UC San Diego
UC San Diego Electronic Theses and Dissertations

Title
LDP : location discovery protocol for data center networks

Permalink
https://escholarship.org/uc/item/92g7h0wq

Author
Pamboris, Andreas

Publication Date
2009

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA, SAN DIEGO

LDP: Location Discovery Protocol for Data Center Networks

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

in

Computer Science

by

Andreas Pamboris

Committee in charge:

Professor Amin Vahdat, Chair
Professor George Varghese
Professor Geoffrey M. Voelker

2009

Copyright
Andreas Pamboris, 2009
All rights reserved.

The Thesis of Andreas Pamboris is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

Chair

University of California, San Diego

2009

EPIGRAPH

We are defined by opportunities, even the ones we miss.

TABLE OF CONTENTS

Signature Page . . . iii

Epigraph . . . iv

Table of Contents . . . v

List of Figures . . . vii

Acknowledgements . . . viii

Abstract of the Thesis . . . ix

Chapter 1 Introduction . . . 1

Chapter 2 Background . . . 4
- 2.1 A Layer 3 approach . . . 4
- 2.2 A Layer 2 approach . . . 5
- 2.3 A Virtual LAN approach . . . 6
- 2.4 The Locator/Identifier Separation Protocol (LISP) . . . 7

Chapter 3 LDP Design . . . 9
- 3.1 Data center network topology . . . 9
- 3.2 Defining position . . . 10
- 3.3 High level design . . . 12
  - 3.3.1 Fat tree specific LDP . . . 12
  - 3.3.2 LDP for general multi-rooted trees . . . 16

Chapter 4 Failure Analysis . . . 23
- 4.1 Assumptions . . . 24
- 4.2 Software switch failures . . . 24
  - 4.2.1 Switch failures restricted to one level of the fat tree 25
  - 4.2.2 Switch failures across multiple levels of the fat tree 27
- 4.3 Miss wiring conditions . . . 28

Chapter 5 Implementation . . . 35
- 5.1 Simulator . . . 35
- 5.2 Testbed Implementation . . . 36
  - 5.2.1 Testbed Description . . . 36
  - 5.2.2 Modifications to the OpenFLow NetFPGA software framework . . . 37

Chapter 6 Evaluation 40
6.1 Simulation 40
6.2 Testbed Implementation 42

Chapter 7 Conclusions 45

Bibliography 47

LIST OF FIGURES

Figure 3.1: Fat tree built out of 4-port switches. 10

Figure 4.1: Core switch believing it is an aggregation switch. 28
Figure 4.2: Edge switch believing it is an aggregation switch. 30
Figure 4.3: Edge switch believing it is a core switch. 30
Figure 4.4: Aggregation switch believing it is an edge switch. 31
Figure 4.5: Aggregation switch believing it is a core switch. 32
Figure 4.6: Core switch believing it is an edge switch. 32
Figure 4.7: Core switch believing it is an aggregation switch. 33

Figure 5.1: System architecture. 38

Figure 6.1: Number of pods created Vs k 40
Figure 6.2: Per pod distribution of switches for k=48 41
Figure 6.3: Number of pods created in a fat tree with k=4 over 50 runs of LDP 43
Figure 6.4: LDP convergence time 43

ACKNOWLEDGEMENTS

Firstly I would like to acknowledge Professor Amin Vahdat for his support as my supervisor and the chair of my committee. His guidance and insightful advices have been the primary ingredients leading me to where i am today.

I would also like to acknowledge all the members of the DC Switch group at the University of California, San Diego. I am very proud for being a member of this group during the past academic year. It has been a great pleasure having to work closely with them towards the completion of my Masters degree.

Further, I would like to acknowledge Professors George Varghese and Geoffrey M. Voelker for their guidance and for serving in my thesis committee.

Finally, I would like to acknowledge the Fulbright organization and my family for providing me with the financial support during my graduate studies.

ABSTRACT OF THE THESIS

LDP: Location Discovery Protocol for Data Center Networks

by

Andreas Pamboris
Master of Science in Computer Science

University of California San Diego, 2009

Professor Amin Vahdat, Chair

Data center architectures are gaining more and more interest nowadays due to the emerging need for hosting tens of thousands of workstations with significant aggregate bandwidth requirements. The underlying network architecture of a data center typically consists of a tree of routing and switching elements that count to thousands of components for the large-scale data centers that are deployed today. Recent research activity has been focusing on many of the challenges that one is faced with on building such large in scale data centers, such as scalability issues, dealing with the possibly inherent non-uniformity of the bandwidth among data center nodes, minimizing the actual cost of building and deploying these data centers, reducing the cabling complexity involved with interconnecting the routing and switching elements that a data center is composed of and coming up with efficient forwarding and routing mechanisms to be used within the data center.

In this context, many solutions to efficient forwarding and routing have been proposed recently that take advantage of a switch's relevant location in the global topology. This study focuses on a specific challenge regarding data center architectures, which is that of providing a plug and play functionality to a data center's switches so that no such switch will require any kind of human intervention or physical device configuration for figuring out it's location in the underlying network architecture. We present a scalable, distributed and efficient Location Discovery Protocol for data center network topologies (LDP). Our solution is generic, allow-

ing for any kind of multi-rooted tree architecture to be used when interconnecting the switching elements that constitute the core of a data center network.

Chapter 1

Introduction

Nowadays data centers that are spread across the Internet are widely used for the purposes of a number of applications, heavy computation and storage. The benefits from commodities of scale are leading to the emergence of large data centers that are required to host applications running on tens of thousands of end hosts. However significant application networking requirements arise across all such cases and attempts to effectively address these requirements have been evident in recent research activity.

It is expected that in the near future a substantial portion of Internet communication will take place within data center networks. Although these networks tend to be highly engineered with a number of common design elements, the routing/forwarding and management protocols that we run in data centers are proving inadequate along a number of dimensions since they were initially designed for the general Internet setting.

Here we address a specific challenge for large in scale data center network designers which is that of being able to perform routing and forwarding effectively and efficiently in such environments. Currently, when considering one's options for performing forwarding in data center networks it all comes down to the high-level dichotomy of creating either a Layer 2 or a Layer 3 network fabric, each with associated tradeoffs (discussed in Chapter 2).

Ideally, one would like to combine the goods of both worlds by using a hybrid solution between the two. However, getting rid of the necessity for using

hierarchically assigned IP addresses (for all the reasons discussed below) and in the same time being able to maintain efficiency and scalability, two aspects sacrificed when using raw MAC addresses as the basis for performing forwarding in a Layer 2 domain, are two conflicting tasks. Any such solution should aim on taking advantage of the fact that modern data centers are interconnected through a hierarchy of switches with good connectivity among racks. A switch's knowledge of its relevant location in the data center would allow for a more efficient forwarding mechanism to be used, based on a special encoding of the destination's location in the data center.

In order to achieve our goal there is a need to separate host identity from host location. One can use either MAC addresses or IP addresses in a data center as the basis for communication (host identity), though typically IP addresses which are the basis of naming with TCP/UDP sockets are used. Modern switch hardware allows for transparent rewriting of MAC address headers for packets. We will leverage this ability to introduce hierarchical, location-dependent pseudo MAC addresses which will serve as a host's location and thus will be used as the basis of forwarding in data center networks.

In order for this to be feasible without enduring the fallacies of a Layer 3 solution, the existence of an efficient, scalable and distributed location discovery mechanism that would allow a switch to figure out its location in the global topology is required. This information would then be transformed into location-dependent pseudo MAC addresses that would be dynamically assigned to a host upon connecting to a specific port of an edge switch within the data center. Allowing IP addresses to serve as identifiers rather than locators would also facilitate persistent connectivity even among mobile communicating or frequently live migrating hosts within a data center network.

This MS thesis focuses on LDP, an efficient, distributed Location Discovery Protocol that allows all of the edge switches in a multi-rooted tree topology discover their relevant location within a data center. LDP makes the design of new routing and forwarding solutions for large-scale data centers much more flexible by reducing the need for manually configuring a switch's position via error-prone administrative

intervention and thus providing a plug and play functionality to a data center's switch.

The rest of this document is divided as follows. In Chapter 2 we discuss the background work on routing and forwarding in data center networks. We present the design of LDP and a preliminary failure case analysis in Chapters 3 and 4 respectively. Details regarding our implementation are described in Chapter 5. We further present the results we have obtained after testing LDP on our evaluation platform and running appropriate simulations in Chapter 6. Finally we end this report with our conclusions in Chapter 7, where we summarize our contribution and identify future work.

Chapter 2

Background

As we have previously mentioned in our introductory section, so far there has been a clear distinction between using a Layer 2 or a Layer 3 approach to routing and forwarding in data center networks. Both approaches require no specialized protocol for figuring out a switch's location within the global network topology. However each approach has its pros and cons, which have led to an ongoing debate on whether or not one should prefer one over the other. These are described below, along with some hybrid solutions that were proposed in the past.

