

2017-01-01

# Multi-layered virtual transport network design and management (PhD Thesis)

---

Wang, Yuefeng. "Multi-Layered Virtual Transport Network Design and Management (PhD Thesis)", Technical Report BUCS-TR-2017-001, Department of Computer Science, Boston University, January 1, 2017. [Available from: <http://hdl.handle.net/2144/21087>]

<https://hdl.handle.net/2144/21087>

*"Downloaded from OpenBU. Boston University's institutional repository."*

BOSTON UNIVERSITY  
GRADUATE SCHOOL OF ARTS AND SCIENCES

Dissertation

**MULTI-LAYER VIRTUAL TRANSPORT NETWORK DESIGN AND  
MANAGEMENT**

by

**YUEFENG WANG**

Master of Science, University of Windsor, 2010  
Bachelor of Engineering, Shandong University, 2008

Submitted in partial fulfillment of the  
requirements for the degree of  
Doctor of Philosophy

2017

© Copyright by  
YUEFENG WANG  
2017

Approved by

First Reader

---

Abraham Matta, PhD  
Professor of Computer Science

Second Reader

---

Azer Bestavros, PhD  
Professor of Computer Science

Third Reader

---

Richard West, PhD  
Associate Professor of Computer Science

## Acknowledgments

First, I would like to thank my parents and my wife for their endless love and encouragement. They always give me great support whenever I face any difficulty or problem. Second, I would like to thank my PhD advisor, Prof. Abraham Matta, for his valuable guidance and advice as well as being a great role model for me. I could not finish my PhD without Abraham's help. Third, I would like to thank Prof. Azer Bestavros, Prof. Richard West, Prof. Jonathan Appavoo, and Prof. Hongwei Xi for serving on my committee and for their valuable feedback. Fourth, I would like to thank Prof. John Day and Prof. Lou Chitkushev, and it has been a great pleasure and experience to work with them on the RINA project. Last, I would like to thank all my friends at BU, with whom I was very happy and fortunate to share the PhD journey, and also I would like to thank faculty and staff members in our CS department for their help.

The PhD study is a long journey and one of a kind experience, and I really enjoyed this journey. Most importantly I am very happy and lucky to earn my PhD in the end. I am really grateful to everyone who helped and supported me during this great journey.

# MULTI-LAYER VIRTUAL TRANSPORT NETWORK DESIGN AND MANAGEMENT

YUEFENG WANG

Boston University, Graduate School of Arts and Sciences, 2017

Major Professor: Abraham Matta, Professor of Computer Science

## ABSTRACT

Nowadays there is an increasing need for a general paradigm that can simplify network management and further enable network innovations. Software Defined Networking (SDN) is an efficient way to make the network programmable and reduce management complexity, however it is plagued with limitations inherited from the legacy Internet (TCP/IP) architecture. On the other hand, service overlay networks and virtual networks are widely used to overcome deficiencies of the Internet. However, most overlay/virtual networks are single-layered and lack dynamic scope management. Furthermore, how to solve the joint problem of designing and mapping the overlay/virtual network requests for better application and network performance remains an understudied area.

In this thesis, in response to limitations of current SDN management solutions and of the traditional single-layer overlay/virtual network design, we propose a recursive approach to enterprise network management, where network management is done through managing various Virtual Transport Networks (VTNs) over different scopes (*i.e.*, regions of operation). Different from the traditional overlay/virtual network model which mainly focuses on routing/tunneling, our VTN approach provides communication service with explicit Quality-of-Service (QoS) support for applications via transport flows, *i.e.*, it involves all mechanisms (*e.g.*, addressing, routing, error and flow control, resource allocation) needed to meet application requirements. Our approach inherently provides a multi-layer solution for overlay/virtual network design.

The contributions of this thesis are threefold: (1) we propose a novel VTN-based man-

agement approach to enterprise network management; (2) we develop a framework for multi-layer VTN design and instantiate it to meet specific application and network goals; and (3) we design and prototype a VTN-based management architecture. Our simulation and experimental results demonstrate the flexibility of our VTN-based management approach and its performance advantages.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Related Work</b>	<b>4</b>
2.1	Network Programmability . . . . .	4
2.1.1	Management Architecture for SDN Networks . . . . .	5
2.1.2	Problems with SDN Management Layer . . . . .	7
2.2	Overlay/Virtual Network Design . . . . .	10
2.3	Transport Network . . . . .	11
<b>3</b>	<b>Virtual Transport Network (VTN) and VTN-based Network Management</b>	<b>13</b>
3.1	Virtual Transport Network (VTN) and Transport Process . . . . .	14
3.2	Advantages of VTN-based Network Management . . . . .	16
3.2.1	Switch Memory Usage . . . . .	16
3.2.2	Resource Utilization . . . . .	17
3.2.3	Routing Overhead . . . . .	19
3.2.4	Transport Overhead . . . . .	24
3.3	Instantiations of the Multi-layer Management Framework . . . . .	26
3.3.1	Routing . . . . .	26
3.3.2	Transport . . . . .	27
<b>4</b>	<b>Multi-layer VTN Design Problem and Algorithms</b>	<b>28</b>
4.1	Problem Definition . . . . .	28
4.1.1	Path Selection Stage . . . . .	28

4.1.2	VTN Design Stage . . . . .	29
4.1.3	A Simple Example . . . . .	29
4.2	Problem Properties . . . . .	31
4.3	Case Study (1) : Constrained Path Selection with Unconstrained VTN Design	32
4.3.1	Constrained Path Selection . . . . .	32
4.3.2	Unconstrained VTN Design . . . . .	34
4.3.3	Performance Evaluation . . . . .	36
4.4	Case Study (2): Unconstrained Path Selection with Constrained VTN Design	39
4.4.1	Unconstrained Path Selection . . . . .	40
4.4.2	Constrained VTN Design . . . . .	40
4.4.3	VTN Packing Problem . . . . .	44
4.4.4	General Algorithm for Constrained VTN Design . . . . .	47
4.4.5	Performance Evaluation . . . . .	49
<b>5</b>	<b>Design of VTN-based Management Architecture</b>	<b>59</b>
5.1	Management Components . . . . .	59
5.1.1	VTN Manager and VTN Manager Agent . . . . .	59
5.1.2	VTN Allocator and VTN Allocator Agent . . . . .	60
5.1.3	VTN Resource Manager (VRM) . . . . .	61
5.1.4	Walk-through of Transport Flow Allocation . . . . .	61
5.2	VTN Formation Protocol . . . . .	62
5.2.1	Objects Exchanged in the Protocol . . . . .	63
5.2.2	Protocol Details . . . . .	64
5.3	Network API . . . . .	65
5.3.1	Management-level API . . . . .	65
5.3.2	User-level API . . . . .	66
5.4	Node Components . . . . .	67
5.4.1	Node Process . . . . .	67

5.4.2	Application Process . . . . .	68
5.4.3	Shim Layer Process . . . . .	70
5.5	Transport Process Components . . . . .	70
5.5.1	Data Transfer Application Entity . . . . .	71
5.5.2	Management Application Entity . . . . .	72
5.5.3	Transport Process (TP) API . . . . .	72
<b>6</b>	<b>Implementation of VTN-based Management Architecture</b>	<b>73</b>
6.1	Key Classes . . . . .	74
6.1.1	Node Process . . . . .	74
6.1.2	Application Process . . . . .	74
6.1.3	Transport Process . . . . .	74
6.1.4	VTN Manager and VTN Manager Agent . . . . .	75
6.1.5	VTN Allocator and VTN Allocator Agent . . . . .	76
6.2	Supporting Classes . . . . .	77
6.2.1	VTN Resource Manager (VRM) . . . . .	77
6.2.2	RIB and RIB Daemon . . . . .	78
6.2.3	Routing Daemon . . . . .	79
6.2.4	Shim Layer Process . . . . .	81
6.3	Configurations Files . . . . .	82
6.3.1	Node Process Configurations . . . . .	82
6.3.2	Application Process Configurations . . . . .	83
6.3.3	Transport (IPC) Process Configurations . . . . .	84
6.3.4	Other Configurations . . . . .	87
<b>7</b>	<b>Evaluation of Architecture Implementation</b>	<b>88</b>
7.1	Routing Performance . . . . .	88
7.1.1	Experiment Design . . . . .	89
7.1.2	Experiments over GENI . . . . .	90

7.2	Video Unicast Performance . . . . .	92
7.2.1	VTN Policies for Video Streaming . . . . .	93
7.2.2	Experiments over GENI . . . . .	94
7.3	Video Multicast Performance . . . . .	99
7.3.1	VTN-based Multicast Solutions . . . . .	102
7.3.2	Experiments over GENI . . . . .	104
<b>8</b>	<b>Conclusions and Future Work</b>	<b>109</b>
8.1	Conclusions . . . . .	109
8.2	Future Work . . . . .	110
<b>A</b>	<b>Proof of Equation (3.8)</b>	<b>112</b>
	<b>Bibliography</b>	<b>115</b>
	<b>Curriculum Vitae</b>	<b>121</b>

## List of Tables

4.1	Path selection formulated as an ILP problem. . . . .	33
5.1	Three sets of management-level APIs in Java. . . . .	65
5.2	Two sets of user-level APIs in Java. . . . .	66
5.3	RIB API in Java. . . . .	70
5.4	Transport Process (TP) API in Java. . . . .	72

## List of Figures

2.1	A general network management architecture for SDN networks. . . . .	5
3.1	Two levels of VTNs, and <i>VTN 3</i> spans a larger scope. Each process inside a VTN is a transport process. . . . .	15
3.2	$n$ application flows going through 4 switches (S1, S2, S3 and S4). . . . .	17
3.3	$n$ non-aggregatable application flows can be aggregated into one flow by VTN. . . . .	17
3.4	Effective per-flow bandwidth requirements for different QoS. . . . .	19
3.5	A simple network with 9 nodes. . . . .	19
3.6	A multi-layer VTN design, where there are three transport processes (colored with the same color) on $A$ , $C$ , $G$ and $E$ , respectively, and there is one transport process on $B$ , $D$ , $F$ , $H$ , and $I$ , respectively. . . . .	21
3.7	Another possible multi-layer design for the same network, where there are 3 transport processes on $A$ and $E$ , respectively, and there is 1 transport process on $B$ , $C$ , $D$ , $F$ , $G$ , $H$ and $I$ , respectively. . . . .	23
3.8	(a) The TCP connection between applications on $A$ and $E$ involves 4 hops under single-layer design. (b) The same flow between applications on $A$ and $E$ can be supported by multi-layer VTNs. . . . .	25
3.9	Average number of transmissions for one successful packet delivery for flows of different length. . . . .	26
4.1	A simple network with 6 nodes. . . . .	29
4.2	The output of the path selection stage includes 3 aggregated flows. . . . .	30
4.3	3 VTNs are designed in the VTN design stage. . . . .	30
4.4	Acceptance ratio of flow requests, where $1 - \epsilon = 90\%$ . . . . .	37

4.5	Average number of memory entries needed for each flow served. . . . .	38
4.6	Number of new VTNs created per 1000 flows served for different values of $M$ . . . . .	39
4.7	The application flow between $A$ on $Host1$ and $B$ on $Host8$ has a path of 7 hops in the level-0 graph. After the initial design step, we have 2 extra levels of VTNs when $max\_dia = 3$ . Note that our multi-layer design has a level-0 VTN with only two transport processes for each physical wire. . . . .	41
4.8	3 VTNs of the level-1 after the initial design step. . . . .	44
4.9	One solution of the VTN packing problem, where $VTN2$ and $VTN3$ from Figure 4.8 are merged into one VTN. . . . .	45
4.10	For $max\_dia = 2$ and $max\_num = 5$ , $VTN1$ and $VTN2$ cannot be merged as the diameter of the merged VTN ( <i>i.e.</i> , 3) exceeds the $max\_dia$ . . . . .	47
4.11	For $max\_dia = 3$ and $max\_num = 5$ , $VTN1$ and $VTN2$ can be merged into a single VTN. . . . .	47
4.12	Comparison of total routing overhead (mean with 90% confidence interval) between multi-layer (upper-bound) and single-layer design for 3 different communication patterns. . . . .	51
4.13	Comparison of average number of transport processes per node (mean with 90% confidence interval) between multi-layer and single-layered design for 3 different communication patterns. . . . .	51
4.14	Savings (mean with 90% confidence interval) in total routing overhead compared to single-layer approach for the skewed distribution, where each user node randomly picks a hotspot node. . . . .	53
4.15	Average number of transport processes per node (mean with 90% confidence interval) compared to single-layer approach for the skewed distribution, where each user node randomly picks a hotspot node. Note that the number for single-layer design is 3. . . . .	53

4.16	Savings (mean with 90% confidence interval) in total routing overhead compared to single-layer approach for the skewed distribution with preference, where each user node picks the closest hotspot node. . . . .	54
4.17	Average number of transport processes per node (mean with 90% confidence interval) compared to single-layer approach for the skewed with preference, where each user node picks the closest hotspot node. Note that the number for single-layer design is 3. . . . .	54
4.18	Total routing overhead (mean of 10 runs) for different values of <i>max_num</i> when <i>max_dia</i> = 5, while the cost for single-layer design is 2500. . . . .	55
4.19	Average number of transport processes per node (mean of 10 runs) for different values of <i>max_num</i> when <i>max_dia</i> = 5, while the number for single-layer design is 3. . . . .	56
4.20	Total routing overhead (mean of 10 runs) for different values of <i>max_dia</i> when <i>max_num</i> = 12, while the cost for single-layer design is 2500. . . . .	56
4.21	Average number of transport processes per node (mean of 10 runs) for different values of <i>max_dia</i> when <i>max_num</i> = 12, while the number for single-layer design is 3. . . . .	57
4.22	Average number of transmissions for one successful packet delivery for different values of <i>max_dia</i> . . . . .	58
5.1	VTN Manager and its agents for a single VTN. . . . .	60
5.2	VTN Allocator and its agents for an enterprise network. . . . .	60
5.3	(a) An enterprise network consists of three nodes ( <i>Node 1</i> , <i>Node 2</i> and <i>Node 3</i> ), and a centralized VTN Allocator. (b) A new VTN ( <i>VTN 3</i> ) is formed to support the flow between <i>App 1</i> and <i>App 2</i> . VAA of each node is not shown in (b). . . . .	62
5.4	Components of a node process. . . . .	67
5.5	Components of an application process. . . . .	69

5.6	Components of a Transport Process. . . . .	71
7.1	GENI resources with 8 nodes (VMs) shown in Jacks. . . . .	89
7.2	A single-layer design with one single VTN spanning all 8 nodes. . . . .	89
7.3	A multi-layer design with 5 VTNs of two levels. . . . .	90
7.4	Total number of LSU messages processed per second by all 8 nodes during steady state for 4 different update frequencies. . . . .	91
7.5	Total size of LSU messages processed per second by all 8 nodes during steady state for 4 different update frequencies. . . . .	92
7.6	Video clients (VLC players) are connected to the video streaming server (Live555 streaming server) through RTSP proxies over a VTN-based network. . . . .	93
7.7	Each node process is running on a GENI VM. Nodes in different aggregates are connected via GRE tunnels, and nodes in the same aggregate are connected via VLANs. . . . .	94
7.8	Video client proxy (on Node 9) and video server proxy (on Node 2) communicate through a level-2 VTN (VTN 4), which is built on top of three level-1 VTNs (VTN 1, VTN 2, and VTN 3). . . . .	95
7.9	GENI resources from four aggregates shown in Flack. . . . .	97
7.10	Path jitter of <i>path 1</i> (the least-hop path) is larger than <i>path 2</i> (the least-jitter path), where path jitter is calculated as the sum of jitter on links along the path (collected by the routing task of Process 20). . . . .	98
7.11	Measured instantaneous jitter for video packets from the video server proxy to the client proxy when VTN 4 uses hop or jitter as link cost. . . . .	99
7.12	Video clients (VLC players) are connected to the RTP video server through RTP proxies over a VTN-based network. . . . .	100
7.13	Video server providing a live video streaming service is running in Network A. One client is in Network C, and one is in Network D. . . . .	101

7.14	Video streaming through unicast connections, where same video traffic is delivered twice over VTN 1 consuming unnecessary network bandwidth. . . . .	101
7.15	Video multicast through an RTP multicast video server. . . . .	102
7.16	Video multicast through multicast service provided by the VTN. . . . .	103
7.17	GENI resources from four InstaGENI aggregates shown in Jacks. . . . .	105
7.18	Comparison of bandwidth usage over VTN1: unicast vs. multicast. . . . .	105
7.19	GENI resources from five InstaGENI aggregates shown in Jacks. . . . .	106
7.20	Video observed when the video multicast server is placed on VM N4 in the NYSERNet aggregate resulting in a path with less jitter. . . . .	107
7.21	Video observed when the video multicast server is placed on VM N2 in the Chicago aggregate resulting in a path with more jitter. . . . .	107
A.1	An absorbing Markov chain for delivering one packet over a TCP connection of $H$ hops, where each circle denotes a possible state. Assume loss rate on each link is $P$ . . . . .	112

## List of Abbreviations

QoS	.....	Quality of Service
SDN	.....	Software Defined Networking
NFV	.....	Network Function Virtualization
RINA	.....	Recursive InterNetwork Architecture
TCP/IP	.....	Transport Control Protocol/Internet Protocol
DNS	.....	Domain Name System
GENI	.....	Global Environment for Network Innovations
NP	.....	Non-deterministic Polynomial
VTN	.....	Virtual Transport Network
TP	.....	Transport Process
IPC	.....	Inter-Process Communication
RIB	.....	Resource Information Base
VA	.....	VTN Allocator
VAA	.....	VTN Allocator Agent
VRM	.....	VTN Resource Manager
DAF	.....	Distributed Application Facility
CDAP	.....	Common Distributed Application Protocol
EFCP	.....	Error and Flow Control Protocol
DTP	.....	Data Transport Protocol
DTCP	.....	Data Transport Control Protocol

## List of Symbols

$G_{n-1} = \langle V, E^{n-1} \rangle$	level- $(n - 1)$ network topology
$V = \{v_i\}$	set of nodes
$v_i$	node $v_i$
$E^{n-1} = \{e_{st}\}$	set of links between nodes
$e_{st}$	(virtual) link between nodes $s$ and $t$
$C_{st}$	capacity (both directions) of link $e_{st}$
$F = \{f_n\}$	set of all flow requests
$N =  F $	total number of flow requests
$b(f_n)$	bandwidth required for flow $n$
$P(f_n) = \{p_k^n\}$	set of possible paths for flow $n$
$ P(f_n) $	number of possible paths for flow $n$
$p(f_n)$	path selected for flow $n$
$l(p_k^n)$	length of path $k$ for flow $n$
$F' = \{f_m\}$	set of accepted flow requests after path selection
$ F' $	number of accepted flow requests
$p(f_m)$	path selected to serve flow $f_m$
$ p(f_m) $	length of the path selected to serve flow $f_m$
$D_z = \langle V_z, E_z \rangle$	VTN $z$
$V_z = \{v_s^z\}$	set of transport processes in VTN $z$
$S_z = \{v_i\}$	set of nodes having processes belonging to VTN $z$
$v_s^z$	transport process $s$ in VTN $z$ (running on some node $v_i$ )

$E_z = \{e_{st}^z\}$	set of (virtual) links in VTN $z$
$e_{st}^z$	(virtual) link between processes $s$ and $t$ in VTN $z$
$D = \{D_z\}$	set of all existing VTNs
$ D $	number of existing VTNs
$D_{level(n)}$	set of VTNs of level- $n$
$D^{n-1} = \{D_z\}$	set of existing VTNs of and below level- $(n - 1)$
$ D^{n-1} $	number of existing VTNs of and below level- $(n - 1)$
$max\_num$ (or $M$ )	max number of transport processes allowed in a VTN
$max\_dia$	length of longest shortest path (in hops) allowed in a VTN

## Chapter 1

### Introduction

Traditionally network management is a complicated and error-prone process that involves low-level and vendor-specific configurations of physical devices. Nowadays computer networks have become increasingly complex and difficult to manage, and new cloud-based service models [49] have become the norm in networking economics, *e.g.*, Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS). These trends increase the need for a general management paradigm to simplify network management and further enable network innovations. Making the network programmable is an efficient way to address the complexity of network management by providing high-level network abstractions and hiding low-level details of physical devices.

A lot of research has been done to address the complexity of network management by providing high-level network abstractions and hiding low-level details of physical devices. Among those efforts, making the network programmable is an efficient way to that end. Active Networking [68] pioneered the research in programmable networking, but it failed to gain popularity due to lack of an immediately compelling problem (or a killer application) [22]. Recently Software Defined Networking (SDN) [54] has drawn considerable attention due to the popularity of OpenFlow [46], a protocol that allows the configuration of switches without exposing their internal details.

In SDN, the network is considered to have two components: (1) control plane which decides on how to handle data traffic, and (2) data plane which forwards data traffic to its destination. SDN focuses on programming the control plane through a network management layer and has been widely deployed in enterprise and data center networks [46, 33].

An OpenFlow-based SDN management layer provides a high-level interface, and network managers can easily manage the network without dealing with the complexity of low-level network devices. However SDN management layers are plagued with limitations inherited from the TCP/IP architecture [78], such as static management, one-size-fits-all structure, and ad-hoc mechanisms with no common management framework.

On the other hand, service overlay networks [42, 24] and network virtualization [7, 14] allow multiple virtual networks to run over a common physical network infrastructure. This can improve resource utilization through network consolidation and provide isolation for security purposes or for developing and testing new network features. However most overlay/virtual networks are used only for routing/tunneling purposes, and do not support transport flows (involving all mechanisms such as addressing, routing, error and flow control, resource allocation and explicit QoS support), which is very important for network resource allocation and utilization. What's more, most overlay/virtual networks are single-layered and lack dynamic scope management, and the joint problem of designing and mapping the overlay/virtual network requests, which also can help achieve better performance, remains an understudied area.

In response to these limitations, we propose a recursive approach to enterprise network management. In our approach, network management is done through managing various Virtual Transport Networks (VTNs), and it is inspired by and built atop a new network architecture, RINA [15, 16], which aims to solve current TCP/IP limitations. Different from the traditional overlay/virtual network model, which mainly focuses on routing/tunneling, a VTN provides communication service with explicit QoS support for applications via transport flows, and it includes all mechanisms (*e.g.*, addressing, routing, error and flow control, resource allocation) needed to support such transport flows. The same VTN mechanism can be recursed over different management scopes to provide end-to-end communication services. One of the biggest advantages of VTN is that it enables scoping at the transport level, thus we can either aggregate multiple transport flows into a single transport flow or split a transport flow into multiple transport flows, which enables better

resource utilization in support of various requirements. Furthermore we provide a multi-layer approach to VTN design, which allows dynamic and fine-grained scope management.

In summary the contributions of this thesis are threefold.

- We propose a recursive approach to enterprise network management, where network management is done through managing various VTNs.
- We propose a framework for multi-layer VTN design to satisfy different application and network requirements, which allows for achieving different goals by setting different constraints and objectives for optimization problems.
- We present the design, implementation and evaluation of our VTN-based management architecture, which enables the VTN-based network management of real networks.

The rest of this thesis is organized as follows. In Chapter 2, we discuss work related to this thesis. In Chapter 3, we present the details of VTN as well as VTN-based network management. In Chapter 4, we explain the multi-layer VTN design problem and two case studies. The design, implementation and evaluation of our VTN-based network management architecture are presented in Chapters 5, 6 and 7, respectively. Chapter 8 concludes this thesis with future work.

**Published Papers:** Part of Chapter 2 is published in [78]. Parts of Chapters 3 and 4 are published in [77]. Parts of Chapters 5 and 6 are published in [77, 80, 76, 75]. Part of Chapter 7 is published in [79, 74].

## Chapter 2

# Related Work

### 2.1 Network Programmability

Traditional networks are managed through low-level and vendor-specific configurations of individual network components, which is a very complicated and error-prone process. Nowadays computer networks are becoming increasingly complex and difficult to manage. This increases the need for a general management paradigm that provides common management abstractions, hides the details of the physical infrastructure, and enables flexible network management. Making the network programmable (pioneered by earlier research in Active Networking [68]) leads to such a general paradigm, as programmability simplifies network management and enables network innovations.

Software Defined Networking (SDN) has been proposed to enable programmable networks. In SDN, the network is considered to have two components: (1) control plane, which determines how to handle and forward data traffic, and (2) data plane, which handles and forwards data traffic toward its destination. SDN separates the control plane and data plane, and focuses on programming the control plane through a network management layer<sup>1</sup>. Through a high-level interface provided by the network management layer, network managers can easily manage the network without dealing with the complexity of low-level network details.

In general, the data plane might not only be a forwarding plane that just stores and forwards packets (or discards them) through packet flow (forwarding) table manipulations,

---

<sup>1</sup>We use the terms “management platform”, “management layer” and “control platform” interchangeably.

but it might also include more application-specific data processing capabilities [12][17]. This is similar to the focus of earlier research in Active Networking, where network devices (switches or routers) are expected to perform computation on and modification of packet contents [68]. In this section we focus on the control plane only for the purpose of programming the forwarding of packet flows, *i.e.*, the network management layer for SDN networks.

In the rest of this section, we present the common management architecture for SDN network, and further identify its open issues and weaknesses.

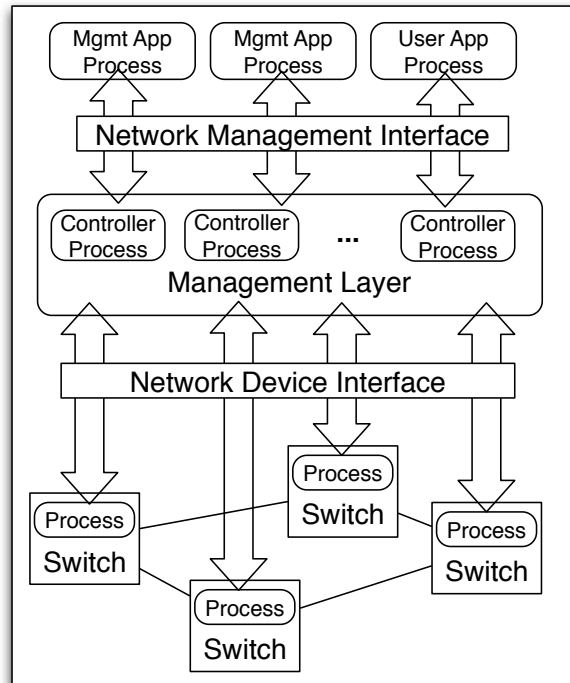


Figure 2.1: A general network management architecture for SDN networks.

### 2.1.1 Management Architecture for SDN Networks

The core of a management architecture for SDN networks is the management layer (such as [28, 41, 63, 72, 23]), as shown in Figure 2.1. A management layer should enable the

monitoring and control of the network. The management layer itself does not manage the network but provides a programmatic interface to management (or user) applications, which in turn manage the network. Examples of management applications include access control, virtual-machine (VM) migration, traffic-aware path selection and path adaptation, and redirecting or dropping suspected attack traffic.

#### 2.1.1.1 Management Architecture Overview

Figure 2.1 shows a general network management architecture for SDN networks. At the bottom are the network devices including switches or routers<sup>2</sup>. There is a process (*switch process*) running on each network device, and this process hides the internal details of the physical device but exposes a *Network Device Interface* (the so-called “Southbound API” [53]). The network device interface provides a standardized way to access the switch processes which operate on the switches. The switch process is responsible for low-level operations on switches such as adding/removing packet flow entries, and configuration of ports and queues. The management layer consists of one or more *controller processes*, which may run on one or more physical servers. Controller processes collaborate to provide the network monitoring and control functionalities. The management layer exposes a *Network Management Interface* (the so-called “Northbound API” [53]) for management (or user) application processes to manage the network.

#### 2.1.1.2 OpenFlow-based SDN networks

In an SDN network, the Network Device Interface can be supported by any mechanism (protocol) that provides communication between the control plane (management layer) and data plane (switch processes). OpenFlow [46] is such a mechanism (protocol) that gives the management layer access to switches and routers. OpenFlow is the first standardized open

---

<sup>2</sup>In this thesis, switches and routers are considered to be the same, and both provide Layer 2 and Layer 3 operations.

protocol that allows network administrators or experimenters to adapt the configuration of switches and routers from different vendors in a uniform way so as to add and remove packet flow state (forwarding) entries.

As OpenFlow can be easily deployed on existing hardware, OpenFlow soon became popular in the research community and industry. OpenFlow enables programming of the hardware without needing vendors to expose the internal details of their devices. OpenFlow is now supported by major vendors, and OpenFlow-enabled switches are commercially available.

OpenFlow is now the most commonly deployed SDN technology and is seen as an enabler of SDN. However, OpenFlow is not the only mechanism to enable SDN and support the Network Device Interface, and any mechanism that could provide communication between the control plane and data plane can be used. Forwarding and Control Element Separation (ForCES) [82] protocol is an example, however it is not adopted by major switch/router vendors. In this section, we focus on OpenFlow-based SDN networks due to its growing popularity, however whether it is OpenFlow or another protocol (*e.g.*, NETCONF [19]) is not relevant due to their same disadvantages of being tied to the TCP/IP architecture.

## **2.1.2 Problems with SDN Management Layer**

### **2.1.2.1 User-level Interface and QoS Support**

There are two types of interface that can be provided by the network management layer: (1) administrator-level interface and (2) user-level interface. An administrator-level interface is provided to the network administrator, who uses this interface to write management applications to monitor and control the network as a whole. This interface is provided by default by all management layers. On the other hand, a user-level interface is provided to network end-users.

End-users write general applications (such as a video conference application, Hadoop-

based [29] or Spark-based [65] application) using this interface to affect the management of their traffic, and as a result, achieve better performance, security or predictable behavior for their applications [23]. To achieve the same goal in an SDN network in the absence of a user-level interface, end-users may either (1) have to out-of-band request service from the network administrator, which is inconvenient and increases the workload on the network administrator, or (2) use a dedicated per-application management controller that runs as the administrator, which makes it hard to combine different application management controllers on the same physical network since decisions from different management controllers may conflict with each other.

However most SDN management layers (such as [28, 41]) only provide the management-level interface, used by network managers to write management applications to monitor and control the network, and they lack a user-level interface. PANE [23] allows users to reserve bandwidth, however other aspects of QoS (Quality of Service) support (including loss rate and delay guarantees) are not supported. QoS support is important because it can not only improve the performance of user applications via guaranteed services but also improve network performance via better resource allocation.

A management layer should provide users with an API that offers predictable network connections as this is crucial for user application performance. However, since existing SDN solutions are tied to the TCP/IP architecture, the rudimentary “best-effort” delivery service of TCP/IP makes it hard for the SDN management layer to support QoS requirements.

### **2.1.2.2 Policy-based Network Management and Layered Scoping**

By *policy-based network management* we mean that network management can be expressed in terms of high-level policies instead of network device configurations, which are low-level and vendor-specific. The network management layer is responsible for translating these high-level policies into low-level and vendor-specific configurations of network devices (switches or routers). Policies are in the form of a set of rules that define a set of

network conditions, responses to these network conditions, and network components that perform these responses [36]. Advantages of policy-based network management include: simplifying device, network and service management, enabling the provision of different services to different users, managing the increasing complexity of programming devices, and supporting business-driven network configurations [66].

One of our contributions of this thesis is defining the concept of layered scopes in network management. A network management layer manages a network over a certain *scope*, where *scope* is a collection of processes running on a subset of nodes (*e.g.*, switches, routers, and hosts) in the network. Members in the same scope follow the same network policies, and collectively provide a particular communication service. New management scopes can be dynamically defined for different purposes (*e.g.*, application performance or network performance).

Layered scoping is important in network management as it enables fine-grained control over the network and better support for policy-based management. However with SDN, scope is flat and only one-level, and cannot be dynamically defined. Essentially SDN is a flat management solution, which lacks levels of management scope, and every component is part of the same and only management scope. This is due to SDN's reliance on the TCP/IP architecture, which notably lacks resource allocation and flow/error control over limited scopes [15]. There is existing SDN work that provides recursive control (*e.g.*, [45, 69]), but it lacks transport-level flow/QoS control over limited scopes and does not support dynamic management of scopes.

