Universidade de Aveiro
2010

Departamento de
Electrónica, Telecomunicações e Informática.

João Pedro Brites
Ferreira Nogueira

DEMONSTRAÇÃO DE CRIAÇÃO DE REDES VIRTUAIS NO ÂMBITO DO OPERADOR

Universidade de Aveiro Departamento de
2010 Electrónica, Telecomunicações e Informática.

João Pedro Brites Ferreira Nogueira

DEMONSTRAÇÃO DE CRIAÇÃO DE REDES VIRTUAIS NO ÂMBITO DO OPERADOR

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia de Electrónica e Telecomunicações, realizada sob a orientação científica da Professora Dra. Susana Sargento, Professora Auxiliar do Departamento de Electrónica, Telecomunicações e Informática da Universidade de Aveiro.

o júri / the jury

presidente / president	**Professor Doutor António Rui de Oliveira e Silva Borges** Professor Associado do Departamento de Electrónica, Telecomunicações e Informática da Universidade de Aveiro
vogais / examiners committee	**Professora Doutora Susana Isabel Barreto de Miranda Sargento** Professora Auxiliar do Departamento de Electrónica, Telecomunicações e Informática da Universidade de Aveiro
	Professora Doutora Maria Solange Pires Ferreira Rito Lima Professora Auxiliar do Departamento de Informática da Escola de Engenharia da Universidade do Minho

agradecimentos / acknowledgements

Esta tese marca o culminar de 5 anos de trabalho na Universidade de Aveiro. Durante estes fantásticos anos, muitas foram as pessoas que me marcaram, incentivaram e ajudaram a desenvolver técnica e pessoalmente.

Quero agradecer aos meus orientadores, Prof. Dra. Susana Sargento e Engenheiro Jorge Carapinha, da PT Inovação, pelo apoio, visão e orientação fornecidos ao longo destes meses. A conjugação das suas diferentes perspectivas e abordagens foi uma mais-valia para o trabalho desenvolvido.

Ao Mestre Márcio Melo, pela presença diária, abundantes discussões e partilha de conhecimentos, fundamentais para a minha progressão e evolução.

À Portugal Telecom Inovação e ao seu departamento de Investigação Aplicada e Disseminação do Conhecimento 1 (IAD1), pelo apoio, oportunidades e condições providenciadas.

A todos os meus amigos pelo acompanhamento constante, momentos de descompressão e motivação.

À minha namorada pela sua enorme paciência, compreensão e encorajamento.

Por fim, quero agradecer a toda a minha família, em particular aos meus Pais, Irmãos e Avós pela confiança depositada nas minhas capacidades e apoio ao longo de todos os anos.

palavras-chave

Virtualização, Redes Virtuais, Mapeamento, Descoberta Distribuída, 4WARD, Internet do Futuro, Xen, VMware

Resumo

A Internet nunca foi pensada para suportar a multiplicidade de serviços e a quantidade de utilizadores que tem actualmente. Conjugando este facto com uma crescente exigência quer a nível de desempenho, quer a nível de flexibilidade e robustez, facilmente se percebe que a arquitectura actual não corresponde nem às necessidades e exigências dos utilizadores actuais nem dos futuros.

A virtualização de rede é, assim, apresentada como uma possível solução para este problema. Ao permitir que um conjunto de redes com requisitos e arquitecturas distintos, optimizados para diferentes aplicações, partilhem uma mesma infra-estrutura e sejam independentes desta, permitirá o desenvolvimento de alternativas que minimizem ou suprimam as limitações conhecidas da Internet actual.

O facto de uma mesma rede física poder ser utilizada para suportar múltiplas redes virtuais é de grande interesse para os operadores. Ao melhorar a utilização da infra-estrutura e a consolidação de recursos, é possível aumentar a rentabilidade da mesma. Além desta mais eficiente utilização, que se traduz numa vantagem competitiva, a virtualização de rede permite o aparecimento de novos modelos de negócio através da dissociação entre serviços e a rede física.

Neste sentido, e no âmbito do projecto 4WARD, esta dissertação propõe-se a desenvolver uma plataforma de virtualização que permita a avaliação, resolução de problemas e testes referentes à criação, monitorização e gestão de redes virtuais existentes numa rede física experimental.

Foram desenvolvidas funcionalidades dinâmicas de monitorização de rede, através das quais é possível detectar situações de falhas, sobre utilização ou problemas de configuração. Também foram desenvolvidos, simulados e implementados algoritmos distribuídos de descoberta de redes físicas e virtuais. Na vertente de gestão da rede, foram implementados mecanismos que permitem actuar sobre os recursos virtuais. Por fim, para que a criação inteligente de redes virtuais fosse possível e efectuada o mais rapidamente possível, foram desenvolvidos algoritmos de mapeamento dinâmico de redes virtuais e optimizados os processos de criação dos respectivos nós.

Por forma a disponibilizar e testar as funcionalidades, foi desenvolvida uma plataforma de virtualização que fornece um ambiente gráfico e que permite, de forma intuitiva, desenhar e configurar redes virtuais, monitorizar as redes existentes em tempo real e actuar sobre elas. Esta plataforma foi desenvolvida de forma modular e poderá servir como base para futuros melhoramentos e funcionalidades.

Os resultados obtidos, além de implementarem as funcionalidades desejadas e de comprovarem a escalabilidade da arquitectura e dos algoritmos propostos, provam que é possível a existência de uma ferramenta única de gestão, monitorização e criação de redes virtuais.

keywords Virtualization, Virtual Networks, Embedding, Mapping, Distributed Discovery, 4WARD, Future Internet, Xen, VMware

abstract

The Internet was never designed to support the huge amount of services and users that it has nowadays. Combined with ever-increasing requirements for performance, flexibility, and robustness, one can easily realize that the current architecture does not match neither the needs nor the demands of the current and future users.

Network virtualization arises as a potential solution for these issues. By letting multiple networks, optimized for different applications with different requirements and architectures, to coexist and share the same infrastructure in an independent way, new alternatives may be developed that bypass the known limitations of the current Internet.

This ability to use the same physical infrastructure to hold multiple virtual networks is of great interest for network operators. By improving its infrastructure utilization and increasing the resource consolidation, higher profitability can be achieved. Besides this competitive advantage, network virtualization enables new business models and the dissociation of the provided services from the physical network.

With that goal in mind, this Thesis, in the scope of the 4WARD project, presents a virtualization platform that will enable the evaluation and solving of the inherent problems associated with the creation, monitoring and management of virtual networks, embedded in an experimental physical network.

The developed dynamic monitoring features make the detection of failures, misconfigurations or overloads possible. In addition, physical and virtual network discovery mechanisms were designed, simulated and implemented. Regarding network management, acting upon virtual resources was also made possible. Finally, in order to optimize and speed-up virtual network creation, dynamic mapping algorithms and optimized node creation processes were developed.

In order to provide and test the specified features, a network virtualization platform was developed containing a graphical user interface that aims to provide the users with a simple, interactive, intuitive way of designing and configuring virtual networks, as well as monitoring and managing them. The developed platform poses itself as a possible platform for future enhancements and added functionalities, due to its modular nature.

The attained results, besides implementing the desired features and having proven the scalability and feasibility of the proposed algorithms, are also the evidence that the existence of a single tool to manage, monitor and create virtual networks is feasible.

Contents

Contents i

List of Figures v

List of Tables ix

Acronyms xi

1 Introduction & Overview 1
- 1.1 Motivation 1
  - 1.1.1 Internet 's Origins and Evolution 1
  - 1.1.2 Current Context 2
  - 1.1.3 Enabling Innovation 3
- 1.2 Purpose 3
- 1.3 Contribution 4
- 1.4 Thesis Outline 4

2 State of the Art 7
- 2.1 Overview 7
- 2.2 Server Virtualization 7
  - 2.2.1 Advantages 8
  - 2.2.2 Disadvantages 9
  - 2.2.3 Virtual Machine Monitor 10
  - 2.2.4 IA-32 Virtualization 11
  - 2.2.5 Virtualization Techniques 12
  - 2.2.6 Virtual Appliances 15
  - 2.2.7 Analysis of Server Virtualization Tools 15
  - 2.2.8 Libvirt: Virtualization API 18
  - 2.2.9 Summary 19
- 2.3 Network Virtualization 20
  - 2.3.1 Design Goals 20
  - 2.3.2 Proposed Business Models 21
  - 2.3.3 Existing Technologies 22
  - 2.3.4 Existing Initiatives 26
  - 2.3.5 Mechanisms for Network Virtualization Support 27
  - 2.3.6 Summary 34
- 2.4 Network Virtualization Platforms 34

2.4.1 GENI 34
2.4.2 VNet Management Demonstrator v0.1 36

3 Platform Requirements Specification **41**
3.1 Introduction 41
3.2 Overall Description 41
3.2.1 Features 41
3.2.2 User Classes 42
3.2.3 Operating Environment 42
3.2.4 Constraints 43
3.2.5 Assumptions and Dependencies 43
3.3 System Features Details 43
3.3.1 Physical and Virtual Resource and Topology Discovery 43
3.3.2 Substrate and Virtual Network Monitoring 43
3.3.3 Virtual Network Creation 44
3.3.4 Virtual Network Management 44
3.4 Interface Requirements 45
3.4.1 Use Cases 45
3.4.2 User Interface 46
3.4.3 Software Interfaces 47
3.4.4 Communication Interfaces 47
3.5 Non-functional Requirements 47
3.5.1 Performance 47
3.5.2 Security 47
3.5.3 Software Quality Attributes 47
3.6 Conclusions 48

4 Architecture & Mechanisms Design **49**
4.1 Introduction 49
4.2 Module Decomposition 49
4.2.1 Control Centre module 49
4.2.2 Manager module 50
4.2.3 Agent module 50
4.3 Data Decomposition 50
4.3.1 Control Centre Data Decomposition 50
4.3.2 Manager Data Decomposition 51
4.3.3 Agent Data Decomposition 51
4.4 Dependencies 51
4.4.1 Control Centre Dependencies 51
4.4.2 Manager Dependencies 51
4.4.3 Agent Dependencies 51
4.5 Interface Description 51
4.5.1 User – Control Centre Interface 51
4.5.2 Manager – Control Centre Interface 51
4.5.3 Agent - Manager Interface 52
4.6 Identification Process 52
4.7 Virtual Network Creation 53

4.7.1 Topology and Configuration 53
4.7.2 Virtual Network Mapping 53
4.7.3 Virtual Resource Creation 55
4.7.4 Simulation Results 56
4.8 Topology Discovery 61
4.8.1 Distributed Topology Discovery 61
4.8.2 Centralized Topology Discovery 66
4.8.3 Simulation Results 68
4.9 Substrate and Virtual Network Monitoring 70
4.10 Virtual Network Management 71
4.11 Conclusions 72

5 Software Implementation 75
5.1 Introduction 75
5.2 Auxiliary Functions and Libraries 75
5.2.1 XML parsing 75
5.2.2 popen_noshell 75
5.2.3 libvirt 75
5.3 Virtual Network Control Centre 76
5.3.1 Databases and Classes 76
5.3.2 Module Decomposition 77
5.3.3 Virtual Network Design & Configuration 81
5.3.4 Virtual Network Monitoring 85
5.3.5 Virtual Network Management 85
5.4 Virtual Network Manager 86
5.4.1 Main Databases and Structures 86
5.4.2 Module Decomposition 86
5.4.3 Virtual Network Mapping 90
5.5 Virtual Network Agent 93
5.5.1 Main Databases and Structures 93
5.5.2 Module Decomposition 93
5.5.3 Resource Data Gathering 100
5.5.4 Virtual Network Creation 101
5.6 Conclusion 102

