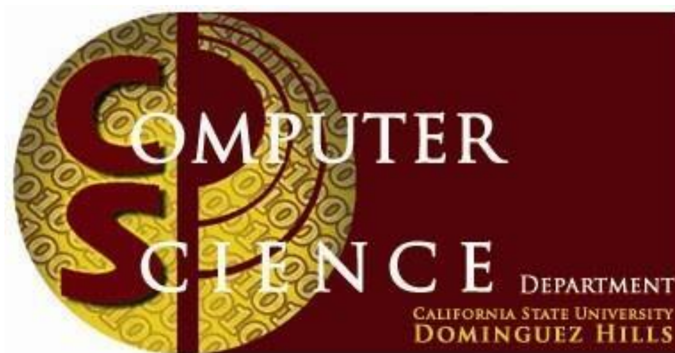
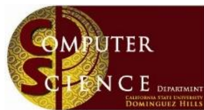


2017 Fall

Master's Project Report

Computer Science Department

California State University, Dominguez Hills.



csc@csudh.edu
http://csc.csudh.edu/

Computer Science Department
College of Natural and Behavioral Sciences
California State University, Dominguez Hills

NSM A-132
1000 East Victoria
Carson, CA 90747
Ph: (310) 243-3398
fax: (310) 243-3153

Master's Project

Select one:
CTC 492 __ ITC 492 __ CSC 492 __ CSC 590 √_ CSC599 __

Semester: Fall **Year:** 2017

Title: ***Energy-Efficient VNF Replication in Virtualized Data Centers***

Prepared by: _JANANI JANARDHANAN **Date**_ 11/23/2017

Dr.Bin Tang ______________ ______________
Faculty advisor ***Signature*** ***Date***

Dr. Mohsen Beheshti ______________ ______________
Committee member ***Signature*** ***Date***

Dr.Jianchao (Jack) Han ______________ ______________
Committee member ***Signature*** ***Date***

ENERGY-EFFICIENT VNF REPLICATION IN VIRTUALIZED DATA CENTERS

Project

Presented

to the Faculty of

California State University Dominguez Hills

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

in

Computer science

---

by

Janani Janardhanan

Fall 2017

ENERGY-EFFICIENT VNF REPLICATION IN VIRTUALIZED DATA CENTERS

AUTHOR: JANANI JANARDHANAN

APPROVED:

Bin Tang, PhD

Faculty Advisor

Mohsen Beheshti, PhD

Committee Member

Jianchao (Jack) Han, PhD

Committee Member

ACKNOWLEDGEMENTS

I would first like to thank my project advisor Dr. Bin Tang for sharing his ideas about such an interesting topic involving cutting-edge technologies like software defined networking, virtualization etc. and also for his very helpful feedback throughout all phases of the project. He was always ready to meet in person and clarify my doubts when needed and encouraged me to work harder to achieve better results.

I would like to gratefully and sincerely thank Dr. Mohsen Beheshti, Professor, Department Chair of Computer Science; I must express my very profound gratitude for his wonderful support and encouragement.

I would also like to thank my committee member, Dr. Jianchao (Jack) Han, professor of Computer Science, for all of his guidance and valuable advising through my studying years.

I would like to thank all the faculty of Department of Computer Science for their direct or indirect support and contribution for my academic growth and excellence.

Finally, and most importantly, I would like to thank my husband and my son for their unfailing support, understanding and patience during the past two years.

TABLE OF CONTENTS

PAGE

APPROVAL PAGE……………………………………………………………………………4

ACKNOWLEDGEMENTS…………………………………………………………………5

TABLE OF CONTENTS……………………………………………………………………6

LIST OF FIGURES…………………………………………………………………………8

ABSTRACT…………………………………………………………………………………9

CHAPTERS

1. INTRODUCTION ……………………………………………………………………11

 1.1 Benefits of Software Defined Networking……………………………………...11

 1.2 Network Function Virtualization and Virtual Network Function…………….....12

2. SPECIFICATION OF REQUIREMENTS ……………………………...………15

 2.1 Middlebox Replication Problem………………………………………………...15

3. BACKGROUND AND LIERATURE REVIEW…………………………………...17

4. NETWORK ARCHITECTURE………………………………………………………21

 4.1 NFV Architecture…………………………………………………………………21

 4.2 The Architecture of Fat-tree Topology……………………………………………22

5. DESIGN AND IMPLEMENTATION OF VIRTUAL NETWORK FUNCTION REPLICATION ALGORITHMS……………………………………………………......24

 5.1 The Proposed Algorithms…………………………………………………………24

  5.1.1 Random Replication Algorithm…………………………………………24

  5.1.2 Closest Next Middlebox First Algorithm………………………………...25

  5.1.3 Exhaustive Middlebox Replication Algorithm………………………...26

5.1.4 Traffic-Aware VNF Replication Algorithm…………………………...26

5.2 Network Design………………………………………………………………29

5.3 Implementation of Algorithms……………………………………………...32

5.3.1 Network Setup…………………………………………………………32

5.3.2 Analysis of Algorithms………………………………………………33

5.3.2.1 Random Replication Algorithm………………………………33

5.3.2.2 Exhaustive Middlebox Replication Algorithm………………34

5.3.2.3 Closest Next Middlebox First Algorithm ……………………35

5.3.2.4 Traffic-Aware VNF Replication Algorithm…………………...37

6 PERFORMANCE EVALUATION AND RESULT ANALYSIS………………………...39

6.1 Plots for k=4…………………………………………………………………...41

6.2 Plots for k=8……………………………………………………………………43

6.3 Individual Performance Analysis……………………………………………45

6.4 Comparative Performance Analysis…………………………………………47

7 FUTURE RESEARCH AND IMPLEMENTATION DIRECTIONS………………………49

8 CONCLUSION…………………………………………………………………………...50

REFERENCES……………………………………………………………………………51

APPENDIX: SOURCE CODE……………………………………………………………52

LIST OF FIGURES

PAGE

1. Network Function Virtualization 14

2. NFV Architecture....................21

3. Fat Tree Topology Architecture....................22

4. Sample Log for Performance Evaluation....................39

5. Plot for m=3, k=4, p= {100,200,300,400,500}....................41

6. Plot for m=5, k=4, p= {100,200,300,400,500}....................42

7. Plot for m=7, k=4, p= {100,200,300,400,500}....................42

8. Plot for m=3, k=8, p= {100,200,300,400,500}....................43

9. Plot for m=5, k=8, p= {100,200,300,400,500}.................... 44

10. Plot for m=7, k=8, p= {100,200,300,400,500}.................... 44

ABSTRACT

A Virtual Network Function (VNF) refers to the implementation of a network function, such as Firewall, Load Balancer, Network Address Translator (NAT), and Intrusion Detection System (IDS) using software decoupled from the underlying hardware. These network functions, also referred to as middleboxes, primarily ensure secure and cost-effective traffic flow in Virtualized Data Center Networks. Individual virtual network functions can be chained together as building blocks to offer a full-scale networking communication service. This is called service-chaining. The efficient placement of these VNFs directly impact network security and performance. Although some studies have been conducted on optimal placement of VNFs, very few of them considered the replication of VNFs in the network for minimizing cost flow, addressing load balancing and fault-tolerance issues. If multiple replicas of a service chain are placed in the network, the traffic flow can be redirected to and load balanced with different service chains for different traffic demands, thus decreasing over all traffic forwarding cost as well.

This paper examines energy-efficient VNF replication for service chains. It proposes three heuristic algorithms, Closest Next Middlebox First, Exhaustive MiddleBox Replication and Traffic-Aware VNF Replication. A Random Replication algorithm is also designed and implemented to prove the effectiveness and efficiency of other algorithms. These algorithms are designed exclusively for fat-tree topology, a widely used data center topology. A fat-tree is a k-ary tree with three tiers of k-port switches connected to physical machines hosting a number of Virtual Machine (VM) pairs.

The Random Replication (RR) algorithm randomly distributes the copies of VNFs in the network by placing them on hosts that satisfy the capacity constraints. The Closest Next Middlebox First (CNMF) algorithm works based on the physical proximity of the communicating VM pairs

and their corresponding subsequent VNFs in a service chain. The Exhaustive MiddleBox Replication (EMBR) algorithm is an extension to CNMF algorithm. EMBR not only considers the next closest VNFs in the path to destination but explores every single admissible path for better and accurate results. The Traffic-Aware VNF Replication (TAVR) primarily focuses on replicating VNFs by prioritizing their usage demands by various VM pairs in the network based on their communication frequencies. The VM pairs are grouped into different traffic frequency groups and replications are done in favor of every group.

While existing researches on VNF replication only focus on achieving load balancing, it is proved with extensive simulations that the proposed solutions also provide energy-efficient VNF replication in terms of reduced network cost. There is a very good scope of interesting and challenging future extensions too.

CHAPTER 1

INTRODUCTION

A Data Center is a facility that centralizes an organization's IT operations and equipment, as well as where it stores, manages, and disseminates its data. Data centers house a network's most critical systems and are vital to the continuity of daily operations. Data centers have evolved significantly in recent years, adopting technologies such as virtualization to optimize resource utilization and increase IT flexibility. As enterprise IT needs continue to evolve toward on-demand services, many organizations are moving toward cloud-based services and infrastructure. A focus has also been placed on initiatives to reduce the enormous energy consumption of data centers by incorporating more efficient technologies and practices in data center management. Data centers built to these standards have been coined "green data centers".

The goal of Software-Defined Networking (SDN) is to enable cloud and network engineers and administrators to respond quickly to changing business requirements via a centralized control console. SDN encompasses multiple kinds of network technologies designed to make the network more flexible and agile to support the virtualized server and storage infrastructure of the modern data center and Software-Defined Networking was originally defined an approach to designing, building, and managing networks that separates the network's control (brains) and forwarding (muscle) planes enabling the network control to become directly programmable and the underlying infrastructure to be abstracted for applications and network services.

1.1 The Benefits of Software Defined Networking

The ultimate benefit of SDN is the ability to dynamically provision the network to address the changing needs of businesses. It also provides the following benefits:

- **Directly Programable Network:** Network directly programmable because the control functions are decoupled from forwarding functions, which enable the network to be programmatically configured by proprietary or open source automation tools, including OpenStack, Puppet, and Chef.
- **Centralized Management:** Network intelligence is logically centralized in SDN controller software that maintains a global view of the network, which appears to applications and policy engines as a single, logical switch.
- **Reduced CapEx**: Capital Expenditure (CapEx) refers to the cost of developing or providing non-consumable parts for the product or system. SDN potentially limits the need to purchase purpose-built, ASIC-based networking hardware, and instead supports pay-as-you-grow models.
- **Reduced OpEX:** Operational Expenditure (OpEx) refers to the ongoing cost for running a product, business, or system. SDN enables algorithmic control of the network of network elements (such as hardware or software switches / routers) that are increasingly programmable, making it easier to design, deploy, manage, and scale networks. The ability to automate provisioning and orchestration optimizes service availability and reliability by reducing overall management time and the chance for human error.
- **Deliver Agility and Flexibility**: Software Defined Networking helps organizations rapidly deploy new applications, services, and infrastructure to readily meet changing business goals and objectives.
- **Enable Innovation:** SDN enables organizations to create new types of applications, services, and business models that can offer new revenue streams and more value from the network.

1.2 Network Function Virtualization and Virtual Network Function:

Network functions virtualization (NFV) offers an alternative way to design, deploy, and manage networking services. It is a complementary approach to software-defined networking (SDN) for network management. While they both manage networks, they rely on different methods. While SDN separates the control, and forwarding planes to offer a centralized view of the network, NFV primarily focuses on optimizing the network services themselves.

When service providers attempted to speed up the deployment of new network services to advance their revenue and growth plans, hey found that hardware-based appliances limited their ability to achieve these goals. They looked to standard IT virtualization technologies and found that NFV helped accelerate service innovation and provisioning.

A virtual network function (VNF) is a virtualized task formerly carried out by proprietary, dedicated hardware. VNF moves network functions out of dedicated hardware devices and into software. This allows specific functions that required hardware devices in the past to operate on standard x86 servers. VNFs carry out specific network functions on virtual machines(VMs) under control of a hypervisor. Such tasks might include firewalling, domain name service (DNS), caching or network address translation (NAT). An operator's network consists of a large number of intermediate Network Functions (NFs). Network Address Translators (NATs), load balancers, firewalls, and Intrusion Detection Systems (IDSs) are examples of such functions. Traditionally, these functions are implemented on physical MiddleBoxes, which are network appliances that perform functions other than standard path discovery or routing decisions for forwarding packets.

MiddleBoxes are based on special purpose hardware platforms that are expensive and difficult to maintain and upgrade. Following the trend of virtualization in large-scale networks,

network functions deployed as MiddleBoxes are also being replaced by Virtual Network Functions (VNFs). Typically, network flows go through several network functions. That means a set of NFs is specified and the flows traverse these NFs in a specific order so that the required functions are applied to the flows. This notion is known as network function chaining or network service chaining.

NFs can modify the traversing network flows in different ways. For example, a Deep Packet Inspector (DPI) can split the incoming flows over different branches according to the type of the inspected packets, each branch having a fraction of the data rate of the incoming flow. Firewalls can drop certain packets, resulting in flows with a lower data rate than incoming flows. A video optimizer can change the encoding of the video, which can result in a higher data rate. There can also be a dependency among a set of NFs that should be applied to the traffic in a network, which requires special attention to the order of traversing the functions in chaining scenarios.

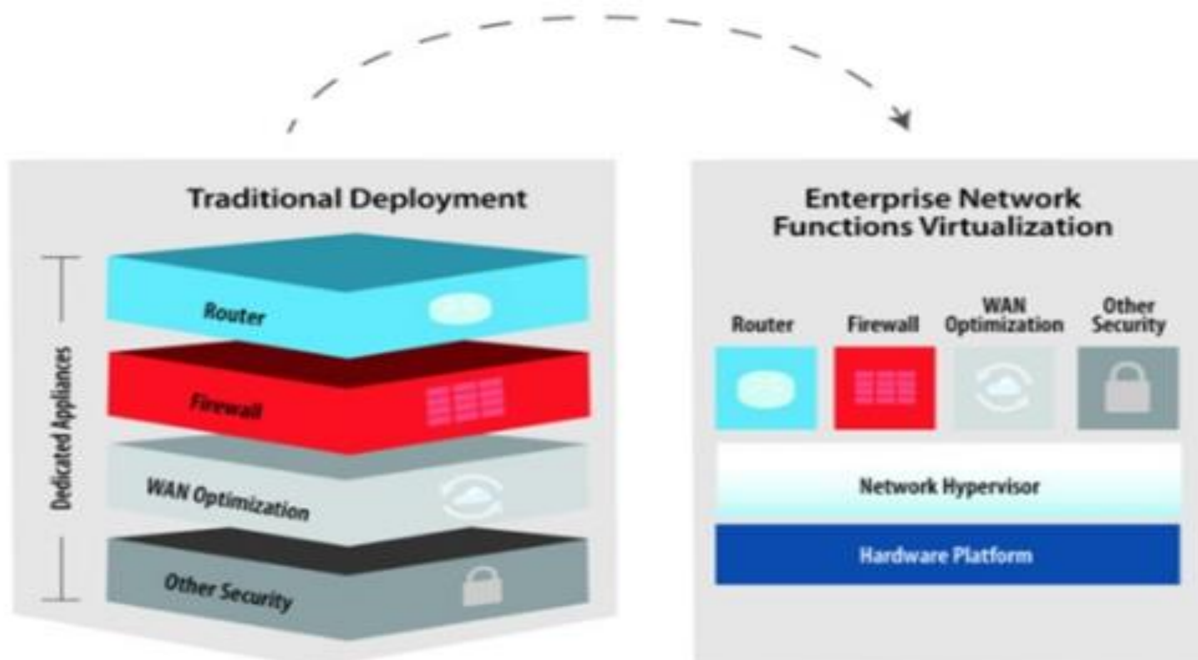


Fig. 1 Network Function Virtualization [10]

CHAPTER 2

SPECIFICATION OF REQUIREMENTS

Before the invent of Software-Defined Networking, network operators had to manually install and configure the middleboxes on hardware devices. It costed money as well as time and effort. While Software defined networking did reduce the tedium, network administrators found further ways to improve on effectively and efficiently operating a data center, like virtualizing network functions. NFV allows us to organize network functions, like building blocks to create communication services that can be deployed quickly and allow increased growth. SDN and NFV can run independently, but they are better when used together. NFV is executable even without an SDN yet these two can be consolidated into a single implementation and can gain more prominence. SDN paired with NFV can reduce costs for service providers.

When the data center has one service chain of a given type and sequence to cater to the needs of all traffic flow in the network, it inevitably ends up in congestion and starving. Therefore, placing replicas of the service chain in the network greatly helps to load balance as well as serve as backups, thus preventing the data center suffering from single point of failure.

