In step 2, the core switch keep-alive timer has expired. The core switch then updates its own local graph of the network and transmits a link status update to the fabric manager. The Fabric Manager in step 3 updates it's global fault matrix for the affected link and transmits an update to all the switches in step 4 over the control plane.

Sections 4.3, 4.5.1, 4.5.2, and 4.6 in this chapter are adapted from material that appears in "PortLand: A Scalable, Fault-Tolerant Layer 2 Data Center Network Fabric". Niranjan Mysore, Radhika; Pamboris, Andreas; Farrington, Nathan Farrington; Huang, Nelson; Miri, Pardis; Radhakrishnan, Sivasankar; Subramanya, Vikram; Vahdat, Amin. In *Proceedings of ACM SIGCOMM 2009*. Section 4.2.1 of this chapter is also adapted from the material that appears in "Hedera: Dynamic

Flow Scheduling for Data Center Networks". Mohammad Al-Fares, Mohammad; Radhakrishnan, Sivasankar; Raghavan, Barath; Huang, Nelson; Vahdat, Amin. In *Proceedings of NSDI 2010*. The thesis author was a co-author for both papers.

Chapter 5

Implementation

For our implementation, we constructed a small scale data center using 1U rack based computers to test our design. Our data center (Appendix A.1) consists of a data plane that is implemented using a fat tree network and a control plane network that is implemented using a standard, commercial, 48-port switch. A dedicated gateway machine serves to isolate traffic within our experimental data plane network from the rest of the campus network traffic. All end hosts and data plane switches are connected to the control plane. We implemented our managed network fabric by leveraging and extending the primitives provided by OpenFlow.

5.1 OpenFlow

This section describes OpenFlow and the functionality that it provides. The OpenFlow protocol [MAB$^+$] defines an interface by which flow entries in switches can be programmed from a centralized controller. As illustrated in Figure 5.1, the primary components of OpenFlow include the hardware flow table, secure channel, and a centralized controller. The Flow Table is a hardware module that contains a set of flow entries. Each Openflow flow entry specifies a criteria for matching a network packet and consists of the ten-tuple as described in Table 5.1. The OpenFlow datapath is simply the path a packet can travel from the hardware flow table, secure channel, and controller.

Each entry can be associated with a set of actions such as forwarding the

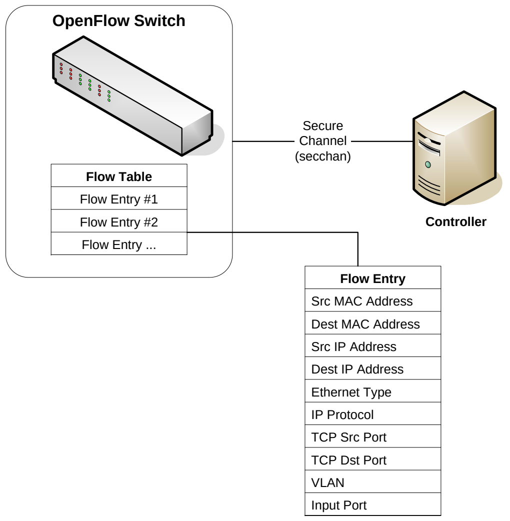


Figure 5.1: OpenFlow Architecture

Table 5.1: OpenFlow Flow Entry

Src MAC	Dest MAC	Src IP	Dst IP	Ether Type	IP Protocol	TCP / UDP Src Port	TCP / UDP Dst Port	VLAN	Input Port

packet out of a specific port, modify a packet field, or drop a packet. Because the flow entry is implemented in hardware, line rate forwarding is possible. For packets that fail to match an existing flow entry, an exception event is generated on the switch and the packet is forwarded to the controller on the control plane using Secure Channel (secchan). Secchan acts as a software packet relay between the underlying hardware flow table and the controller. The controller is a server process that can run on any standard PC and is used to classify incoming packets and apply policy updates back to the switches. According to its specification, OpenFlow switches must at the minimum support the following operations.

- Forward a flows packets to one or more ports.
- Encapsulate and forward a flows packet to a controller.
- Drop a flows packet.
- Forward a flows packet through a switch's network stack.

The protocol supports three message types. Controller-to-switch messages are initiated by the controller to directly manage or inspect the state of a switch. Asynchronous messages are initiated by the switch and are used to raise network or switch state events to the controller. Symmetric messages are initiated by either the switch or controller and are typically used for request / response type interactions between the switch and controller. For our project, we made extensive modifications to the OpenFlow primitives to implement our managed network fabric.

5.2 Constructing a Data Center

5.2.1 Dataplane

For our network data plane, we constructed a 3 tier, K=4 fat tree network as illustrated in Figure 5.2. The data plane consists of 4 pods interconnecting 16 hosts. All the pods are interconnected using 4 core switches. Each pod consists of 2 edge and 2 aggregation switches providing non-blocking traffic to 4 end hosts. Each switch is implemented using a NetFPGA PCI card [LMW$^+$]. The NetFPGA card contains 4 1GigE network ports and a Xilinx FPGA that can be programmed with custom hardware extensions. Each NetFPGA is installed in a 1U Dell Dual-Core Xeon computer running Red Hat Linux Kernel Version 2.6.18-92.1.18.el5 and the Openflow control plane software.

Each NetFPGA is programmed to act as a flow table based OpenFlow switch that can forward packets at line rate. Flow entries are stored in either the NetFPGAs 32-entry TCAM for wild card packet matches or 32KB of SRAM for exact packet matches. The kernel module for the NetFPGA provides a software interface to the underlying hardware for creating, editing, and removing flow table entries. Packets that don't match any of the hardware flow entry tables are raised as software exceptions and forwarded to the NetFPGA kernel module. The kernel module in turn forwards the packet exception to the secchan user process which in turn encapsulates the packet to the controller running in the control plane for processing.

Two key features were missing in the version of OpenFlow that we used for our implementation. The first missing feature was hardware support for MAC rewriting that would be used for mapping AMACs to PMACs as described in the previous design section, PortLand Naming Scheme. As a workaround, we modified each end host's data plane network interface so that it used a PMAC based on the end host's location in the network. The second missing feature was device driver support for programming wild cards in the source and destination MAC address flow entry fields. Prefix and suffix wild carding are currently only supported for IP address fields. As a compromise, we preprogrammed the default forwarding entries on each switch with a full PMAC forwarding entry as opposed to a wild card version originally envisioned in the design.

