Abstract

AMBADKAR, ADITYA RADHAKRISHNA. Support for Virtualization in Data Centers. (Under the direction of Dr. Yannis Viniotis.)

Modern data centers largely depend of virtualization. It has presented engineers with lot of opportunities to innovate.

I would not be exaggerating in saying, live virtual machine (VM) migration was a killer app for virtualization. Due to its numerous benefits pertaining to ease of management of servers, power, policies etc. it soon became one of the most used features in virtual data centres. The flip side of the coin is, right from the beginning live VM migration has been a challenge to network designers and administrators. They need to account for the migration domain, VM state, downtime and what not to support it. The center point of these problems is maintaining the external state associated to the VM such as the live TCP connections, firewall policies, VLAN information etc. after the VM migrates to its destination. Several solutions have been proposed to solve this problem but, they are either restrictive in allowed domain of the migration or they have architectural restrictions where user has to design the data center ground up to apply them.

We solved this problem from a different perspective. The desired outcome of this work was designer should not be required to change their network architecture or build their data center ground up just to accommodate a solution that solves part of all the challenges that data center have to face. We wanted a solution that could be easily integrated in running data centers and to our surprise we found the solution that is simple and easy to scale to large data centers.

In this report, we have defined the problem and the proposed solutions can be easily integrated in existing systems. The solution does not require designers to make adjustments to their designs to accommodate it. For the purposes of this report, we have defined a tree, DC-Tree, and focused on it to propose an algorithm and its seven sub-algorithms to solve the problem. We have also presented mathematical proofs and experimental data supporting the algorithms.

The report starts with introduction to concepts in the domain. We have provided working definitions of ideas like data centres, virtualization, services and their associated problems. The Introduction chapter is followed by problem definition along with the proposed solution. We have presented the mathematical proof of the solution along with algorithm definition. To further test the algorithm we implemented it on our testbed and ran few experiments that have been presented in the Verification and Implementation section and Report ends with notes on how we want to proceed in future in the Conclusion and Future work chapter.

Support for Virtualization in Data Centers

by
Aditya Radhakrishna Ambadkar

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Computer Networking

Raleigh, North Carolina

2013

APPROVED BY:

Dr. Mihail L. Sichitiu	Dr. Michael Devetsikiotis

Dr. Yannis Viniotis
Chair of Advisory Committee

Dedication

To my parents, Ujwala and Radhakrishna Ambadkar

Biography

Aditya was born in the city of black gold (coal), Chandrapur, in state of Maharashtra in India. He is the youngest of Ujwala and Radhakrishna Ambadkar's children.

Aditya finished his schooling and undergraduate education from his home town, Aurangabad. He was actively involved in study of robotics during his undergraduate days and application of various algorithms developed for networks triggered his interest in study of computer networking. During undergraduate years, he studied programming, computer networking and network security that gave him solid foundation in the field.

After graduation, he choose to work for a small company over a lucrative offer from a multinational corporation. This gave him higher responsibilities and presented more opportunities to learn. Impressed by his domain knowledge, the management moved him to the position of contractor for Cisco Systems. In his work with Cisco, he was got the opportunity to delve deeper in the world of virtualization and data centres. He was fascinated by the challenges the field presented and decided to pursue further education in this domain.

He chose Computer Networking at North Carolina State University because of the breadth and depth of courses that it offers in the domain. At NC State, Aditya worked on projects involving enterprise network design, Software Defined Networks (Cisco onePK), survivability of data centers, behavior of TCP in virtual data center environments etc. He has also contributed to open source community via Wireshark.

Looking at his area of interest, Dr. Viniotis accepted him as his research student leading to the work presented in this report.

He looks forward to start his professional career and help build smarter data centres for the future.

Acknowledgements

I would like to thank my parents for the sacrifices they have made for me to be able to complete my graduate education.

I would like to thank my Gurus Dr. Viniotis, Param Pujya Shri Pund Kaka, Param Pujya Shri Narayan Kaka Dhekne Maharaj and Dr. Mukundraj Maharaj. They have made sure that I am always focused on my goals. Their support has not only been in the form of knowledge but also to keep my soul at peace through the ups and downs of life. Their guidance and support has always been a constant and profound influence.

My sisters Mukta, Anupama and Amruta and my brother Abhijit have played a crucial role in keeping me sane all this while.

I would like to thank my friends Deepti, Anand and Mayur for the numerous discussions, coding issues, reviewing material and bearing handling my moments of frustration etc. (the list is long here).

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 Data Centres

There is no single definition of a data center. For the purpose of this report, we define a data center as a collection of compute devices that provide services.

As we have a collection of devices working together; an obvious requirement to build a data center is providing ability to the distributed compute to communicate as effectively as possible. Supporting this requirement is easier said than done. For decades, designers and managers of these networks have struggled to build such networks. This has lead to lot of research in this area.

1.1.1 History

If we consider the concept of having lot of compute power which is co-located, we can guess that information technology industry stated with this idea in form of Mainframes. One of the first such attempts was ENIAC developed by US Army in 1960's. ENIAC was released in the pre-microprocessor era and could perform about 5000 operations per second. Intel released first commercial microprocessor in 1970's. We can see more data centres were built for businesses in this period. We also see efforts towards disaster recovery during this period but the advancement of PCs lead to decrease in popularity of Mainframes [4].

There was a boom in PC industry during 1980's and one of the most important realizations was need of management of IT infrastructure. Companies started consolidating servers in dedicated rooms around 1990's but data centres had to wait for the dot-com boom to regain their popularity. Companies built data centres to aid both system development and operations. Lot of large corporations like Rackspace started their business during the dot-com boom era [4].

Since then data centres have seen tremendous growth. Now with the advancement in internet technologies more and more applications are being delivered through data centres. Large

corporations are running their businesses using data centres and at the same time leveraging them to deliver services to their customers. This popularity of data centres is a result of the fact that todays data centres have tremendous compute and storage capacity, they are highly reliable (tier four data centres have more than 99.995% uptime which means less than 30 mins downtime per year), they are efficient and easy to manage. One of the major technology that led to such data centres is virtualization. Almost all of the data centres today use virtualization in some form.

One of the recent trends due to such growth in virtual data centres is provision of virtual services. A virtual service is a service that is delivered via abstract (typically virtual) infrastructure hence making it scalable on demand, optimized for such virtual delivery but the receiving end never realizes any difference.

We have made an attempt to support virtualization of services in data centres in this report. We want to make support for virtualization in data centres seamless and provide solutions that can be readily integrated in not only the newly built data centres but also the existing one's without significant changes.

1.1.2 Different topologies, their requirements and benefits

Lot of different designs have been proposed over decades for data center fabrics [26, 24, 12, 14, 30]. Following is a brief discussion on some of the most popular research and industry works in the area.

A. **Cisco Data center 3.0:**

Today, the most popular data center architecture is a tiered structure. Cisco Systems released the following as their Data Centres 3.0 architecture in July 2011 [1]. It is built on the concept of investment protection, allowing incremental integration of new technologies and platforms into the existing infrastructure to meet changing business needs. The architecture is based on a solid hierarchical foundation providing high availability and continued scalability. It constructs the network in to a three layered hierarchy consisting of access, aggregation and core layers.

Figure 1.1 illustrates the primary network switching layers of the hierarchical network design reference model for the data center environment. The access layer is where the devices and servers are connected to the data center. It is serving to connect hosts to the infrastructure and thus providing network access. The aggregation layer, as the name suggests performs the function of aggregating the traffic from the access level. It is characterized by a high degree of high-bandwidth port density capacity and is optimized for traffic distribution and link fanout capabilities to access layer switches. The core layer is characterized by a high degree of redundancy and bandwidth capacity and it is optimized

for availability and performance. Hierarchical network design has been commonly used in

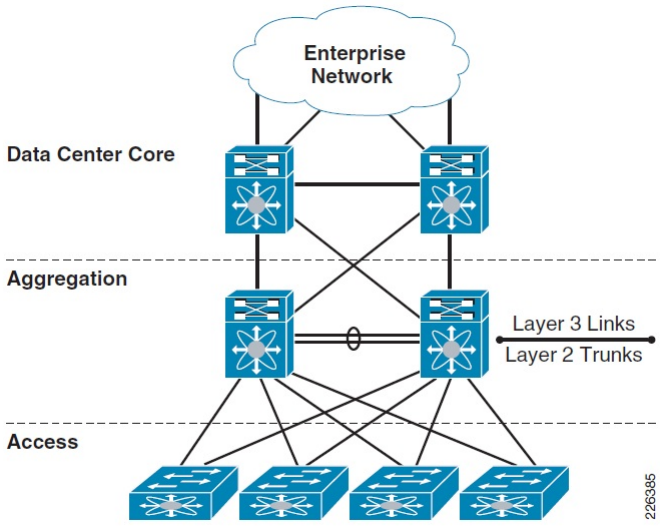


Figure 1.1: Hierarchical Network Design Reference Model [1]

enterprise networking for many years. This model uses redundant switches at each layer of the network topology for device-level failover that creates a highly available transport between end nodes using the network. Data center networks often require additional services beyond basic packet forwarding, such as server load balancing, firewall, or intrusion prevention. These services might be introduced as modules populating a slot of one of the switching nodes in the network, or as standalone appliance devices. Each of these service approaches also supports the deployment of redundant hardware to preserve the high availability standards set by the network topology.

A structured data center environment uses a physical layout that correlates tightly to the hierarchy of the network topology. Careful planning in conjunction with networking requirements and an eye toward flexibility for the future is critical when designing the physical data center environment is important. Taking a modular approach to data center design provides flexibility and scalability in both network topology design and utilization of physical resources.

B. **DCell:**

In DCell [15], Guo et. al. proposed a novel data center architecture based on cell structure instead of the popular tree based structure. DCell is based on three design goals: First, the network infrastructure must be scalable to a large number of servers and allow for

incremental expansion. Second, data center network (DCN) must be fault tolerant against various types of server failures, link outages, or server-rack failures. Third, DCN must be able to provide high network capacity to better support bandwidth-hungry services.

DCell uses a recursively defined structure to interconnect servers. High level DCells are recursively built using low level DCells forming a fully connected graph. DCell architecture addresses various failures at link, server and server-rack levels - by its rich physical connectivity and distributed fault tolerant routing protocol.

Using experimental results authors proved that DCell is resilient to both link and node failures. Also, the paper shows that DCell detects link failures much faster than node failures. Paper also shows that DCell achieves higher throughput than tree based schemes. It achieves scale, fault-tolerance, high network capacity and incremental expansion.

C. **PortLand:**

PortLand [26] is a scalable and fault-tolerant layer 2 data center network fabric. It uses a location based soft state of a pseudo-mac (PMAC) address to achieve routing at layer 2.

Mysore et. al. define an algorithm to dynamically find node's PMAC and maintain the PMAC to actual MAC (AMAC) mapping in a centralized fabric manager. The PMAC calculation algorithm runs a Location Discovery Protocol (LDP) which is also defined in the paper. PortLand relies on its routing protocol for fault tolerance. It uses keep alive messages of LDP to detect a link or node failure. Upon not receiving a keepalive for a configurable period of time, a node assumes a link failure. Fault correction is done with following steps: (1) The detecting node informs the fabric manager about the failure (2) The fabric manager maintains a logical fault matrix with per-link connectivity information for the entire topology and updates it with the new information (3) The fabric manager informs all affected switches of the failure, which then individually recalculate their forwarding tables based on the new version of the topology.

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

D. **VL2:**

VL2 [12] architecture is motivated by extensive study on traffic in Microsoft's data centres. Authors found tremendous volatility in data center workload, traffic, and failure patterns. VL2 consists of a network built from low-cost switch ASICs arranged into a Clos topology that provides extensive path diversity between servers.

VL2's design explores a new split in the responsibilities between host and network using a layer 2.5 shim in server's network stack to work around limitations of the network devices. It uses Valiant Load Balancing (VLB) to spread traffic across all available paths without any centralized coordination or traffic engineering. Using VLB, each server independently picks a path at random through the network for each of the flows it sends to other servers in the data center.

Authors performed a stressed performance test on VL2 and it was found that the combination of Clos with VLB gives acceptable performance in the network. The convergence was tested with all-to-all data shuffle along with disconnection of links between intermediate and aggregation switches until only one intermediate switch remains connected and the removal of one additional link would partition the network. It was seen that even under such severe failures, network converged in less than one sec time.

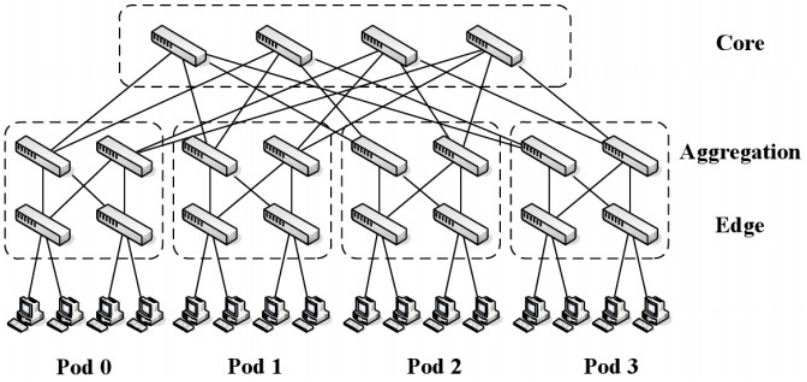
E. **Applying NOX to data centres:**

NOX is a piece of the Software Defined Networking ecosystem. It is specifically, a platform for building network control applications [27]. We have already discussed two network topologies PortLand and VL2. Both have a centralized component to handle address mappings. VL2 and PortLand provide a small set of logically centralized primitives whereas, NOX provides a logically centralized programming environment with a powerful set of abstractions [13].