2.1 A Layer 3 approach

A Layer 3 approach traditionally assigns IP addresses to hosts hierarchically based on the switch they are directly connected to. By carefully assigning these IP addresses relatively small forwarding tables are formed across all of data center's switches. Efficient routing can be achieved by employing standard intra-domain routing protocols such as the link state protocol, OSPF [6]. What is more certain OSPF extensions such as Equal Cost Multipath [3] can be used to allow load balancing among available equal cost paths that can be used between any pair of communicating hosts, which is a rather common case when considering network topologies that are used in today's large-scale data centers. It is well known that these large in scale network topologies are quite vulnerable to network failures, i.e. either failed links or switches. This issue is also handled by the OSPF protocol

which is able to detect such failures by means of "Hello" packets that are sent by each switch on all of their ports periodically. Reception of these packets basically serve as proof that the corresponding neighbor and/or link are still alive in a sense. Once a failure is detected by the inability to receive a corresponding "Hello" packet on a specific port for some amount of time, the switch that detects it broadcasts this information among all switches in the routing domain which then adjust their forwarding table entries accordingly. Lastly, transient loops with Layer 3 forwarding is less of an issue because the IP-layer TTL/hop count limits per-packet resource consumption while forwarding tables are being asynchronously updated.

At a first glance one could falsely come to the conclusion that a Layer 3 approach would suffice for the routing and forwarding requirements of a data center. However a closer look to any such solution comes to identify important weaknesses that are significantly limiting to the effectiveness and efficiency of the normal operation of a data center. One thing to account for is the administrative burden imposed. Layer 3 forwarding does impose a significant administrative overhead to appropriately assign IP addresses, set subnet identifiers on a per switch basis, synchronize DHCP server state with subnet identifiers, etc. What is more, end host virtualization has gained a lot of importance over the years for many reasons such as being able to facilitate load balancing, planned upgrades, or energy/thermal management. Hence every so often virtual machines may be migrated from one physical machine to another, requiring assignment of a new IP address based on the identifier of the new first-hop switch. Such an operation would for example break all open TCP connections to the migrating host, require invalidation of any session state maintained across the data center, etc, which makes a Layer 3 solution less attractive to the eyes of a data center architecture designer.

2.2 A Layer 2 approach

Due to the limitations imposed by a Layer 3 approach, as these are depicted above, data center designers have also considered alternative approaches to

forwarding such as deploying a Layer 2 network, thus performing forwarding based on at MAC addresses. The primary advantage gained by following this approach is that the administrative overhead involved is minimal compared to Layer 3 forwarding. No kind of physical device configuration is required thus providing for a "plug and play" functionality much desired by data center administrators.

However other important issues arise by the use of a Layer 2 fabric that need to be dealt with. For starters, one important challenge that one must account for is that standard Ethernet bridging [8] does not scale to networks with tens of thousands of hosts because of the need to support broadcast across the entire fabric. What is more, performance-wise, the presence of a single forwarding spanning tree would severely limit performance in topologies that consist of multiple available equal cost paths, even if this is optimally designed. This may very well be considered as the main drawback of following such an approach since efficiency is of critical importance for the effective operation of a large scale data center.

2.3 A Virtual LAN approach

As one would expect some attempts have tried to combine the above two approaches. One such middle ground between a Layer 2 and Layer 3 fabric is that of employing Virtual LANs (VLANs) to allow a single logical Layer 2 fabric to cross multiple switch boundaries. A Virtual LAN is essentially a group of hosts that communicate as if they were attached to the broadcast domain, regardless of their physical location. VLANs have the same attributes as physical LANs, however they also allow for end stations to be grouped together even if they are not connected to the same network switch.

Although this approach has proven to be feasible for smaller-scale topologies, a number of drawbacks associated with its use have also been identified. Flexibility for one is sacrificed since resources must be explicitly assigned to each VLAN at each participating switch. Thus dynamically changing communication patterns would impose a great overhead to the use of VLANs if efficiency were to be maintained. Although one could claim that flexibility with respect to VM mi-

gration increases with VLANs, an inherent limitation of this approach is the fact that migration is confined to target physical machines that belong to the same VLAN. What is more the amount of state that must be maintained by each switch in a VLAN is significant, accounting for all hosts in that VLAN. Finally, VLANs also use a single forwarding spanning tree, just as in the case of a Layer 2 fabric, which as we have explained previously is limiting to the performance of a data center.

2.4 The Locator/Identifier Separation Protocol (LISP)

The Locator/Identifier Separation Protocol by David Meyer, Cisco Systems, is perhaps the most related to our work approach. Although it was not especially designed to facilitate data center networks, one can very easily infer its direct applicability to data center-like architectures. Designed to address issues regarding the scalability of the routing system as well as the impending exhaustion of the IPv4 address space for the Internet, this protocol's design principle is to separate the *locator* (describing how a device is attached to the network) and the *identifier* (uniquely identifying the device) in the numbering of Internet devices, which so far have been combined in a single numbering space, i.e. the IP address. This is often referred to as the "Loc/ID split".

LISP is a map-and-encap protocol. A very high overview of how this protocol works is described below. A new header is appended to any existing Layer 3 packet. Thus a packet contains an "inner-header", with the source and destination addresses set to the corresponding *identifiers*, and an "outer-header", with the source and destination addresses set to the corresponding *locators*. When an encapsulated packet arrives at the destination border router, the router decapsulates the packet, maps the destination *identifier* to a *locator* that corresponds to an entry point in the destination domain (hence an *identifier*-to-*locator* mapping system is needed), and sends it on to its destination.

This and other similar models rely on an additional level of indirection in

the addressing architecture to make the routing system reasonably scalable. Since packets in transmit are sourced with an *identifier* in their "Destination Address" field and *identifiers* are generally not routable, the destination *identifier* must be mapped to a *locator* in order to deliver the packet to another domain. Such approaches obtain the advantages of the level of indirection afforded by the "Loc/ID split" while minimizing changes to hosts and to the core routing system. LISP is designed to be implemented in a relatively small number of routers and aims to improve site multihoming, decouple site addressing from provider addressing, and reduce the size and dynamic properties of the core routing tables.

However, the limitations imposed by using models like this are also evident. To start with, when an endpoint moves changes to the mapping between its *identifier* and a set of *locators* for its new network location may be required, something that most probably would lead to packets being delayed or even dropped. One must also carefully consider the encapsulation overhead, since the addition of the LISP header might cause the encapsulated packet to exceed the path Maximum Transmission Unit (MTU). Lastly, there is an associated trade-off in the mapping system among the state required to be held by network elements, the rate of updates to the mapping system, and the latency incurred when looking up an *identifier*-to-*locator* mapping, something that must be also carefully handled when employing such an approach.

Chapter 3

LDP Design

The sole purpose of LDP is to impose a naming hierarchy on all the edge switches within a data center network topology. In other words having this location discovery protocol run on top of a multi-rooted tree topology ensures that each edge switch within this topology will relatively quickly acquire a unique name, distinguishing it from the rest of the edge switches. Eventually this name will serve as a locator to be used when routing/forwarding packets from one end host to another.

3.1 Data center network topology

At this point it is important to clarify that we consider a data center network topology to be any kind of a 3-tier, multi-rooted tree of switching and/or routing elements; a topology that is to our knowledge what any large company uses to build its data centers nowadays. A very well known example of such a topology is called a fat tree [4], a special instance of a Clos topology that may be used to interconnect commodity Ethernet switches as proposed by [2]. A k-ary fat tree is organized as described next. There are k pods, each containing two layers of $k/2$ switches. Each k-port switch in the lower layer is directly connected to $k/2$ hosts, whereas its remaining $k/2$ ports are connected to $k/2$ of the k ports in the aggregation layer of the hierarchy. What is more there are $(k/2)^2$ k-port core switches, which have one of their ports connected to each of the k pods. This is

done in a way that ensures that consecutive ports in the aggregation layer of each pod switch are connected to core switches on $k/2$ strides.

A fat tree built out of k-port switches has the ability to support $k^3/4$ hosts. Fat trees are rearrangeably non-blocking, meaning that for arbitrary communication patterns, there is some set of paths that will saturate all the bandwidth available to the end hosts in the topology. What is more, another advantage of the fat tree topology is that all switching elements used to built it are identical, enabling one to leverage cheap commodity parts for all of the switches in the communication architecture. The simplest non-trivial instance of the fat tree with k = 4 is illustrated in Figure 3.1.

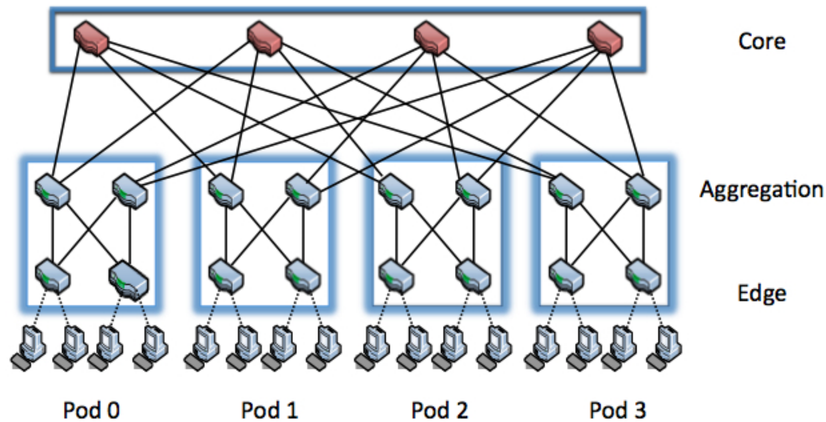

Figure 3.1: Fat tree built out of 4-port switches.