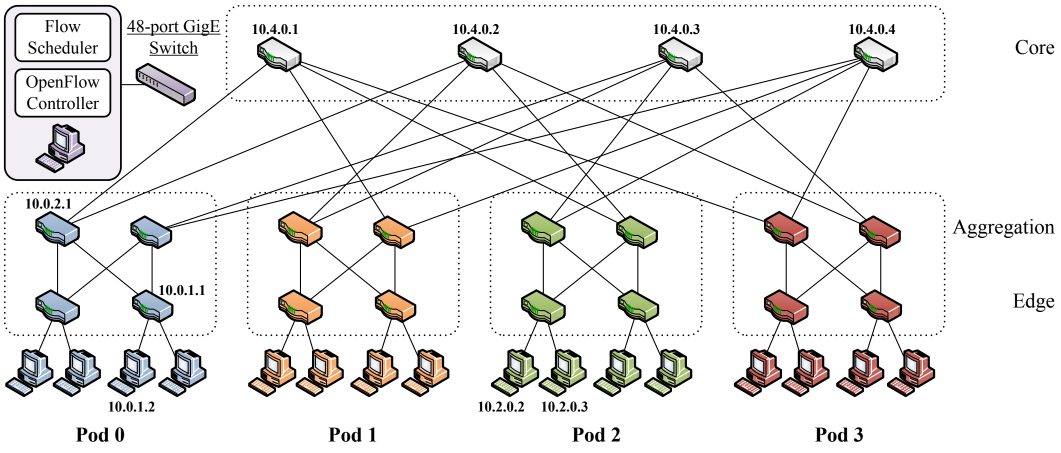


Figure 5.2: Data Center Data Plane

5.2.2 Control Plane

All switches and end hosts are connected to a Quanta 48-port 1 GigE switch to form a private 192.168.1.x control plane network as illustrated in Figure 5.3. A gateway server providing NAT, DHCP, and NFS services insulates our experimental network traffic from the rest of the campus and serves as an administrative hub and staging point for pushing software updates to our data center. A dedicated quad-core server is used for hosting OpenFlow controller processes. Because the Quanta 48-port switch enables optimal non-blocking communication between connected hosts, the control plane is also used as a baseline to compare the effectiveness of our flow scheduling techniques.

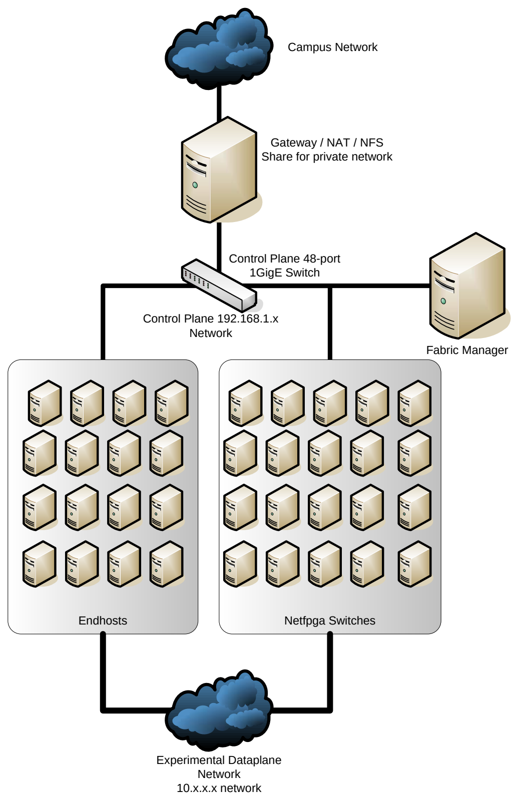

Campus Network
Gateway / NAT / NFS Share for private network
Control Plane 48-port 1GigE Switch
Fabric Manager
Control Plane 192.168.1.x Network
Endhosts
Netfpga Switches
Experimental Dataplane Network 10.x.x.x network

Figure 5.3: Data Center Control Plane

5.2.3 Hosts

As in actual data centers, all of our end hosts are 1U Quad-Core Intel Xenon mounted in racks similar to Figure A.1. A small number of end hosts were constructed using low cost commercial parts purchased over the Internet and exemplify a minimal, commoditized end host running in the giant data centers of Yahoo, Google, and Microsoft. Table 5.2 tallies the list of components and the total cost to build a low end server. All the 1U end hosts feature dual 1 GigE NIC interfaces. One interface is connected to the data plane network while the other one is connected to the control plane network. Each host runs the Linux operating system and mounts the NFS drive located on the NAT/Gateway machine. The shared NFS drive provides a convenient mechanism for pushing software updates to all the end hosts in a quick manner. End hosts are used for running the benchmarking software that we used to evaluate our managed network fabric as well as Hadoop, the parallel processing service.

Table 5.2: End Host Components 2008 Prices

1.	Intel Pentium E2220 Allendale 2.4 Ghz Dual Core Processor	\$89.99
2.	Seagate 80 GB 2.5" 7200 RPM Hard Drive	\$59.99
3.	Supermicro 1U Case with Power Supply	\$109.99
4.	Supermicro PDSBM-LN2+-O Intel Server motherboard	\$139.88
5.	Corsair 2 GB RAM	\$54.49
	Total Cost	\$454.45

5.3 Managed Network Fabric

This section describes the implementation for the managed network fabric. The reference OpenFlow code base that we used as our foundation implements a learning switch. Packets that fail to match a hardware forwarding flow entry are raised as exceptions to secchan. Secchan serves as a relay between the underlying NetFPGA kernel module and the central controller. The Central Controller implements the learning switch functionality and pushes layer 2 forwarding rules back to the switch. Both secchan and the controller are implemented using a single threaded event based model. From a data path standpoint, both secchan and the controller provide hooks for handling packet-in and packet-out events between the boundaries of each component. To implement our managed network fabric, we extended the primitives and functionality provided by the reference OpenFlow implementation by customizing the packet handling datapath and adding additional modules to support our special handling of certain incoming packets. Figure 5.4 describes the modules we added to each OpenFlow components.

To minimize the integration effort between the existing code base, most of our modules were implemented using a queue based interface as illustrated in Figure 5.6 and Figure 5.8. The use of queues such as the transmit queue found in both secchan and the controller/fabric mananger helped decouple our code from the stock Openflow codebase and served to serialize the exchange of events between the main event loop and the different threads we created for implementing our functionality. The following sections describe in detail the system implementation for flow creation, flow scheduling, ARP requests, and fault detection.

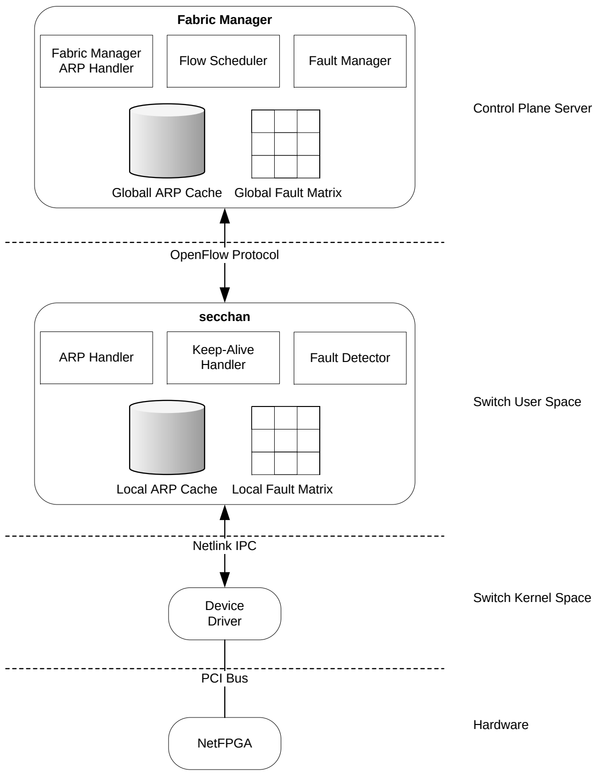


Figure 5.4: Managed Network Fabric Functional Architecture

5.3.1 Fault Detection

As described in Figure 5.4, fault detection consists of components that we have added to both the secchan and the controller process. Within secchan, we maintain a vector, *localLinks*, that corresponds to each interface on the switch. A value of 1 indicates that a packet was received on a interface. 0 indicates no activity and -1 indicates that the interface is not in use. The default data path has been modified so that whenever secchan receives an incoming packet on an interface, the corresponding slot in the *localLinks* vector is updated with the value of 1. The system interprets any packet received on an interface as evidence that a particular link is still active. Figure 5.5 illustrates the secchan data path.