6 Tests & Results 105
6.1 Introduction 105
6.2 Testbed Description & General Assumptions 105
6.3 Data Gathering 107
6.3.1 Cold Boot 107
6.4 Network Discovery 108
6.4.1 Cold Network Discovery 108
6.4.2 Hot Network Discovery 109
6.5 Virtual Network Mapping & Creation 111
6.5.1 Virtual Network Mapping 111
6.5.2 Virtual Network Creation 112
6.6 Conclusions 113

7 Conclusions **115**
7.1 Final Conclusion 115
7.2 Future Work 116

Bibliography **119**

List of Figures

1.1 Cisco IP Traffic Forecast 2

2.1 Server Consolidation Through Virtualizaton 9
2.2 Type 1 Hypervisor Architecture 11
2.3 Type 2 Hypervisor Architecture 11
2.4 Full Virtualization on the IA-32 architecture 13
2.5 Paravirtualization on the IA-32 architecture 14
2.6 Hardware-Assisted Virtualization on the IA-32 architecture 15
2.7 Virtual Machine vs. Virtual Appliance 15
2.8 Xen's Architecture 16
2.9 Libvirt: Virtualization API 18
2.10 Libvirt control methods. 19
2.11 SP and InP business model 22
2.12 SP, VNP, VNO and InP business model 23
2.13 MPLS Packet Labelling and Label Swapping 24
2.14 Consumer vs. Provider Edge based VPN 25
2.15 Cisco Nexus 1000V Architecture [15] 30
2.16 VNet Demonstrator v0.1 Testbed 36
2.17 VNet Demonstrator v0.1 Architecture 37
2.18 VNet Demonstrator v0.1 Agent detail 38
2.19 VNet Demonstrator v0.1 Manager detail 39

3.1 Simplified VNet Creation use-cases 45
3.2 Simplified VNet Management and Monitoring use-cases 46

4.1 Global view of the existing modules. 50
4.2 Virtual Network Control Center - User Interface. 52
4.3 Agent and Control Centre ID attribution process. 53
4.4 VNet Creation use-cases 54
4.5 Virtual Resource creation. 56
4.6 Virtual Network Mapping Simulation Scenario 1 - Maximum accepted Virtual Networks. 58
4.7 Virtual Network Mapping Simulation Scenario 2 59
4.8 Virtual Network Mapping Simulation Scenario 3 - Maximum accepted Virtual Networks. 61
4.9 Topology Discovery – Assembling neighbourhood knowledge. 62
4.10 Virtual topology discovery example 63

4.11 Discovery algorithm - Bootstrap diagram 66
4.12 Discovery algorithm – Resource message received 67
4.13 Discovery algorithm – Delete resource message received 67
4.14 Distributed discovery algorithm simulation example. 69
4.15 Discovery Algorithm Scalability Tests – Number of Physical Nodes. 70
4.16 Discovery Algorithm Scalability Tests – Number of Virtual Networks. 71
4.17 VNet Management and Monitoring use-cases 72

5.1 Control Centre's Classes and Lists. 76
5.2 Control Centre's VNet list hash map and class. 76
5.3 Control Centre's Resource and Link classes. 77
5.4 Control Centre's Display classes. 78
5.5 Control Centre Start Up. 78
5.6 Control Centre Module. 79
5.7 Control Centre Model Thread Diagram. 79
5.8 Control Centre View Thread Diagram. 79
5.9 Control Centre's Dropdown & Coolbar menus. 81
5.10 Control Centre's Canvas. 81
5.11 Control Centre View Thread main loop Diagram. 81
5.12 Control Centre's Radial resource placement. 82
5.13 Control Center - New Resource Diagram. 82
5.14 Configuring a new Virtual Resource 83
5.15 Control Center - New Link Diagram. 84
5.16 Configuring a new Link 84
5.17 Control Center - Commit & Save Diagrams. 84
5.18 Control Center - Get VNet Diagram. 85
5.19 Control Center - Virtual Node Monitoring. 85
5.20 Manager's VNet Entry. 86
5.21 Connected Agents and Control Centres Entries. 87
5.22 Manager - Start-Up. 87
5.23 Manager Module 88
5.24 Manager - Agent Connection Accept Thread. 88
5.25 Manager - Control Centre Connection Accept thread. 89
5.26 Manager - Status Update thread. 90
5.27 Manager - Command Send thread. 90
5.28 Manager - Receive XML Diagram. 92
5.29 Manager - Map Virtual Network Diagram. 92
5.30 Node and Link Discovery Entries. 93
5.31 Agent Module 94
5.32 Agent - Start-Up diagram. 94
5.33 Agent - Manager Connection thread diagram. 95
5.34 Agent - Keep Alive and ID Request thread diagram. 95
5.35 Agent - Template thread diagram. 96
5.36 Agent - Status Update thread diagram. 96
5.37 Agent - Update Resource Information diagram. 97
5.38 Agent - Delete Resource diagram. 97
5.39 Agent - Status Send thread diagram. 97

5.40 Agent - Link Management thread diagram. 98
5.41 Agent - Link Management thread main loop diagram. 98
5.42 Agent - Passive Link Discovery thread diagram. 99
5.43 Agent - Active Link Discovery thread diagram. 99
5.44 Agent - Neighbour Hello thread diagram. 99
5.45 Agent - Create new Bridge. 101
5.46 Agent - Create Virtual Node Diagram. 102
5.47 Agent - Virtual Node Request Diagram. 102

6.1 Testbed Network. 106
6.2 Reference Virtual Network. 107
6.3 Agent Cold boot results. 108
6.4 Cold network discovery results. 109
6.5 Distributed vs. Centralized network discovery results. 110
6.6 Virtual Network Mapping results. 111
6.7 Virtual Network Creation results. 113

List of Tables

4.1 Virtual Network Mapping- Virtual Nodes' parameters pool. 57
4.2 Virtual Network Mapping Simulation Scenario 1- Physical Nodes' parameters pool. 57
4.3 Virtual Network Mapping Simulation Scenario 2- Number of embedded virtual networks. 59
4.4 Virtual Network Mapping Simulation Scenario 3- Physical Nodes' parameters pool with doubled node capacity. 60
4.5 Virtual Network Mapping Simulation Scenario 3- Physical Nodes' parameters pool with doubled link capacity. 60
4.6 Distributed discovery - 1st simulation parameters. 68

5.1 Manager to Control Centre message types. 80
5.2 Control Centre to Manager message types. 80
5.3 Agent to Manager message types. 89
5.4 Manager to Agent message types. 91

6.1 Testbed specification. 106

Acronyms

AMD Advanced Micro Devices

API Application Programming Interface

ARPAnet Advanced Research Projects Agency Network

ATM Asynchronous Transfer Mode

BGP Border Gateway Protocol

CABO Concurrent Architectures are Better than One

CAN Content Addressable Network

CAPEX Capital Expenditure

CE Costumer Edge

CMS Cambridge Monitor System

CPU Central Processing Unit

CSPF Constrained Shortest Path First

CV Computer Virtualization

DNS Domain Name System

DoS Denial of Service

DR Designated Root

EPT Extended Page Tables

FEC Forwarding Equivalence Class

FTTH Fiber To The Home

GB Gigabytes

GENI Global Environment for Network Innovations

GGID GENI Global Identifier

GMC GENI Management Core

GMPLS	Generic Multi Protocol Label Switching
GP	Generic Path
GPS	Global Positioning System
GUI	Graphical User Interface
HDD	Hard Disk Drive
HVM	Hardware Virtual Machine
IDD	Isolated Driver Domain
IDE	Integrated Drive Electronics
InP	Infrastructure Provider
IOMMU	Input/Output Memory Management Unit
IP	Internet Protocol
IPTV	Internet Protocol Television
IPv6	Internet Protocol Version 6
ISA	Instruction Set Architecture
ISO	International Standards Organization
ISP	Internet Service Provider
I/O	Input / Output
KB	Kilobytes
L1VPN	Layer 1 VPN
L2	Layer 2
L2VPN	Layer 2 VPN
L3	Layer 3
L3VPN	Layer 3 VPN
LAN	Local Area Network
LER	Label Edge Router
LSR	Label Switching Router
LSP	Label-Switched Path
LVM	Logical Volume Management
MAC	Media Access Control

MIB	Management Information Base
MMU	Memory Management Unit
MPLS	Multi Protocol Label Switching
Mpps	Mega packets-per-second
MTU	Maximum Transmission Unit
NAT	Network Address Translation
NCP	Network Control Program
NFS	National Science Foundation
NGSP	Next Generation Service Provider
NPT	Nested Page Tables
NR	Neighbourhood Resource Availability
NV	Network Virtualization
NVE	Network Virtualization Environment
NVSS	Network Virtualization System Suite
NWGN	New Generation Network
ON	Overlay Network
OPEX	Operational Expenditure
OS	Operating System
OSI	Open Systems Interconnection
P2P	Peer-to-Peer
PDMA	Packet Division Multiple Access
PE	Provider Edge
PoP	Point-of-Presence
PPP	Point-to-Point Protocol
QoS	Quality of Service
RAM	Random Access Memory
RSVP	Resource Reservation Protocol
RVI	Rapid Virtualization Indexing
SDH	Synchronous Digital Hierarchy

SLA	Service Level Agreement
SMP	Symmetric Multiprocessing
SONET	Synchronous Optical Network
SP	Service Provider
SSH	Secure Shell
SSL	Secure Sockets Layer
STP	Spanning Tree Protocol
SWT	Standard Widget Toolkit
TCO	Total Cost of Ownership
TCP	Transmission Control Protocol
TDM	Time Division Multiplexing
TTL	Time-To-Live
UDP	User Datagram Protocol
US	United States
VC	Virtual Computer
VEM	Virtual Ethernet Module
VLAN	Virtual Local Area Network
VNCC	Virtual Network Control Centre
VoIP	Voice Over IP
VPN	Virtual Private Network
VM	Virtual Machine
VMM	Virtual Machine Monitor
VN	Virtual Network
VNO	Virtual Network Operator
VNP	Virtual Network Provider
VSM	Virtual Supervisor Module
XML	Extensible Markup Language

Chapter 1

Introduction & Overview

1.1 Motivation

1.1.1 Internet 's Origins and Evolution

The Internet has its origins in the first large-scale packet switching network, Advanced Research Projects Agency Network (ARPAnet). Originally, ARPAnet's communications were based on Network Control Program (NCP) [18], a protocol that combined addressing and transport. Although it performed quite well for the initially small network, scalability and flexibility concerns started to spread among the development community. Novel routing mechanisms were proposed [42] and it also became clear that separating transport from addressing was a requirement for a general-purpose network. In response to the scaling issues, in 1982 the Domain Name System (DNS) system was deployed and replaced the *host.txt* file for naming Internet systems [51, 25, 61].

On January 1st 1983, Transmission Control Protocol (TCP)/Internet Protocol (IP) was implemented and replaced NCP as the standard interface for network communications. This was a revolutionary step that required an update to all existing nodes, about four-hundreds of them [24]. This was probably the last time such an update was possible.

Since its inception, the Internet has mostly suffered evolutionary, rather than revolutionary, updates, since it is much easier to deploy a new protocol that fills a gap than it is to replace a protocol that despite not optimal, still works.

In a commercial network, change can only happen if the motivation is sufficient, i.e. if the economic benefit to be attained is significant or if collapse is eminent. Economic reasons are not typically the crucial factor when changing a core network, in part because interoperability between providers must be guaranteed, and changes that are interoperable do not differentiate an Internet Service Provider (ISP) from its competitors. Thus, the main driving force for change and innovation is the need to fix an immediate issue.