In parallel to this thesis's work, there is other work investigating layered and recursive management. For example, [11] investigates how to prioritize flow requests under limited network resources and provide self-adaptation within a scope. [26] shows a layered architecture requires fewer number of addresses compared to the flat IP-based approach. However their layered network structure is static and given, while this thesis looks at the multi-layered management design problem based on performance/cost objectives for application and network.

### 2.1.2.3 Other Problems

Network virtualization allows multiple isolated virtual networks to be built on top of the same physical infrastructure. It can improve resource utilization through network consolidation and provide isolation for security purposes or for developing and testing new network features. Some SDN management layers (such as [63, 18, 61, 35]) support network virtualization, but they mainly focus on routing and access control, and do not consider other mechanisms (such as error and flow control and resource allocation) for transport purpose, which is important for network resource utilization.

There are also other problems with SDN, such as mobility and security, which are also made challenging due to limitations inherent in the TCP/IP architecture.

## 2.2 Overlay/Virtual Network Design

Both service overlay networks [42, 24] and network virtualization [7, 14] allow multiple virtual networks to run over a common physical network infrastructure for better resource utilization. Most overlay networks are implemented in the application layer, and they aim to add new features and fix problems in the Internet, such as resiliency [6], multicast [34], QoS guarantees [67], security [40] and so on. On the other hand, network virtualization enables researchers to experiment easily with new architectures and protocols on virtualized networks [25], and provides end-to-end communication services by connecting computing resources (virtual machines) with virtual links [55].

A lot of work has been done on provisioning overlay/virtual networks over the Internet, such as VLAN, VPN, MPLS, and recent SDN-based virtualization solutions (*e.g.*, [63, 35]). However, most overlay/virtual networks are used only for routing/tunneling purposes, and not for providing scoped transport flows (involving all mechanisms such as error and flow control, resource allocation, explicit QoS support), which would allow better network resource allocation and utilization.

There is existing work on how to design the virtual/overlay network topology. Some

work (*e.g.*, [37, 43, 5]) focuses on how different overlay topologies (*e.g.*, mesh, tree) affect overlay network performance (such as routing) given the location of overlay/virtual nodes. Some other work (*e.g.*, [30]) focuses on where to place overlay/virtual nodes for better performance (such as resiliency) without considering the overlay connectivity. However these approaches are single-layered, *i.e.*, there may be multiple overlay/virtual networks over the same physical infrastructure, but they all belong to the same and single layer (level).

Most overlay/virtual network design approaches consider designing the overlay/virtual network and mapping the design as two separate problems. Typically the problem of designing the overlay/virtual networks is solved by service providers, and the mapping/embedding problem is solved by infrastructure providers. Some work (*e.g.*, [70, 84, 10, 21]) attempts to solve the joint problem of designing and mapping the virtual/overlay network request, and they simultaneously consider where to place the overlay/virtual nodes and how to connect them in the overlay to reduce the cost of building the virtual/overlay network and satisfy different requirements (such as bandwidth, resiliency).

However, it is not well studied about how to dynamically and holistically design, map and eventually form the virtual/overlay network to satisfy different application-specific requirements and reduce network management overhead. Most importantly, most overlay/virtual networks are single-layered and lack dynamic scope management due to being tied to the TCP/IP architecture.

### 2.3 Transport Network

The concept of *transport network* is not new, and a lot of work has been done on how to build such a transport network, *e.g.*, Optical Transport Network (OTN) [32] and Multiprotocol Label Switching-Transport Profile (MPLS-TP) [52].

OTN is able to provide transport service over optical channels and help manage network complexity. The OTN is designed to provide support for optical networking using

wavelength-division multiplexing (WDM). The benefits of OTN include universal container supporting any service type, standard multiplexing hierarchy, end-to-end optical transport transparency of customer traffic, *etc.*

MPLS-TP is a variant of the MPLS protocol and it is used in packet switched data networks. MPLS-TP is designed to overcome deficiencies of packet technology. It attempts to improve OAM (operations, administration and maintenance) functions to detect and isolate faults and to provide protection and restoration, and end-to-end QoS.

## Chapter 3

# Virtual Transport Network (VTN) and VTN-based Network Management

In this chapter, in response to the limitations of current SDN management solutions and of the traditional single-layer overlay/virtual network design, we present the details of our VTN-based approach for enterprise network management. Different from SDN which is mainly based on the TCP/IP architecture, our VTN-based approach is inspired by and built on top of a new network architecture<sup>1</sup>, the Recursive InterNetwork Architecture (RINA) [15, 16, 57, 13]. This thesis extends the RINA specification [57] to include the allocation of multi-layered VTNs.

RINA is based on the principle that *networking is Inter-Process Communication (IPC) and only IPC*. RINA solves shortcomings of the TCP/IP architecture by addressing the communication problem in a more fundamental and structured way, and provides communication services with explicit QoS support via transport flows by using a recursive building block (the IPC layer, which we call VTN), and this build block is repeated over different scopes to provide different communication services. The building block involves all kinds of mechanisms (*e.g.*, enrollment, authentication, addressing, routing, error and flow control, resource allocation) to support transport flows over a certain scope. RINA separates mechanisms and policies, and each building block can have its own policies (*e.g.*, routing, naming, access control, *etc.*) while using the same mechanisms.

The concept of *transport network* is not new, and a lot of work has been done on

---

<sup>1</sup>A VTN is termed a *DIF* in [15, 16, 13], to mean a Distributed Inter-Process Communication (IPC) Facility, a layer that can be repeated to provide communication services.

how to build such a transport network, *e.g.*, Optical Transport Network (OTN) [32] and Multiprotocol Label Switching-Transport Profile (MPLS-TP) [52]. The IPC layer in RINA has the properties of both virtual network and transport network, thus we use the term *Virtual Transport Network (VTN)* in this thesis to denote this IPC layer.

### 3.1 Virtual Transport Network (VTN) and Transport Process

A Virtual Transport Network (VTN) is the basic building block in our network management. The job of a VTN is to provide communication service with QoS support via transport flows for applications. Unlike a regular virtual network which mainly focuses on routing/tunneling, a VTN involves all kinds of mechanisms (*e.g.*, enrollment, authentication, addressing, routing, error and flow control, resource allocation) needed to support transport flows over a certain management scope. A transport flow provides end-to-end communication service with QoS parameters, which differs from a tunnel which is usually hard-coded, and just provides best-effort service over an overlaid routing path (tunnel) without resource allocation and flow and error control.

A transport process is a process that is capable of establishing transport flows within the VTN at requested QoS levels. Each VTN consists of a set of transport processes which run on different nodes (hosts)<sup>2</sup>, and the operations of its member processes are contained in the VTN itself. VTN is a secure container, where every transport process has to be explicitly enrolled into the VTN through an authentication and enrollment procedure. Each transport process contains a data transfer component supporting transport flows between different applications. The VTN provides communication service to application processes by exposing a flow allocation interface.

Each VTN has its management scope, *i.e.*, each VTN includes a limited number of transport processes running on a limited number of physical nodes. Each VTN maintains the mapping between applications and transport processes, *i.e.*, application name resolution within its scope. Note that in our management approach, there is no global address space

---

<sup>2</sup>In this thesis, we use the terms *node* and *host* interchangeably.

for application processes and an application process is only reachable over certain scopes (instead of the global scope). Thus we need the VTN structure to support communication between application processes, *i.e.*, two application processes are able to communicate if only if they have a common underlying VTN.

The same VTN mechanism can be repeated to provide a larger-scope transport service for applications by recursively using the smaller-scope transport services provided by existing VTNs. Namely, we can build VTNs of different levels, *i.e.*, multi-layered VTNs, to provide transport services over different scopes. Different VTNs use the same mechanisms but may use different network policies (*e.g.*, policies for routing and error and flow control), and the transport processes inside the same VTN follow the same policies specific to the particular VTN.

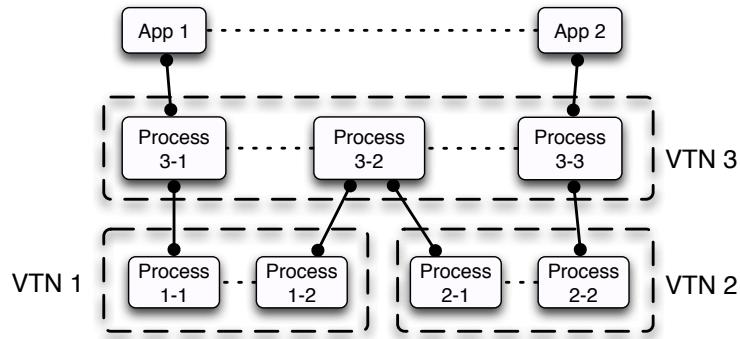


Figure 3.1: Two levels of VTNs, and *VTN 3* spans a larger scope. Each process inside a VTN is a transport process.

Figure 3.1 shows a simple example of VTNs providing transport services over different scopes. *VTN 1* and *VTN 2* each spans a smaller scope, and can provide transport services to applications inside its scope. If an application *App 1* in one scope wishes to communicate with another application *App 2* in another scope, and since *VTN 1* and *VTN 2* cannot satisfy such request, we need a higher level *VTN 3* which spans both scopes and provides a transport service across the larger scope. Recursively, we can repeat VTN to provide an even larger-scope transport service, *i.e.*, any two application processes can communicate

as long as a common underlying VTN can be found or built.

In the VTN-based network management, VTN is the basic building block, which modularizes network management. VTN encapsulates a range of operation (scope) by exposing a service specification that can be composed to form a larger-scope (high-level) VTN that ultimately meets user/application requirements.

## 3.2 Advantages of VTN-based Network Management

In this section, we highlight four advantages enabled by our VTN-based network management: (1) reducing switch memory usage, (2) improving resource utilization, (3) reducing routing overhead and (4) reducing transport overhead. The first two advantages are enabled by aggregation, and the last two advantages are enabled by layered scoping.

### 3.2.1 Switch Memory Usage

VTN allows flow aggregation which can help reduce the memory usage in switches, as well as achieve better resource utilization. Most OpenFlow switches use TCAM (Ternary Content Addressable Memory [60]) to store flow forwarding entries (rules) to increase packet processing speed, but TCAM is expensive and has limited storage capacity. Consequently, SDN management layers have to deal with the TCAM problem by reducing the number of flow rules. Most SDN work (such as [83, 38, 39, 48]) focuses on flow rules for access control or firewalling, however, due to SDN's reliance on the TCP/IP architecture, nothing much can be done to reduce the number of flow rules for end-to-end routing purpose other than using shortest path routing [51].

In our approach, flow aggregation can be easily supported by forming a new VTN, which provides a higher level of abstraction. Multiplexing and demultiplexing can be easily done with VTN's own naming and addressing mechanisms.

A simple scenario is shown in Figure 3.2. For SDN, if these  $n$  flows are not aggregatable due to distinct IP prefixes and port numbers (*i.e.*, wildcard rules cannot apply), we need

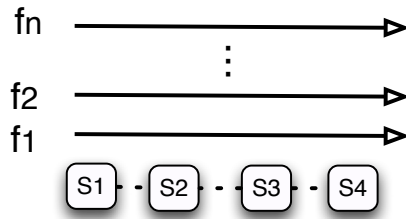


Figure 3.2:  $n$  application flows going through 4 switches (S1, S2, S3 and S4).

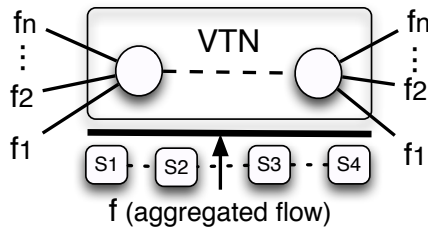


Figure 3.3:  $n$  non-aggregatable application flows can be aggregated into one flow by VTN.

a total number of  $4 \times n$  forwarding rules in switches. It gets worse as the number of non-aggregatable flows increases. However with VTN, we can aggregate these  $n$  flows into one flow  $f$  as shown in Figure 3.3, so we do not need a flow rule for each of the  $n$  flows in each switch but rather only one rule for the aggregate flow. Thus the total number of flow rules needed are 4 rules for the aggregated flow, and  $n$  rules for multiplexing and  $n$  rules for demultiplexing, at the source and destination switches, respectively, for a total of  $2n + 4$ .

### 3.2.2 Resource Utilization

VTN explicitly provides QoS support when applications request a transport service. It is important to regular users, as they can know what kinds of service they can get ahead of time, and use this information to improve their application performance. It is also important to network managers, as they can predict more accurately resource consumption, do better resource allocation and ultimately achieve better network utilization.

Flow aggregation enabled by VTN also improves resource utilization. Consider the example in Figure 3.2 again, where each flow asks for a guaranteed throughput, and let

$X_i$  ( $i = 1, \dots, n$ ) be the instantaneous traffic demand of each flow. Assume the instantaneous traffic demand for each flow follows the same uniform distribution, where the maximum instantaneous throughput is  $max$ , mean throughput is  $\mu$  and standard deviation is  $\sigma$ . Assume for each flow we reserve a bandwidth of  $\eta \times max$ , where  $\eta \in [0, 1]$  denotes the effective per-flow bandwidth requirement. Assume the QoS requirement is defined as the probability (denoted by  $1 - \epsilon$ ) that the instantaneous traffic demand for all flows does not exceed the reserved total bandwidth.

For SDN-based management without flow aggregation, to satisfy this QoS requirement, we need

$$\prod_{i=1}^n Prob(X_i \leq \eta \times max) > 1 - \epsilon \quad (3.1)$$

For our VTN-based management with flow aggregation, according to the *Central Limit Theorem*, the aggregated instantaneous flow rate follows a normal distribution, and to satisfy the same QoS requirement, we only need

$$Prob\left(\frac{\sum_{i=1}^n X_i}{n} \leq \eta \times max\right) > 1 - \epsilon \quad (3.2)$$

We can easily see that for the same  $1 - \epsilon$ , we need a bigger  $\eta$  to satisfy Equation (3.1) than Equation (3.2). That means using our VTN-based management with flow aggregation, we can satisfy the same QoS requirement with less per-flow bandwidth reservation. Namely we can serve more flow requests given limited link capacity.

Next we show this advantage through an example. Assume on average there are  $n$  flows between a pair of switches, which can be aggregated into one flow by a higher-level VTN, and the instantaneous traffic demand of each flow follows a uniform distribution between 0 Mbps and 1 Mbps. Figure 3.4 shows the effective per-flow bandwidth requirement (*i.e.*,  $\eta$ ) to satisfy different QoS requirements (*i.e.*,  $1 - \epsilon$ ). For the same  $n$ , SDN solutions (without aggregation) require more effective per-flow bandwidth requirement (almost close to 100%

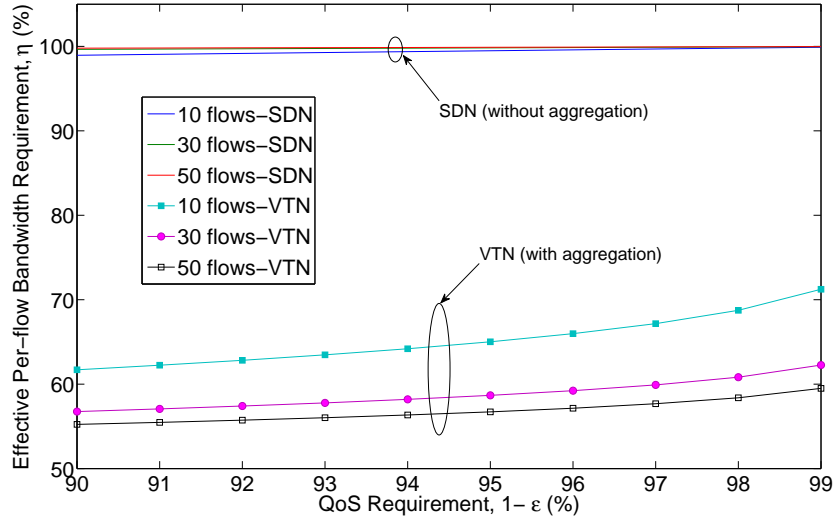


Figure 3.4: Effective per-flow bandwidth requirements for different QoS.

of the peak demand for higher QoS) than our solution. Also Figure 3.4 shows that with our solution, as the number of aggregated flows increases, the effective per-flow bandwidth requirement decreases to satisfy the same level of QoS. This shows that the more flows that are aggregated, the better performance our management approach achieves. Similar advantage of flow aggregation was also shown in previous work [8].

### 3.2.3 Routing Overhead

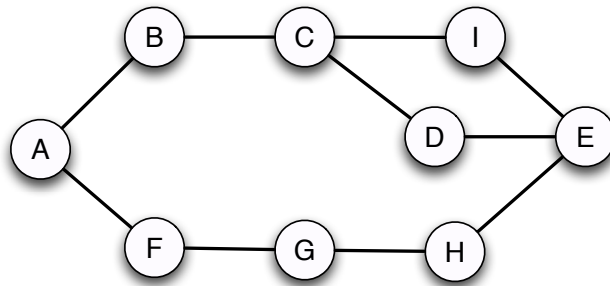


Figure 3.5: A simple network with 9 nodes.

Figure 3.5 shows a simple network with 9 nodes, and we would like to provide commu-

nication between applications on Node  $A$  and Node  $E$ . For the single-layer design, there is only one VTN with 9 transport processes (one on each of these 9 nodes), which is a one-to-one mapping to the physical network. *Note that, communication over each physical link is managed by a level-0 VTN, however, we call such a design “single layer” as this layer provides global communication over the whole network by spanning all nodes.* Assume the network uses a link-state routing protocol with hop count as path cost, and link-state update (LSU) messages are periodically scheduled every  $t$  seconds. An LSU message is broadcast by a node to all other nodes of the network only if the node measures significant change in the link state (*e.g.*, throughput, delay). Further assume that the average transmission time of an LSU message over a link is  $t^*$  seconds, where  $t > t^*$  as it typically requires multiple message (probe) transmissions by a node to measure the state of its outgoing links.

In this example, there are three shortest paths from  $A$  to  $E$  ( $path_1$ :  $A-F-G-H-E$ ,  $path_2$ :  $A-B-C-I-E$  and  $path_3$ :  $A-B-C-D-E$ ). Assume in steady state,  $A$  chooses  $path_1$  to route its packets to  $E$ . When link  $H-E$  is down, it takes up to  $t + 3t^*$  seconds for  $A$  to detect this failure and switch to another path. More generally for the single-layer design, given the diameter of the network is  $D$  hops, then it takes at most  $t + (D - 1)t^*$  seconds for a node to detect a (single) significant link change.

Under the multi-layer approach, we can have a multi-layer design<sup>3</sup> as shown in Figure 3.6, where  $VTN5$  provides the communication service between applications on  $A$  and  $E$ .  $VTN5$  consists of four virtual links supported by four underlying (level-1) VTNs:  $VTN1, VTN2, VTN3$  and  $VTN4$ . Each VTN is *independently managed* and we assume that each uses a link-state routing protocol with hop count as path cost. For level-1 VTNs, LSU messages are periodically scheduled every  $t$  seconds, similar to the single-layer design. However, in the level-2 VTN, LSU messages are periodically scheduled every  $T = t + (d_1 - 1)t^*$  seconds, where  $d_1$  is the maximum diameter of a level-1 VTN (in this example,  $d_1 = 2$ .) The reason is to allow a level-1 VTN sufficient time to adapt internally

---

<sup>3</sup>Level-0 VTNs for each physical link are not shown.

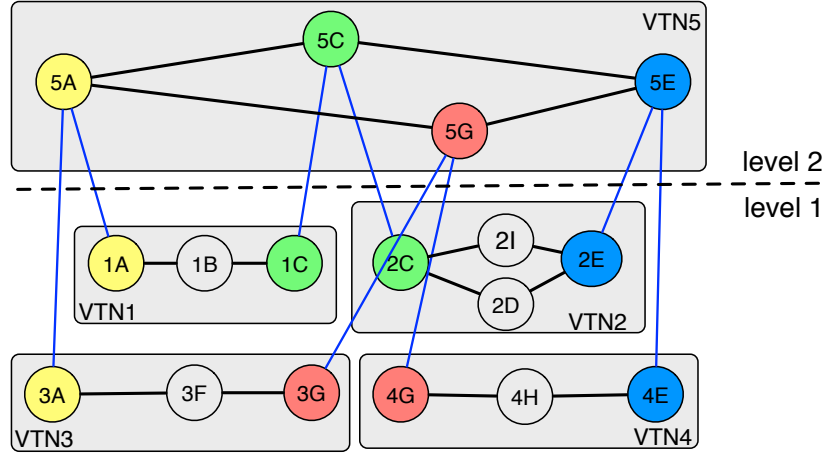


Figure 3.6: A multi-layer VTN design, where there are three transport processes (colored with the same color) on  $A$ ,  $C$ ,  $G$  and  $E$ , respectively, and there is one transport process on  $B$ ,  $D$ ,  $F$ ,  $H$ , and  $I$ , respectively.

(if possible) to a significant link-state change that affects the state of the higher level (virtual) link supported by this lower level VTN. This in turn avoids the triggering of an LSU message at the higher level, thus reducing overall routing overhead. Note that if an LSU message is triggered due to a significant link-state change detected in the level-2 VTN, in the worst case, it takes this message  $T + (d_2 - 1) \times d_1 t^*$  seconds to reach every other node, where  $d_2$  is the diameter of the level-2 VTN (in this example,  $d_2 = 2$ ), and  $d_1 t^*$  represents the message transmission time over each virtual link, which requires transmission over an underlying (level-1) VTN.

Continuing with our numerical example, inside  $VTN5$ , assume transport process  $5A$  (on node  $A$ ) uses the lower path ( $5A-5G-5E$ ) to route data packets to  $5E$  (on node  $E$ ). Assume physical link  $H-E$  fails. Then the (virtual) link  $5G-5E$ , which is supported by  $VTN4$  via a path that includes the failed link  $H-E$ , is ultimately detected to be down at node  $5A$  after  $T + (d_2 - 1) \times d_1 t^* = t + (d_1 - 1)t^* + (d_2 - 1) \times d_1 t^* = t + 3t^*$  seconds.

We can see that in this numerical example, the multi-layer design takes the same time to detect link failure compared to the single-layer design. Next we show that, given the same failure recovery performance, the multi-layer design reduces routing overhead. For

a network using a link-state routing protocol, we define the *routing overhead* as the total number of LSU messages received per second by all nodes of the network. Thus *the routing overhead for a network is at most equal to the square of its size (in number of nodes) multiplied by its LSU frequency.*

For the single-layer design, in steady state, the total routing overhead, expressed in messages per second, is given by:

$$O_{single} = \frac{9^2}{t} = \frac{81}{t} \quad (3.3)$$

For the multi-layer design (shown in Figure 3.6), the total routing overhead, expressed in messages per second, is given by:

$$O_{multi} = \frac{4^2}{T} + 3 \times \frac{3^2}{t} + \frac{4^2}{t} \quad (3.4)$$

The terms in Equation (3.4) represent the routing overhead for *VTN5*, for each of *VTN1*, *VTN3* and *VTN4*, and for *VTN2*, respectively. Since  $T > t$ , we have:

$$O_{multi} = \frac{16}{T} + \frac{43}{t} \quad (3.5)$$

$$\begin{aligned} &< \frac{16}{t} + \frac{43}{t} \\ &= \frac{59}{t} \end{aligned} \quad (3.6)$$

From Equations (3.3) and (3.6), we conclude that the routing overhead under the multi-layer design is lower than that of the single-layer design.

In practice, the routing overhead under the multi-layer design could be even lower. Assume in Figure 3.6, inside *VTN5*, transport process *5A* uses the upper path (*5A-5C-5E*) to route its data packets to transport process *5E*, and within *VTN2*, transport process *2C* uses the lower path (*2C-2D-2E*) to route its data packets to transport process *5E* to support the (virtual) link *5C-5E*. If the physical link *D-E* fails, *VTN2* can support the virtual link *5C-5E* via another path (*2C-2I-2E*), so the high-level VTN is not affected by

this link change, and inside  $VTN5$  no LSU message is triggered by this lower-level link failure. In this case, the first term in Equation (3.5) vanishes and the routing overhead under the multi-layer design is given by:

$$O_{multi} = \frac{43}{t} \quad (3.7)$$

In this simple example, we can see that the multi-layer design reduces routing overhead. *The key idea is that we limit the scope in which link-state messages are propagated, and avoid unnecessary communication with remote nodes.*

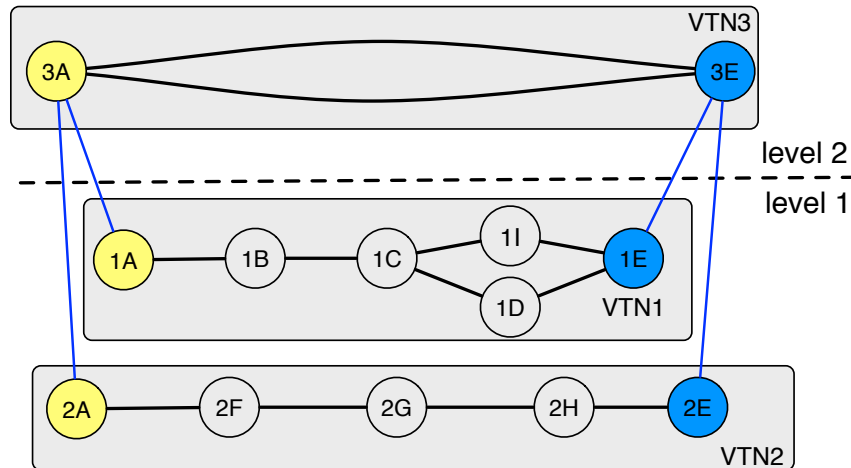


Figure 3.7: Another possible multi-layer design for the same network, where there are 3 transport processes on  $A$  and  $E$ , respectively, and there is 1 transport process on  $B$ ,  $C$ ,  $D$ ,  $F$ ,  $G$ ,  $H$  and  $I$ , respectively.

Another possible multi-layer design is shown in Fig 3.7, where there are two level-1 VTNs ( $VTN1$  and  $VTN2$ ) and one level-2 VTN ( $VTN3$ ). In  $VTN3$ , there are two direct (virtual) links between  $3A$  (on node  $A$ ) and  $3E$  (on node  $E$ ), and each of them is supported by one level-1 VTN. For example, within  $VTN3$ , when the lower link (supported by  $VTN2$ ) is down due to link failure on physical link  $H-E$ , transport process  $3A$  switches to the upper link (supported by  $VTN1$ ). In this design, for level-1 VTNs, again assume that LSU messages are periodically scheduled every  $t$  seconds, so for the level-2 VTN,

*i.e.*, *VTN3*, the time needed to detect the *H-E* link failure is given by  $t + (d_1 - 1)t^* = t + 3t^*$ , and since *VTN3* has only two processes, no LSU messages are exchanged within *VTN3*. During  $t$  seconds, the routing overhead is  $6^2 = 36$  for *VTN1* and  $5^2 = 25$  for *VTN2*, so the total routing overhead per  $t$  seconds is  $36 + 25 = 61$ . This is also smaller than that of the single-layer design, which requires 81 messages per  $t$  seconds (cf. Equation (3.3)).

Observe that for any network, there may be many different possible multi-layer designs, and our goal is to come up with a design with best network and application performance. This goal is achieved by solving the multi-layer VTN design problem discussed in Chapter 4. For example, the multi-layer design of Figure 3.6 yields lower routing overhead (cf. Equations (3.6) and (3.7)) than that of Figure 3.7 since lower-level VTNs are of smaller size, which limits the scope of propagation for LSU messages.

### 3.2.4 Transport Overhead

Our VTN-based approach allows transport flows to start and end anywhere compared to only end-to-end in Internet.

For a TCP connection of  $H$  hops, assume each hop has a packet loss rate of  $P$ , then the expected number of transmissions for all hosts along the path to successfully deliver one packet can be computed using Equation (3.8)<sup>4</sup>.

$$E_{tcp} = \frac{\left(\frac{1}{1-P}\right)^H - 1}{P} \quad (3.8)$$

Consider breaking one TCP connection into  $m$  segments, and let each segment provide reliable transport service. Then the expected number of transmissions for all hosts along the path to successfully deliver one packet is the summation of expected number of transmissions for each segment as follows.

$$E_{multi-seg} = m \times \frac{\left(\frac{1}{1-P}\right)^{\frac{H}{m}} - 1}{P} \quad (3.9)$$

---

<sup>4</sup>The proof of Equation (3.8) can be found at the Appendix A.

For the network shown in Figure 3.5, assume the packet loss rate on each link is 10%, and our goal is to provide reliable end-to-end communication between two applications, one on  $A$  and another on  $E$ .

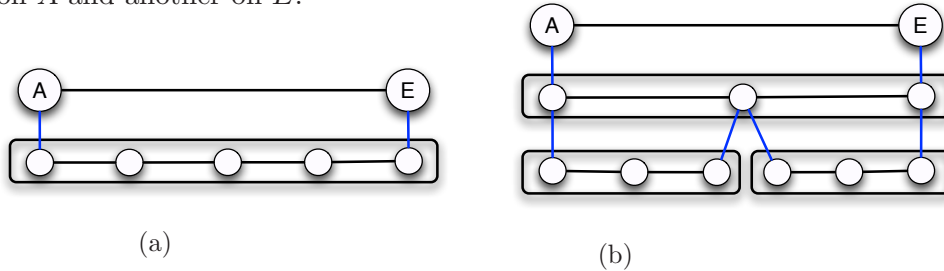


Figure 3.8: (a) The TCP connection between applications on  $A$  and  $E$  involves 4 hops under single-layer design. (b) The same flow between applications on  $A$  and  $E$  can be supported by multi-layer VTNs.

Figure 3.8(a) shows the 4-hop TCP connection between  $A$  and  $E$  for the single-layer design. The expected number of transmissions for successfully sending one packet from  $A$  to  $E$  is 5.24 (obtained from Equation (3.8)). However we can use a multi-layer design as shown in Figure 3.8(b). We can first achieve reliable communication for each link in the high-level VTN via the two low-level VTNs, and consequently the high-level VTN provides reliable communication for the applications on  $A$  and  $E$ . In this case, the average number of transmissions is 4.69 (obtained from Equation (3.9) and assuming there is no packet loss due to congestion in the high-level VTN).

Figure 3.9 shows the average number of transmissions per successful packet delivery for flows of different length. We can see that the longer the flow is, the more improvement the multi-layer design can achieve. Also the more reliable segments the flow is divided into, the larger the improvement of the multi-layer design.

In this simple example, we can see that the multi-layer design reduces the transport overhead. *The key idea is that we break a large transport scope into small scopes, and retransmission is only done over each smaller scope (instead of end-to-end over the whole large scope), thus reduce the transport overhead.*

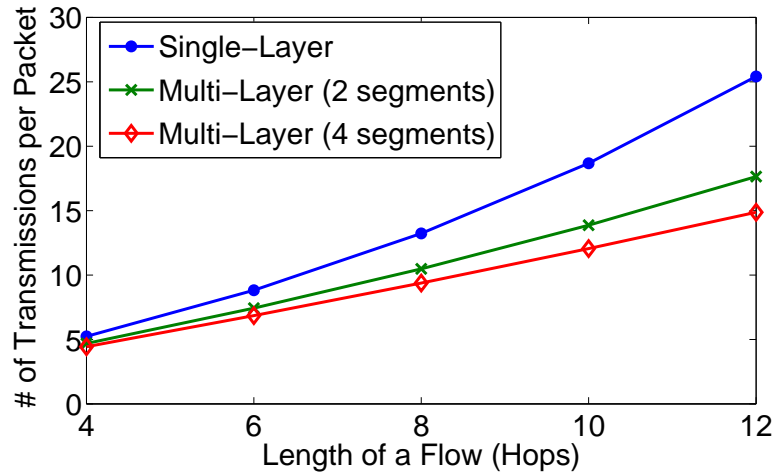


Figure 3.9: Average number of transmissions for one successful packet delivery for flows of different length.

### 3.3 Instantiations of the Multi-layer Management Framework

There is existing work using similar ideas of scoping to achieve better network performance, however most of them only focus on one aspect. Our multi-layer design provides a unified framework which enables scoping for different purposes at the same time. Most existing work related to scoping can be seen as instantiations of our unified framework. Here we show two examples of such instantiations.

#### 3.3.1 Routing

Hazy Sighted Link State (HSLs) [62] is a routing protocol that aims to scale link-state routing for ad hoc networks by limiting the scope of link-state updates in space and over time.