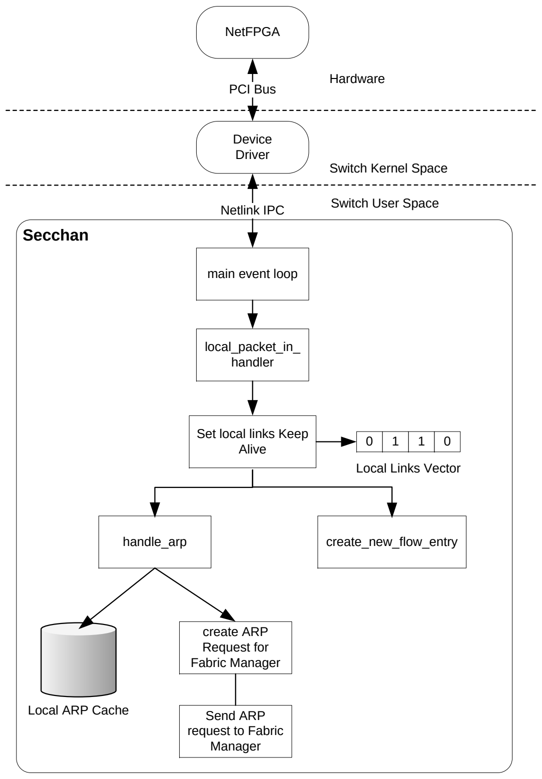


Figure 5.5: Secchan Datapath

To cover the case where no application traffic is transmitted over a link, we have modified secchan so that a background thread as illustrated in Figure 5.6 periodically transmits a *keep-alive* message on each interface that is facing another switch. The thread creates the keep alive message and serializes it to the main event loop thread for processing using the Pending Queue. The main-event loop thread then handles the packet out logic. The *keep-alive* message is simply a standard Ethernet packet as described in Listing A.1 where the source mac address is populated with the unique identifier of the sending switch for identification purposes. *Keep-alive* messages are transmitted every 40 milliseconds. To prevent a flow entry from being set in the hardware, we have added an additional packet in handler in secchan to suppress the receival of the *keep-alive* message.

To detect the failure of a link, we have added an additional fault detection thread (Figure 5.6) to secchan that periodically wakes up every 65 milliseconds

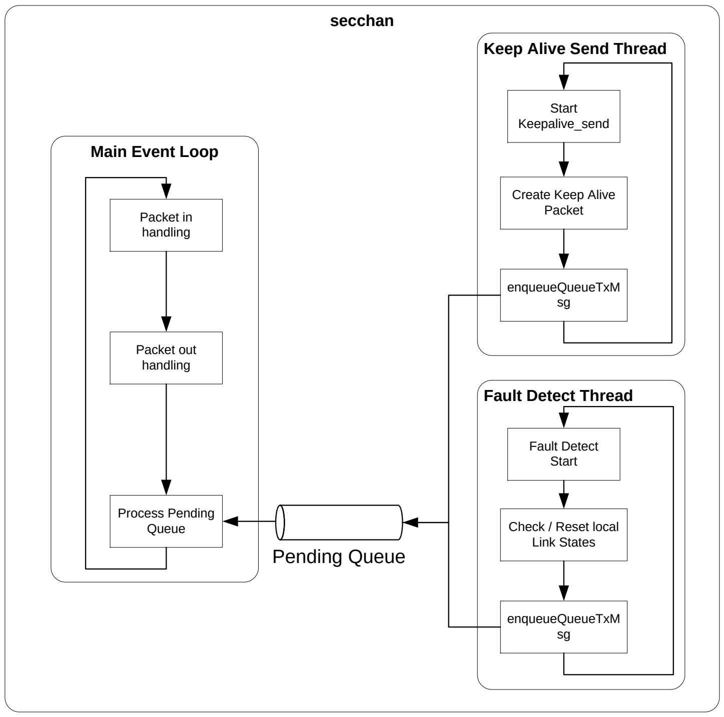


Figure 5.6: Secchan Threads

and monitors the *localLinks* vector. The fault detection inspects each valid vector element and interprets a value of 1 as evidence that a connected link is still valid. If a value of 0 is discovered within the 65 millisecond window, the fault detect thread will update the local graph matrix and transmit a *Fault Notification* message to the fabric manager. After inspecting a vector element, the fault detection thread resets the element value to 0. The *Fault Notification* message contains the graph edge as identified by the source and destination switch and the state of the edge. The message is implemented using a custom OpenFlow vendor message and is described in Listing A.5.

Upon receiving the Fault Notification message, the fabric manager updates the fault matrix and sends a Fault Detect message (Listing A.5) to all connected switches. The Fault Detect message informs each switch that a network link state has changed. Apart from the type, the format of the Fault Detect is the same as the Fault Notify message.

5.3.2 Flow Creation

By default, packets that don't match a hardware flow entry are raised as packet exceptions to secchan which in turn relays the packet to the controller for classification and processing. The Controller would then interpret the packet and respond with a policy update in the form of a flow entry action.

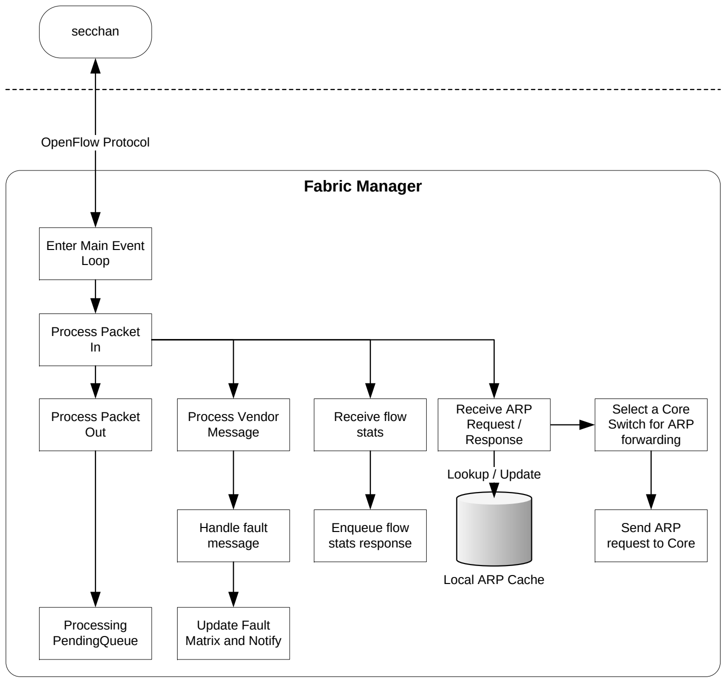


Figure 5.7: Fabric Manager Datapath

We modified the default datapath by adding packet handling logic to secchan as illustrated in Figure 5.5 so that instead of relaying a new packet to the fabric manager for classification, secchan creates a new flow entry where the action is a redirect to an upward bound interface based on a ECMP style hash. The ECMP hash is created using the packets 10-tuple. Secchan simply intercepts the packet and pushes a new Flow Entry to the underlying switch using the OpenFlow message described in Listing A.8.