As the Internet's user base grew, so did the problems and requirements. For example, the initial implementation of TCP caused several network collapses due to congestion, since the network resources were operating at full capacity but no *useful work* was being done. The TCP retransmission strategy was clogging the network with unnecessary retransmitted packets. It became evident that congestion control was a necessity in a network [45]. Instead of trying to implement congestion control in a protocol-independent manner, a quick-fix was to implement congestion control mechanisms for TCP; such implementation was backwards compatible and did a good job at solving the issue.

Although efforts were made and standards were created to implement a lot of features that were missing from the original design such as security, multicast, Quality of Service (QoS), explicit congestion notification and mobile IP, most of these technologies have not been widely deployed. In spite of being useful, they solve problems that are not immediately pressing, and therefore are best described as *enhancements* rather than fixes to the architecture.

Even though many extensions failed, Multi Protocol Label Switching (MPLS) and Virtual Private Networks (VPNs) are examples of modifications that succeed, most likely because they provided workarounds to some limitations of the Internet protocols within an ISP.

1.1.2 Current Context

Today's Internet purpose is many-fold; it is widely used in business, defense, media and social connections. It has become a critical part of modern society and the current global economy relies heavily on it. This strong dependency will continue to increase as more and more services converge to a digital medium: circuit-switch telephony is being converted to Voice Over IP (VoIP); television broadcast using Internet Protocol Television (IPTV) is becoming common; cloud computing is gaining momentum; high definition content is becoming the norm and the user base keeps on growing. Recent forecasts from Cisco, predict that in 2013, the annual global IP traffic will exceed two-thirds of a Zettabyte (667 Exabyte) [16]. The expected traffic growth is shown in figure 1.1.

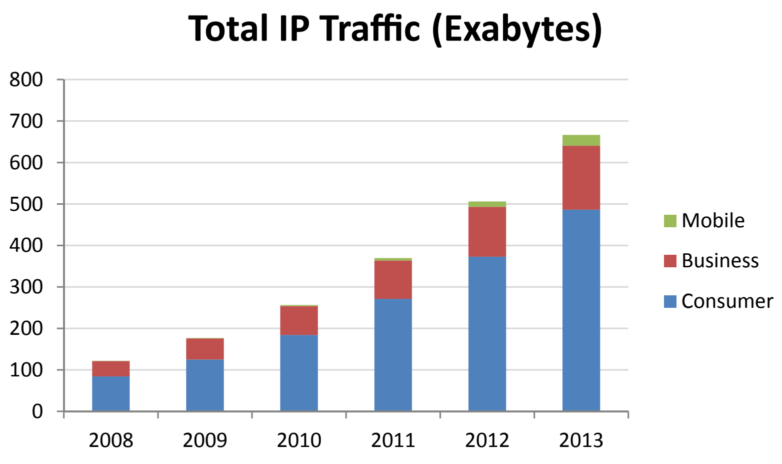


Figure 1.1: Cisco IP Traffic Forecast

This kind of strain on the existing Internet backbones may force the rethinking of the current architecture. Technologies developed to aid this convergence, such as QoS and IP Multicast, have not seen a wide adoption and their capacity may not be able to keep up with user demand.

The stagnation observed in the current core protocols of the Internet, which has been referred to as the *Internet ossification* [65], presents an obstacle to innovation by only allowing more efficient implementations of existing network layer protocols instead of providing the means for testing and deploying new ones. While the possibility exists for developing and deploying high layer protocols and new physical and link technologies, the network layer cannot be modified. Although IP has been designed to take into account future options to allow extensibility, its options remain largely unused due to the use of hardware-assisted routing: routing packets without options is much faster. The use of options could also lead to Denial of Service (DoS) attacks on the routers; therefore, packets using options are likely to be filtered [24]. Internet Protocol Version 6 (IPv6) tries to remedy the options issue by providing separate end-to-end options from hop-by-hop IP options, but its adoption has been very slow.

The Internet, thus, became a victim of its own success.

1.1.3 Enabling Innovation

If the Internet intends to get better, fundamental changes need to take place: innovation should not only be allowed but encouraged; there should be enough flexibility and isolation to allow experimental networks to be deployed without affecting other running networks; renewal and change must become ordinary processes.

This competition between novel networks and their respective protocols and architectures will allow the Internet to evolve and better suit their users and services' needs.

One way to build such a *diversified* [65] Internet could be by providing a versatile infrastructure that could be "sliced" to create several independent networks. These networks would then be able to run their own protocols and services and comply with different Service Level Agreements (SLAs). A controlled, isolated infrastructure sharing scheme could potentially reduce the complexity of testing, deploying and managing new networks, protocols and services while assuring legacy compatibility and a seamless integration with the existing ISPs and their networks. It would provide a non-disruptive solution to introduce disruptive technologies.

By implementing virtualization solutions on both the networks' nodes and links, it is possible to create virtual networks, which are by themselves the so called "slices" of the substrate network: they provide resource sharing, isolation and independency from the underlying physical network. The ability to create these virtual networks on-demand and on-the-fly is of great interest for network operators since it furnishes them with the means to create and provide custom-tailored networks to their customers while at the same time increasing the average resource usage of their network, thus enhancing their business advantages.

1.2 Purpose

The need for providing network operators with the tools to instantiate and manage virtual networks on top of existing physical networks was highlighted on the previous section. To that end, this thesis' purpose is many-fold: to design a virtualization platform and evaluate the respective mechanisms and algorithms that enable the embedding, management and monitoring of these virtual networks.

A virtualization platform, containing a graphical user interface, shall be created in order to test this solution on a physical testbed and prove the feasibility and applicability of network virtualization concepts.

The node and link virtualization concepts shall be explored and discussed. Several approaches, initiatives, existing products, tools, mapping and discovery algorithms will be considered. By studying these different parts required for network virtualization, a solution shall be found and implemented.

Special attention will be given to the discovery and mapping aspects of virtual networks. The developed algorithms shall be tested for their scalability properties, and conclusions will be made about their performance and overhead.

1.3 Contribution

As the result of the accomplishment of the proposed objectives, this Master's Thesis contributes with a virtualization platform that enables the deployment, management and monitoring of virtual networks running on a substrate network, as well as with innovative physical and virtual topology discovery and mapping algorithms.

The topology discovery algorithm implemented is simple, fast, scalable, and presents a low discovery overhead. It assures that up-to-date link information is always available and therefore allows proper network monitoring.

A heurist mapping algorithm that strives to optimize the virtual links' and nodes' placement was also proposed and evaluated, both through simulation and experiments.

The platform's Graphical User Interface (GUI) aggregates the developed functionalities and provides an easy, intuitive and interactive way of dynamically monitoring, managing and creating virtual networks.

This Thesis was developed within *Portugal Telecom Inovação*, and its participation in the ICT FP7 4WARD project. As a result of the work done on this Thesis, a demonstration was made in the final review meeting of the 4WARD project in Kista, Sweden. This Work Package 3 demonstration was chosen as the main work to show in this final evaluation, and was highly appreciated by both the European Commission's reviewers and the general audience.

In addition, a contribution was also made to 4WARD's 3rd Workpackage Deliverable 3.2.1 [2]; an internal paper regarding the virtualization framework and developed algorithms was accepted for publication in the magazine *Saber e Fazer* from Portugal Telecom Inovação; a paper regarding virtual network discovery was submitted to IEEE Globecom 2010 Workshop on Network of the Future.

In the future, it is also planned the writing of two papers regarding virtual network mapping and the virtualization platform.

1.4 Thesis Outline

Chapter 2 will begin by providing a global overview of virtualization, existing tools and technologies. Focus will be given not only on server virtualization, the enabler of the developed software, but also on network virtualization, its goals, challenges and existing initiatives. At the end of the chapter, the base software that stirred this thesis will be depicted.

The following chapter, Chapter 3, will deal with the requirements and specification aspects of the developed software. The main goals, desired features, interfaces, target users and performance requirements will be explored.

The Thesis will then proceed in Chapter 4 with providing an architecture to the virtualization platform and its associated mechanisms. In order to address the desired features, algorithms and mechanisms will be proposed. As a result, algorithms for dynamic topology discovery and virtual network mapping will be developed and tested through simulation.

On Chapter 5, a more detailed description of each one of the software's modules will be provided. Its main data structures, internal organization and mechanisms will be described and a thorough analysis will be made on how the primary functionalities were implemented.

Chapter 6 shows and examines the experimental results. These results shall validate the feasibility of the demonstrator, reveal that the proposed algorithms perform as predicted, and that the desired functionalities work as expected.

This Thesis will terminate on Chapter 7 with a final conclusion on the developed tool, associated mechanisms, algorithms and performance, based on the attained results. Finally, several suggestions will be made on how to improve the platform and add new functionalities.

Chapter 2

State of the Art

2.1 Overview

Although the main purpose of this thesis is to explore concepts and develop solutions for network virtualization, server virtualization works as an enabler for it and is thus discussed on section 2.2. The following section, 2.3, will deal with network virtualization aspects. On both cases, their main advantages, disadvantages and existing technologies will be discussed, although with a greater emphasis on network virtualization.

The chapter ends with an analysis of existing network virtualization platforms, on section 2.4.

Since this chapter will span many different concepts, the summary will be distributed throughout the several sections.

2.2 Server Virtualization

Back when computer systems were invented, most systems were large, expensive to operate and there was a great usage demand. Therefore, they had to evolve to become time-sharing systems so that multiple users could use them simultaneously. However, with a growing number of computers, users and applications, it became apparent that time-sharing was not always ideal. The misuse of the system by any user, intentional or not, could jeopardize all users and grind the entire system to a halt. For companies that could afford it, buying multiple computers mitigated these problems. Therefore, having multiple isolated systems was a wish of many organizations [55].

In order to respond to organizations wishes (and make some money out of it), the first research Hypervisor to offer full virtualization support was implemented on IBM's CP-40 system in January 1967 (which preceded the IBM's revolutionary CP-67/Cambridge Monitor System (CMS))[17]. It supported multiple instances of client Operating Systems (OSs), in particular the CMS. The virtualization increased robustness and stability, allowing beta and experimental OSs to be deployed and debugged without affecting other stable running OSs. This was a great advantage since there was no need for additional, expensive, development systems.

Once enterprise-grade restricted, server virtualization is getting more and more common in personal computers and workstations due to a great increase in computing performance in the past two decades. In 1993 Intel launched the Intel Pentium architecture, which was a true milestone in the personal computing history: it was the first superscalar x86 processor. It explored instruction level parallelism (pipeline) and exhibited an impressive performance for that time. Currently, dual-core Central Processing Units (CPUs) running in excess of 3GHz are common, and so are becoming quad, six, eight and recently released twelve-core (e.g. Advanced Micro Devices (AMD)'s Opteron 6174) processors effectively turning a desktop commodity computer into a powerhouse of computing performance. This increase in performance and computing resource availability brought consumer computers together with virtualization technologies and their advantages.

After analysing the virtualization's main virtues and issues, the following subsections will only deal with technologies and solutions available in the literature.

2.2.1 Advantages

Besides the resource consolidation aspect, there are many other advantages to be attained with server virtualization, some of them will be described next.

Safety

By separating environments with different security requirements using Virtual Machine (VM)s, one may select the OS that best matches the required services and tools. Therefore, a security attack on one system would not compromise the others, due to the isolation property.

Trust and availability

New or experimental versions of software may be tested on the hardware that they will later use without jeopardizing production workloads, and so, virtual systems may be used as low-cost test systems.

Optimize resource utilization

Since different workloads tend to show peak resource use at different times of the day and week, implementing multiple workloads in the same physical server can improve system utilization.

Additionally, because multiple OS types and releases may run on a single system, each virtual system may run the OS that best matches its application or user requirements, thus further improving the resource utilization. An example of workload consolidation may be observed in figure 2.1

Cost

Virtualization allows multiple workloads and systems to be combined into a single physical server, reducing the costs of hardware and operations. The usual approach is to consolidate small servers into more powerful ones. Studies show that the cost reduction may vary from 29% to 64% in some cases [43].