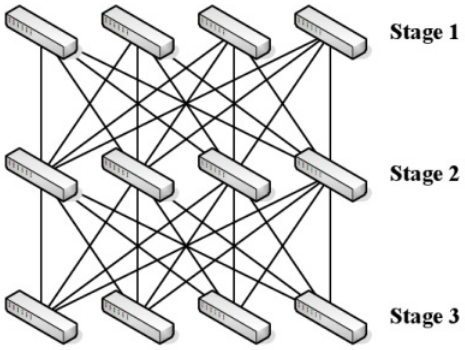
A NOX-managed network consists of one or more controllers running NOX and a set of controllable switches. The controllable switches allow NOX to specify flow entries in their forwarding table. These flow entries are of the form $< header; action >$, where the second item describes the action that should be applied to any packet that matches the header fields listed in the first item. Commonly used actions include: forward to port x, forward to a controller, drop, and forward to port x with the following header fields rewritten with y.

NOX provides fault tolerance through its ability to control routes. Existing switch mechanisms can determine link failures and notify NOX. When informed of a failed link by a switch, a NOX controller then flushes all the flow entries at that switch which use the failed link. When the next packet from each of these affected flows arrives at the switch, the packet will be forwarded to a controller which will then establish a set of flow entries along a new path (avoiding the failed link). In addition, OpenFlow provides the capability to install a second flow entry of a lower priority, allowing NOX to have previously calculated and installed back-up paths in case of failures in order to minimize latency.

Most of these architectures are based off two basic tree structures: Fat tree [21] show in Figure 1.2a and Clos [10], [8] shown in Figure 1.2b. **For the purposes of this report we are focusing only on fat trees.**



(a) Fat tree k=3



(b) CLOS

Figure 1.2: Two most popular trees used in data center architectures

Fat tree is basically a tree with increasing bandwidth as we move from the leaves to the root, hence the name fat tree. A benefit of the fat bandwidth at the root (provided using additional links) is we have lot of equal path multi path (ECMPs) thus providing not only additional bandwidth but also lot of redundancy. This tremendous bandwidth and redundancy is one of the strong reasons why fat trees are so popular in data center architectures. The downside of having large number. of ECMPs is we need infrastructure to manage them. Otherwise we would just end up shutting the ECMPs down to avoid loops. This report provides an innovative solution on utilizing the ECMPs in fat tree networks.

Another strength of fat trees is that they are modular in nature. Modularity helps scalability in data centres as we can easily add new modules to the existing infrastructure thus adding capacity. This property was so important that three giants in the field of data center infrastructure viz. VMWare Inc., Cisco Systems and EMC Corporation came together and formed a company VCE (`www.vce.com`) in late 2009 to manufacture modules of data center. They intend their customers to directly use to these modules to scale their data centers thus saving time and increasing reliability.

Fat tree is a special case of Clos tree. Clos tree defines an identical entry and exit circuit and series of interconnecting circuits that are again identical. Mapping to Figure 1.2b stage 1 can be the entry circuit, stage 3 the exit circuit and stage 2 the internal circuit.

1.2 Problem Definition

In this section we discuss problems in contemporary virtual data centres that motivated us to do this work. Specific problem that we have solved has been defined in section 2.1.

1.2.1 Migration of services

Data centres today provide various services. Some of the examples of such services can be email service provided by say Gmail or Microsoft Exchange which are deployed in data centres, video on demand service provided by companies like Netflix through Amazon's AWS cloud data centres, Xbox 360 and Sony Playstation network etc.

The most popular way to deploy services is to deploy them as services running on virtual machines. Virtual machines are quickly replacing bare metal servers as platform to run services. Virtualization enables hardware and CPU sharing between multiple virtual machines (VMs) i.e. user can run multiple VMs on the same physical host. There is complete isolation provided between different VMs. This provides several advantages such as maximum utilization of resources like CPU, memory etc., exploiting and sharing resource pools, reducing the number of physical servers required, ease of management, effective power management etc.

One of the most important feature of virtualization is live migration of VMs. In live VM migration, a running VM migrates from one physical host to other. The key of live VM migration (which added live to the name) is that the all the parties involved in migration should not realize that anything changed [25], the only change that is noticeable is the slight downtime when the VM stops on source and starts on destination. Lot of commercial solutions like VMWare vMotion, Citrix XenMotion etc. have been developed which address live VM migration.

When a virtual service (service optimized for and delivered via virtual infrastructure) migrates, the infrastructure has to accommodate the change to maintain the "live" nature of

migration. Some of the challenges for the infrastructure to support the change are as follows —

- Maintain IP address of the VM so as to maintain TCP connections
- Maintain network state associated to the VM: VLAN information, ACLs, firewall rules etc.
- Change forwarding information in the network to avoid traffic overhead on the network during convergence

These challenges were strong motivating factors for us while undertaking this research activity.

1.2.2 Failures

The value of data center lies in its availability, hence, failures in a data center can get extremely costly. According to a survey conducted by InformationWeek* in May 2011 downtime cost was about $26.5 Billion.

It is quite obvious that with hundreds of thousands of devices and millions of links, failures will be common in data centres. To give a clear picture of pattern of failures in data centers we present results of a study done by Greenburg et. al. in [12] on failures in Microsoft data centers in 2009. Greenburg et. al. defined a failure as the event that occurs when a system or component is unable to perform its required function for more than 30 seconds. As expected, most failures seen in this study were small in size (e.g., 50% of network device failures involved less than 4 devices and 95% of network device failures involve less than 20 devices) and large correlated failures were rare (e.g. the largest correlated failure seen in this study involved 217 switches). However, significant downtimes were seen: 95% of failures were resolved in 10 mins, 98% in less than 1 hr, 99.6% in less than 1 day but 0.09% lasted greater than 10 days.

When the impact of these failures was analysed it was found that for 0.3% of the failures, all redundant components in a network device group became unavailable. One such incidence involved a core device failure that affected 10 Million users for about four hours.

To avoid such failures network operators place lot of infrastructure behind avoiding such incidences and one of the tool in their arsenal is live VM migration. Once failure is detected in a part of the network administrators migrate virtual machines to keep the services up and running. Typically this migration takes place on an out of band network thus making this a viable option.

As migration is a tool against failures, it is an important motivating factor in this report.

*IT Downtime Costs $26.5 Billion In Lost Revenue - `http://www.informationweek.com/storage/disaster-recovery/it-downtime-costs-265-billion-in-lost-re/229625441`

1.3 Solution Characteristics

This section discusses the goals we had set for ourselves for the purposes of this research.

1.3.1 Zero downtime

Zero downtime during migration indicates none of the involved parties in the migration realize anything changed. This is the most important consideration from the point of view of a live migration.

If VM migration is not characterized by zero downtime it essentially means the service goes down and as we have discussed service downtime in data centres can cost billions of dollars.

1.3.2 Management traffic overhead due to broadcasts

When a VM migrates it is typically migrated in a L2 domain so that the IP address can be maintained. IP address cannot change as it is essential to maintain the open TCP connections and hence to keep the involved parties unaware of the change.

The downside of L2 domain is its broadcast nature of operation. When the VM comes up at the destination the IP address is maintained but its placement in L2 domain changes and a broadcast is generated to find the new location. To avoid large overhead a gratuitous ARP is generated once VM is up. This significantly reduces the overhead on a per migration basis but if we consider thousands of moving VM we can see significant overhead.

This overhead not only consumes bandwidth in the network thus wasting an important resource but also increases potential of broadcast storms. Another important consideration is that the longer it takes for the network to converge, longer is the service downtime, which, as discussed is unacceptable.

1.3.3 Service Level Agreements

A Service Level Agreement (SLA) binds service provider to provide guarantees typically regarding maximum downtime, response time etc. SLAs are important from the point of view of our discussion as providers are typically bound by these factors like response time, service availability etc. via SLAs.

The significance of SLAs in deployments has made research in better support for virtualization in data centres critical.

1.4 Related Work

Due to the significance of data centres and the wide range of associated challenges data centres have attracted significant attention of research community. We have seen promising designs from the academic world as well as large corporations. We are focusing on tree based architectures in this section as our work is currently focused on solving problems for tree based deployments.

Cisco Systems developed a distributed virtual switch, Nexus 1000v (N1kv) [17]. Cisco Nexus 1000V Series Switches are virtual machine access switches that are an intelligent software switch implementation based on IEEE 802.1Q standard for VMware vSphere environments running the Cisco NX-OS Software operating system. Nexus 1000v provides policy-based virtual machine connectivity and mobile virtual machine security and network policy.

With N1kv, virtual servers can use the same network configuration, security policy, diagnostic tools, and operational models as their physical server counterparts attached to dedicated physical network ports. Virtualization administrators can access pre-defined network policies that follows mobile virtual machines to help ensure proper connectivity, saving valuable time for virtual machine administration.

The feature that enables network policies to follow virtual machines solves the issue of fast convergence as well as maintenance of network state but N1kv is essentially a single switch which spans across multiple physical servers. Hence the issues are still unresolved for a generic architecture.

In PortLand [26], Mysore et. al. present a new approach of maintaining VM's IP address via a pseudo MAC mechanism. They have reported liner message exchange over of the usual $O(n^2)$ messages during convergence. Our approach is different from PortLand as we do not need any architecture change as well as need of modification to the network hardware or software (PortLand needs the switches to run custom algorithms).

VL2 [12] achieves some of our goals with the help of a layer 2.5 shim header which requires change in the network elements as well as the server software. Another downside of VL2 approach is the defined all IP environment. Our approach is capable of not only performing in pure L2, L3 or L2 + L3 domains but also is generic enough to work in non TCP/IP environments.

Applying NOX to data centres [13] solves the problems without much of PortLand and VL2 like restrictions but demands the architecture to be built around NOX controller. Our approach keeps off such restrictions on the architecture. One can easily deploy it in any architecture (we have tested it for architecture that can be reduced to DC-Trees, as discussed in section 2.1) and gives administrators to apply it on migration domain of their choice.

Similar benefits of our work are seen over some of the other significant academic works such as: [14], [24], [20], [2] etc. Work presented in [11] Ghorbani et. al. helped us in making the recommendation on the sequence of migration presented in this report.

1.5 Implementation Overview

Our goal was to find a generic solution which does not restrict network designers and operators to an architecture or technology. Our solution focuses on this goal but because of the restrictions on availability of infrastructure in our lab, the implementation had to focus on a specific scenario. We built a generic fat tree topology using Cisco 7200 virtual routers and used Cisco's Software Defined Networking (SDN) technology — onePK [16]. Even tough the implementation was based on a single topology and leveraged Cisco technology our demonstrations prove our ideas and use the technologies as one of the ways of implementation of the functions.

1.5.1 Software Defined Networking

Software Defined Networking (SDN) is a concept developed from PhD work of a Stanford University student, Martin Casado. His work on Ethane a logically centralized architecture for managing security policies [6] brought the idea of controlling the network using software. The idea was to programatically control the network. Instead of forwarding packets based on traditional algorithms running in control plane of an element, we punt packets from data path and forward them according to custom control plane algorithms.

This idea soon became revolution in the networking industry and was adapted by Internet giants like Google to control their networks. OpenFlow [23], a flow level interface to network datapath, is a open standard in this domain. It is supported by most of the major vendors in their network devices. All major vendors are also introducing proprietary implementations of SDN such as Cisco onePK.

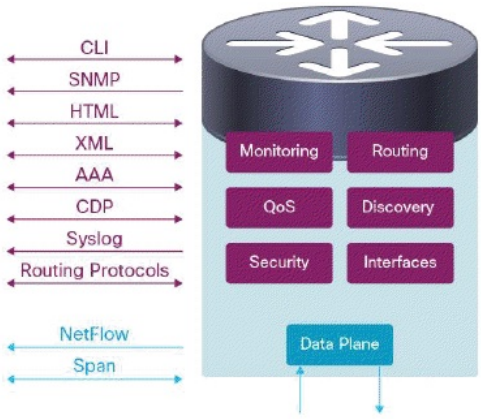


Figure 1.3: How software interacts with networks today

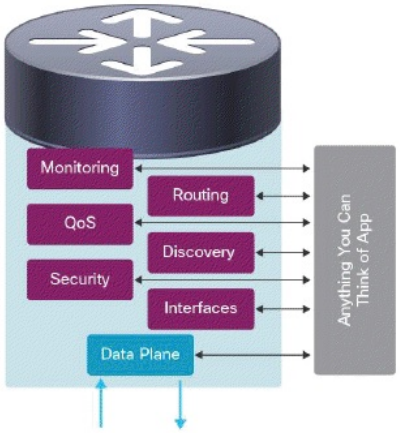


Figure 1.4: How software interacts with networks with onePK

The idea of SDN attracted attention because of its applications in area of network monitoring and management. Currently networks are monitored/managed using various protocols such as SNMP, Syslogs, NetFlow etc. We have our interest in the monitoring/management applications as our goal, to maintain connectivity of a service, is essentially a network management application.

Figure 1.3 and Figure 1.4 shows the difference in how the monitoring/management approach is changed with introduction of SDN technologies, specifically onePK. We can see that non SDN approach has familiar toolkits, multiple knobs, controlled access and many special purpose tools but lacks a programmatic approach and has gaps and inconsistencies. With the use of SDN we can use modern programming languages to develop applications that seamlessly run across multiple platforms. We also have data plane interaction APIs which allow us to modify data plane packets thus extending our control [28].

1.5.2 Cisco ONE Platform Kit (onePK) API

Cisco introduced its SDN package in its Open Networking Environment (ONE) in 2011. The major difference between the open standards and Cisco is the open standard, OpenFlow, open up data path APIs whereas Cisco opened APIs to program all seven layers in its onePK.

As our goal was to maintain the state of the network to support the migration onePK's ability to program the network at all seven layers led us to choosing it over the open standard.