5.3.3 ARP

To implement proxy ARP, we modified the secchan and controller datapath by adding additional modules. As illustrated in Figure 5.5, we added an additional

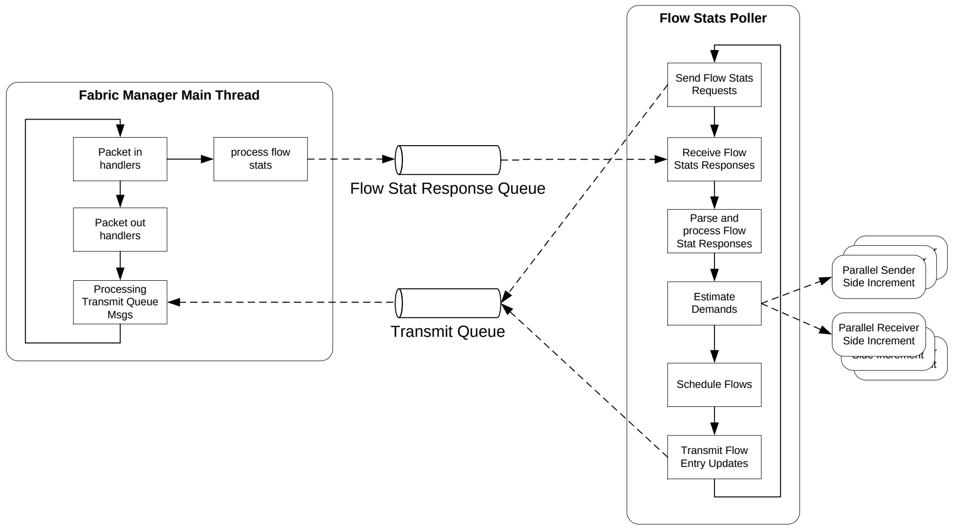

Fabric Manager Main Thread
Packet in handlers
process flow stats
Packet out handlers
Processing Transmit Queue Msgs
Flow Stat Response Queue
Transmit Queue
Flow Stats Poller
Send Flow Stats Requests
Receive Flow Stats Responses
Parse and process Flow Stat Responses
Estimate Demands
Schedule Flows
Transmit Flow Entry Updates
Parallel Sender Side Increment
Parallel Receiver Side Increment

Figure 5.8: Fabric Manager Threads

packet-in handler in secchan that intercepts ARP requests. ARP requests will always appear in secchan because we prevent a hardware flow entry from being created. Upon receiving the ARP request, secchan stores the sending hosts MAC address in the local ARP cache if it doesn't already exist and checks whether the requested IP address exists in the cache. If the requested IP can be resolved, secchan will create a ARP response and return it back to the requesting end host. Listing A.2 describes the ARP request and response packet format.

If secchan cannot resolve the ARP request locally, it encapsulates the ARP request into a custom OpenFlow vendor message and transmits it to the fabric manager. The format of the encapsulated ARP request is described in Listing A.6. As illustrated in Figure 5.7, we have modified the controllers datapath so that encapsulated ARP requests are handled. Upon receiving an encapsulated ARP request, the controller updates its global ARP cache with the requesting hosts MAC address and checks whether the requested IP address exists in the cache. If the ARP request can be resolved, the controller creates a encapsulated ARP response message as described in Listing A.6 and sends it back to the originating switch/secchan. In the case where the controller cannot resolve the ARP request, it will randomly select a connected core switch and forward the ARP request. Because the flow entries in a core switch are all downward facing, the ARP request will be forwarded to all connected pods.

When an endhost responds to ARP request, the edge switch that the host is connected to intercepts the ARP response. In our current implementation, secchan updates the local ARP cache if the responding hosts IP/MAC address does not already have a cache entry. An update is also sent to the fabric manager which allows it to also update the global ARP cache. Based on the positional pseudo MAC of the original ARP requester, the ARP response is forwarded out a specific upward facing switch interface. Eventually, the ARP response is received by the Core switch which in turn just forwards it downwards towards the applicable pod until it reaches the requesting end host.

5.3.4 Flow Scheduler

The Flow Scheduler has been implemented as a modular shared library. The Fabric Manager executes the Flow Scheduler as a background thread. The public interface exposed by the Flow Scheduler is described in Listing A.12. As described earlier in the design, the Flow Scheduler consists of 3 steps that are executed for each iteration of the thread, (1) Detection of Large Flows, (2) Estimation of Flow Demands, and (3) Scheduling of Flows. The overall implementation structure is described in Figure 5.8.

To detect whether large flows exist in Step 1, the Flow Scheduler requests Flow Stats from all connected edge switches using the OpenFlow message described in Listing **??**. The request is serialized to the main event loop for processing by using the Transmit Queue. Upon receiving the Flow Stats request message, each switch's secchan process will relay the flow stats request to the underlying hardware and eventually relay a Flow Stats Response (Listing **??**) message back to the Fabric Manager. Each OpenFlow Flow Stat Response message contains a listing of all flows currently programmed for a particular switch and information regarding each flow. Flow Stat Responses that meet or exceed the elephant flow threshold are collected up and serialized back to the Flow Scheduler using the Flow Stat Response Queue.

In Step 2, the set of flow stat responses is used to populate the demand matrix. A demand estimation for all the large flows is necessary before scheduling can occur. As described earlier in the design, each row in the matrix represents a sending host and each column represents a receiving host. Once the matrix is initiated, the demand estimator first iterates through all the senders and attempts to increase the flow capacity. After processing all the senders, the demand estimator then iterates through all the receivers and decrements flow capacity for any excess entries. This process is repeated until convergence occurs. In order to speed up processing, we parallelized the increment/decrement step for the senders and receivers by spawning a set of worker threads (Figure 5.8) and creating a thread barrier to allow for the consolidation of computed results among the worker threads. We spawn one worker thread for each available core. The amount of rows or columns that each worker thread will handle is calculated by dividing the number of hosts by the number of available cores. In the original implementation, incrementing the senders and decrementing the receivers was performed in a serial fashion.

Originally, the demand matrix was implemented as a two dimensional array. However, it was discovered while running large size simulations in Hedera that the time to allocate the array and the time to traverse the matrix that was mostly sparse was significant. For this thesis, we revised the demand matrix so that it was implemented using a custom sparse matrix data structure. The definitions for the sparse matrix are described in Listing **??**.

In Step 3, flow scheduling occurs based on the estimated flow demands generated in step 2. The flow scheduler applies one of the implemented algorithms, Global First Fit or Simulated Annealing, and generates a set of policy updates to the switches to reassign the existing elephant flows using the OpenFlow Flow Modification Message (Listing A.8). We have also integrated the fault detection into the flow placement logic. Once a core switch has been assigned for a flow, we use the fault matrix graph to check whether the path to the core is still valid. If the path to a core switch is broken, the flow scheduler will not reassign the existing flows entries.

Chapter 6

Evaluation

In this section, we evaluate our system using a set of experiments to gauge the efficiency and scalability of our implementation. We describe the experiments carried out on our system in four parts. In the first part, we measured convergence and control overhead for unicast communication in the presence of link failures. In the second part, we examine the viability of our fabric manager to support a large scale data center using projections. For the third part, we evaluate the feasibility of migrating a virtual machine within our fabric. Lastly, we measured the bisection bandwidth achieved by the system based on varying communication patterns.

6.1 Fault Detection Convergence Time

In this experiment we measured the convergence time for a UDP flow while introducing a number of random link failures. A sender transmits UDP packets at a fixed rate of 250Mbps to a receiver in a different pod. We ran 20 iterations for each link failure case and computed the average convergence time to restablish communication. Figure 6.1 plots our our results as a function of increasing numbe of randomly induced faults . Convergence time starts at approximately 65ms for a single failure and increases slowly due to the additional overhead of handling each additional failure.