Although lots of studies have been conducted on optimal placement of service chains in data centers, very few of them considered replication of service chains. The ultimate goal of this project was to design and implement efficient algorithms to create multiple copies of an ordered sequence of virtual network functions in the Data Center Network such that minimum cost flow is ensured along with providing dynamic provisioning, load balancing and high availability.

2.1 MIDDLEBOX REPLICATION PROBEM (MRP)

2.1.1 Middlebox Model

There are m middleboxes (of different types) $M = \{mb_1, mb_2, ..., mb_m\}$, where mb_i $(1 < j < m)$ is located at switch $SW_j \in V_s = \{SW_1, SW_2, ..., SW_{|Vs|}\}$. V_s is the set of switches holding the replicas of the middlebox instances distributed across the network. Each switch has a capacity, indicating number of middleboxes it can store. The capacity of switch SW_i is *cap(k).* The objective of MRP is to replicate middleboxes and place them onto switches such that the capacity constraint is satisfied and also when each communicating VM pairs traverse to one instance of $mb_1, mb_2, ...mb_m,$ each in that order, it results in minimum communication cost and energy consumption.

2.1.2 Problem Formulation of MRP

MRP consists of two stages. In the first stage, it decides how to replicate each middlebox and places its instances into different switches while satisfying the capacity constraints of switches. In the second stage, it decides for each VM pair, which instance of each middlebox mb_1, mb_2, $...mb_m$ to traverse in that specific order. Formally, the MRP is to select a set of switches $S_j = \{S_1, S_2, S_3, ...S_m\}$, where S_j is the set of switches each of which store an instance of *mbj*. Thus, there are different sets of switches for each middlebox type ranging from S_1 to S_j. Then for each VM pair (v_i, v_i'), find the sequence of switches $mb_{i,1} \in S_1 \cup \{SW(1)\}$, $mb_{i,2} \in S_2 \cup \{SW(2)\}$, etc. and finally, $mb_{i,m} \in S_m \cup \{SW(m)\}$ to traverse in that order to visit each middlebox instance, such that total communication cost is minimized. Let Ci be the communication cost for VM pair (v_i, v_i') with a middlebox replication scenario 'r'. Then,

$$C_i^r = c(S(v_i), mb_{i,1}) + \sum_{j=1}^{m-1} c(mb_{i,j}, mb_{i,j+1}) + c(mb_{i,m}, S(vi'))$$

If total energy consumption of all the *l* VM pairs with middlebox replication r is *Cr.* Then,

$$C^r = \sum_{i=1}^{l} C_i^r = c(S(v_i), mb_{i,1}) + \sum_{j=1}^{m-1} c(mb_{i,j}, mb_{i,j+1}) + c(mb_{i,m}, S(vi'))$$

The objective is to obtain the middlebox distribution under capacity constraints and with $C^r_{min.}$

CHAPTER 3

BACKGROUND AND LITERATURE REVIEW

NFV is about separating network functions from proprietary hardware and then consolidating, and running those functions as virtualized applications on a commodity server. NFV focuses on virtualizing network functions such as firewalls, WAN acceleration, message routers, message border controllers (used in VoiP networks), content delivery networks (CDNs) and other specialized network applications. Communication Service Providers spend huge amounts of money buying and maintaining specialized network hardware; thus, companies such as AT&T, Sprint, CenturyLink and other global CSPs have been receiving much of the attention from vendors who are working on NFV solutions [8].

There are many studies conducted on optimal placement of VNFs in the network. The authors of [5] propose a sampling approach using markov-chains called SAMA. They provide effective solution to reducing operational and network cost which they term as OPNET problem. They solve OPNET in a network by iteratively executing two steps i) finding the subset of nodes to deploy VNFs and ii) placing VNFs to minimize the total cost incurred in the system. This approach reduces the state space of feasible solutions that directly impacts to the convergence time. The, the controller chooses a configuration, which is owning the smaller cost. These phases are repeated until the underlying Markov chain converges to the stationary distribution. Every service chain c aims to find nodes that have enough available resources such that it sheds the smallest space left when being placed into those nodes. The idea of replicating VNFs with minimal network cost is improved from this research.

The problem of VNF placement with replications, and especially the potential of VNFs replications to help load balance the network is discussed in [4]. The authors design and compare

three optimization methods, including Linear Programing (LP) model, Genetic Algorithm (GA) and Random Fit Placement Algorithm (RFPA) for the allocation and replication of VNFs. The genetic algorithm is sub-divided into three interrelated genetic sub-algorithms: Traffic Engineering (TE-GA), Resource Allocation (RA-GA) and Resource Replication (RRGA) algorithms. TE-GA algorithm selects a set of admissible paths based on the input parameters and calculates the network cost. The output is used as the input for the RA-GA algorithm which is responsible for allocating the original VNFs. The placement is carried out respecting the sequence order for the chosen admissible path, based on which placement produces a lower network cost after of routing the data center traffic. The selected nodes will be used as the input for replication, where the algorithm will try to find alternative paths with the maximum number of allowed replicas. The alternatives paths are used in the RR-GA algorithm to allocate replicas based on the network cost, akin to RA-GA.

Starting with one replica set, the network cost is checked and compared with the case without replication. If the cost decreases, then, the algorithm tries to allocate a second replica checking if the cost improves the previous case with one replica only. This procedure continues until the increment of the number of replicas can no longer improve the cost. Their results show how the optimum VNF placement and replication in the network can significantly improve load balancing in comparison to simply building servers in the preferred nodes by the network operator. Although, this research work delves a little deeper into replication when compared to other existing ones, their main focus is only on achieving effective load balancing.

The same authors have discussed about optimizing link utilization and resource cost on [6]. They study the chaining and virtualization of the additional functions related to the dataplane on different physical locations (small data centers) but only in the mobile core network. The number of required replicas will be in relation with the network traffic demands. Therefore, by knowing

how many replicas are necessary, they place them to maintain a good network load balancing. In addition, the usage of additional network locations can increase the number of required servers and Data Centers (DC), which potentially will increase the network costs. Once the background traffic is load balanced, it will not be affected by the control of the data center traffic, but it has to be considered as a fixed input parameter for the next model called Resource Allocation (RA). This model is used to allocate optimally VNFs in the network trying to minimize the cost associated to the used resources, maximizing the network load balancing. The optimum placement of VNFs and replicas can provide the optimum locations for the data centers, which will be responsible for the instantiation of VNFs in the network.

Their research paper formulates the Link Capacity Dimensioning, Traffic Engineering (TE) model and The Resource Allocation (RA) models as optimization problems subject to a set of constraints. A. Link Capacity Dimensioning model allows the initial link dimensioning with the aim to minimize the required capacities for a given topology and a given traffic matrix. The TE model minimizes the utilization cost of all links in the network. In RA model, the number of admissible paths for each service chain s is constrained by the number of replicas. Therefore, with no replicas, a certain service chain can only use one path, while increasing number of replicas, the number of admissible paths proportionally increases. At the same time, the sequence order of VNFs in the service chain has to be maintained. They study optimization results obtained for the exclusive minimization of the load balancing and exclusive minimization of the network costs. Their solution is only unique with respect to the number of used DCs and not to the number of assigned VNFs per DC.

Unlike previous work [4] which only provides VNF replication for load balancing purposes, the solutions provided in this project also achieve minimal network cost. The best

optimal algorithms for VNF placement provided in [5] inspired to extend the concept for VNF replications in this project. Although [6] has very similar intention, their algorithms are designed exclusively for mobile core networks, and they consider replication across distributed data centers. In contrast, this project is designed for commercial fat-tree data center and replicas are placed within the same data center.

CHAPTER 4

NETWORK ARCHITECTURE

4.1 NFV Architecture

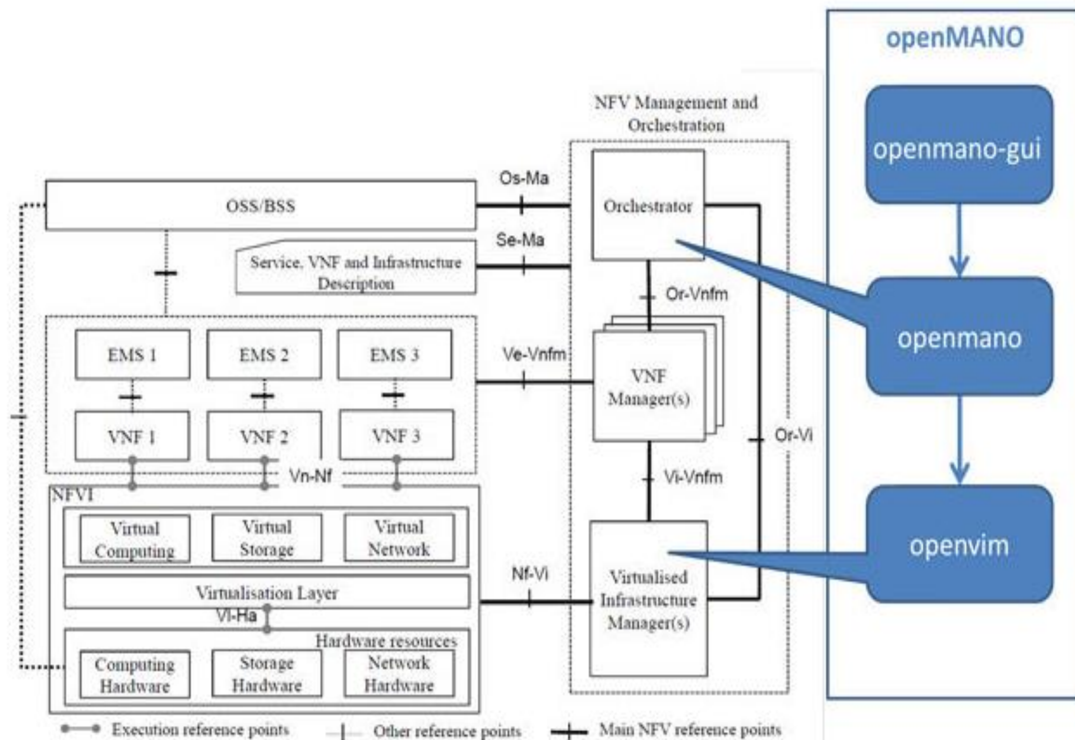


Fig. 2 – NFV Architecture [8]

The NFV architecture is basically described by three components: Services, NFV Infrastructure (NFVI) and NFV Management and Orchestration (NFV-MANO) [6]. A Service is the composition of VNFs that can be implemented in virtual machines running on operating systems or on the hardware directly. The hardware and software resources are provided by the NFVI that includes connectivity, computing, storage, etc. Finally, NFV-MANO is composed by the orchestrator, VNF managers and Virtualized Infrastructure Managers responsible for the management tasks applied to VNFs. In NFV-MANO, the orchestrator performs the resource allocation based on the conditions to perform the assignment of VNFs chains on the physical resources. The sub-task running in the orchestrator, known as VNF Forwarding Graph Embedding

(VNF-FGE) or VNF placement problem, tries to find the optimum place to allocate VNFs with regard to some specific objective, such as minimization of computation resources, minimization of power consumption, network load balancing, etc.

4.2 The Architecture of Fat-tree Topology

Many of the commercial data center networks adopt a special instance of clos topologies called Fat Tree. Any node can be reached from any other node by traversing a unique path through the common ancestor. Fat Tree topologies are popular for their nonblocking nature, providing many redundant paths between any 2 hosts. Such topologies are used in commercial data centers and to build fast and efficient super computers such as NSA's "Black Widow" [11] that watches millions of domestic and international phone calls and emails every single day.

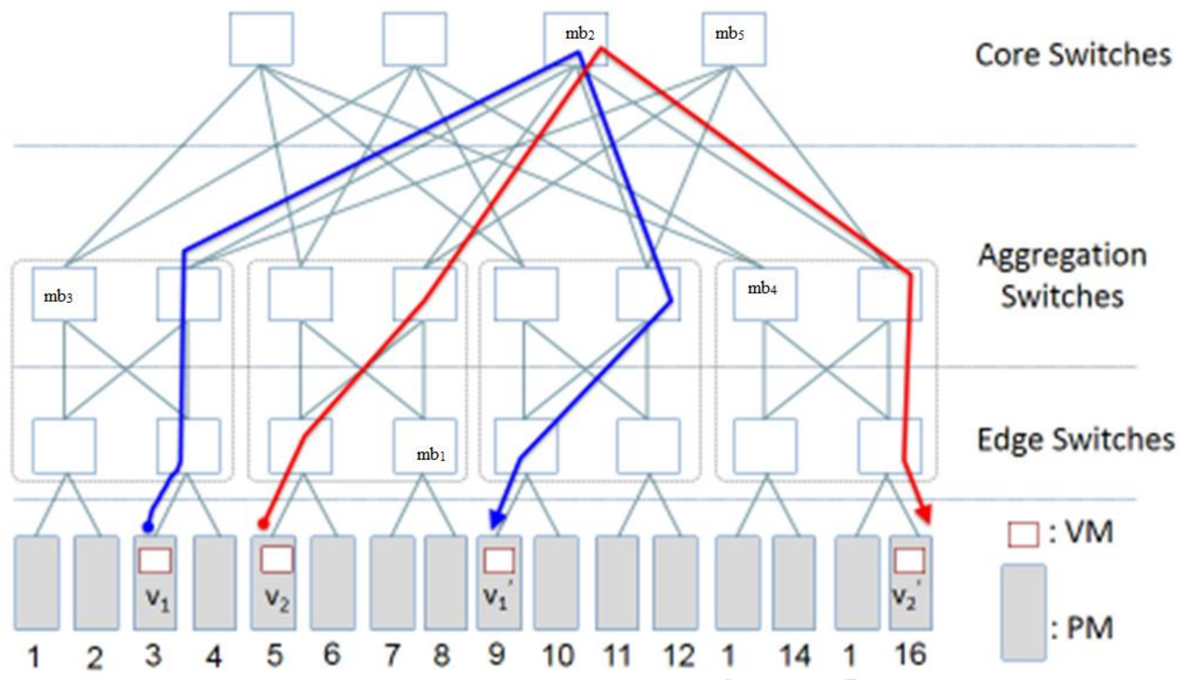


Fig. 3. Fat Tree Topology Architecture, A k-ary fat tree topology with k = 4

Google implemented a slight modification of Fat Tree topology to interconnect commodity Ethernet switches to produce scalable large data centers [11]. The topology consists of k-port

routers along with commodity compute nodes at the leaves of the tree. The basic building block of the data center is called a pod. A Fat Tree consists of k pods, each containing two layers of k/2 switches namely edge switches and aggregation switches. Each k-port switch in the lower layer (edge switch) is directly connected to k/2 hosts. Each of the remaining k/2 ports is connected to k/2 of the k ports in the aggregation layer of the hierarchy. There are $(k/2)^2$ K-port core switches. Each core switch has one port connected to each of the k pods. Thus, in total there are $5k^2/4$ switches in the network. Also, fat-tree topology supports connecting $k^3/4$ physical machines or hosts to the edge switches.

A k-ary fat-tree is shown in Fig.3. In a fat-tree ‘k’ is the number of ports of each switch and in this sample k=4; thus, there are 20 switches across all three layers and 16 physical machines. There is also an original sequence of a service chain ‘c’ with 5 MiddleBox types mb_1, mb_2, mb_3, mb_4 and mb_5.

CHAPTER 5

DESIGN AND IMPLEMENTATION OF VIRTUAL NETWORK FUNCTION REPLICATION ALGORITHMS

Replicating virtual network functions that should serve traffic is not only interesting but challenging as well. The two important goals of a network administrator are reducing power consumption (the number of active nodes in the network) and reducing traffic forwarding cost. These two goals are ideal but contradictory. To reduce power consumption, it is important to implement algorithms like server consolidation and turn off as many inactive nodes as possible where as to decrease traffic forwarding cost, it is important to have many admissible paths from node to another so that traffic can flow on the best path. Having many nodes also make the network fault-tolerant and load balanced. Keeping these trade-offs in mind, the following algorithms were proposed and implemented during the project development phase. Their performances are thoroughly evaluated with extensive simulations, and the results are vividly presented in this report which are visually appealing as well. The constraints, pre-requisites, best and worst-case analysis and other issues with each of them are reported too. Following are the proposed algorithms for service chains.

5.1 THE PROPOSED ALGORITHMS:

5.1.1 **Random Replication Algorithm**

With this algorithm, the placement of VNF replicas is carried out as random-fit, whereby all valid solutions according to the constraints are considered and one of these solutions is randomly chosen. To find a valid path from one virtual machine to another in a communicating VM pair, after the random placement of VNFs, the algorithm searches for the admissible paths

that traverse the VNFs in the correct order and chooses one path randomly for every VM pair. After the required number of maximum replication is achieved in the network, the algorithm stops by yielding the output consisting of chosen nodes for middlebox replication and the average network cost when traffic flows via all virtual machine pairs.