Under HSLs, in steady state, a node sends link-state updates at higher frequency to nodes that are closer to it, and at lower frequency to nodes that are far away from it. Namely, all nodes whose distance (in hops) from a given node lies in the range  $(2^i, 2^{i+1}]$  ( $i = 0, 1, \dots$ ) can be seen as forming a level- $i$  VTN whose link-state update frequency is  $\frac{freq}{2^i}$  (assuming  $freq$  is the frequency for its direct neighbors).

### 3.3.2 Transport

WTCP [59] (transport level) and Snoop [9] (link level) are two separate protocols for improving the performance of TCP connections involving a wireless last hop.

They both view a TCP connection as two segments: (1) the part between the fixed host and base station and (2) the part between the base station and mobile host. They both maintain the end-to-end TCP connection semantics between the fixed host and mobile host, *i.e.*, the base station is transparent to both ends. The base station buffers the TCP segments and locally retransmits them based on the timeouts and acknowledgements. Using local retransmission between the base station and mobile host, they avoid unnecessary end-to-end retransmissions. The two segments of a TCP connection can be seen as two VTNs, and the end-to-end communication is provided by another high-level VTN. This high-level VTN uses the services provided by the two underlying VTNs, which provide transport service over their own scope.

## Chapter 4

# Multi-layer VTN Design Problem and Algorithms

In this chapter, we present the multi-layer VTN design problem, which determines the VTN structure needed to support application flow requests. To the best of our knowledge, our work is the first to formulate the flow allocation problem as a multi-layer VTN design problem.

### 4.1 Problem Definition

For a set of application flow requests, the *multi-layer VTN design problem* is to determine the VTN structure, which includes: (1) the number of VTNs needed, (2) the level each VTN belongs to, and (3) the nodes<sup>1</sup> where the transport processes of each VTN should be created. Note that the designed VTN structure, *i.e.*, the output of the multi-layer VTN design algorithm, can be formed on real networks using our VTN-based management architecture (details in Chapter 5 - Chapter 7). The notations used in this chapter can be found in *List of Symbols* on page xviii.

The multi-layer VTN design problem can be further divided into two stages: (1) *the path selection stage*, and (2) *the VTN design stage*.

#### 4.1.1 Path Selection Stage

In the path selection stage, a path on the given level- $(n - 1)$  graph is selected (if possible) for each given flow request such that its QoS requirements (*e.g.*, throughput, delay) are satisfied. Because of explicit QoS support, we are able to easily find paths for flow requests

---

<sup>1</sup>As mentioned earlier, in this thesis, we use the terms *node* and *host* interchangeably.

with different QoS requirements. Also due to scoping at the transport level enabled by VTN, we can achieve better resource utilization compared to other management approaches (such as SDN or traditional approaches). Note that flows between the same source and destination may be aggregated into a single aggregated flow, and the multiplexing and demultiplexing of flows can be easily supported by VTN's own naming and addressing mechanisms.

#### 4.1.2 VTN Design Stage

The VTN design stage determines the VTN structure based on the paths selected in the previous stage, where the VTNs are multi-layered. Paths selected are enforced by the VTN structure via consistent policies across related VTNs. If shortest path routing is used, then shortest path routing for VTNs at all levels guarantees that the shortest path is used on the level-0 topology. Also for SDN-style routing, installing forwarding rules on corresponding transport processes of VTNs can also guarantee that selected paths are used on the level-0 topology

#### 4.1.3 A Simple Example

Here we show an example that briefly explains the two stages of the multi-layer VTN design problem.

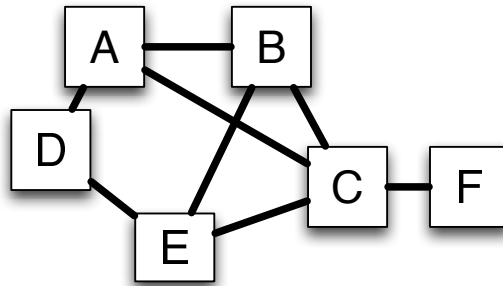


Figure 4.1: A simple network with 6 nodes.

In this example, we have a simple network with 6 nodes as shown in Figure 4.1. Assume

there are a total of 45 flow requests, 10 flows between applications on Node B and Node F, 20 flows between applications on Node B and Node D, and 15 flows between applications on Node A and Node E. Next we show how these flow requests can be supported by solving the multi-layer VTN design problem.

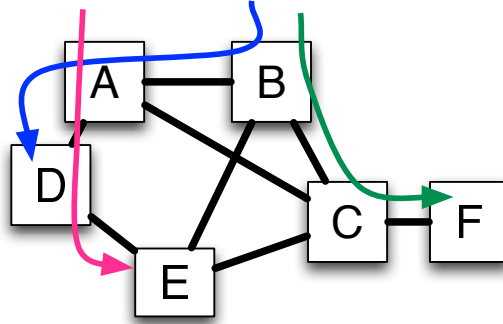


Figure 4.2: The output of the path selection stage includes 3 aggregated flows.

Assume the network capacity is enough to support all these flows, and shortest path routing is used. So after finishing the path selection stage, the output includes 3 aggregated flows as shown in Figure 4.2, one aggregated flow B-F on selected path B-C-F, one aggregated flow B-D on selected path B-A-D, and one aggregated flow A-E on selected path A-D-E.

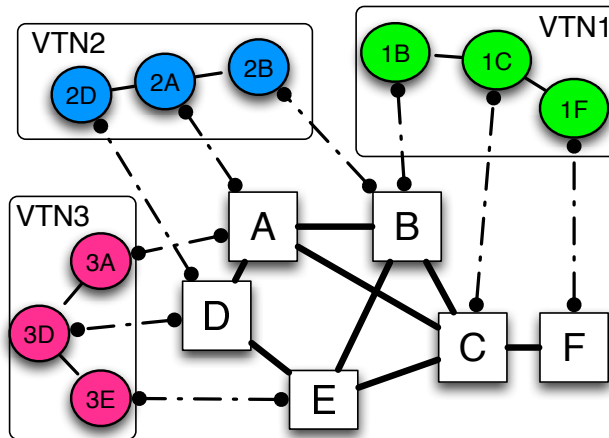


Figure 4.3: 3 VTNs are designed in the VTN design stage.

Then after solving the VTN design stage, the output includes 3 newly designed VTNs (shown in Figure 4.3), which are needed to support these 3 aggregated flows (one VTN for each aggregated flow). In the end, the original 45 flow requests are supported by the designed VTN structure.

## 4.2 Problem Properties

Most existing overlay/virtual networks are single-layered. There may be multiple overlay/virtual networks over the same physical infrastructure, but they all belong to the same and single layer (level), *i.e.*, just one level above the Internet layer, which is flat and global. However the multi-layer design can help reduce the management overhead via dynamic scoping. *The most important aspect for the multi-layer VTN design problem is that each stage can be modeled as a separate optimization problem with different performance/cost goals to satisfy different requirements.*

The multi-layer VTN design is a recursive problem. Initially, transport over each physical link is provided by a level-0 VTN, which only has two transport processes, one on each end of a physical link. So for the base case of our recursion, the level-0 network graph is given by the virtual links supported by level-0 VTNs and nodes running level-0 VTN transport processes. For the inductive case (level- $n$ ,  $n \geq 1$ ), the given graph is a virtual graph, where each link between two nodes is a virtual link, and two nodes have a (virtual) link between them if they have transport processes belonging to a common existing underlying level- $(n - 1)$  VTN.

Note that the framework for VTN design and mapping can be used as a stand-alone tool. It can be used not only within our VTN-based management architecture, but also to solve existing overlay/virtual network problems. For example, cloud service providers (*e.g.*, Amazon AWS [3] and Microsoft Azure [4]) can use it to optimize their virtual network provisioning.

Next we show two case studies which demonstrate how our multi-layer design framework

can be instantiated to meet different specific network goals.

### 4.3 Case Study (1) : Constrained Path Selection with Unconstrained VTN Design

In the first case study, we have a constrained path selection stage and an unconstrained VTN design stage, and we focus on the flow aggregation enabled by VTN, which enables better resource utilization via flow multiplexing and demultiplexing at multiple levels. We show how we model the path selection stage as an optimization problem to achieve better resource utilization. We also demonstrate that, compared to SDN-based approaches, our approach can serve more flow requests (subject to bandwidth constraints) while using less memory in switches. Note that the notations used in this section can be found in *List of Symbols* on page xviii.

#### 4.3.1 Constrained Path Selection

Given a network topology  $G_{n-1} = \langle V, E^{n-1} \rangle$ <sup>2</sup> and a total of  $n$  flow requests, the goal is to find a path (if possible) for every flow request satisfying its throughput requirement<sup>3</sup>. This path selection can be formulated as an *Integer Linear Programming (ILP)* problem as shown in Table 4.1. In this problem,  $O_k^n = 1$ , if  $p(f_n) = p_k^n$  (*i.e.*, among all possible paths for  $f_n$ , path  $k$  is selected); otherwise 0. The objective function is to maximize the number of flow requests served while seeking paths with shorter length (by using the inverse of path length as weight) for each flow as long as the link capacity constraints are satisfied. Line (1) guarantees only one path is selected (if possible) for each flow  $f_n$ . In Line (2),  $\chi_n^{st} = 1$  if  $e_{st} \in p(f_n)$  (*i.e.*, link  $e_{st}$  is on the path selected for  $f_n$ ); otherwise 0. Line (2) guarantees that the bandwidth requirements of all flows going through a link do not exceed the capacity of that link.

---

<sup>2</sup>In this case study, our path selection is done over the level-0 network topology  $G_0 = \langle V, E^0 \rangle$ , which is a one-to-one mapping to the physical topology.

<sup>3</sup>We use throughput as an example of QoS support. Other features such as latency and packet loss can be easily considered.

<p>Objective: maximize <math>\sum_{n=0}^N \sum_{k=0}^{ P(f_n) } O_k^n \times \frac{1}{l(p_k^n)}</math>, such that</p> <p><math>\forall f_n \in F : \sum_{k=0}^{ P(f_n) } O_k^n \leq 1</math> <span style="float: right;">(1)</span></p> <p><math>\forall e_{st} \in E : \sum_{n=0}^N b(f_n) \times \chi_n^{st} \leq C_{st}</math> <span style="float: right;">(2)</span></p>
--

Table 4.1: Path selection formulated as an ILP problem.

We are able to use CPLEX[31] to solve the ILP problem in Table 4.1. Note that some flow requests may not be served due to link capacity constraints, but with aggregation we can satisfy the same QoS requirement with less effective bandwidth usage (cf. Figure 3.4). Thus after aggregation we have more capacity left and in turn we are able to accept more flow requests. So after solving the current ILP problem, we compute the effective bandwidth usage on each link (cf. Equation (3.2)) and update its link capacity  $C_{st}$ . Then we repeat solving the path selection problem in Table 4.1 using CPLEX for the unaccepted flow requests based on the new residual link capacity. This procedure is stopped when no more flow requests can be accepted or all flow requests have been accepted.

In the end, the set of all flow requests that can be accepted is denoted as  $F'$ . The accepted flow requests which are mapped on the same level-0 path between the same pair of source and destination switches can be aggregated into one flow. In practice, we aggregate a set of aggregatable flows only if the aggregated flow uses less memory entries than without aggregation. Each  $f_m \in F'$  will be then supported by some VTN which is designed in the next stage. In other words, all accepted flows mapped on the same level-0 path are all supported by a single path in some level- $n$  VTN, where each (virtual) link of that VTN is supported by a flow through one level- $(n - 1)$  VTN. Through aggregation we can accept more flow requests while reducing the number of forwarding entries installed on the intermediate switches as discussed in Section 3.2.1.

### 4.3.2 Unconstrained VTN Design

In this stage, we determine how many new VTNs need to be formed on top of an existing level- $(n - 1)$  network topology (*i.e.*,  $G_{n-1}$ ) and existing level- $(n - 1)$  VTNs (*i.e.*,  $D^{n-1}$ ), and determine which hosts each new VTN has to span to support each  $f_m \in F'$ , where  $F'$  is the set of flow requests that are accepted after the previous stage. The outputs of this stage are: (1) a set of new VTNs, (*i.e.*,  $\{D_z\}$ ), (2) the host ( $v_i$ ) that each new transport process in each VTN (*i.e.*,  $v_s^z$ ) is assigned to, and (3) how each new process is connected, *i.e.*, virtual links ( $\{e_{st}^z\}$ ) in each VTN. Note that a new VTN may need to be supported by multiple level- $(n - 1)$  VTNs as shown in Figure 3.1. The goal of the VTN design stage is to support all  $f_m \in F'$  with a set of VTNs (and corresponding new transport processes).

A recursive VTN design algorithm is shown in Algorithm 1. In this algorithm, a VTN is designed to support each  $f_m \in F'$  if the VTN has transport processes on each host along the path (*i.e.*,  $p(f_m)$ ), and there is a (virtual) link between processes on each pair of neighboring hosts along the path. So for  $f_m \in F'$ , the corresponding level- $n$  VTN needs to span all hosts along the path, and the flow is mapped to a path in this level- $n$  VTN, where each link on this path is supported by a flow in one level- $(n - 1)$  VTN.

Assume each VTN can have at most  $M$  ( $M \geq 2$ ) transport processes. By setting different values for  $M$ , we could decide the total number of newly created VTNs as well as their size. Namely, a larger  $M$  yields larger management scopes, and a smaller  $M$  yields smaller ones. Let us denote  $\Delta(f_m, D_j)$  as the number of new transport processes to be added to an existing VTN  $D_j$  so that  $D_j$  can support  $f_m$ . For example, assume  $p(f_m)$  includes 5 hosts, and  $D_j$  already has transport processes on 3 of these hosts, then  $\Delta(f_m, D_j)$  is 2.

We may use an existing VTN (lines 4-6) or expand an existing VTN (lines 8-16) to support a flow, as long as the number of transport processes in that VTN does not exceed  $M$ . If we cannot support this flow using existing VTNs, we create a new VTN (lines 19-24). Note that due to the limitation of  $M$  (line 18), we may need to build a VTN of even higher

---

**Algorithm 1** VTN\_Design ( $F', G_{n-1}, D^{n-1}, M$ )

---

```

1:  $t = |F'|$ ,  $E^n = E^{n-1}$ ,  $D^n = D^{n-1}$ 
2: while  $t > 0$  do
3:   get a  $f_m \in F'$  with largest  $|p(f_m)|$ 
4:   if any existing  $D_k$  in  $D^n$  has processes on all  $v_i \in p(f_m)$  then
5:     add links to  $D_k$  and  $E^n$ , for each pair of neighbors in  $p(f_m)$ 
6:      $F' = F' / f_m$ 
7:   else
8:     get a  $D_j \in D^n$  with largest  $\Delta(f_m, D_j)$ 
9:     if  $(|V_j| + \Delta(f_m, D_j)) \leq M$  then
10:      for all  $v_i \in p(f_m)$  do
11:        if no existing process in  $D_j$  is assigned to  $v_i$  then
12:           $s = |V_j| + 1$ ,  $V_j = V_j \cup v_s^j$  and assign  $v_s^j$  to  $v_i$ 
13:        end if
14:      end for
15:      add links to  $D_j$  and  $E^n$ , for each pair of neighbors in  $p(f_m)$ 
16:       $F' = F' / f_m$ 
17:    else
18:      if  $|p(f_m)| \leq M$  then
19:         $l = |D^n| + 1$ ,  $V_l = \phi$ 
20:        for all  $v_i \in p(f_m)$  do
21:           $s = |V_l| + 1$ ,  $V_l = V_l \cup v_s^l$ , and assign  $v_s^l$  to  $v_i$ 
22:        end for
23:        add links to  $D_l$  and  $E^n$ , for each pair of neighbors in  $p(f_m)$ 
24:         $D^n = D^n \cup D_l$ , and  $F' = F' / f_m$ 
25:      else
26:         $f_m$  cannot be supported by the current level
27:      end if
28:    end if
29:  end if
30:   $t = t - 1$ 
31: end while
32:  $D_{level(n)} = \text{Merge\_VTN}(D_{level(n)}, M)$ 
33: if  $F' = \phi$  then
34:   Return  $G_n = \langle V, E^n \rangle$  and  $D^n$ 
35: else if  $F' \neq \phi$  then
36:   VTN_Design ( $F', G_n, D^n, M$ )
37: end if

```

---

level to serve a flow  $f_m$  (line 26), and this is done by the recursive call in line 36. As we have more levels of VTNs, we would have more (virtual) links (lines 5, 15 and 23) and yield higher level paths with smaller length. But in this case study, we focus on building only one more level of VTNs (*i.e.*, level-1 VTNs) by choosing an  $M$  value that is larger than  $|p(f_m)|$  for all  $f_m$  (line 18).

At the end of each recursion call, we try to further reduce the number of VTNs by merging existing VTNs in the same level, *i.e.*,  $D_{level(n)}$ , using Algorithm 2 (line 32). Two VTNs can be merged if the number of transport processes in the merged VTN does not exceed  $M$ . Two VTNs ( $D_x$  and  $D_y$ ) are merged into a new VTN ( $D_z$ ) as follows. First, the set of hosts, which have transport processes belonging to the new VTN, is the union of the two sets of hosts in  $D_x$  and  $D_y$ , *i.e.*,  $S_z = S_x \cup S_y$ . Second, two transport processes on two hosts in the new VTN have an (virtual) edge between them if either (1) they have an edge in  $D_x$  or  $D_y$ ; or (2) there exists a common underlying lower-level VTN that can support a transport flow between them.

---

**Algorithm 2 Merge\_VTN** ( $D_{level(n)}, M$ )

---

```

1:  $D'_{level(n)} = \phi$ 
2: while  $D_{level(n)} \neq \phi$  do
3:   pick a  $D_i \in D_{level(n)}$ 
4:   while  $|V_i| < M$  do
5:     if  $\exists D_j \in D_{level(n)}$ , where  $i \neq j$  and  $|V_i \cup V_j| \leq M$  then
6:        $D_i = \text{merge}(D_i, D_j)$ ,  $D_{level(n)} = D_{level(n)} / D_j$ 
7:     else
8:       break // exit inner while loop
9:     end if
10:  end while
11:   $D'_{level(n)} = D'_{level(n)} \cup D_i$ ,  $D_{level(n)} = D_{level(n)} / D_i$ 
12: end while
13: return  $D'_{level(n)}$ 

```

---

### 4.3.3 Performance Evaluation

In this section, we present the performance evaluation of the first case study. As mentioned in Section 3.2.1, our VTN-based management approach allows aggregation of flows that cannot be aggregated using SDN-based management approaches due to distinct IP prefixes

and port numbers. Flow aggregation helps achieve better resource utilization to accept more flow requests and reduce memory usage in switches for storing forwarding rules.

We use BRITE [47], a topology generation tool to generate an enterprise network (50 switches and 200 directed links) using the Waxman model ( $\alpha = 0.15$  and  $\beta = 0.2$ ), and then randomly generate flow requests between pairs of switches. Each physical link has a capacity of 100 Mbps, and each flow request has an instantaneous traffic demand which follows a uniform distribution between  $[0, 1]$  Mbps. Assume the QoS requirement (cf.  $1 - \epsilon$  in Equation (3.1) and Equation (3.2) in Chapter 3) for each flow request is 90%.

#### 4.3.3.1 Flow Request Acceptance Ratio

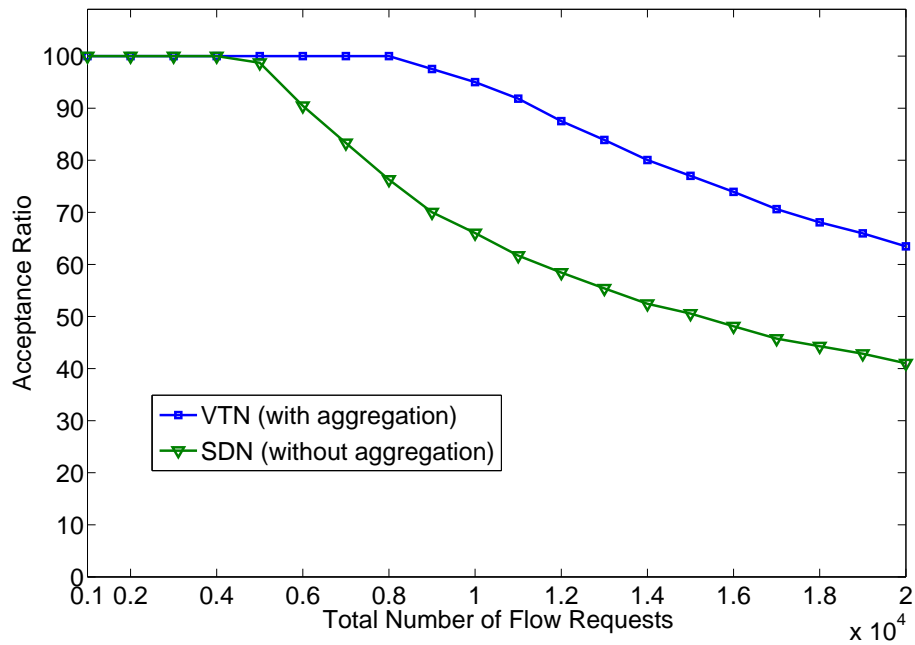


Figure 4.4: Acceptance ratio of flow requests, where  $1 - \epsilon = 90\%$ .

Because of aggregation, our management approach is able to satisfy the same QoS requirement with less bandwidth reservation (cf. Figure 3.4), and thus can accept more flow requests. Figure 4.4 shows the acceptance ratios for different number of flow requests. When the number of flow requests is less than 5,000, both VTN and SDN solutions can

accept all flow requests (*i.e.*, acceptance ratios are both 100%) since the total traffic demand is less than the network capacity. But as the number of flow requests increases (higher traffic demand), we can see that our solution has a higher acceptance ratio than an SDN-based solution.

#### 4.3.3.2 Memory Usage

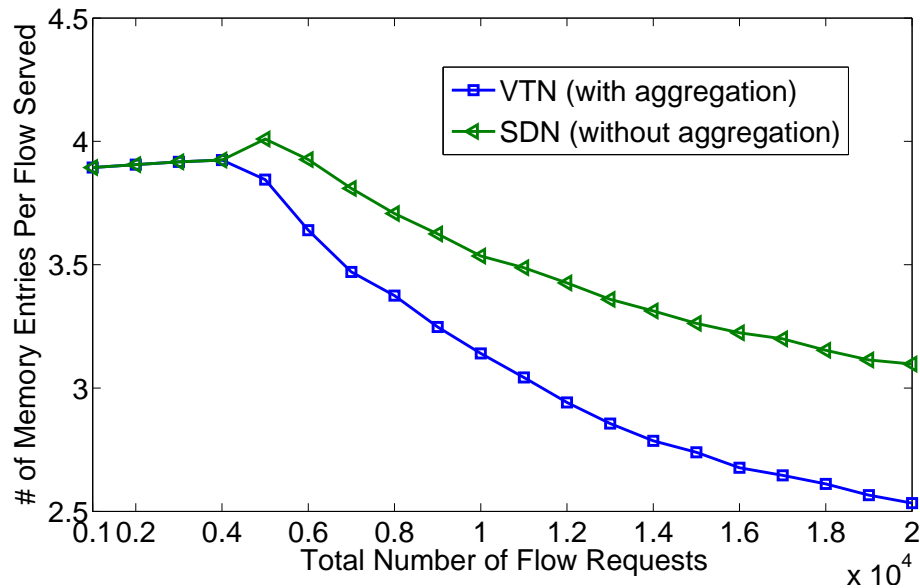


Figure 4.5: Average number of memory entries needed for each flow served.

Our management approach is able to save memory usage in switches for storing forwarding rules due to aggregation (cf. Figure 3.2 and Figure 3.3). Figure 4.5 shows the average number of memory entries needed to serve each flow as the number of flow requests increases. We can see that our VTN-based solution has less per-flow memory usage compared to an SDN-based solution. Also we can see that as the number of flow requests increases, per-flow memory usage decreases for our VTN-based solution. This is because more flows can be aggregated, *i.e.*, the higher the number of flow requests, the better performance our VTN-based management approach achieves. Note that per-flow memory usage for the SDN-based solution (which depends on average path length) also decreases,

and this is because the flow requests are randomly generated, and the average path length of flows accepted decreases as we have more flow requests.

#### 4.3.3.3 Number of New VTNs

Figure 4.6 shows the average number of new VTNs created per 1000 flows served for different values of  $M$  (*i.e.*, maximum number of processes allowed in a VTN) obtained using the VTN design algorithm (Algorithm 1). We can see that as the number of flow requests increases, the number of new VTN needed per 1000 flows served decreases. Also as expected, the higher the number of processes allowed in a new VTN, the less the number of VTNs needed.

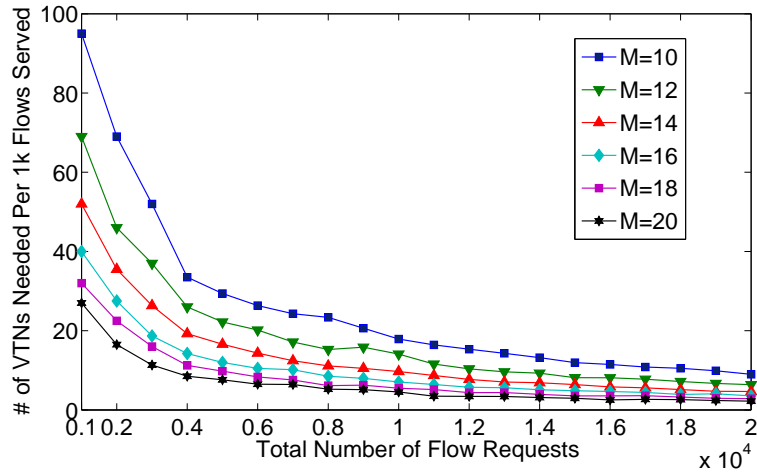


Figure 4.6: Number of new VTNs created per 1000 flows served for different values of  $M$ .

Here we only show the comparison for a network with 50 switches, but our comparisons for larger networks and more flow requests show similar results.

#### 4.4 Case Study (2): Unconstrained Path Selection with Constrained VTN Design

In the second case study, we have an unconstrained path selection stage and a constrained VTN design stage, and we focus on leveraging the VTN structure to partition the network

into smaller scopes. We show how we model the VTN design stage as an optimization problem to reduce management overhead. We demonstrate that our multi-layer design approach can reduce the routing and transport overhead through the flexibility of designing VTNs at different levels (layers), while achieving the same performance compared to the single-layer approach.

#### 4.4.1 Unconstrained Path Selection

In this case study, we assume the selected path on the level-0 graph to support each flow request is given. The path selection stage can be solved using a shortest path algorithm or by solving an *Integer Linear Programming (ILP)* problem to satisfy different QoS requirements as discussed in Section 4.3.1.

#### 4.4.2 Constrained VTN Design

For this constrained VTN design stage, we propose a two-step design algorithm. The first step is the initial design step (Section 4.4.2.1), where we build the initial VTN structure based on the path selected for each flow request on the level-0 graph. The second step is the optimization step (Section 4.4.2.2), where we optimize the initial VTN structure with the aim to reduce the total number of VTNs. By reducing the number of VTNs at each layer (level), we can further reduce the management overhead of the multi-layered VTNs. Note that the notations used in this section can be found in *List of Symbols* on page xviii.

##### 4.4.2.1 Initial Design Step

The inputs of this step are the set of flow requests (*i.e.*,  $F = \{f_n\}$ ), path selected on the level-0 graph for each flow request ( $p(f_n)$ ), and maximum diameter allowed for a VTN (*i.e.*,  $max\_dia$ ). Note that  $max\_dia$  is the main factor affecting the transport overhead as discussed in Section 3.2.4. In this step, for each flow request, we recursively build a VTN to support this flow. Due to the  $max\_dia$  constraint, it may result in multiple levels of VTNs.

Before explaining the details of our design algorithm, we first show a simple example in Figure 4.7, where a flow between two applications on *Host1* and *Host8* has 8 hosts along its selected path on the level-0 graph, and  $max\_dia = 3$ . We assume no existing VTN can support this flow, and  $S(level)$ , where  $level = 1, 2, \dots$ , denotes the path at this level computed on level- $(level - 1)$  graph. Thus  $S(1)$  represents the given selected path  $p(f_n)$  on the level-0 graph.

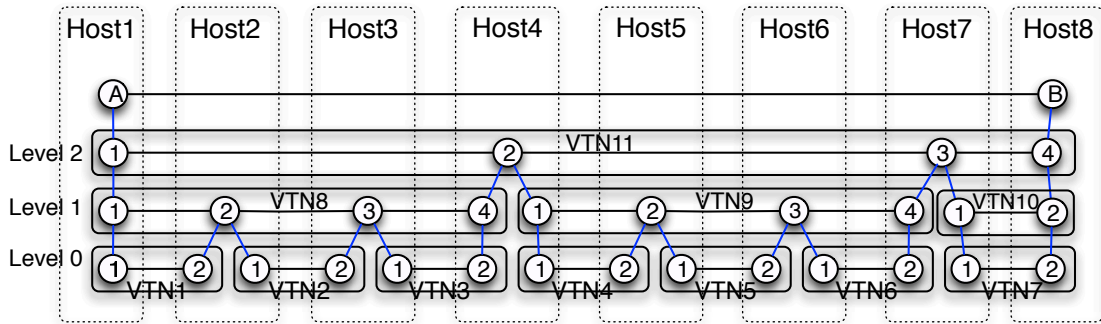


Figure 4.7: The application flow between  $A$  on *Host1* and  $B$  on *Host8* has a path of 7 hops in the level-0 graph. After the initial design step, we have 2 extra levels of VTNs when  $max\_dia = 3$ . Note that our multi-layer design has a level-0 VTN with only two transport processes for each physical wire.

Since the length of the path selected at level-0 (*i.e.*,  $S(1)$ ) is larger than the  $max\_dia$  constraint, we need to build a VTN which is supported by VTNs of multiple levels. For level-1, the path (*i.e.*,  $S(1)$ ) contains all 8 hosts. Due to the  $max\_dia$  constraint, we build 3 VTNs ( $VTN8$ ,  $VTN9$ , and  $VTN10$ ) at this level to support high-level paths. For level-2, the path (*i.e.*,  $S(2)$ ) contains 4 hosts ( $Host1$ ,  $Host4$ ,  $Host7$  and  $Host8$ ), and each (virtual) link at this level is supported by an existing lower-level VTN. For level-2's path, only one VTN ( $VTN11$ ) is needed as  $|S(2)| \leq max\_dia$ , which does not violate the  $max\_dia$  constraint. In the end, our VTN design algorithm yields 4 VTNs ( $VTN8$ ,  $VTN9$ ,  $VTN10$  and  $VTN11$ ) of 2 extra levels (level-1 and level-2). Note that  $VTN10$  maps one-to-one to  $VTN7$ , and in practice we remove all VTNs that map one-to-one to another VTN. Also note that on the same host (such as  $Host4$ ) two processes belonging to two different

VTNs at the same level (such as Process 4 of *VTN8* and Process 1 of *VTN9*), as well as the high-level process (Process 2 of *VTN11*), enable relaying over a larger scope via the higher-level VTN (*VTN11*).

---

**Algorithm 3** Initial\_VTN\_Design ( $F, D, max\_dia$ )

---

```

1: while  $F \neq \phi$  do
2:   get a flow request  $f$  from  $F$ , and remove  $f$  from  $F$ 
3:   if  $\exists$  some  $D_z \in D$ , s.t.,  $src(f) \in S_z$  and  $dest(f) \in S_z$  then
4:     do nothing //  $f$  can be served by an existing VTN
5:   else
6:     let  $level = 1$ ,  $S(level) = \mathbf{Path\_Selection}(f, level, G_{level-1})$ 
7:     while  $|S(level)| > max\_dia$  do
8:       Design_VTNs( $S(level), D, level$ )
9:        $level++$ 
10:       $S(level) = \mathbf{Path\_Selection}(f, level, G_{level-1})$ 
11:    end while
12:    Design_VTNs( $S(level), D, level$ )
13:  end if
14: end while
15: return  $D$ 

```

---

Our algorithm for the initial design step is shown in Algorithm 3. For each flow request, we first check if we can find an existing VTN that can serve this flow (*i.e.*, whether both the source host and destination host of the flow have transport processes belonging to that VTN). If so, we reuse the existing VTN (lines 3-4). Otherwise, we need to build a new VTN (which can be recursively built) to serve this request (lines 6-12). If the length of the selected path at a given level is larger than *max\_dia* (line 7), we need to build multiple VTNs at this level to support high-level VTNs, which results in VTNs of multiple levels (layers). Starting from level-1 (line 6), we first figure out the path at each level, *i.e.*,  $S(level)$ , a sequence of hosts which should have transport processes at this level, using a path selection algorithm (such as the shortest path algorithm), then we design VTNs for this level (line 8) using Algorithm 4. Our algorithm stops when a selected path at some level does not violate the *max\_dia* constraint (line 7), then we design a VTN of the top level (line 12).