We repeated the same experiment for TCP communication. We monitored network activity using wireshark at the sender while injecting a link failure along

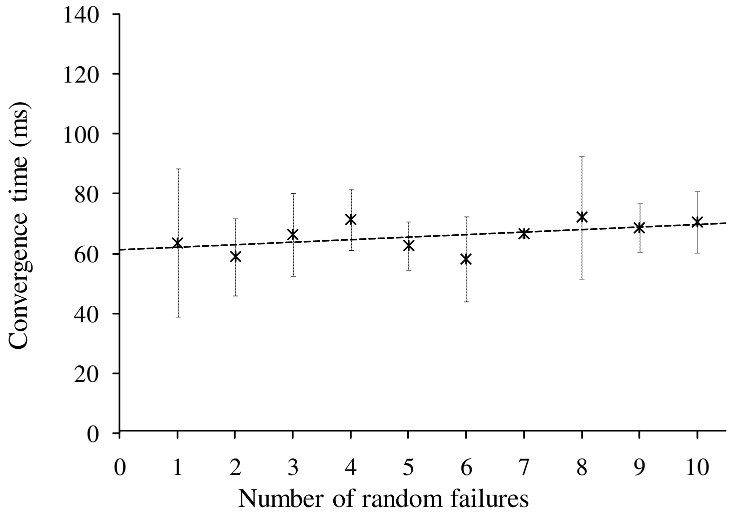


Figure 6.1: Convergence time with increasing faults.

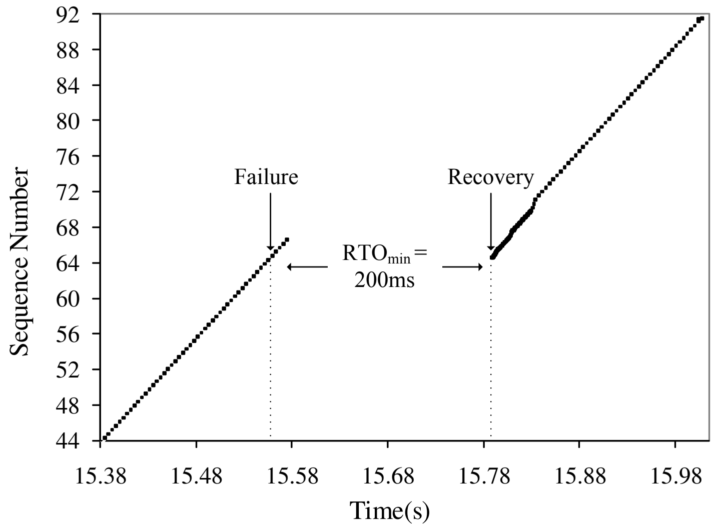


Figure 6.2: TCP Flow Fault Convergence.

the path between sender and receiver. To determine when a fault was injected, we monitored the TCP sequence numbers at the receiver to get an indication of when packets were being received. As illustrated in Figure 6.2, 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 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.

6.2 Scalability

One area of concern regarding the managed network fabric is scalability of the fabric manager for larger topologies. Because of the complexities involved with customizing secchan, we were not able to stress test the fabric manager with test clients that would simulate control plane traffic from thousands of hosts. However, we used measurements from our existing system to project the requirements for a larger network fabric.

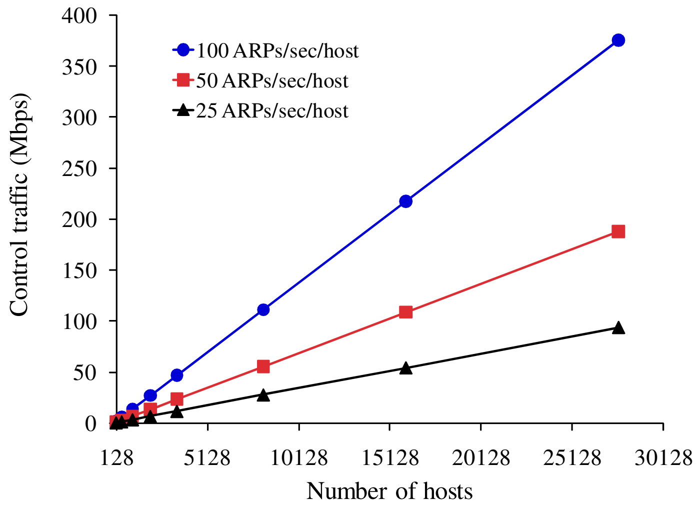


Figure 6.3: Fabric Manager Control Traffic.

Figure 6.3 describes the ARP control traffic expected by the fabric manager based on the overall number of hosts in a data center. We considered the cases where each host transmits 25, 50, and 100 ARP requests/second to the fabric manager. Since end host ARP caches maintain entries for 60 seconds, 25 ARPs/second is already an extreme condition. In a data center containing 27,648 hosts transmitting 100 ARPs per second, the fabric manager will need to manage 376Mbits/s of control traffic.

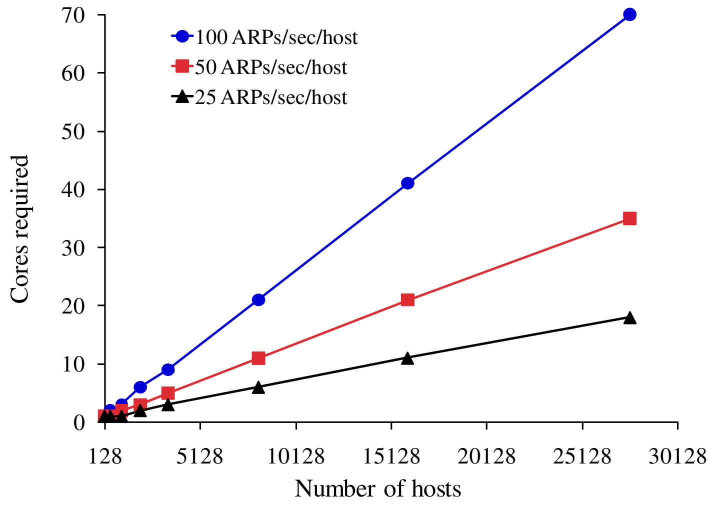


Figure 6.4: CPU Requirements for ARP requests.

Our measurements indicate that approximately 25 μs of time is required to process each ARP request/response in our non-optimized implementation. We envision that multiple fabric managers could be deployed to different cores or physical machines in order to to allow the system to scale out. Figure 6.4 describes the CPU requirements for the fabric manager as a function of the number of hosts in the data center generating different rates of ARP requests. Based on our projects, an estimated number of 70 independent cores would be required to meet the demands of the highest level of ARPs/second in a large data center.

6.3 VM Migration

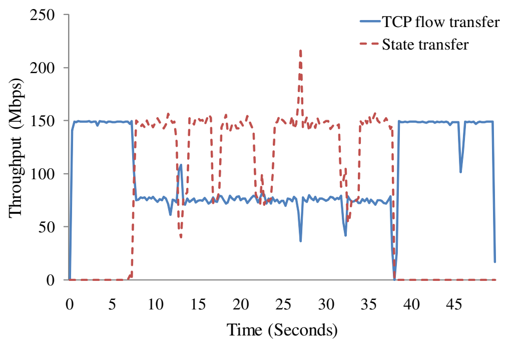


Figure 6.5: VM Migration.

Figure 6.5 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 approximately 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).