5.1.2 Closest Next Middlebox First Algorithm

The Closest Next Middlebox First algorithm is designed exclusively for service chains. It replicates middlebox instances one by one by placing a middlebox instance (mb_x) in a node closest to one of the copies of mb_{x-1} instances. The node that hosts an mb_x is chosen to yield the lowest overall traffic-flow cost on the network. The number of maximum replications that could be placed on the network depends on the number of switches in the network and total number of required network functions or middlebox types. VNF replication using this method successfully places at least one copy of a middlebox type on every node on the network. When all the nodes in the network have a copy of a VNF instance, the replication is done. Then, each VM pair is assigned to its closest service chain for relaying traffic.

In a fat-tree network, the maximum number of replicas of a service chain depends directly on the number of switches and inversely on number of middlebox types. That is,

Number of maximum replications (R_{max}) = Floor (Number of Switches/MB Types).

Thus, the number of replications for every middlebox type is the same here. With the given original sequence of the service chain, a host node for the replica of each middlebox type is searched starting from the first edge switch to the last core switch every time. The important criterion to choose a switch as the host is to see if it gives the least cost for the VNF that is considered. The special step that is followed in this algorithm is that every path within the service chain has to be a shortest path too, that is one yielding the least cost. For instance, suppose that

there are 5 VNFs in the service chain and if the VNF to be replicated is VNF-3, when traffic flow from the VM pair (v, v') is considered, VNF-1 is chosen as the one closest to sender 'v' and the VNF-2 closest to the chosen VNF-1 is the next in the service chain, etc. By following the procedure of always choosing the closest next middlebox first for traffic engineering, R_{max} replicas are placed on the network on the switches that yield the minimum overall traffic flow cost within the network for all VM pairs.

5.1.3 Exhaustive MiddleBox Replication Algorithm

In EMBR algorithm, not only the next closest VNF but also all possible combinations of VNF replicas are iteratively considered to achieve further optimization in over all traffic cost. In a fat-tree network, the maximum number of replicas of a service chain depends directly on the number of switches and inversely on number of middlebox types. That is, the R_{max} value is computed in a similar way as CNMF algorithm. Thus, the number of replications for every middlebox type is the same in EMBR too.

With the given original sequence of the service chain, a host node for the replica of each middlebox type is searched starting from the first edge switch to the last core switch every time. The important criterion to choose a switch as the host is to see if it gives the least cost for the VNF that is considered. Unlike CNMF algorithm, in this algorithm all combinations of all admissible path from sender to receiver is thoroughly analyzed before fixating a switch to be the host for a given middlebox type. The only drawback is the long convergence time for larger network.

5.1.4 Traffic-Aware VNF Replication Algorithm

Traffic-Aware VNF Replication algorithm classifies the VM pairs in the network into groups based on their communication or traffic-flow frequency. The total number of replications in the entire network depends on the number of switches and middlebox types. However, the

number of replications allocated in favor of a traffic group is determined by the probability distribution of that traffic group and by the frequency of communication between each VM pair in the traffic group. This replication is done in the order of priority of the traffic group; most frequently communicating VM Pairs are given the highest priority. The primary advantage of this algorithm is the efficient replication of VNFs based on expected traffic flow.

In the fat-tree network, the VM pairs after being placed on their respective host servers are associated to a traffic class group based on their rate or frequency of communication. This algorithm categorizes the VM pairs under 4 groups namely 'Very Frequent Communicators', 'Frequent Communicators', 'Medium Communicators' and 'Rare communicators'. Each traffic group has its own distribution count as well. For example, one of the frequency distribution is [40%,45%,12%,3%]. This means that out of 100% of the VM pairs present in the network, 40% are very frequently exchanging data traffic, 45% are frequently communicating but not highly frequent as the group 1, 12% VM pairs are communicating at a lower rate and 3% are rarely communicating. R_{max} still remain the same as other algorithms but the replicas are distributed across the traffic frequency groups based on their probability distribution in the data center.

For example, the following computation makes it easier to comprehend the idea behind traffic-aware replication algorithm,

Total num of possible replications = R_{max} = floor (#switches/#mb_types)

Replications in favor of Very_frequent group = G1 = floor (number of VM pairs on very frequent group /Total number of vmpairs) * R_{max}

Replications in favor of Frequent group = G2 = floor (number of VM pairs in Frequent group/Total number of vmpairs) * R_{max} etc..

The algorithm begins replicating service chains in favor of VM pairs belonging only to Very Frequent group. After G1 number of replicas out of R_{max} are replicated, it moves to replicating G2 replicas for frequent group in the remaining available switches with the required capacity constraints. Thus, every group has dedicated service chains for them to load balance. They can use the other group's service chain if the chosen service chain provides better cost for the VM pair's communication and if that service chain is not used by a VM pair belonging to a superior group.

5.1.5 Non-Service-Chain Scenario

There will seldom be the need for traversing the network functions in any random order. Although the scope of this project is VNF replication of service chains, the following algorithm was designed and implemented for non-service chain scenario as an extension.

5.1.5.1 Non-Sequential Middlebox Replication Algorithm

In the previous replication algorithms, the replication of a middlebox for a service chain scenario was in favor of that middlebox itsef. That is, of all the available switches that satisfy the capacity constraint of the middlebox type, the switch that gave the least overall traffic cost was chosen as the final host. For the non-sequential middlebox replication, the replication is done in favor of the switches. There by, a switch chooses to host a middlebox type if that middlebox type gives the least cost for that switch. In this way, every switch in the network is placed with a middlebox type that gives the least cost among all other middlebox types.

There are few other constraints to be considered. For instance, replication should happen judiciously. Every middlebox type should have fair share or equal number of replicas in the network. The algorithm ensures not to place too many copies of the same middlebox type within the same pod. The middlebox types residing on the core switches also should be as unique as

possible. Once the replica copies are placed onto the switches, a shortest path algorithm like Breadth First Search can be used to traverse through the required middleboxes for the traffic flow.

5.2 NETWORK DESIGN

To deploy and test the algorithms, it is important to simulate a fat-tree network. The Fat-tree network is created based on the user input of the number of ports ‘k’. The communicating virtual machine pairs are randomly distributed in the network across different physical machines by ensuring their capacity constraints. Then, one original copy of the service chain is placed on the network.

Object-Oriented Design(OOD) is the best way to develop such a complex project. Code reuse was extremely important because all the algorithms discussed in the proposal were required to do redundant functions like create fat-tree, distribute virtual machine pairs, calculate traffic cost etc. Also, with the help of encapsulation, unnecessary implementation details are hidden from the user. Additionally, an object of the fat-tree network can be used to control how these algorithms or users interact with the fat-tree itself, thus preventing errors. Keeping these and other obvious advantages of OOD, the project was designed as described in the next few paragraphs.

The key entity of a network are its devices. ‘Device’ is the common class for creating a switch or server in the network. These devices have various properties associated with them like their configurations or capacity constraints, if the device is a switch or server and if it is a server, then the list of virtual machines it holds or if it is a switch, then the list of middleboxes it holds etc. The code snippet below gives the class’ details.

```
class Devices{

        int DeviceID;

        int capacity;
```

```
        boolean isServer;

        int podID;

//      boolean isVirtual;

        ArrayList<Integer> VM;

        ArrayList<Integer> MB;

        ArrayList<Integer> mb_preference_list;

        ArrayList<Integer> neighbors;

        final static int Server_Capacity = 10; //# of VMs a server holds

        final static int Switch_Capacity = 1;  //# of MBs a switch holds

        Devices(int id, int capacity, boolean isServer){

                this.DeviceID = id;

                this.capacity = capacity;

                this.isServer = isServer;

                this.neighbors = new ArrayList<Integer>();

                if(this.isServer){

                        VM = new ArrayList<Integer>();

                        mb_preference_list = new ArrayList<Integer>();

                        MB = null;

                }

                else{

                        VM = null;

                        mb_preference_list=new ArrayList<Integer>();

                        MB = new ArrayList<Integer>();
```

```
                }
        }
        }
```

Fat-tree itself is another entity. Fat-tree consists of devices and links between them. Hence the relationship between FatTree and Device class is aggregation. Thus, the FatTree class contains details of the number of switches, servers and links in the network. Following is the code snippet of a part of FatTree class.

```
public class FatTree {
        int Num_Ports;
        int Num_Servers;
        int Num_EdgeSw; // # of edge/access switches
        int Num_AggSw; // # of aggregation switches
        int Num_CoreSw; // # of core switches
        int Num_AllSwitches;
        int Num_AllDevices;
        Devices[] devices; // objects of Devices class to hold details of every device -
switch/server
        Integer[][] cost; // cost/weight from a node/device i.e., the link cost
        FatTree(){
                Num_Ports = 0;
                Num_Servers = 0;
                Num_EdgeSw = 0;
```

```
            Num_AggSw = 0;

            Num_CoreSw = 0;

    }

//other code

}
```

Then several methods were written to perform different operations on the network. The possible operations on the network are:

1. Randomly distribute the virtual machines across the servers
2. Randomly pair up different virtual machines
3. Place one original copy of middlebox instances on the network
4. Calculate the cost of every node from every other node in the network.
5. Calculate traffic flow cost when traffic flows between one VM and another in a VM pair

Kindly refer Appendix to view the source code for all of them.

Every algorithm discussed in the proposed framework is implemented as a separate class which aggregates the 'Fattree' class to run on the fat-tree topology.

5.3 IMPLEMENTATION OF ALGORITHMS

5.3.1 NETWORK SETUP:

- R_{max} is set to $5k^2/4m$. This is because the total number of switches in a fat-tree network is '$5k^2/4$' and the total number of middlebox types is 'm'. So, the maximum number of copies of all 'm' middlebox types that can be placed on the network is given by number of switches over number of middlebox types.

- In all the conducted experiments, the number of virtual machines a server or physical machine could hold, the “Server_Capacity’, was set to 100 (since number of VM pairs were tested from 100 to 500 even in k=4 scenario).
- The number of middlebox instances a switch(Switch_Capacity) could hold is set to 1.
- Link cost of immediate neighbors is set to 1.

5.3.2 ANALYSIS OF ALGORITHMS:

5.3.2.1 Random Replication Algorithm

Input:

K – Number of ports

F – An object of FatTree network

M – Number of middlebox types

C – The original sequence of the service chain

P – The VM pairs placed on the physical machines of the network

Algorithm:

1. For every middlebox type in the service chain {mb_1, mb_2 ,…mb_m}
2. If the current middlebox type mb_x’s replica count has not reached R_{max}
3. Randomly choose a switch as host for mb_x
4. If the chosen switch’s capacity satisfies the capacity constraints of mbx, place the replica copy of mb_x on that switch.
5. Else, go to Step 3
6. If all middlebox types have R_{max} replicas, stop the algorithm.

Explanation:

By using the algorithm above, every middlebox type is ensured to have R_{max} replicas in the network provided all switches satisfy the capacity constraints. The host is randomly chosen by only considering the capacity of the host. Once, random replica copies of all middlebox types are thus placed across the network, every VM pair can choose a random service chain to send traffic from source to destination. Though random procedures can work well at times, they are not always reliable. Random Replication algorithm can only be used in scenarios where VM pairs communicate very rarely and energy conservation is not significant.

Time-Complexity:

O (R_{max} * M*5K^2/4) => O (K^4). This is the worst-case execution time for the Random Replication algorithm. In the best case, where every switch it randomly chooses for the first time is the correct host for a middlebox type mb_m, the time complexity is O (K^2).

5.3.2.2 Exhaustive Middlebox Replication Algorithm:

Input:

K – Number of ports

F – An object of FatTree network

M – Number of middlebox types

C – The original sequence of the service chain

P – The VM pairs placed on the physical machines of the network

Algorithm:

1) For placing every replica copy ‘R’ from {1, 2, ….R_{max}}

2) For every middlebox type ‘M’ in the service chain {mb_1, mb_2 ,…mb_m}

3) For every switch ‘S’ as host in the fat-tree network

4) If the chosen switch's capacity 'cap' satisfies the capacity constraints of mb_x

5) For All 'R' middlebox replica copies of $\{mb_1, mb_2 \ldots mb_{x-1}\}$

6) For All 'R-1' middlebox replica copies of $\{mb_{x+1}, mb_{x+2}, \ldots mb_m\}$

7) For All 'P' VM pairs in the network

8) If the switch 'S' yields the minimum cost for that middlebox type 'M', place 'M' on 'S' and decrease its available capacity.

Explanation:

In this algorithm, we exhaust all possible combinations of middlebox instances so as to achieve the ideal or perfect result. The only drawback of this algorithm is its convergence time. However, it is commonly known that network orchestration for Quality of Service (QoS) services is time consuming during the initial set up, but once it is set up and is running, the service remains unaffected until disabled deliberately by the network administrator.

Time Complexity:

$O(R_{max} * M*5K^2/4 *R_{max}*R_{max} *P) \Rightarrow O(PK^8/M^2)$. This is the execution time for the algorithm.

5.3.2.3 Closest Next Middlebox First Algorithm

Input:

K – Number of ports

F – An object of FatTree network

M – Number of middlebox types

C – The original sequence of the service chain

P – The VM pairs placed on the physical machines of the network

Algorithm:

1) Initialize a property called next closest middlebox to every VM pair as original mb_1.
2) Initialize next closest middlebox to every middlebox up to $_{mbm-1}$ in the original sequence. That is for mb_1, set the closest next middlebox as mb_2, for mb_2 the closest next middlebox is mb_3 etc.
3) For placing every replica copy 'R' from {1, 2 ,….R_{max}}
4) For every middlebox type 'M' in the service chain {mb_1, mb_2 ,…mb_m}
5) For every switch 'S' as host in the fat-tree network
6) If the chosen switch's capacity 'cap' satisfies the capacity constraints of mb_x
7) For All 'P' VM pairs in the network
8) Choose closest next middlebox of every device up to mb_x.
9) From all available mb_{x+1}, choose closest mb_{x+1} to current mb_x.
10) Choose closest next middlebox from chosen mb_{x+1} to mb_m
11) Send traffic via all 'P's using the service chain obtained from step 8-10
12) If the switch 'S' yields the minimum overall cost for that middlebox type 'M', place 'M' on 'S' and decrease its available capacity.
13) If mb_x is mb_1

    For all 'P', check if current mb_x can be set as closest next mb_1.
14) Else

    For all 'R' replicas of mb_{x-1}, check and set if mb_x is the closest next

Time-Complexity:

O (R_{max} * M*5K^2/4 *(2P+R_{max})) => O (PK^4+K^6) which is approximately O(K^6). This is the execution time for the algorithm. Although the execution time is better than Exhaustive

Replication algorithm, reduction in traffic cost is greatly achieved by the former since all possible combinations are checked before deciding a host. Shortest path may not be the best solution in all cases. This algorithm can be tremendously useful when quick set up is required.

5.3.2.4 Traffic-Aware VNF Replication Algorithm:

Input:

K – Number of ports

F – An object of FatTree network

M – Number of middlebox types

C – The original sequence of the service chain

P – The VM pairs placed on the physical machines of the network

Algorithm:

1. For all ‘P’ VM pairs associate them to their respective traffic frequency group in {0,1,2,3} based on their frequency of communication per time unit.
2. Calculate the probability distribution for each traffic group as follows

   Probability distribution of a group G = (Number of VM pairs in G/ P)

   where P is the total number of VM pairs available in the network.
3. For every group G, calculate the number of replications that can be allocated to that group by using the following formula

   Number of replicas(R_g) for a group G = Probability distribution of G * R_{max}

   Thus, for G={0,1,2,3}, $R_0 + R_1 + R_2 + R_3 = R_{max}$
4. For every group G
5. For every possible replica ‘R’ within the group from {1,2…R_g}

6. For every middlebox type 'M' in service chain from {mb_1, mb_2 …. mb_m}
7. For every switch 'S' as host in the fat-tree network
8. If the chosen switch's capacity 'cap' satisfies the capacity constraints of mb_x
9. If mb_x is mb_1, create a temporary service chain from original service chain with mb_1 being mb_x
10. Else, create a service chain from {mb_1,mb_2…mb_{x-1}} from the current replication 'R', retain mb_x and choose {mb_{x+1},…mb_m} from original service chain.
11. For all P_g VMpairs belonging to that group G
12. If current mb_x yields the minimum overall traffic cost which is expected to be lesser than or equal to the original cost yielded by the service chain before replication, place 'M' on 'S'.

Time-Complexity:

$O(G * R_{max}*M*5K^2/4 *P) \Rightarrow O(G * (5K^2/4M)*M*(5K^2/4) *P) \Rightarrow O(GPK^4)$. This algorithm performs better than all proposed algorithms. Once the replicas are set up, a service chain preference list can be created for all VM pairs to choose a best service chain for each VM pair. To do that the execution time would be O(PRmax). Instead, it could also be set in Step 11-12 by checking if the current traffic cost is the minimum traffic cost yielded so far for the pair 'p'.