3.2 Defining position

In this subsection we define our notion of a switch's location within a topology such as the one described above. We first divide all the switches within this architecture into 3 distinct categories depending on their distance from the root switches they are connected to. A switch may be a core switch, i.e. it is itself one of the roots of the tree, an aggregation switch which implies that it is directly connected to at least one core switch, and finally an edge switch which implies that

it is a leaf node of the tree, hence directly connected to at least one aggregation switch and one end host.

A core switch needs no other information other than the knowledge that it is a core switch, in order to forward packets to a certain destination host. Hence core switches do not need to acquire a unique name, but simply to learn the level of the tree that they belong in. For the rest of the switches, i.e. the aggregation and edge switches, two additional numbers are used to uniquely identify their location in a data center topology. We call these pod and position numbers respectively. Any combination of these two numbers describing a switch S must be unique meaning that no other switch that has a path connecting it to S is allowed to have the same two numbers assigned to it. To be more precise what this protocol aims to do is to group edge and aggregation switches into different cliques (which we refer to as pods) of at most k switches each, where k is a parameter that in the case of a fat tree is set to the number of ports per switch used to construct it. Each switch within each pod will have a unique position number that will distinguish it from other switches belonging in the same pod.

Using a standard fat tree topology, illustrated in Figure 3.1, one can easily observe that the notion of pods is inherently present. By removing all the core switches from this topology one can easily distinguish disjoint connected graphs of size k. Using 8 bits for representing the pod number and another 8 bits for representing the position number, it is clear that we have the ability to uniquely name all edge and aggregation switches within a fat tree topology that is, in theory, capable of hosting 256x128x128 = 4.194.304 hosts (i.e. if the appropriate switching/routing elements were available). This number of hosts is more than what anyone would have imagined to be able to support in a data center. Needless to say we do have the ability to use 16 bits for representing a switch in a Layer-2-like solution to routing/forwarding by simply using 16 of the available 48 bits used for representing an Ethernet address in a packet's Layer 2 header. As one can imagine the additional 32 bits will be used to specify the port number at which a host is connected to an edge switch (8 bits required) as well the destination's virtual machine ID since each physical machine will probably be hosting multiple virtual

machines. Note that the last two numbers can automatically be configured at each edge switch locally. All these fields basically constitute the location-dependent pseudo MAC addresses to be used as the basis of routing/forwarding as explained in Chapter 1.

3.3 High level design

What follows is a detailed description of LDP's design. Using the above notion of a switch's location within a multi-rooted tree topology we have designed a scalable protocol that allows all switches within the topology determine their location in a distributed manner using only local information and efficiently.

We describe two different approaches for LDP. One is specifically tied to fat tree topologies and the other one generalizes to any kind of 3-tier, multi-rooted tree topologies. Both approaches are very similar to each other, however the first one takes advantage of the additional knowledge/assumption that the underlying network topology is indeed a fat tree, something that makes the whole process simpler. Clearly one could very well use the second approach for fat tree topologies too without sacrificing correctness in any way. However by doing so he might end up creating pods that do not exactly match the ones illustrated in Figure 3.1, although as we will show in Section 6 this is certainly not the common case.

3.3.1 Fat tree specific LDP

For the purposes of LDP the switches in a fat tree topology periodically send a Location Discovery Message (LDM) on all of their ports. This is used to convey information to neighboring switches, but also to monitor liveness in steady state. LDMs contain the following information:

- *Switch identifier (switch_id)*: a unique 48-bit identifier for each switch, e.g., the lowest MAC address of all local ports.

- *Pod number (pod)*: an 8-bit number shared by all switches in the same pod. Switches in different pods will have different pod numbers. This value is

never set for core switches.

- *Pod identifier (group_id)*: a unique 48-bit number shared by all switches in the same pod. Switches in different pods will have different pod identifiers at all times which assists in eventually ensuring uniqueness of pod numbers assigned to each pod.This value is never set for core switches.

- *List of pod numbers already taken (taken_pods)*: what a switch has heard from the rest of the network via its neighbors regarding the pod numbers already assigned to other pods. This is basically a list of 2-tuples in the form $< group_id,\ pod >$.

- *Position (pos)*: an 8-bit number assigned to each edge and/or aggregation switch, uniquely identifying it within each pod.

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

- *Port number and up/down (dir)*: a switch-local view of the port number along which the LDM is forwarded. Up/down is an additional bit which indicates whether the port is facing downward or upward in the multi-rooted tree.

Initially, all values other than the switch identifier and port number will be unknown and we assume the fat tree topology described in Section 3.1. We assume all switch ports are in one of the following three states: disconnected, connected to an end host, or connected to another switch.

LDP for fat trees is a three step protocol. The first step is tree level discovery and allows each switch in the 3-tier multi-rooted tree discover their level in the tree, i.e. whether they are edge, aggregation or core switches. This information facilitates the subsequent steps. The second step deals with defining the position of each switch within its pod. As we have previously explained this number uniquely identifies each switch within its pod. The last step involves assigning a pod number

to each pod and making sure that this is unique across pods. All three steps are described in detail next.

The key insight behind the first step of LDP is that edge switches receive LDMs only from aggregation switches they are directly connected to since end hosts do not speak LDP and thus do not generate LDMs. We use this observation to bootstrap tree level assignment in LDP. Edge switches learn their level by determining that some fraction of their ports are connected to hosts. Level assignment then flows up the tree. Aggregation switches set their level once they learn that some of their ports are connected to edge switches and finally core switches learn their level once they confirm that all of their ports are connected to aggregation switches.

Algorithm 1 presents the processing performed by each switch in response to LDMs. Lines 2-4 are concerned with position assignment and will be described below. In line 6, the switch updates the set of switch neighbors that it has heard from. In lines 8-9, if a switch is not connected to more than $k/2$ neighbor switches for sufficiently long, it concludes that it is an edge switch. The premise for this conclusion is that edge switches have at least half of their ports connected to end hosts. Once a switch comes to this conclusion, on any subsequent LDM it receives it infers that the corresponding incoming port is an upward facing one. While not shown for simplicity, a switch can further confirm its notion of position by sending pings on all ports. Hosts will reply to such pings but will not transmit LDMs. Other switches will both reply to the pings and transmit LDMs.

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

Determining the level for core switches is somewhat more complex, as addressed by lines 14-20. A switch that has not yet established its level, first verifies that all of its active ports are connected to other switches (line 14). It then ver-

ifies in lines 15-18 that all its neighbors are aggregation switches who have not yet set the direction of their ports connecting them to it (aggregation switch ports connected to edge switches would have already been determined to be downward facing). If these conditions hold, the switch can conclude that it is a core switch and set all its ports to be downward facing (line 20).

Edge switches must acquire a unique position number within their pod in the range of $0..\frac{k}{2}-1$. This process is addressed by Algorithm 2. Each edge switch proposes a randomly chosen number in the appropriate range to all of the aggregation switches in the same pod. If the proposal is accepted by a majority of these switches, then it is finalized and this value will be included in future LDMs sent from the edge switch. As shown in lines 2-4 and 33 of Algorithm 1, aggregation switches will hold a proposed position number for some period of time before timing it out in the case of multiple simultaneous proposals for the same position number.

Lastly, LDP leverages a gossip based approach to ensure uniqueness of pod numbers (*pod*) assigned to all switches belonging in the same pod. In lines 8-10 of Algorithm 2, the edge switch that adopts position 0 randomly selects a pod number within the range of $0...k$ and also sets the pod identifier ($group_id$) of this pod to its own switch identifier. We note here that this identifier is guaranteed to uniquely identify the pod since by definition no two switches can have the same switch identifier. On the other hand the pod number which is randomly selected might very well collide with the pod number chosen by any other edge switch belonging in another pod. These pod number and pod identifier values are passed on to the rest of the pod's switches in lines 23-25 of Algorithm 1. Each switch S maintains a list of 2-tuples of the form $<pod,\ group_id>$ ($taken_pods$) corresponding to what it has heard of from the rest of the network via its neighbors regarding already assigned pod numbers. In line 26 S updates this list according to any LDM it receives and also removes entries from the list that might have timed out. Whenever S detects a collision (i.e. a pod with the same pod number but with a smaller identifier than its own) it goes ahead and selects a different pod number at random. It does so by taking under consideration the already taken

pod numbers it knows of in order to avoid future collisions (lines 27-28).

We leave a description of the entire algorithm accounting for a variety of failure and partial connectivity conditions to Section 4.