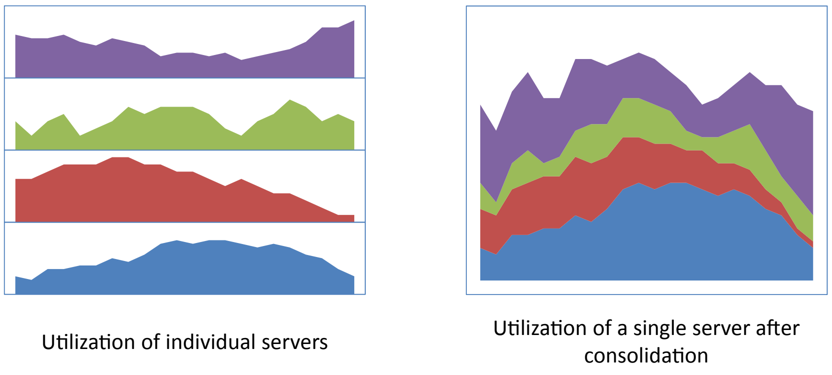


Figure 2.1: Server Consolidation Through Virtualizaton

Load balancing

Since the VM is completely controlled and encapsulated by the hypervisor, migration is made possible and is relatively easy to do, hence enabling load balancing among multiple virtualized servers [66].

Legacy applications

If a company in need to update its servers chooses to do so and migrate to different operating systems, it is possible to continue running legacy applications on the old OS within a VM, which reduces migration costs. One example of the support of this approach is the “XP Mode” supported by “Microsoft Windows 7”.

Versatility

The ability of taking snapshots of running VMs, the easiness of migration, shifting of assigned resources, priority allocations, deployment of new VMs and the existence of Virtual Appliances are a big plus when considering virtualization.

2.2.2 Disadvantages

Even though the advantages are plenty, there are also some disadvantages, which will be considered next.

Safety

According to a safety specialist from Gartner [38], nowadays, VMs are actually less safe than physical machines due to the Virtual Machine Monitor (VMM) [12].

This is an interesting point of view since, if the host OS is compromised, the entire guest VMs will also become compromised. Because the VMM is a software layer, there may be vulnerabilities.

Management

Virtual environments need to be created, monitored, maintained and configured. Although there are a few products that aim to integrate those functionalities, they are usually not optimal and are hard to use [12].

Performance

A few questions remain to be answered on the subject, the first one relates to assessing the performance penalty introduced by the VMM. In order to establish a systematic, standard performance benchmark, the SPEC Virtualization Committee was created [59] but no results have been made available as of this date.

Some independent results conducted by VMware comparing ESX Server with Xen (that were promptly questioned and re-run by Xen-Source [12]) show that the performance impact is not significant.

2.2.3 Virtual Machine Monitor

A virtualized system includes a new layer of software, called the VMM or Hypervisor. It uses a thin layer of software or firmware to achieve fine-grained, dynamic resource sharing. The main role of the VMM is to arbitrate access to the underlying physical host's platform resources so that these resources can be effectively shared among multiple "guest" OSs [46].

In 1974 Popek and Goldeberg said that "For any computer a virtual machine monitor may be constructed if the set of sensitive instructions for that computer is a subset of the set of privileged instructions" [50], where sensitive instructions refers to instructions which, in a virtualization context, may interfere in the execution of other OS who share the same hardware resources, therefore compromising the isolation between guest OSs.

They also defined three main characteristics believed to be essential to the architecture of virtualizable machines [50]:

- *Any program run under the VMM should exhibit an effect identical with that demonstrated if the program had been run on the original machine directly.* The only notable exception to this rule was timing. The software (or hardware) assisting the virtual machine must sometimes intervene to manage the resources used by it, thus altering the timing characteristics of the running virtual machine.

- *A statistically dominant subset of the virtual processor's instructions is executed directly by the real processor.* This means that a virtual machine is not the same as an emulator. An emulator analyses and intervenes on every instruction performed by the real processor, whereas a virtual machine occasionally relinquishes the real processor to the virtual processor.

- *The VMM is in complete control of system resources.* This means that the running virtual machine does not have direct access to the underlying hardware, and every resource must go through the VMM.

There two main types of Hypervisors:

- Hardware based Hypervisors (Type 1 or native VM, figure 2.2.3) are run directly on the computer hardware and monitor the guest OSs which in turn runs one level above the Hypervisor (Ex: Xen, VMware ESX Server).

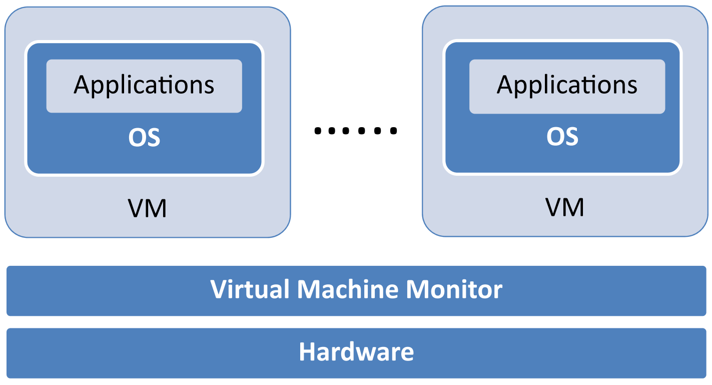


Figure 2.2: Type 1 Hypervisor Architecture

- Software based Hypervisors (Type 2 or hosted VM, figure 2.2.3) run within a conventional host operating system on the second software level (the first level is dedicated to the host's OS), thus the VM runs on the third software level above the hardware (Ex: Microsoft Virtual Server, VMware Server, Java VM).

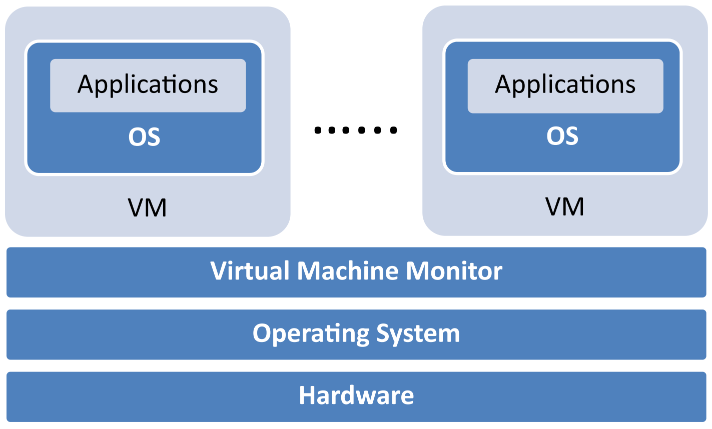


Figure 2.3: Type 2 Hypervisor Architecture

2.2.4 IA-32 Virtualization

IA-32 (x86) is the dominant Instruction Set Architecture (ISA) nowadays. It is widely used both in personal computers and in high-end, highly-reliable server applications; hence, virtualizing IA-32 would have tremendous benefits and applications. Nevertheless, unlike mainframes, x86 computers were not designed to support full virtualization; they were designed to run directly on the bare-metal hardware, so they assume they fully own the computer hardware.

The x86 ISA defines four processor operation modes, named *rings* identified from 0 to 3. In the most commonly used x86 OSs (Microsoft Windows and UNIXes) only two modes are used: Ring 0 which detains the higher privileges and is used by the OS (kernel mode), and ring 3, with lower privileges which is used by user processes (user mode).

This design decision of not taking into account virtualization is most apparent in a small set of essential x86 instructions (17 in total) [53] that are not required to be run on privileged mode. These sensitive instructions behave differently in kernel mode and in user mode, so if kernel mode code is run in user mode, some instructions may not throw exceptions but instead return incorrect (compared to the kernel mode) results. Therefore the VMM has to scan all user mode code and replace these sensitive instructions with explicit calls to the VMM.

To overcome these obstacles, a procedure was outlined by Robin and Irvine [53]:

- *Non-sensitive, non-privileged instructions*: These may be run directly on the processor; the instructions are known to be safe;
- *Sensitive, privileged instructions*: Trap. Since the virtual machine is run in user mode, when it attempts to use an instruction that is privileged, the CPU issues an interruption. The VMM traps this interrupt and performs whatever steps are necessary to emulate the instruction for the virtual machine;
- *Sensitive, non-privileged instructions*: Any of these 17 instructions of the IA-32 architecture cannot be trapped, therefore, the VMM must monitor the running VM to make sure that it does not execute these instructions.

Since the virtualization of the IA-32 architecture has so many obstacles, the IA-32 does not meet the criteria specified previously by Popek and Goldberg [50] and some clever hacking had to be done.

So, in practice, privileged instructions must be binary translated (an instruction set is emulated by another through code translation) to run safely on the processor. Another aspect of the IA-32 that makes it a difficult platform to virtualize is its "open-nature", i.e. there is a great amount of diverse and different devices and device-drivers available which do not make the virtualization effort easy.

The first commercial software supporting virtualization on the x86 platform was released on February 28th of 1999 by VMware and attained reasonable performance. The user-level code was run directly in the hardware, to attain the maximum performance, and privilege code was run using binary translation.

2.2.5 Virtualization Techniques

As clarified below, there are three main solutions to virtualize the x86 platform [68]:

- Full Virtualization using binary translation;
- OS assisted virtualization or paravirtualization;
- Hardware assisted virtualization.

Full virtualization

Implementing a full virtualization architecture, as illustrated in figure 2.4, raises many challenges: the first problem is the number of different devices that must be supported by the VMM, which is very high. To solve this problem, the implementations of full virtualization usually use generic devices (Ex: VMware) that work properly for most devices but that do not guarantee their best performance. One other inconvenient is the fact that guest operating system does not know that is being executed over the VMM and so, every instruction must be tested by the VMM prior to being executed on the hardware.

The last issue is bound to memory virtualization and virtual memory management which is extremely difficult on the IA-32 architecture. If a given guest OS uses virtual memory, when an application makes a request for a page of memory, the OS translates the memory address from the applications "virtual" space into the system's real space using a page table. Unused page tables may be written to disk when they become inactive or another application requires memory. All these procedures are typically performed using special CPU instructions for memory management. In a virtualized environment, the VMM must intercept all virtual memory calls to the CPU and translate "virtual machine" space into the system's real space using another page table, get the memory (which may be on memory or disk) and then return it to the virtual machine. While this may not look too harsh on the performance side, one must keep in mind that before the VMM received the memory access call, a page table lookup on the VMM's page table had already been performed to see where the memory was located, thus requiring in total at least two context switches between the VM and the VMM, which is very expensive for a single, simple, memory access.

Although there are a lot of problems with full virtualization (performance related, mostly) there are also some advantages. Full virtualization provides the best isolation and security for virtual machines and simplifies migration and portability, as the same OS may run virtualized or natively. Examples include VMWare Workstation, User Mode Linux, Microsoft's Virtual PC and Xen (from V3.0).