6.4 Benchmark Communication Suite

In the absence of actual network traces from production data centers, we constructed a set of communication patterns to be perform on our testbed to simulate expected network activity. In each pattern, a sender transmits a TCP based flow. The following patterns were constructed.

(1) Stride(i): A host with index x sends to the host with index $(x + i) mod(num\_hosts)$.

(2) Staggered Prob *(EdgeP, PodP)*: A host sends to another host in the

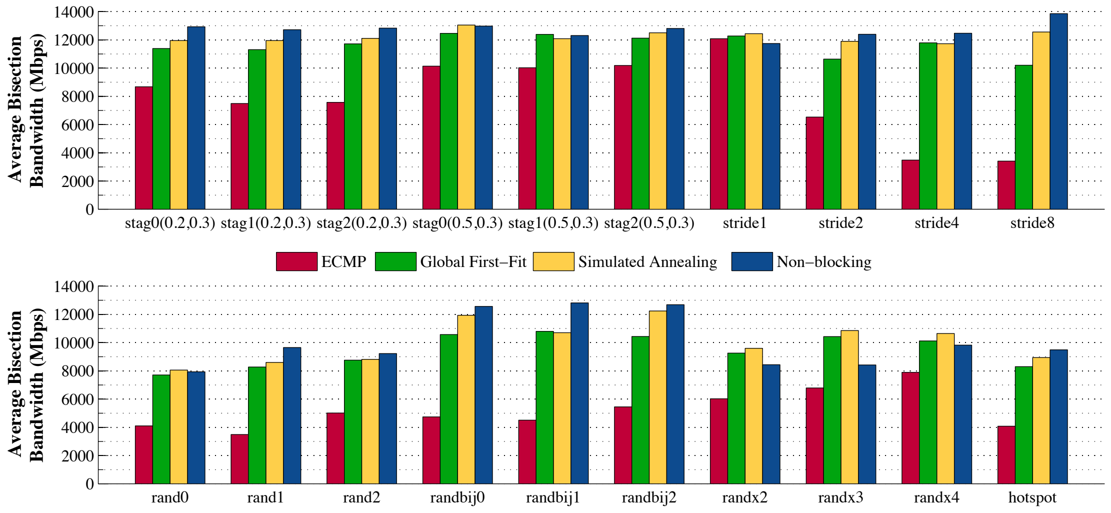


Figure 6.6: Physical testbed benchmark suite results.

same edge switch with probability *EdgeP*, and to its same pod with probability *PodP*, and to the rest of the network with probability *1-EdgeP - PodP*.

(3) Random: A host sends to any other host in the network with uniform probability. We include bijective mappings and ones where hotspots are present.

6.5 Testbed Benchmark Results

We executed the benchmark communication patterns on our testbed data plane network as follows. 16 hosts establish socket sinks for incoming traffic and measure the incoming bandwidth. Each host starts their flows in succession according to the pattern described earlier. Each pattern runs for 60 seconds and we observed the average bisection bandwidth for the middle 40 seconds.

To provide a contrast the flow scheduling techniques implemented in our system, we executed our communication patterns on the control plane network. In addition to the data plane network, all of our hosts are connected on a separate control plane network that's implemented by our non-blocking 48-port 1 GigE Ethernet switch. In addition, we also compare our flow scheduling techniques against ECMP, the current state of the art multipathing solution in layer 3 routers.

Figure 6.6 summarizes the bisection bandwidth for our benchmark results,

Table 6.1: Statistics for a 117GB Data Shuffle.

	ECMP	GFF	SA	Control
Shuffle time (s)	438.44	335.50	335.96	306.37
Host completion (s)	358.14	258.70	261.96	226.56
Bisec. BW (MB/s)	336.26	468.19	463.64	552.86
Goodput (MB/s)	24.61	32.18	32.88	38.59

ECMP, and the control plane. In all the tests, Global First Fit and Simulated Annealing outperform static hashing (ECMP) and come very close to matching the optimal bisection bandwidth (15.4 Gb/s goodput) provided by the control plane.

6.6 Data Shuffle

Another common communication pattern in data centers is the all-to-all in-memory data shuffle. A data shuffle is an operation typically exhibited by MapReduce/Hadoop operations where every host transfers a large amount of data to every other host participating in the shuffle. For our experiment, we configured each host in our testbed to sequentially transfer 500MB to every other host using TCP. In total, a 117GB suffle was performed. Table 6.1 describes the results of the shuffle. In general, the flow scheduler in our system provides a 39% improvement in bisection bandwidth over static ECMP hash-based routing.

Chapter 7

Conclusion

The emergence of large scale data centers containing tens to hundreds of thousands of physical and virtual nodes has created many challenges from a networking standpoint. Not only is there significant traffic coming in and out of data centers, there is also signficant traffic being communicated within data centers. The increasing use of applications such as MapReduce and Hadoop has seen the emergence of significant all to all communication traffic patterns within data centers. Existing data center implementations using legacy layer 2 and layer 3 protocols are hard pressed in being able to deliver high bisection bandwidth to end hosts while remaining fault tolerant. In this thesis, we have presented the design and implementation of a managed network fabric for data centers that addresses existing shortcomings in current data center implementations. The key contributions of this thesis are:

1. Construction of a functioning data center using a Fat Tree Network.
2. Working implementation of a managed network fabric system on top of the Fat Tree Network that demonstrates the viability of using a centralized fabric manager that treats the network as a single logical unit.
3. Implementation and evaluation of our design techniques that address the shortcomings of layer 2 and layer 3 protocols for constructing large size networks for data centers.

7.1 Future Work

In this section, we describe areas in which future work can be explored for our implementation.

- **Address missing functionality in OpenFlow and Netfpga:** As described earlier, the version of OpenFlow and NetFPGA that we used did not support hardware based MAC address rewriting and MAC address wild card matching in the flow table entries. The addition of these features would enable the system design to be fully realized.

- **Improve Flow Scheduling Algorithms:** Presently, the simulated annealing algorithm iterates multiple times before settling on a core switch to assign for a particular elephant flow. However, if the path to the core is broken, the scheduled results are simply discarded and the compute cycles are simply wasted. Ideally, for each iteration of the algorithm, the present state of paths to all available core switches should be considered by the algorithm when it probabilistically decides to reassigns flows from one core switch to another.

- **Explore redundancy and clustering of fabric manager:** The present implementation of secchan only allows one data path to be supported by one controller. To improve fabric manager fault tolerance and scalability, it would be ideal if an OpenFlow datapath could be modified so that multiple fabric managers can be supported.

- **Integrate location discovery protocol:** In the current implementation, all edge, aggregation, and core switches are manually configured based on the role it plays in the fabric. Ideally, a discover protocol should be used so that switches can automatically discover its role in the network.

- **Expanded Evaluation:** At the present moment, we have only evaluated the system based on our benchmark communication test suite. Although we were able to run Hadoop jobs on our managed network fabric, the end hosts lacked disk drives that could deliver enough bandwidth to cause jobs to be network bound. Ideally, for future evaluations, it would be ideal to use either disk arrays or solid state disks so that an evaluation of actual all to all communication traffic can be performed. In addition, it would also be ideal to validate that our system gracefully degrades in the presence of increasing link failures.