3.3.2 LDP for general multi-rooted trees

Moving on to LDP for general multi-rooted tree topologies, the notion of having the components of the underlying network fabric periodically send LDMs remains exactly the same. LDMs contain the exact same fields as the ones described above. Similar to our previous approach, all values other than the switch identifier and port number will initially be unknown to each switch. However in this case we make no assumptions on the layout of these switches other than the fact that they form a 3-tier, multi-rooted tree. As for the fat tree specific LDP, three steps are involved in this case too.

The first step of LDP, i.e. tree level discovery, works in the exact same manner as the one explained above. Algorithm 3 presents the processing performed by each switch in response to any kind of LDP-related messages received. Lines 2-20 specifically correspond to the processing performed in response to LDMs, which is exactly the same with the processing performed for the purposes of the fat tree specific protocol. This has been explained thoroughly in the previous section.

The main difference of this approach relies on the fact that the notion of a fat-tree-like pod may not be assumed any more. Thus the second step of LDP for the general multi-rooted trees deals with overlaying pods on the set of arbitrarily connected level 0 and 1 switches. Edge and aggregation switches must somehow coordinate their actions towards grouping themselves into pods of at most k switches each. We achieve this by randomly selecting pod leaders from the set of aggregation switches that will initiate the creation of a new pod by inviting edge switches which they are directly connected to and are also available to join the new pod to do so. Along with an offer to join the new pod an edge switch might also receive a share of the new pod's membership tokens which it will use to carry on the same procedure recursively by inviting other aggregation switches to join. This goes on until the new pod of size k is established or until no more

accessible switches are available to join the new pod.

Algorithm 4 is all about making this happen. Here we will explain in more detail how this process of creating a new pod works. We introduce 3 additional types of LDP messages that might be exchanged between neighboring switches during this process. These are the QUERY, REPLY and JOIN messages which we explain below. All aggregation switches wait for a randomly selected amount of time within the range $0...2T$, where T is the round trip time between 2 neighboring switches (line 2 of Algorithm 4). After this amount of time has passed, if the corresponding aggregation switch has still not joined a pod, it initiates the creation of a new pod by following the steps depicted below. First it randomly selects a pod number within the permissible range $0...k$, ensuring that this number does not belong in the list of pod numbers it has heard of so far (line 3). It also sets the new pod's identifier to its own switch identifier (line 4). It then sets its position within the newly created pod to 0 and queries all of its neighbors asking them whether or not they have joined a pod yet, by sending a QUERY message on all of its ports (lines 9-10). In line 11 the aggregation switch waits for a round trip time to collect the REPLY messages from its corresponding neighbors that have not yet joined a pod (line 31 of Algorithm 3). Lines 23-24 of Algorithm 3 handle the reception of such a QUERY message by any switch. The receiver switch first checks whether it has already replied to a QUERY message previously sent by another switch and is still locked on a state waiting for a JOIN message from it. It achieves this by using the *locked* variable which is set to true (line 24 of Algorithm 3) right before sending the REPLY message and which timeouts after $2T$ time has elapsed. Hence if this receiver switch is not locked, has not joined a pod yet and is not a core switch, it locks on a state waiting for a JOIN message from the sender and then immediately replies back to the sender letting it know that it is available to join a pod.

Here we introduce the notion of a pod's membership tokens, each of which corresponds to a specific position within that pod. Once a switch initiates the creation of a new pod, it immediately has k tokens in its possession (i.e. $0...k$). It keeps token 0 for itself and then tries to split the remaining tokens equally among its neighbors that have not yet joined a pod. Having explained that, in

lines 12-13 of Algorithm 4 the aggregation switch initiating the creation of the new pod divides the remaining tokens into equal shares among the switches who replied back to its QUERY message and finally sends them a JOIN message which includes their tokens share as well as the new pod's identifier and number.

Algorithm 4 is used in the exact same way by switches that receive spare tokens from any of their neighbors (line 28 of Algorithm 3). This eventually leads to the creation of a new pod with at most k switches. In this case however, instead of randomly selecting a pod number and setting their pod identifier to their own switch identifier, switches use the corresponding values included in the JOIN message they have just received (lines 7-8 of Algorithm 4) to set the values of these quantities. What is more they set their position within the pod to be equal to the smallest in value token from those given to them via the JOIN message and they split the rest (if any) among their available-to-join neighbors in the same fashion as the one described previously.

Algorithm 1 (Fat tree): $LDP_listener_thread()$

```
1:  While (true)
2:      For each tp in tentative_pos
3:          If (curr_time − tp.time ≤ timeout) tentative_pos ← tentative_pos − {tp};
4:      ▷ Case 1: On receipt of LDM P
5:      Neighbors ← Neighbors ⋃{switch that sent P}
6:      If (curr_time − start_time > T and |Neighbors| ≤ k/2)
7:          level ← 0; incoming_port ← up;
8:          Acquire_position_thread();
9:      If (P.level = 0 and P.dir = up)
10:         level ← 1; incoming_port ← down;
11:     Else If (P.dir = down)
12:         incoming_port ← up;
13:     If (level = −1 and |Neighbors| = k)
14:         is_core ← true;
15:         For each switch in Neighbors
16:             If (switch.level ≠ 1 or switch.dir ≠ −1)
17:                 is_core ← false; break;
18:         If (is_core = true)
19:             level ← 2; Set dir of all ports to down;
20:     If (P.pos ≠ −1 and P.pos ⊈ Pos_used)
21:         Pos_used ← Pos_used ⋃ {P.pos};
22:     If (P.pod ≠ −1 and level ≠ 2)
23:         pod ← P.pod; group_id ← P.group_id;
24:     Update taken_pods according to P.taken_pods′ contents and the timeout value;
25:     If({< pod, X >} ⊂ taken_pods and X ≤ group_id)
26:         pod ← random()%k, s.t. pod ⊈ taken_pods;
27:     ▷ Case 2: On receipt of position proposal P
28:     If (P.proposal ⊈ (Pos_used⋃tentative_pos))
29:         reply ← {"Yes"};
30:         tentative_pos ← tentative_pos⋃{P.proposal};
31:     Else reply ← {"No", Pos_used, tentative_pos};
```

Algorithm 2 : $Acquire_position_thread()$

1: $taken_pos = \{\}$;

2: **While** ($pos = -1$)

3: $proposal \leftarrow random()\%\frac{k}{2}$, s.t. $proposal \not\subseteq taken_pos$

4: Send proposal on all upward facing ports

5: Sleep(T);

6: **If** (more than $\frac{k}{4} + 1$ switches confirm $proposal$)

7: $pos = proposal$;

8: **If**($pos = 0$)

9: $pod = random()\%k$;

10: $group_id = switch_id$;

11: Update $taken_pos$ according to replies;

Algorithm 3 (General multi-rooted tree): *LDP_listener_thread*()

```
1:  While(true)
2:      ▷ Case 1: On receipt of LDM P
3:      Update taken_pods according to P.taken_pods' contents and the timeout value;
4:      If(pod ≠ P.pod and group_id = P.group_id)
5:          pod ← P.pod;
6:      If({< pod, X >} ⊂ taken_pods and X ≤ group_id)
7:          pod ← random()%k, s.t. pod ⊈ taken_pods;
8:      If(curr_time − start_time > T)
9:          If(LDMs received ≤ k/2)
10:             level ← 0;
11:             input_port ← upward facing port;
12:     If(P.level = 0 and P.dir = up)
13:         level ← 1;
14:         input_port ← downward facing port;
15:         Group(0, k − 1, −1; group_id);
16:     Else if(P.level = 1 and P.dir = up)
17:         level ← 2;
18:         input_port ← downward facing port;
19:     Else if(P.dir = down)
20:         input_port ← upward facing port;
21:
22:     ▷ Case 2: On receipt of QUERY message P
23:     If(¬locked and pod = −1 and level ≠ 2)
24:         locked ← true;              ▷ Lock for T on S waiting for a JOIN packet
25:         Send a REPLY packet to S;         ▷ Stating availability to join a group
26:
27:     ▷ Case 3: On receipt of JOIN message P
28:     Group(P.tok_start, P.tok_end, P.pod, P.group_id);
29:
30:     ▷ Case 4: On receipt of REPLY message P
31:     L ← L ⋃ {switch that sent P};
```

Algorithm 4 : $Group(tok_start,\ tok_end,\ in_pod,\ in_group_id)$

```
1: If(in_pod = -1)
2:     Sleep(random()%(2 × T));
3:     If(pod = -1)
4:         pod ← random()%k, s.t. pod ⊈ taken_pods;
5:         group_id ← switch_id;
6: Else
7:     pod ← in_pod;
8:     group_id ← in_group_id;
9: position ← tok_start;
10: Send a QUERY packet to all neighbors;              ▷ Asking them if they are grouped
11: Sleep(T);                                ▷ In the meanwhile store responses in list L
12: Divide [tok_start + 1, tok_end] equally into |L| shares;
13: Send JOIN packets with the corresponding tokens share to all members of L;
```

Chapter 4

Failure Analysis

The following analysis aims to characterize the LDP protocol in the presence of any kind of exceptional conditions such as:

- Switch failures
  - Hardware switch failures
  - Software switch failures
- Miss wiring conditions

In this section we propose solutions to these various cases, wherever this is feasible, and in the same time we identify very rare corner cases in which the protocol might fail. We focus on fat tree topologies since our evaluation platform (described in detail in Section 5) has been built along these lines. However we note that our approach very easily generalizes to any kind of multi-rooted trees as well as partially connected fat trees, with only slight modifications depending on the actual layout and wiring between the switching elements involved.