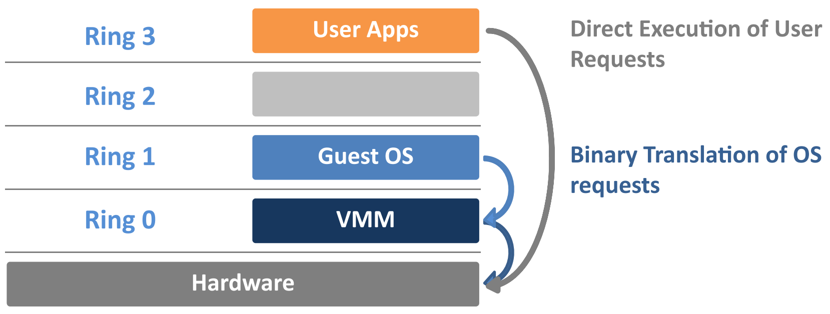

Ring 3
User Apps
Direct Execution of User Requests
Ring 2
Ring 1
Guest OS
Binary Translation of OS requests
Ring 0
VMM
Hardware

Figure 2.4: Full Virtualization on the IA-32 architecture

Paravirtualization

Paravirtualization (figure 2.5) attempts to mitigate the problems existing in the x86 architecture that usually cause the VMM to intervene too often to perform protected tasks. Instead of going directly to the CPU to perform the task, the OS is modified to call the VMM [1] and let it handle the protected task. As such, there is no need for the VMM to constantly monitor the guest VM and the performance of this technique has proven to be superior to the one of full-virtualization (without hardware assistance). The main drawback of this approach is the need to modify guest OSs, and so, its portability is poor.

Xen [9] and Denali [70] are examples of the implementation of this technique.

[1]The commonly used terminology is *hypercall*.

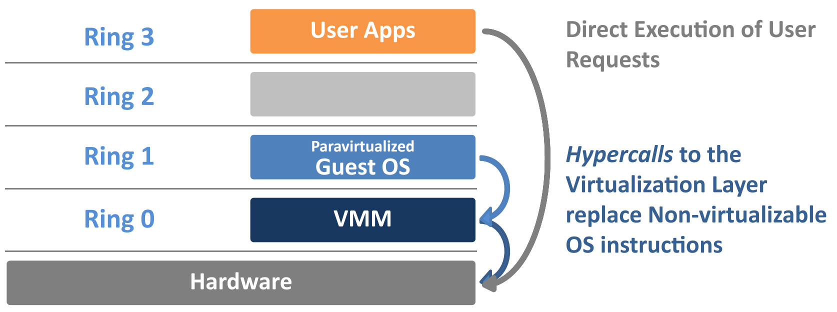

Ring 3
Ring 2
Ring 1
Ring 0
User Apps
Paravirtualized Guest OS
VMM
Hardware
Direct Execution of User Requests
Hypercalls to the Virtualization Layer replace Non-virtualizable OS instructions

Figure 2.5: Paravirtualization on the IA-32 architecture

Hardware Assisted Virtualization

In order to address the difficulties experienced in x86 virtualization, that many developers could not handle, most modern consumer x86 processors include some form of hardware virtualization support, usually either *Intel-VTx* or *AMD-V*. Even though AMD-V and Intel-VTx were developed differently and are incompatible, they serve the same purpose.

This virtualization support came in the form of processor extensions, and the first generation mainly addressed the issue of privileged instructions, offered no support for Memory Management Unit (MMU) virtualization, and exhibited a performance equal (and sometimes worst [4]) than that of binary translation techniques. These extensions allow the hypervisor to run below ring 0, in the so called *root* mode (figure 2.6). Nevertheless, it made the implementation of virtualization software simpler, by allowing the use of classic *trap-and-emulate* techniques.

The next step in hardware assisted virtualization was to develop a way to virtualize Input / Output (I/O) and devices. The first AMD specification of Input/Output Memory Management Unit (IOMMU), which provides a way of virtualizing I/O traffic and performing I/O communication translation at the hardware level (as opposed to software level), was released in 2006 [6] and is currently implemented in some workstation platforms using Opteron processors with four or more cores and specific chipsets. In the consumer side, there is currently only one chipset supporting IOMMU:, AMD 890FX. AMD's IOMMU was recently renamed to a more commercial name: *AMD-Vi*. Intel's solution to I/O is dubbed "Intel-VTd" and is similar to AMD's implementation [3].

The final step in assisting virtualization was to provide proper means of efficiently accessing the system memory. To this end a second level address translation was needed and AMD created the Nested Page Tables (NPT) [5] or Rapid Virtualization Indexing (RVI) while Intel created the Extended Page Tables (EPT), used in current processors based on the *Nehalem* architecture [28].

The combination hardware assisted processor virtualization, I/O and devices virtualization and memory access virtualization provides the necessary framework for x86 architecture to be efficiently virtualized. Not all current systems support these three techniques, but some already do (mostly workstations) and it is likely that in the future all consumer, commodity, computers will support them.

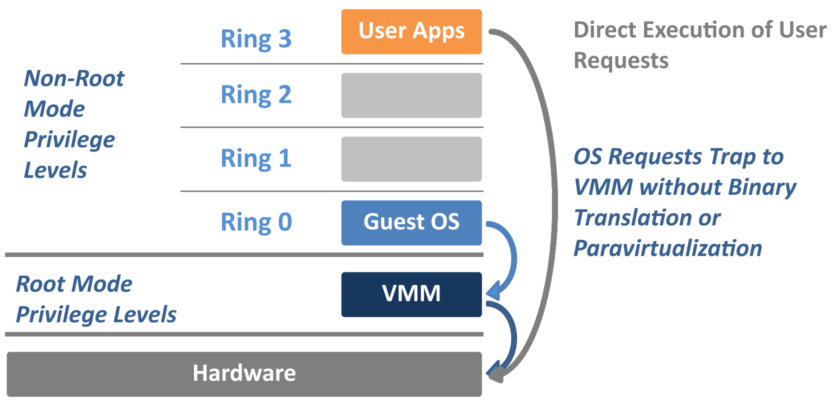

Ring 3
User Apps
Direct Execution of User Requests
Non-Root Mode Privilege Levels
Ring 2
Ring 1
OS Requests Trap to VMM without Binary Translation or Paravirtualization
Ring 0
Guest OS
Root Mode Privilege Levels
VMM
Hardware

Figure 2.6: Hardware-Assisted Virtualization on the IA-32 architecture

2.2.6 Virtual Appliances

Virtual appliances (figure 2.7) are pre-built, pre-installed, and usually pre-configured at some degree. They are software solutions VMs that are packaged, updated, maintained and managed as a unit, that allow the deployment and management of pre-integrated solution stack.

If, for example, one needs a network performance monitor, solutions exist (Nagios [44], Cacti [11]...) that allows their deployment without needing to install software on the running systems; it is simply required to load the appliance and configure it. This approach saves installation time, potential hassles with initial setup of the software, and isolation from the other running services.

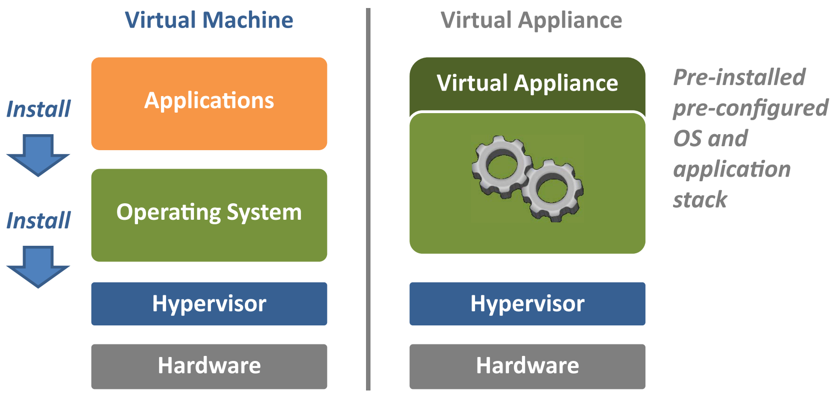


Figure 2.7: Virtual Machine vs. Virtual Appliance

2.2.7 Analysis of Server Virtualization Tools

Several tools exist nowadays; OpenVZ, Xen, VMware, Denali, Microsoft's Virtual PC, Sun xVM and Oracle VM are some of the most know tools. In the following pages, only Xen and VMware will be discussed, since they can both be considered a reference in the way they implement virtualization.

Xen

Xen is one of the most popular paravirtualization tools and was developed by the University of Cambridge [9]. It has the main goal of paravirtualizing commodity operating systems and aims for 100% binary compatibility for applications running in its virtual machines.

One of the main advantages of using Xen is due to the fact that it performs better than other full virtualization alternatives (with no hardware assistance). Although the guest operating system has to be ported to Xen, this is not a real disadvantage nowadays, since the most common OSs already provide versions supporting Xen, such as Windows XP, FreeBSD, NetBSD and the most popular Linux/Unix distributions (Fedora, Ubuntu, Debian, Open Solaris,...).

In order to understand how Xen supports paravirtualization, one must understand two fundamental concepts: the *domain* and the *hypervisor*. The hypervisor, or VMM, was already discussed. The domains are Xen's virtual machines. There are two types of domains: the *domain0* (or dom0), which is privileged, and the *domainU*s (or domUs), which are not privileged. When the host computer is started, a domain 0, privileged, virtual machine is created. This domain accesses a control interface and executes management applications. The domUs can only be created, started, shutdown or modified from within the dom0. In the dom0, a special virtual machine is run, Linux with a modified kernel, that has access to the resources of the underlying physical machine and is allowed to communicate with the other domU virtual machines.

Domain 0 has the drivers to the underlying hardware, while domUs have virtual drivers that must go through the domain 0 in order to access the physical resources.

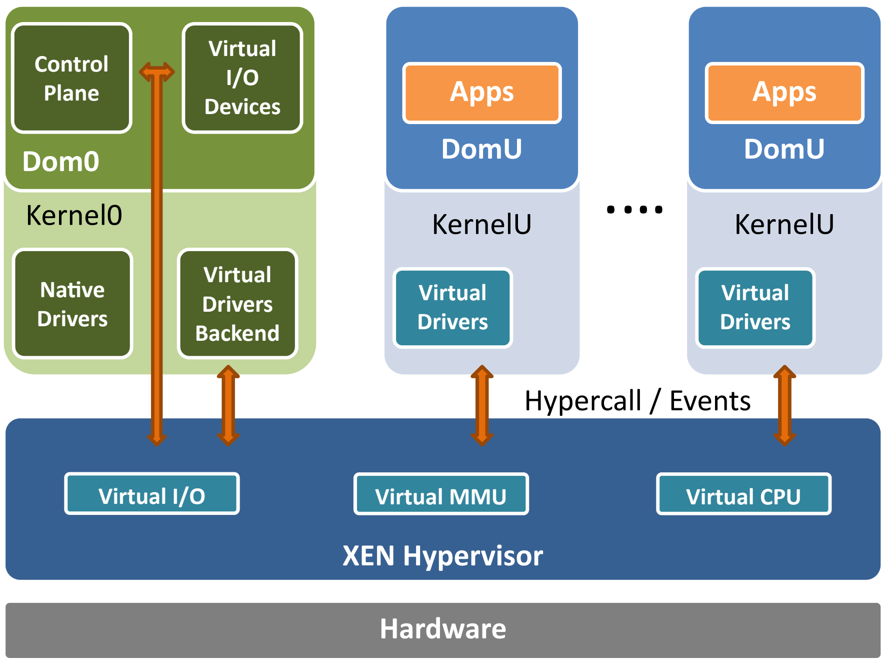


Figure 2.8: Xen's Architecture

The memory virtualization works by using a memory pool. Each VM is assigned a given amount of memory which may be dynamically changed without needing to stop or reboot the VM. Each VM may have one or more virtual interfaces. The communication between the guest OSs and Xen is performed using asynchronous I/O rings. A global overview on Xen's architecture is displayed on figure 2.8.

If the computer hardware where Xen is running supports virtualization, the latest versions of Xen, (3.0 and superior) allow for one other domain type: the Hardware Virtual Machine (HVM). In this domain, full virtualization is performed, which allows unmodified OSs to run over Xen's hypervisor. Although the main purpose of Xen was actually to avoid full-virtualization on the IA-32, with the advent of hardware assisted virtualization, much of the performance benefits were lost, especially in systems that support processor virtualization, I/O virtualization, and also memory virtualization. It currently supports up to 64 Symmetric Multiprocessing (SMP) machines, and up to 64GB of RAM.