Chapter 2

Algorithm Design

2.1 Problem Definition

To define the problem that we have solved in this report, we are giving working definitions of few terms. We will discussed these terms followed by the formal definition of the problem.

Virtual Service:

A virtual service is any application running on a virtual machine providing a service to its user. The virtual machine will be connected to the network through a VM host. For instance, Microsoft exchange server running on a VM inside a private NCSU cloud is a virtual service offered by NCSU's private cloud.

Network Elements:

A network element is the network hardware which is used to provide access to the services offered by the network. Access, aggregation and core switches form good examples of network elements relevant to the discussion.

Given these definitions the problem definition is as follows:

Given migration of a virtual service from an initial placement to a final placement in a hierarchical network designed as a Fat tree topology, can we find all the network elements that NEED a state change [31] to support the migration?

We want to move a virtualized application from a source to a destination. Assume that a virtual application is running on S_i. Say, the traffic, entering the data centre from core switch, reaches S_i via the orange path, see Figure 2.1. Now we want to migrate the running virtual application from S_i to S_f. A trigger similar to start of VMware vMotion [25] can send message to the controller [13] (which is running the proposed algorithm) indicating a change in service's

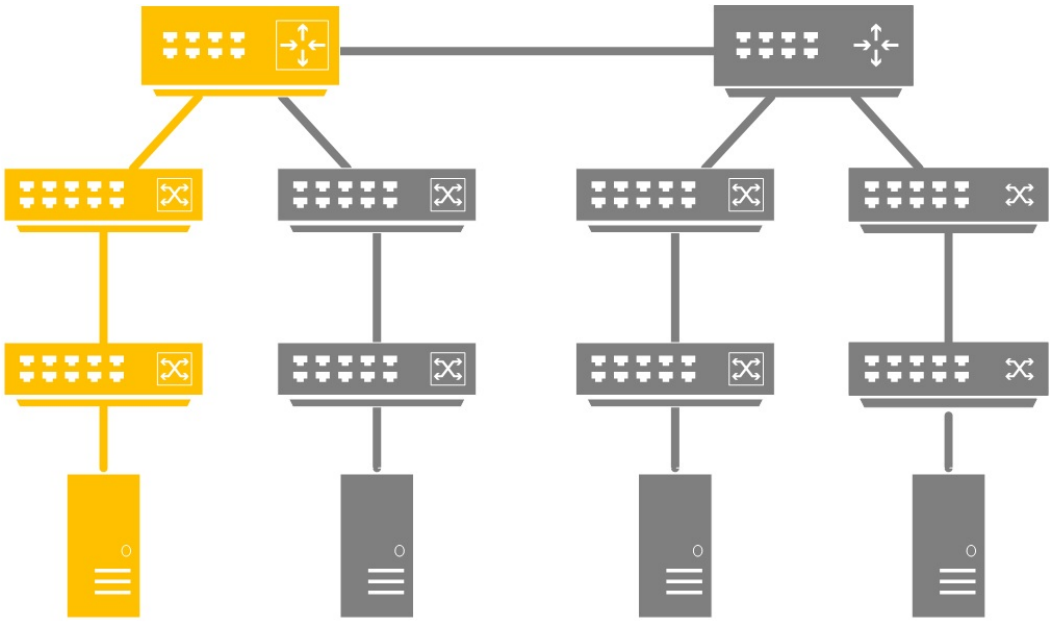


Figure 2.1: Path from core switch to access switch connected to initial placement

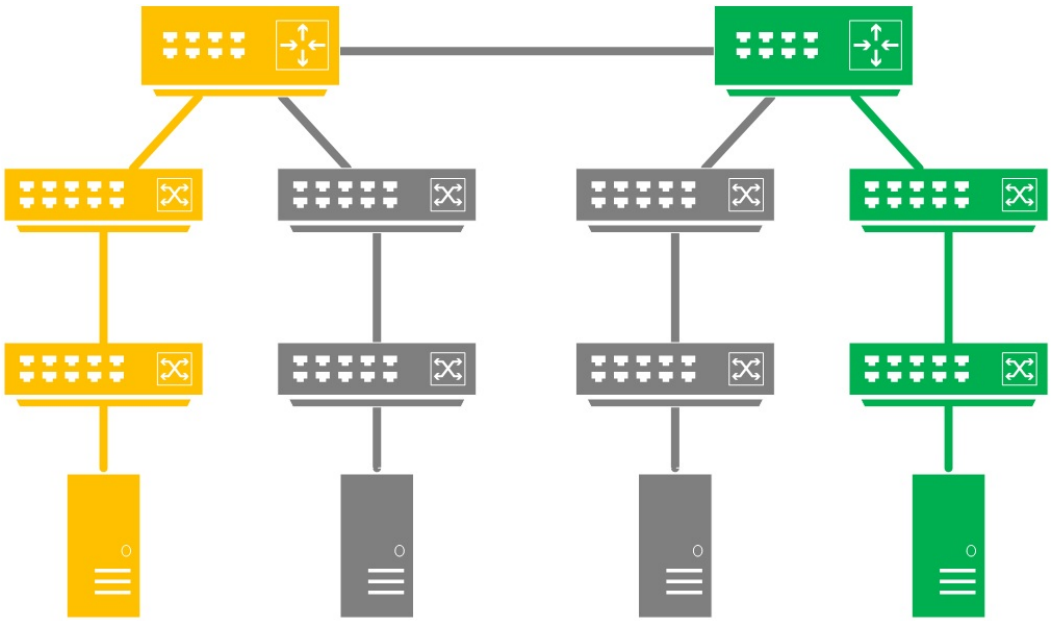


Figure 2.2: Path from core switch to access switch connected to final placement

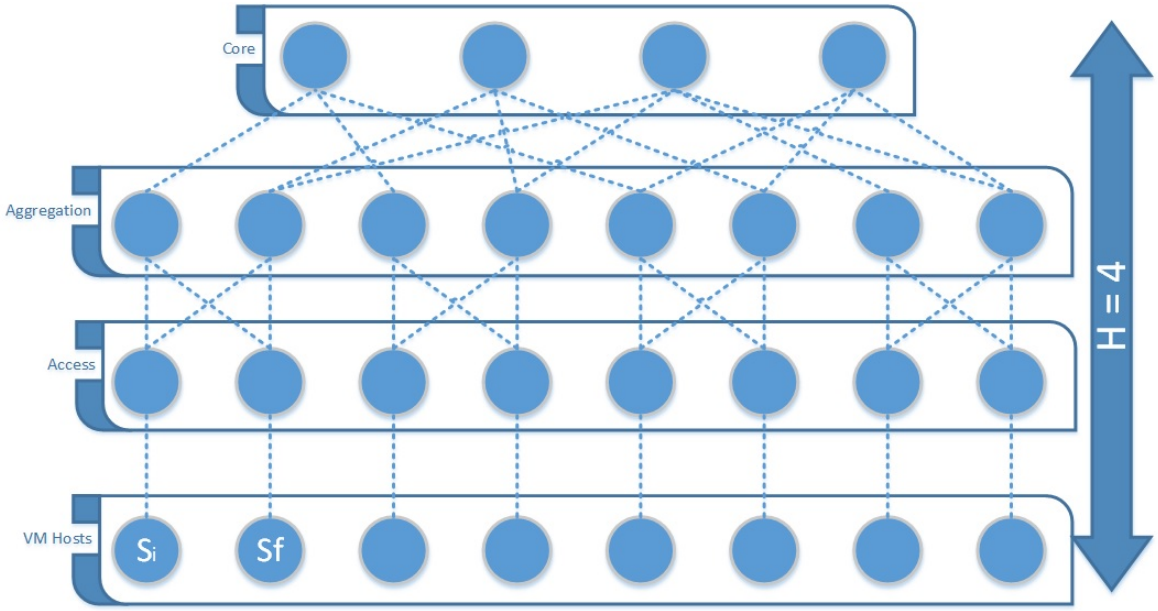


Figure 2.3: DC-Tree of height four

reachability. The controller reprograms the network (*green* and *orange* elements) to support the migration i.e. maintains the service's reachability after the change.

As a simple example, we can add the flow entries to the *green* elements and delete flow entries from the *orange* elements. This can be extended to program the network with other state information associated with the virtual application such as firewall rules, Access Control Lists (ACL), Virtual Local Area Network (VLAN) configuration etc.

Along with finding the elements that NEED a state change to support the migration, we are asking the following questions:

- Can we solve the problem without stringent restrictions on design of network fabric?
- Can we solve the problem without restrictions on IP address allocation?
- Can we solve the problem without need of new native hardware/software features?
- Can solve the problem without restrictions on domain of migration of the virtual service

While working on a solution, we mapped our problem to a new problem, that we call a ***Common Root Problem***. Once the *common root problem* is defined, we have discussed how solving it helps us solve our original problem of finding affected elements to support migration of a virtual service.

The common root problem is defined for a tree [9] (we call this tree a DC-Tree, see Figure 2.3). DC-Tree is a tree with following properties:

1. Has maximum height of four
2. Can have multiple roots as long as it is a connected graph [9]
3. Has at least two leaves

DC-Tree tree is analogues to network topology of a hierarchical and modular virtual data centre where the root is a core element and leaves are the VM Hosts. The middle levels are the aggregation and access elements.

Being a connected graph [9] DC-Tree will always have a path between any of its leaves. If we trace a path from the source leaf to destination leaf we will start climbing up the hierarchy until we reach a point from which we can climb down to reach the destination leaf.

Considering the DC-Tree and this sub tree which takes us from the source to the destination *common root problem* is defined as follows:

> *While travelling from a source leaf to a destination leaf in a DC-Tree can we find a common root and elements of a feasible sub tree?*

Solution of the *common root problem* solves the problem under consideration as follows:

- Source leaf can be our source VM host and the destination leaf can be our destination VM host
- The feasible sub tree is the path the migrating virtual application will take
- Elements of the sub tree that fall on the source side of the sub tree are the elements that need the delete operation of the virtual application state
- Elements of the sub tree that fall on the destination side of the sub tree are the elements that need the add operation of the virtual application state
- Elements that are common to add and delete lists need modification operation of virtual application state i.e. as we are deleting existing state and adding new state from the same element a modification will be an efficient operation. If modify operation is not supported

for the state in consideration special care should be taking in choosing the sequence of add and delete operation to avoid invalid application state

Figure 2.4 shows an example of the source side sub tree, destination side sub tree and the common root.

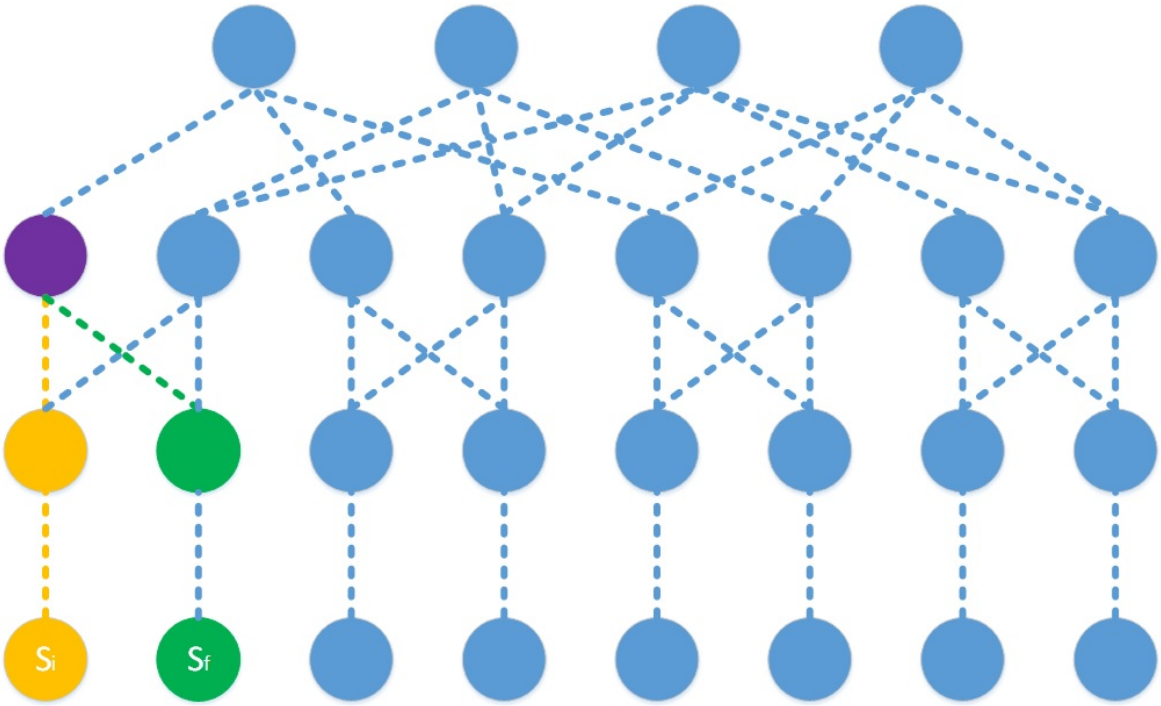


Figure 2.4: DC-Tree with affected elements

2.2 Algorithm Definition

This section defines the algorithm as a pseudocode along with discussion on how the algorithm is validated followed by the validation.

2.2.1 Pseudocodes

We will start this section with definition of some of the terms used in the pseudocode. These definitions will be followed by pseudocodes of the main algorithm 1 which solves the problem under consideration along with other sub-algorithms 2, 3, 5, 6 and 7 that are essentially function

called in the Algorithm 1. After each pseudocode we have presented discussion on the algorithm explaining its working, few comments along with its asymptotic analysis.