What is also worth noticing is that handling software switch failures also addresses all exceptional cases caused by hardware switch failures. In both cases the effect is the same; a switch is not able to participate in the location discovery process, thus does not convey important information to the rest of the switches in the data center network topology. This might potentially cause problems to the normal operation of LDP. However software switch failures can be considered worse

than hardware switch failures since the occurrence of the latter can be discovered by the faulty switch's neighbors, thus allowing them to take appropriate action upon such events. In the case of software failures, the neighbor of any such faulty switch will have no clue whether its neighbor has crashed, is an end host and thus does not speak the LDP protocol anyway or has been removed by an administrator for any reason. Thus all of the cases that arise due to hardware failures can be viewed as a subset of those caused by software failures and so by effectively handling the latter we can omit an analysis on the former.

4.1 Assumptions

For our analysis we consider the following assumptions:

- All the pods within the fat tree topology are fully configured
- The fat tree might have an arbitrary number of pods and corresponding core switches to interconnect them

4.2 Software switch failures

By software switch failure we refer to the condition where a switch crashes, however the links it was connected to might still be up. Therefore the other end of these links might be unaware of this irregularity, meaning that they are unable to immediately detect this failure. Depending on the position of the faulty switches we may come across the following cases:

- Switch failures restricted to only one level of the fat tree
- Switch failures across multiple levels of the fat tree

In the following sections we address both these cases in more detail.

4.2.1 Switch failures restricted to one level of the fat tree

X edge switch failures

In this subsection we examine the case where an arbitrary number X of edge switches might crash. We will go through all the possible scenarios and see how LDP would perform under these circumstances.

First we consider the primitive case where the X faulty switches are a genuine subset of a pod's edge switches. This scenario will create no problem to the normal operation of LDP since the portion of the non-faulty edge switches that are connected to an aggregation switch within that pod will at some point inform the latter of its level in the tree by sending an LDM on their upward facing port indicating that they are level 0 switches.

However the X faulty switches may consist of all edge switches in a pod A. The corresponding aggregation switches in A will initially believe that they are edge switches since they will not receive LDMs on all of their downward facing ports. This case can be addressed since a core switch connected to pod A will identify the problem once it receives LDMs from both edge and aggregation switches, which is impossible under normal circumstances. Thus it can signal an exception bringing this matter to an administrator's attention.

Lastly, we have the case where the X faulty switches correspond to all the edge switches of a fat tree. All aggregation switches will initially think that they are edge switches since they will not receive LDMs on their downward ports. However this case can be handled in a similar to the one previously described manner since core switches will receive LDMs on all of their ports from switches claiming to be edge switches, which is again impossible under normal circumstances.

X aggregation switch failures

This subsection deals with the case where an arbitrary number X of aggregation switches might crash. We will go through all the possible scenarios of this case too and see how LDP would perform under these circumstances.

We first consider the case where these X switches consist of a fraction (< 1)

of the switches connected to a core switch. Although at first sight one might think that this case would mislead a core switch into believing that it is an edge switch, this is not the case. A core switch will receive LDMs from aggregation switches which will report sending them on their upward facing ports. Hence even if more than half of such a core's ports are connected to faulty aggregation switches it will not be tricked into believing that it is an edge switch.

It might also be the case that a subset of the X switches consists of all of a core switch's neighbors. In this case the corresponding core switch is entirely cut off from the rest of the topology, which therefore will not create any problems for the rest of the switches in the fat tree.

The last case we examine in this subsection is the one where a subset of the X faulty switches consists of all the aggregation switches within a pod. Such an event would lead to an entire pod getting disconnected from the rest of the fat tree. This case however would not create any problems for the rest of the switches.

X core switch failures

Here we examine the case where an arbitrary number X of core switches might crash. We break down this case into the following instances:

The general case is when a genuine subset of these X switches consists of a fraction (< 1) of the switches connected to an aggregation switch. Since level discovery is initiated by level 0 switches (i.e. edge switches) and moves upwards in the tree, this case will have no effect to the normal operation of LDP.

However if a subset of the X faulty switches consists of all core switches connected to an aggregation switch S, then one would expect that S would at some point infer that it is an edge switch. Nevertheless, once S receives an LDM from a switch reporting to be a level 0 switch it will detect this anomaly and can therefore signal an exception notifying an administrator to take appropriate action.

Finally if the X switches consist of all core switches, it is clear that the fat tree will be partitioned into k disjoint graphs which correspond to the k pods of the fat tree. This case can be treated in the same exact way as the one we have described above.

4.2.2 Switch failures across multiple levels of the fat tree

Although we have talked about switch failures restricted to one level of a fat tree so far, it is clear that any combination of switch failures across multiple levels of the tree is possible and probably the common case. However, covering all such possible cases is an almost impossible task. We have made an effort to cover the majority of these cases in the following section, but we would like to note here that we do not claim that this analysis is 100% complete.

All the switches connected to a switch B are down

If such a scenario were to occur then switch B would be the only switch to be affected. Obviously B would assume that it is an edge switch since for a sufficiently long time it would not receive LDMs on $\geq k/2$ of its ports. For the duration of this condition though, this would have no effect whatsoever to the correctness of our protocol. B would anyway be cut off from the rest of the topology, making it useless in a sense for the operations of the data center. However one could claim that having B assume to be an edge switch could cause problems if at any point in time any of the switches connected to B recovered from their failure. For addressing these concerns we add another rule to our protocol stating that any switch must receive at least one LDM before setting its location-defining variables to any particular value. This way B would preserve its initial values (i.e. -1) under these exceptional circumstances making it impossible to affect other switches even if it became accessible to the rest of the network at any point in time.

Less than half of the switches connected to a core switch B are down whereas the rest believe to be edge switches

To better understand this scenario consider Figure 4.1.

In this case the aggregation switches colored in black will assume they are edge switches whereas the core switch connected to them (also colored in black) will infer that it is an aggregation switch since it will only receive LDMs from switches claiming to be edge switches. For addressing this case we add another rule to our protocol stating that if any aggregation switch does not receive an

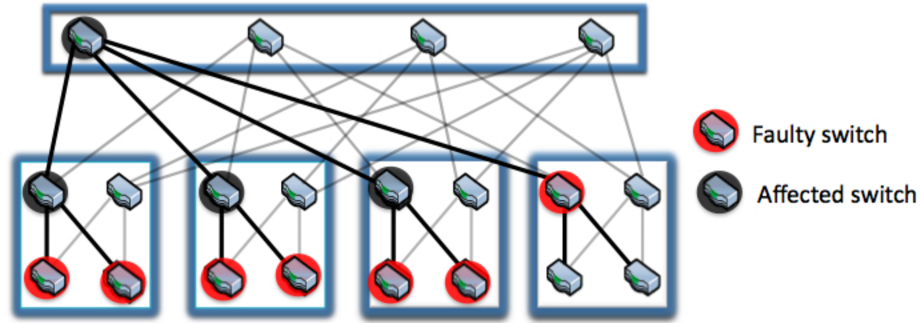

Figure 4.1: Core switch believing it is an aggregation switch.

LDM from a core switch for a sufficiently long period of time it signals an exception bringing this event to an administrator's attention, after reinitializing it's location-defining variables. Hence the black core switch will at some point in time signal this exception thinking that it is an aggregation switch and thus it should receive at least one LDM from a core switch.

4.3 Miss wiring conditions

We may have all sorts of miss wiring in a fat tree topology. Analyzing each case separately by taking under consideration all possible combinations is a task almost impossible to carry out to completion. Thus we use an alternative approach for our analysis on any kind of miss wiring conditions. Our approach is much simpler and arises from the following observation:
If any of the following rules is violated, the switch that observes this violation signals an exception to a central authority which in turn informs an administrator to fix the problem. The miss wiring then can be dealt with appropriately.

- **Rule 1:** We know that in a "correctly" wired fat tree topology each switch can only receive LDMs from switches belonging in the following groups of levels:
  - {Level 0, Level 2} for aggregation switches

  - {Level 1} for edge and core switches

  Thus we can have each switch note the levels of the switches from which it receives LDMs and periodically have them check whether they belong in one of the above categories. If not, for instance when a switch receives an LDM from a level 0 and a level 1 switch, then this switch must signal an exception.

- **Rule 2:** Aggregation switches should receive LDMs from level 0 switches with their *dir* field marked as up, and LDMs from level 2 switches with their *dir* field marked as down. Furthermore, edge switches should receive LDMs from level 0 switches with their *dir* field marked as down whereas core switches should receive LDMs from level 1 switches with their *dir* field marked as up. If any of the instances of this rule is violated then the corresponding switch observing this will also signal an exception.

- **Rule 3:** If a core switch receives LDMs from the same pod on more than one of its ports it must signal an exception since each of a core switch's ports should be connected to a different pod.