In Xen, multiple VMs may communicate with each other using virtual networks that do not require any physical interface. It is therefore possible to setup full networks, with multiple Virtual Computers (VCs) in a single physical computer, making it useful for testing client-server environments, for example.

VMware

VMware is one of the most popular virtualization tools for the x86 platform. There are tools for all kinds of systems, from personal computers to datacenters. There are four main categories for the available products: Management and automation, virtual infra-structure, cloud computing and virtualization platforms.

The most prevalently used ones (VMware Player, Workstation, Server and Fusion – for MacOS) all rely on the *Hosted Virtual Machine Architecture*. They install like a regular application on a host OS. When the software is run, the application portion, *VMApp*, uses a previously loaded driver, *VMDriver*, to establish the VMM component that will run directly on the hardware and control the guest VMs.

In this scheme, VMware does not need to provide drivers to every single device existing for the IA-32 architecture; instead, it relies on the drivers of the host OS. If a guest VM performs an I/O operation, the VMM will intercept it and perform it in the host OS on its behalf, using the *VMDriver*; therefore it is avoided the need to interact directly with the devices.

This approach may introduce a lot of performance penalty for I/O intensive tasks, but for CPU intensive tasks, the performance is similar to that of a physical system. Each VM is exposed to a set of generic devices, such as a PS/2 keyboard and mouse, floppy and CD-ROM drives, an Integrated Drive Electronics (IDE) controller, a Soundblaster audio card, serial and parallel ports, a standard graphics display card, USB ports, and any number of AMD PCNet Ethernet adapters. This standardization helps with portability across platforms as all VMs are configured to run on the same virtual hardware, regardless of the physical hardware on the system.

The lastest version of Vmware Workstation (v7) allows the use of 4 processors at most, up to 32GB per VM and supports over 200 OSs.

One feature that adds versatility is memory over-committing (assigning VMs more memory than physically available). Since it is not very likely that all VMs will be needing their full assigned memory at all times, memory over commiting allows a more effective (dynamic) sharing of existing physical resources.

Just like Xen, VMware Workstation also provides utilities for setting up virtual networks.

2.2.8 Libvirt: Virtualization API

Basic Architecture

Libvirt is a virtualization library developed by Red Hat that strives to provide a common API for multiple virtualization environments and hypervisors, such as Xen, KVM, QEMU and VirtualBox (figure 2.9). It provides a hypervisor-agnostic API that allows the building of customized tools to manage guest operating systems running on a host Linux node. Although originally designed as a management API for Xen, it was extended to work with other hypervisors that despite being implemented differently provided common functionalities.

The actual implementation was done in C, but bindings exist for other languages, for example: Python, Perl, Ruby, Java and OCaml.

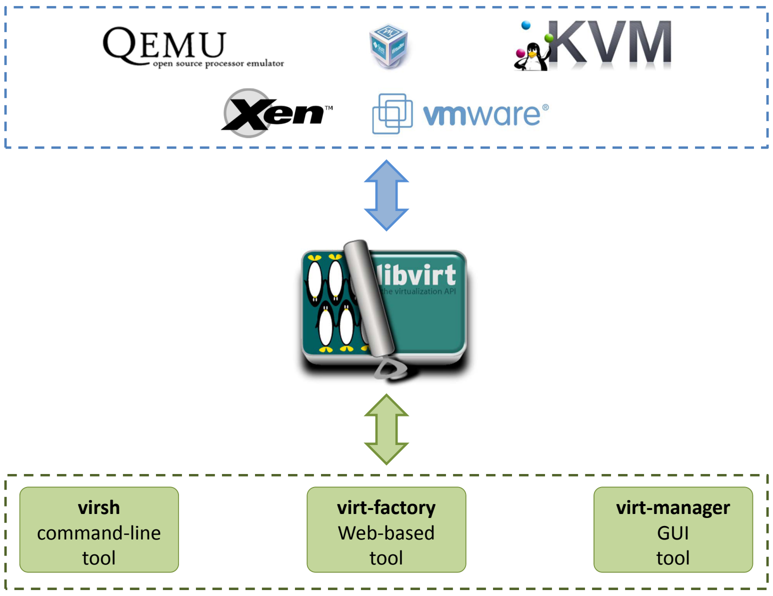


Figure 2.9: Libvirt: Virtualization API

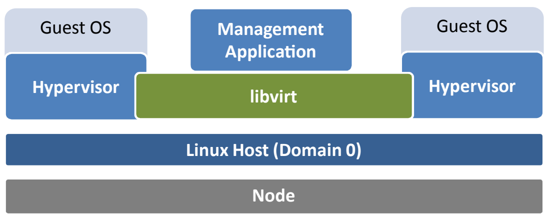
Control Methods

In libvirt, two distinct important concepts exist, the *node* refers to the physical host and the *domain* refers to the guest operating system.

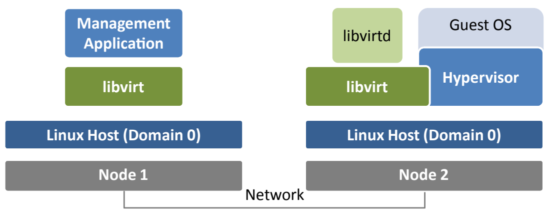
There are two distinct control methods; in the first one, demonstrated in figure 2.10(a), the management application and domains exist on the same node, so, the management application uses libvirt to control the local domains. On the second possible model, figure 2.10(b), remote communication is required and is performed using *libvirtd*, the libvirt daemon running on remote nodes that is installed after the installation of libvirt and that automatically sets-up the proper drivers for the node's hypervisor.

Hypervisor Support Mechanisms

Multiple-hypervisor support is possible due to a driver-based architecture that allows a common API to service different hypervisors in a similar way. Using this architecture, some



(a) Libvirt - Local control.



(b) Libvirt - Remote control.

Figure 2.10: Libvirt control methods.

hypervisor-specific functions may not be available through libvirt. If a given hypervisor does not support some common features, they are marked as unsupported.

Virtualization Shell

Virsh, short for virtualization shell, is an application built on top of libvirt that allows the use of a command line interface to perform most of the libvirt's functionality without needing to actually write a program using libvirt.

API Overview

The provided API can be divided into four main categories:

- The *hypervisor connection API*, which is responsible for establishing and maintaining the connection to the hypervisor, that can be either local or remote;
- The *domain API* deals with domain management by performing creation tasks, status monitoring and general configuration;
- The *network API* is in charge of the virtual network components' management, such as bridge binding, interface attachment and configuration;
- The *storage volume and pool API* allows the association and management of storage components, such as Logical Volume Management (LVM) partitions and disk images;

Summary

From the libvirt's analysis, one can see that this is an invaluable tool for anyone developing even the simplest virtual machine management application: the flexibility and level of control provided are enormous and greatly simplify the hard task of managing VMs. By providing a common interface for multiple hypervisors, the portability issues and effort duplication are greatly reduce; hence, it promotes a faster and better application development.

2.2.9 Summary

Computer virtualization evolved a lot in the past few years, in just a matter of ten years consumer-level virtualization has become not only possible but also incredibly advanced. The existing solutions, especially the ones relying on hardware-assisted virtualization can reach performance levels similar to native systems. Hardware assisted virtualization is becoming a reality and both AMD and Intel offer fully virtualizable x86 platforms in the workstation and server markets.

The concept of virtual appliances will allow applications and functionalities to be added on-the-fly with short deployment and configuration times and without affecting the running services.

Further investigation and development will need to take place in order to address potential security issues and develop management tools that can lower the complexity of managing multiple virtual machines.

All in all, computer virtualization is an enabler of new and revolutionary technologies that will allow experimental, fearless, testing on production environments and also smoother upgrade paths.

2.3 Network Virtualization

A Virtual Network (VN) may be defined as a group of virtual resources (e.g. virtual routers/switches and virtual machines) interconnected via dedicated virtual links (e.g. Virtual Local Area Network (VLAN)), agnostic of the underlying hardware, that allows the coexistence of multiple virtual networks on the same physical substrate.

Just like the previously discussed server virtualization, Network Virtualization (NV) also fosters the consolidation of resources; in this case the resource is the existing physical network. It aims not only to reduce the Total Cost of Ownership (TCO) (Capital Expenditure (CAPEX) plus Operational Expenditure (OPEX)) of operators, but also to provide them with the flexibility of running network protocols independently of the physical infrastructure. This protocol independency will allow them to better adjust each VN to the desired applications and services. Instead of considering only the current two main services (Data and Voice), many other services could easily be created with different speed, safety, and timing requirements.

As it stands, NV bears the burden of being the tool to revolutionize the Internet [8]. It is seen as the only possible escape route from the current ossified Internet model where dramatic architectural changes are not possible, but required.

When deployed, NV, will allow the Internet as it is today to seamlessly run on top of it, just like any other network, and will allow experimenting and evaluating new technologies and architectures with complete isolation and without disrupting, or risking the disruption, of the currently running services.

In the following sections, the goals, current research state of NV, underlying technologies and existing initiatives will be analysed and evaluated. Due to the fact that NV aims to be "the choice" for future networking, it presents many technical and conceptual challenges which will be discussed next.

2.3.1 Design Goals

Network Virtualization, as described in the previous paragraphs, presents several goals that can be further subdivided in small objectives to be reached. These objectives should also work as guidelines when developing an algorithm or protocol for VNs, and should also serve the purpose of providing a means of comparison for different NV architectures.

Flexibility, Programmability and Heterogeneity

A virtual networking environment should be flexible on every aspect. The flexibility should come in the form of programmability, i.e. being able to program the functionality of every network element and protocol, being able to use whatever topology is desired and must be able to virtualize every type of underlying network technology, regardless if it is optical, wireless or copper.

Scalability and Manageability

One of the prime reasons to employ network virtualization, is to allow the coexistence of multiple networks on the same substrate; therefore, scalability is one fundamental part of network virtualization. An infrastructure provider should allow as many VNs as the underlying infrastructure can accommodate, without affecting the VNs' performance. The management tasks should be made modular and introduce accountability at every layer of networking. This modularity should simplify network management since it is simpler to manage multiple virtual networks running in parallel than a single more complex network [22].

Isolation and Security

Isolation must be guaranteed in order to prevent that a misbehaving virtual network affects other unrelated networks, whether due to misconfiguration, high resource usage or security breaches.

Legacy Support

Backwards compatibility is fundamental if the envisioned network virtualization environment is to become true. There must be a seamless upgrade path from current network technologies and protocols to the future ones, and NV must be able to replace the current Internet without breaking it.

2.3.2 Proposed Business Models

Due to the extra degree of freedom, i.e. the possibility of having multiple networks on a single substrate network, and also the fact that the services and their underlying protocols are no longer bound to the infrastructure, decoupling the traditional role of the ISP in two or more seems logical. This decoupling is being proposed with different models.

Some authors ([22]) suggest the decoupling in two (figure 2.11):

- *Infrastructure Provider (InP):*

  InPs are responsible for deploying and managing the physical infrastructure and offer their resources to different service providers. Their main differentiation factor may be the quality of the resources, geographic location, tools and freedom delegated to their customers.

- *Service Provider (SP):*

SPs aggregate resources from one or more InP, to build virtual networks and offer end-to-end services. They can also provide network services, or even a subset of their network, to other SPs.