CHAPTER 6

PERFORMANCE EVALUATION AND RESULT ANALYSIS

In order to evaluate the performances of the algorithms discussed above, extensive simulations were performed and results were logged in Microsoft Excel. Every algorithm outputs the hosts in which the middleboxes are replicated and also the average overall traffic cost in the network with the current configuration. Each algorithm was executed ten times to arrive at an average overall traffic cost in the network by modifying different parameters in the network. An example snapshot of the result log is pasted below:

Case 1 : k=4, p= 100	Exp1	Exp2	Exp3	Exp4	Exp5	exp6	exp7	exp8	exp9	exp10	Average Traffic Cost
1. m=3	1000	850	1000	950	1000	1080	890	870	910	970	952
2. m=5	1200	1300	1300	1300	1300	1390	1200	1240	1300	1370	1290
3. m=7	1400	2000	1500	2100	1800	1675	1700	1460	1580	1620	1683.5
Case 2: k=4, p=200											
1. m=3	2000	1700	1900	2000	1700	2000	1800	1870	1820	1770	1856
2. m=5	2800	2400	2600	2400	2800	2615	2500	3070	2900	2645	2673
3.m=7	3400	3000	3400	3000	3400	3130	3200	4100	3340	4000	3397
Case 3: K=4, p=300											
1. m=3	3100	2550	2850	2550	3100	3250	3300	3050	3000	2980	2973
2. m=5	3750	3900	3900	3750	3900	3670	4000	3900	3800	4230	3880
3. m=7	6000	4800	4800	5100	4500	4980	5000	5500	4900	5000	5058
Case 4: k=4, p=400											
1. m=3	4000	3400	3600	3800	4000	3960	3700	3600	3760	4100	3792
2. m=5	4600	5000	5000	5400	5200	5355	5300	5510	5500	5480	5234.5
3. m=7	7200	6000	6800	6400	6000	6980	6600	7000	7000	6000	6598
Case 5:k=4, p=500											
1.m=3	4250	4250	4250	4000	4000	5420	5000	5200	4290	5100	4576
2.m=5	5500	6750	6500	6500	6750	7550	7000	6620	6890	7000	6706
3.m=7	8500	7500	8000	6500	6000	8140	7800	9000	6780	7560	7578
Case 1 : k=8, p= 100											
1. m=3	1050	950	1000	1050	950	1080	1080	1000	970	990	1012
2. m=5	1300	1300	1200	1300	1200	1390	1280	1400	1350	1400	1312
3. m=7	1600	1600	1900	1600	1900	1675	1660	1600	1760	1900	1719.5

Fig.4. Sample Log for Performance Evaluation

The parameters that are configured in the network during these simulations are as follows:

- ❑ *K = 4 and 8*
- ❑ *P = 100, 200, 300, 400, and 500*
- ❑ *m = 3, 5 and 7.*

‘K’ is the number of switch ports in a k-ary fat-tree. ‘P’ is the number of VM pairs residing on the physical machines connected to the edge switches. ‘m’ is the number of middlebox types in the service chain. The average traffic cost for each case are plotted as graphs (column charts).

Even though ten runs were used to compute the average, it is always good to account for the variability of data to indicate the error or uncertainty in a reported measurement. Standard Deviation is used to indicate the extent of deviation for a group as a whole. Excel’s inbuilt function STEDEV.S was used to compute the standard deviation in the trials. CONFIDENCE is Excel’s inbuilt function to compute the Confidence Interval. It returns a value that we can use to construct a confidence interval for a population mean.

The confidence interval is a range of values. If the average traffic cost x, is at the center of this range and the range is x ± CONFIDENCE. For any population mean, μ0, in this range, the probability of obtaining a sample mean further from μ0 than x is greater than alpha; for any population mean, μ0, not in this range, the probability of obtaining a sample mean further from μ0 than x is less than alpha.

Syntax:

CONFIDENCE (alpha, standard_dev, size)

- Alpha is the significance level used to compute the confidence level. The confidence level equals 100*(1 – alpha) %, or in other words, an alpha of 0.05 indicates a 95 percent confidence level.
- Standard_dev is the population standard deviation for the data range and is assumed to be known.
- Size is the sample size.

Error Bars were included in the plots to give a general idea of how precise a measurement is, or conversely, how far from the reported value the true (error free) value might be. The error bars were customized to use the Confidence Interval values as error metric.

6.1 Plots for k=4

The average traffic cost is plotted for 4-ary fat-tree which has the following set up

Number of switches: $5k^2/4 = 5*16/4 = 20$

Number of servers: $k^3/4 = 4*4*4/4 = 16$

With the above setup by varying parameters like number of middlebox types 'm' in the service chain and number of communicating VM pairs 'p', the following results were obtained.

The VMs were randomly placed on different servers based on capacity constraints and randomly paired up among themselves. One original sequence of service chain is placed in the network already.

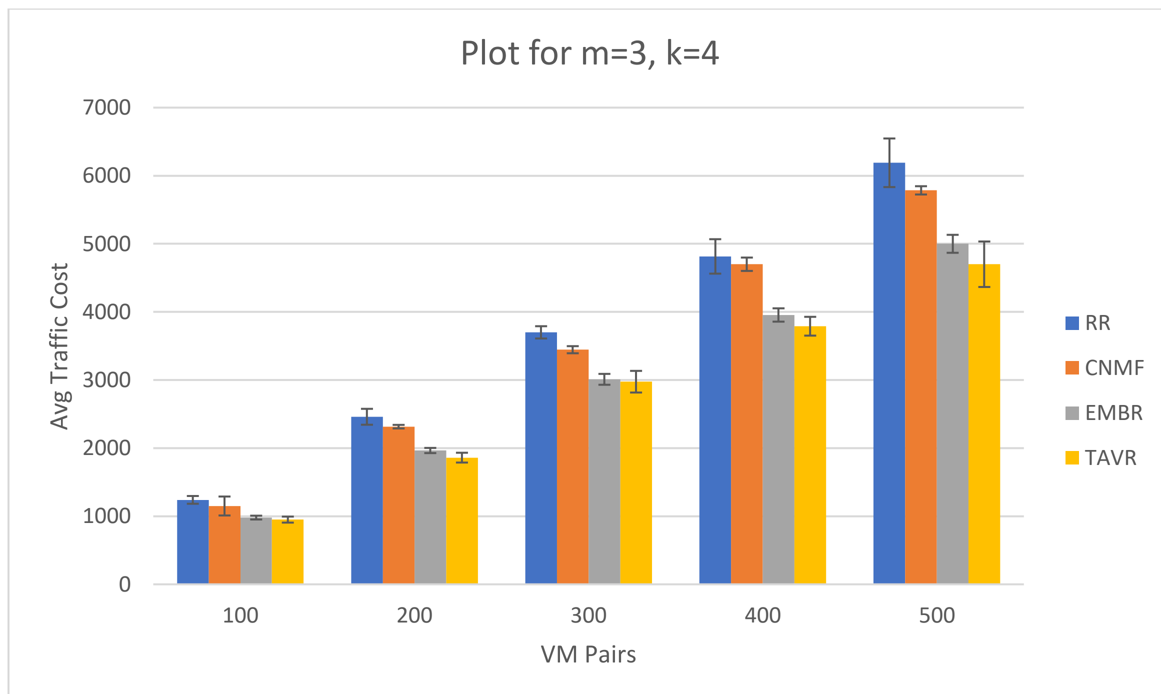


Fig. 5 Plot for m=3, k=4, p={100,200,300,400,500}

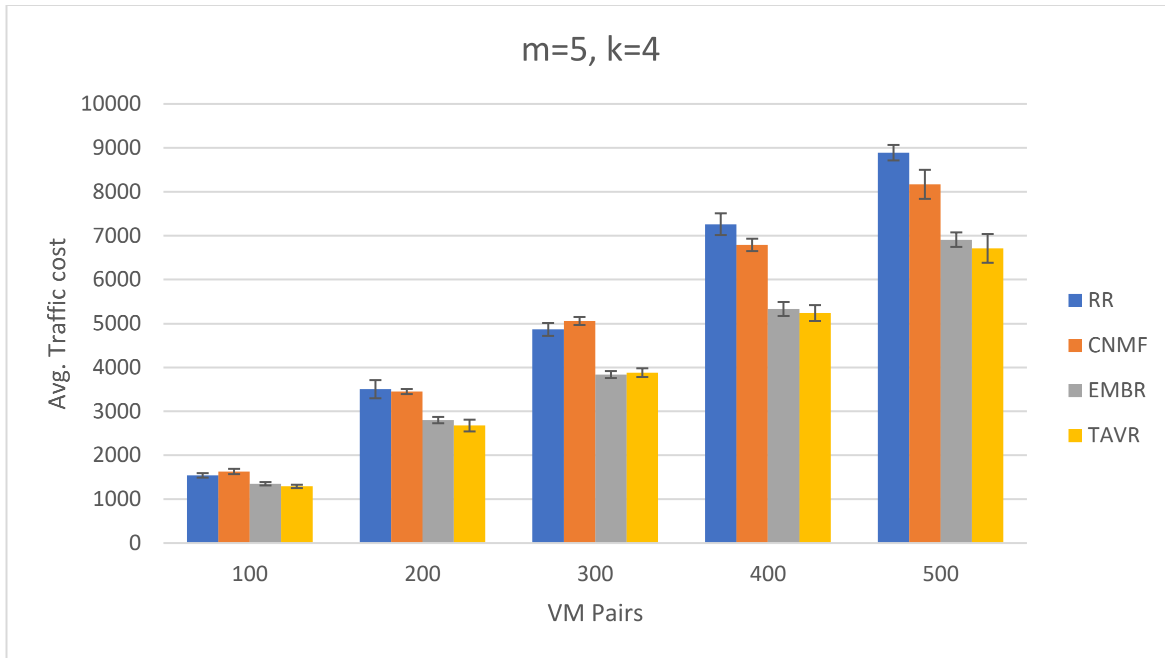


Fig. 6 Plot for m=5, k=4, p={100,200,300,400,500}

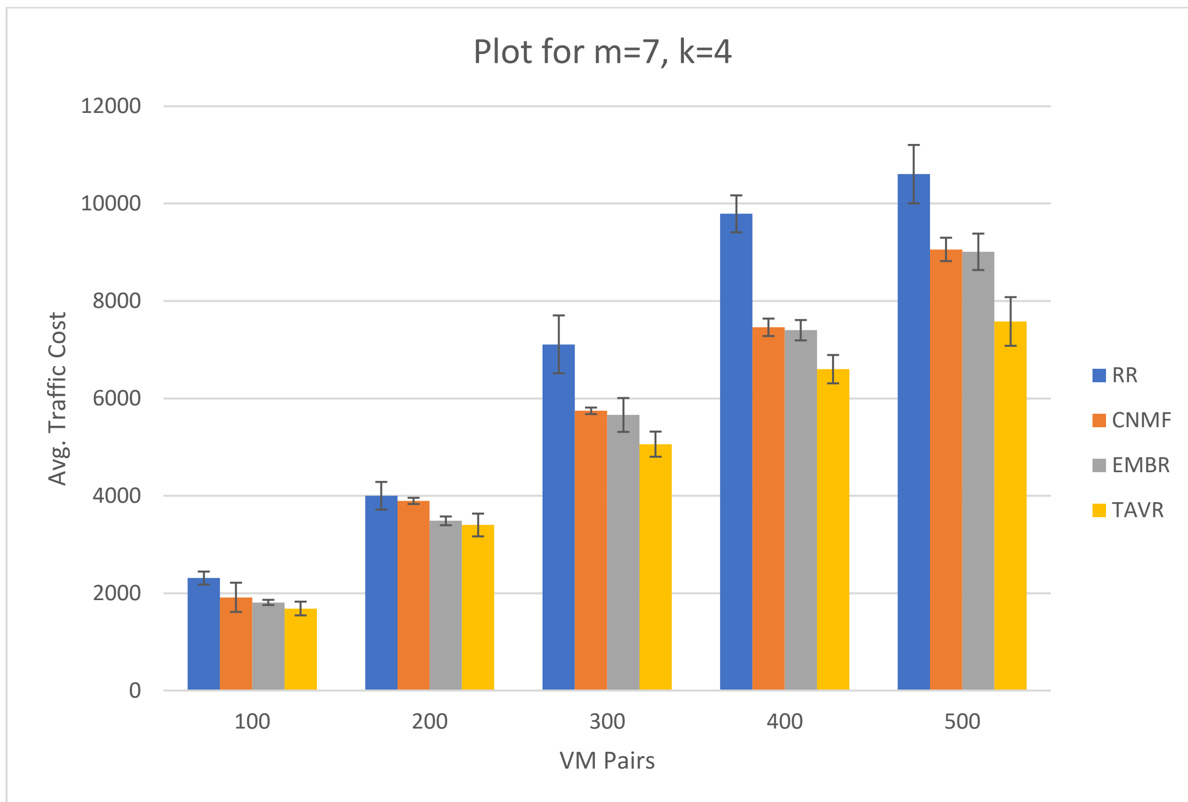


Fig. 7 Plot for m=7, k=4, p={100,200,300,400,500}

6.2 Plots for K=8

The average traffic cost is plotted for 8-ary fat-tree which has the following set up

Number of switches: $5k^2/4 = 5*64/4 = 80$

Number of servers: $k^3/4 = 8*8*8/4 = 128$

With the above setup by varying parameters like number of middlebox types 'm' in the service chain and number of communicating VM pairs 'p', the following results were obtained. The VMs were randomly placed on different servers based on capacity constraints and randomly paired up among themselves. One original sequence of service chain is placed in the network already.