Appendix A

Appendix

Figure A.1: Our Experimental Data Center

A.1 Data Types and Message Formats

```
struct eth_header {
    uint8_t eth_dst[ETH_ADDR_LEN];
    uint8_t eth_src[ETH_ADDR_LEN];
    uint16_t eth_type;
} __attribute__((packed));

struct DCSwitch_KeepAlivePacket {
    struct eth_header header;
    uint8_t pad[50];
} __attribute__((packed));
```

Listing A.1: Keep Alive Packet Format

```


struct arp_eth_header {
    /* Generic members. */
    uint16_t ar_hrd;            /* Hardware type. */
    uint16_t ar_pro;            /* Protocol type. */
    uint8_t ar_hln;             /* Hardware address length. */
    uint8_t ar_pln;             /* Protocol address length. */
    uint16_t ar_op;             /* Opcode. */

    /* Ethernet+IPv4 specific members. */
    uint8_t ar_sha[ETH_ADDR_LEN]; /* Sender hardware address. */
    uint32_t ar_spa;            /* Sender protocol address. */
    uint8_t ar_tha[ETH_ADDR_LEN]; /* Target hardware address. */
    uint32_t ar_tpa;            /* Target protocol address. */
} __attribute__((packed));

typedef struct arp_packet
{
        struct eth_header eth;
        struct arp_eth_header arp;
} arp_packet;
```

Listing A.2: ARP Packet Format

```
/* Header on all OpenFlow packets. */
struct ofp_header {
    uint8_t version;   /* OFP_VERSION. */
    uint8_t type;      /* One of the OFPT_ constants. */
    uint16_t length;   /* Length including this ofp_header. */
    uint32_t xid;      /* Transaction id associated with this packet.
                          Replies use the same id as was in the
                          request to facilitate pairing. */
};

/* Vendor extension. */
struct ofp_vendor_header {
    struct ofp_header header;   /* Type OFPT_VENDOR. */
    uint32_t vendor;            /* Vendor ID:
                                 * - MSB 0: low-order bytes are
                                 *   IEEE OUI.
                                 * - MSB != 0: defined by OpenFlow
                                 *   consortium. */
    /* Vendor-defined arbitrary additional data. */
};
```

Listing A.3: OpenFlow Message Headers

```
struct DCSwitch_OFP_Msg {
    struct ofp_vendor_header vendorHeader;

    uint32_t type;
} __attribute__((packed));
```

Listing A.4: Custom OpenFlow Message Header Format

```

struct DCSwitch_OFP_Edge_Fault {
    uint32_t src_switch_index;
    uint32_t tar_switch_index;

    uint32_t edgeState;

} __attribute__((packed));

struct DCSwitch_OFP_Fault {
    struct DCSwitch_OFP_Msg dcswitchOFPMsg;

    uint32_t edge_faults_size;

    struct DCSwitch_OFP_Edge_Fault edgeFaults[0];
} __attribute__((packed));
```

Listing A.5: Fault Detection and Notification Message

```
typedef struct DCSwitch_Control_Msg {
    struct DCSwitch_OFP_Msg dcswitchOFPMsg;
    char msg[MAXMESG];
} DCSwitch_Control_Msg;
```

Listing A.6: Encapsulated ARP Request

```

/* Fields to match against flows */
struct ofp_match {
    uint32_t wildcards;         /* Wildcard fields. */
    uint16_t in_port;           /* Input switch port. */
    uint8_t dl_src[OFP_ETH_ALEN]; /* Ethernet src address. */
    uint8_t dl_dst[OFP_ETH_ALEN]; /* Ethernet dest address. */
    uint16_t dl_vlan;           /* Input VLAN. */
    uint16_t dl_type;           /* Ethernet frame type. */
    uint8_t nw_proto;           /* IP protocol. */
    uint8_t pad;                /* Align to 32-bits. */
    uint32_t nw_src;            /* IP src address. */
    uint32_t nw_dst;            /* IP dest address. */
    uint16_t tp_src;            /* TCP/UDP src port. */
    uint16_t tp_dst;            /* TCP/UDP dest port. */
};
```

Listing A.7: OpenFlow Flow Entry Match Message

```
/* Flow setup and teardown (controller -> datapath). */
struct ofp_flow_mod {
    struct ofp_header header;
    struct ofp_match match;      /* Fields to match */

    /* Flow actions. */
    uint16_t command;             /* One of OFPFC_*. */
    uint16_t idle_timeout;        /* Idle time before discarding
                                     (seconds). */
    uint16_t hard_timeout;        /* Max time before discarding
                                     (seconds). */
    uint16_t priority;            /* Priority level of flow entry.*/
    uint32_t buffer_id;           /* Buffered packet to apply to
                                     (or -1). Not meaningful for
                                     OFPFC_DELETE*. */
    uint16_t out_port;            /* For OFPFC_DELETE* commands,
                                     require matching entries to
                                     include this as an
                                     output port. A value of
                                     OFPP_NONE indicates no
                                     restriction. */
    uint8_t pad[2];               /* Align to 32-bits. */
    uint32_t reserved;            /* Reserved for future use. */
    struct ofp_action_header actions[0]; /* The action length is
                                            inferred from the
                                            length field in the
                                            header. */
};
```

Listing A.8: OpenFlow Flow Entry Modification Message

```
enum ofp_flow_mod_command {
    OFPFC_ADD,                /* New flow. */
    OFPFC_MODIFY,             /* Modify all matching flows. */
    OFPFC_MODIFY_STRICT,      /* Modify entry strictly matching
                                 wildcards */
    OFPFC_DELETE,             /* Delete all matching flows. */
    OFPFC_DELETE_STRICT,      /* Strictly match wildcards and
                                 priority. */
    OFPFC_MODIFY_ADD          /* Modify all matching flows and
                                 add a new flow if no match was
                                 found */
};
```

Listing A.9: OpenFlow Flow Entry Modification Constants

```
/* Body for ofp_stats_request of type OFPST_FLOW. */
struct ofp_flow_stats_request {
    struct ofp_match match;   /* Fields to match */
    uint8_t table_id;         /* ID of table to read
                                 (from ofp_table_stats)
                                 or 0xff for all tables. */
    uint8_t pad;              /* Align to 32 bits. */
    uint16_t out_port;        /* Require matching entries
                                 to include this as an output port.
                                 A value of OFPP_NONE indicates no
                                 restriction. */
};
OFP_ASSERT(sizeof(struct ofp_flow_stats_request) == 40);
```

Listing A.10: OpenFlow Flow Stats Request Message

```
/* Body of reply to OFPST_FLOW request. */
struct ofp_flow_stats {
    uint16_t length;          /* Length of this entry. */
    uint8_t table_id;         /* ID of table flow came from. */
    uint8_t pad;
    struct ofp_match match;   /* Description of fields. */
    uint32_t duration;        /* Time flow has been alive in
                                 seconds. */
    uint16_t priority;        /* Priority of the entry. Only
                                 meaningful when this is not an
                                 exact-match entry. */
    uint16_t idle_timeout;    /* Number of seconds idle before
                                 expiration. */
    uint16_t hard_timeout;    /* Number of seconds before
                                 expiration. */
    uint16_t pad2[3];         /* Pad to 64 bits. */
    uint64_t packet_count;    /* Number of packets in flow. */
    uint64_t byte_count;      /* Number of bytes in flow. */
    struct ofp_action_header actions[0]; /* Actions. */
};
OFP_ASSERT(sizeof(struct ofp_flow_stats) == 72);
```

Listing A.11: OpenFlow Flow Stats Response Message

```

struct flow_list*
estimate_demands(struct flow_list *list_of_flows,
                 const int k);

struct flow_list*
schedule_flows(struct flow_list *list_of_flows,
               const int k, const int scheduler);
```