Algorithm 4 shows the details on designing VTNs at a given level. We may need to build multiple VTNs at the same level, when the length of the computed path (*i.e.*,  $S(level)$ )

---

**Algorithm 4** Design\_VTNs ( $S(level), D, level$ )

---

```

1: let  $S(level) = \{\psi_0, \psi_1, \dots, \psi_{k-1}\}$ , where  $k = |S(level)|$ 
2: let  $z = |D| + 1$ ,  $S_z = \phi$  // create a new VTN
3: for ( $m = 0$ ;  $m < k$ ;  $m++$ ) do
4:   if  $|S_z| < max\_dia + 1$  then
5:      $S_z = S_z \cup \psi_m$  // assign a new transport process to host  $\psi_m$ 
6:   else
7:      $S_z = S_z \cup \psi_m$  // assign a new transport process to host  $\psi_m$ 
8:     for all pairs  $\langle s, t \rangle \in S_z \times S_z$  do
9:       add edge  $e_{st}^z$  to  $E_z$ , if  $\exists D_x \in D^{level-1}$  can support it
10:    end for
11:     $D = D \cup D_z$  // add  $D_z$  to  $D$ 
12:     $z++$ ,  $S_z = \phi$  // a new VTN is needed
13:     $S_z = S_z \cup \psi_m$  // assign a new transport process to host  $\psi_m$ 
14:   end if
15: end for

```

---

for this level is longer than  $max\_dia$ . If a host should have a transport process of VTN  $D_z$  residing on it, we assign a new transport process of this VTN (line 5 or line 7) to this host. After reaching the maximum diameter of a VTN, we then add (virtual) edges to this VTN (lines 8-10). An (virtual) edge between a pair of transport processes on two hosts in a VTN will be added, if and only if there is an existing lower-level VTN that can support this edge (line 9), *i.e.*, the hosts at the two ends of the virtual link have transport processes belonging to that underlying VTN. In other words, that underlying VTN can support a flow between this pair of transport processes on these two hosts. Then we create a new VTN (line 12), assign a transport process on the current host (line 13), and continue to next host.

#### 4.4.2.2 Optimization Step

The initial design step yields an initial VTN structure where we may have different VTNs at different levels. In the second step, we aim to optimize the VTN structure and try to reduce the number of VTNs in each level while not violating the two constraints (*i.e.*,  $max\_dia$  and  $max\_num$ ). Note that  $max\_num$  is the main factor effecting the routing overhead as discussed in Section 3.2.3. This step is finished by solving the *VTN Packing Problem* explained in the next section.

#### 4.4.3 VTN Packing Problem

The *VTN Packing Problem* is an important subproblem in the optimization step. Note that  $S_z$  denotes the set of nodes having processes belonging to VTN  $D_z$ . Given the set of VTNs at each level (*i.e.*,  $D_{level(n)} = \{D_\alpha\}$ ,  $n = 1, 2, \dots$ ), our goal is to find the minimal set of VTNs (*i.e.*,  $D'_{level(n)} = \{D_\beta\}$ ), *s.t.*, for every  $S_\alpha$  of  $D_\alpha \in D_{level(n)}$ , there exists some  $S_\beta$  of  $D_\beta \in D'_{level(n)}$  where  $S_\alpha \subseteq S_\beta$ . In other words, we need to pack all VTNs at the same level yielding a minimum number of new VTNs without violating the constraints (*max\_dia* and *max\_num*). Note that the VTN packing problem is NP-hard, and it can be simply proved by reduction from the *Sharing-aware VM Packing Problem* in [64] which is also NP-hard, where in that problem the physical host can be considered as the VTN, and the memory page can be considered as the transport process.

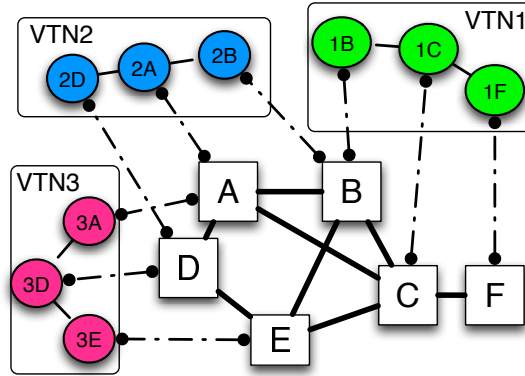


Figure 4.8: 3 VTNs of the level-1 after the initial design step.

Figure 4.8 and Figure 4.9 show a simple example of the VTN packing problem, where the network contains 6 hosts ( $A, B, C, D, E$  and  $F$ ). Figure 4.8 shows 3 level-1 VTNs after the initial design step. In the optimization step we need to find the minimum number of VTNs to pack these 3 VTNs without violating the constraints ( $max\_dia = 2$  and  $max\_num = 4$ ). Figure 4.9 shows one feasible solution with two VTNs, and we cannot further reduce the number of VTNs due to the two constraints.

We give a heuristic algorithm for the VTN Packing Problem as shown in Algorithm 5.

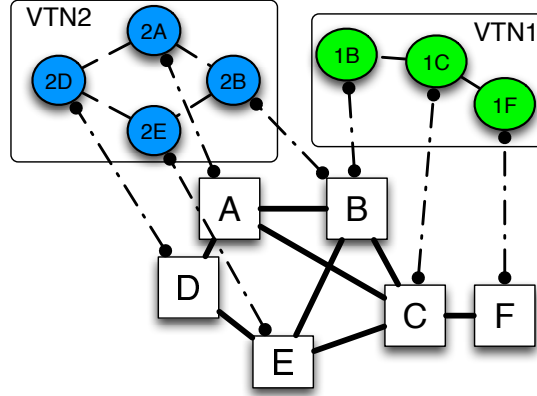


Figure 4.9: One solution of the VTN packing problem, where  $VTN2$  and  $VTN3$  from Figure 4.8 are merged into one VTN.

The input to this algorithm is the set of VTNs at the same level (*i.e.*,  $D_{level(n)}, n = 1, 2, 3, \dots$ ). For the set of VTNs at the same level, we keep merging one VTN with other VTNs until it can no longer be merged (lines 3-13). We apply Algorithm 5 to the set of VTNs at each layer from the initial design step, and eventually get the final optimized VTN structure.

---

**Algorithm 5**  $VTN\_Packing(D_{level(n)})$

---

```

1: let  $D(n) = D_{level(n)}$ , and  $D'_{level(n)} = \phi$ 
2: while  $D(n) \neq \phi$  do
3:   get a VTN  $D_x$  from  $D(n)$  and remove  $D_x$  from  $D(n)$ 
4:   let  $D'(n) = D(n)$  // look at the remaining VTNs
5:   while  $D'(n) \neq \phi$  do
6:     get a VTN  $D_y$  from  $D'(n)$ , and remove  $D_y$  from  $D'(n)$ 
7:     if  $Test\_Merge(D_x, D_y) == TRUE$  then
8:        $D_x = merge(D_x, D_y)$  //merge VTN  $D_x$  and  $D_y$ 
9:       remove  $D_y$  from  $D(n)$ 
10:    else
11:      continue //  $D_x$  and  $D_y$  cannot be merged
12:    end if
13:  end while //end inner while loop
14:  add  $D_x$  to  $D'_{level(n)}$ 
15: end while //end outer while loop
16: return  $D'_{level(n)}$ 

```

---

Two VTNs ( $D_x$  and  $D_y$ ) are merged into a new VTN ( $D_z$ ) as follows. First, the set of hosts, which have transport processes belonging to the new VTN, is the union of the two sets of hosts in  $D_x$  and  $D_y$ , *i.e.*,  $S_z = S_x \cup S_y$ . Second, two transport processes on two

hosts in the new VTN have an (virtual) edge between them if either (1) they have an edge in  $D_x$  or  $D_y$ ; or (2) there exists a common underlying lower-level VTN that can support a transport flow between them.

The *Test\_Merge* method (line 7) checks whether two VTNs can be merged, and two VTNs ( $D_x$  and  $D_y$ ) can be merged into a new VTN  $D_z$ , if only if the following requirements are satisfied: (1) there exists some host which has transport processes belonging to both VTNs, *i.e.*,  $S_x \cap S_y \neq \phi$ ; (2) the number of transport processes of  $D_z$  does not exceed  $max\_num$ ; (3) the diameter of  $D_z$  does not exceed  $max\_dia$ ; and (4)  $g(D_z) \leq \theta \times [g(D_x) + g(D_y)]$ , for some management cost function  $g$ , and relaxation parameter  $\theta$ . Note that by choosing a different cost function  $g$  and relaxation parameter  $\theta$ , we can reduce a different management overhead after merging.

When  $\theta \leq 1$ , requirement (4) guarantees that we always reduce management overhead on a certain aspect (based on the choice of cost function  $g$ ) after merging two VTNs. However for some  $\theta > 1$ , our design algorithm may still be able to reduce the total management overhead. We find experimentally that for some  $\theta > 1$ , we may have a larger overhead after one step of merging, but after several more steps of further merging, we may end up with less overall management overhead. In many cases, a local optimum at each stop may not yield a global optimum.

As an example, we can let  $g(D_z) = |V_z|^2$ , which can capture the routing overhead discussed in Section 3.2.3, then the fourth requirement is given by  $|V_z|^2 \leq \theta \times (|V_x|^2 + |V_y|^2)$ , *i.e.*, the square of the number of processes in the merged VTN  $D_z$  must be smaller than the sum of the square of the sizes of the two given VTNs multiplied by some relaxation parameter  $\theta$ .

Figure 4.10 and Figure 4.11 show two simple examples for the optimization step, where the cost function in requirement (4) on merging,  $g(D_z)$ , is set to  $|V_z|^2$ , and the relaxation parameter  $\theta = 1$ . Figure 4.10 has two VTNs at the same level (Figure 4.10(a)), however they cannot be merged due to violating the *max\_dia* constraint (Figure 4.10(b)). Figure 4.11 also has two VTNs at the same level (Figure 4.11(a)), and they can be merged

into a new VTN (Figure 4.11(b)).

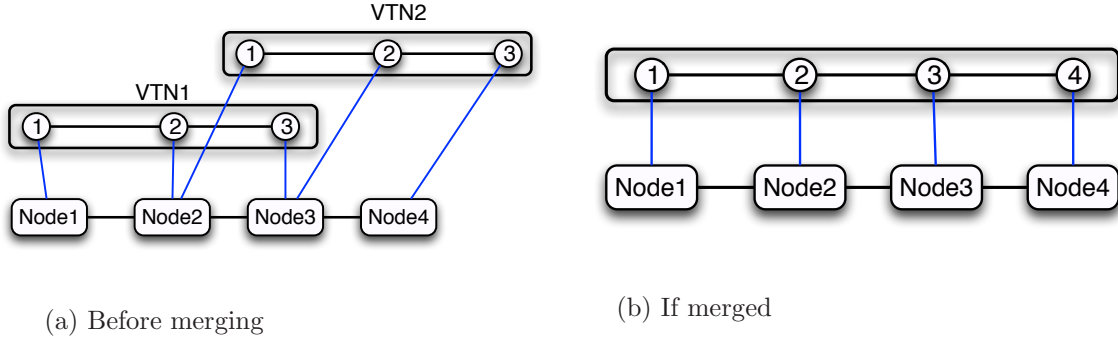


Figure 4.10: For  $max\_dia = 2$  and  $max\_num = 5$ ,  $VTN1$  and  $VTN2$  cannot be merged as the diameter of the merged VTN (*i.e.*, 3) exceeds the  $max\_dia$ .

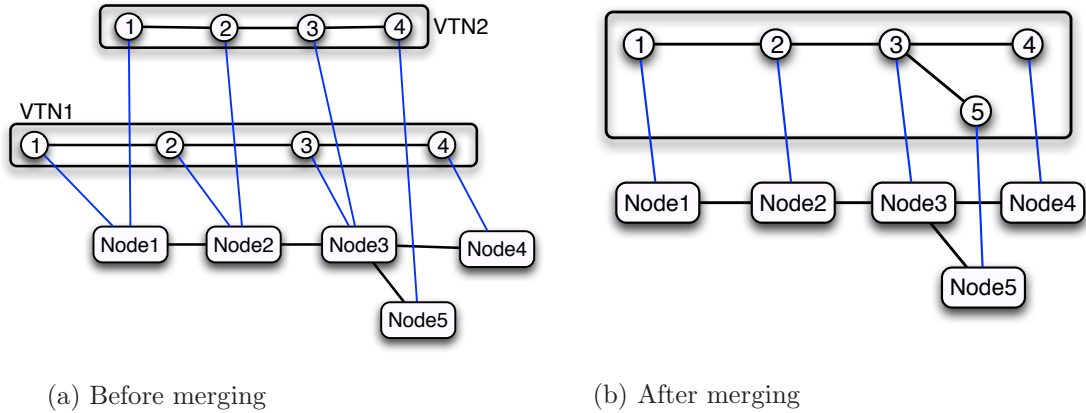


Figure 4.11: For  $max\_dia = 3$  and  $max\_num = 5$ ,  $VTN1$  and  $VTN2$  can be merged into a single VTN.

#### 4.4.4 General Algorithm for Constrained VTN Design

In Algorithm 6, we show the general algorithm for the constrained VTN design stage. The inputs include: (1) the set of flow requests (*i.e.*,  $F$ ), (2) level-0 VTNs (*i.e.*,  $D_{level(0)}$ ), which only contains level-0 VTNs at the beginning, (3) the cost function for solving the VTN packing problem (*i.e.*,  $g$ ), (4) set of all possible values for the maximum diameter allowed in a VTN (*i.e.*,  $\{max\_dia\}$ ), (5) set of all possible values for the maximum number of transport processes allowed in a VTN (*i.e.*,  $\{max\_num\}$ ), and (6) set of all possible

relaxation parameters for solving the VTN packing problem (*i.e.*,  $\{\theta\}$ ).

In Algorithm 6, the design solution (*i.e.*,  $Design$ ) is initially set to be the single-layer design (line 1), and the minimum management cost (*i.e.*,  $MinCost$ ) is set to the cost of the single-layer design for the given cost function  $g$  (line 2). Then by choosing all different possible values for (1) maximum diameter allowed in a VTN (*i.e.*,  $max\_dia$ ), (2) maximum number of transport processes allowed in a VTN (*i.e.*,  $max\_num$ ), and (3) relaxation parameter (*i.e.*,  $\theta$ ), we may obtain different VTN structures with different management overhead after finishing the VTN design stage (lines 6 – 10), then we compute the management cost for the current VTN structure (*i.e.*,  $D$ ). If the cost is less than the current minimum cost, then we choose the current design (lines 11 – 14). In other words, if our multi-layer design algorithm cannot find a solution that is better than the single-layer design, we use the single-layer design as the final solution, which guarantees our design solution is no worse than the single-layer design.

---

**Algorithm 6**  $Constrained\_VTN\_Design(F, D_{level(0)}, g, \{max\_dia\}, \{max\_num\}, \{\theta\})$

---

```

1:  $Design = Single\_Layer\_Design$ 
2:  $MinCost = Compute\_Cost(Design, g)$ 
3: for all  $max\_dia$  do
4:   for all  $max\_num$  do
5:     for all  $\theta$  do
6:        $D = D_{level(0)}$ 
7:        $D = Initial\_VTN\_Design(F, D, max\_dia)$ 
8:       for all  $D_{level(n)} \in D$  ( $n > 0$ ) do
9:          $D_{level(n)} = VTN\_Packing(D_{level(n)})$  with  $max\_dia, max\_num, g,$  and  $\theta$ 
10:      end for
11:      if  $Compute\_Cost(D, g) < MinCost$  then
12:         $Design = D$ 
13:         $MinCost = Compute\_Cost(D, g)$ 
14:      end if
15:    end for
16:  end for
17: end for
18: return  $Design$ 

```

---

In summary, our multi-layer design approach first builds an initial VTN structure based on the path selected for each flow request, then optimize the VTN structure by solving the VTN packing problem at each layer. Note that we may not always need to build new VTNs to serve new flow requests. In other words, for the online case of serving new flow

requests, we can expand existing VTNs after we have an existing VTN structure, and we can reduce the overall time needed to satisfy flow requests.

#### 4.4.5 Performance Evaluation

In this section, we present the simulation results of this case study, and demonstrate its advantages by looking at the routing overhead and transport overhead. For our experiments, we again use BRITE [47] to generate an enterprise network (50 nodes and 100 undirected links) using the Waxman model (where  $\alpha = 0.15$  and  $\beta = 0.2$ ), and the diameter of this network is 6 hops. Also for merging, we set the cost function  $g(D_z)$  to  $|V_z|^2$  and relaxation parameter  $\theta$  to 1, to reduce the routing overhead when solving the VTN packing problem (cf. Section 4.4.2.2).

##### 4.4.5.1 Routing Overhead

In this section, we look at different communication patterns over the network, and analyze the routing overhead of our multi-layered approach compared to the traditional single-layer approach. Note that the path selected for each flow request uses its shortest path.

For our multi-layer design algorithm, we try different values of  $max\_dia \in [2, 6]$ . For each fixed  $max\_dia$ , we try different values of  $max\_num \in [max\_dia, 12]$ . As discussed in Section 3.2.3, we have the LSU messages propagated less frequently at the higher-level VTNs compared to the lower-level VTNs, to allow lower-level VTNs to adapt internally (if possible) to significant link-state changes and avoid triggering LSU updates at the higher-level VTNs, and we can still achieve the same performance, *i.e.*, the time needed to detect link changes in the network. But in our experiments, we assume LSU messages are exchanged at the rate of one packet per second for level-1 VTNs, and the LSU frequency for level- $n$  ( $n \geq 2$ ) VTNs is also set the same as level-1 VTNs. Namely, we use the same update frequency at all levels as a worst case, *i.e.*, to obtain an upper bound on the routing overhead. Then we compute the total routing overhead as the summation of processing overhead for each VTN (ignoring all level-0 VTNs for each physical wire).

Besides the routing overhead, we also analyze the cost of our multi-layer design approach, *i.e.*, the average number of transport processes created per node (including level-0 VTNs which are the same for the single-layer design as well as all multi-layer designs, *i.e.*, one level-0 VTN for each physical link). For the single-layer design, each node has an average degree of two, *i.e.*, it has two transport processes in two level-0 VTNs, in addition to one process for the single-layer VTN spanning the whole network (50 nodes), thus for the single-layer design, the average number of transport processes per node is 3.

### **Experiment (1): Uniform Distribution and Uniform Distribution with Flow Length Constraint**

In the first experiment, we look at three communication patterns, and for each pattern we generate 5 sets of flow requests, *i.e.*, 100 flows, 200 flows, 300 flows, 400 flows and 500 flows. We run each experiment 10 times, and compute the mean and 90% confidence interval for each metric.

For the first pattern, we uniformly generate flow requests between any pair of nodes, and each flow is identified by the pair of source node and destination node. For the second and third pattern, we still uniformly generate flow requests, but the length of each flow on the physical topology (*i.e.*, level-0 topology) is less than a certain threshold. In the second pattern, flow length is less than or equal to 2 physical hops, and in the third pattern, flow length is less than or equal to 3 physical hops.

Figure 4.12 shows the comparison of total routing overhead between multi-layer (upper-bound) and single-layer design for 3 different communication patterns. Note that the single-layer design is the same for all 3 patterns (*i.e.*, one VTN spanning all 50 nodes), so their costs are equal and shown using the same black line. We can see that our multi-layer approach is better than (or equal to) the single-layer approach for all 3 communication patterns, this is because our design algorithm (Algorithm 6) guarantees that if the single-layer has less routing overhead than the multi-layer design, we use the single-layer design. Also we can see that, when nodes are more likely to communicate with other nodes that

are closer, our multi-layer approach performs better. What’s more, the less flow requests, the better our approach performs.

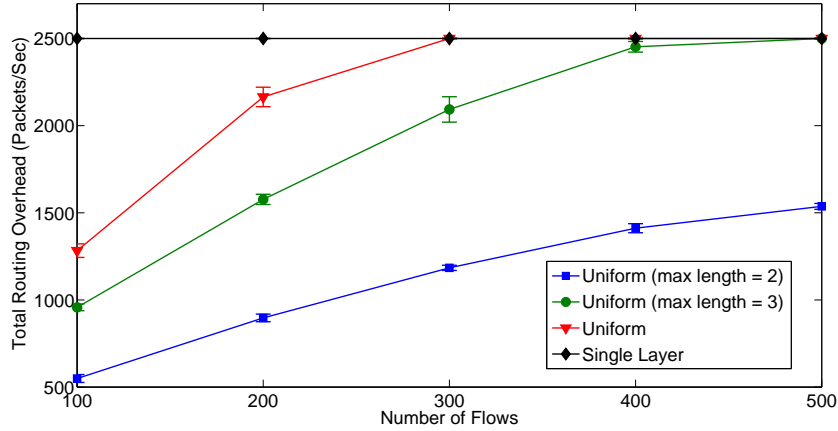


Figure 4.12: Comparison of total routing overhead (mean with 90% confidence interval) between multi-layer (upper-bound) and single-layer design for 3 different communication patterns.

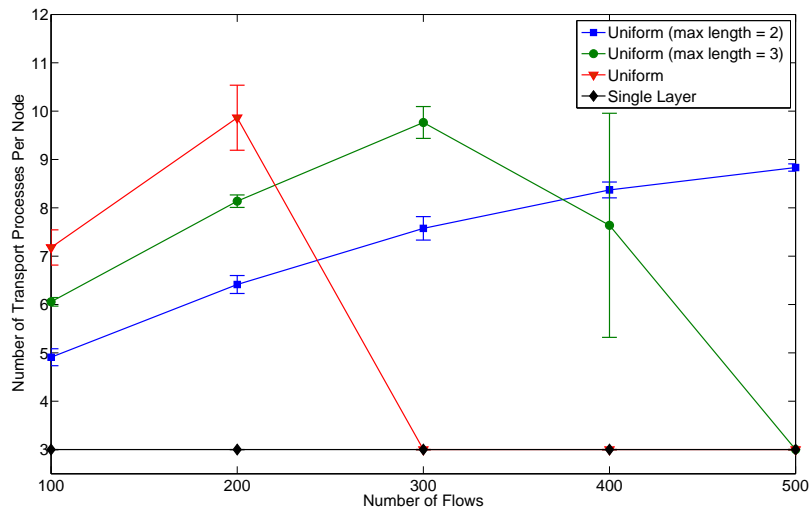


Figure 4.13: Comparison of average number of transport processes per node (mean with 90% confidence interval) between multi-layer and single-layered design for 3 different communication patterns.

Figure 4.13 shows the comparison of average number of transport processes created

per node between the multi-layer and single-layer design for 3 different communication patterns. The single-layer design is the same for all 3 patterns (*i.e.*, 3 transport processes per node), so their costs are equal and shown using the same black line. We can see that our multi-layer approach achieves better routing overhead at the cost of creating more transport processes on the nodes in the network. Note that the red line (uniform distribution) and the green line (uniform distribution with maximum flow length 3) both drop to 3 (*i.e.*, same as the single-layer design) because our design algorithm (Algorithm 6) guarantees that if the single-layer has less overhead than the multi-layer design, we use the single-layer design.

### **Experiment (2): Skewed Distribution and Skewed Distribution with Preference**

In the second experiment, we look at two patterns of skewed distributions. In this experiment, we have a set of hotspot nodes in the network, and a set of user nodes talking to these hotspot nodes. We try different values for the number of randomly generated hotspot nodes (5, 10 and 15), and different values for the number of randomly generated user nodes (20, 25, 30 and 35). Assume each user node only talks to one hotspot node, so the flow number for each setting is equal to the number of user nodes.

In the first pattern, each user node randomly picks a hotspot node (out of all hotspot nodes) to contact. In the second pattern, each user node only picks the hotspot node that has the closest distance (in physical hops) to it. Again we run each experiment 10 times, and compute the mean and 90% confidence interval for each metric.

Figure 4.14 and Figure 4.15 show the comparison of the savings in routing overhead and the average number of transport processes created per node for different number of hotspot nodes and different number of user nodes compared to the single-layer design for the skewed distribution. We can see that as we have more user nodes (*i.e.*, more flow requests), the saving under our multi-layer design decreases, and the average number of transport processes created per node increases, as we need more VTNs to serve more flow requests. This is the same as our observation in Experiment (1), the less flow requests, the

better our multi-layer approach performs.

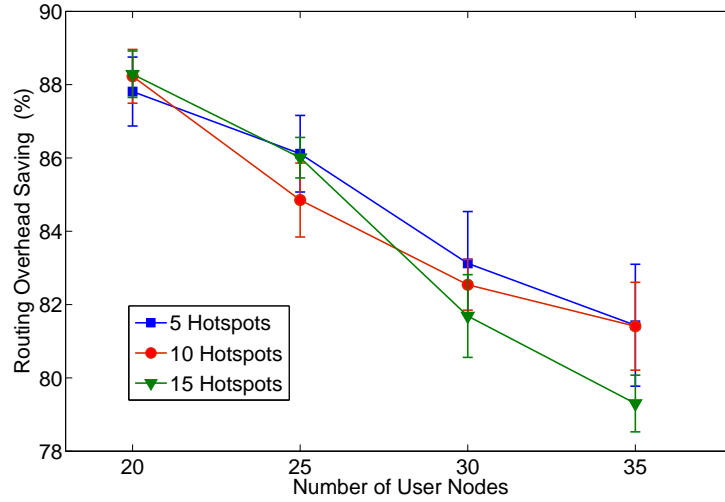


Figure 4.14: Savings (mean with 90% confidence interval) in total routing overhead compared to single-layer approach for the skewed distribution, where each user node randomly picks a hotspot node.

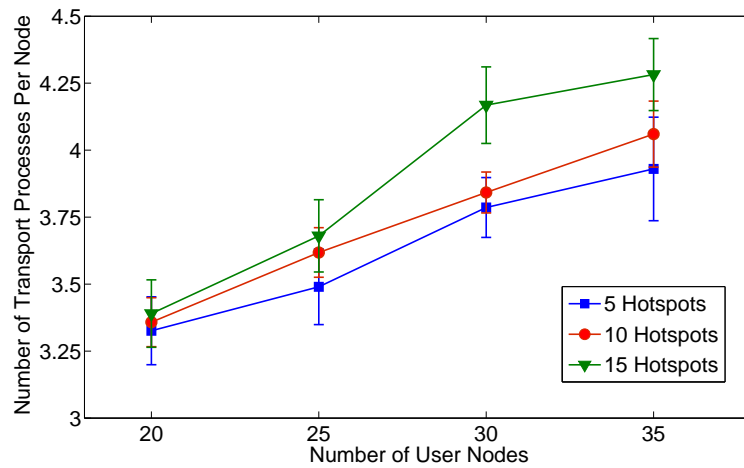


Figure 4.15: Average number of transport processes per node (mean with 90% confidence interval) compared to single-layer approach for the skewed distribution, where each user node randomly picks a hotspot node. Note that the number for single-layer design is 3.

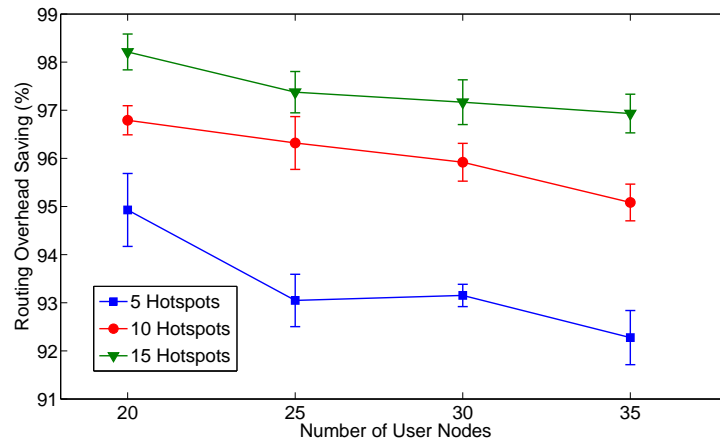


Figure 4.16: Savings (mean with 90% confidence interval) in total routing overhead compared to single-layer approach for the skewed distribution with preference, where each user node picks the closest hotspot node.

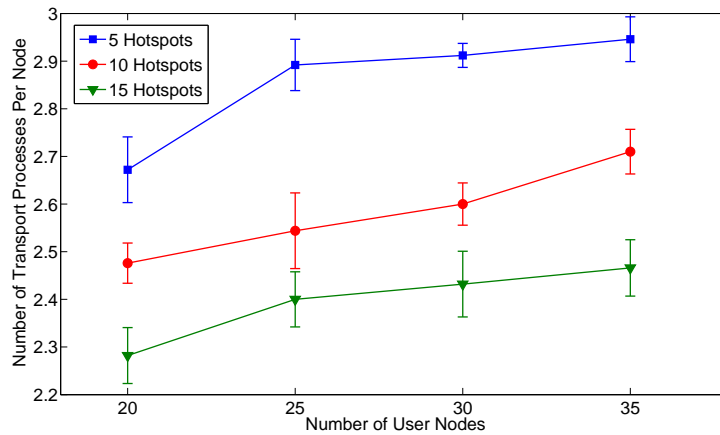


Figure 4.17: Average number of transport processes per node (mean with 90% confidence interval) compared to single-layer approach for the skewed with preference, where each user node picks the closest hotspot node. Note that the number for single-layer design is 3.

Figure 4.16 and Figure 4.17 show the comparison of the savings in routing overhead and the average number of transport processes created per node for different number of hotspot nodes and different number of user nodes compared to the single-layer approach for the skewed distribution with preference. We can see that for the same number of user nodes,

the more hotspot nodes in the network, the better our multi-layer approach performs in routing overhead. This is because each user node is more likely to pick a hotspot node that is closest to it when there are more randomly generated hotspot nodes in the network, and thus the flow request can be served by VTNs of smaller size and less levels, which yields less average number of transport processes created per node. Again we can see that the less flow requests, the better our multi-layer approach performs.

### Experiment (3): Parameter Sensitivity Analysis

In the third experiment, we randomly generate 123 unique pairs of source and destination (out of all 1225 possible unique pairs), *i.e.*, 123 unique flow requests, and analyze how different values of *max\_dia* and *max\_num* may affect the performance of our multi-layer design by looking at the total routing overhead and the average number of transport processes created per node. We run each experiment 10 times to compute the mean for each metric.

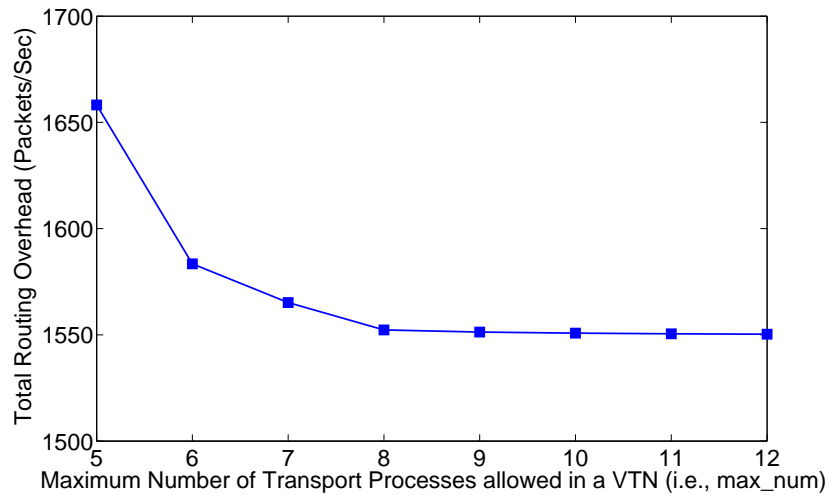


Figure 4.18: Total routing overhead (mean of 10 runs) for different values of *max\_num* when *max\_dia* = 5, while the cost for single-layer design is 2500.