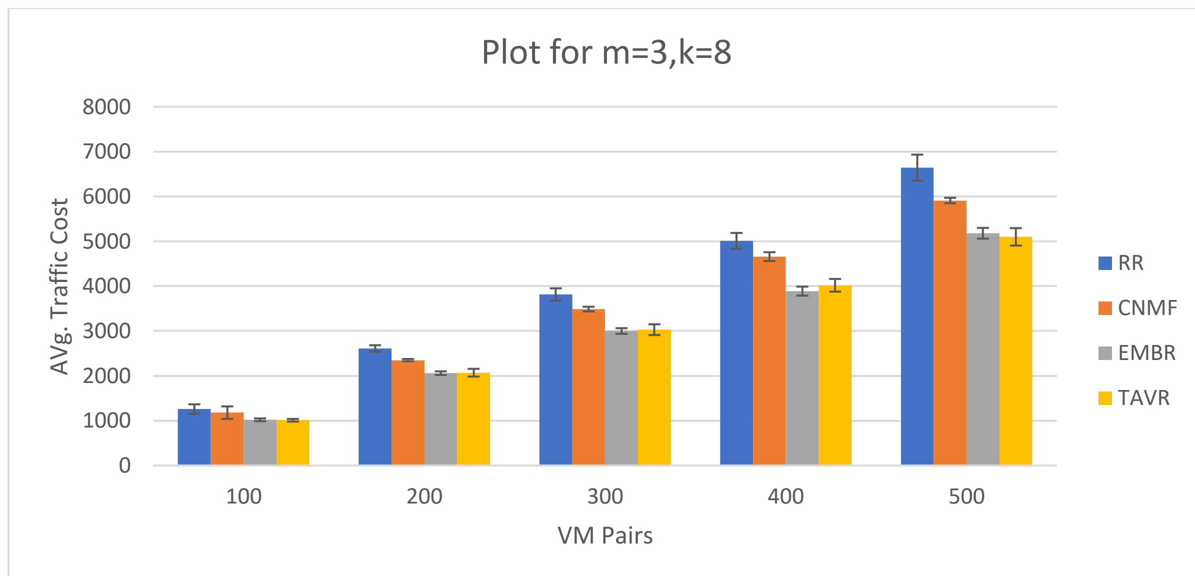


Fig. 8 Plot for m=3, k=8, p={100,200,300,400,500}

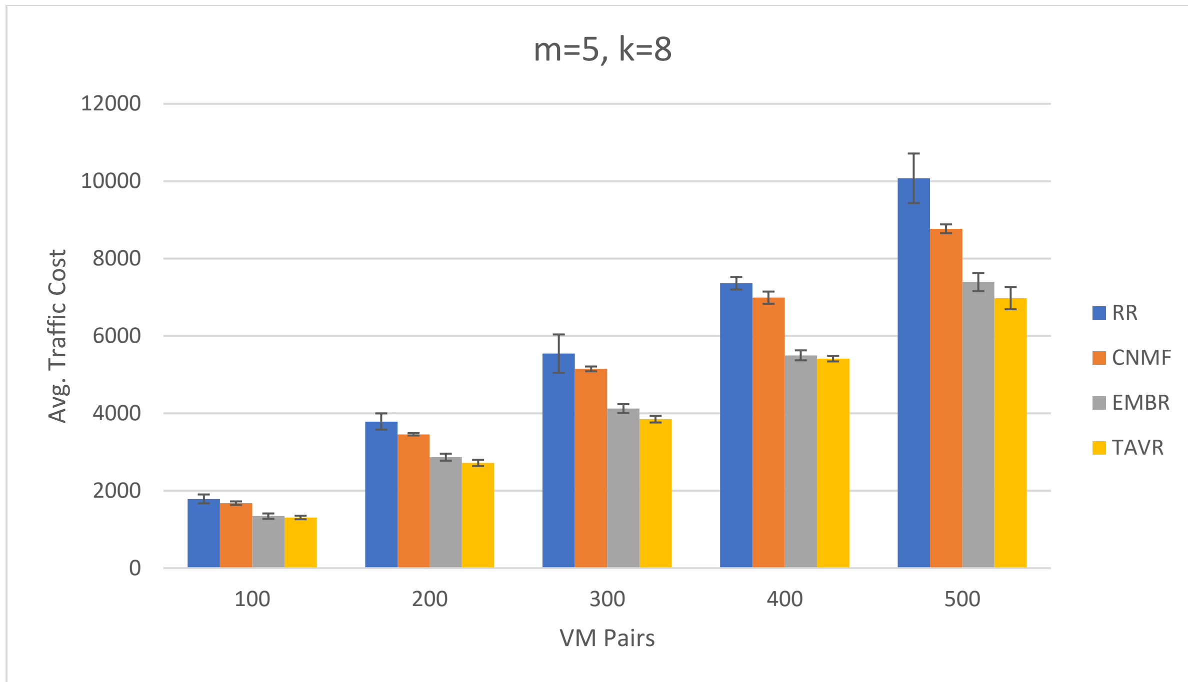


Fig. 9 Plot for m=5, k=8, p={100,200,300,400,500}

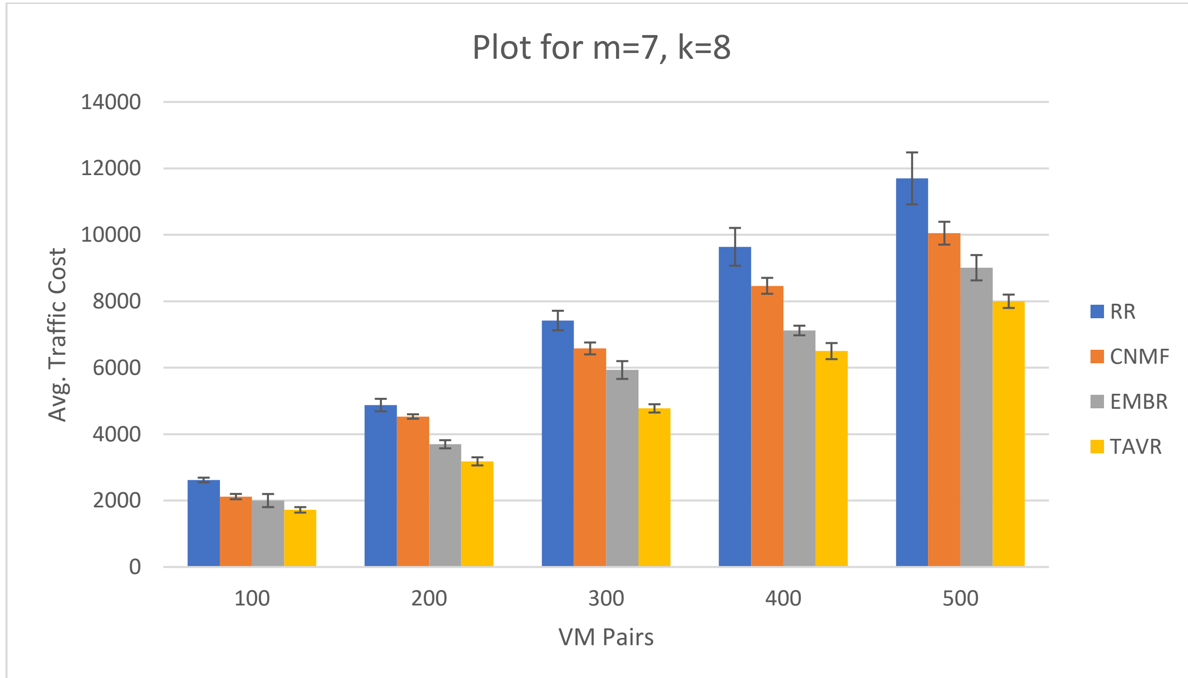


Fig. 10 Plot for m=7, k=8, p={100,200,300,400,500}

6.3 INDIVIDUAL PERFORMANCE ANALYSIS

From above plots, it is clear that the Random Replication performs poorly in terms of reduced traffic cost. It is also not reliable. Random Replication is useful in cases where only load balancing is important and over all traffic cost can be compromised, i.e., having replica copies just for the purpose of high availability. Such a use case is rare.

While Closest Next Middlebox First does provide reasonable results, choosing the shortest path within the service chain doesn't perform satisfactorily at all times. Thus, CNMF is an algorithm which yields commendable results at one time and not the best results at other time. For instance, let us consider that m=3 i.e., M={mb_1, mb_2, mb_3}. If a pair (v,v') exchange information, according to CNMF, v chooses closest mb_1, mb_1 chooses its closest mb_2 and mb_2 chooses its closest mb_3 and finally the chosen mb_3 relays the traffic to v'. At every relay, the closest for only the temporary source and destination is considered. But the overall cost yielded by (v-> mb_1) + (mb_1 –> mb_2) + (mb_2 -> mb_3) + (mb_3 – v') may not be the least cost achievable in the existing network. In this case, CNMF doesn't provide best cost.

To bridge the gaps with CNMF, Exhaustive MiddleBox Replication was designed. EMBR explores all possible combinations of available copies of middlebox types for eg., when m=3, M={mb_1, mb_2, mb_3}.Let's assume that there are two replicas of M already placed in the network. During the third iteration, that is to place the third replica of mb_1, cost yielded by both M_1={mb_1, $mb_{(2,1)}$, $mb_{(3,1)}$} and M_2={mb_1, $mb_{(2,2)}$, $mb_{(3,2)}$} are computed and compared to choose the M_x configuration that yields the lowest cost. This operation is repeated for the replication of every middlebox type. Thus, there is no chance to miss the best cost yielding service chain because all combinations are explored. The only

drawback of this algorithm is convergence time. But the algorithm has to be done for only initial set up or during a change in the network. It doesn't require to be run in an everyday basis. Thus, in data centers where long convergence time is expected for initial set up of the network, this is an ideal algorithm.

Traffic-Aware VNF replication algorithm is very efficient in scenarios where expected traffic flow among the VM pairs is already known. Based on the frequency of communication parameter that is configured for each VM pair, they are grouped into 4 groups and replications are done in favor of the groups. Thus, to fix a service chain, the overall cost that it yields in the entire network need not be computed. If a service chain seems to yield the best result for a traffic group, it is associated with members belonging to that group. If a service chain is the best for more than one group, more frequently communicating VM pair has the privilege to use it. TAVR continues the replication process only as long as there are service chains that could be replicated that yield traffic cost that is at least the same as the traffic cost yielded by original service chain. If there are rare cases where no other distribution of VNFs can yield a cost lesser than or equal to the original cost, then TAVR doesn't place any replica in the network and that is the only drawback with this algorithm. That's the reason why TAVR has larger Confidence Intervals in certain cases in the plots.

6.3 COMPARATIVE PERFORMANCE ANALYSIS

Attributes/Algorithms	RR	CNMF	EMBR	TAVR
Execution time (w.r.to K)	O (K^4)	O(K^6).	O(K^8).	O(K^4).
Advantages	1. Quick and easy way. 2. Load balancing achieved.	1. Reliable in cases where the closest VNFs serve as the best service chain	1. Ideal algorithm that doesn't miss the best cost for overall traffic flow cost as all combinations of VNFs are explored to form a service chain.	1.Best result yielding algorithm in typical networks were traffic flow is known already.
Disadvantages	1. Unreliable in terms of energy efficiency.	1. Not consistent results	1. Long convergence time.	1. Replicas cannot be placed if none of the possible replicas can yield a cost lesser than original traffic cost.

Performance	1. Average traffic cost keeps getting larger with increase in m, k and p	1. Although it doesn't perform as good as EMBR/TAVR, with the increase in m and k, it performs better and closer to EMBR because with more middlebox types and switches that hold these middleboxes, the shortest path is more often the best path.	1. EMBR performs the best among all algorithms. It performs slightly lower than TAVR in few cases as EMBR can have replicas of service chain which may produce a cost greater than original service chain. Also, for every replica, it is checked if it is optimal for all VM pairs. EMBR provides consistent results with increase in m, k and p values.	1. TAVR performs close to EMBR or at times better, as TAVR places replicas which always yield traffic ost lesser than original service chain's traffic cost. Also, every replica has to be evaluated only for the traffic group of VM pairs it belongs to. So, with increase in m, k an p, TAVR yields the best result.

CHAPTER 7

FUTURE RESEARCH AND IMPLEMENTATION DIRECTIONS

All algorithms discussed in this project primarily works for fat-tree network. The authors of [6] have worked on core mobile network. There are other widely used Data Center topologies like DCell, Leaf-Spine, Butterfly, Jellyfish etc. These algorithms can be improved to make them more generalized.

Also, the four algorithms that were implemented only performed VNF replication for service chain scenarios. As discussed in design and analysis, there could be cases where the middleboxes do not have to be visited in a particular order. Although Non-Sequential Middlebox Replication was proposed, it is not extensively tested for efficiency unlike other service-chain algorithms. Also, instead of basing the non-sequential replication on node preference, there can be better solutions as well.

The algorithms discussed here also do not include scenarios where different communicating VM pairs have different service chains of different lengths. If the VNFs are not combined as service chains, then a middlebox prioritization scheme is to be used to prioritize middlebox instances based on their demand on the network and the replication must be done accordingly.

As we keep bringing in different dimensions like the above mentioned to the replication problem, there is indeed a vast scope of extension and improvement to this project.

CHAPTER 8

CONCLUSION

With the rapid growth and demand for SDN and NFV technologies, it is imperative for network managers to adopt SDN and NFV to further optimize the network performance. SDN coupled with NFV help us meet the on-going demand for high-bandwidth applications, as well to enable simplified network management and reduced operation cost. With the ever-growing Data Centers, it is quintessential to maintain the quality of service and reduce the operational and network cost as well. Thus, it becomes inevitable to design methodologies to even focus on minute yet significant details like VNF replication on a virtualized network environment.

The algorithms developed and tested during this project are highly efficient in ensuring minimum cost flow in Data Center Networks in which the traffic flow of any traffic type between the communicating VM pairs must be processed by several network functions. All four algorithms can be used in different scenarios. Yet, pertaining to reduced overall cost, Traffic-Aware VNF replication outperforms others in a network in which expected traffic flow is given. Exhaustive Middlebox Replication algorithm is more generalized and an ideal solution to achieve the optimal average traffic cost in the network. EMBR and TAVR perform very closely in most cases but with increase in number of middlebox types and number of communicating VM pairs, TAVR outperforms EMBR by 12%-15%.

Thus, the algorithms developed in this project are energy-efficient for service chains serving different traffic demand in a fat-tree Data center. These algorithms when implemented with other proposed solutions in future research directions add more value to the future of Network Function Virtualization coupled with Software Defined Networking.

REFERENCES

[1] Sevil Mehraghdam, Matthias Keller, Holger Karl, "Specifying and Placing Chains of Virtual Network Functions", IEEE 3rd International Conference on Cloud Networking (CloudNet), 2014.

[2] Francisco Carpio and Jukan, "Balancing the Migration of Virtual Network Functions with Replications in Data Centers", arXiv:1705.05573v1 [cs.NI], 16 May 2017.

[3] Rami Cohen, "Near Optimal Placement of Virtual Network Functions", IEEE Conference on Computer Communications (INFOCOM), 2015.

[4] Francisco Carpio, Samia Dhahri and Admela Jukan, "VNF Placement with Replication for Load Balancing in NFV Networks", arXiv:1610.08266v1 [cs.NI], 26 October 2017.

[5] Pham, Nguyen H. Tran, Shaolei Ren, Walid Saad, Choong Seon Hong, "Traffic-aware and Energy-efficient vNF Placement for Service Chaining: Joint Sampling and Matching Approach", IEEE Transactions on Services Computing, 2017.

[6] Francisco Carpio, Wolgang Bziuk and Admela Jukan, "Replication of Virtual Network Functions: Optimizing Link Utilization and Resource Costs", arXiv:1702.07151v1 [cs.NI] 23 Feb 2017.

[7] http://www.tomsitpro.com/articles/nfv-network-functions-virtualization-telecom,1-1756.html

[8] https://www.sdxcentral.com/nfv/definitions/nfv-elements-overview/

[9] https://community.fs.com/blog/sdn-nfv-the-future-of-network.html

[10] *yourdailytech.com*

[11] Brian Lebiednik, Aman Mangal, Niharika Tiwari, " A Survey and Evaluation of Data Center Network Topologies", arXiv:1605.01701v1 [cs.DC] 5 May 2016.