We claim Algorithm 1 is generic enough to support any topology as long as a topology has a modular hierarchical architecture, such as fat tree, Clos etc. i.e. if a topology can be reduced to a DC-Tree, Algorithm 1 should stand true to find the affected elements that NEED a state change to support the migration. Other algorithms give the idea of what is the expected output for given inputs by these algorithms and are written specific to the fat tree topology (shown in Figure 1.2a). They can be changed to make the implementation efficient based on the information of the environment.

Parent:

Parent of an element (say e) in a DC-Tree (see 3 on page 16) is an element which is higher in the hierarchy to e and will forward a packet destined for e to an element lower in the hierarchy so that it reaches e

Child:

Child of an element (say e) in a DC-Tree (see 3 on page 16) is an element lower in hierarchy to e and uses e as an immediate element to reach the root

Abstract Node:

An abstract node is a combination of two or more network elements of same level in hierarchy all of which are connected to a common element lower in the hierarchy. We have discussed abstract node, its uses in sections 2.2.3 and 2.3.

Algorithm 1 Find add, delete and modify lists of affected network elements

Input: Trigger of service migration provides source S_i and destination S_f of the migrating service

Output: Add, delete and modify lists of network elements that NEED a state change to support the migration

1: $S_iParentsUngroupped = FindParentsInHierarchy(S_i)$ ▷ Refer to Algorithm 2
2: $S_iParents = IfRequiredAbstractNodes(S_iParentsUngroupped)$ ▷ Refer to Algorithm 3
3: $S_fParentsUngroupped = FindParentsInHierarchy(S_f)$ ▷ Refer to Algorithm 2
4: $S_fParents = IfRequiredAbstractNodes(S_fParentsUngroupped)$ ▷ Refer to Algorithm 3
5: $CommonRoot = FindCommonRoot(S_iParents, S_fParents)$ ▷ Refer to Algorithm 4
6: $AddList = FindAddList(CommonRoot, S_fParents)$ ▷ Refer to Algorithm 5
7: $DelList = FindDelList(CommonRoot, S_iParents)$ ▷ Refer to Algorithm 6
8: $ModifyList = FindCommonElements(CommonRoot)$ ▷ Refer to Algorithm 7

Algorithm 1 starts with finding the parents of the source. This step gives information on which elements are utilized when a packet is sent to the source S_i. This information helps us in finding all possible paths to/from S_i. This is followed by node abstraction the reasons for which have been discussed. We have also discussed few applications of abstraction in section 2.3. The same process is repeated on S_f as we also need the information on possible paths to S_f. Once we have the information on the (abstracted) parents of both S_i and S_f we can easily find a common root. Abstraction of nodes helps avoid a sub-tree splitting in two branches and eventually merging at some other point thus avoiding issues with the process to find common root. With the help of information on the parents and the common root we can proceed to find the expected output of add, delete and modify lists as per algorithms 5, 6 and 7.

Running time of Algorithm 1 depends on the running time of each of its step. We have discussed running time of each step is discussed after the proposed algorithm for that step. We have revisited running time analysis of Algorithm 1 after end of the last step - Algorithm 7. This running time analysis is done based on $n \rightarrow no.\ of\ ports\ on\ an\ element$ and $m \rightarrow no.\ of\ modules\ of\ a\ fat\ tree$. We have defined a module of a fat tree as shown in Figure 2.5

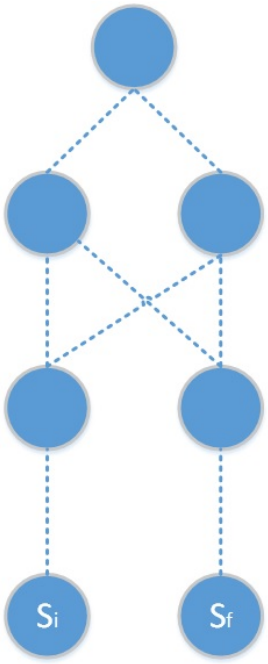


Figure 2.5: Fat tree module used for asymptotic analysis

For each algorithm, we have presented a worst case analysis and a practical cases analysis. All the analysis is based on per migration basis i.e. considering single run of Algorithm 1

By practical case analysis we mean analysis using values that are practical to today's data centers. We feel such an analysis is essential as parameters used for analysis n and m, will take maximum values of few hundreds for n and less than 100 for m. These values are not computational intensive for today's hardware even if running time is factorial, which is orders of magnitude greater than the actual running time of our algorithm.

We think that the communication delay will be significantly greater than the computational time in some of the cases and hence the practical cases analysis compares computational time of the algorithm with communication time. ***This analysis will help designers decide how the network application is deployed in the network so that they achieve best support for virtualization.***

Algorithm 2 Find parents in hierarchy of network elements → *FindParentsInHierarchy()*

Input: Information of VM host (S_i or S_f)

Output: Parents of the VM host in the network hierarchy

1: $AccessElemets = FindNeighborsOneLevelHighInHierarchy(S)^*$

2: $AggregationElements = FindNeighborsOneLevelHighInHierarchy(AccessElemets)$

3: $CoreElements = FindNeighborsOneLevelHighInHierarchy(AggregationElements)$

Algorithm 2 gives us the information on all the parents as defined in the start of this subsection. We start with finding the access elements as the service host will be directly connected to it. Once access is found aggregation can be found using Access (please note that access is used as a point in hierarchy while searching for parents of service host) and same can be done for core.

As discussed earlier in this section we have presented the worst case and practical case runtime analysis of Algorithm 2 —

Worst case analysis: Assuming the access switch has n ports and there are m modules in the fat tree. With m modules, in the worst case aggregation switches will have $m + 2$ (m connections to core switches and 2 to access switches) active ports. Hence the running time of the algorithm will be $O(n + m)$. As in normal cases $n >> m$ we can say that Algorithm 2 will run in $O(n)$.

Practical case analysis: If we consider one module at a time, an element will be connected to one access switch, access will be connected to two aggregation switches and each ag-

*Function *FindNeighborsOneLevelHighInHierarchy()* in implementation specific as it will depend on available features of SDN technology applied

gregation will be connected to at most m core switches. In a practical case, the uplink connections will be less costly than $\theta(n)$ as typically highest speed links are used for uplinks. For instance, a switch will $1G$ ports will typically have two to four $10G$ links to be used as uplinks. With the use of such information the application can be made to find parents with $m + 2 + 1$ searches $\implies$ running time of $\theta(1)$. As discussed earlier this will be insignificant considering the compute power typically available to programmers. Assuming a processor of $3GHz$ capacity, running though 100 numbers will not take more than few hundred nano seconds. Let us assume the computation takes $1\mu s$.

A typical connection link in data center will have 1Gbps capacity and a onePK call to find Cisco Discovery Protocol (CDP) neighbour is 1622 bytes. This data size increases by about 153 bytes per new entry in the CDP table. With these numbers we can say that a call to find one neighbour will take about $12.9\mu s$ plus the queuing delays. Lets approximate this value to $10ms$ which is a typical maximum allowed delay for real time applications in data centers [3]. We can see that, the computation is three orders of magnitude smaller than the connection time and hence we can conclude that *the running time of Algorithm 2 will be affected by the connection time and not the computation time.*

Algorithm 3 finds if node abstraction is required and if required it creates an abstract node, maintains the mapping of abstract node to its members and returns a list of elements which has all connected peers abstracted. As mentioned earlier algorithm can be changed as per implementation and depending on the environment factors. The main purpose of discussing Algorithm 3 is to give users an idea as to what is the expected behaviour of the algorithm.

We have list of all parents of an element as the input to the algorithm. Algorithm starts by sorting this list based on the hierarchy as the key and weight of value as Access $<$ Aggregation $<$ Core. Depending on the input lists we can choose the most efficient algorithm for sorting. Once sorted we start by comparing the first element with the second, if they are equal (i.e. if they are connected peers) we add only the first element to a list which maintains mapping of abstract node to its members. This operation is indicated in Step 17 of Algorithm 3.

Considering the fat tree as shown in Figure 1.2a we will have at the most one abstract node per level and hence Algorithm 3 does not handled the case of having separate abstract nodes per level.

If the elements are unequal we have two possibilities: we have a non-abstract element or we have the last member element of the abstract node in consideration. We have used the $PartOfAbstractNode$ flag to indicate this.

Algorithm 3 Abstract nodes when required → *IfRequiredAbstractNodes()*

Input: List of peers in hierarchy as key-value pair (key being element identifier and value being one of access, aggregation or core)

Output: Modified list with peers grouped†

Initialize array of lists $A[hierarchy][networkelements]$ ▷ Hierarchy takes values access, aggregation or core
SORT($S_i ParentsUngroupped$) ▷ Sort based on the level in hierarchy of the element from access to core. Assume sorted list is SortedList. S_i can be replaced by S_f
$counter \leftarrow 0$ ▷ counter keeps track of the duplicates free list, DFList
$PartOfAbstractNode = FALSE$ ▷ Flag to indicate an element is member of an abstract node
if $S_i ParentsUngroupped.length > 1$ **then**
    for $i := 0 \rightarrow (S_i ParentsUngroupped.length - 1)$ **do**
        if $SortedList[i]$ *neq* $SortedList[i+1]$ **then** ▷ Two abstract nodes are equal when their member elements have one-to-one match
            if $PartOfAbstractNode$ **then** ▷ If element is member of abstract node
                $A[hierarchy][hierarchy\_counter] = SortedList[i]$ ▷ $hierarchy\_counter++$ is separate counter to keep track of no. of access, aggregation and core elements. We can use three counters or single counter which resets when each level is completely explored
                $PartOfAbstractNode = FALSE$ ▷ Resetting the flag as next element is not member of this abstract node
                $DFList[counter] = A[hierarchy]$ ▷ Add abstract node to the duplicate free list
            else
                $DFList[counter] = SortedList[i]$ ▷ Add node to the duplicate free list
            end if
            $counter++$
        else
            $A[hierarchy][hierarchy\_counter] = SortedList[i]$
            $hierarchy\_counter++$
            $PartOfAbstractNode = TRUE$
        end if
    end for
end if
$DFList[counter] = SortedList[S_i ParentsUngroupped.length - 1]$ ▷ Always inserting last element into new array
Return $DFList$, A

In the first case i.e. if we have non-abstract element we avoid the costly insert operation to the list which maintains the mapping of abstract node to its members and directly add element to the list of duplicate free elements. Whereas in the second case i.e. if the element is the last member of an abstract node we add it to both the lists - list maintaining mapping of members to abstract node and duplicate free list.

Running time analysis for Algorithm 3 is presented below —

Worst case analysis: To calculate running time of the algorithm we can start with the $SORT$ operation. For worst case analysis of the algorithm lets assume there is no special knowledge about the data and we are sorting with running time of $nlgn$. We are sorting $S_iParentsUngroupped$ and maximum possible parents for any device would be $1+2+m$ (maximum 1 access + maximum 2 aggregation per access + maximum m core elements) where m is the number of modules. This implies the FOR in Step 6 will run $m+2$ times. In the worst case, we will have 1 compare and 1 insert operations for access elements, 3 compares and $1+2$ inserts (one for insert in sorted list and two for inserts in members of a abstract node list) for aggregation elements and similarly $m+1$ comparisons and $m-1+2$ inserts for core elements which adds up to $2m+10$ operations and we have one final insert in Step 23. Adding it all up we have a running time of $O(m^2)$.

Practical case analysis: In Algorithm 3 we are working on lists that are in memory i.e. there is no communication with network elements so we will not have any unlike Algorithm 2 we have a running time that is completely dependant on the computational time. But as discussed earlier, even tough the algorithm runs in second power of m for all practical purposes, the computational time will be insignificant. Implementation of this algorithm should make use of the knowledge of environment and possible data to make the computation as fast as possible and Algorithm 3 should not affect the placement of the network application in the topology.

[†]Any algorithm that can find duplicates, replace them with an abstract node and keep a record of all the network elements represented by the abstract node can be used here. The return value is the list with all the duplicates replaced.

Algorithm 4 Find common root of two lists → *FindCommonRoot()*

Input: List of elements of grouped parents of S_i and S_f

Output: Common root of lists[‡]

1: **if** $AccessElement.S_iParents = AccessElement.S_fParents$ **then**
2: return $AccessElement.S_iParents$
3: **else if** $AggregationElement.S_iParents = AggregationElement.S_fParents$ **then**
4: return $AggregationElement.S_iParents$
5: **else**
6: return $CoreElement.S_iParents$
7: **end if**

The purpose of Algorithm 4 is to find the common root (as defined in section 2.1) amongst list of parents of S_i and S_f. We start with list parents of S_i and S_f which has connected peers combined in abstract nodes. First step is comparison of access level elements. If common root is found at access level we need not go to higher levels as this will be the most efficient common root. We climb up to aggregation in step 3 and if aggregation is not the common root we can return core without an additional comparison.

If the first available common root is not the desired criteria this algorithm should be replaced accordingly.

Running time of Algorithm 4 is $\theta(1)$. This is already a very fast algorithm and in any practical case it should not affect the placement of network application in the data center.

Algorithm 5 Find list of elements that need add operation → *FindAddList()*

Input: Information on the common root and list of elements in hierarchy of destination substrate

Output: Elements which need add operation

1: return $getChildrenOfRoot(CommonRoot, S_fParents)$

The purpose of Algorithm 5 is to find the list of elements that need ADD operation i.e. we need to add state to these elements to support the migration. The idea is basically all the elements that are children of common root in the S_f parents sub tree. This is true as otherwise we should have found a different common root. So Algorithm 5 simply return children of common

[‡]Finding common root cannot fail as the graph is fully connected and we have abstracted parent elements that are connected peers

root in the S_f parents sub tree.

Running time of Algorithm 5 is same as Algorithm 2 hence Algorithm 5 runs in $O(n)$. The same argument as Algorithm 2 applies to Algorithm 5 in affecting the computational time and the placement of network application in the data center.