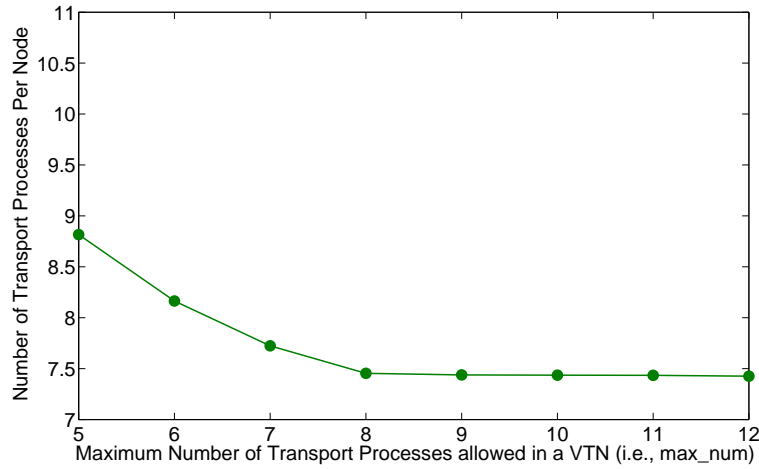


Figure 4.19: Average number of transport processes per node (mean of 10 runs) for different values of  $max\_num$  when  $max\_dia = 5$ , while the number for single-layer design is 3.

Figure 4.18 and Figure 4.19 show the total routing overhead and average number of transport processes created per node for different values of  $max\_num$  when  $max\_dia = 5$ . As the  $max\_num$  increases, they both decrease in the beginning, but then flatten out. They decrease in the beginning because by allowing more transport processes (bigger  $max\_num$ ) in the VTN, we can merge more VTNs in the optimization step. However, they eventually flatten out because we cannot further merge the VTNs due to the constraint of  $max\_dia$ .

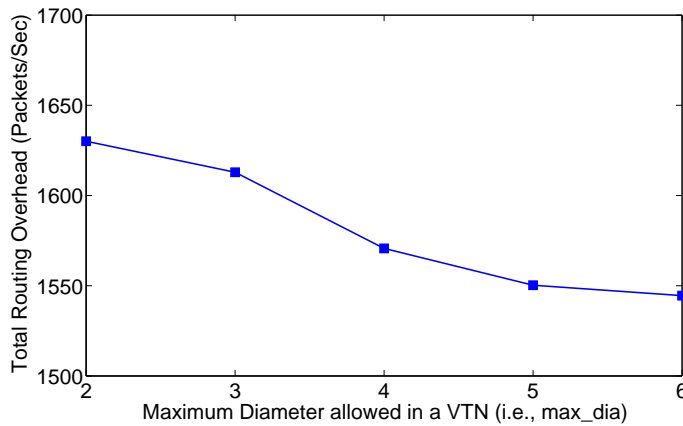


Figure 4.20: Total routing overhead (mean of 10 runs) for different values of  $max\_dia$  when  $max\_num = 12$ , while the cost for single-layer design is 2500.

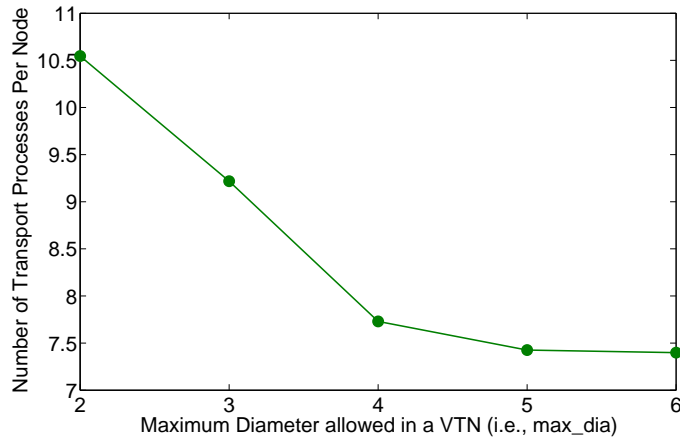


Figure 4.21: Average number of transport processes per node (mean of 10 runs) for different values of  $max\_dia$  when  $max\_num = 12$ , while the number for single-layer design is 3.

Figure 4.20 and Figure 4.21 show the total routing overhead and average number of transport processes created per node for different values of  $max\_dia$  when  $max\_num = 12$ . As  $max\_dia$  increases, we have less routing overhead and less number of transport processes. The reason is that we are not bounded by the constraint of  $max\_dia$ , and we can merge more VTNs, which leads to less number of transport processes.

#### 4.4.5.2 Transport Overhead

In this section, we look at the transport overhead. Assume each of the 100 physical links in our experiment network has a loss rate of 10% on both directions. We compare the transport overhead, *i.e.*, average number of transmissions needed in order to successfully deliver a packet. In our experiment, we analyze all flow requests (out of all possible 1225 flow requests) whose selected path (shortest path) is longer than 3 hops, and there are a total of 327 of such flow requests. We assume each flow has an infinite supply of packets to send, and we compute the average number of transmissions for successfully delivering one packet from each of these 327 flows.

As shown in Figure 4.22, we can see that our multi-layer design has less transport overhead compared to the single-layer design (which is one network spanning all 50 hosts).

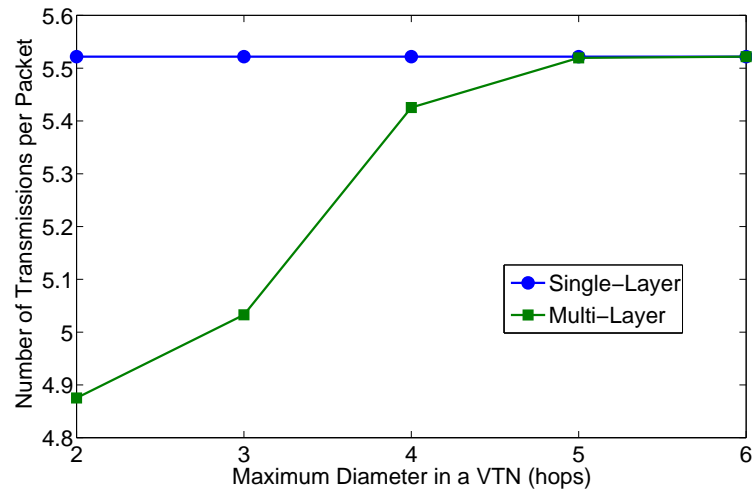


Figure 4.22: Average number of transmissions for one successful packet delivery for different values of  $max\_dia$ .

For the multi-layer design, as the diameter allowed for a VTN increases, the transport overhead also increases. This is because as we break a TCP connection into less number of reliable segments, each segment is still a relatively long TCP connection, *i.e.*, smaller  $m$  in Equation (3.9) (Section 3.2.4). When  $max\_dia$  is the same as the diameter of the network (*i.e.*, 6), multi-layer has the same transport cost as the single-layer design. Namely, when  $m = 1$ , Equation (3.9) is the same as Equation (3.8) (Section 3.2.4).

## Chapter 5

# Design of VTN-based Management Architecture

In this chapter, we explain the design of our VTN-based management architecture which enables the VTN-based network management. First, we present the management components, and external API provided by our management architecture, as well as the details of VTN formation protocol which enables the dynamic VTN formation, *i.e.*, realizing the output of the multi-layer VTN design problem explained in Chapter 4. Then, we explain design of other components and internal API of our management architecture. This thesis extends the RINA specification [57] to include the allocation of multi-layered VTNs.

## 5.1 Management Components

In this section we describe the management components of our VTN-based management architecture. To this end, we realize the management function through distributed management applications. Next we explain the components (*i.e.*, distributed applications) for managing (1) a single VTN; and (2) all VTNs.

### 5.1.1 VTN Manager and VTN Manager Agent

As shown in Figure 5.1, the distributed application for managing a single VTN includes a *VTN Manager* and its *VTN Manager Agents*.

Every VTN has a VTN manager, which is a process that can be implemented in a centralized or distributed fashion, and it manages the whole VTN by specifying different network policies inside the VTN such as routing, access control, and transport policies. Also

VTN manager is responsible for authentication and enrollment of new transport processes into the VTN. A VTN manager agent is part of each transport process inside the VTN, and it exposes a programmable interface for the VTN manager to translate high-level network policies to transport process's configurations.

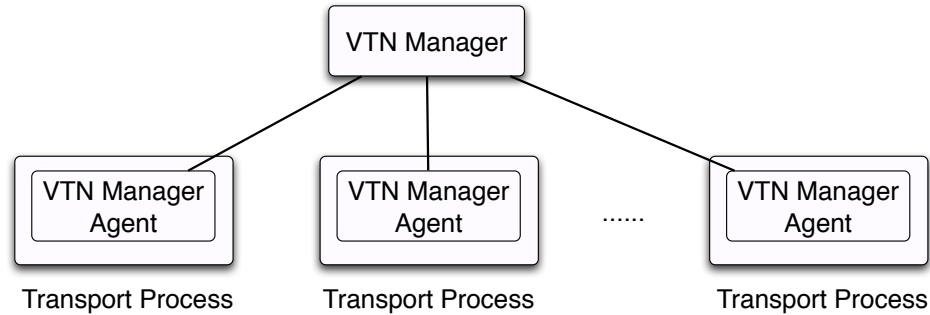


Figure 5.1: VTN Manager and its agents for a single VTN.

### 5.1.2 VTN Allocator and VTN Allocator Agent

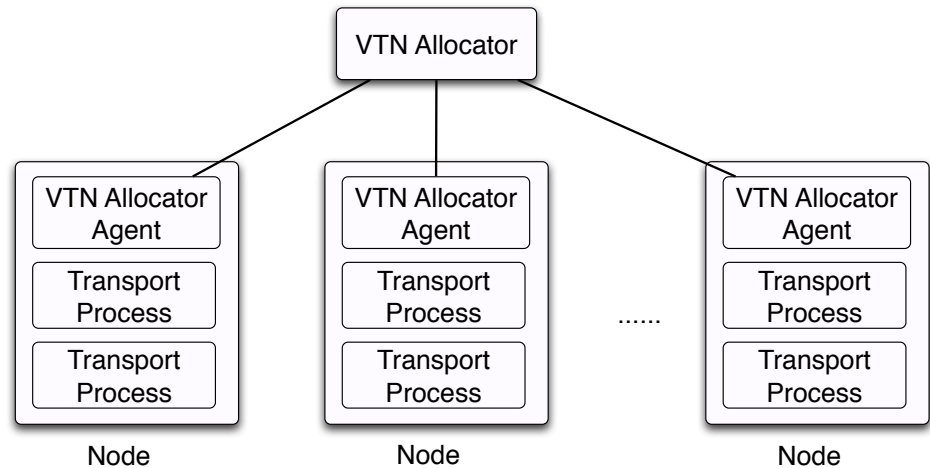


Figure 5.2: VTN Allocator and its agents for an enterprise network.

As shown in Figure 5.2, the distributed application for managing all VTNs inside the enterprise network includes a *VTN Allocator (VA)* and its *VTN Allocator Agent (VAA)*.

The network has one VA, which is a management process that can be implemented in a centralized or distributed fashion, and it manages all VTNs as a whole. Each node (host) inside the network has a VAA, which exposes a programming interface allowing the VA to create new transport processes on the node and thus build new VTNs across multiple nodes within the network. The VA manages the network by managing existing VTNs and building new VTNs dynamically to support different application flow requests.

Fault tolerance of the VTN Manager and VTN Allocator is beyond the scope of this thesis and left for future work. Consistency for distributed implementation can be enabled by implementing any of the required distributed locking and consistency algorithms, and also out of the scope of this thesis.

### **5.1.3 VTN Resource Manager (VRM)**

In our approach, every application process (including management application processes, transport processes and regular user application processes) has a component called *VTN Resource Manager (VRM)*, which manages the use of all VTNs available to this particular process. It is the job of VAA of the node to decide which VTNs are accessible to particular processes. An application process uses its VRM to allocate transport flows with QoS requirements to other processes, and the VRM in turn passes the flow allocation requests to VTNs via the interface exposed by VTN.

### **5.1.4 Walk-through of Transport Flow Allocation**

Next we walk through the process of transport flow allocation. When an application wants a transport flow with a certain QoS requirement to another application process, it uses the flow allocation interface exposed by its VRM. When the VRM gets the request, it first checks whether any of its available VTNs can reach that application. If the VRM finds such a VTN, it uses the VTN interface to allocate the flow through the transport process belonging to that VTN on the same node. If no VTN is found, the VRM sends the flow request to the VAA of the node, which then forwards the flow request to the VA of the

network, and eventually the VA determines how to build a new VTN which consists of new transport processes running on the source node, destination node and some intermediate nodes.

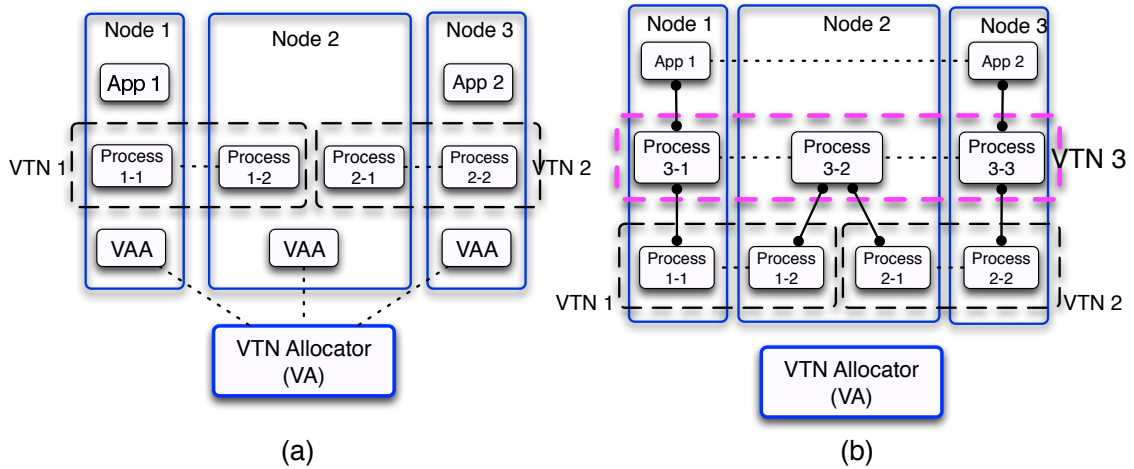


Figure 5.3: (a) An enterprise network consists of three nodes (*Node 1*, *Node 2* and *Node 3*), and a centralized VTN Allocator. (b) A new VTN (*VTN 3*) is formed to support the flow between *App 1* and *App 2*. VAA of each node is not shown in (b).

As shown in Figure 5.3(a), when *App 1* on *Node 1* asks its VRM (not shown) for a flow to *App 2*, and its VRM cannot find an existing VTN to reach *App 2*, *App 1*'s VRM then sends the request to *Node 1*'s VAA, which forwards the request to the VA. The VA figures out that a new VTN is needed to support the flow, then it builds a new VTN (*VTN 3*) spanning all three nodes (Figure 5.3(b)). After the new VTN is ready, the VAA of *Node 1* notifies the VRM of *App 1*, which eventually uses this new VTN (*VTN 3*) to create a flow to *App 2*. Note that links between processes of *VTN 3* constitute *virtual transport links* and are not simply routing tunnels.

## 5.2 VTN Formation Protocol

New VTNs may need to be formed in support of transport flows (Section 5.1.4). In this section, we explain how the VTN Allocator (VA) and VTN Allocator Agent (VAA) interact with each other via a VTN formation protocol to create new VTNs.

## 5.2.1 Objects Exchanged in the Protocol

The key aspect of the VTN formation protocol is two objects exchanged between the VA and VAA, one is the flow request object and the other one is the VTN request object.

### 5.2.1.1 Flow Request Object

The VAA on a node sends a flow request object to the VA when a certain transport flow cannot be supported using existing VTNs on the node. The flow request object specifies the source and destination application information as well as QoS requirements including throughput, delay, and loss rate. The flow request object also supports advanced flow requirements (policies) such as which nodes to bypass or go through, or whether encryption is needed or not. The flow policies inside the request object are specified when the application uses the Flow Allocation API (shown in Table 5.1 (2)) exposed by its VRM to allocate the transport flow.

### 5.2.1.2 VTN Request Object

The VTN request object supports two operations: VTN creation and deletion. A VA sends a VTN request object to multiple VAAs on different nodes once it determines how the new VTN should be formed, *i.e.*, the new VTN should have new transport processes running on which nodes. The VTN request object specifies policies for the new VTN, including policies for addressing, routing, error and flow control, *etc.* Also it supports other policies such as which application can use this VTN, the lifetime of this VTN as well as resource allocation policies. Furthermore, the VTN request object can specify the connectivity among transport processes, and the enrollment and authentication policies for new transport processes to join the VTN.

### 5.2.2 Protocol Details

A new VTN needs to be formed when existing VTNs cannot satisfy a new transport flow request. Next we explain how the VTN formation protocol works step by step. When the VA receives a flow request object from a VAA, it first inspects the object to see if it is a valid request. If valid, it checks the network state and determines whether there exists a path (a chain of nodes) from the node where the source application runs to the node where the destination application runs. Once a path is found, the VA can decide which nodes the new VTN should span, *i.e.*, the design of the VTN. Namely, this procedure (including finding a path and designing the new VTN) is to solve the *Multi-layer VTN Design Problem* discussed in Chapter 4.

Once the VA determines the design of the new VTN, it sends the VTN request object specifying the policies of the new VTN to all VAAs of the nodes along the path (including source and destination nodes). When a VAA receives the VTN request, it first inspects the object to see if it is a valid request. If valid, the VAA creates a new transport process as a member of the new VTN, then the VAA sends a VTN response to the VA indicating that the new transport process on this node is ready. After the VA receives the responses from all VAAs to which it sent the VTN request, and if all responses indicate that all new transport processes are ready, the VA sends a flow response to the VAA that sent the initial flow request indicating that a new VTN is ready and the associated VRM can use it to create the transport flow. If any of the VAA's responses indicates failure of the creation of the new transport process, the VA sends a VTN delete request to all other VAAs to delete the newly created transport processes for that VTN. Then the VA sends a negative flow response to the VAA that asked for the new transport flow indicating that the flow request cannot be satisfied.

### 5.3 Network API

Our management architecture provides two sets of APIs: (1) the management-level API, used by network managers to manage the network (*i.e.*, writing management applications) by programming the VTN Allocator and VTN Manager; and (2) the user-level API, used by regular users to program their own applications (*i.e.*, writing user-defined applications), and it helps users affect their application traffic to improve user application performance.

#### 5.3.1 Management-level API

Network managers manage networks through management applications, and our management-level API includes three sets of APIs as shown in Table 5.1.

<b>(1) VTN Formation API</b>
public boolean createVTN (VTNRequest vtnRequest); public boolean deleteVTN (VTNRequest vtnRequest);
<b>(2) Flow Allocation API</b>
public int allocateFlow(Flow flow); public boolean deallocateFlow(int handleID); public void send(int handleID, byte[] msg) throws Exception; public byte[] receive(int handleID);
<b>(3) Information API</b>
public int createEvent(SubscriptionEvent subscriptionEvent); public boolean deleteEvent(int subscriptionID); public Object readSub(int subID); public void writePub(int pubID, byte[] obj);

Table 5.1: Three sets of management-level APIs in Java.

##### 5.3.1.1 VTN Formation API

VTN Allocator Agents (VAAs) expose the VTN Formation API (shown in Table 5.1 (1)), which is used by the VTN Allocator (VA) to create new VTNs or delete existing VTNs. A VTN is instantiated with different policies such as routing policies, addressing policies

and flow and error control policies.

### 5.3.1.2 Flow Allocation API

The VRM of each application process (cf. Section 5.1.3) exposes the Flow Allocation API (shown in Table 5.1 (2)), which is used to create/delete transport flows with QoS requirements as well as to send/receive data messages over existing flows between applications.

### 5.3.1.3 Information API

Based on a publish/subscribe model (a pulling mechanism to retrieve information is also supported), the Information API (shown in Table 5.1 (3)) allows management applications to retrieve/publish network information (such as existing transport flows' status or VTN information on other nodes) from/to other management applications. This API (similar to NIB API in Onix [41]) allows management applications to access network information.

## 5.3.2 User-level API

The user-level API includes two sets of APIs: (1) the Flow Allocation API; and (2) the Information API. They are the same as the ones in the management-level API.

<b>(1) Flow Allocation API</b>
<pre>public int allocateFlow(Flow flow); public boolean deallocateFlow(int handleID); public void send(int handleID, byte[] msg) throws Exception; public byte[] receive(int handleID);</pre>
<b>(2) Information API</b>
<pre>public int createEvent(SubscriptionEvent subscriptionEvent); public boolean deleteEvent(int subscriptionID); public Object readSub(int subID); public void writePub(int pubID, byte[] obj);</pre>

Table 5.2: Two sets of user-level APIs in Java.

Users use these two APIs to write their own applications, where they use the Flow Allocation API create/delete transport flows with QoS requirements as well as to send/receive data messages over existing flows between user-defined applications; and they use the Information API to retrieve or publish user-specific information between user-defined applications.

## 5.4 Node Components

Next we explain the design of other components in our management architecture. In our management architecture, a *node* is a container where *application processes* (Section 5.4.2) and *transport processes* (Section 5.5) reside, and a *node process* is running on each physical node.

### 5.4.1 Node Process

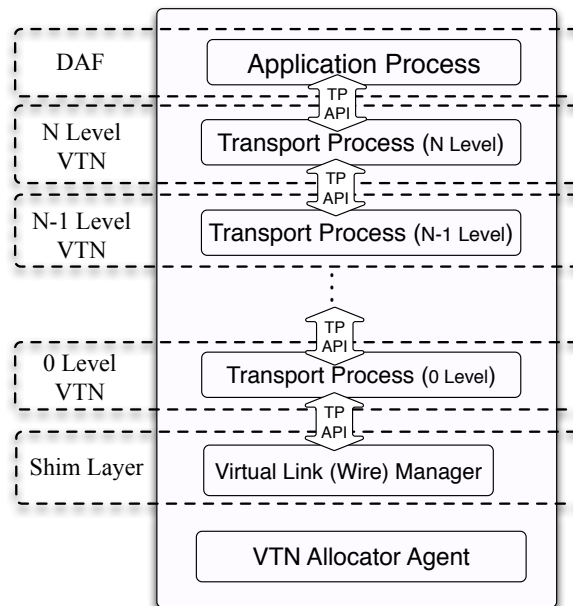


Figure 5.4: Components of a node process.

As shown in Figure 5.4, application processes or high-level transport processes communicate with their peers using the communication service provided by underlying low-level transport processes, which expose the *Transport Process (TP) API* (details in Table 5.4) and act as points of attachment. The mapping from an application process (or higher-level transport process) to the lower-level transport process is resolved by the underlying VTN. Note that in Figure 5.4, we only show one application process and one transport process at each level, however, practically there might be multiple application processes on the same node, and multiple transport processes of the same level.

Physical connectivities between transport processes in level-0 VTNs are emulated by a shim layer. More generally, using the shim layer enables our management architecture running on top of Ethernet, TCP, UDP, or SDN-based networks. Details can be found in Section 5.4.3.

As discussed in Section 5.1.2, the enterprise network has a VTN Allocator (VA), and each node has a VTN Allocator Agent (VAA), which exposes a programming interface allowing the VA to create new transport processes on the node and thus build new VTNs across multiple nodes within the network.

#### 5.4.2 Application Process

A *Distributed Application Facility (DAF)* [57, 80] is a collection of distributed application processes with shared states. Each DAF performs a certain function such as video streaming, weather forecast or communication service. Different DAFs use the same mechanisms but they may use different policies for different purposes and over different scopes. For example, a *VTN*, *i.e.*, a collection of transport processes, is a special DAF whose job is only to provide communication services for application processes. Also the management components for managing a single a VTN (or an enterprise network) form a management DAF whose job is to manage a single VTN (or an enterprise network).

The *Common Distributed Application Protocol (CDAP)* [57] is the only application protocol required, and it is used for both network management applications and regular

user applications. For example, the objects exchanged for the VTN formation protocol are encapsulated in the CDAP messages. CDAP defines a set of operations, create/delete, read/write, and start/stop on remote objects, and connect/release to enable authentication and coordination among management applications. Users are free to define new types of object for their own application purposes, as long as instances of such application agree on the same data representation.

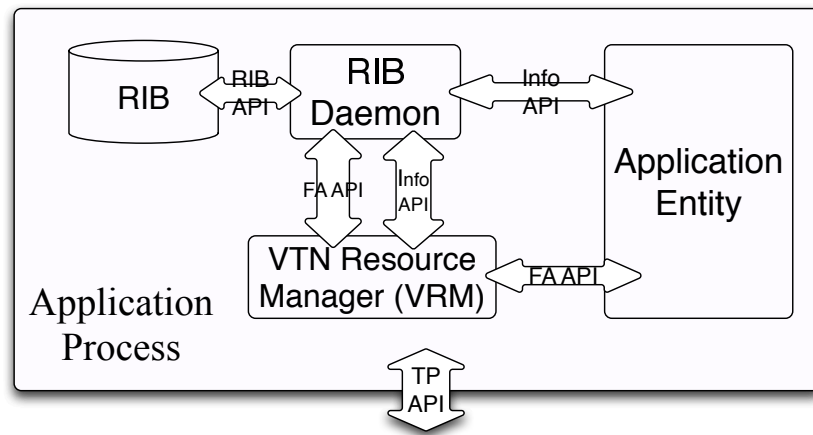


Figure 5.5: Components of an application process.

Figure 5.5 shows the common components of an application process. The *Resource Information Base (RIB)* is the database that stores all information related to the operations of an application process, and RIB can be accessed via *the RIB API* (details in Table 5.3). The *RIB Daemon* helps other components of the application process access information stored in the local RIB or in a remote application’s RIB. As discussed in Section 5.1.3, each application process also has an *VTN Resource Manager (VRM)*, which manages the use of underlying transport processes belonging to lower-level VTNs that provide communication services for this application process, and VRM manages underlying VTNs via the *Transport Process (TP) API* (details in Section 5.5.3) exposed by each transport process. The *Application Entity* is the container in which users can implement different management or application-specific functionalities.

```

public void addAttribute(String attributeName, Object attribute);
public Object getAttribute(String attributeName);
public boolean updateAttribute(String attributeName, Object attribute);
public void removeAttribute(String attributeName);

```

Table 5.3: RIB API in Java.

As discussed in Section 5.3, the *Flow Allocation (FA) API* and *Information API* are provided for users to write management (or regular) applications and to support new network management policies. The Information API (provided by the RIB Daemon) is based on a publish/subscribe model, which supports the creation and deletion of a subscription event (a *Pub* or *Sub* event), the retrieval of information through a *Sub* event, and the publication of information through a *Pub* event. The RIB Daemon also supports the traditional pulling mechanism to retrieve information. The Flow Allocation (FA) API (provided by the VRM) allows allocating/deallocating a connection (transport flow) to other application processes, and sending/receiving messages over existing connections.

### 5.4.3 Shim Layer Process

In order to deploy our VTN-based management architecture on top of legacy networks (such as over Ethernet, TCP or UDP), we have a shim layer in our design. Physical connectivity between transport processes at level 0 are emulated by the shim layer, and the shim layer includes functionalities such as resolving a user-defined level-0 transport process name to an IP address and a port number. The shim layer consists of a collection of shim layer processes, *i.e.*, the virtual link (wire) manager on each node, and each virtual link (wire) manager manages the emulated physical wires for that particular node.

## 5.5 Transport Process Components

Figure 5.6 illustrates how different components of a transport process interact with other components. Note that transport process is just a special application process (Section 5.4.2)

whose job is only to provide communication services, so it has *RIB*, *RIB Daemon* and *VRM*. Recursively each transport manages the use of underlying VTN via the Transport Process (TP) API. The application entity for a transport process corresponds to two parts: (1) data transfer application entity, and (2) management application entity.

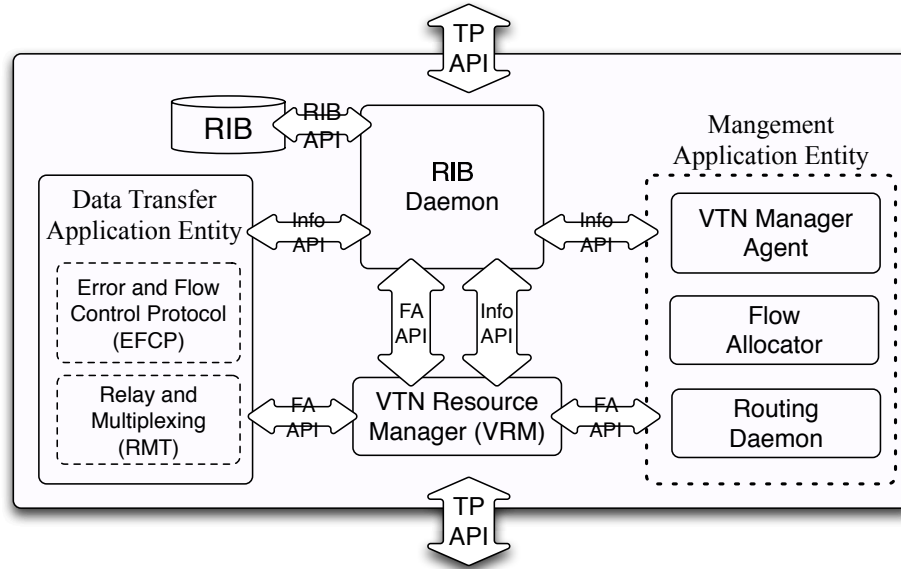


Figure 5.6: Components of a Transport Process.

### 5.5.1 Data Transfer Application Entity

The *Data Transfer Application Entity* is responsible for data transfer for each existing flow, and the Relay and Multiplexing Task (RMT). The Error and Flow Control Protocol (EFCP) [57] is the data transfer protocol used within a VTN. The functions of this protocol ensure reliability and flow control as required. The original design of the EFCP is modeled after Richard Watson's Delta-t transmission protocol [81], and includes a Data Transport Protocol (DTP) and a Data Transport Control Protocol (DTCP). When a DTP packet is received, the data transfer application entity inspects its header information, and it delivers it to the corresponding application process using this transport process based on the connection ID if the transport process is the destination. If the destination is not itself,

it checks the forwarding table and sends the packet to the next-hop toward the destination.

### 5.5.2 Management Application Entity

The *Management Application Entity* includes 3 components: (1) *VTN Manager Agent*, (2) *Flow Allocator*, and (3) *Routing Daemon*.

As discussed in Section 5.1.1, *VTN Manager Agent* is a part of each transport process inside the VTN, and it exposes a programmable interface for the *VTN Manager* to translate high-level network policies to transport process's configurations.

The *Flow Allocator* is the component that is responsible for the *Transport Process (TP) API* invocation from application processes (or higher-level transport processes), and maintains every transport flow allocated by this transport process.

The *Routing Daemon* is responsible for the routing inside the VTN, so transport processes are able to talk to other members in the same VTN.

### 5.5.3 Transport Process (TP) API

The *Transport Process (TP) API* is used by an application process's VRM to access a transport process: (1) via its Flow Allocator to create and delete a flow to another application process, (2) via its Data Transfer Application Entity to send and receive messages over an existing flow, (3) to register in the underlying VTN such that this application process can be reached through the transport process.

Recursively, an N-level transport process is seen as an application process which uses an (N-1)-level transport process's communication services.

<pre>public int allocateFlow(Flow flow); public void deallocateFlow(int portID); public void send(int portID, byte[] msg) throws Exception; public byte[] receive(int portID);</pre>
<pre>public void registerApplication(ApplicationProcessNamingInfo apInfo, FlowInfoQueue flowInfoQueue); public void deregisterApplication(ApplicationProcessNamingInfo apInfo);</pre>

Table 5.4: Transport Process (TP) API in Java.

## Chapter 6

# Implementation of VTN-based Management

## Architecture

In this chapter we present the implementation of our VTN-based management architecture, which enables VTN-based management on real networks. Our implementation allows not only managing an enterprise network by programming management applications but also creating new user applications by programming user-defined applications.