- **Rule 4:** If any switch receives LDMs from the same switch on more than one of its ports or receives an LDM from itself then it must signal an exception since this also implies miss wiring.

However, the above rules do not cover all cases of miss wiring since there might be a weird combination of such events that will not violate them, hence leading to an undetected irregularity. We identify these rare cases below.

Firstly, we consider the case where an edge switch might wrongly believe that it is an aggregation switch. This will happen when a core switch A is directly connected to an edge switch B, which in turn is connected to another edge switch C. Furthermore B's other upward facing ports are either connected to other core switches or are simply down. This is illustrated in Figure 4.2.

In this case B will receive LDMs from only level 0 and level 2 switches, thus rule 1 is not violated. We cannot rely on rule 2 to resolve this case either since B receives an LDM from C's upward facing port and an LDM from A's downward

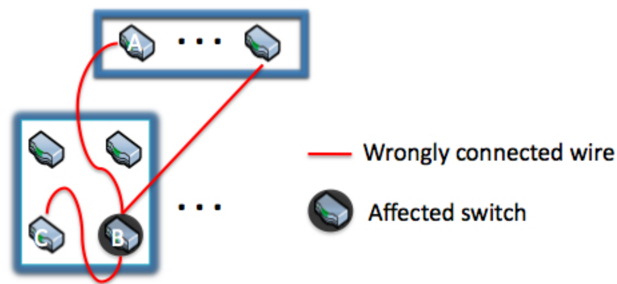

Figure 4.2: Edge switch believing it is an aggregation switch.

facing port, as it would expect as an "aggregation" switch. Rules 3 and 4 are also not violated since 3 only refers to core switches, whereas B thinks it is an aggregation switch, and also no switch in the above example is connected more than once to any other switch.

The second case we examine here is the one where an edge switch B might believe it is a core switch if all of its ports are either directly connected each to a different aggregation switch (on one of their upward facing ports) or they are down. This is illustrated in Figure 4.3.

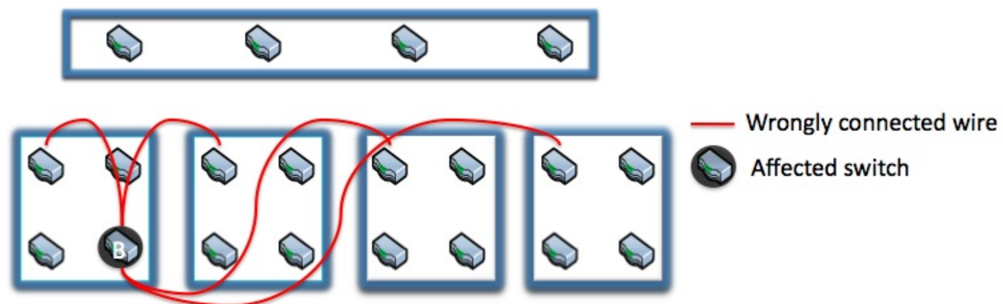

Figure 4.3: Edge switch believing it is a core switch.

In this case B will receive LDMs only from level 1 switches, something that again does not violate rule 1. Furthermore B receives LDMs only from the upward facing ports of the corresponding aggregation switches, as it would expect as a "core" switch, thus rule 2 would not help either. Rules 3 and 4 are not violated since B is connected only once to each pod and also no switch in the above example

is connected more than once to any other switch.

What is more an aggregation switch might infer that it is an edge switch if all its downward facing ports are either connected to end hosts or are simply down, and all its upward facing ports are either connected to the downward facing ports of an aggregation switch or are down. This is illustrated in Figure 4.4.

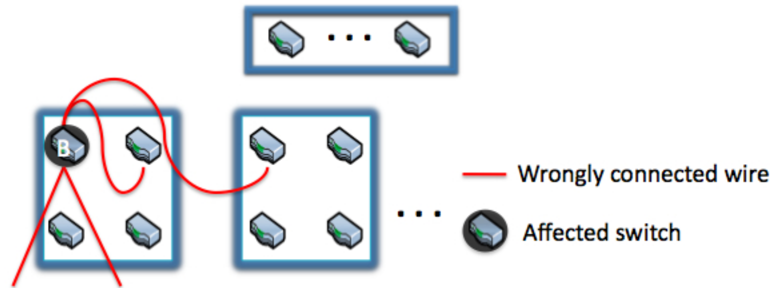

Figure 4.4: Aggregation switch believing it is an edge switch.

Here B will receive LDMs from only level 1 switches, thus rule 1 is not violated. Rule 2 is also not violated since B receives LDMs from only downward facing ports of the corresponding aggregation switches, as it would expect as an "edge" switch. Rule 3 only refers to core switches, whereas B thinks it is an edge switch and as in all previous cases rule 4 is not violated since no switch in the above example is connected more than once to another switch.

The fourth case we examine is the one where an aggregation switch believes it is a core switch if all of its ports either connected to the upward facing ports of other aggregation switches or are down. This case is illustrated in Figure 4.5.

Again B will receive LDMs from only level 1 switches which satisfies rule 1. Furthermore B will receive LDMs on only upward facing ports of the corresponding aggregation switches, as it would normally expect as a core switch. Rules 3 and 4 are satisfied since B is connected only once to each pod and no switch in the above example is connected more than once to another switch.

Next we examine the case where a core switch believes it is an edge switch if all of its ports are either connected to end hosts, are down or are connected to the downward facing ports of aggregation switches. This is illustrated in Figure 4.6.

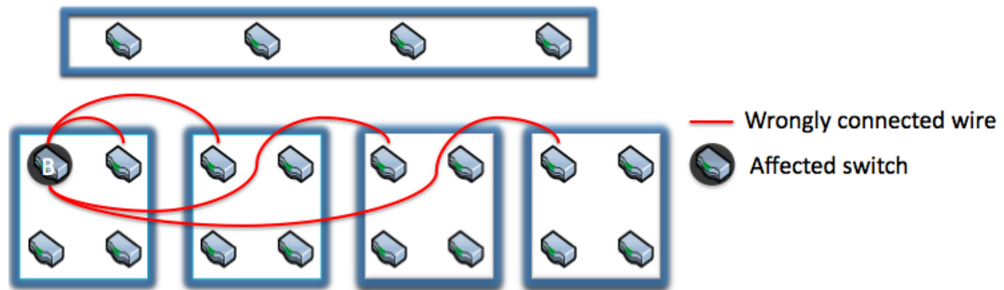

Figure 4.5: Aggregation switch believing it is a core switch.

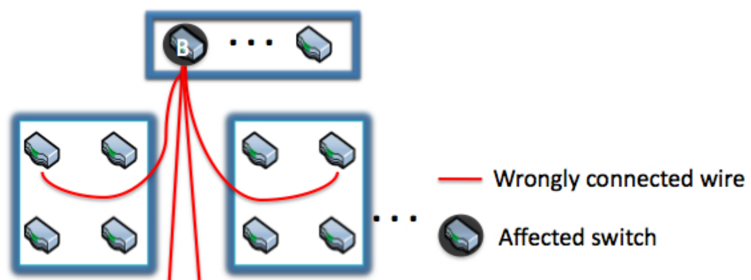

Figure 4.6: Core switch believing it is an edge switch.

Here B will receive LDMs from only level 1 switches thus misleading it to believe that it is an edge switch. Hence rule 1 is not violated. Rule 2 is also not violated since B receives LDMs from only downward facing ports of the corresponding aggregation switches, as it would expect as an ”edge” switch. Rule 3 is obviously not violated since it only refers to core switches, whereas B thinks it is an edge switch. Lastly, rule 4 is satisfied since no switch in the above example is connected more than once to another switch.

The last case we will examine is the one where a core switch believes it is an aggregation switch if all of its ports are either connected to edge switches, are down or are connected to the downward facing ports of core switches. This is illustrated in Figure 4.7.

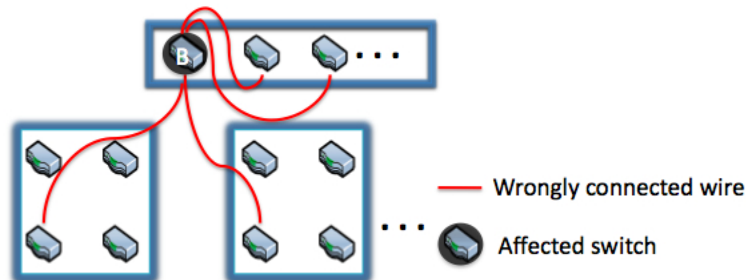

Figure 4.7: Core switch believing it is an aggregation switch.

In this case B will receive LDMs from only level 0 and level 2 switches thus rule 1 is satisfied. Rule 2 is also satisfied since B receives an LDM from only the upward facing ports of the corresponding edge switches and the downward facing ports of the corresponding core switches, as it would expect as an ”aggregation” switch. Once again rule 3 is not violated since it only refers to core switches, whereas B thinks it is an aggregation switch. And of course rule 4 is not violated since no switch in the above example is connected more than once to another switch.