APPENDIX

SOURCE CODE

1. CONSTRUCTION OF FAT-TREE AND OTHER COMMON METHODS

```
/* This Java Program constructs a fat-tree topology for a Datacenter
 * Methods:
 * 1. void createFatTree()
 * 2. int[][] calculateCost(int Num_AllDevices)
 * 3. void randomDistributeVM(int VMPairs)
 * 4. void randomDistributeMB(int total_mbs)
 * 5. Void randomPairVM(int VMPairs)
 */

import java.util.*;

class Devices{
	int DeviceID;
	int capacity;
	boolean isServer;
	int podID;
//	boolean isVirtual;
	ArrayList<Integer> VM;
	ArrayList<Integer> MB;
	ArrayList<Integer> mb_preference_list;
	ArrayList<Integer> neighbors;
	final static int Server_Capacity = 200; //# of VMs a server holds
	final static int Switch_Capacity = 1;  //# of MBs a switch holds
	Devices(int id, int capacity, boolean isServer){
		this.DeviceID = id;
		this.capacity = capacity;
		this.isServer = isServer;
		this.neighbors = new ArrayList<Integer>();
		if(this.isServer){
			VM = new ArrayList<Integer>();
			mb_preference_list = new ArrayList<Integer>();
			MB = null;
		}
		else{
			VM = null;
			mb_preference_list=new ArrayList<Integer>();
			MB = new ArrayList<Integer>();
		}
	}
}

public class FatTreeConstruction {
	int Num_Ports;
	int Num_Servers;
	int Num_EdgeSw; // # of edge/access switches
	int Num_AggSw; // # of aggregation switches
	int Num_CoreSw; // # of core switches
	int Num_AllSwitches;
	int Num_AllDevices;
	Devices[] devices; // objects of Devices class to hold details of every device
- switch/server
	Integer[][] cost; // cost/weight from a node/device to another
	int[][] VM_V_Pairs;
	HashMap<Integer,Integer> VM_Lookup; // VM_ID -> PM_ID
```

```
	HashMap<Integer,ArrayList<Integer>> MB_Lookup; // MB_ID -> 
ListOfSwitchesholdingMBinstances
	HashSet<Integer> original_MB_instances; //holds the switch IDs that have the 
original instances of MB only
	HashMap<Integer,ArrayList<Integer>> podSwitches;

	FatTreeConstruction(){
		Num_Ports = 0;
		Num_Servers = 0;
		Num_EdgeSw = 0;
		Num_AggSw = 0;
		Num_CoreSw = 0;

	}

/* Method Name : createFatTree
 * Purpose : Gets input on number of switch ports and constructs fat-tree topology
 * Details : 1. Number of switch ports: Num_ports
 *                     2. Number of pods           : Num_ports
 *                     3. Number of edge_switch : (Num_ports ^ 2)/2
 *                     4. Number of aggregate_sw: (Num_ports ^ 2)/2
 *                     5. Number of core switch : (Num_ports/2) ^ 2
 *                     6. Number of servers(PM) : (Num_ports ^ 3)/4
 *
*/
public int createFatTree(int ports){
	/*Scanner input = new Scanner(System.in);
	Num_Ports = input.nextInt();
	input.close();*/
	Num_Ports = ports;
	while(Num_Ports % 2 != 0 || Num_Ports < 4 ){
		System.out.println("Enter an even number >=4 for number of ports as 
DataCenter is based on a fat-tree topology!");
	}
	Num_Servers = (int)(Math.pow(Num_Ports, 3))/4;
	Num_CoreSw = (int)(Math.pow((Num_Ports/2), 2));
    Num_AggSw = (int)(Math.pow(Num_Ports, 2))/2;
    Num_EdgeSw = Num_AggSw;
    Num_AllSwitches = Num_EdgeSw + Num_AggSw + Num_CoreSw; // total # of switches
    Num_AllDevices = Num_AllSwitches + Num_Servers; // total # of switches and servers
    devices = new Devices[Num_AllDevices];
    podSwitches = new HashMap<Integer,ArrayList<Integer>>();
    int podID=1, maxpodID = Num_Ports;
    int nonCoreSwitchCount=0;
    ArrayList<Integer> switchList;
    for (int counter = 0 ; counter < Num_AllDevices; counter++){
	if(counter < Num_Servers)
		devices[counter] = new Devices(counter+1,0,true);
	else
	{
		devices[counter] = new Devices(counter+1,0,false);

				if (counter >= Num_Servers && counter < 
Num_Servers+Num_EdgeSw+Num_AggSw){
			devices[counter].podID = podID;
			if (podSwitches.containsKey(podID))
				switchList = podSwitches.get(podID);
			else
				switchList = new ArrayList<Integer>();
			switchList.add(counter);
			podSwitches.put(podID, switchList);
			nonCoreSwitchCount++;
			if (nonCoreSwitchCount == Num_Ports/2){
```

```
                                if (podID < maxpodID)
                                       podID++;
                                else
                                       podID=1;
                                nonCoreSwitchCount=0;
                         }
                  }

           }
     }
     System.out.println("Number of Servers/Physical Machines: "+Num_Servers);
     System.out.println("Number of Edge switches:  "+Num_EdgeSw);
     System.out.println("Number of Aggregate switches:  "+Num_AggSw);
     System.out.println("Number of Core switches: "+Num_CoreSw);
     for(int i=1;i<=Num_Ports;i++){
        System.out.println("swiches belonging to pod "+i+" are:
"+podSwitches.get(i).toString());
     }
     return Num_AllDevices;
     }

public void resetFatTree(int mb_types){
       int[] original_mbs = new int[mb_types];

       @SuppressWarnings("rawtypes")
       Iterator it = MB_Lookup.entrySet().iterator();
       while(it.hasNext()){
              Map.Entry<Integer,ArrayList<Integer>> pair = (Map.Entry)it.next();
              original_mbs[pair.getKey()-1] = pair.getValue().get(0); // original mb is
stored in 0th index
       }
       // MB_Lookup.clear();
       for (int i=0;i<mb_types;i++){
              ArrayList<Integer> switchList=new ArrayList<Integer>(1);
              switchList.add(original_mbs[i]);
              MB_Lookup.put(i+1, switchList);
       }

       for (int i=Num_Servers;i<Num_AllDevices;i++){
                     //Resetting all switches that do no host original middlebox
                     if (!original_MB_instances.contains(i )){
                            this.devices[i].MB=new ArrayList<Integer>();
                            this.devices[i].capacity=0;
                     }
                     this.devices[i].mb_preference_list=new ArrayList<Integer>();

       }
       for(int i=0;i<Num_Servers;i++){
              this.devices[i].mb_preference_list=new ArrayList<Integer>();
              this.devices[i].capacity=0;
              this.devices[i].VM=new ArrayList<Integer>();
       }
       System.out.println("Reset Completed for All Devices");

}

/* Method Name : calculateCost()
 * Purpose : Calculates cost between any two node in the fat-tree
 * Details : Input: Total number of all the devices in the fat-tree(int)
 *                      Output: int[][] cost
*/

public Integer[][] calculateCost(int Num_AllDevices){
```

```
        cost = new Integer[Num_AllDevices][Num_AllDevices];
        //initialize costArray to Integer.MAX_VALUE
        for ( int loop = 0; loop < Num_AllDevices;loop++)
        Arrays.fill(cost[loop], 10000);  // An impossible large value to start with .
Integer.MAX_VALUE behaves unexpected

        // cost of every node to itself is 0
        for(int counter = 0;counter < Num_AllDevices;counter++)
                cost[counter][counter]=0;

        // cost between servers and edge switches to which they are directly connected
is 1
        int Server_id = 0;
        for(int switch_id=Num_Servers; switch_id < Num_Servers+Num_EdgeSw;switch_id++
){
                for (int counter = 0; counter < Num_Ports/2; counter++){
                        cost[Server_id][switch_id] = 1;
                        cost[switch_id][Server_id] = 1;
                        devices[Server_id].neighbors.add(switch_id);
                        devices[switch_id].neighbors.add(Server_id);
                        Server_id++;
                }
        }

        //cost between directly connected edge switches and aggregate switches is 1
        int agg_switch = Num_Servers + Num_EdgeSw ;
        int temp = agg_switch;
        for(int switch_id=Num_Servers; switch_id < Num_Servers+Num_EdgeSw;switch_id++
){
                for (int counter = 0; counter < Num_Ports/2; counter++){
                        cost[agg_switch][switch_id] = 1;
                        cost[switch_id][agg_switch] = 1;
                        devices[agg_switch].neighbors.add(switch_id);
                        devices[switch_id].neighbors.add(agg_switch);
                        agg_switch++;

                }
                if ( (switch_id+1)%(Num_Ports/2) != 0)
                        agg_switch = temp;
                else
                        temp = agg_switch;
        }

        //cost between directly connected aggregate switch and core switch is 1
        int core_sw = Num_Servers + Num_EdgeSw + Num_AggSw;
        temp = core_sw;
        for(agg_switch= Num_Servers + Num_EdgeSw ; agg_switch <
Num_Servers+Num_EdgeSw+Num_AggSw;agg_switch++ ){
                for (int counter = 0; counter < Num_Ports/2; counter++){
                        cost[agg_switch][core_sw] = 1;
                        cost[core_sw][agg_switch] = 1;
                        devices[agg_switch].neighbors.add(core_sw);
                        devices[core_sw].neighbors.add(agg_switch);
                        core_sw++;

                }
                if ( (agg_switch+1)%(Num_Ports/2) == 0)
                        core_sw = temp;

        }

        // Minimum cost calculation between every node in the DataCenter using Floyd's
algorithm.
```

```
   for(int k = 0 ; k < Num_AllDevices ; k++){
      for(int i = 0 ; i < Num_AllDevices ; i++){
         for(int j = 0 ; j < Num_AllDevices ; j++){

             cost[i][j] = Math.min(cost[i][j], cost[i][k]+cost[k][j]);

         }
      }
   }


       return cost;
}
/*Method name: void randomDistributeVM
 */
public void randomDistributeVM(int VMPairs){
       Random generator = new Random();
       int RandomPM;
       VM_Lookup = new HashMap<Integer,Integer>();

       //Randomly place the VMs in different servers across the Datacenter by checking
the capacity constraint
       for (int counter = 0; counter < 2*VMPairs; counter++){
       //     System.out.println(counter);
              do{
                     // Causes very long running loop until a desired PM is
found.Increased server_capacity to 50 from 10 to fix this
                     System.out.println("Finding the right host...");
              RandomPM = generator.nextInt(Num_Servers);
              System.out.println(RandomPM);
              }while(devices[RandomPM].capacity == Devices.Server_Capacity);

                     devices[RandomPM].VM.add(counter+1);  // add the VM the server
                     devices[RandomPM].capacity++;
                     VM_Lookup.put(counter+1, RandomPM);
       }
}
/*Method name: void randomDistributeMB
 * Purpose : Distributes middlebox of each type across the switches creating one
instance each
 * Details : Input: The number of middlebox types
 *                  Populates the MBLookup HashMap with the distribution
 *
 */
public void randomDistributeMB(int mb_types){
       original_MB_instances = new HashSet<Integer>();
       Random generator = new Random();
       int RandomSw;
       ArrayList<Integer> switch_list;
       MB_Lookup = new HashMap<Integer,ArrayList<Integer>>();
       //Randomly place the VMs in different servers across the Datacenter by checking
the capacity constraint
       for (int counter = 0; counter < mb_types; counter++){
              do{
              RandomSw = (Num_Servers)+generator.nextInt(Num_AllDevices-Num_Servers);
// MBs can only be placed on switches
              }while(devices[RandomSw].capacity == Devices.Switch_Capacity);

                     devices[RandomSw].MB.add(counter+1);  // add the middlebox to the
switch
                     devices[RandomSw].capacity++;
                     switch_list=new ArrayList<Integer>(1);
                     switch_list.add(RandomSw);
```

```
                    original_MB_instances.add(RandomSw);
                    MB_Lookup.put(counter+1, switch_list);
                    System.out.println("Middleboxes are placed"+MB_Lookup.size());
        }
}
/*Method name: randomPairVM()
 * Purpose : Random pairing of the available VMs for traffic flow between them
 * Details : Input: Total number of VMPairs
 *
 */
public int[][] randomPairVM(int VMPairs) throws CustomException{
        if(VMPairs < 1)
                throw new CustomException("There should be atleast one VM pair");
        VM_V_Pairs=new int[VMPairs][2];
        HashSet<Integer> pairedVMs = new HashSet<Integer>(); // To keep track of
already paired VMs
        Random generator = new Random();
        int vm1,vm2;

        for(int counter = 0; counter < VMPairs; counter++){
                do{
                        vm1 = generator.nextInt(2*VMPairs)+1;
                }while(pairedVMs.contains(vm1));
                pairedVMs.add(vm1);
                do{
                         vm2 = generator.nextInt(2*VMPairs)+1;
                }while(pairedVMs.contains(vm2));
                pairedVMs.add(vm2);
                VM_V_Pairs[counter][0] = vm1;
                VM_V_Pairs[counter][1] = vm2;
                //System.out.println(VM_V_Pairs[counter][0]+" "+VM_V_Pairs[counter][1]);
        }

        return VM_V_Pairs;
}

/*Method name: CalculateTrafficFlowCost(int[] MB_Switches)
 * Purpose : Calculates the cost for Traffic flow between all VMPairs in the network
 * All Traffic must flow through the sequence of given Middleboxes.
 */
 public int calculateTrafficFlowCost(int[] MB_Switches, ArrayList<Integer>
traffic_group){

         if(MB_Switches == null || VM_V_Pairs == null)
                 return -1;

         int TotalCost=0;
         if (traffic_group == null){
         for(int counter = 0; counter  < VM_V_Pairs.length; counter++){
                 if(VM_V_Pairs[counter] == null)
                         return -1;
                 int VM1 = VM_V_Pairs[counter][0];
                 int VM2 = VM_V_Pairs[counter][1];

         //calculate cost between VM1 and first middlebox
         TotalCost += cost[VM_Lookup.get(VM1)][MB_Switches[0]];

         //calculate cost between the given sequence of middleboxes
         for(int counter_in = 0; counter_in < MB_Switches.length-1; counter_in++){
                 TotalCost += cost[MB_Switches[counter_in]][MB_Switches[counter_in+1]];
         }

        //calculate cost between last middlebox and VM2
```

```
                TotalCost += cost[VM_Lookup.get(VM2)][MB_Switches[MB_Switches.length-
1]];
        }
        }
        else{
                for(int counter : traffic_group){
                        if(VM_V_Pairs[counter] == null)
                                return -1;
                        int VM1 = VM_V_Pairs[counter][0];
                        int VM2 = VM_V_Pairs[counter][1];

                //calculate cost between VM1 and first middlebox
                TotalCost += cost[VM_Lookup.get(VM1)][MB_Switches[0]];

                //calculate cost between the given sequence of middleboxes
                for(int counter_in = 0; counter_in < MB_Switches.length-1;
counter_in++){
                        TotalCost +=
cost[MB_Switches[counter_in]][MB_Switches[counter_in+1]];
                }

               //calculate cost between last middlebox and VM2
                        TotalCost +=
cost[VM_Lookup.get(VM2)][MB_Switches[MB_Switches.length-1]];
                }
        }
        return TotalCost;
 }
 /*
  * Method for associating VM Pairs to a traffic frequency group
  * Frequency distribution for [very_frequent, frequent, medium, less] is formulated
as [40%,45%,12%,3%]
  */
 public HashMap<Integer,ArrayList<Integer>> frequencyMapper(){
        HashMap<Integer,ArrayList<Integer>> frequencyMap = new
HashMap<Integer,ArrayList<Integer>>();
        int VMPairs = VM_V_Pairs.length;
        int very_frequent_length = (int)Math.floor(0.4 * VMPairs);
        int frequent_length = very_frequent_length+(int)Math.floor(0.45 * VMPairs);
        int medium_length = frequent_length + (int)Math.floor(0.12 * VMPairs);
        int less_length = medium_length + (int) Math.floor(0.03 * VMPairs);
        ArrayList<Integer> traffic_group = new ArrayList<Integer>();
        int frequency = 0;
        for (int counter=0;counter<VMPairs;counter++){
                if (counter == very_frequent_length || counter == frequent_length ||
counter == medium_length ){
                        frequencyMap.put(frequency, traffic_group);
                        traffic_group = new ArrayList<Integer>();
                        frequency++;
                }
                traffic_group.add(counter);
                if (counter == less_length-1){
                        frequencyMap.put(frequency, traffic_group);
                }
        }
        return frequencyMap;
 }
/*
 * Method for calculating cost only within a traffic group
 */

  /*
  * This method calculates traffic cost for a vm pair
```

```
  */
 public int calculateVMPairTrafficCost(int pairID, int[] mbs){
        int TotalCost = cost[VM_Lookup.get(VM_V_Pairs[pairID][0])][mbs[0]];

        //calculate cost between the given sequence of middleboxes
        for(int counter_in = 0; counter_in < mbs.length-1; counter_in++){
              TotalCost += cost[mbs[counter_in]][mbs[counter_in+1]];
        }

       //calculate cost between last middlebox and VM2
              TotalCost += cost[VM_Lookup.get(VM_V_Pairs[pairID][1])][mbs[mbs.length-
1]];

        return TotalCost;
 }
```

2. RANDOM REPLICATION ALGORITHM

```
/*
 * This Java Program is used to randomly replicate middleboxes inside a k-ary fat-
tree network.
 */
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Random;

public class RandomReplication {

       public static void randomReplicator(FatTreeConstruction network,int mb_types){
              boolean isreplicable=true;
              int numReplication = network.Num_AllSwitches/mb_types;
              int sw;
              int rand_mb_cost;
              double avg_cost=0;
              int best_cost= Integer.MAX_VALUE;
              int worst_cost = Integer.MIN_VALUE;
              int[] original_mbs= MB_Replication.getOriginalMBSequence(mb_types);
              int[] random_mbs;

              while(isreplicable){
                     int if_execution_counter=0;
                     for(int i=1;i<=mb_types;i++){
                            if(network.MB_Lookup.get(i).size() < numReplication){ // if
a mb is still replicable
                                   if_execution_counter++;
                                   Random generator = new Random();
                                          do{
                                                 sw =
network.Num_Servers+generator.nextInt(network.Num_AllSwitches);
                                          }while(network.devices[sw].capacity >=
Devices.Switch_Capacity);
                                          network.devices[sw].MB.add(i);
                                          network.devices[sw].capacity++;
                                          ArrayList<Integer> switch_list =
network.MB_Lookup.get(i);
                                          switch_list.add(sw);
                                          network.MB_Lookup.put(i,switch_list );
                                          random_mbs= Arrays.copyOf(original_mbs,
mb_types);
                                          random_mbs[i-1]=sw;
```

```
                                        rand_mb_cost =
network.calculateTrafficFlowCost(random_mbs,null);
                                        if (rand_mb_cost < best_cost)
                                              best_cost = rand_mb_cost;
                                        if(rand_mb_cost > worst_cost)
                                              worst_cost = rand_mb_cost;
                                        avg_cost += rand_mb_cost;
                           }
                    }
                    if (if_execution_counter == 0)
                           isreplicable = false;

             }
             //System.out.println("Avg cost flow "+(avg_cost/(numReplication
*mb_types)));
             System.out.println("Average best cost is
"+best_cost/network.VM_V_Pairs.length);
             System.out.println("Average worst cost is
"+worst_cost/network.VM_V_Pairs.length);
             System.out.println("Average overall cost from RR is
"+(best_cost+worst_cost)/2);

      }

}
```