Our implementation [73] is at the user level, and it supports dynamic formation of VTNs and multiple management policies (*e.g.*, naming and routing policies). The current implementation, called ProtoRINA (version 2.0), consists of about 70k lines of Java code excluding support libraries and configurations. It has been tested on our BU campus network and on the GENI testbed [25]. Also in our implementation, all objects (CDAP messages, data transfer messages, and other messages) are serialized and deserialized using the Google Protocol Buffers [58].

Note that our simulation analysis in Chapter 3 and Chapter 4 shows the benefits of QoS support in our VTN-based management, but in our implementation we focus on reachability only, where a flow request is specified by the source and destination application names.

## 6.1 Key Classes

### 6.1.1 Node Process

As mentioned in Section 5.4, a *node process* is running on each physical node, and it is the container where application processes and transport processes reside. The `node` package includes the implementation for the node process and all its components.

Each node process (defined in `node.impl/RINANode.java`) has a configuration file (Section 6.3.1). When a node is initialized, transport processes and application processes on the node are bootstrapped by the node process, based on its own configuration file.

### 6.1.2 Application Process

The `application` package includes the implementation for the basic application process (details in Section 5.4.2) and its components.

Each application process (defined in `application.impl/Application.java`) can have a configuration file (Section 6.3.2) that includes all information to bootstrap it. Users can extend the `Application` class to write their own user-specific applications, and then update the bootstrap logic of the `RINANode` class so that the new application can be correctly started by the node process.

### 6.1.3 Transport Process

The `rina` package includes the implementation for the transport process (details in Section 5.5) and its components, and each transport process (defined in `rina.ipc.impl/IPCImpl.java`) has a configuration file (Section 6.3.3). Note that in our implementation, the transport process is denoted as *IPC* process, since it provides inter-process communication (IPC) service, and *VTN* is denoted as *DIF*, since a VTN is actually a Distributed IPC Facility.

In our current implementation of transport process's data transfer application entity (defined in `rina.ipc.ae/DataTransferAE.java`), we support both unicast and multicast

data transfer. By default multicast is turned off for a VTN, but it can be turned on in the transport process's configuration file. Note that since this thesis focuses on network management, only a simple version of DTP is supported, and DTCP is not yet supported.

Also all CDAP messages (such as ones used for flow allocation and information retrieval) received by a transport process are handled by its management application entity (defined in `rina.ipc.ae/ManagementAE.java`).

#### 6.1.4 VTN Manager and VTN Manager Agent

As mentioned in Section 5.1.1, every VTN has a *VTN Manager* and each transport process has a *VTN Manager Agent*. In our implementation, the VTN manager of a VTN is implemented in a distributed fashion, where the VTN manager agent of each transport process can act as a VTN manager. The function of the transport process's VTN manager agent is implemented in its management application entity (defined in `rina.ipc.ae/ManagementAE.java`).

The first member of a VTN specifies all policies (*e.g.*, addressing and routing policies) needed for the operations of the VTN. As mentioned earlier, new transport processes have to be explicitly enrolled into the VTN in order to talk to other transport processes inside the VTN, and the enrollment procedure can be done by any existing member of the VTN via its VTN manager agent. During the enrollment procedure, the existing member passes all policies to the new member through its VTN manager agent.

Next we explain how a new transport process is enrolled into a VTN by an existing member through exchanging a sequence of CDAP messages. Assume a new transport process `ProcessN` is going to be enrolled into a VTN by an existing member `Process1`.

First, `ProcessN` sends an *M\_CONNECT* message containing some authentication information to `Process1`. If the received authentication information is valid, `Process1` returns an *M\_CONNECT\_R* with a positive result to `ProcessN`. This message informs `ProcessN` that `Process1` is ready to start the enrollment. `ProcessN` then sends an *M\_START* to `Process1`, and after receiving it, `Process1` replies with an *M\_START\_R*, which contains

a dynamically generated address (private to this VTN) for the new member `ProcessN`.

After obtaining such address, the enroller `Process1` sends a sequence of `M_CREATE` messages to `ProcessN` which contain information about the current members in the VTN and the applications reachable through this VTN. Upon completion, `Process1` sends an `M_STOP` to `ProcessN` indicating the termination of all the information required for the enrollment, and `ProcessN` replies with an `M_STOP_R` to `Process1`. After receiving an `M_STOP_R`, `Process1` sends an `M_START` to `ProcessN` to confirm that the enrollment procedure is completed. `ProcessN` is now able to operate inside the VTN. Finally, `Process1` sends an `M_CREATE` message to other existing transport processes (if there are any) to inform them about the new enrolled member.

This enrollment procedure is designed to be used in an environment where transport processes might disappear for short periods of time and reappear, *i.e.*, process crashes.

### 6.1.5 VTN Allocator and VTN Allocator Agent

As discussed in Section 5.1.2, the network has one *VTN Allocator (VA)*, which manages all VTNs as a whole, and each node inside the network has a *VTN Allocator Agent (VAA)*, which exposes a programming interface allowing the VA to create new transport processes on the node and thus build new VTNs across multiple nodes within the network.

In our implementation, VA is implemented in a centralized fashion. Also note that in our implementation, *VTN Allocator (VA)* is denoted as *DIF Allocator (DA)*, and *VTN Allocator agent (VAA)* is denoted as *DIF Allocator Agent (DAA)*.

The `rina.da` package includes the implementation for VA (DA) (defined in `rina.da/DIFAllocator.java`) and other supporting components. As a component of the node process, VAA (DAA) is defined in `node.components/DIFAllocatorAgent.java`

In our implementation, a VA (DA) runs on a node in the network, and communicates with all VAAs (DAAs) in the network via TCP connections. The VA (DA) listens to a well-known IP/port, and VAA (DAA) of each node also listens to a given local IP/port. When a node process is bootstrapped, its VAA (DAA) registers the mapping of its node

name to IP/port with the VA (DA), so VA (DA) is able to reach VAA (DAA) of each node in the network.

The VA (DA) also provides a directory service (a process named *IDD*, defined in `rina.idd/IDDProcess.java`) for the network, which is responsible for VTN name and application name resolution. For a query about a VTN name, the directory service returns information of transport (IPC) processes that are responsible for enrolling new members into the VTN. For a query about an application name, the directory service returns information of the VTNs and their transport (IPC) processes which are able to help reach the application process.

In our current implementation, the directory service keeps listening to a well-known IP/port on the VA node, and it is reachable by any level-0 transport (IPC) process through the TCP shim layer. Non level-0 transport (IPC) processes and application processes communicate with the directory service using any of their underlying level-0 transport (IPC) processes on the same node. Each time a new VTN or a new application is started, they register the corresponding directory information with the directory service.

## 6.2 Supporting Classes

### 6.2.1 VTN Resource Manager (VRM)

As discussed in Section 5.1.3, every application (or transport) process has a component called *VTN Resource Manager (VRM)*, which manages the use of all transport processes available to this particular process on the same node. An application (or transport) process uses the Flow Allocation API (Section 5.3.1.2) exposed by its VRM to allocate transport flows to other processes.

In our implementation, *VTN Resource Manager (VRM)* is denoted as *IPC Resource Manager (IRM)*, since it manages all inter-process communication (IPC) resources. The VRM (IRM) of an application process is implemented in `application.component.impl/IPCResourceManagerImpl.java`, and the VRM (IRM) of a transport (IPC) process is

implemented in `rina.irm.impl/IRMImpl.java`. The main difference is that the VRM of a transport (IPC) process also manages the shim layer process which emulates physical connectivity.

### 6.2.2 RIB and RIB Daemon

As discussed in Section 5.4.2, every application (or transport) process has a *Resource Information Base (RIB)*, which is the database that stores all information related to the operations of that process. The *RIB Daemon* helps other components of the application (or transport) process access information stored in the local RIB or in a remote application (or transport) process's RIB.

In our implementation of the application process and transport process, the information stored in the local RIB (defined in `rina.rib.impl/RIBImpl.java`) is directly accessed via the *RIB API* (Table 5.3). The `rina.ribDaemon` package contains the implementation for the RIB Daemon (defined in `rina.ribDaemon.impl/RIBDaemonImpl.java`) and its components.

The RIB daemon is based on a Publish/Subscribe model and exposes the Information API (Section 5.3). It allows the creation and deletion of subscription events (*Pub* events or *Sub* events) to publish/retrieve information to/from other remote application (or transport) process's RIB at certain frequencies. The RIB daemon maintains all subscription events, including *Pub* events and *Sub* events. For each subscription event, the RIB daemon creates a corresponding event handler (defined in `rina.ribDaemon.util/EventHandler.java`) which is responsible for updating the event. For each *Sub* event to a remote process's RIB, the event handler translates the event to a CDAP *M\_CREATE* message, which contains a *Sub* event object and sends it to the remote process. For each *Pub* event, the event handler creates a publisher process (defined in `rina.ribDaemon.util/Publisher.java`) to periodically publish information to its subscribers. When a process receives a *Sub* object from another process, it adds the sender process to the subscriber list of a predefined *Pub* event (if any). When a process receives a *Pub* object (in a CDAP *M\_CREATE\_R* message)

from another process, it updates the corresponding *Sub* event.

Next we explain how to define a new set of *Pub/Sub* events in the RIB daemon, so that users can use the RIB daemon to support their own user-specific applications.

We assume that the new event's attribute name is "*newAttribute*". In the package `rina.ribDaemon.util`, first modify the following method in the `Publisher.java` file,

```
private void updatePubValue()
```

and add the following code, specifying the byte array value that the publisher wants to publish to the subscribers in the new event:

```
else if (this.attributes.equals("newAttribute"))
{
    //newAttribute supporting code here
}
```

Next modify the following method in the `EventHandler.java` file,

```
public void updateSubEvent(byte[] value)
```

and add the following code, specifying how the *Sub* event value is updated when a subscriber receives an update from the publisher:

```
else if (this.attributes.equals("newAttribute"))
{
    //newAttribute supporting code here
}
```

### 6.2.3 Routing Daemon

The *Routing Daemon* (defined in `rina.routing/RoutingDaemon.java`) is part of the transport process and responsible for the routing inside the VTN, and all routing-related components are included in the `rina.routing` package.

In the current implementation, we support both the link state routing protocol and the distance vector routing protocol. Also we support two kinds of link-cost policies: hop as link cost and jitter as link cost. Both routing protocols are implemented using the

Information API (Section 5.3.1.3), *i.e.*, the Pub/Sub system provided by the RIB daemon (Section 6.2.2).

For a link state routing protocol, once joining a VTN, the routing daemon of a transport process first creates two *Pub* events as follows.

```
SubscriptionEvent event = new SubscriptionEvent (EventType.PUB,
    checkNeighborPeriod , "checkNeighborAlive");
this.ribDaemon.createEvent(event);

SubscriptionEvent event1 = new SubscriptionEvent (EventType.PUB,
    routingEntrySubUpdatePeriod , "linkStateRoutingEntry ");
this.ribDaemon.createEvent(event1);
```

Then, for all its direct neighbors, the transport process creates two corresponding *Sub* events as follows.

```
SubscriptionEvent event = new SubscriptionEvent (EventType.SUB,
    checkNeighborPeriod , "checkNeighborAlive", neighbor);
this.ribDaemon.createEvent(event);

SubscriptionEvent event1 = new SubscriptionEvent (EventType.SUB,
    routingEntrySubUpdatePeriod , "linkStateRoutingEntry ", neighbor);
this.ribDaemon.createEvent(event1);
```

The “checkNeighborAlive” event helps the routing daemon check if a direct neighbor process is alive or not and also collect the link cost information about the link between them. The “linkStateRoutingEntry” event lets the routing daemon keep sending its adjacent links’ cost to all its neighbors, so that neighbors know its link cost information. Also when a transport process receives a link-state update message from a neighbor, it forwards it to all its direct neighbors (except the sender and originator), so the link-state message is propagated in the whole VTN. Through the two sets of events mentioned above, the routing daemon of each transport process collects the link cost information of the whole VTN, and builds its the forwarding table using the Dijkstra’s algorithm.

Similarly for the distance vector routing protocol, it uses a “distanceVector” event to send/receive distance vector information to/from its neighbors. Together with the “check-

NeighborAlive” event, the routing daemon can collect the path cost information of the whole VTN, and compute its forwarding table using the Split Horizon with Poison Reverse algorithm.

What’s more, we allow specifying different update frequencies for exchanging different routing messages inside the VTN, and this is done by setting different update frequencies for the *Pub/Sub* events.

#### 6.2.4 Shim Layer Process

As mentioned in Section 5.4.3, each node has a shim layer process which together forms the shim layer so we can deploy our management architecture on legacy networks. Each level-0 transport process has a *Wire Manager* (defined in `rina.irm.util/WireManager.java`) which is part of its VRM (IRM), and it is responsible for emulating physical connectivity for the level-0 VTNs.

In our current implementation, the physical connectivity is emulated using TCP connections, and each level-0 transport process listens to one well-known IP/port. The wire manager maintains all the TCP connections (emulated wires) between the level-0 transport processes, and it is initiated by the VRM (IRM) when a level-0 transport process is created. Note that in our current implementation, a level-0 VTN can have more than 2 transport processes, *i.e.*, it can span more than one emulated wire.

The (user-defined) name of each level-0 transport process is resolved to a pair of IP and port number, and the registration/resolution is done by a centralized shim layer directory service (defined in `rina.tcp.ds/DirectoryService.java`) which runs on a node in the network. When a level-0 transport process is created, it registers the mapping between its name and IP/port with the shim layer directory service, so that any level-0 transport process can be reached through a TCP connection.

Before a message is sent over a TCP connection, it is prepended with a variable-length integer (as `varint` defined by Google Protocol Buffers [58]) indicating the length of the message. The format of the variable-length integer is as follows, where the last byte is

required, and other bytes are optional.

[<one-bit value 1><7-bit unsigned integer>] <one-bit value 0><7-bit unsigned integer>

For each byte, the high-order bit indicates whether there are further bytes to come, and the low-order 7 bits are used to store the representation of the number in 7 bits.

## 6.3 Configurations Files

In order to use our implementation for network management, users need to specify the configurations for their network. Several configurations are necessary for users, including configurations for *node process*, *application process* and *transport process*. In this section, we explain how to set up these configuration files.

### 6.3.1 Node Process Configurations

As mentioned in Section 5.4, a node process is running on each physical node, and it is the container where application processes and transport processes reside. Users need to set up a configuration for each node in their network.

A node's configuration file includes information of the node itself and information of all processes (application processes and transport processes) residing on it. For a process, the information includes the process's naming information and location of its configuration file. When a node is initialized, transport processes and application processes on the node are bootstrapped based on their own configuration files.

The following is an example of a node's configuration file. In this example, the name of this node is *Node1*, and the configuration of its VAA (DAA) can be found in the configuration file *DAA.properties*. This node has two transport (IPC) processes (*BostonU1* and *BostonU2*) belonging to two VTNs (*VTN1* and *VTN2*), and one application process (*App1*). The location of configuration file for each process is also specified. Note that a process is identified by the concatenation of name and instance.

```
node.name = Node1
#VAA (DAA) information
```

```

DAA.configurationFile = DAA.properties
#Info of the first transport process (BostonU1)
IPC.1.DIF = VTN1
IPC.1.name = BostonU
IPC.1.instance = 1
IPC.1.level = 0
IPC.1.configurationFile = BostonU1.properties
#Info of the second transport process (BostonU2)
IPC.2.DIF = VTN2
IPC.2.name = BostonU
IPC.2.instance = 2
IPC.2.level = 1
IPC.2.configurationFile = BostonU2.properties
#Application Info
application.name = App
application.instance = 1
application.configurationFile = Appl.properties

```

The following is the configuration file of this node's VAA (DAA). The VA (DA) of the network runs on a machine with IP address *10.10.1.1* and listens to port number 7000. The VAA (DAA) of this node uses its local port number 7010 to communicate with VA (DA) over a TCP connection.

```

#Local TCP port
daa.port = 7010
#Centralized VA (DA) info
da.name = 10.10.1.1
da.port = 7000

```

To have more processes on a node, users only need to add configuration information for each process in the node's configuration file, and they will be started by the node process during its bootstrap.

### 6.3.2 Application Process Configurations

Each application process can have a configuration file (if needed) that includes all information to bootstrap it. This information includes application process's naming and service

information, as well as information about underlying transport processes that are used by this application process. Users can also include their own application-specific properties in the configuration file when writing their own applications.

The following is an example application process's configuration file. In this example, the name of this application is *App1*, and it is an RTP video proxy. It uses two underlying transport processes (*BostonU1* and *BostonU2*) on the same node to talk to other applications.

```

application.name = App
application.instance = 1
service.name = RTPVideoProxy
#Info of underlying transport processes
underlyingIPC.1 = BostonU1
underlyingIPC.2 = BostonU2

```

### 6.3.3 Transport (IPC) Process Configurations

Each transport (IPC) process has a configuration file, and it specifies the information needed to bootstrap this transport process, such as authentication policy, naming policy and routing policy. As mentioned earlier, in our implementation a transport process is denoted as *IPC* process, and VTN is denoted as *DIF*.

The following is an example transport process's configuration file. In this example, the transport process's name (concatenation of name and instance) is *BostonU1*, and it will operate inside *VTN3* which is a level-1 VTN. The enrollment of a new member into this VTN requires it to provide user and password information, and the enrollment procedure (details in Section 6.1.4) will be performed by an existing member *BostonU2* which will assign an internal address to it after the enrollment procedure. The transport process is instantiated to use a link-state routing protocol where link-state updates are sent to neighbor processes every 10 seconds, a neighbor is considered down if no probe message is received during the past 2 seconds, and the path cost is calculated using hop count. Also this transport process uses two underlying transport processes (*BostonU4* belonging to

*VTN4* and *BostonU5* belonging to *VTN5*) to talk to its peer processes inside *VTN3* after enrollment. Note that the authenticator's information can be specified in the configuration file, or it can be dynamically obtained by querying the directory service of VA (DA) with the VTN (DIF) name, and the directory service replies with the authenticator's information for the queried VTN (DIF).

```

rina.ipc.name = BostonU
rina.ipc.instance = 1
rina.ipc.level = 1
#Authentication information
rina.dif.name = VTN3
rina.ipc.enrolled = false
rina.enrollment.authenPolicy = AUTHPASSWD
rina.ipc.userName = BU
rina.ipc.passWord = BU
rina.authenticator.name = BostonU
rina.authenticator.instance = 2
#Routing information
rina.routing.protocol = linkState
rina.routingEntrySubUpdatePeriod = 10
rina.checkNeighborPeriod = 2
rina.linkCost.policy = hop
#Info of underlying transport processes
underlyingIPC.1 = BostonU4
rina.underlyingDIF.name.1 = VTN4
underlyingIPC.2 = BostonU5
rina.underlyingDIF.name.2 = VTN5

```

Note that if a transport process is instantiated as the first member of a VTN (DIF), then the authentication information part in the configuration file is different from above. The following shows an example where the transport process (*BostonU2*) is the first member of *VTN3*, and it has an internal address 1. This transport process is able to enroll new members into the VTN at the beginning.

```

rina.ipc.name = BostonU
rina.ipc.instance = 2
rina.ipc.level = 1
#Authentication information

```

```

rina.dif.name = VTN3
rina.ipc.enrolled = true
rina.address = 1
rina.enrollment.authenPolicy = AUTHLPASSWD
rina.ipc.userName = BU
rina.ipc.passWord = BU
#Routing information
rina.routing.protocol = linkState
rina.routingEntrySubUpdatePeriod = 10
rina.checkNeighborPeriod = 2
rina.linkCost.policy = hop
#Info of underlying transport processes
underlyingIPC.1 = BostonU7
rina.underlyingDIF.name.1 = VTN4
underlyingIPC.2 = BostonU8
rina.underlyingDIF.name.2 = VTN5

```

As mentioned earlier, each level-0 transport process has a wire manager which forms the shim layer of the network. The following is an example of a level-0 transport process's configuration file. The name of this transport process is *BostonU9*, and it is the first member of a level-0 VTN (*VTN0*). This level-0 transport process listens to local TCP port number 11120, the directory service of the shim layer is reachable at port number 11111 on a machine with IP *10.10.2.1*, and the directory service of VA (DA) is reachable at port number 8888 on a machine with IP *10.10.1.1*. This transport process has one emulated wire (*wire0*) to a neighbor transport process (BostonU10).

```

rina.ipc.name = BostonU
rina.ipc.instance = 9
rina.ipc.level = 0
#Authentication information
rina.dif.name = VTN0
rina.dif.enrolled = true
rina.address = 1
rina.enrollment.authenPolicy = AUTHLPASSWD
rina.ipc.userName = BU
rina.ipc.passWord = BU
#Shim layer info
TCPPort = 11120

```

```

#Shim layer neighbor (emulated wire) info
rina.underlyingDIF.name.1 = wire0
neighbour.1 = BostonU10
#Shim layer directory service info
rina.dns.name = 10.10.2.1
rina.dns.port = 11111
#VA (DA)'s directory service info
rina.idd.name = 10.10.1.1
rina.idd.port = 8888

```

In our current implementation, a level-0 transport process can have more than one emulated wire, and users only need to add the corresponding neighbor (wire) info to the `Shim layer neighbor (emulated wire) info` in its configuration file.

#### 6.3.4 Other Configurations

We also need to set up the configurations for the VA (DA), and the shim layer directory service. For the VA (DA), we need to configure two TCP port numbers: one for communication of the VA (DA) process with VAAs (DAAs) of the nodes in the network, and another one for the directory service process for the resolution of VTN (DIF) and application names. For the shim layer directory service process, we need to specify its TCP port number so that it can be reached for the resolution of user-defined level-0 transport process names.

## Chapter 7

# Evaluation of Architecture Implementation

In this chapter, we evaluate our implementation of the VTN-based management architecture. Our evaluations are performed on three different aspects: (1) routing performance, (2) unicast video application, and (3) multicast video application. Our implementation is tested both on our BU campus network and the GENI [25] testbed.

GENI (Global Environment for Network Innovations) [25] is a nationwide suite of infrastructure that supports large-scale experiments, and it enables research and education in networking and distributed systems. Through GENI, users can obtain computing resources (*e.g.*, virtual machines (VMs) and raw PCs) from different physical locations (GENI aggregates), and connect these computing resources with layer-2 (stitched VLAN) or layer-3 (GRE Tunnel) links. GENI allows users install customized software or even customized operating systems on these computing resources and also allows users control on how network switches handle traffic flows. GENI also enables experiments on SDN such as providing OVS [56] switches support and other OpenFlow support. What's more, GENI provides a variety of instrumentation and measurement tools (such as jFed, Jacks, Omni, GENI Desktop, LabWiki, Flack, *etc.*), to configure, run and measure user-specific experiments.

### 7.1 Routing Performance

In this section, we look at how our multi-layer approach can improve routing performance through experiments on a real network compared to the single-layer approach.

### 7.1.1 Experiment Design

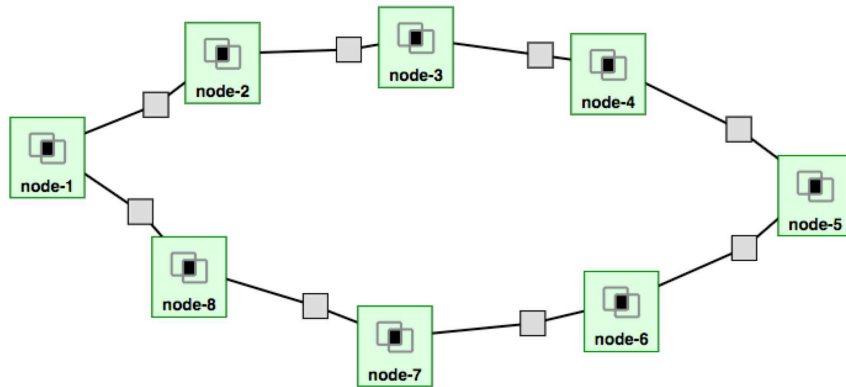


Figure 7.1: GENI resources with 8 nodes (VMs) shown in Jacks.

Figure 7.1 shows a network topology with 8 nodes reserved from the GPO InstaGENI aggregate of the GENI testbed. Assume we would like to provide communication service between applications on Node1 and Node5. Next we show two different designs which both can provide such communication service.

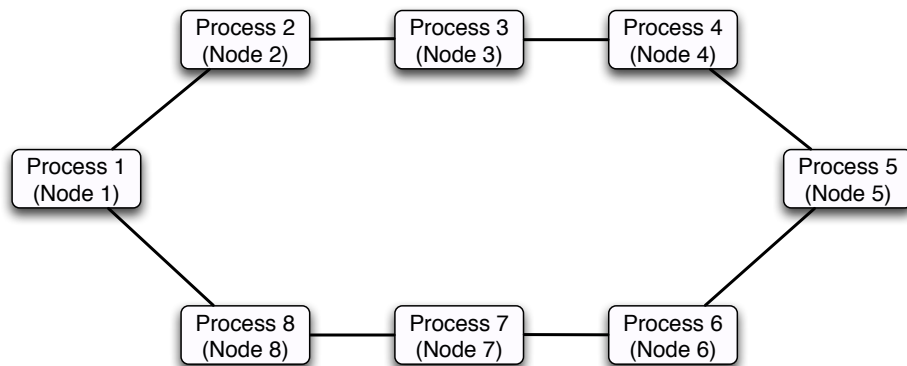


Figure 7.2: A single-layer design with one single VTN spanning all 8 nodes.

Figure 7.2 shows a single-layer design where we only have one VTN which spans all 8 nodes, and there is one transport process on each node.

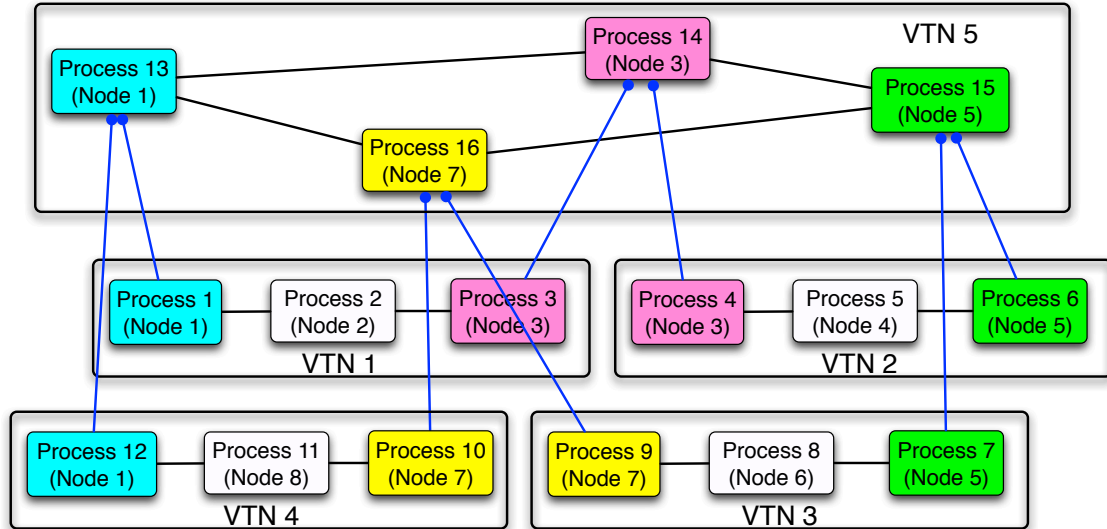


Figure 7.3: A multi-layer design with 5 VTNs of two levels.

On the other hand, Figure 7.3 shows a multi-layer design with 5 VTNs, and transport processes running on the same node are depicted with the same color.

In this multi-layer design, there are 4 level-1 VTNs (VTN1, VTN2, VTN3 and VTN4). Each of these level-1 VTNs has transport processes running on 3 nodes: VTN1 spans Node1, Node2 and Node3; VTN2 spans Node3, Node4 and Node5; VTN3 spans Node5, Node6 and Node7; and VTN4 spans Node1, Node8 and Node7. There is one level-2 VTN, *i.e.*, VTN5, which directly provides communication service for applications on Node1 and Node5. VTN5 has transport processes running on 4 nodes (Node1, Node3, Node5, and Node7) and each of the (virtual) links inside VTN5 is supported by one level-1 VTN.

### 7.1.2 Experiments over GENI

We try each of these two designs on the network reserved on GENI (shown in Figure 7.1). We use link-state routing with the same update frequency for all VTNs, and we run exper-

iments for each design using 4 different update frequencies for the link-state update (LSU) messages, where LSU messages are set to be exchanged every 5, 10, 15, and 20 seconds.

For each update frequency, we measure the total number of LSU messages and their total size processed by all 8 nodes during steady state.

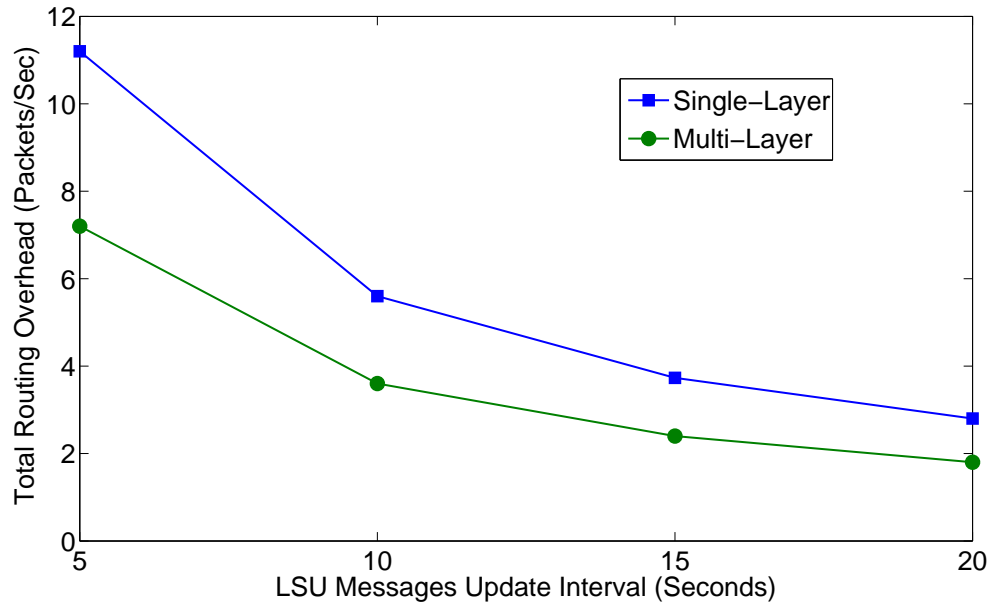


Figure 7.4: Total number of LSU messages processed per second by all 8 nodes during steady state for 4 different update frequencies.

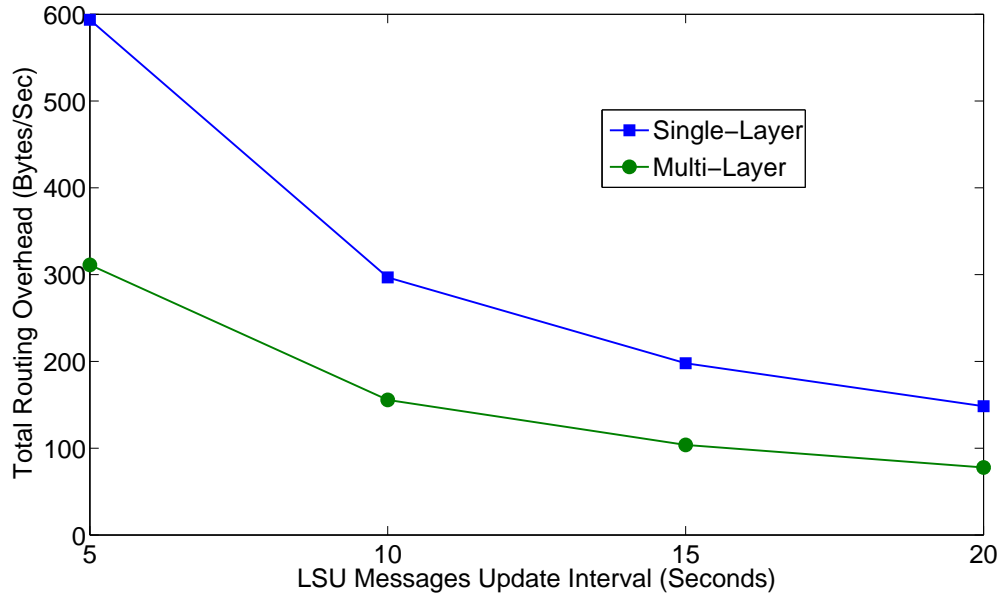


Figure 7.5: Total size of LSU messages processed per second by all 8 nodes during steady state for 4 different update frequencies.