All the above cases that we have indicated can be considered very rare, in the sense that for them to occur by accident a great portion of the topology's links must be miss wired. Although at a first glance these cases might seem to

involve only a small number of specific links of the entire fat tree topology, as shown in the figures above, when generalizing these cases for fat trees with k pods we can easily see that this is not the case. To be more precise the first case would involve the miss wiring of $\frac{k}{2} + 1$ specific links with respect to an edge switch. The second case would involve the miss wiring of k specific links with respect to an edge switch. Similarly the third and fourth cases would involve the miss wiring of k specific links with respect to an aggregation switch. And lastly the fifth and sixth cases would be triggered by the miss wiring of k specific links with respect to a core switch. Furthermore it is safe to assume that if such combinations of miss wiring were to occur then other switches would also be affected (assuming that all switches in the topology would have all their ports connected somewhere). These other switches would most probably be able to detect the violation of any of the 4 rules we mentioned at the beginning of this section.

Chapter 5

Implementation

5.1 Simulator

For the purposes of evaluating our solution for the general multi-rooted tree case we required the need for implementing a simulator that would simulate the behavior of Algorithms 3 and 4 when applied on large multi-rooted tree topologies. Therefore we implemented a centralized simulator that first went ahead and created random 3-tier multi-rooted trees and then run LDP for the general multi-rooted trees (described in Section 3.3.2) on these topologies. The random multi-rooted trees were created so that they obeyed the following rules:

- The multi-rooted tree would consist of 3 layers of switches, as in the case of a fat tree, i.e. the core, aggregation and edge layer.
- We preserved the analogy of a fat tree with respect to the number of switches each layer contains. Therefore the number of switches in each layer of any such random multi-rooted tree was defined by a parameter k. Both edge and aggregation layers contained $\frac{k^2}{2}$ switching elements each, whereas the core layer contained $\frac{k^2}{4}$ switching elements.
- Again as in the case of a fat tree, each switching element used to construct the multi-rooted tree was restricted to having k available ports for connecting to other switching elements. The only types of links that were allowed

during the construction of this topology were those connecting core switches to aggregation switches and aggregation switches to edge switches.

The simulator was implemented using the C programming language. We used the gcc v4.0.1 compiler to compile it. The only parameter that the simulator required to run was the value of k to be used both to construct the random multi-rooted tree topology and for defining the target size (in number of switching elements) of the pods to be created.

A limitation we were forced to cope with was the fact that we did not have enough resources for building a distributed simulator, i.e. one that would allow each switching element in large topologies run on a separate machine. Therefore we were forced to use a centralized approach towards implementing this simulator. This made it hard to capture the exact behavior of the switching elements where each of them would run in parallel as one would expect when employing LDP in real time environments. Nevertheless we note that this limitation only negatively affects our results since its direct impact is the fact that it does not allow for switches with spare tokens to compete in parallel for acquiring available switches (that have not yet joined a pod). Therefore this leads to the creation of somewhat more unbalanced, in terms of number of switches, pods.

5.2 Testbed Implementation

5.2.1 Testbed Description

Our evaluation platform closely matches the layout in Figure 3.1. To be more precise our testbed consists of 20 4-port NetFPGA PCI card switches [5], each of which contains 4 GigE ports along with Xilinx FPGA for hardware extensions. We house the NetFPGAs in 1U dual-core 3.2 GHz Intel Xeon machines with 3GB RAM. The testbed interconnects 16 end hosts, 1U quad-core 2.13GHz Intel Xeon machines with 3GB of RAM. All these machines run Linux 2.6.18-92.1.18el5.

The switches run OpenFlow v0.8.9r2 [1], which basically provides the means to control a switch's forwarding table. So far OpenFlow has already been ported

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

When OpenFlow switches are initially started, they attempt to open a secure control channel to a central controller. Thus using the OpenFlow protocol the controller can query, insert, modify flow entries, or perform several more actions. The switches maintain many useful statistics per flow and per-port, such as total byte counts, flow duration, etc. By default a switch behaves as follows: if an incoming packet was not matched in any of the TCAM entries or SRAM table, the switch notifies the controller of the table miss and buffers that packet until further notice. The controller may then instruct the switch to output that packet on a certain port. It additionally may instruct the switch to insert a flow entry in its table to match that specific flow. In such a case subsequent packets that match that entry will naturally not be transmitted to the controller, unless they are specifically ordered to do so for some reason.

5.2.2 Modifications to the OpenFLow NetFPGA software framework

OpenFlow on a NetFPGA consists of the following three components:

- The flow table and some associated actions which is implemented in a hardware/software kernel module.
- A secure channel that connects a switch to a remote controller.
- The OpenFlow protocol which provides an open and standard way for the switches and the controller(s) to communicate.

We leverage and also extend these primitives provided by the OpenFlow NetFPGA software framework to realize a managed data center network fabric that looks like Figure 5.1

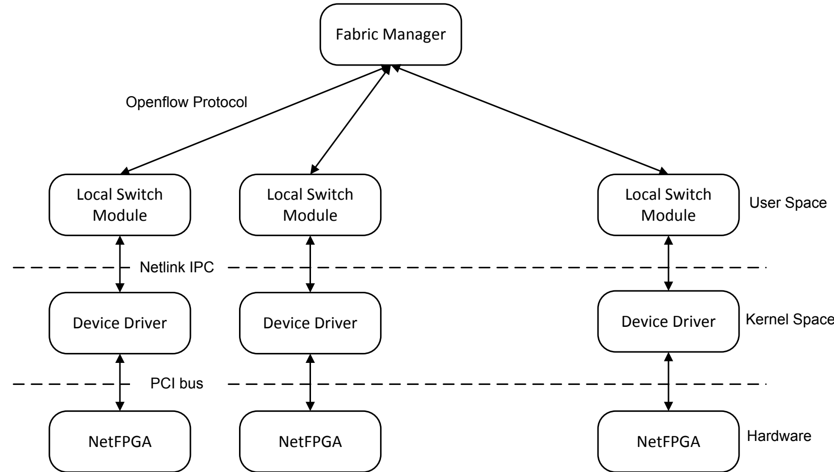

Figure 5.1: System architecture.

Secure Channel (secchan) is a user-level process that runs on each NetFPGA switch. This local switch module is primarily used to encapsulate and forward packets from the NetFPGA switch to the controller. Packets are typically forwarded to the controller if they fail to match an existing flow entry in the switch. What is more secchan also implements the OpenFlow Protocol and acts as an interface to the underlying hardware flow table.

The controller is another user-level process that has the capability to instruct each connected switch to add or remove flow entries, thus basically implements the policy for shaping network traffic.

The OpenFlow protocol defines the interface in which switches can be programmed from a controller. Doing so, it supports three types of messages:

- Controller-to-switch messages which are initiated by the controller to directly manage or inspect the state of a switch.

- Asynchronous messages which are initiated by the switch and are used to raise network or switch state events to the controller.

- Symmetric messages which are initiated by either a switch or the controller and are typically used for request/response type interactions between the two.

We have not made any modifications to these primitives. In general, each of our modules implements a set of public functions that encapsulates the available services and underlying data structures. Both secchan and the controller were originally implemented as single threaded event based servers that multiplex a set of file descriptors/sockets for packet processing. Thus modules with functionality that exhibit blocking I/O are implemented using background worker threads to minimize the performance impact on the main event handling thread. We use queues as the interfaces for communicating events between modules which post back finished tasks for the main event thread to consume.

We chose to implement the LDP functionality as separate modules from the existing code base to minimize the effect of future SDK upgrades. These LDP modules are only invoked by the secchan module. We wanted to exclude the controller from this process thus making our implementation more efficient and fully distributed. To conclude this section we basically added two new threads to facilitate LDP, i.e. an LDP sender and an LDP receiver thread. The former is typically a thread that sends out the LDMs every 100ms on all of a switch's ports. The latter corresponds to the functionality described in Algorithms 3 and 4, that is it processes the LDMs received and handles the process of overlaying pods on the set of aggregation and edge switches.

Chapter 6

Evaluation

6.1 Simulation

In order to evaluate the effectiveness of the LDP that was designed for general multi-rooted tree topologies we used our custom made simulator described in Section 5.1. Our goal was to analyze the pod configurations obtained for varying in size random multi-rooted trees by keeping track of the number of pods that were formed as well as the number of switches per pod.