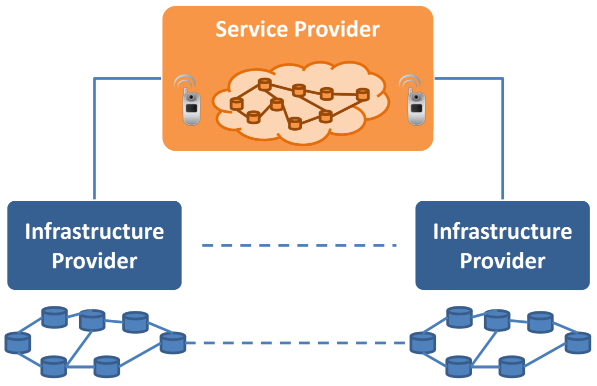


Figure 2.11: SP and InP business model

While others propose a more complex model ([58] – figure 2.12):

- *Infrastructure Provider (InP):*

  Just like in the previous model, its role is to provide and manage the physical infrastructure;

- *Virtual Network Provider (VNP):*

  The VNP main function is to assemble virtual resources from one or more InP, in order to build a virtual topology;

- *Virtual Network Operator (VNO):*

  The VNO should be responsible for the installation and operation of the VNs provided by the VNP, according to the needs of the SP;

- *Service Provider (SP):*

  In this business model, the SP is solely responsible for using the VN and providing some services (application or network services).

2.3.3 Existing Technologies

This section describes some of the existing technologies that emulated network virtualization, to allow the coexistence of logically isolated networks. Focus will be given on four significant technologies: VLANs, MPLS, VPNs and Overlay Networks (ONs).

VLANs

VLANs provide a means of separating broadcast domains into smaller ones, allowing the creation of functional groups. It is a logically separated IP subnetwork and allows multiple IP networks and subnets to exist on the same switched network. Since they are logical entities

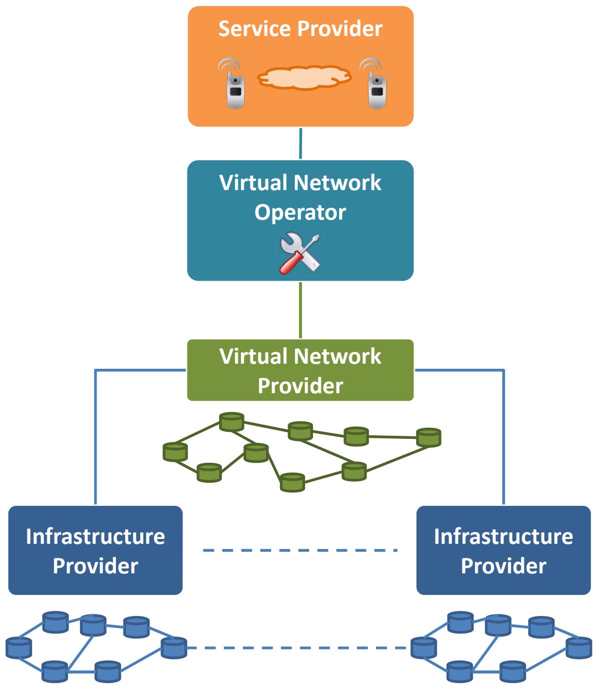


Figure 2.12: SP, VNP, VNO and InP business model

(each VLAN has an ID, from 1 to 4094) configured in software, they are very flexible in terms of management and reconfiguration, provide increased security (groups with sensitive data may be separated from the rest of the network), higher performance (due to broadcasting domain separation) and are cost effective (VLANs can be used in Linux by simply installing the respective module).

The use of VLANs can also provide increased QoS, since logical networks for different services may be created. A VLAN with higher priority may be created for VoIP and another one with less priority could be created for data. They are essentially Layer 2 constructs, even though implementations in different layers do exist. To ensure that VLAN members or groups are properly identified and handled, frame coloring (tagging) is used. With frame coloring, packets are given the proper VLAN ID at their origin so that they may be properly processed as they pass through the network. The VLAN ID is then used to enable switching and routing engines to make the appropriate decisions as defined in the VLAN configuration. This tagging allows the multiplexing of frames from different VLANs into trunks, i.e. multiple VLANs may use a single physical link, while remaining logically separated at layer 2. The standard that defines VLAN tagging is IEEE 802.1q.

Although useful for companies' Local Area Networks (LANs), 802.1q suffers from some problems: considering a costumer with a provided private link interconnecting multiple sites, if VLAN tagging is used, the provider cannot use VLANs again since that would modify the costumer's packets. In order to address this issue, 802.1ad, also called QinQ or Q-tunneling, was standardized in 2005. This new protocol allows the addition of an extra tag to the already tagged packet, therefore enabling costumer tagged packets through the operators' networks. 802.1ad was replaced with an improved version, 802.1ah, in 2008.

MPLS

MPLS [56] was created with the goal of increasing the switching speed of core networks, where routers are interconnected using high speed links (in excess of 40Gbps most of the times) and rely on layer 3 forwarding.

In order to improve forwarding speeds, MPLS's approach uses label switching. The use of labels introduces connection oriented mechanisms inside the connectionless IP networks, since the packets are forwarded solely based on these labels.

When a packet arrives at an edge router, a Label Edge Router (LER), MPLS determines its Forwarding Equivalence Class (FEC). All packets belonging to a particular FEC will go through the same path, or a set of paths in case of multi-path routing.

The FEC is encoded in a short fixed length value (label) and when a packet is forwarded to the next hop, the label is sent with it; hence, in subsequent hops there is no need to analyze the packets' network layer header. The packet's label will be used as an index into a table that specifies the next hop and corresponding new label, which replaces the previous one. This label swapping procedure is done by the Label Switching Routers (LSRs) in hardware, at line speeds. In figure 2.13 an illustrative example of a packet going through an MPLS enabled network is shown.

When an ingress router reads a packet's network layer information, QoS data may be extracted and a particular FEC is chosen according to it; thus, MPLS can be used to differentiate services. MPLS is widely used in core networks with major applications in traffic engineering and VPNs.

The VPN capability is of high importance; using MPLS, the traffic of a given enterprise is transferred transparently through the Internet with performance and safety guarantees. VPNs may be implemented using label stacking, i.e. a frame that already belongs to a Label-Switched Path (LSP) may travel through another LSP, thus multiple LSPs may be aggregated into a single LSP, similarly to Asynchronous Transfer Mode (ATM). MPLS supports unlimited stacking.

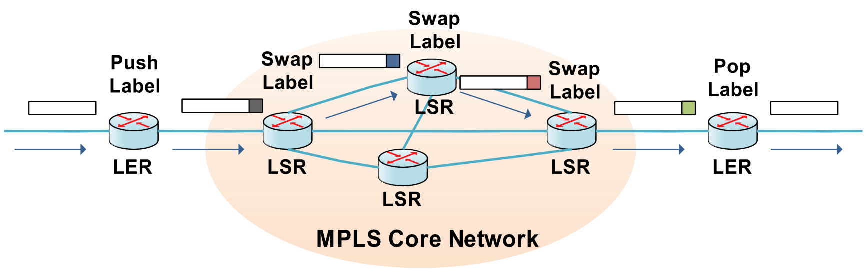


Figure 2.13: MPLS Packet Labelling and Label Swapping

VPNs

A VPN allows the creation of private networks over the public Internet infrastructure while maintaining confidentiality and security. It is usually used by corporations who wish to have their branches connected, or to allow employees to access remotely to the corporations' networks. To remain private, the traffic is encrypted. Instead of using a dedicated Layer 2 connection, such as a leased line, a VPN uses virtual connections that are routed through the Internet.

Two main types of VPNs can be considered, taking into account whether the operator knows about them and plays a role in the VPN establishment or not.

- *Costumer Edge (CE) based approach :*

  The provider network is unaware of the existence of the VPN (figure 2.14(a)). The CE devices are responsible for creating and destroying the tunnels between themselves.

  These are usually Layer 3 VPNs (L3VPNs), since layer 3 protocols are used to carry data between the CEs. There are also some VPN solutions that work on upper layers of the Open Systems Interconnection (OSI) model (transport, session or application).

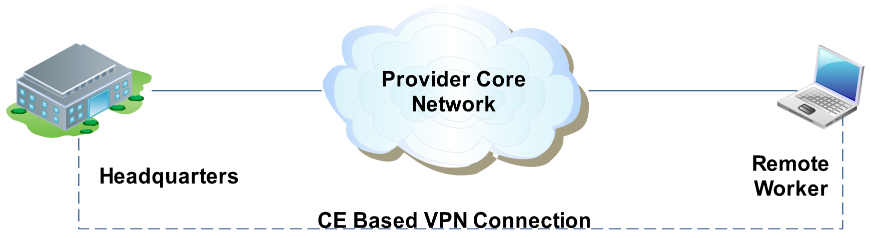
- *Provider Edge (PE) based approach:*

  The provider network is responsible for VPN configuration and management. A connected CE may behave as if it were connected to a private network (figure 2.14(b)).

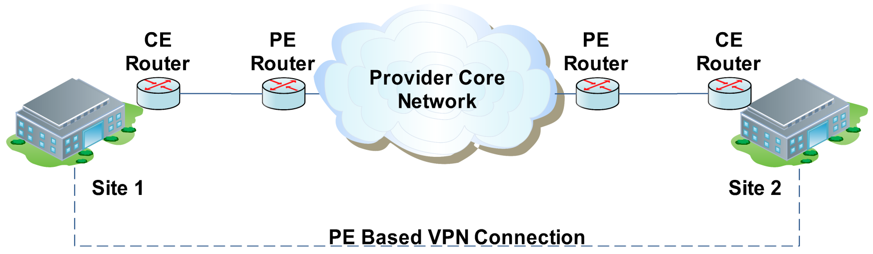
  In most cases, these provider VPNs are based on layer 3 protocols, but can also be based on layer 1 or 2.

  Layer 2 VPNs transport frames between connected sites. They are independent of the upper layer protocols and are more flexible than layer 3 VPNs.

  Layer 1 VPNs emerged from the need to extend layer 2 / layer 3 packet-switching concepts to advanced circuit-switching domains (such as optical switching and GMPLS [40]). This approach to VPNs allows multi-service backbone where customers can offer their own services with payloads of any layer (e.g. ATM, Time Division Multiplexing (TDM), IP,...) and provides complete isolation from other VPNs.



(a) Consumer Edge based VPN



(b) Provider Edge based VPN

Figure 2.14: Consumer vs. Provider Edge based VPN

Overlay Networks

Overlay Networks such as the Content Addressable Network (CAN) [52], Chord [60], Pastry [57] and Viceroy [39] create a virtual topology on top of the physical topology with the purpose of implementing a network service that is not available in the existing network. They are not geographically restricted, are flexible and adaptable to changes. The most flagrant example is the Internet which was built on top of existing phone network, as an overlay, that ended up shaping it. Multiple overlay designs have been proposed to address several issues such as providing QoS guarantees [62], enabling multicasting [29], file sharing [37] and protection from DoS [32].

Through the use of open-platform solutions such as PlanetLab [48], the test of overlay networks does not imply large expenditures or complexity. PlanetLab aims to provide both a research testbed and a deployment platform for new service oriented network architectures, and hopes that the weight of the developed overlay networks will end up changing the current Internet architecture. Currently, there are 1086 nodes and 506 sites worldwide [49].

As it stands, overlay networks are seen as a way to deploy small fixes to the "broken Internet" and rely heavily on the IP application layer. Despite having the potential to test novel architectures (using virtual testbeds for example [8]), their current design and implementation is not capable of supporting radically different architectures [13].

2.3.4 Existing Initiatives

AKARI

AKARI, a Japanese project that started in 2005, presents a clean-slate approach to design a future Internet that shall be ready by 2015, and is expected to support human development for 50 to 100 years.