Listing A.12: Flow Scheduler Interface

```
struct host_demand_node_optimized {
    float demand;              // demand per flow
    uint8_t flag;
    uint8_t receiver_limited;  // Used and reset in
                               // receiver_side_decrement

    uint16_t num_flows;        // number of flows between
                               // the source and destination pair
};
```

Listing A.13: Demand Estimator End Host Data Structure Definitions

```
/**
 * Represents an individual cell in the matrix
 */
struct matrix_node_optimized {
    /**
     * row, column coordinates
     */
    uint32_t src_host;
    uint32_t dst_host;

    /**
     * value
     */
    struct host_demand_node_optimized nodeValue;

    /* handle for row - hash table indexed by dst_host */
    UT_hash_handle rr;

    /* handle for column - hash table indexed by src_host */
    UT_hash_handle cc;
};
```

Listing A.14: Sparse Matrix Node Data Structure Definition

```
/**
 * matrix data structure
 */
struct matrix_optimized {
    /**
     * All the rows of the matrix
     *
     * An array of hash tables - one for each row (src_host)
     * of the matrix
     * Memory for the array must be allocated when the matrix
     * is created.
     * Each hash table is a pointer to the first element in
     * the uthash hash table allows all column elements for a
     * row to be searched
     */
    struct matrix_node_optimized **rows;

    /**
     * All the columns of the matrix
     * An array of hash tables - one for each colum (dst_host)
     * of the matrix
     * Memory for the array must be allocated when the matrix
     * is created
     * Each hash table is a pointer to the first element in the
     * uthash hash table
     * Allows all row elements for a column to be searched
     */
    struct matrix_node_optimized **columns;

    /**
     * # of matrix cells populated in the matrix
     */
    int numElements;
};
```

Listing A.15: Sparse Matrix Data Structure Definition

```
struct matrix_optimized*
matrix_new_optimized(const int k);

void
matrix_delete_optimized(struct matrix_optimized*,
                        const int k);

/**
 * Returns NULL if matrix node doesn't exist
 */
struct matrix_node_optimized*
matrix_get_node_optimized(struct matrix_optimized *matrix,
                          uint32_t srcHost,
                          uint32_t dstHost);

/**
 * Returns a pointer to the new entry after inserting the new entry
 */
struct matrix_node_optimized*
matrix_insert_node_optimized(struct matrix_optimized* matrix,
                             struct flow_node *aFlowNode);
```

Listing A.16: Sparse Matrix Interface Definitions

Bibliography

[AFLV] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A Scalable, Commodity Data Center Network Architecture. In *Proceedings of ACM SIGCOMM, 2008.*

[AFRR$^+$] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *Proceedings of NSDI, 2010.*

[Bor] D. Borthakur. The Hadoop Distributed File System. http://hadoop.apache.org/core/docs/current/hdfs_design.pdf.

[Car] A. Carter. Do It Green: Media Interview with Michael Manos, 2007. http://edge.technet.com/Media/Doing-IT-Green.

[CCF$^+$] Matthew Caesar, Donald Caldwell, Nick Feamster, Jennifer Rexford, Aman Shaikh, and Jacobus van der Merwe. Design and Implementation of a Routing Control Platform. In *Proceedings of NSDI, 2005.*

[CFP$^+$] Martin Casado, Michael J. Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. Ethane: Taking Control of the Enterprise. In *Proceedings of ACM SIGCOMM, 2007.*

[cis] Cisco Data Center Infrastructure 2.5 Design Guide. http://www.cisco.com/univercd/cc/td/doc/solution/dcidg21.pdf.

[DG] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of OSDI, 2004.*

[GGL] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Proceedings of SOSP, 2003.*

[GHM$^+$] Albert Greenberg, Gisli Hjalmtysson, David A. Maltz, Andy Myers, Jennifer Rexford, Geoffrey Xie, Hong Yan, Jibin Zhan, and Hui Zhang. A Clean Slate 4D Approach to Network Control and Management. *ACM SIGCOMM CCR, 2005.*

[GJK$^+$] Albert Greenberg, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David Maltz, Parveen Patel, and Sudipta Sengupta. VL2: A Scalable and Flexible Data Center Network. In *Proceedings of ACM SIGCOMM, 2009.*

[GLL$^+$] Chuanxiong Guo, Guohan Lu, Dan Li, Haitao Wu, Xuan Zhang, Yunfeng Shi, Chen Tian, Yongguang Zhang, and Songwu Lu. BCube: A High Performance, Server-Centric Network Architecture for Modular Data Centers. In *Proceedings of ACM SIGCOMM, 2009.*

[GLM$^+$] Albert Greenberg, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. Towards a Next Generation Data Center Architecture: Scalability and Commoditization. In *Proceedings of ACM PRESTO, 2008.*

[GWT$^+$] Chuanxiong Guo, Haitao Wu, Kun Tan, Lei Shi, Yongguang Zhang, and Songwu Lu. DCell: A Scalable and Fault-Tolerant Network Structure for Data Centers. In *Proceedings of ACM SIGCOMM, 2008.*

[had] Apache Hadoop Project. http://hadoop.apache.org.

[Hof08] Ty Hoff. Google Architecture. http://highscalability.com/google-architecture, November 2008.

[IBY$^+$] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed Data-parallel Programs from Sequential Building Blocks. In *Proceedings of ACM EuroSys, 2007.*

[KCR] Changhoon Kim, Matthew Caesar, and Jennifer Rexford. Floodless in SEATTLE: A Scalable Ethernet Architecture for Large Enterprises. In *Proceedings of ACM SIGCOMM, 2008.*

[LMW$^+$] John W. Lockwood, Nick McKeown, Greg Watson, Glen Gibb, Paul Hartke, Jad Naous, Ramanan Raghuraman, and Jianying Luo. NetFPGA–An Open Platform for Gigabit-rate Network Switching and Routing. In *Proceedings of IEEE MSE, 2007.*

[MAB$^+$] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling Innovation in Campus Networks. *ACM SIGCOMM CCR, 2008.*

[MPF$^+$] Radhika Niranjan Mysore, Andreas Pamporis, Nathan Farrington, Nelson Huang, Pardis Miri, Sivasankar Radhakrishnan, Vikram Subramanya, and Amin Vahdat. PortLand: A Scalable, Fault-Tolerant Layer 2 Data Center Network Fabric. In *Proceedings of ACM SIGCOMM, 2009.*

[PED$^+$09] Radia Perlman, Donald Eastlake, Dinesh G. Dutt, Silvano Gai, and Anoop Ghanwani. Rbridges: Base Protocol Specification. Technical report, Internet Engineering Task Force, 2009.

[Rab] L. Rabbe. Powering the Yahoo! Network. http://ycorpblog.com/2006/11/27/powering-the-yahoo-network.

[Sha08] Stephen Shankland. Google Spotlights Data Center Inner Workings. http://news.cnet.com/8301-10784_3-9955184-7.html, May 2008.

[SMC] Malcolm Scott, Andrew Moore, and Jon Crowcroft. Addressing the Scalability of Ethernet with MOOSE. In *EuroSys Poster session, 2008.*

[YMN$^+$] Hong Yan, David A. Maltz, T. S. Eugene Ng, Hemant Gogineni, Hui Zhang, and Zheng Cai. Tesseract: A 4D Network Control Plane. In *Proceedings of NSDI, 2007.*