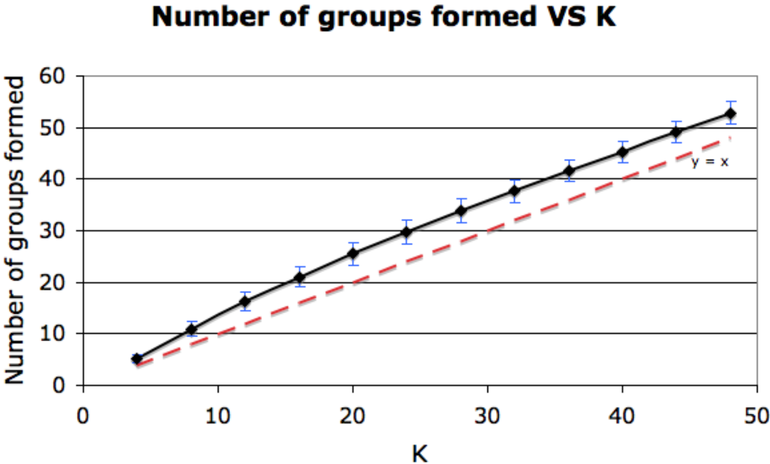

Figure 6.1: Number of pods created Vs k

Figure 6.1 shows the number of groups/pods created when employing LDP

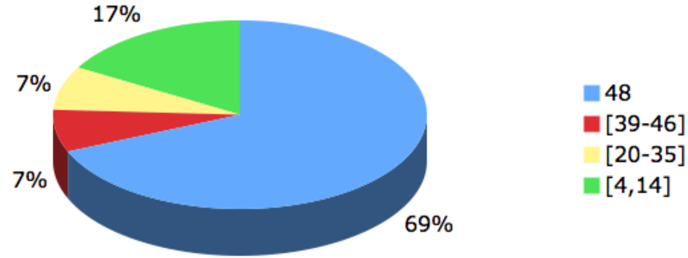

Figure 6.2: Per pod distribution of switches for k=48

on varying in size random multi-rooted trees. Since the target size of each pod is set to k ideally we would hope to split the set of aggregation and edge switches into k pods (as in the case of a corresponding k-fat tree) at each run of our experiment. If this was the case then we would obtain a graph such as the one noted by the red dotted line in Figure 6.1. We run 50 simulations for each k value shown in Figure 6.1. The graph shows the average number of pods that were created for each k across the 50 different samples. We also included error bars to indicate the standard deviation over this average value. Our results indicate that slightly more than k pods were created in each of our runs. However taking under consideration the limitation we addressed in Section 5.1 as well as the fact that the connectivity between the edge and aggregation layers of the randomly created multi-rooted tree was arbitrary we consider the outcome of this experiment reasonably good.

In Figure 6.2 we present the per pod distribution of the aggregation and edge switches for the above experiment that corresponds to the k value set to 48. We note that more or less the same trend was observed for all other k values. Thus we only present one such graph which actually corresponds to the worst case obtained throughout the simulations that we run. Ideally we would like each of the pods created to consist of the same number of switches (which in the best case would be equal to k, i.e. the target pod size we are using) leading to a more

balanced configuration. However our results showed that approximately 70% of the pods that were created managed to acquired k switches. 17% of the pods created only acquired 4-14 switches. We believe this defect was mostly due to the limitation imposed by the use of a centralized simulator that does not allow for switches from different (under construction) pods to compete for inviting other available switches to join their pod. We believe that in real time environments this disparity in pod sizes will be much less evident.

6.2 Testbed Implementation

Although we could not test our LDP implementation on large general multi-rooted trees, due to the limited resources that were available to us, we nevertheless tested it on our small testbed (described in Section 5.2.1). Our goal was to observe how the LDP designed for the general multi-rooted trees would perform in a real time environment and especially when applied to a fat tree topology. More specifically we wanted to measure the performance of our design in terms of the time it takes to converge. We also wanted to make sure that for fat tree topologies the pods that would be configured would match the notion of the pods that are inherently present in a fat tree. Our results with respect to the above are summarized in Figures 6.3 and 6.4.

In Figure 6.3 we illustrate how successful the LDP protocol that was designed for the general multi-rooted tree topologies was in discovering the "correct" pods of our fat tree as these are identified in Figure 3.1. We run our protocol on the testbed for 50 times, by rebooting all the switches after each attempt, and recorded the pod configuration each time. For 92% of the attempts all four pods where properly configured whereas for the remaining 8% one of the "correct" pods was split into 2 separate pods, each of them containing 2 switches.

We also measured the time taken for LDP to converge i.e. for all edge switches to acquire a unique pod and position number. We run the same experiment for 50 times and measured the total time during which the switches went through all three steps of the protocol, as these were described in Sections 3.3.1

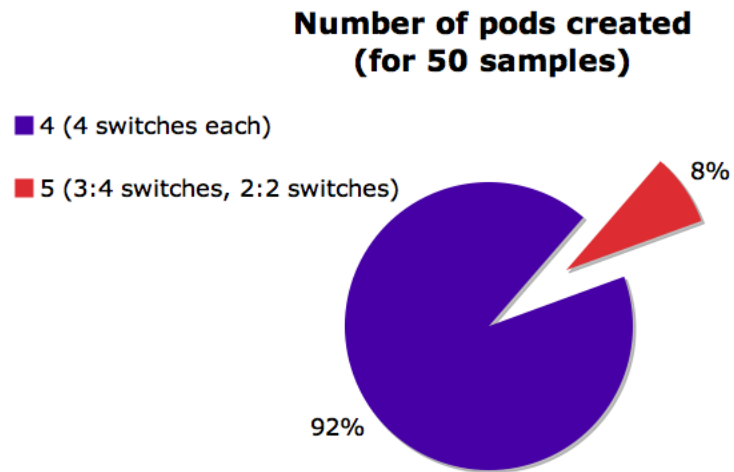

Figure 6.3: Number of pods created in a fat tree with k=4 over 50 runs of LDP

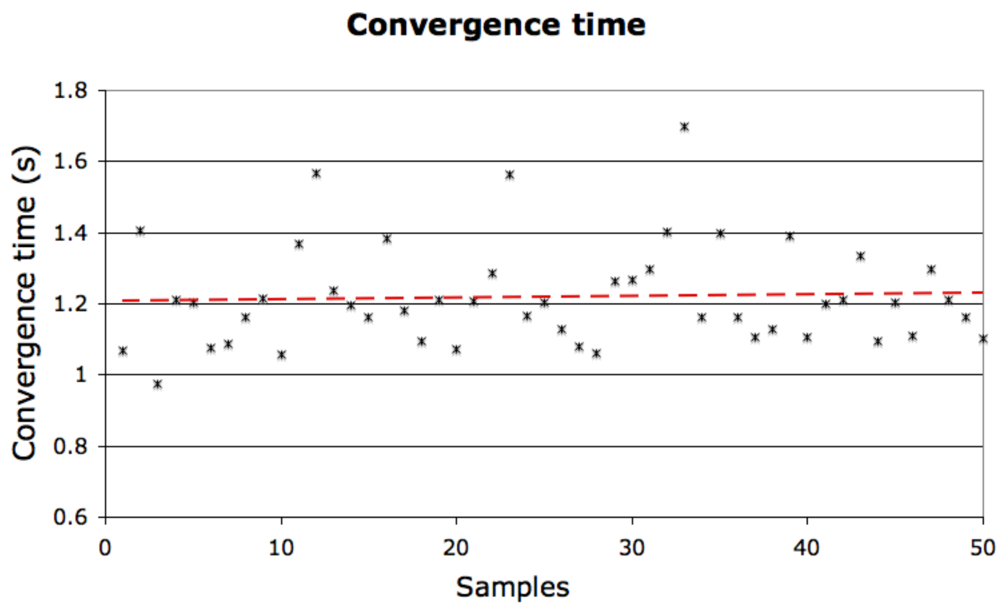

Figure 6.4: LDP convergence time

and 3.3.2. As shown in Figure 6.4, on average it took LDP approximately 1.2 seconds to reach a stable state. However we only allowed each switch that acquired position 0 in any pod to select a random pod number in the range of [0, 3] so that we could test our design under the worst case scenarios where possibly many collisions in initial pod number assignments would occur. The breakdown of this convergence time is summarized below:

- Tree level discovery: 360 - 480 ms
- Pod creation phase: 320 - 530 ms
- Gossip based protocol for ensuring uniqueness of pod numbers: 210 - 660 ms

Chapter 7

Conclusions

This Master's thesis addresses the problem of separating host identity from host location in data center environments. Being able to decouple identity from location allows for more efficient routing/forwarding solutions to be employed in a data center; solutions which do not suffer from any of the well known Layer 2 solution's fallacies and in the same time require no physical device configuration thus extending plug and play functionality for data center networks.

We believe we have managed to partially achieve our goal by presenting a family of Location Discovery Protocols intended to run on top of 3-tier multi-rooted tree topologies, which is what most data centers look like nowadays. These protocols are efficient and fully distributed allowing for our approach to scale up to data centers consisting of tens of thousands of switching elements. Our implementation of the actual protocols on our evaluation platform as well as the simulations we run showed that LDP is effective and takes around 1.2 seconds to converge, although we do believe that this can be further reduced to some value in the order of milliseconds.

This work is an ongoing effort, thus we do not claim to have found the perfect solution yet. We are currently working on multiple fronts such as making our protocols more robust to network failures (which are the common case in such environments) and also being able to create pods in a more efficient manner so as to maximize bisection bandwidth both within each pod and across pods for the general multi-rooted tree topologies. However the basis of our protocols is

presented in this document and we are optimistic that really soon we will have a more complete and sound solution to this problem.

Bibliography

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

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

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

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

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

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

[7] R. N. Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat. PortLand: A Scalable, Commodity Data Center Network Architecture. In *SIGCOMM '09: Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, 2009.

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