Algorithm 6 Find list of elements that need delete operation → *FindDelList()*

Input: Information on the common root and list of elements in hierarchy of source substrate
Output: Elements which need delete operation
1: return $getChildrenOfRoot(CommonRoot, S_iParents)$

The purpose of Algorithm 6 is to find the list of elements that need DELETE operation i.e. we need to remove state from these elements to support the migration. It works similar to Algorithm 5. Here we return all the elements that are children of common root in the S_i parents sub tree.

Running time of Algorithm 6 is same as Algorithm 2 hence Algorithm 6 runs in $O(n)$. The same argument as Algorithm 2 applies to Algorithm 6 in affecting the computational time and the placement of network application in the data center.

Algorithm 7 Find list of elements that need modify operation → *FindModifyList()*

Input: Information of elements that need both add and modify operations
Output: Elements which need modify operation
1: $returnCommonRoot$ ▷ If anything other than common root should be part of this list, we have an error calculating common root

Algorithm 7 identifies the elements which were essential for communication with service before the migration and are also required after the migration. These elements are simply the elements which are part of both the ADD and DELETE lists and hence to find elements that need modification instead of ADD or DELETE Algorithm 7 iterates through both the lists to find common elements.

This algorithm is useful only if MODIFY operation is supported for the state that is being modified. If such operation is not supported we should simply DELETE followed by ADD or vice versa depending on the situation.

Running time of Algorithm 7 is $\theta(1)$. The same argument as Algorithm 4 applies to Algorithm 7 in affecting the computational time and the placement of network application in the

data center.

Asymptotic analysis of Algorithm 1 —

Worst case analysis: Considering the running time of Steps 1 through 8 of Algorithm 1 we can conclude the running time of Algorithm 1 to be $O(n + m^2)$.

If we look for the interpretation of the $O(n + m^2)$ value we can see that as the number of affected elements by the migration increases n will increase (as n will be total no. of port checked during one iteration of the algorithm) and m will stay constant. So we can conclude that as the number of affected elements increases the algorithms computational running time will increase. Its computation time will be significantly affected by the number of modules but this effect will be more or less constant and we can offset it with using smart deployment and design of the network application.

Practical case analysis: In a practical case, $n >> m$ but m^2 will be significant enough to affect n and hence the computational running time of Algorithm 1 will be $O(n + m^2)$..

The computation time for Steps 1 through 8 of Algorithm 1 will still be very low, few hundred μs and because of Steps 1, 3, 6 and 7 i.e. Algorithm 2, Algorithm 5 and Algorithm 6 we will have significant communication wait times. For best performance the network and application designers should customize the application by applying techniques such as running the algorithm in a distributed fashion, periodic learning of the topology, making use of static data, making use of element properties that will not change etc. so that the no. of connections required to find possible parents of an element are reduced.

From point of view of memory only Algorithm 3 performs memory sensitive operations. We have array SORT, FIND, INSERT and DELETE operations. We have also suggested a multidimensional array to store hash map of abstract element with its member elements. In the worst case, Algorithm 3 can consume $O(n^2)$ memory where n is the number of parents of S_i or S_f.

Software designers need to implement these structures based on their environment to keep memory consumption of the algorithm to minimum.

2.2.2 Validation Procedure

To validate the working of the algorithm we have taken a two fold approach:

1. Mathematical proofs
2. Experimental data

The mathematical proofs run step by step through the pseudocode and validate that we are getting expected results and the experiments prove that the results obtained corroborate

the claim of finding *all affected network elements that NEED a state change to support the migration of the virtual application.*

Mathematical proofs are presented in Section 2.2.3 as these present primary proof of the algorithm. The experiments are presented in Chapter 3.

2.2.3 Validation

As discussed earlier, in this report we are focusing on the fat tree architecture and hence all the validation arguments have been made from the point of view of fat tree. However, we have a claim that this algorithm is generic enough to be applied to other modular hierarchical architectures such as Clos which can be reduced to a DC-Tree without modifications. A NCSU student, Aditya Vaja, has validated this algorithm for Clos in an independent project [29]

The goal of the algorithm Algorithm 1 (see page 18) is *to find all the network elements that NEED a state change to support a virtual application migration.* To validate that Algorithm 1 provides this information (provided in form of three lists: Add, Delete and Modify suggesting the operation to be performed on their member network elements) we have organized this section as follows: we start with definition of an abstract node which is essential for the algorithm to support equal cost multi paths (ECMPs) in the network. This definition is followed by reduction of all possible cases of paths from S_i to S_f to three generic cases. Once we complete the reduction we have validated the algorithm by running through it step by step for each of the three generic cases with reasoning of why a particular output is expected (and given by the algorithm) for a given input. This completes the theoretical validation part which is followed by experimental support in chapter 3.

When a server or end host is connected in a fabric based of this tree, the path between the two for any given packet can be reduced to one of the three typologies shown in Figure 2.6. We have introduced a concept of *abstract node* to aid this reduction. As defined earlier, an *abstract node* is a *combination of two or more network elements of same level in hierarchy all of which are connected to a common element lower in the hierarchy. We will abstract network elements when an element lower or higher in the hierarchy can reach the destination through any of the members of the abstract node and exactly which element is chosen doesn't affect the algorithm.* This does add a level of implementation complexity where we need to figure out the changes in members of abstract node especially when peers of the hierarchy are connected. The guarantee in an abstract node is the members will be affected. Abstract nodes also give us an opportunity to run specialized algorithm to choose an member element to satisfy requirements like traffic engineering.

To prove that any placement of initial server S_i and final server S_f can be reduced to three cases as per Figure 2.6 please refer to Figure 2.7. In Figure 2.7 yellow indicates requirement of

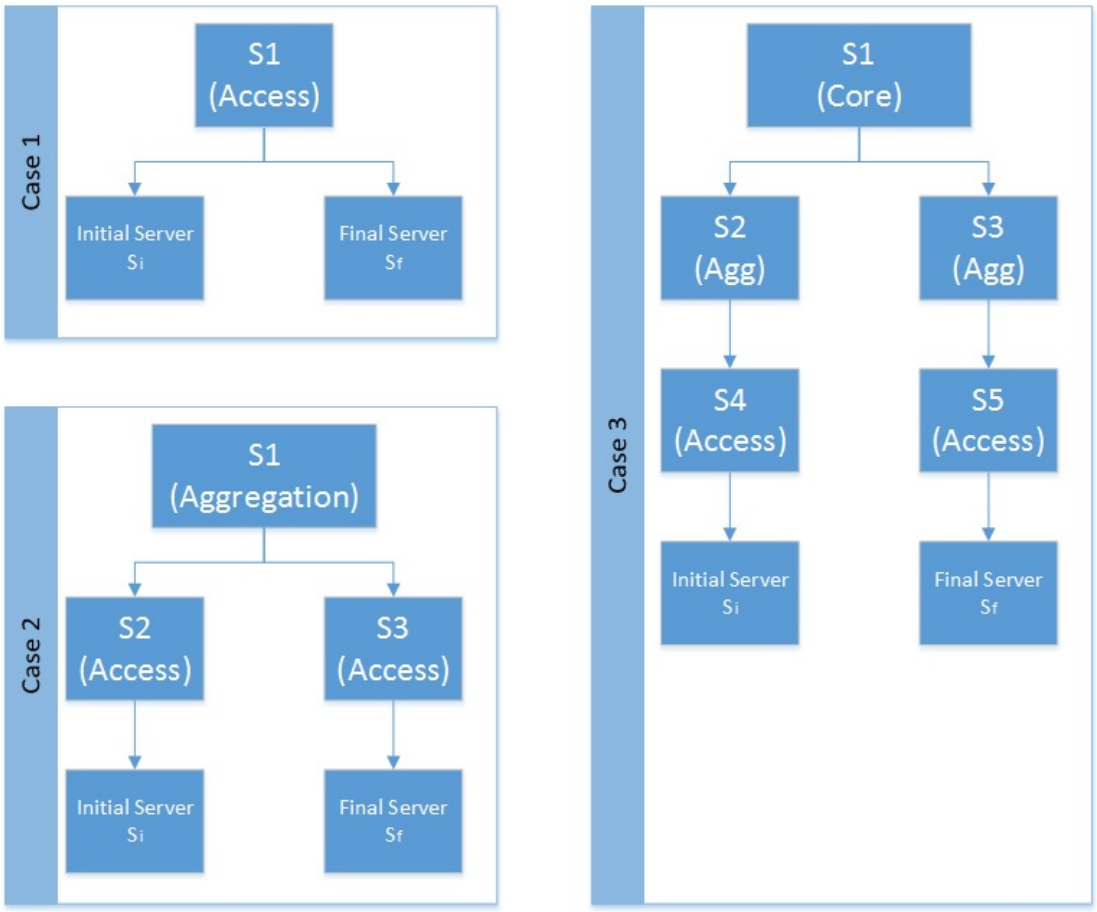


Figure 2.6: Three possible reductions of Fat tree

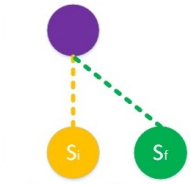
deletion, green indicates requirement of addition and purple indicates requirement of modification. An important point to note here is these path are not the actual connections but path that a TCP (or any other protocol under consideration) can take. It is possible that there is physical connectivity but the port is blocked by say Spanning Tree Protocol (SPT) then we are not taking this path in to account.

If S_i and S_f are connected to the same access switch we have a trivial reduction to case 1 of Figure 2.6. This reduction is as shown in Figure 2.7a.

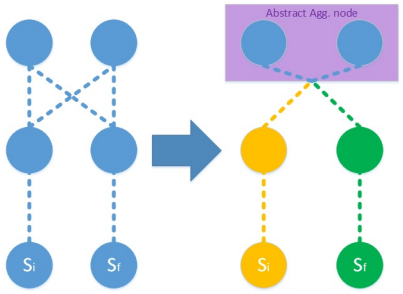
If the path from S_i to S_f climbs up to the aggregation level and then starts climbs down the reduction is as per Figure 2.7b and hence we can place it in case two of Figure 2.6.

If the path from S_i to S_f climbs up to the core level and then climbs down the reduction is as per Figure 2.7c and hence we can place it in case three of Figure 2.6.

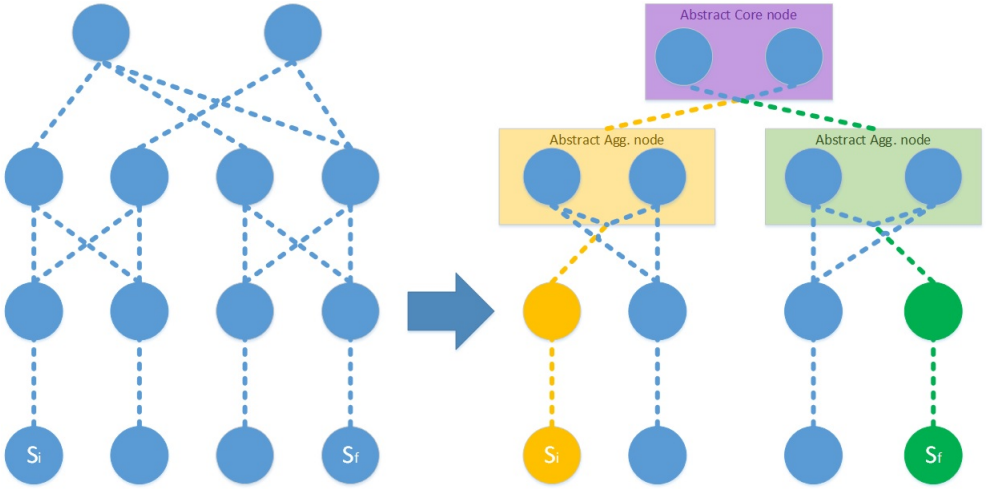
There is an important consideration in the discussion above, these **cases are mutually**



(a) Reduce to Case 1



(b) Reduce to Case 2



(c) Reduce to Case 3

Figure 2.7: Proof of reduction of all possible placements of S_i and S_f to three cases

exclusive from each other. This can be proved as follows: When a path exists from S_i to S_f at access level in a connected graph [9] we will find a path from S_i to S_f at aggregation and core levels but, as per Algorithm 4 discussed on page 24, we will have common root at access level instead of aggregation or core and hence these cases are mutually exclusive. Due to this behaviour it is important to consider path weights based on a criteria of choice (no. of hops, connection speed, congestion on the path etc.) when tracing paths. This also gives us a strong motivation to run algorithm on per migration basis.

It is a well known best practice that the server or VM hosts are to be connected to an access element [19] but if a server is connected to a non-access network element algorithm handles the situation as long as we maintain that actual hierarchy of the element (eg. aggregation or core).

This reduces our scope to three possible cases. Following proofs show that proposed algo-

rithms function for all of these cases. We will run step by step through the algorithm and show that for a given input we are getting expected output with reasoning on why the expected output satisfies the claim of finding *all the network elements that NEED a state change to support the migration*. All the algorithms receive identifying information on the VM as inputs. This is represented by S_i and S_f for source and destination respectively. This information can be any parameter which will allow us to trace a path from the source to destination which can be used by VM migration. An example can be IP address of the VM hosts, IP address of the VM etc.

Case 1 - Source and destination VM host is connected to the same access switch (see Figure 2.6):

Input:

S_i and S_f can be any two distinct VM hosts connected to the same access switch. For the proof we are assuming S_i and S_f both are connected to S1.

Output:

$\{AddList = \phi\}$, $\{DelList = \phi\}$ and $\{ModifyList = S1\}$

Discussion:

$\{AddList = \phi\}$ as the concerned access switch already has the state information so nothing new will be added, $\{DelList = \phi\}$ as the concerned access switch needs the state information so we cannot delete it and $\{ModifyList = S1\}$ as the destination is connected to the same switch (access in this case) and we need to MODIFY the mapping of state information to point from the old VM host to the new VM host.