Figure 7.4 shows the total number of LSU messages processed per second by all 8 nodes during steady state for 4 different update frequencies, and Figure 7.5 shows the total size (in bytes) of LSU messages processed per second by all 8 nodes during steady state for 4 different update frequencies. We can see that, expected as our discussion in Section 3.2.3, our multi-layer design can yield less routing overhead (both in the number and size of LSU messages) compared to the single-layer design by limiting the scope in which LSU messages are propagated and avoiding unnecessary communications.

## 7.2 Video Unicast Performance

In this section, we show how our VTN-based management architecture can improve application performance through application-specific policies by using video unicast streaming as an example.

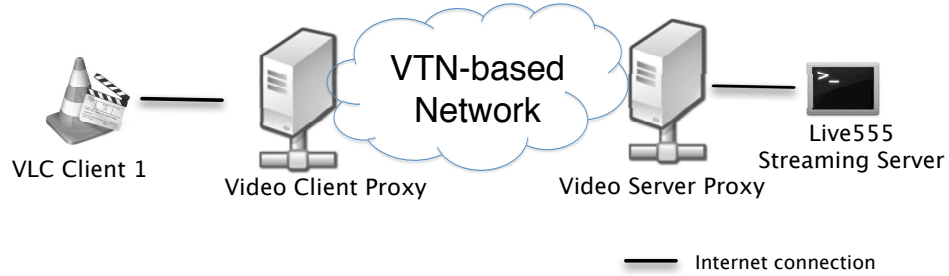


Figure 7.6: Video clients (VLC players) are connected to the video streaming server (Live555 streaming server) through RTSP proxies over a VTN-based network.

In order to support Real-Time Streaming Protocol (RTSP) video streaming application running on legacy nodes, two video proxies are used as shown in Figure 7.6. The video client proxy is connected to the video client over the Internet, and the video server proxy is connected to the video streaming server also over the Internet. These two proxies are connected over a VTN-based network that is composed of VTNs, where routing policies (and other policies) can be easily configured for each VTN. Video proxies support the Real-Time Streaming Protocol (RTSP) and redirect all traffic between the video client and the video streaming server to the communication service provided by the VTN-based network. The details of these two proxies can be found in [85]. In this experiment, we use the VLC player (version 2.0.4) [71] as the video client, and the Live555 server (version 0.78) [44] as the video streaming server. The video file used in our experiment is encoded in the H.264/MPEG-4 AVC (Part 10) format, and can be found at [1].

### 7.2.1 VTN Policies for Video Streaming

As mentioned earlier, our VTN-based management approach separates mechanisms and policies, where different VTNs use the same mechanism but can be instantiated with different policies. For the VTN that directly provides communication service between the two video proxies, we use the link-state routing protocol, and we test two link-cost policies (hop and jitter), and then observe how they affect the performance of the unicast video

application.

## 7.2.2 Experiments over GENI

### 7.2.2.1 Experiment Design

As shown in Figure 7.7, we reserve GENI resources (VMs, VLANs, and GRE tunnels) from four GENI aggregates (Georgia Tech, UIUC, NYU, and Cornell University). VMs in different aggregates are connected using GRE tunnels, and VMs in the same aggregate are connected using VLANs. Each node process (Section 5.4) is running on a GENI VM, and we use 13 nodes (Node 1 to Node 13). Node 1, Node 2, and Node 3 are running on VMs from the NYU aggregate. Node 4, Node 5, Node 6, and Node 7 are running on VMs from the Georgia Tech aggregate. Node 8, Node 9, and Node 10 are running on VMs from the UIUC aggregate. Node 11, Node 12, and Node 13 are running on VMs from the Cornell aggregate.

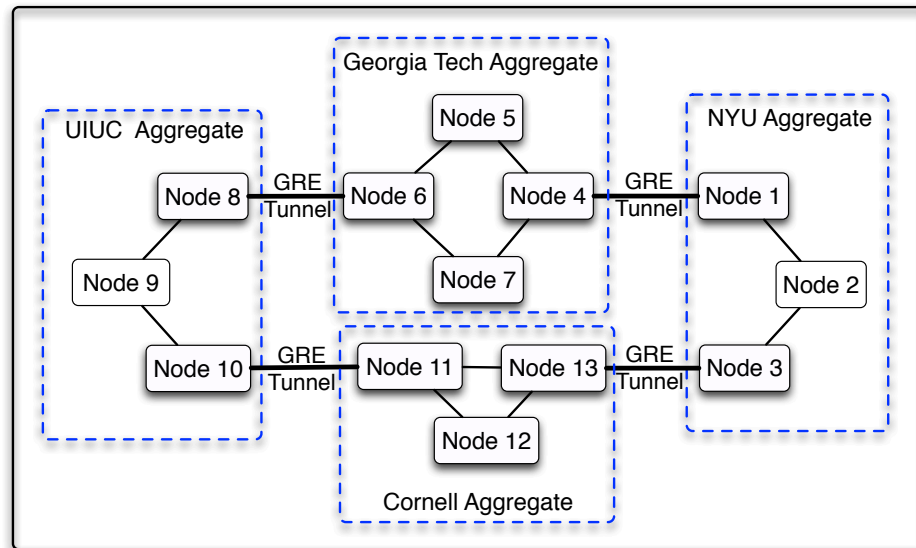


Figure 7.7: Each node process is running on a GENI VM. Nodes in different aggregates are connected via GRE tunnels, and nodes in the same aggregate are connected via VLANs.

To provide communication service for the video client proxy located at Node 9 and the

video server proxy located at Node 2, in our experiment, we use a VTN design shown in Figure 7.8, and focus on different link-cost policies.

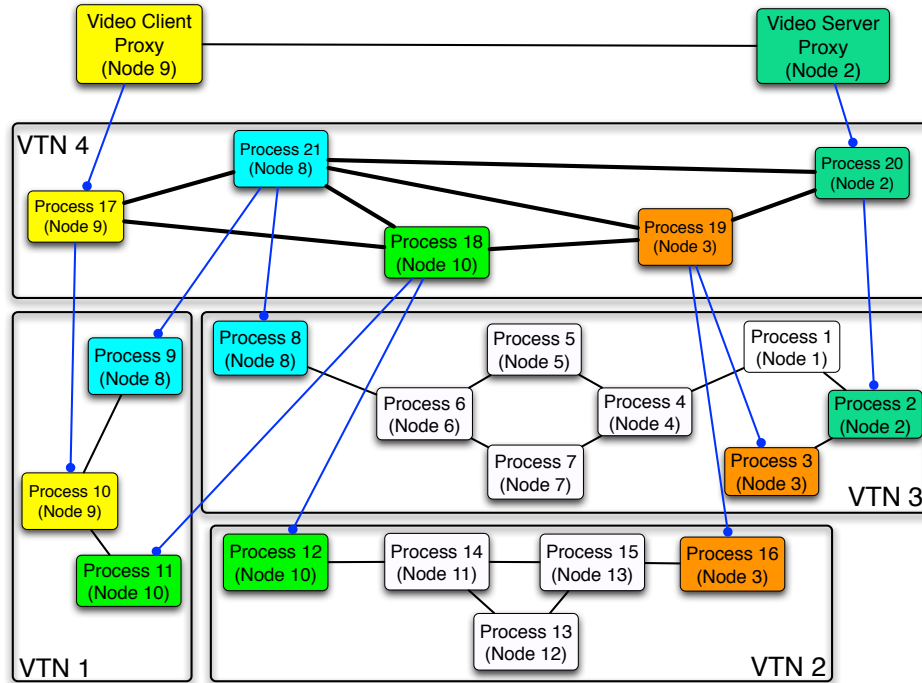


Figure 7.8: Video client proxy (on Node 9) and video server proxy (on Node 2) communicate through a level-2 VTN (VTN 4), which is built on top of three level-1 VTNs (VTN 1, VTN 2, and VTN 3).

As shown in Figure 7.8, there are three level-1 VTNs (VTN 1, VTN 2, and VTN 3), and one level-2 VTN (VTN 4). VTN 1 has three members: Process 9 (on Node 8), Process 10 (on Node 9), and Process 11 (on Node 10). VTN 2 has five members: Process 12 (on Node 10), Process 13 (on Node 12), Process 14 (on Node 11), Process 15 (on Node 13), and Process 16 (on Node 3). VTN 3 has eight members: Process 1 (on Node 1), Process 2 (on Node 2), Process 3 (on Node 3), Process 4 (on Node 4), Process 5 (on Node 5), Process 6 (on Node 6), Process 7 (on Node 7), and Process 8 (on Node 8). VTN 4 has five members: Process 17 (on Node 9), Process 18 (on Node 10), Process 19 (on Node 3), Process 20 (on Node 2), and Process 21 (on Node 8). Processes hosted

on the same node are depicted in the same color in Figure 7.8.

In Figure 7.8, the video client proxy on Node 9 uses Process 17, which recursively uses Process 10. The video server proxy on Node 2 uses Process 20, which recursively uses Process 2. Process 21 on Node 8 uses Process 8 and Process 9, Process 18 on Node 10 uses Process 11 and Process 12, and Process 19 on Node 3 uses Process 3 and Process 16.

So the video client proxy and video server proxy communicate through a connection supported by the underlying VTN 4, which recursively uses the communication services provided by three level-1 VTNs. The connection between the video server proxy (using Process 20) and the video client proxy (using Process 17) is mapped to a path inside VTN 4, and each link in VTN 4 is supported by a level-1 VTN.

We use the network emulation tool, *NetEm* [50], to emulate link delay and jitter. Delay (300ms) with variation ( $\pm 200$ ms) is emulated on two physical links between Node 1 and Node 2, and between Node 8 and Node 9. This emulation leads to jitter on link Process 9-Process 10 in VTN 1, and on link Process 1-Process 2 in VTN 3. This in turn is reflected as link jitter in the higher-level VTN 4, where four links (Process 17-Process 21, Process 18-Process 21, Process 20-Process 21, and Process 19-Process 21) exhibit jitter that they inherit from underlying paths.

### 7.2.2.2 Experimental Results

We run our experiments on a network reserved on GENI as shown in Figure 7.9, which corresponds to Figure 7.7.

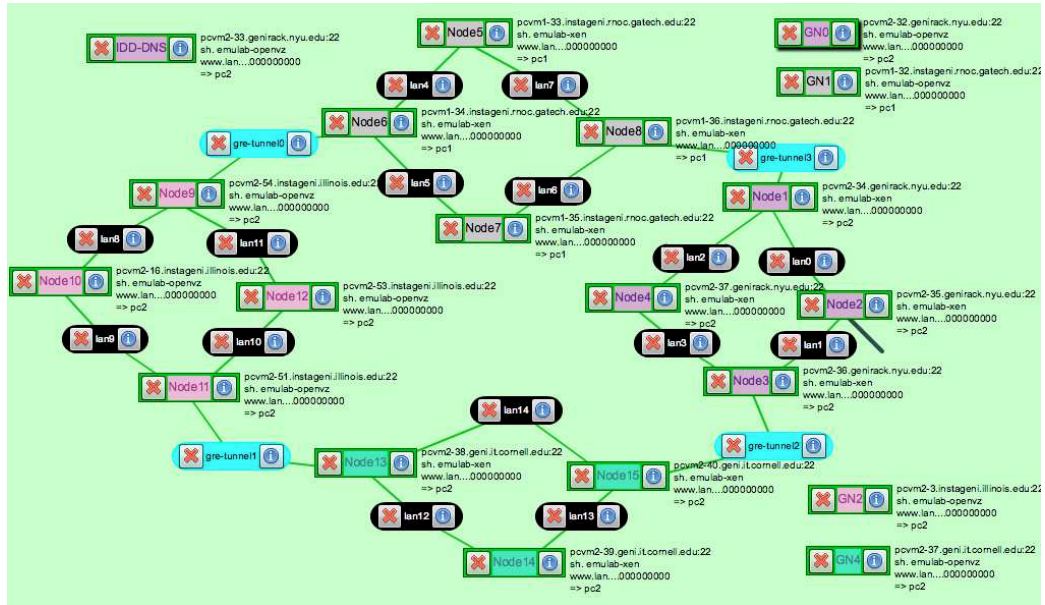


Figure 7.9: GENI resources from four aggregates shown in Flack.

As we can see from Figure 7.8, the connection between the video server proxy and video client proxy can be routed via one of the seven possible loop-free paths inside VTN 4 between Process 20 and Process 17. These paths are: *path 1* (Process 20 - Process 21 - Process 17), *path 2* (Process 20 - Process 19 - Process 18 - Process 17), *path 3* (Process 20 - Process 21 - Process 19 - Process 18 - Process 17), *path 4* (Process 20 - Process 21 - Process 18 - Process 17), *path 5* (Process 20 - Process 19 - Process 21 - Process 17), *path 6* (Process 20 - Process 19 - Process 21 - Process 18 - Process 17), and *path 7* (Process 20 - Process 19 - Process 18 - Process 21 - Process 17).

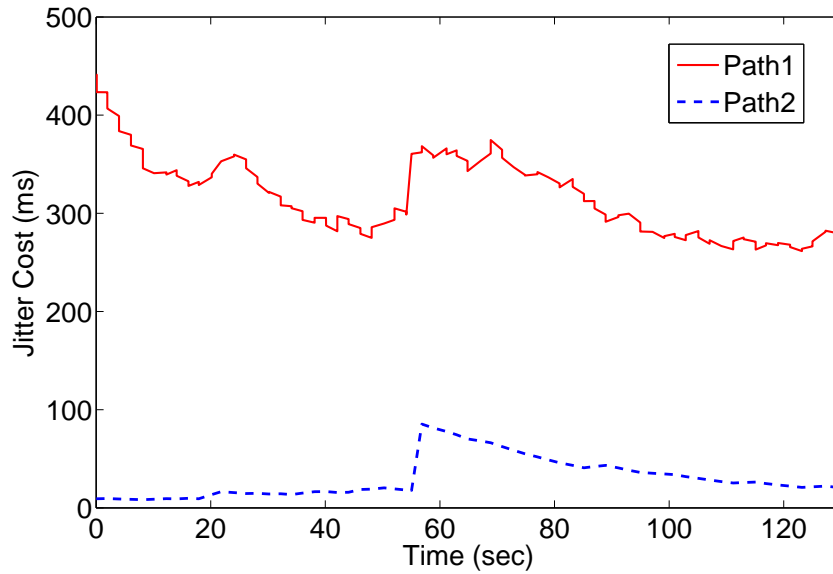


Figure 7.10: Path jitter of *path 1* (the least-hop path) is larger than *path 2* (the least-jitter path), where path jitter is calculated as the sum of jitter on links along the path (collected by the routing task of `Process 20`).

Figure 7.10 shows the path jitter of *path 1* (least-hop path) and *path 2* (least-jitter path), where the jitter of a path is calculated as the sum of jitter on all links along that path (collected by the routing task of `Process 20`). If jitter is used as link cost, the connection between `Process 20` and `Process 17` is routed on *path 2*. On the other hand, if hop is used as link cost, the connection is routed on *path 1*.

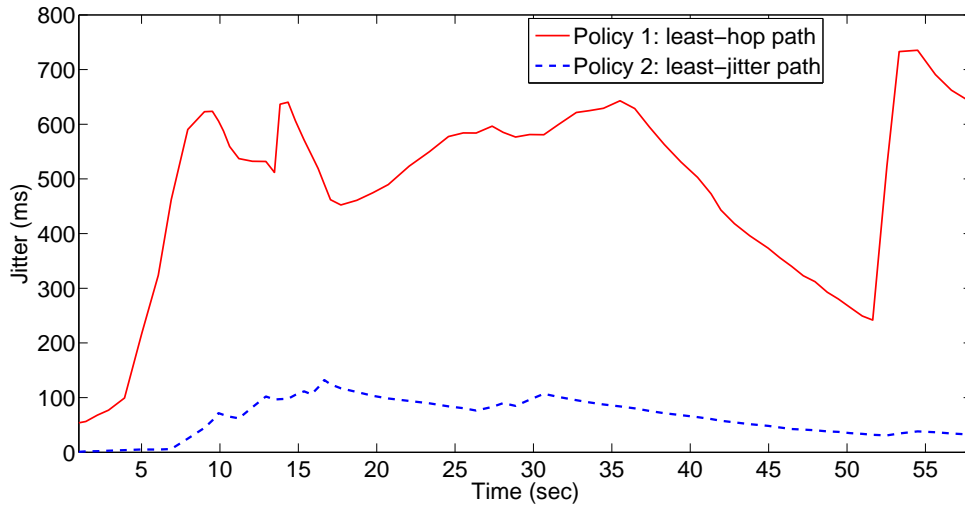


Figure 7.11: Measured instantaneous jitter for video packets from the video server proxy to the client proxy when VTN 4 uses hop or jitter as link cost.

For video applications, it is important to choose a path with least jitter, otherwise video quality degradation is observed. Figure 7.11 shows the measured instantaneous jitter that is experienced by video packets from the video server proxy to the client proxy, as the video server streams the same video to a player client. To calculate the jitter, we sample video packets received by the client proxy at the rate of one every 60 packets. We can see that under least-jitter routing, the video packets experience much less jitter compared to least-hop routing.

As a consequence of this increased jitter, we indeed observe that the video freezes more often when using hop rather than jitter as the link-cost policy. Measuring video quality at the client side (player) using metrics such as signal-to-noise ratio (SNR) and mean opinion score (MOS), is left for future work.

### 7.3 Video Multicast Performance

In this section, we explain how video can be efficiently multicast to different clients on demand as an example of application-driven network management.

In order to support RTP (Real-time Transport Protocol) video streaming over the

VTN-based network, RTP proxies (server proxy and client proxy) are used as shown in Figure 7.12. The RTP server proxy is connected to the video server over the Internet, and each RTP client proxy is connected to a video client also over the Internet. The RTP server proxy and RTP client proxies are connected over the VTN-based network which consists of VTNs. Namely, the RTP server proxy redirects all RTP traffic between the RTP server and RTP client to the communication channel provided by the VTN-based network. In our experiments, we use the VLC player [71] as the video client, and the Live555 server [44] as the RTP video server. The video file used in the experiments is an MPEG Transport Stream file, which can be found at [2].

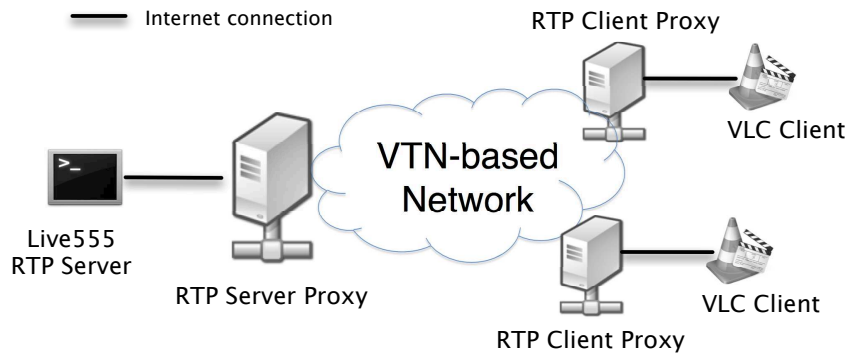


Figure 7.12: Video clients (VLC players) are connected to the RTP video server through RTP proxies over a VTN-based network.

Figure 7.13 shows a scenario, where the enterprise network is made up of four subnetworks. The RTP server and RTP server proxy are running in **Network A**, and they provide a live video streaming service. There are two video clients along with RTP client proxies (one in **Network C** and the other one in **Network D**) that would like to receive video provided by the RTP video server. **Network A** and **Network B** are connected through VTN 1, **Network B** and **Network C** are connected through VTN 2, and **Network B** and **Network D** are connected through VTN 3. VTN 1, VTN 2 and VTN 3 are three level-zero VTNs that can provide communication services for two connected networks. For simplicity, the Live555 RTP server and VLC clients are not shown in the following figures.

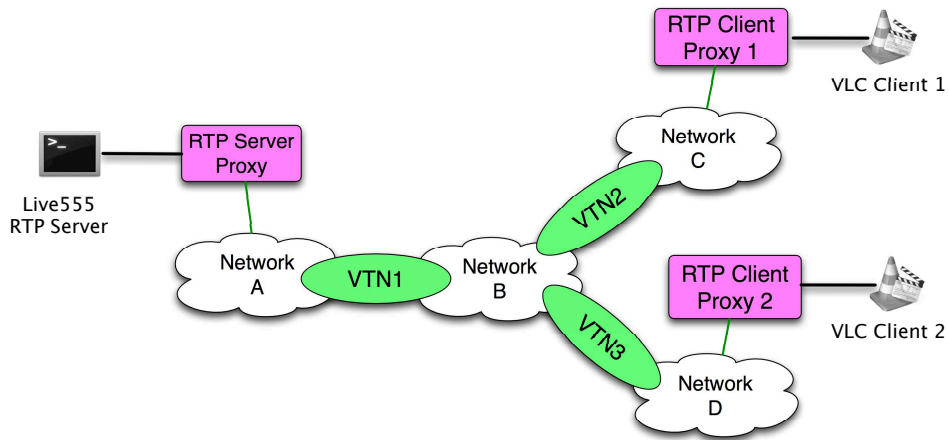


Figure 7.13: Video server providing a live video streaming service is running in Network A. One client is in Network C, and one is in Network D.

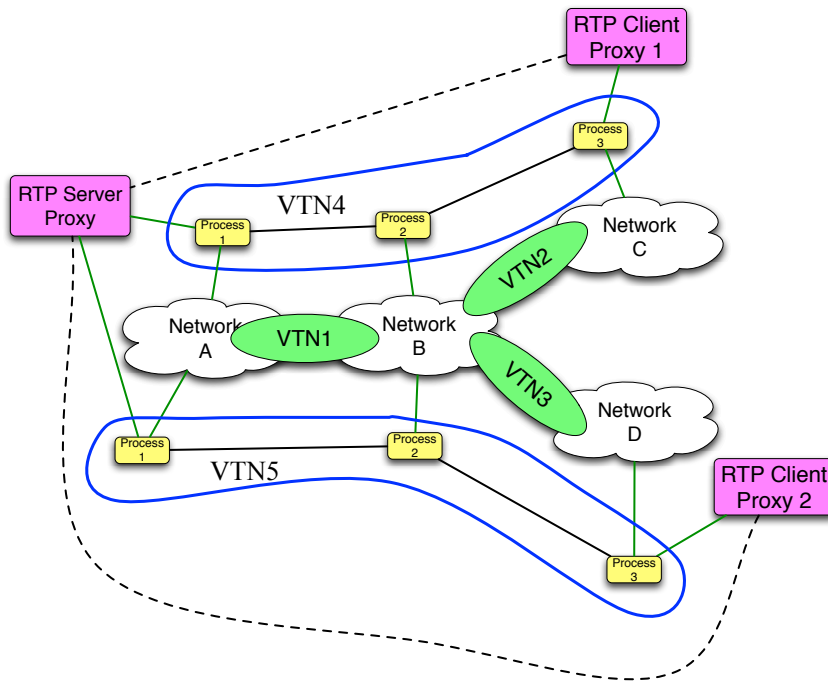


Figure 7.14: Video streaming through unicast connections, where same video traffic is delivered twice over VTN 1 consuming unnecessary network bandwidth.

A very simple way to meet clients' requirements is as follows. Two video clients can receive live streaming service from the video server through two unicast connections sup-

ported by two separate VTNs as shown in Figure 7.14. The unicast connection between RTP Client Proxy 1 and the video server proxy is supported by VTN 4, which is a level-one VTN formed based on VTN 1 and VTN 2. The unicast connection between RTP Client Proxy 2 and the video server proxy is supported by VTN 5, which is a level-one VTN formed based on VTN 1 and VTN 3. However, it is easy to see that the same video traffic is delivered twice over VTN 1, which consumes unnecessary network bandwidth. In order to make better use of network resources, it is necessary to use multicast to stream the live video traffic. Next we show two different solutions of managing the existing VTNs to support multicast, *i.e.*, two ways of application-driven network management.

### 7.3.1 VTN-based Multicast Solutions

#### 7.3.1.1 Solution One: Application-level Multicast

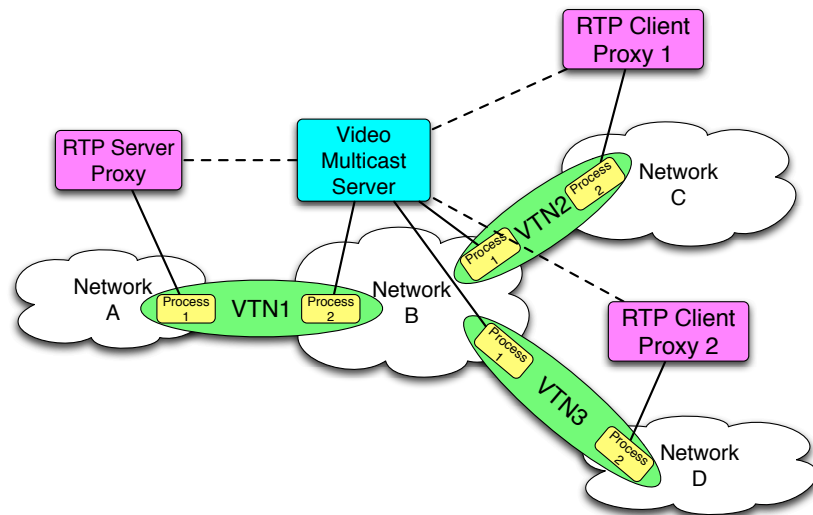


Figure 7.15: Video multicast through an RTP multicast video server.

The first solution is enabled through a video multicast server as shown in Figure 7.15. The connection between the video server proxy and the video multicast server is supported by VTN 1. The connection between the video multicast server and RTP Client Proxy 1

is supported by VTN 2, and the connection between the video multicast server and RTP Client Proxy 2 is supported by VTN 3. The video server proxy streams video traffic to the video multicast server, which multicasts video traffic to each client through two unicast connections supported by VTN 2 and VTN 3, respectively. We can see that the video traffic is delivered only once over VTN 1 compared to Figure 7.14. In this case, we only rely on existing level-zero VTNs, and no new higher-level VTN is created.

Actually the video multicast server provides a VNF (Virtual Network Function [20]) as in NFV (Network Function Virtualization), *i.e.*, our VTN-based management approach can implicitly support NFV. In a complicated network topology with more local networks, if there are more clients from different local networks needing the live streaming service, we can instantiate more video multicast servers, and place them at locations that are close to the clients, thus provide better video quality and network performance (such as less jitter and bandwidth consumption).

**7.3.1.2 Solution Two: VTN-level Multicast**

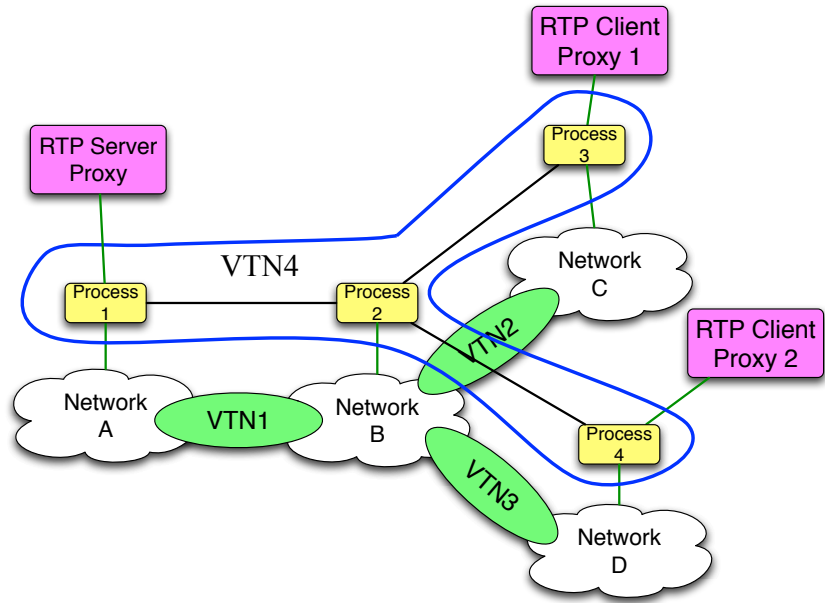


Figure 7.16: Video multicast through multicast service provided by the VTN.

The second solution is supported using the multicast service provided by the VTN mechanism. As shown in Figure 7.16, we form a level-one VTN (VTN 4) on top of existing level-zero VTNs. The video server proxy creates a multicast channel through VTN 4, and streams live video traffic over this multicast channel. Each client joins the multicast channel to receive the live video traffic. Note that the allocation of a multicast connection is the same as the allocation of a unicast connection.

Here we can see that our VTN-based management approach implicitly supports SDN [54] by allowing the dynamic formation of new VTNs, what's more, it allows instantiating different policies for different VTNs. In a complicated network topology with more local networks, if there are more clients from different local networks accessing the live streaming service, we can either dynamically form new higher-level VTNs or expand the existing VTNs providing the multicast service.

### 7.3.2 Experiments over GENI

In this section, we show the experimental results of our VTN-based multicast solutions over the GENI [25] testbed.

#### 7.3.2.1 Bandwidth Usage

As shown in Figure 7.17, we reserve four VMs from four InstaGENI aggregates (Rutgers, Wisconsin, Chicago and NYSERNet), and we connect the VMs using stitched VLANs. Each aggregate corresponds to one network in Figure 7.13, where the RTP server and RTP server proxy are running on VM N1 in the Rutgers aggregate, the RTP Client Proxy 1 is running on VM N4 in the Chicago aggregate, and the RTP Client Proxy 2 is running on VM N3 in the NYSERNet aggregate.

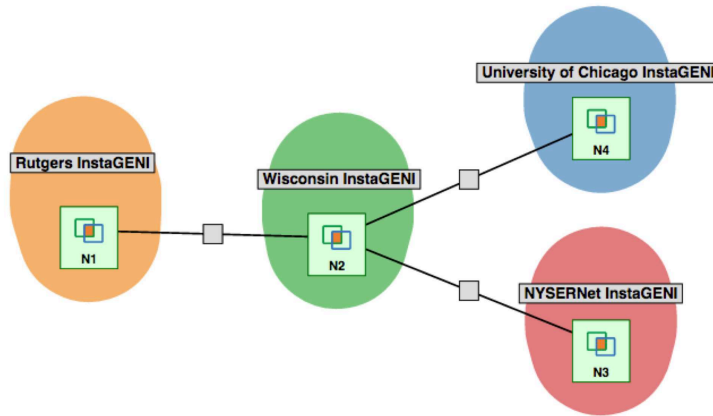


Figure 7.17: GENI resources from four InstaGENI aggregates shown in Jacks.

Figure 7.18 shows the bandwidth usage for the unicast solution and the two multicast solutions over VTN 1 (cf. Figure 7.13), *i.e.*, the link between VM N1 in the Rutgers aggregate and VM N2 in the Wisconsin aggregate in Figure 7.17. We can see that, as expected, the bandwidth usage for the two multicast solutions are close to half of that of the unicast solution.

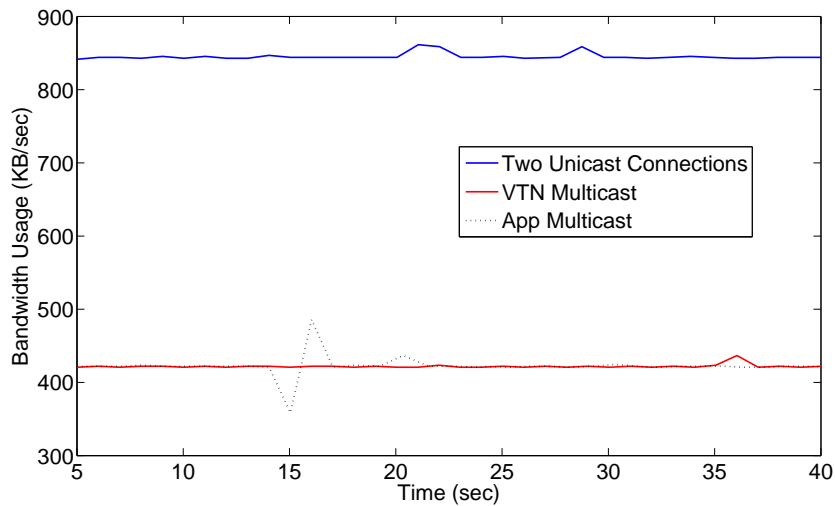


Figure 7.18: Comparison of bandwidth usage over VTN1: unicast vs. multicast.