3. CLOSEST NEXT MIDDLEBOX FIRST ALGORITHM

```
/*
 *  This Java Program is used to replicate middleboxes inside a k-ary fat-tree
network.
 *  A VM pair source or a VNF(i) chooses the closest next VNF(j) in the sequence from
all the availables VNF(j)s
 */

import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.Comparator;
import java.util.Iterator;
import java.util.Map;
import java.util.Map.Entry;


public class improvedClosestNextMB {
      static int primary_switch;
      FatTreeConstruction network;
      static boolean[] isVisited;

      improvedClosestNextMB (FatTreeConstruction network){
             this.network= network;
             this.isVisited=new boolean[network.Num_Servers];

      }

      /*
       * Comparator implementation for sorting the mb_preference_list of a switch in
the increasing order of cost between that switch and that mb.
       */
```

```
	public Comparator<Integer> CostComparator = new Comparator<Integer>(){

		@Override
		public int compare(Integer switch1, Integer switch2) {
			// TODO Auto-generated method stub
		return Integer.compare(network.cost[primary_switch][switch1],
		network.cost[primary_switch][switch2]);
		}

	};

	public void initializeMbPreferenceList(int mb_types){
		for(int i=0;i<network.Num_Servers;i++)
			isVisited[i]=false;
		int[] original_mbs= MB_Replication.getOriginalMBSequence(mb_types);

		//Initializing MB Preference List for all VM Pairs
		for(int p=0;p<network.VM_V_Pairs.length;p++){

network.devices[network.VM_Lookup.get(network.VM_V_Pairs[p][0])].mb_preference_list.cl
ear();

	network.devices[network.VM_Lookup.get(network.VM_V_Pairs[p][0])].mb_preference_
list.add(original_mbs[0]);
		//	System.out.println("Mb pref list size after initialization
"+network.devices[network.VM_Lookup.get(network.VM_V_Pairs[p][0])].mb_preference_list.
size());
		}

		//Initializing Next preferable mb in the sequence for all mbs
		Iterator<Entry<Integer, ArrayList<Integer>>> it =
	network.MB_Lookup.entrySet().iterator();
		while(it.hasNext()){
			Map.Entry<Integer, ArrayList<Integer>> pair= (Map.Entry)it.next();
			int m= (int)pair.getKey();
			if(m==mb_types)
				break;
			ArrayList<Integer> mbs = (ArrayList<Integer>) pair.getValue();
			for (Integer mb: mbs){
			//	System.out.println("Setting pref list for "+mb+" as
"+original_mbs[m]);

	network.devices[mb].mb_preference_list.add(original_mbs[m]);
			}
		}
	}

	public void closestNextReplication(int mb_types){
		double avg_cost=0;
		int[] original_mbs= MB_Replication.getOriginalMBSequence(mb_types);
		int originalCost = network.calculateTrafficFlowCost(original_mbs,null);
		initializeMbPreferenceList(mb_types);
		int replication = network.Num_AllSwitches/mb_types;
		//For every replication
		for(int i=0;i<replication;i++){
			//Place a replica of every middlebox
			for(int j=1;j<=mb_types;j++){

				int mincost=Integer.MAX_VALUE;
				int mincostswitch=-1;
				//System.out.println("Runing algrithm to replicate "+j);
				//check availability and resulting vost in every switch
			for(int k=network.Num_Servers;k<network.Num_AllDevices;k++){
```

```
	if(network.devices[k].capacity<Devices.Switch_Capacity){
			int currentCost=0;
	//Find path to current switch from every VM pair
	//Find next closest MB from current switch
	//calculate cost and compare
	for(int vm=0;vm<network.VM_V_Pairs.length;vm++){
		int[] test_mbs=new int[mb_types];
		if (j==1)
			test_mbs[0]=k;
		else

test_mbs[0]=network.devices[network.VM_Lookup.get(network.VM_V_Pairs[vm][0])].mb_prefe
rence_list.get(0);
		for(int m=1;m<mb_types;m++){
			if (m+1==j)
				test_mbs[m]=k;
			else if (m==j){
				ArrayList<Integer> availableNextMbs =
network.MB_Lookup.get(m+1);
				int minNextMBCost = Integer.MAX_VALUE;
				int nextMBswitch=-1;
				for(int mb: availableNextMbs){
				if (network.cost[k][mb] < minNextMBCost){

					minNextMBCost=network.cost[k][mb];

					nextMBswitch = mb;
				}
				}
			test_mbs[m] = nextMBswitch;
			}
			else

	test_mbs[m]=network.devices[test_mbs[m-1]].mb_preference_list.get(0);
						//			System.out.println("Chosen
mb "+m+" in the sequence is "+test_mbs[m]);
			}
			currentCost+= network.calculateVMPairTrafficCost(vm,test_mbs);

		   }
		   if(currentCost < mincost){
			mincost = currentCost;
			mincostswitch = k;
		   }

		}
		}

	if (mincostswitch != -1){

	MB_Replication.updateDevices(mincostswitch,j);
	avg_cost += (double)mincost;
	// The next set of lines of code is for including new mb switch in previous
mb's preference list and sort it based on cost
	if(j==1){
		for(int sw=0;sw<network.Num_Servers;sw++)
			isVisited[sw]=false;
			for(int vm=0;vm<network.VM_V_Pairs.length;vm++){
			primary_switch = network.VM_Lookup.get(network.VM_V_Pairs[vm][0]);
```

```
if (!isVisited[primary_switch]){

	network.devices[primary_switch].mb_preference_list.add(mincostswitch);

	Collections.sort(network.devices[primary_switch].mb_preference_list,CostCompara
tor);
								isVisited[primary_switch]=true;
	}

}

}
else{
	ArrayList<Integer> previous_mbs = network.MB_Lookup.get(j-1);
	for (Integer mb: previous_mbs){
		primary_switch=mb;

	network.devices[primary_switch].mb_preference_list.add(mincostswitch);

	Collections.sort(network.devices[primary_switch].mb_preference_list,CostCompara
tor);
}
}
if(j < mb_types)
	for (Integer mb_next: network.MB_Lookup.get(j+1)){

	network.devices[mincostswitch].mb_preference_list.add(mb_next);

	Collections.sort(network.devices[mincostswitch].mb_preference_list,CostComparat
or);
}
}
}
}
		sendTraffic(mb_types,replication);
}
public void sendTraffic(int mb_types, int totalReplicas){
		int[] test_mbs=new int[mb_types];
		int totalCost=0;
		for (int vm=0;vm<network.VM_V_Pairs.length;vm++){
			int vm_switch=network.VM_Lookup.get(network.VM_V_Pairs[vm][0]);

	test_mbs[0]=network.devices[vm_switch].mb_preference_list.get(totalReplicas-1);
			for (int i=1;i<mb_types;i++){
			test_mbs[i]=network.devices[test_mbs[i-
1]].mb_preference_list.get(totalReplicas-1);
			}
			totalCost += network.calculateVMPairTrafficCost(vm, test_mbs);
		//	System.out.println("Total cost flow in the network after sending
traffic via vm pair "+vm+" is "+totalCost);
		}
		System.out.println("Average worst cost flow among all VM pairs is
"+totalCost/network.VM_V_Pairs.length);
		int worst_cost = totalCost;
		totalCost=0;
		for (int vm=0;vm<network.VM_V_Pairs.length;vm++){

	test_mbs[0]=network.devices[network.VM_Lookup.get(network.VM_V_Pairs[vm][0])].m
b_preference_list.get(0);
			for (int i=1;i<mb_types;i++)

test_mbs[i]=network.devices[test_mbs[i-1]].mb_preference_list.get(0);
totalCost += network.calculateVMPairTrafficCost(vm, test_mbs);
```

```
                    //System.out.println("Total cost flow in the network after sending
traffic via vm pair "+vm+" is "+totalCost);
              }
              System.out.println("Average best cost flow among all VM pairs is
"+totalCost/network.VM_V_Pairs.length);
              int bestCost=totalCost;
              System.out.println("Avg cost flow  from CNMBF:
"+(bestCost+worst_cost)/2);
       }

}
```

4. EXHAUSTIVE MB REPLICATION ALGORITHM

```
/*
 *  This Java Program is used to replicate middleboxes inside a k-ary fat-tree
network.
 *  The sequence of VNFS that yield the best lowest cost along with the current VNF
under consideration VNF(i) are chosen from all the available VNFs
 */

import java.util.ArrayList;
import java.util.Arrays;


public class ExhaustiveMbReplication {
       static void replicateMB(FatTreeConstruction network,int mb_types){
              int num_replication=network.Num_AllSwitches/mb_types;
              int[] mb_pointer = new int[mb_types];
              int last_mb;
              double avg_cost=0;
              int best_cost=Integer.MAX_VALUE, worst_cost=Integer.MIN_VALUE;
              for(int i=1;i<num_replication;i++){ //total iterations
                     for(int j=1;j<=mb_types;j++){ //place copy of this middlebox
                            int mincost=Integer.MAX_VALUE;
                            int mincost_switch = -1;
                            for(int
k=network.Num_Servers;k<network.Num_AllDevices;k++){ // try placing the copy on every
available switch
                                   if(network.devices[k].capacity <
Devices.Switch_Capacity){
                                   int[] test_mbs= new int[mb_types];
                                   Arrays.fill(mb_pointer, 0);
                                   //System.out.println("Trying to place mb "+j+" on
"+k);

                                   int current_mb;
                                   if(j==1){
                                          last_mb=1;
                                          test_mbs[0]=k;
                                          current_mb = mb_types-1;
                                   }
                                   else if (j== mb_types){
                                          last_mb=0;
                                          test_mbs[j-1]=k;
                                          current_mb = mb_types-2;
                                   }
                                   else
last_mb=0;
test_mbs[j-1]=k;
current_mb = mb_types-1;
```

```
}

while( current_mb >= last_mb ){ // until all combinations are exhausted for the
current replication
       if (j < current_mb+1 && mb_pointer[current_mb] == i)
              break;
       if(j > current_mb+1 && mb_pointer[current_mb] > i)
              break;
              for(int mb=last_mb;mb<current_mb;mb++){
                     if (mb+1==j)
                            continue;
              test_mbs[mb]=network.MB_Lookup.get(mb+1).get(mb_pointer[mb]);
                                                  //System.out.println("Choosing
"+test_mbs[mb]+" for mb "+(mb+1));
              }
              int host;
              while(true){
                     if (j < current_mb+1 && mb_pointer[current_mb] == i)
                            break;
                     if(j > current_mb+1 && mb_pointer[current_mb] > i)
                            break;
                     if (current_mb+1==j){
                            current_mb--;
                            continue;
                     }

                     for(int mb=current_mb;mb<mb_types;mb++){
                            if (mb==j-1)
                                   continue;
                            host = mb_pointer[mb];

       test_mbs[mb]=network.MB_Lookup.get(mb+1).get(host);
}
int trafficCost = network.calculateTrafficFlowCost(test_mbs, null);
if(trafficCost < mincost){

       mincost = trafficCost;
       mincost_switch = k;

       int prev_mb=-1;
       for(int mb=mb_types-1;mb>=current_mb;mb--){
              if (j==mb+1){
                     if (j==mb_types){
                            prev_mb = current_mb;
                     }
                     else{
                            prev_mb=mb+1;
                            continue;
                     }
                     if(mb == mb_types-1){

                            mb_pointer[mb]++;
                     }
                     else{
                      if (prev_mb+1 < j ){
                             if(j==mb_types){
                                   mb_pointer[prev_mb]++;
                                   continue;
                             }

       if(mb_pointer[prev_mb] > i){ //changed > i to ==i
```

```
                                                                //
        System.out.println("crossed "+i+" for "+(prev_mb));


        mb_pointer[prev_mb]=0;

        mb_pointer[mb]++;
        }
        }
        else{
        if(mb_pointer[prev_mb] > i-1){ //changed > i to ==i
                                                                        //
        System.out.println("crossed "+(i-1)+" for "+(prev_mb));

        mb_pointer[prev_mb]=0;

        mb_pointer[mb]++;
        }
        }

        }
        prev_mb=mb;
        }
        }
        mb_pointer[current_mb]=0;

        current_mb--;
        if(current_mb==j)
                current_mb--;
        if (current_mb >= 0)
                mb_pointer[current_mb]++;
                                                                                        }

        }
        }

        if (mincost_switch!=-1){
                                        MB_Replication.updateDevices(mincost_switch,j);
                                        if (mincost < best_cost)
                                                best_cost = mincost;
                                        if (mincost > worst_cost)
                                                worst_cost = mincost;
                                        avg_cost += (double)mincost;
                                }


        }
        }
                //System.out.println("Average cost flow from exhaustive repplication is
"+avg_cost/(num_replication*mb_types));
                System.out.println("Average best cost is
"+best_cost/network.VM_V_Pairs.length);
                System.out.println("Average worst cost is
"+worst_cost/network.VM_V_Pairs.length);
                System.out.println("Avg cost flow  from EMBR:
"+(best_cost+worst_cost)/2);
}
}
```