Another way of interpreting AddList, DelList and the ModifyList is these are the elements that NEED a state change to support the migration. With the state information in S1 modified to point to the new location of the virtual service, network will bring the traffic according to the existing rules till S1 and S1 will take action according to the modified information thus supporting the migration.

Proof:

Algorithm 1 Step 1: $FindParentsInHierarchy(S_i)$

Algorithm 2 Step 1: $AccessElements = S1$

Algorithm 2 Step 2: $AggregationElements = \phi$

Algorithm 2 Step 3: $CoreElements = \phi$

Return value: $S_iParentsUngrouped = S1$

Algorithm 1 Step 2: $IfRequiredAbstractNodes(S_iParentsUngrouped \rightarrow S1)$

Algorithm 3 Step 1: $A[access][\,] \rightarrow NULL, A[aggregation][\,] \rightarrow NULL,$
$A[core][\,] \rightarrow NULL$

Algorithm 3 Step 2: $SortedList = SORT(S1)$

Algorithm 3 Step 3: $counter \rightarrow 0$

Algorithm 3 Step 4: $PartOfAbstractNode \rightarrow FALSE$

Algorithm 3 Step 5: **IF** $S_iParentsUngroupped.length > 1 \rightarrow$ **FALSE** go to **END IF**

Algorithm 3 Step 23: $DFList[0] = S1$

Return value: $S_iParents = \{S1\}, A$

Algorithm 1 Step 3: same as Step 1

Return value: $S_fParentsUngrouped = S1$

Algorithm 1 Step 4: same as Step 2

Return value: $S_fParents = \{S1\}, A$

Algorithm 1 Step 5: FindCommonRoot($S_iParents$, $S_fParents$)

Algorithm 4 Step 1: **IF** $AccessElements.S_iParents = AccessElements.S_fParents \rightarrow$ **TRUE**

Algorithm 4 Step 2: **return** $AccessElements.S_iParents$

Return value: $CommonRoot = S1$

Algorithm 1 Step 6: FindAddList($CommonRoot$, $S_fParents$)

Algorithm 5 Step 1: getChildrenOfRoot($CommonRoot$, $S_fParents$)

Return value: $AddList = \phi$

Algorithm 1 Step 7: FindDelList($CommonRoot$, $S_iParents$)

6 Step 1: getChildrenOfRoot($CommonRoot$, $S_iParents$)

Return value: $DelList = \phi$

Algorithm 1 Step 8: FindModifyList($CommonRoot$)

Return value: $ModifyList = S1$

End of case 1:

We have found the result as expected - $\{AddList = \phi\}$, $\{DelList = \phi\}$ *and* $\{ModifyList = S1\}$

Case 2 - Source and destination VM host is connected to the same access switch (see Figure 2.6):

Input:

S_i and S_f can be any two distinct VM hosts connected to the same aggregation switch

and necessarily to distinct access switches. For sake of the proof we are assuming S_i to be connected to S2 and S_f to S3. S2 and S3 are connected by S1 at aggregation level.

Output:
$\{AddList = S3\}$, $\{DelList = S2\}$ and $\{ModifyList = S1\}$

Discussion:
$\{AddList = S3\}$ as S_f is connected to S3 and we need S3 to have state information of the virtual service to support the migration, $\{DelList = S2\}$ as S_i is connected to S2 and we no longer need S2 to have state information for the virtual service, in fact S2 having state information can cause issues due to stale data and $\{ModifyList = S1\}$ as the source and destination switches, S2 and S3 respectively, are connected to S1 (aggregation in this case) and we need to MODIFY the mapping of state information to point from S2 to S3 to support the migration.

As discussed earlier, another way of interpreting AddList, DelList and the ModifyList is these are the elements that NEED a state change to support the migration. With the state information added to S3, removed from S2 and modified in S1 modified to point to the new location of the virtual service. The traffic that comes to S2 so as to reach the service due to stale information will not find the service connected to S2 (as we have removed the state from S2) and will start looking for the location based on existing algorithms. As soon as it reaches S1 we get a direction and required state to reach the service via S3. Similarly, the traffic that was trying to reach the service via S1 or S3 (eg. come to S1/S3 then go towards S2 to reach the service) will now be able to access the service without any issues thus supporting the migration.

Proof:

Algorithm 1 Step 1: $FindUngrouppedParents(S_i)$

Algorithm 2 Step 1: $AccessElements = S2$

Algorithm 2 Step 2: $AggregationElements = S1$

Algorithm 2 Step 3: $CoreElements = \phi$

Return value: $S_iParentsUngrouped = S2, S1$

Algorithm 1 Step 2: $IfRequiredAbstractNodes(S_iParentsUngrouped)$

Algorithm 3 Step 1: $A[access][\,] \rightarrow NULL, A[aggregation][\,] \rightarrow NULL,$
$A[core][\,] \rightarrow NULL$

Algorithm 3 Step 2: $SortedList = SORT(S2, S1)$

Algorithm 3 Step 3: $counter \rightarrow 0$

Algorithm 3 Step 4: $PartOfAbstractNode \rightarrow FALSE$

Algorithm 3 Step 5: **IF** $S_iParentsUngroupped.length > 1 \rightarrow$ **TRUE**

Algorithm 3 Step 6: **FOR** $i = 0 \rightarrow (S_iParentsUngrouped.length - 1) \rightarrow$ **TRUE**

Algorithm 3 Step 7: *IF* $SortedList[0]$ *neq* $SortedList[1] \rightarrow$ **TRUE**

Algorithm 3 Step 8: *IF* $PartOfAbstractNode \rightarrow$ **FALSE** go to Step 13

Algorithm 3 Step 13: $DFList[0] = S2$

Algorithm 3 Step 15: $counter = 1 \rightarrow$ go to Step 6

Algorithm 3 Step 6: **FOR** $i = 0 \rightarrow (S_iParentsUngrouped.length - 1) \rightarrow$ **FALSE** go to Step 23

Algorithm 3 Step 23: $DFList[1] = S1$

Return value: $S_iParents = \{S2, S1\}, A$

Algorithm 1 Step 3: same as Step 1

Return value: $S_fParentsUngrouped = S3, S1$

Algorithm 1 Step 4: same as Step 2

Return value: $S_fParents = \{S3, S1\}, A$

Algorithm 1 Step 5: FindCommonRoot($S_iParents$, $S_fParents$)

Algorithm 4 Step 1: **IF** $AccessElements.S_iParents = AccessElements.S_fParents \rightarrow$ **FALSE** go to Step 3

Algorithm 4 Step 3: **IF** $AggregationElements.S_iParents = AggregationElements.S_fParents \rightarrow$ **TRUE**

Algorithm 4 Step 4: **return** $AggregationElements.S_iParents$

Return value: $CommonRoot = S1$

Algorithm 1 Step 6: FindAddList($CommonRoot$, $S_fParents$)

Algorithm 5 Step 1: getChildrenOfRoot($CommonRoot$, $S_fParents$)

Return value: $AddList = S3$

Algorithm 1 Step 7: FindDelList($CommonRoot$, $S_iParents$)

6 Step 1: getChildrenOfRoot($CommonRoot$, $S_iParents$)

Return value: $DelList = S2$

Algorithm 1 Step 8: FindModifyList($CommonRoot$)

Return value: $ModifyList = S1$

End of case 2:

We have found the result as expected - $\{AddList = S3\}$, $\{DelList = S2\}$ *and* $\{ModifyList = S1\}$

Case 3 - Source and destination VM host is connected to the same access switch (see Figure 2.6):

Input:

S_i and S_f can be any two distinct VM hosts connected to the same core switch and necessarily to different access and aggregation switch. For sake of the proof we are assuming S_i to be connected to S4 and S_f to S5 in Figure 2.6. S4 is connected to S2 at aggregation level and S5 is connected to S3 at aggregation level. S2 and S3 are connected to each other via core switch S1.

Output:

$\{AddList = S3, S5\}$, $\{DelList = S2, S4\}$ and $\{ModifyList = S1\}$

Discussion:

$\{AddList = S3, S5\}$ as S_f is directly connected to S5 and we need S3 and S5 to have state information of the virtual service to support the migration, $\{DelList = S2, S4\}$ as S_i is directly connected to S5 and we no longer need S2 and S5 to have state information for the service, in fact S2 and S4 having state information can cause issues due to stale data and $\{ModifyList = S1\}$ as the source and destination aggregation switches, S2 and S3 respectively, are connected to S1 (core switch in this case) and we need to MODIFY the mapping of state information to point from S2 to S3 to support the migration of the virtual service.

As discussed earlier, another way of interpreting AddList, DelList and the ModifyList is these are the elements that NEED a state change to support the migration. Similar to the discussion for Case 1 and Case 2 we can show that with addition of information to S3 and S5, deleting it from S2 and S4 and modifying it in S1 we can support the migration.

Proof:

Algorithm 1 Step 1: $FindUngrouppedParents(S_i)$

Algorithm 2 Step 1: $AccessElements = S4$

Algorithm 2 Step 2: $AggregationElements = S2$

Algorithm 2 Step 3: $CoreElements = S1$

Return value: $S_iParentsUngrouped = S4, S2, S1$

Algorithm 1 Step 2: $IfRequiredAbstractNodes(S_iParentsUngrouped)$

Algorithm 3 Step 1: $A[access][\,] \rightarrow NULL, A[aggregation][\,] \rightarrow NULL,$ $A[core][\,] \rightarrow NULL$

Algorithm 3 Step 2: $SortedList = SORT(S4, S2, S1)$

Algorithm 3 Step 3: $counter \rightarrow 0$

Algorithm 3 Step 4: $PartOfAbstractNode \rightarrow FALSE$

Algorithm 3 Step 5: **IF** $S_iParentsUngroupped.length > 1 \rightarrow$ **TRUE**

Algorithm 3 Step 6: **FOR** $i = 0 \rightarrow (S_iParentsUngrouped.length - 1) \rightarrow$ **TRUE**

Algorithm 3 Step 7: *IF* $SortedList[0]$ *neq* $SortedList[1] \rightarrow$ **TRUE**

Algorithm 3 Step 8: *IF* $PartOfAbstractNode \rightarrow$ **FALSE** go to Step 13

Algorithm 3 Step 13: $DFList[0] = S4$

Algorithm 3 Step 15: $counter = 1 \rightarrow$ go to Step 6

Algorithm 3 Step 6: **FOR** $i = 0 \rightarrow (S_iParentsUngrouped.length - 1) \rightarrow$ **TRUE**

Algorithm 3 Step 7: *IF* $SortedList[1]$ *neq* $SortedList[2] \rightarrow$ **FALSE**

Algorithm 3 Step 8: *IF* $PartOfAbstractNode \rightarrow$ **FALSE** go to Step 13

Algorithm 3 Step 13: $DFList[1] = S2$

Algorithm 3 Step 15: $counter = 2 \rightarrow$ go to Step 6

Algorithm 3 Step 6: **FOR** $i = 0 \rightarrow (S_iParentsUngrouped.length - 1) \rightarrow$ **FALSE** go to Step 23

Algorithm 3 Step 23: $DFList[2] = S1$

Return value: $S_iParents = \{S4, S2, S1\}, A$

Algorithm 1 Step 3: same as Step 1

Return value: $S_fParentsUngrouped = S5, S3, S1$

Algorithm 1 Step 4: same as Step 2

Return value: $S_fParents = \{S5, S3, S1\}, A$

Algorithm 1 Step 5: FindCommonRoot($S_iParents$, $S_fParents$)

Algorithm 4 Step 1: **IF** $AccessElements.S_iParents = AccessElements.S_fParents \rightarrow$ **FALSE** go to Step 3

Algorithm 4 Step 3: **IF** $AggregationElements.S_iParents = AggregationElements.S_fParents \rightarrow$ **FALSE** go to Step 6

Algorithm 4 Step 6: **return** $CoreElements.S_iParents$

Return value: $CommonRoot = S1$

Algorithm 1 Step 6: FindAddList($CommonRoot$, $S_fParents$)

Algorithm 5 Step 1: getChildrenOfRoot($CommonRoot$, $S_fParents$)

Return value: $AddList = S3, S5$

Algorithm 1 Step 7: FindDelList($CommonRoot$, $S_iParents$)

6 Step 1: getChildrenOfRoot($CommonRoot$, $S_iParents$)

Return value: $DelList = S2, S4$

Algorithm 1 Step 8: FindModifyList($CommonRoot$)

Return value: $ModifyList = S1$

End of case 3:

We have found the result as expected - $\{AddList = S3, S5\}$, $\{DelList = S2, S4\}$ *and* $\{ModifyList = S1\}$

As a final note, on the theoretical validation section we would like to discuss why we think this algorithm successfully achieves our goals. These are the same questions that we asked while defining the problem in Section 2.1.

When we started thinking about the problem of finding all network elements that need a state change to support a virtual service migration there was an important consideration to keep in mind, the solution should be practical to implement without affecting the existing network environment. We did not want to propose a solution which restricts designers with requirements similar to those listed below —

- Specific design of network fabric. Till now we can say with confidence that we can support at least one forms of modular and hierarchical architectures: fat tree and it is well known that designers have lot of other motivating factors to choose such designs. Claim is we should be able to reduce any modular hierarchical network to a DC-tree but we have not yet validated the claim on other similar topologies and it is a work in progress
- Restrictions on IP addressing, in fact we don't even need the network to be TCP/IP
- Need to add new native hardware features
- Restrictions on virtual service migration domain

The proposed Algorithm 1 imposes only two conditions:

- Topology should be reducible to DC-Tree
- For any practical importance we should be able to find parents and children of network elements along a service migration path

Hence we think we succeeded in our aim.