### 7.3.2.2 Video Quality

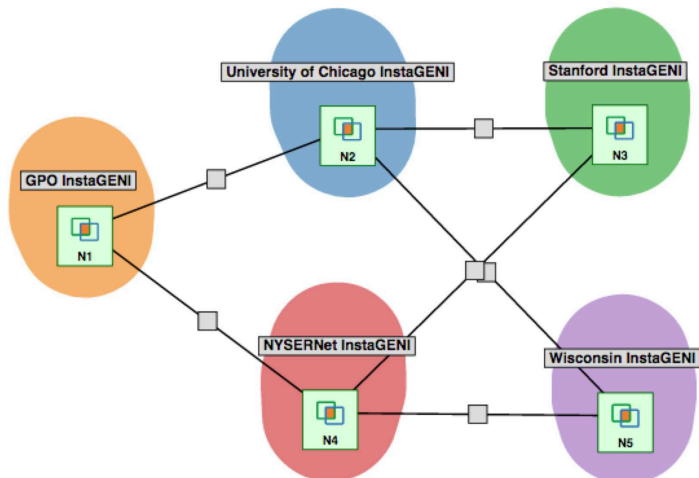


Figure 7.19: GENI resources from five InstaGENI aggregates shown in Jacks.

As shown in Figure 7.19, we reserve five VMs from five InstaGENI aggregates (GPO, Chicago, NYSErNet, Stanford, and Wisconsin), and we connect the VMs using stitched VLANs. The RTP server and RTP server proxy are running on VM N1 in the GPO aggregate, the RTP Client Proxy 1 is running on VM N3 in the Stanford aggregate, and the RTP Client Proxy 2 is running on VM N5 in the Wisconsin aggregate. The goal is to observe the effect on the video quality at the video client side when placing the video multicast server (cf. Section 7.3.1.1) in different locations, *i.e.*, placing the video multicast server either on VM N2 in the Chicago aggregate or VM N4 in the NYSErNet aggregate.

Since GENI does not yet allow specifying parameters when reserving stitched VLANs, such as capacity, packet loss and latency, we use a network emulation tool, NetEm [50] to add delay ( $1000\text{ms} \pm 500\text{ms}$ ) on the link between VM N1 in the GPO aggregate and VM N2 in the Chicago aggregate. In order to observe video quality, we have VLC players running locally on our BU campus network and connect them to the RTP client proxies running on GENI aggregates (*i.e.*, VM N3 and N5) via Internet connections. Note that the jitter on the Internet connections is negligible, and the jitter in our experiments is mainly from

jitter emulated on GENI links.



Figure 7.20: Video observed when the video multicast server is placed on VM N4 in the NYSERNet aggregate resulting in a path with less jitter.

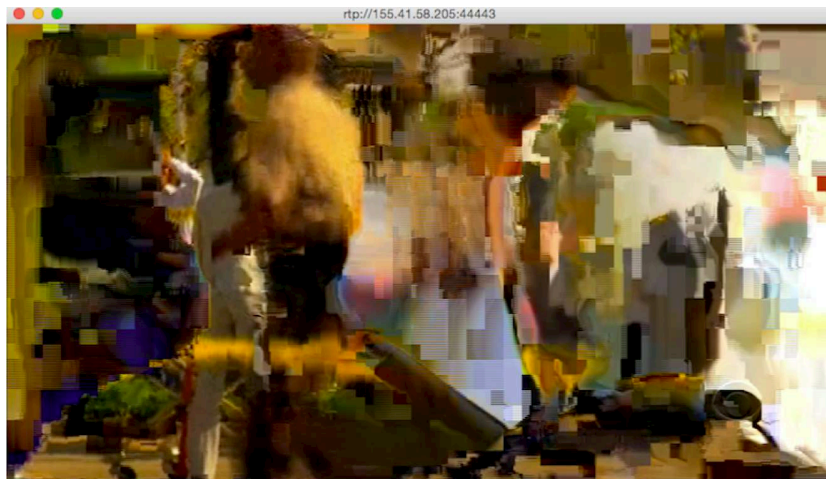


Figure 7.21: Video observed when the video multicast server is placed on VM N2 in the Chicago aggregate resulting in a path with more jitter.

We run a VLC player locally and connect it with the RTP Client Proxy 1 running on VM N3 in the Stanford aggregate. Figure 7.21 shows the video observed when placing the multicast server on VM N2 in the Chicago aggregate. Figure 7.20 shows the video observed when placing the multicast server on VM N4 in the NYSERNet aggregate. We can see that

by placing the video multicast server at a location experiencing less jitter we can achieve better video quality.

## Chapter 8

# Conclusions and Future Work

### 8.1 Conclusions

Software Defined Networking (SDN) is an efficient way to make the network programmable and reduce management complexity, however it is plagued with limitations inherited from the legacy Internet (TCP/IP) architecture. On the other hand, service overlay networks and virtual networks are widely used to overcome deficiencies of the Internet. However, most overlay/virtual networks are single-layered and lack dynamic scope management. Furthermore, how to solve the joint problem of designing and mapping the overlay/virtual network requests for better application and network performance remains an understudied area.

In response to limitations of current SDN management solutions and of the traditional single-layer overlay/virtual network design, in this thesis we propose a recursive approach to enterprise network management, where network management is done through managing various Virtual Transport Networks (VTNs) over different scopes (*i.e.*, regions of operation). Also we propose a framework for multi-layer VTN design to satisfy different application and network requirements, which allows for achieving different goals by setting different constraints and objectives for optimization problems. What's more, we present the design, implementation and evaluation of our VTN-based management architecture, which enables the VTN-based network management on real networks. Our simulation and experimental results demonstrate the flexibility of our VTN-based management approach and its performance advantages.

## 8.2 Future Work

We believe the idea of scoped and multi-layered management proposed in thesis will shape the future of computer networking management, and we will continue our research on demonstrating its benefits and advantages.

**Online VTN Design:** it is important to investigate the online VTN design problem for serving new flow requests, where new flow requests arrive when there is already some existing VTN structure. It might be more efficient to extend an existing VTN (by adding transport processes on different nodes) to serve a new flow request compared to creating a new VTN. Furthermore, how to reorganize/reconfigure the existing VTN structure, when the physical conditions of the network and the demand of existing flows change, is an important aspect. Also it would be interesting to explore the multi-layer VTN design problem from an algorithmic perspective and further improve our heuristic.

**Multi-domain Network Management:** managing a large network, which consists of multiple subnetworks belonging to different administrative domains, is a more complex task compared to the management of an enterprise network. Our VTN-based architecture can also be applied to multi-domain network management, where we can easily aggregate and break management scopes. Recursively, the VTN Allocator of each domain can be considered as an (enterprise-level) VTN Allocator Agent to the multi-domain (global) VTN Allocator. So by recursing the same VA-VAA structure, our management layer can create new VTNs spanning nodes belonging to multiple administrative domains to support larger-scope transport flows.

**QoS Support:** one important future work is to keep adding new components (such as a complete error and flow control component) to our implementation and enable QoS support for more metrics (such as throughput and delay). Also it would be interesting to investigate the algorithms and mechanisms needed for QoS support to enable better resource allocation and utilization on real networks.

**Security:** the security of the management architecture is a key factor in the operations of an enterprise network, and it would be an interesting topic to investigate the possible attacks on and risks of our VTN-based management architecture and further develop corresponding defensive solutions.

**Adoption/Deployment:** strengthening the flexibility of the shim layer in our implementation is essential to overlay our architecture over more different legacy networks such as Ethernet, OpenFlow, *etc.* Besides, it would be very interesting to investigate the possibility of developing dedicated devices for deploying our VTN-based management architecture.

**Formal Verification of Protocols and Interfaces:** one important direction is to use formal methods to verify the correctness of our protocols and guarantee the proper usage of our API (both admin-level and user-level), which contribute to the reliability and robustness of the management architecture.

**Application Use Cases:** our VTN-based approach can enable better application performance, and another important future work is to research how to design and apply application-specific VTN policies to improve performance for more real applications.

## Appendix A

### Proof of Equation (3.8)

$$\text{Equation (3.8): } E_{tcp} = \frac{\left(\frac{1}{1-P}\right)^H - 1}{P}$$

**Proof:** Equation (3.8) can be derived using an absorbing Markov chain. As shown in Figure A.1, each circle denotes a possible state of the current packet, where  $S_0$  is the initial state where a packet is to be sent by the sender, and  $S_H$  is the absorbing (final) state when the packet is received by the receiver. For any intermediate state, it has a probability of  $1 - P$  of transitioning to the following state if the packet does not get lost; if the packet gets lost (with probability  $P$ ), it goes back to the initial state ( $S_0$ ). This absorbing Markov chain has  $H$  transient states ( $S_0$  to  $S_{H-1}$ ), and 1 absorbing state ( $S_H$ ), so the expected number of total transmissions for all hosts along the path to successfully deliver one packet is the same as the expected number of steps from the initial state  $S_0$  to the absorbing state  $S_H$ .

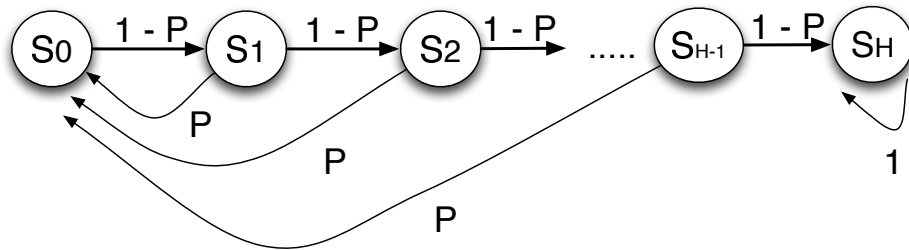


Figure A.1: An absorbing Markov chain for delivering one packet over a TCP connection of  $H$  hops, where each circle denotes a possible state. Assume loss rate on each link is  $P$ .

Generally, for an absorbing Markov chain with transition matrix  $P$ , assume it has  $t$

transient states and  $r$  absorbing state, then

$$P = \begin{bmatrix} Q & R \\ 0 & I_r \end{bmatrix}$$

where  $Q$  is a  $t$ -by- $t$  matrix and  $I$  is the  $r$ -by- $r$  identity matrix. The fundamental matrix of an absorbing Markov chain is  $N = (I - Q)^{-1}$ , and the expected number of steps from the initial state to the absorbing states is  $t = Nc$ , where  $c$  is a column vector all of whose entries are 1 [27].

For the absorbing Markov chain in Figure A.1,  $t = H$  and  $r = 1$ , so its transition matrix is

$$P[(H+1) \times (H+1)] = \begin{bmatrix} P & 1-P & 0 & 0 & \dots & 0 & 0 \\ P & 0 & 1-P & 0 & \dots & 0 & 0 \\ P & 0 & 0 & 1-P & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ P & 0 & 0 & 0 & \dots & 0 & 1-P \\ 0 & 0 & 0 & 0 & \dots & 0 & 1 \end{bmatrix}$$

where

$$Q[H \times H] = \begin{bmatrix} P & 1-P & 0 & 0 & \dots & 0 \\ P & 0 & 1-P & 0 & \dots & 0 \\ P & 0 & 0 & 1-P & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ P & 0 & 0 & 0 & \dots & 1-P \\ P & 0 & 0 & 0 & \dots & 0 \end{bmatrix}$$

Then its fundamental matrix is

$$N = (I - Q)^{-1} = \begin{bmatrix} 1 - P & P - 1 & 0 & 0 & \dots & 0 \\ -P & 1 & P - 1 & 0 & \dots & 0 \\ -P & 0 & 1 & P - 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ -P & 0 & 0 & 0 & \dots & P - 1 \\ -P & 0 & 0 & 0 & \dots & 1 \end{bmatrix}^{-1}$$

$$= \begin{bmatrix} (1 - P)^{-H} & (1 - P)^{1-H} & \dots & (1 - P)^{-2} & (1 - P)^{-1} \\ \vdots & \vdots & \ddots & \vdots & \vdots \end{bmatrix}$$

So the expected number of steps from  $S_0$  to  $S_H$  is

$$t = Nc = (1 - P)^{-1} + (1 - P)^{-2} + \dots + (1 - P)^{-H} = \frac{\left(\frac{1}{1-P}\right)^H - 1}{P}$$

Then we prove that the expected number of total transmissions for all hosts along the path to successfully deliver one packet is  $E_{tcp} = \frac{\left(\frac{1}{1-P}\right)^H - 1}{P}$ .

## Bibliography

- [1] <http://csr.bu.edu/rina/videoSample/>.
- [2] <http://csr.bu.edu/rina/RTPVideoSample/>.
- [3] Amazon Web Service. <https://aws.amazon.com/>.
- [4] Microsoft Azure. <https://azure.microsoft.com/>.
- [5] D. Adami, C. Callegari, S. Giordano, G. Nencioni, and M. Pagano. Design and Performance Evaluation of Service Overlay Networks Topologies. In *Performance Evaluation of Computer Telecommunication Systems, 2009. SPECTS 2009. International Symposium on*, volume 41, pages 296–303, July 2009.
- [6] D. Andersen, H. Balakrishnan, F. Kaashoek, and R. Morris. Resilient Overlay Networks. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, pages 131–145, New York, NY, USA, 2001. ACM.
- [7] T. Anderson, L. Peterson, S. Shenker, and J. Turner. Overcoming the Internet Impasse through Virtualization. *Computer*, (4):34–41, 2005.
- [8] G. Appenzeller, I. Keslassy, and N. McKeown. Sizing Router Buffers. In *SIGCOMM 2004*, New York, NY, USA. ACM.
- [9] H. Balakrishnan, V. N. Padmanabhan, S. Seshan, and R. H. Katz. A Comparison of Mechanisms for Improving TCP Performance over Wireless Links. In *IEEE/ACM Transactions on Networking*, volume 5, pages 756–769, Dec 1997.
- [10] I. B. Barla, D. Schupke, M. Hoffmann, G. Carle, et al. Optimal Design of Virtual Networks for Resilient Cloud Services. In *Design of Reliable Communication Networks (DRCN), 2013 9th International Conference on the*, pages 218–225. IEEE, 2013.
- [11] J. Barron, M. Crotty, E. Elahi, R. Riggio, D. R. Lopez, and M. P. de Leon. Towards Self-adaptive Network Management for a Recursive Network Architecture. In *NOMS 2016 - 2016 IEEE/IFIP Network Operations and Management Symposium*, pages 1143–1148, April 2016.
- [12] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN. *SIGCOMM Comput. Commun. Rev.*, 43(4):99–110, Aug. 2013.
- [13] Boston University RINA Lab. <http://csr.bu.edu/rina/>.

- [14] N. M. K. Chowdhury and R. Boutaba. A Survey of Network Virtualization. *Computer Networks*, 54(5):862 – 876, 2010.
- [15] J. Day. *Patterns in Network Architecture: A Return to Fundamentals*. Prentice Hall, 2008.
- [16] J. Day, I. Matta, and K. Mattar. Networking is IPC: A Guiding Principle to a Better Internet. In *Proceedings of ReArch'08 - Re-Architecting the Internet (co-located with CoNEXT)*, New York, NY, USA, 2008.
- [17] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. RouteBricks: Exploiting Parallelism to Scale Software Routers. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSPP '09, pages 15–28, New York, NY, USA, 2009. ACM.
- [18] D. Drutskey, E. Keller, and J. Rexford. Scalable Network Virtualization in Software-Defined Networks. *Internet Computing, IEEE*, 17(2):20–27, March 2013.
- [19] R. Enns. NETCONF Configuration Protocol . *RFC 4741*, 2006.
- [20] ETSI. Network Functions Virtualisations (NFV) - White Paper. [https://portal.etsi.org/Portals/0/TBpages/NFV/Docs/NFV\\_White\\_Paper3.pdf](https://portal.etsi.org/Portals/0/TBpages/NFV/Docs/NFV_White_Paper3.pdf).
- [21] J. Fan and M. H. Ammar. Dynamic Topology Configuration in Service Overlay Networks: A Study of Reconfiguration Policies. In *INFOCOM*, 2006.
- [22] N. Feamster, J. Rexford, and E. Zegura. The Road to SDN. *ACM Queue*, 11(12):20, 2013.
- [23] A. D. Ferguson, A. Guha, C. Liang, R. Fonseca, and S. Krishnamurthi. Participatory Networking: An API for Application Control of SDNs. *SIGCOMM Comput. Commun. Rev.*, 43(4):327–338, Aug. 2013.
- [24] J. Galán-Jiménez and A. Gazo-Cervero. Overlay Networks: Overview, Applications and Challenges. *IJCSNS*, 10(12):40, 2010.
- [25] GENI. <http://www.geni.net/>.
- [26] E. Grasa, B. Gastn, S. van der Meer, M. Crotty, and M. A. Puente. Simplifying Multi-layer Network Management with RINA. In *TNC 2016*, June 2016.
- [27] C. M. Grinstead and J. L. Snell. *Introduction to Probability*. American Mathematical Soc., 2012.
- [28] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. NOX: Towards an Operating System for Networks. *SIGCOMM Comput. Commun. Rev.*, 38(3):105–110, July 2008.
- [29] Hadoop. <http://hadoop.apache.org/>.

- [30] J. Han, D. Watson, and F. Jahanian. Topology Aware Overlay Networks. In *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE*, volume 4, pages 2554–2565. IEEE, 2005.
- [31] IBM CPLEX Optimizer. <http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/>.
- [32] ITU. ITU-T G.709: Interfaces for the Optical Transport Network. <https://www.itu.int/rec/T-REC-G.709/en>, 2003.
- [33] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat. B4: Experience with a Globally-deployed Software Defined Wan. *SIGCOMM CCR*, 43(4):3–14, Aug. 2013.
- [34] J. Jannotti, D. K. Gifford, K. L. Johnson, M. F. Kaashoek, et al. Overcast: Reliable Multicasting with an Overlay Network. In *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation-Volume 4*, pages 14–14. USENIX Association, 2000.
- [35] X. Jin, J. Gossels, J. Rexford, and D. Walker. CoVisor: A Compositional Hypervisor for Software-defined Networks. In *NSDI 2015*, Berkeley, CA, USA. USENIX Association.
- [36] M. Jude. Policy-based Management: Beyond The Hype. *Business Communication Review*, pages 52–56, 2001.
- [37] M. Kamel, C. Scoglio, and T. Easton. Optimal Topology Design for Overlay Networks. *NETWORKING 2007. Ad Hoc and Sensor Networks, Wireless Networks, Next Generation Internet*, pages 714–725, 2007.
- [38] N. Kang, Z. Liu, J. Rexford, and D. Walker. Optimizing the "One Big Switch" Abstraction in Software-defined Networks. In *CoNEXT 2013*, pages 13–24, New York, NY, USA, 2013.
- [39] Y. Kanizo, D. Hay, and I. Keslassy. Palette: Distributing tables in software-defined networks. In *INFOCOM 2013*.
- [40] A. D. Keromytis, V. Misra, and D. Rubenstein. SOS: Secure Overlay Services. In *Proceedings of the 2002 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, SIGCOMM '02*, pages 61–72, New York, NY, USA, 2002. ACM.
- [41] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker. Onix: A Distributed Control Platform for Large-scale Production Networks. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation, OSDI'10*, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

- [42] J. Kurian and K. Sarac. A Survey on the Design, Applications, and Enhancements of Application-layer Overlay Networks. *ACM Comput. Surv.*, 43(1):5:1–5:34, Dec. 2010.
- [43] Z. Li and P. Mohapatra. On Investigating Overlay Service Topologies. *Computer Networks*, 51(1):54–68, 2007.
- [44] LIVE555 Media Server. <http://www.live555.com/>.
- [45] J. McCauley, A. Panda, M. Casado, T. Koponen, and S. Shenker. Extending SDN to large-scale networks. *Open Networking Summit*, 2013.
- [46] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, Mar. 2008.
- [47] A. Medina, A. Lakhina, I. Matta, and J. Byers. BRITE: An Approach to Universal Topology Generation. In *MASCOTS 2001*.
- [48] M. Moshref, M. Yu, A. Sharma, and R. Govindan. Scalable Rule Management for Data Centers. In *NSDI 2013*, Berkeley, CA, USA. USENIX Association.
- [49] National Institute of Standards and Technology. The NIST Definition of Cloud Computing, 2011.
- [50] NetEm. Linux Foundation. <http://www.linuxfoundation.org/collaborate/workgroups/networking/netem>.
- [51] X. N. Nguyen, D. Saucez, C. Barakat, and T. Turletti. Optimizing Rules Placement in OpenFlow Networks: Trading Routing for Better Efficiency. In *ACM SIGCOMM HotSDN 2014*, Chicago, USA.
- [52] B. Niven-Jenkins, D. Brungard, M. Betts, N. Sprecher, and S. Ueno. Requirements of an MPLS Transport Profile. RFC 5654, 2009.
- [53] Open Networking Foundation. <https://www.opennetworking.org/>.
- [54] Open Networking Foundation White Paper. Software-Defined Networking: The New Norm for Networks, April, 2012.
- [55] Open Stack. Neutron Project. <https://wiki.openstack.org/wiki/Neutron>.
- [56] Open vSwitch. <http://www.openvswitch.org/>.
- [57] Pouzin Society. RINA Specification Handbook, 2013.
- [58] Protocol Buffers. <https://developers.google.com/protocol-buffers/>.
- [59] K. Ratnam and I. Matta. WTCP: An Efficient Mechanism for Improving Wireless Access to TCP Services. *International journal of communication systems*, 16(1):47–62, 2003.

- [60] B. Salisbury. TCAMs and OpenFlow - What Every SDN Practitioner Must Know. <https://www.sdxcentral.com/articles/contributed/sdn-openflow-tcam-need-to-know/2012/07/>, 2012.
- [61] E. Salvadori, R. Doriguzzi Corin, A. Broglio, and M. Gerola. Generalizing Virtual Network Topologies in OpenFlow-Based Networks. In *Global Telecommunications Conference (GLOBECOM 2011), 2011 IEEE*, pages 1–6, Dec 2011.
- [62] C. A. Santiv  nez, R. Ramanathan, and I. Stavrakakis. Making Link-state Routing Scale for Ad Hoc Networks. In *Proceedings of the 2Nd ACM International Symposium on Mobile Ad Hoc Networking and Computing*, pages 22–32, New York, NY, USA, 2001. ACM.
- [63] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar. FlowVisor: A Network Virtualization Layer. In *Technical Report, OpenFlow-TR-2009-1, OpenFlow Consortium*, 2009.
- [64] M. Sindelar, R. K. Sitaraman, and P. Shenoy. Sharing-aware Algorithms for Virtual Machine Colocation. In *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '11*, pages 367–378, New York, NY, USA, 2011. ACM.
- [65] Spark. <http://spark.apache.org/>.
- [66] J. Strassner. *Policy-based network management: solutions for the next generation*. Morgan Kaufmann, 2003.
- [67] L. Subramanian, I. Stoica, H. Balakrishnan, and R. H. Katz. OverQoS: An Overlay Based Architecture for Enhancing Internet QoS. In *NSDI*, volume 4, page 6, 2004.
- [68] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. J. Minden. A Survey of Active Network Research. *Comm. Mag.*, 35(1):80–86, Jan. 1997.
- [69] The SPARC Project: Split Architecture for Carrier Grade Networks. <http://www.fp7-sparc.eu/>.
- [70] S. L. Vieira and J. Liebeherr. Topology Design for Service Overlay Networks with Bandwidth Guarantees. In *Quality of Service, 2004. IWQOS 2004. Twelfth IEEE International Workshop on*, pages 211–220. IEEE, 2004.
- [71] VLC Media Player. <http://www.videolan.org/>.
- [72] A. Voellmy, J. Wang, Y. R. Yang, B. Ford, and P. Hudak. Maple: Simplifying sdn programming using algorithmic policies. *SIGCOMM Comput. Commun. Rev.*, 43(4):87–98, Aug. 2013.
- [73] Y. Wang. ProtoRINA 2.0. <http://csr.bu.edu/rina/protorina/2.0>, 2016.

- [74] Y. Wang, N. Akhtar, and I. Matta. Programming Routing Policies for Video Traffic. In *International Workshop on Computer and Networking Experimental Research using Testbeds (CNERT 2014), co-located with ICNP 2014*, Raleigh, NC, USA, October 2014.
- [75] Y. Wang, F. Esposito, I. Matta, and J. Day. Recursive InterNetworking Architecture (RINA) Boston University Prototype Programming Manual. In *Technical Report BUCS-TR-2013-013, Boston University*, 2013.
- [76] Y. Wang, F. Esposito, I. Matta, and J. Day. RINA: An Architecture for Policy-Based Dynamic Service Management. In *Technical Report BUCS-TR-2013-014, Boston University*, 2013.
- [77] Y. Wang and I. Matta. A Recursive Approach to Network Management. In *Technical Report BUCS-TR-2015-014, Boston University*, 2015.
- [78] Y. Wang and I. Matta. SDN Management Layer: Design Requirements and Future Direction. In *Workshop on COntrol, Operation, and appLication in SDN Protocols (CoolSDN 2014), co-located with ICNP 2014*, Raleigh, NC, USA, October 2014.
- [79] Y. Wang, I. Matta, and N. Akhtar. Application-Driven Network Management with ProtoRINA. In *Proceedings of the IEEE/IFIP Network Operations and Management Symposium (NOMS)*, April 2016.
- [80] Y. Wang, I. Matta, F. Esposito, and J. Day. Introducing ProtoRINA: A Prototype for Programming Recursive-Networking Policies. *ACM SIGCOMM Computer Communication Review (CCR)*, July 2014.
- [81] R. W. Watson. Timer-based Mechanisms in Reliable Transport Protocol Connection Management. *Computer Networks (1976)*, 5(1):47–56, 1981.
- [82] L. Yang, R. Dantu, T. Anderson, and R. Gopal. Forwarding and control element separation (ForCES) framework. *RFC 3746, April*, 2004.
- [83] M. Yu, J. Rexford, M. J. Freedman, and J. Wang. Scalable Flow-based Networking with DIFANE. In *Proceedings of the ACM SIGCOMM 2010 Conference, SIGCOMM '10*, pages 351–362, New York, NY, USA, 2010. ACM.
- [84] L. Zhou and A. Sen. Topology Design of Service Overlay Network with a Generalized Cost Model. In *Global Telecommunications Conference, 2007. GLOBECOM'07. IEEE*, pages 75–80. IEEE, 2007.
- [85] Y. Zhu. RINA Video Streaming Application Proxies. [https://github.com/yuezhu/rina\\_video\\_streaming](https://github.com/yuezhu/rina_video_streaming).

## Curriculum Vitae

### Yuefeng Wang

<i>Address</i>	Department of Computer Science, Boston University 111 Cummington Mall, MCS-138, Boston, MA 02215, USA
<i>Email</i>	wyf@bu.edu
<i>Website</i>	<a href="http://blogs.bu.edu/wyf/">http://blogs.bu.edu/wyf/</a>
<i>Education</i>	<p><b>PhD</b> in Computer Science          · Boston University, Boston, MA, USA Sep 2010 – Dec 2016          · Advisor: Prof. Abraham Matta</p> <p><b>Master of Science</b> in Computer Science          · University of Windsor, Windsor, ON, Canada Sep 2008 – Jun 2010          · Advisor: Prof. Dan Wu</p> <p><b>Bachelor of Engineering</b> in Computer Science and Technology          · Shandong University, Jinan, Shandong, China Sep 2004 – Jun 2008          · Advisor: Prof. Shengfa Gao</p>
<i>Research Interests</i>	Computer Network Management, Multi-layer Network Design, Future Network Architecture, Software-Defined Networking (SDN), Network Function Virtualization (NFV), Content Delivery Networking (CDN)
<i>Teaching Experiences</i>	<p><b>Teaching Fellow</b>, Spring 2016, Boston University          · CS 103: Introduction to Internet Technologies and Web Programming</p> <p><b>Teaching Fellow</b>, Fall 2015, Boston University          · CS 455/655: Introduction to Computer Networks</p> <p><b>Lecturer</b>, Summer 2015, Boston University          · CS 112: Introduction to Computer Science II</p> <p><b>Teaching Fellow</b>, Fall 2014, Boston University          · CS 455/655: Introduction to Computer Networks</p> <p><b>Teaching Assistant</b>, Winter 2010, University of Windsor          · 03-60-104: Computer Concepts for End-Users</p> <p><b>Teaching Assistant</b>, Fall 2009, University of Windsor          · 03-60-100: Key Concepts in Computer Science</p>

**Teaching Assistant**, Winter 2009, University of Windsor  
 · 03-60-322: Object Oriented Analysis and Design

**Teaching Assistant**, Fall 2008, University of Windsor  
 · 03-60-415: Advanced and Practical Database System

*Working  
Experiences*

**Research Assistant**, Sep 2010 - Dec 2016  
 · Boston University, Boston, MA USA

**Research Intern**, Jun 2012 - Aug 2012/May 2016 - Aug 2016  
 · Akamai Technologies, Cambridge, MA USA

**Research Assistant**, Sep 2009 - Apr 2010  
 · University of Windsor, Windsor, ON Canada

*Publications*

1. **Yuefeng Wang**, Abraham Matta, and Nabeel Akhtar. “Application-Driven Network Management with ProtoRINA”. IEEE/IFIP Network Operations and Management Symposium (NOMS 2016), Istanbul, Turkey, April 2016.
2. Nabeel Akhtar, Abraham Matta, and **Yuefeng Wang**. “Managing NFV using SDN and Control Theory”. IEEE/IFIP International Workshop on Management of the Future Internet (ManFI 2016), co-located with NOMS 2016, Istanbul, Turkey, April 2016.
3. Flavio Esposito, Abraham Matta, and **Yuefeng Wang**. “VINEA: An Architecture for Virtual Network Embedding Policy Programmability”. IEEE Transactions on Parallel and Distributed Systems (TPDS), February 2016.
4. **Yuefeng Wang** and Abraham Matta. “A Recursive Approach to Network Management”. Technical Report BUCS-TR-2015-014, Boston University, December 2015.
5. **Yuefeng Wang** and Abraham Matta. “SDN Management Layer: Design Requirements and Future Direction”. Workshop on Control, Operation, and Application in SDN Protocols (CoolSDN 2014), co-located with ICNP 2014, Raleigh, NC, October 2014.
6. **Yuefeng Wang**, Nabeel Akhtar, and Abraham Matta. “Programming Routing Policies for Video Traffic”. Workshop on Computer and Networking Experimental Research using Testbeds (CNERT 2014), co-located with ICNP 2014, Raleigh, NC, October 2014.
7. **Yuefeng Wang**, Abraham Matta, Flavio Esposito, and John Day. “Introducing ProtoRINA: A Prototype for Programming Recursive-Networking Policies”. Editorial Note at ACM SIGCOMM Computer Communication Review. Issue of July 2014.
8. **Yuefeng Wang**, Abraham Matta, and Nabeel Akhtar. “Experimenting with Routing Policies Using ProtoRINA over GENI”. Third GENI Research and Educational Experiment Workshop (GREE 2014), Atlanta, GA, March 2014.

9. **Yuefeng Wang**, Flavio Esposito, Abraham Matta and John Day. “RINA: An Architecture for Policy-Based Dynamic Service Management”. Technical Report BUCS-TR-2013-014, Boston University, November 2013.
10. **Yuefeng Wang**, Flavio Esposito, Abraham Matta, and John Day. “Recursive InterNetworking Architecture (RINA) Boston University Prototype Programming Manual”. Technical Report BUCS-TR-2013-013, Boston University, November 2013.
11. Flavio Esposito, **Yuefeng Wang**, Abraham Matta, and John Day. “Dynamic Layer Instantiation as a Service”. Demo in the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI), April 2013.
12. **Yuefeng Wang**, Flavio Esposito, and Abraham Matta. “Demonstrating RINA Using the GENI Testbed”. GENI Research and Educational Experiment Workshop (GREE 2013), Salt Lake City, Utah, March 2013.