5. TRAFFIC AWARE VNF REPLICATION ALGORITHM

```
/*
 *  This Java Program is used to replicate middleboxes inside a k-ary fat-tree
network.
 *  The sequence of VNFS that yield the best lowest cost which is atleast the same as
original cost are replicated.
* The idea is based on probability distribution of VM pairs in the network and the
priority of their communication frequency.
*/

import java.util.*;
import java.util.Map.Entry;

class MB_Replication {
	static FatTreeConstruction network; //an object of the network topology
	static int[] memoization_prev_closestMB; // an array to implement dynamic
programming for saving memory while calculating
	//the distance from one middlebox to another in order to end up with min
distance eventually
	static double avg_cost=0; //for experimental purposes
	static HashMap<Integer,ArrayList<Integer>> serviceChains;


	/*
	 * Method to get the original sequence of MBs
	 */
	public static int[] getOriginalMBSequence(int mb_types){
		int[] original_mbs = new int[mb_types];
		@SuppressWarnings("rawtypes")
		Iterator it = network.MB_Lookup.entrySet().iterator();
		while(it.hasNext()){
			Map.Entry<Integer,ArrayList<Integer>> pair = (Map.Entry)it.next();
			original_mbs[pair.getKey()-1] = pair.getValue().get(0); //
original mb is stored in 0th index
		}
		return original_mbs;
	}

	/*
	 *
	 */
	public static void updateDevices(int switchID,int mb){
		network.devices[switchID].MB.add(mb);
		network.devices[switchID].capacity++;
		ArrayList<Integer> switch_list = network.MB_Lookup.get(mb);
		switch_list.add(switchID);
		network.MB_Lookup.put(mb,switch_list );
		//System.out.println("Middlebox "+mb+" replicated on "+switchID);
	}

	/*Method name: replicateMB(int mb_types)
	 * Purpose : Replicates the given types of middleboxes across the network
	 * Populates the MBLookup HashMap with the list of switch-ids hosting every
middlebox
	 * Details : Input : Number of middlebox types
	 *                     Pre-condition : Fat-tree network is already created,
original instance of middleboxes are distributed
	 *              Assumptions    : A switch can hold only one instance of a
middleboxtype
	 *                    Running Time : O(N/M * M * N) = O(N^2)
```

```
        */
       public static void replicateMB_shortestPath_withinMBs(int mb_types,
ArrayList<Integer> traffic_group){

              //These arrays hold the switchIDs that host the middleboxes.
              int[] original_mbs;
              int[] mb_copys;
              avg_cost=0;
              //Calculate Total cost without replication
              original_mbs= getOriginalMBSequence(mb_types);
              int originalCost =
network.calculateTrafficFlowCost(original_mbs,traffic_group);
              System.out.println("Before replication cost "+originalCost);

              // Replication Algorithm Begins
              Arrays.fill(memoization_prev_closestMB,-1);
              System.out.println("Total iterations required
"+network.Num_AllSwitches/mb_types);
              for(int counter = 0; counter < network.Num_AllSwitches/mb_types;
counter++){
                     //Every iteration places one replica copy of each mb type in the
network

                     for(int counter_in = 0; counter_in < mb_types; counter_in++){

                     //     System.out.println("Currently replicating middlebox
"+(counter_in+1));
                            int mincost = Integer.MAX_VALUE;
                            int switchID = network.Num_Servers;
                            int mincost_switch = switchID;
                            int nearestPrevMB = -1;
                            int[] prev_mb_positions = new int[counter+2]; // An array
to hold all the host switch ids of the previous middlebox type
                            // As each outer for loop creates one replica of the Prev
middlebox and there is a original mb

                            if(counter_in > 0){
                                   // From middlebox type 2 to n . Note: middleboxes are
indexed from 1 to n and not 0 to n-1
                                   for ( int
loop=0;loop<network.MB_Lookup.get(counter_in).size(); loop++){
                                          prev_mb_positions[loop] =
network.MB_Lookup.get(counter_in).get(loop);
                                   }
                            }

                            //Find the switch that incurs least cost in the network
when it holds the replica copy of given middlebox
                            while(switchID < network.Num_AllDevices) {

                                   if(network.devices[switchID].capacity <
Devices.Switch_Capacity){
                                   //     System.out.println("Verifying if "+switchID+"
can host a replica");
                                          mb_copys = new int[mb_types]; //This array
holds the switches hosting preceding middlebox types that incur min cost

                                          //Initialize middlebox copies with middlebox
originals
                                          for(int loop=0;loop<mb_types;loop++)
                                                 mb_copys[loop]=original_mbs[loop];

                                          mb_copys[counter_in] = switchID;
```

```
                                        //Find the shortest route to the current
switch considered for MB replication from the first middlebox
                                        if(counter_in > 0){

                                              int loop=counter_in-1;
                                              int current_mb = switchID;

                                              while(loop >= 0){


      if(memoization_prev_closestMB[current_mb-network.Num_Servers] == -1){

                                                          mb_copys[loop] =
findMinCostPreviousMB(prev_mb_positions,switchID);

                                                    }
                                                    else{

                                                          mb_copys[loop] =
memoization_prev_closestMB[current_mb-network.Num_Servers];
                                                    }
                                                    current_mb = mb_copys[loop];
                                                    loop--;
                                              }
                                        }

                                        int current_cost =
network.calculateTrafficFlowCost(mb_copys, traffic_group);
                                  //    System.out.println("Calculated cost is
"+current_cost);
                                        if(current_cost < mincost){
                                              mincost = current_cost;
                                              mincost_switch = switchID;
                                              if(counter_in > 0)
                                              nearestPrevMB = mb_copys[counter_in-1];
                                        }
                                  }
                                  switchID++;
                            }
                            if(mincost < Integer.MAX_VALUE){ // otherwise all switches
are exhausted
                            //store closest previous middlebox for future shortest path
computation

                            if(counter_in > 0)
                                  memoization_prev_closestMB[mincost_switch-
network.Num_Servers] = nearestPrevMB;

                            // Place the replica copy in the switch. TO DO : Move lines
112 to 116 to a new function
                            avg_cost += (double)mincost;
                            network.devices[mincost_switch].MB.add(counter_in+1);
                            network.devices[mincost_switch].capacity++;
                            ArrayList<Integer> switch_list =
network.MB_Lookup.get(counter_in+1);
                            switch_list.add(mincost_switch);
                            network.MB_Lookup.put(counter_in+1,switch_list );
                            System.out.println("Middlebox "+(counter_in+1)+" replicated
on "+mincost_switch+ " which incurs least cost of "+mincost);
                            }
                      }
                      if (counter+1 == (network.Num_AllSwitches/mb_types))
```

```
                    System.out.println("Avg cost flow from CNMBF:
"+(avg_cost/((counter+1)*mb_types)));
                }


        }

        /*Method Name : findMinCostPreviousMB(int[] prev_mb_positions,int switchID)
         * Purpose : Figure out the switch id that hosts the preceding middlebox type
which minimizes cost flow
         *                  for eg., mb1 if we are currently replicating mb2
         *                  This switch should incur minimum cost between mb1 and mb2
         *
         */
        public static int findMinCostPreviousMB(int[] prev_mb_positions,int switchID){
                int mincost=Integer.MAX_VALUE;

                int minswitch = prev_mb_positions[0];
                for(int counter=0; counter < prev_mb_positions.length; counter++){
                        if (network.cost[prev_mb_positions[counter]][switchID] <
mincost){
                                mincost =
network.cost[prev_mb_positions[counter]][switchID];
                                minswitch = prev_mb_positions[counter];
                        }
                }
                return minswitch;
        }

        /* Method  Name : printMBs()
         * Purpose : Print all middlebox types and their hosts.
         *
         */
        public static void printMBs(){
                Iterator<Entry<Integer, ArrayList<Integer>>> it =
network.MB_Lookup.entrySet().iterator();
                System.out.println("The Middleboxes are: ");
                while(it.hasNext()){
                        Map.Entry<Integer, ArrayList<Integer>> pair =
(Map.Entry)it.next();
                        System.out.print("MiddleBox "+pair.getKey()+" is now available on
following switches: ");
                        for(int i=0;i<pair.getValue().size();i++)
                                System.out.print(pair.getValue().get(i)+" ");
                        System.out.println();
                }
        }
        /* Method  Name : printVMss()
         * Purpose : Print all Vms and their hosts.
         *
         */

        public static void printVMs(){
                Iterator it = network.VM_Lookup.entrySet().iterator();
                while(it.hasNext()){
                        Map.Entry<Integer, Integer> pair = (Map.Entry)it.next();
                        System.out.print("VM "+pair.getKey()+" is now available on switch:
"+pair.getValue());

                        System.out.println();
                }
        }
```

```
	/* Method  Name : printVMPairss()
	 * Purpose : Print all the paired-up VMs
	 *
	 */

	public static void printVMPairs(){
		System.out.println("The paired up VMs are: ");
		for(int loop=0;loop<network.VM_V_Pairs.length;loop++){
			System.out.println("("+network.VM_V_Pairs[loop][0]+","+
network.VM_V_Pairs[loop][1]+")");
		}
	}
	public static void replicate_ServiceChain(int mb_types, ArrayList<Integer>
traffic_group,int replication_counter, int current_count){

		int[] original_mbs = new int[mb_types];
		avg_cost=0;
		//Calculate Total cost without replication
		@SuppressWarnings("rawtypes")
		Iterator it = network.MB_Lookup.entrySet().iterator();
		while(it.hasNext()){
			Map.Entry<Integer,ArrayList<Integer>> pair = (Map.Entry)it.next();
			original_mbs[pair.getKey()-1] = pair.getValue().get(0); //
original mb is stored in 0th index
		}
		int originalCost =
network.calculateTrafficFlowCost(original_mbs,traffic_group);
		System.out.println("Before replication cost "+originalCost);


		/*
		 * to be replicated = mb type's number
		 * array of currently considered mbs = current_mbs : create place-holder
for to_be replicated instance. rest are from previous/original service chain
		 * for mbs other than mb1 , previous mb instances are  from same service
chain and successive mb instances are from previous/original service chain
		 */

		int[] current_mbs = Arrays.copyOf(original_mbs, mb_types); // current_mbs
hold the mb instances of the lowest cost service chain everytime
		int to_be_replicated =0; //starting with mb type 1
		boolean isreplicable = true;
		int previous_chain_cost = originalCost;
		int current_chain_cost=0;
		int switchID = network.Num_Servers;
		int current_replication_count = 0;
		while(isreplicable){
		// The service chain is replicated as a set or not replicated at all
			while(to_be_replicated < mb_types){

					int mincost = Integer.MAX_VALUE;
					int mincost_switch=-1;

					while(switchID < network.Num_AllDevices) {

							if(network.devices[switchID].capacity <
Devices.Switch_Capacity){
								//
System.out.println("Considering "+switchID + " with capacity
"+network.devices[switchID].capacity);

	current_mbs[to_be_replicated]=switchID;
```

```
                                                int newCost =
network.calculateTrafficFlowCost(current_mbs,traffic_group);
                                                if (newCost < mincost){
                                                      mincost=newCost;
                                                      mincost_switch = switchID;
                                                }

                                          }
                                          switchID++;

                              }
                              if (mincost_switch!=-1){
            //                      System.out.println("Trying to replicate
"+to_be_replicated+" on "+mincost_switch);
                                    current_mbs[to_be_replicated]= mincost_switch;
//retain the middlebox on switch that incurred lowest cost of all for this service
chain
                                    network.devices[mincost_switch].capacity++;
                              }
                              to_be_replicated++; //move to next middlebox type
                              if (to_be_replicated == mb_types) //when the replica
of the last middlebox type is successfully placed
                                    current_chain_cost = mincost;
                              switchID = network.Num_Servers; // reset to switch 1
                  }

                  // if new cost resulted from new service chain  <
previous/original service chain cost , isreplicable remains true.Otherwise set it to
false
                  if (current_chain_cost <= originalCost){
                        previous_chain_cost = current_chain_cost;
                        avg_cost += (double) current_chain_cost;
                        to_be_replicated=0;
                        current_chain_cost=0;
                        switchID = network.Num_Servers;
                  //    System.out.println("New Replication Of Service Chain with
cost "+previous_chain_cost);
                  //
      System.out.println("****************************************************************
*********************");
                        // write MBlookup hash table with new instances and
increase capacity of the corresponding switches
                        for(int mb=0;mb<mb_types;mb++){
                              int deviceID = current_mbs[mb];
                        //    System.out.print(deviceID+" ");
                              network.devices[deviceID].MB.add(mb+1);
                              network.devices[deviceID].capacity++;
                              ArrayList<Integer> switch_list =
network.MB_Lookup.get(mb+1);
                              switch_list.add(deviceID);
                              network.MB_Lookup.put(mb+1,switch_list );
                        }
                        ArrayList<Integer> intList = new ArrayList<Integer>();
                        for (int index = 0; index < current_mbs.length; index++)
                        {
                            intList.add(current_mbs[index]);
                        }
                        serviceChains.put(current_count, intList);
                        current_count++;
                  //
      System.out.println("****************************************************************
*********************");
```

```
                    current_mbs=original_mbs; //comment this out if previous
chain cost is considered
                    current_replication_count++;


                }
                else{
                //     System.out.println("Replication failed afer "+
current_replication_count+" replications !!!!!!!!!!!");
                    //System.out.println("There could be no more replications
of service chain with lowest cost than existing service chains");
                //
       System.out.println("*****************************************************
**********************");
                    isreplicable=false;
                }
                if(replication_counter!=-1)
                    if (current_replication_count == replication_counter)
                        isreplicable = false;
            //     System.out.println("Average cost flow
"+avg_cost/current_replication_count);

            }

        }
        public static void  createServiceChainPreference(){
            int totalBestCost=0;
            int totalWorstCost=0;
            for(int vm=0;vm<network.VM_V_Pairs.length;vm++){
                int mincost=Integer.MAX_VALUE;
                int maxCost = Integer.MIN_VALUE;
                if (serviceChains.size()== 0){
                    int[] original_mbs = new int[network.MB_Lookup.size()];
                    //Calculate Total cost without replication
                    @SuppressWarnings("rawtypes")
                    Iterator it = network.MB_Lookup.entrySet().iterator();
                    while(it.hasNext()){
                        Map.Entry<Integer,ArrayList<Integer>> pair =
(Map.Entry)it.next();
                        original_mbs[pair.getKey()-1] =
pair.getValue().get(0); // original mb is stored in 0th index
                    }
                    int originalCost =
network.calculateTrafficFlowCost(original_mbs,null);
                    System.out.println("Avg cost flow  from TVAR:
"+originalCost);
                    return;
                }
                Iterator it=serviceChains.entrySet().iterator();
                while(it.hasNext()){
                    Map.Entry sc=(Map.Entry)it.next();
                    int[] mbs= new
int[((ArrayList<Integer>)sc.getValue()).size()];
                    int i=0;
                    for(int sw:(ArrayList<Integer>)sc.getValue() )
                        mbs[i++]=sw;
                    int cost=network.calculateVMPairTrafficCost(vm,mbs);
                    if (cost<mincost)
                        mincost=cost;
                    if(cost>maxCost)
                        maxCost=cost;
                }
                totalBestCost+=mincost;
```

```
                    totalWorstCost+=maxCost;
                }
                System.out.println("Average Best cost is 
"+totalBestCost/network.VM_V_Pairs.length);
                System.out.println("Average Worst cost is 
"+totalWorstCost/network.VM_V_Pairs.length);
                System.out.println("Avg cost flow  from TVAR: 
"+(totalBestCost+totalWorstCost)/2);

        }
/* Method Name : traffic_aware_replication(mb_types,VMPairs)
 * Purpose : Gets input on number of middlebox types and number of VMpairs in the fat-
tree topology and performs replication of mbs
 * Details : 1. With total possible replications = floor(total num of switches/ total 
number of middlebox types)
 *                      2. Calculate possible replications for {very frequent, frequent, 
medium, less} traffic groups. Their frequency distribution is obtained from 
frequencyMapper()
 *                      3. Number of edge_switch : (Num_ports ^ 2)/2
 *                      4. Number of aggregate_sw: (Num_ports ^ 2)/2
 *                      5.
 *
 */
        public static void traffic_aware_replication(int mb_types, int VMPairs){
                HashMap<Integer,ArrayList<Integer>> frequencyMap = 
network.frequencyMapper(); // this hashmap holds the traffic_groupID->vmpairs array 
belonging to the traffic group
                ArrayList<Integer> replicationDistribution = new ArrayList<Integer>(); // 
this array holds the number of possible replications for traffic groups
                int total_replication = network.Num_AllSwitches/mb_types;
                int replication_for_very_frequent = (int)
(((double)frequencyMap.get(0).size())/VMPairs * total_replication);
                int replication_for_frequent = replication_for_very_frequent + (int)
(((double)frequencyMap.get(1).size())/VMPairs * total_replication);
                int replication_for_medium = replication_for_frequent + (int)
(((double)frequencyMap.get(2).size())/VMPairs * total_replication);
                int replication_for_less = replication_for_medium + (int)
(((double)frequencyMap.get(3).size())/VMPairs*total_replication);
                replicationDistribution.add(replication_for_very_frequent);
                replicationDistribution.add(replication_for_frequent);
                replicationDistribution.add(replication_for_medium);
                replicationDistribution.add(replication_for_less);
                int replication_pointer;
                avg_cost=0;
                ArrayList<Integer> traffic_group;
                int current_replication=0;
                serviceChains = new HashMap<Integer,ArrayList<Integer>>();
                for(int i=0; i < 4; i++){
                        replication_pointer = replicationDistribution.get(i);
                        traffic_group = frequencyMap.get(i);
                    //
        System.out.println("##############################################################
######################");
                    //      System.out.println("Running replication algorithm for 
traffic group  "+i);
                        replicate_ServiceChain(mb_types, traffic_group, 
replication_pointer, current_replication);
                        current_replication += replication_pointer;
                        printMBs();
                    //
        System.out.println("##############################################################
######################");
```

```
            }
            createServiceChainPreference();

        }
        public static void main(String[] args){
            try{
             network = new FatTreeConstruction();
             network.createFatTree(4);
             //Check createFatTree() method
            System.out.println("Total Devices: "+network.Num_AllDevices+" total pods:
"+network.podSwitches.size());
            Integer[][] cost = network.calculateCost(network.Num_AllDevices);
        /*  for(int i=0;i<cost.length;i++){
                for(int j=0;j<cost[0].length;j++)
                    System.out.println(i+" "+j+" "+cost[i][j]);
            }*/
            int[] vm_pairs = new int[]{100,200,300,400,500};
            int[] mbs=new int[] {3,5,7};
            for(int mb : mbs){
                for(int vp : vm_pairs){

        System.out.println("###########################################################
#######################");
                    System.out.println("Middlebox Types: "+mb+", Virtual
Machine Pairs "+vp);

        System.out.println("###########################################################
#######################");

            network.randomDistributeVM(vp);
                    printVMs();
            network.randomDistributeMB(mb);
            printMBs();
            network.randomPairVM(vp);
            printVMPairs();

            /*
             * ALGORITHM 3: Traffic Aware replication
             */
            //network.resetFatTree(5);

        System.out.println("###########################################################
#######################");
            System.out.println("Starting Traffic-Aware Algorithm ");

        System.out.println("###########################################################
#######################");
            long start = System.currentTimeMillis();
            traffic_aware_replication(mb,vp);
            long end = System.currentTimeMillis();
            System.out.println("Execution time "+(end-start));
            printMBs();
            /*
             * ALGORITHM 4: Non-sequential mb replication
             */
            //network.resetFatTree(5);
            //NonSequentialMBReplication.replicate_mb_non_sequential(network,3);
            //printMBs();
            /*
             * ALGORITHM 1:  Random replication
```

```
                */

        System.out.println("###########################################################
#######################");
                System.out.println(" Starting Random Replication Algorithm:");

        System.out.println("###########################################################
#######################");

                network.resetFatTree(mb);
                start= System.currentTimeMillis();
                RandomReplication.randomReplicator(network, mb);
                end = System.currentTimeMillis();
                System.out.println("Execution time "+(end-start));
                printMBs();
                /*
                 * Improved closest next
                 */
                network.resetFatTree(mb);

        System.out.println("###########################################################
#######################");
                System.out.println(" Starting Closest Next MiddelBox First Algorithm:");

        System.out.println("###########################################################
#######################");

                start= System.currentTimeMillis();
                System.out.println("Improved Closest next MB Algorithm:");
                improvedClosestNextMB CNB= new improvedClosestNextMB(network);
                CNB.closestNextReplication(mb);
                end = System.currentTimeMillis();
                System.out.println("Execution time "+(end-start));
                printMBs();
                network.resetFatTree(mb);


        System.out.println("###########################################################
#######################");
                System.out.println(" Starting Exhaustive MB Replication Algorithm:");

        System.out.println("###########################################################
#######################");
                start= System.currentTimeMillis();
                ExhaustiveMbReplication.replicateMB(network, mb);
                end = System.currentTimeMillis();
                System.out.println("Execution time "+(end-start));
                printMBs();
                network.resetFatTree(mb);
                }
                }
                }
                catch(Exception e){
                        System.out.println(e.getMessage());
                }
        }


}
```