2.3 Discussion

The focus of this section is discussion on how some of the claims made in this chapter are possible. We have tried to prove and validate the claims as and when we made them. This section focuses on two claims proving which would have been too length and slightly out of context for the discussion.

Claim 1: Algorithm can be extended to other modular hierarchical topologies like Clos

As the name suggests, modular and hierarchical topologies have small modules of hierarchical elements which are repeated. These topologies are highly scalable and easy to maintain hence they are very popular in data centre architectures.

To apply the algorithm to any modular hierarchical design we start by finding a suitable size module which can act as an independent entity. For instance, in the fat tree we have single core switch with block of two aggregation and two access switches as a possible module. We have leveraged this module in the proofs as well as the implementation. Once we drill down to such a module generic implementation of functions that form Algorithm 2, Algorithm 3 can be implemented. Once we have the information of the grouped parents, Algorithm 4, Algorithm 5 and Algorithm 6 can be computed and as Algorithm 7 takes inputs from Algorithm 5 and Algorithm 6 it can also be computed.

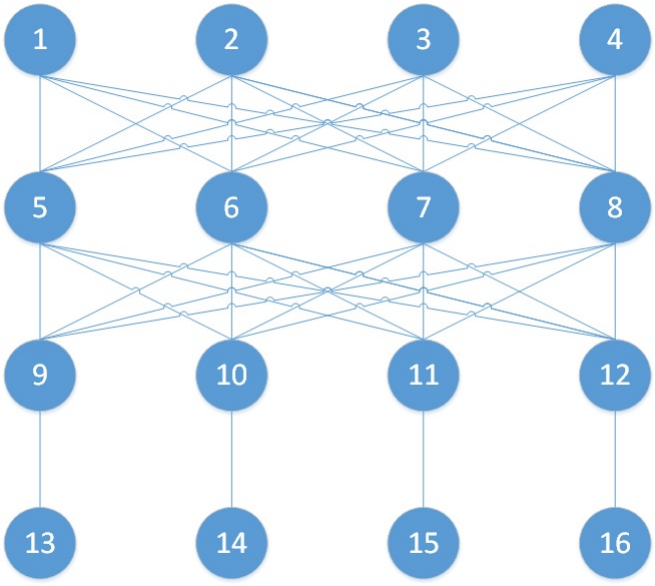


Figure 2.8: DC-Tree made from Clos topology

Let us discuss this claim using an example of Clos topology. Figure 2.8 shows a DC-tree which represents Clos as shown in Figure 1.2b of Chapter 1. We can see that we have tree of height four where the top level can be similar to the core followed by aggregation and access where the bottommost level being the VM hosts.

In this particular case we will consider each level as a module. Assuming we want to migrate from VM host 13 to VM host 15 we will be able to see the outputs of Algorithm 1 as follows:

Algorithm 1 Step 1: $S_iParentsUngroupped = \{9, 5, 6, 7, 8, 1, 2, 3, 4\}$

Algorithm 1 Step 2: $S_iParents = \{9, \{AbstractNodeWithMembers\{5, 6, 7, 8\}\}, \{AbstractNodeWithMembers\{1, 2, 3, 4\}\}\}$

Algorithm 1 Step 3: $S_fParentsUngroupped = \{11, 5, 6, 7, 8, 1, 2, 3, 4\}$

Algorithm 1 Step 4: $S_fParents = \{11, \{AbstractNodeWithMembers\{5, 6, 7, 8\}\}, \{AbstractNodeWithMembers\{1, 2, 3, 4\}\}\}$

Algorithm 1 Step 5: $CommonRoot = \{AbstractNodeWithMembers\{5, 6, 7, 8\}\}$

Algorithm 1 Step 6: $AddList = 11$

Algorithm 1 Step 7: $DelList = 9$

Algorithm 1 Step 8: $ModifyList = \{AbstractNodeWithMembers\{5, 6, 7, 8\}\}$

Claim 2: Abstract node can be used to apply specialized algorithms to perform functions such as load balancing, traffic engineering etc. on its member elements.

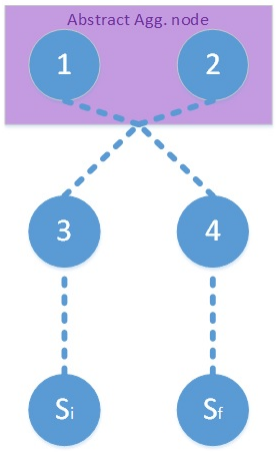


Figure 2.9: Representation of abstract node on aggregation level

If we use Figure 2.9 as a reference figure, there are two points which facilitate Claim 2. If we focus our attention at the point between say node 3 to the abstract node, this is the point where we should run our special algorithms for ingress traffic to abstract node members node 1 and node 2. Similarly the egress traffic can be concentrated at another virtual point between aggregation and core levels to run egress algorithms.

Let us focus our attention to the first point. The first point is of special interest as algorithms such as load balancing and traffic engineering make lot of sense at the ingress port. Assume traffic from node 3 to abstract node. We punt the ingress traffic for special analysis where we can run load balancers for abstract node members or implement a token bucket to police ingress traffic. It is important to note here that these algorithms will run in software (possibly a network application running as a process in elements operating system).

Chapter 3

Implementation and Verification

3.1 Implementation Overview

After theoretical discussion in Chapter 2 we have tried to explain how each of the requirement and claim is satisfied experimentally in this chapter.

This section gives an overview on how we have implemented the proposed algorithm and discusses what we are trying to achieve from these experiments. Each of the experiment discusses in details why and how it was performed.

We have taken an Object Oriented Programming leveraging Software Defined Networking (SDN) approach for implementation of the algorithm. The implementation was done using object oriented programming in $Java^{TM}$ [18] and the interaction with network elements implemented using Cisco's software defined networking environment API onePK [16].

Apart from its usual benefits a very strong motivation behind using object oriented approach was making the implementation independent of the underlying network programming technology. Current implementation uses onePK function calls but these functions are implemented as abstract functions. So one can add code to find the supported programming method: onePK, OpenFlow [23], command line interface (CLI) etc. and call expected function accordingly. Couple of other considerations that lead to choice of Java as implementation language were availability of Java onePK APIs and Java's rich support for building GUIs. GUI was important as we wanted to build GUI simulators to demonstrate working of the algorithm.

"Do SDN, it's cool."

— Joe Clarke *

Along with this advice from Joe, we had another strong motivation to choosing SDN to program the network. SDN interfaces have provided a very powerful way to manage each flow in the network as per users requirement. These flows can be process using custom algorithms thus giving us the ability to fully leverage the potential of the proposed algorithm. The implementation of algorithm benefits with following reasons:

- With SDN we can run custom algorithms without changing the vendor hardware and software thus allowing us to process packets/program network based on out algorithm
- We can Add, Delete and Modify flow entries on network elements from the network application thus helping us to demonstrate experiments like experiment III
- We can receive triggers from network for very specific flows and apply actions on them based on the output of the custom algorithm eg. punt a packet with a given 5-tuple and always send it to the default gateway

Considering above points it is quite clear that the implementation would be very complex and highly error prone with current automation techniques.

While choosing the technology options for SDN we had choice of onePK and OpenFlow. Choice of onePK was easy in our case as we were looking for manipulation of more than the forwarding information. We wanted to take the complete state of the VM in consideration and onePK provided the L1-L7 visibility in properties over OpenFlow which mainly talks about flow entries [16].

Some of the limitations that we faced in performing experiment that were not exactly algorithm issues were —

- We did not have actual hardware that could support onePK hence we used IOL instances. This prevented us from performing performance testing of our algorithm against existing well known solutions
- onePK currently is supported only on routers and hence we could not simulate a real data centre environment which typically is a L2 + L3 domain. This also lead to simulated VM migration instead of actual VM migration for the experiments
- Infeasibility of creating networking larger than 16 elements i.e. 2 modules (leading to a max. of 8 affected elements)

*Joe Marcus Clarke is a Distinguished Engineer with Cisco Systems Inc. and a FreeBSD developer. To find more about him please visit `http://www.marcuscom.com`

3.2 Testbed Details

We have described the global testbed details in this section. We feel this information is important for reproducibility of the results mentioned in this Chapter. Any experiment specific changes to the environment, both software and hardware, are discussed along with the experiments.

We have used Cisco's IOS on Linux (IOL) virtual machines to run virtual instances of Cisco 7200 routers. All the virtual routers run on single VM. The Java based network application is also running on the same VM. Following are the details on each component:

VM:

- CPU: 1 x 2.925 GHz, Intel Xenon X3470 processor
- RAM: 2 GB
- OS: CentOS 5 Linx kernel 2.6.35.14-106.fc14.i686
- No memory or CPU over provisioning

Application Environment:

- Java 1.6.0_20
- OpenJDK runtime environment IcedTea6 1.9.10
- OpenJDK client VM (build 19.0-b09)
- No environment variables set

IOL Instance:

- CPU: Inter x86 processor
- RAM: 512 MB
- OS: Cisco IOS Software, Experimental version 15.3 (20120814:192507) [surf-pi20-eft 118]

VM Host:

- Dell PowerEdge R310
- CPU: 4 x 2.925 GHz, Intel Xenon X3470 processor with hyperthreading active
- RAM: 16 Gb
- OS: VMWare ESXi 5.0.0 build 469512.

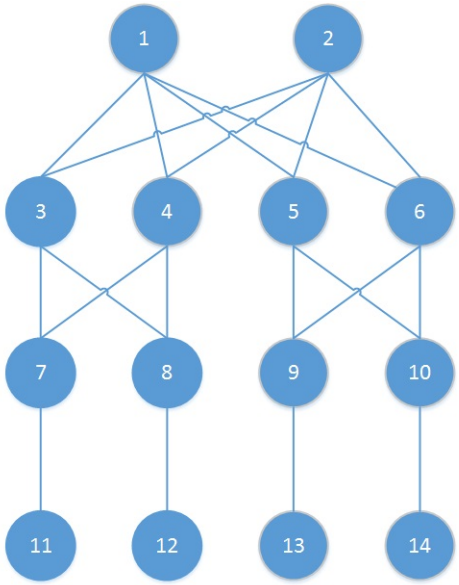


Figure 3.1: Network topology used for Experiments

3.3 Verification

This section experimentally validates the claims or conclusions found in Chapter 2.

3.3.1 Experiment I

This experiment aims at verifying that Algorithm 1 *finds the elements that NEED a change to support the migration.*

Description:

For verification of the Algorithm 1 we implemented it as a onePK application and ran cases 1, 2 and 3 as discussed in Section 2.2.3 on the topology in our lab. Just to reiterate, following are the three cases along with their expected inputs and outputs. All the cases are presented in reference to Figure 3.1. Elements 7, 8, 9 and 10 are access elements, 3, 4, 5 and 6 are aggregation elements and 1 and 2 are core elements. Please note that we are saying elements as our testbed consists of routers (currently onePK only supports Cisco ISR G2, ASR G2 and Nexus routers [16]).

Case 1: Source and destination VM host is connected to the same access element. Expected input S_i and S_f can be any two distinct VM hosts connected to the same access elements. Expected output assuming access element to be 7 and S_i and S_f to be servers hanging off 7 — $\{AddList = \phi\}$, $\{DelList = \phi\}$ and $\{ModifyList = 7\}$.

Case 2: Source and destination VM host is connected to the same access element. Expected input S_i and S_f can be any two distinct VM hosts connected to the same abstracted aggregation elements. Expected output assuming abstracted aggregation elements to be 3, 4 and S_i and S_f to be 11 and 12 respectively — $\{AddList = 8\}$, $\{DelList = 7\}$ and $\{ModifyList = 7\}$.

Case 3: Source and destination VM host is connected to the same core element. Expected input S_i and S_f can be any two distinct VM hosts connected to the same abstracted core element. Expected output assuming core element to be 1 and S_i and S_f to be 11 and 13 respectively — $\{AddList = 5, 6, 9\}$, $\{DelList = 3, 4, 7\}$ and $\{ModifyList = 1\}$.

Setup:

Network topology was designed as shown in Figure 3.1. As all the elements are L3 devices each link is assigned its own subnet. S_i and S_f are simulated using IOL routers. Management IPs of S_i and S_f are used as initial and final location identifiers.

Discussion:

It was observed that the expected outputs were seen for all the three cases.

3.3.2 Experiment II

This experiment aims at validating the relation of algorithms running time as per two major criteria — number of affected elements and summation of number of active ports and number of modules in the fat tree. It was discussed in Section 2.2.1 that in the worst case algorithm runs in $O(n + m^2)$ and in a practical case algorithms running time will be affected by the number of connections it has to make i.e. number of elements that will be affected (as there will be one connected per element).

Motivation:

Algorithm claims that in the worst case it will run as per number of affected elements i.e. number of connections that need to made to find information instead of the computation time. This claim can be verified experimentally by studying the growth of time required to find the elements that need a state change with respect to the mentioned criteria.

Knowledge of the relation between the number of affected elements to the time required is important as this is a measure of the scalability of the algorithm.

Description:

In Chapter 2 we have seen the algorithms run time to be $O(n + m^2)$ for a fat tree shown in Figure 1.2a. From Figure 1.2a we can see that m is 2 and hence $m^2 = 4$ will always be

an insignificant value for any switch of practical size hence we can say that the algorithm for Figure 1.2a will run in $O(n)$.

Considering a practical case the claim is we should see a performance as per the number of elements that are affected i.e. number of elements that we need to connect to as the communication overhead should be significantly greater than the computation overhead.

In this experiment we show two graphs showing comparison of avg. computation time with linear growth rate one as per number of affected elements and other per number of active ports. The interpretation of the observed running time as per the graphs is presented in Results and Discussion.

Setup:

This experiment uses the global setup with addition of elements to simulate S_i and S_f being under same access switch, same abstracted aggregation and same abstracted core.

To calculate average timings we repeatedly executed the experiment by changing the access, aggregation and core elements that will be affected while keeping the constraint on the number of devices that are affected.

As discussed in section 3.1 the worst case on the number of affected elements was limited to eight and as each connected on IOL device had to be configured manually alongwith manual assignment of IP subnets hence limiting the worst case on number of ports to 20.

Result:

As a part of comparing running time of the algorithm and finding the most significant factor affecting it, we calculated average running time with different number of active ports as well as different number of affected elements. These values are plotted in Figure 3.2 and Figure 3.3.

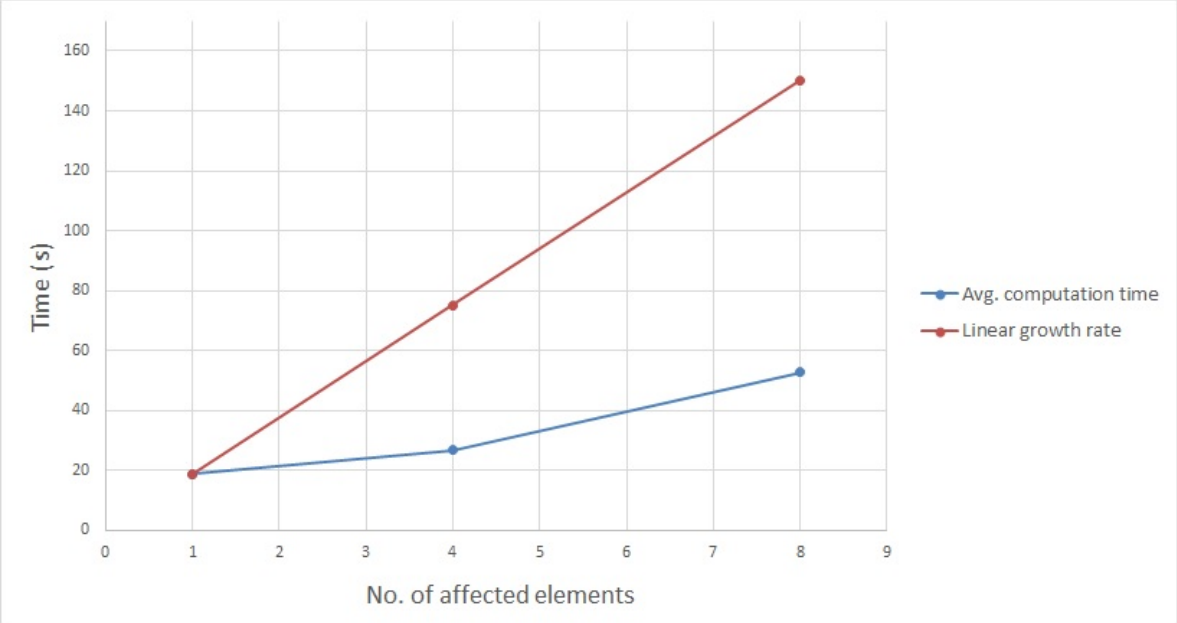


Figure 3.2: Comparison of linear growth rate with actual growth rate with respect to number of affected elements due to the migration

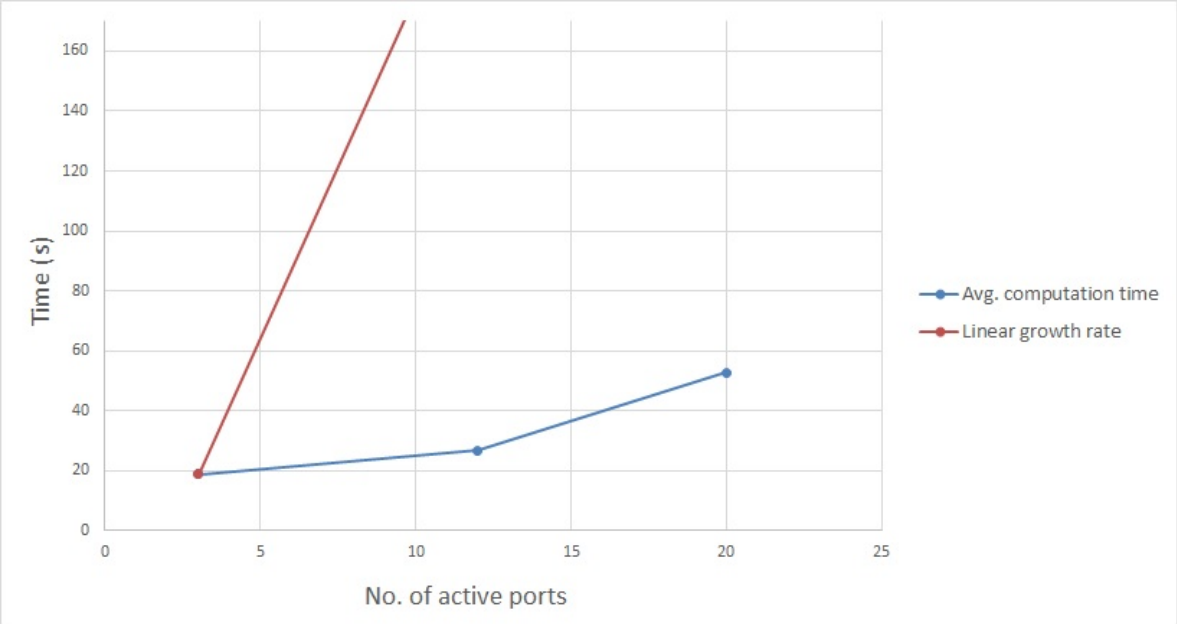


Figure 3.3: Comparison of linear growth rate with actual growth rate with respect to number of active network interfaces involved in the migration

We can see from Figure 3.2 and Figure 3.3 that both number of active ports and number of affected elements grow faster than the actually observed running time. This is consistent with the $Big - Oh$ nature of the analysis.

We can also observe that number of affected elements give a much tighter bound as compared to number of active ports which validates the claim that the running time of the algorithm is closely relates to the number of connections that have to be made as compared to number of compute time required for the algorithm.

An important point to note is the observed graphs are based on values calculated on out testbed and on this particular testbed a tighter bound can be calculated but it will not give us a generic value.

This experiment demonstrates the running time in a completely L3 domain but same results are expected in both pure L2 and L2 + L3 domains. In fact, as mentioned before we expect same behaviour even in non TCP/IP networks but we could not demonstrate due to lack of infrastructure.

3.3.3 Experiment III

Most important requirement to support migration of virtual services in a data centre is the ability to maintain the state of the service as well as the state associated with the service. We have well known technologies to maintain the state of the service some of the popular ones discussed in [25], [7], [22] etc. This experiments shows that using the proposed algorithm Algorithm 1 we can migrate state of a service.

Description:

In this experiment we demonstrate that we have achieved that goal of maintaining the entire network state associated with the VM and hence supporting migration even outside the expected domain of migration.

As discussed earlier we have a pure L3 domain in our testbed. Live migration is not typically supported in L3 domain as we need to maintain the IP address for migration to be live. So, for the demonstration, we migrated the entire subnet of VMs from source to destination and changed the routing to support the migration. This demonstrates that the algorithm can support migration of state associated with the VMs. The whole subnet has to be migrated as we cannot connect same subnet on two sides of a router.

We observed RIB entries on all elements on the testbed to confirm that expected changes were seen and more importantly unexpected changes were not seen.

Setup:

The setup is same as the general setup with few minor changes. We have created a loopback interface with 255.255.255.255 subnet to represent an entire subnet. This interface is pre-created on source and destination with the interface on destination in shutdown state. To simulate the migration we shut-down the interface on source followed by a no-shut-down on the destination (same sequence as live VM migration).

The location of source and destination can be changed to change the number of affected elements in the networks same way as experiment II.

Result:

We observed that expected RIB entries were seen on affected elements and no unexpected changes were made by the application. This confirmed the claim that state of the network can be migrated by changing the elements that NEED a state change to support the migration. We used the Add, Delete and Modify lists provided by the Algorithm 1 to select the appropriate operation on an element.

Chapter 4

Future Work and Conclusion

We achieved our goal of *finding the elements that need a state change to support migration* and verified it both mathematically and experimentally.

We proved mathematically that the Algorithm 1 in combination with its sub algorithms works successfully in a DC-Tree environment to find the affected elements due to migration. We also experimentally demonstrated the working of Algorithm 1 in a pure L3 domain.

We showed that Algorithm 1 works in liner time in most practical cases and also made recommendations as to how a designer can get best performance out of Algorithm 1 in various practical deployment use cases.

Along with the use of Algorithm 1 in finding the affected elements due to migrations we described its potential use in cases like load balancing traffic in equal cost network elements, traffic engineering etc.

We could achieve these without any restrictions on architecture as long as we have a DC-Tree arrangement, which is quite generic. We also do not require any special changes to the hardware and/or the software.

As good technology students we want to continue to improve our work [5]. As a step in this direction we have set the following path for the future —

- Scale and load testing on real hardware
- Test algorithm on various topologies and if required improve it to support them
- Implement in a mixed environment (onePK − openflow − cli combination)
- Validate in non TCP/IP environments

References

[1] Data center design: Ip network infrastructure. `http://www.cisco.com/en/US/docs/solutions/Enterprise/Data_Center/DC_3_0/DC-3_0_IPInfra.html`.

[2] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In *ACM SIGCOMM Computer Communication Review*, volume 38, pages 63–74. ACM, 2008.

[3] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitu Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, Murari Sridharan, C Faster, and D Maltz. Dctcp: Efficient packet transport for the commoditized data center. In *Proc. Sigcomm*, 2010.

[4] Angela Bartels. Data center evolution: 1960 to 2000. `http://www.rackspace.com/blog/datacenter-evolution-1960-to-2000/`,, August 2011.

[5] Ross Callon. The twelve networking truths. *Internet RFC 1925*.

[6] Martin Casado, Michael J Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. Ethane: Taking control of the enterprise. In *ACM SIGCOMM Computer Communication Review*, volume 37, pages 1–12. ACM, 2007.

[7] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pages 273–286. USENIX Association, 2005.

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

[9] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2001.

[10] William James Dally and Brian Patrick Towles. *Principles and practices of interconnection networks*. Morgan Kaufmann, 2004.

[11] Soudeh Ghorbani and Matthew Caesar. Walk the line: consistent network updates with bandwidth guarantees. In *Proceedings of the first workshop on Hot topics in software defined networks*, pages 67–72. ACM, 2012.

[12] Albert Greenberg, James R Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A Maltz, Parveen Patel, and Sudipta Sengupta. Vl2: a scalable and flexible data center network. In *ACM SIGCOMM Computer Communication Review*, volume 39, pages 51–62. ACM, 2009.

[13] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martín Casado, Nick McKeown, and Scott Shenker. Nox: towards an operating system for networks. *ACM SIGCOMM Computer Communication Review*, 38(3):105–110, 2008.

[14] 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. *ACM SIGCOMM Computer Communication Review*, 39(4):63–74, 2009.

[15] 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 *ACM SIGCOMM Computer Communication Review*, volume 38, pages 75–86. ACM, 2008.

[16] Cisco Systems Inc. Cisco's one platform kit (onepk). `http://www.cisco.com/en/US/prod/iosswrel/onepk.html`.

[17] Cisco Systems Inc. Nexus 1000v. `http://www.cisco.com/en/US/products/ps9902/index.html`.

[18] Oracle Inc. Java se 6 documentation. `http://docs.oracle.com/javase/6/docs`.

[19] Cisco Data Center Infrastructure. Cisco data center infrastructure 2.5 design guide, Nov. 2012.

[20] Changhoon Kim, Matthew Caesar, and Jennifer Rexford. Floodless in seattle: a scalable ethernet architecture for large enterprises. In *ACM SIGCOMM Computer Communication Review*, volume 38, pages 3–14. ACM, 2008.

[21] Charles E Leiserson. Fat-trees: universal networks for hardware-efficient supercomputing. *Computers, IEEE Transactions on*, 100(10):892–901, 1985.

[22] Yingwei Luo, Binbin Zhang, Xiaolin Wang, Zhenlin Wang, Yifeng Sun, and Haogang Chen. Live and incremental whole-system migration of virtual machines using block-bitmap. In *Cluster Computing, 2008 IEEE International Conference on*, pages 99–106. IEEE, 2008.

[23] 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 Computer Communication Review*, 38(2):69–74, 2008.

[24] Jayaram Mudigonda, Praveen Yalagandula, Mohammad Al-Fares, and Jeffrey C Mogul. Spain: Cots data-center ethernet for multipathing over arbitrary topologies. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, pages 18–18. USENIX Association, 2010.

[25] Michael Nelson, Beng-Hong Lim, Greg Hutchins, et al. Fast transparent migration for virtual machines. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 25–25, 2005.

[26] 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 *ACM SIGCOMM Computer Communication Review*, volume 39, pages 39–50. ACM, 2009.

[27] NOX Reop. Nox. http://www.noxrepo.org/.

[28] Cisco Systems. One platform kit: The power to innovate, 2012.

[29] Aditya Vaja. Live vm network migration. http://goo.gl/ps1W3, 2013.

[30] Haitao Wu, Guohan Lu, Dan Li, Chuanxiong Guo, and Yongguang Zhang. Mdcube: a high performance network structure for modular data center interconnection. In *Proceedings of the 5th international conference on Emerging networking experiments and technologies*, pages 25–36. ACM, 2009.

[31] K. Li Z. Zhou D. Zhang Y. Gu, C. Li. State migration - draft-gu-opsawg-policies-migration-02 expires september 2012.