Given the current trends in bandwidth usage, that closely follows Moore's Law, it is expected that in 2015 10Gbps fiber connections will be common for home users (Fiber To The Home (FTTH)). Thus, it is reasonable to assume that a future Internet will be largely based on optical (with optical packet switching and optical paths) and wireless (using Packet Division Multiple Access (PDMA) in combination with other multiplexing techniques) technologies.

Based on these premises, AKARI plans to design and prototype scalable networks with self-* properties (self-healing, self-organizing, self-configuration, self-routing,...) and with autonomic and distributed control.

According to the AKARI roadmap, the New Generation Network (NWGN) design should be completed this year.

GENI

Based on the experience acquired from using PlanetLab, Global Environment for Network Innovations (GENI) [23] is a United States (US)' long term virtual laboratory initiative that focuses on providing realistic experimental facilities in order to evaluate alternative architectural structures, by deploying prototype networks and running controlled experiments.

It is a generalization of the PlanetLab approach, comprised of network resources (links, nodes...) that are virtualizable and programmable; Thus, they can be shared and partitioned between many researchers and implement radical new designs. GENI is programmable at any level of abstraction (e.g. optical, IP, application,...), where researchers may control how the nodes behave, and is able to incorporate a wide variety of network technologies such as optical, wireless, sensors and phones. The National Science Foundation (NFS) GENI is composed of a fiber backbone (with 25 Point-of-Presence (PoP)), programmable core routers, optical switches, programmable edge devices, Wi-Fi, WiMax , Congnitive Radio and Sensors subnets.

CABO

Concurrent Architectures are Better than One (CABO) [22] is a full virtualization initiative that aims to provide a separation between the physical network infrastructure and the services that run on it. This split should simplify network management, by relinquishing the responsibility for the physical devices to the infrastructure providers, allowing a service provider to run several simple virtual networks concurrently and also encourage competition between SPs and InPs, since the services are no longer tied to a given infrastructure. Its pluralistic philosophy, advocates flexible and extensible systems supporting multiple simultaneous network architectures.

Unlike other network virtualization initiatives, CABO's main intent is not to revolutionize the current internet architecture, but rather to provide a common framework that fosters better network services and more robust management operations. Concurrent networks are supported through virtualization of links and routers (composing the VNs), which in turns add versatility regarding to the protocols running and geographic location.

4WARD

The 4WARD European Project's goal is to explore new approaches that should enable a plurality and multitude of interoperable network architectures.

The approach taken toward a new Internet architecture was not merely a technical one. Business models and impact studies were conducted, regulatory issues were taken into consideration, and application scenarios were proposed in order to bridge the gap between innovative research results and socio-economic advantages. Four main business roles were identified as in 2.3.2: Infrastructure Provider (InP), Virtual Network Provider (VNP), VNO and SP. 4WARD's research was focused in virtualization, discovery, monitoring, management (In-Network management) and provisioning techniques for network resources.

2.3.5 Mechanisms for Network Virtualization Support

Although very promising, NV presents many challenges, not only technical but also business related.

The Internet is resistant to fundamental changes. Even with complete and tested NV frameworks developed, one huge issue will be to persuade the existing ISPs to deploy this framework; their current business model is well defined and huge investments have been made in the current infrastructure. One possibility would be to consider the creation of a Next Generation Service Provider (NGSP), employing NV, coexisting directly with current ISPs but with substantial business advantages and added-value regarding its competition (better resource utilization, protocol independency, tiered service quality, . . .). This would in turn attract an increasing number of users over time.

Aside from the business issues, there are several challenges that must be addressed if network virtualization is to succeed. Some of these challenges and related mechanisms will be presented next.

Router Virtualization

Two approaches currently exist relating virtual routers:

- *Hardware Virtual Routers*:

  Both Juniper and Cisco currently offer the possibility of running multiple virtual router instances on some of their routers. Juniper's approach to having more than one router in a single physical one is twofold [47]:

  - *Virtual Routers :*

    These virtual routers provide separated routing tables and are therefore able to provide layer 3 isolation. They are simplified routing instances running "under" the main routing daemon and have associated interfaces (logical or physical interfaces). Their feature set is reduced (no Border Gateway Protocol (BGP) signaling for example). Juniper's M, T and J series support this technology.

  - *Logical Routers :*

    Logical routers partition a single physical router into multiple logical devices, multiple daemons, that perform independent routing tasks, and therefore provide a stronger isolation than virtual routers (a logical router may contain several virtual routers). They can be thought of as a collection of smaller full-blown routers, with some exceptions but with more features than virtual routers, running inside a single housing. Juniper's M and T series support this technology.

  Cisco [14] also provides similar approaches in their XR-12000 and CRS-1 systems.

- *Software Virtual Routers*:

  The recent increase in the number of cores of commodity CPUs (achieving 6 cores for consumer-grade CPUs and 12 cores for workstation/server CPUs), along with advances in Random Access Memory (RAM) whose modules have become increasingly dense (4 Gigabytes (GB) modules are common nowadays) render commodity computer hardware as a strong candidate for router virtualization.

  Although an incredibly high computing power is provided by these multi-core computers, a main bottleneck exists: the main memory. It has been shown in [19] that the memory subsystem is the main limiting factor for high performance packet forwarding due to its high latency. When forwarding small (e.g. 64-byte) packets which reside in non-contiguous memory location, the forwarding rate is bottlenecked by the memory latency. By increasing the packet size, the forwarding rate can be greatly improved. In [19], by increasing the packet size from 64-byte to 1024-byte, the forwarding rate increased from 2.5Mpps to 7.1Mpps, effectively achieving the line rate on all used interfaces.

  Several router software solutions exist, such as Click [33] modular router and XORP [71]. The Virtual Router Project [20] aims to optimize such solutions (Click in particular) to virtual environments. Although promising results have already been attained (7.1Mpps forwarding rate), there is still a long way to go if virtualized computer-based routers intend to compete with hardware routers.

Switch Virtualization

Although some commercial solutions exist, switch virtualization still remains a hot topic nowadays. On the one hand, there are vendor solutions that offer high performance and interface fan-out at the cost of programmability and flexibility. On the other hand, open platforms, PC based, offer the desired programmability but not the required performance nor the needed interface fan-out. In the next paragraphs, vendor solutions offering virtualization will be discussed as well as open source PC-based solutions.

In 2008 Cisco shipped its first software virtual switch (Nexus 1000V, figure 2.15) developed in cooperation with VMware for their vSphere environments (operating inside the ESX and ESXi bare-metal hypervisors). The Cisco Nexus 1000V [15] is a distributed switch composed of two primary components, Virtual Ethernet Modules (VEMs) that run inside the hypervisor, and an external Virtual Supervisor Module (VSM), that manages the VEMs. The VEMs are configured by the VSM and perform advanced network features, such as providing QoS, Private VLANs, link aggregation, access control lists, port security and monitoring tools. It provides a common switch management model, similar to other Cisco switches. Due to its purely virtualized architecture and integration with VMware's software, this virtual switch is fully aware of all server virtualization events, and seamlessly supports server and VEM migration along with its security, statistics and network properties.

Cisco also provides hardware virtual switches, Nexus 7000V for example, presenting the physical switch as multiple logical devices.

Juniper, on the other hand, only provides hardware virtual switches (EX-Series) that work in a similar fashion as the previously discussed virtual routers (but with no routing), i.e. a given physical switch may contain several isolated switching instances.

Open source, software based solutions also exist. One such solution is provided by Crossbow [63, 64], a virtual switching and virtual interface software that exists for the OpenSolaris platform with the goal of facilitating networks in a box. It helps expanding the networking feature set of Xen, by supporting advanced features such as VLANs, bandwidth assignment per virtual network interface, link aggregation, IP multipath and QoS mechanisms.

One other solution is provided by OpenFlow [41], a software stack designed to be installed on top of physical switches, that allows defining data flows using software (software-defined networking). When a data packet arrives at a typical switch, its header is examined and proper action is taken. OpenFlow allows users to define (through a *controller*) the action that should be taken for a given packet header, regardless of the protocol being run.

The operations are "flow-based", and therefore, protocol independent. This flow-based approach allows for safe testing of new protocols on production environments; the existing protocols are assigned the proper flow, and new ones may be tested by assigning different flows. This flow separation and programmability can therefore be seen as a form of virtualization.

Considering the studied solutions, OpenFlow's approach seems to be the most promising one, since it can be installed on existing physical switches with a protocol independency on its flows. This approach should allow reduced expenditures and an significant versatility. Regarding Cisco's Nexus 1000V, it is tied to VMware and, therefore, cannot be integrated with other virtualization environments.

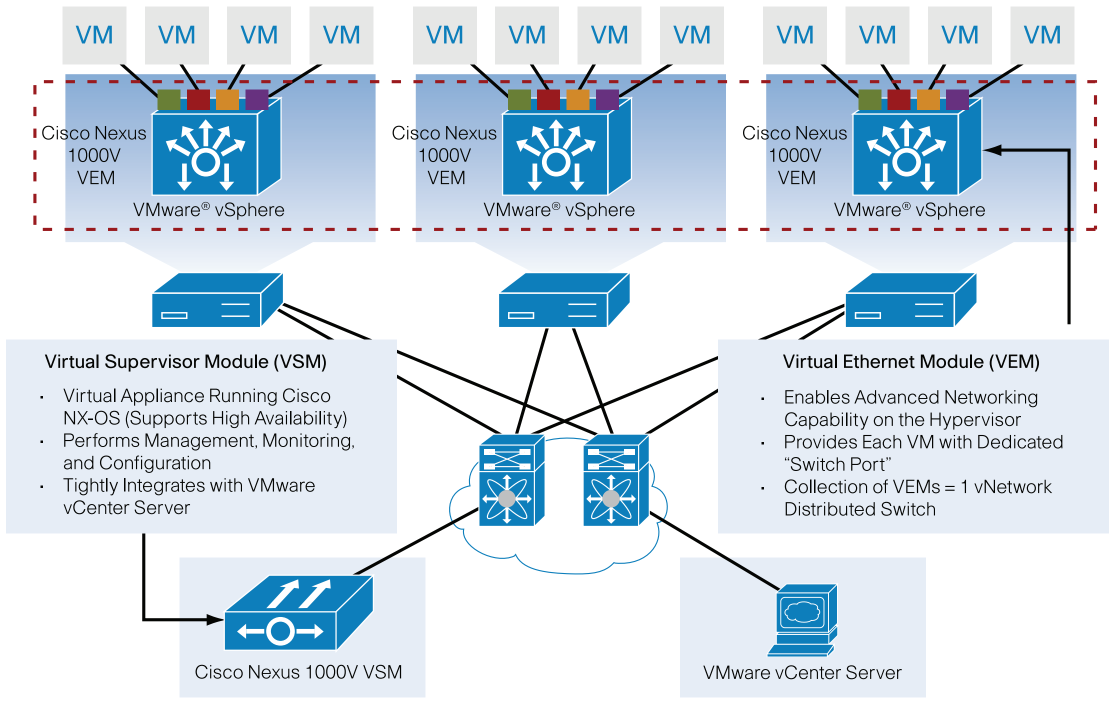

VM
Cisco Nexus 1000V VEM
VMware® vSphere
Virtual Supervisor Module (VSM)
Virtual Appliance Running Cisco NX-OS (Supports High Availability)
Performs Management, Monitoring, and Configuration
Tightly Integrates with VMware vCenter Server
Virtual Ethernet Module (VEM)
Enables Advanced Networking Capability on the Hypervisor
Provides Each VM with Dedicated "Switch Port"
Collection of VEMs = 1 vNetwork Distributed Switch
Cisco Nexus 1000V VSM
VMware vCenter Server

Figure 2.15: Cisco Nexus 1000V Architecture [15]

Hardware virtual switches provide high performance and allow consolidation but with reduced programmability.

Virtual Network Mapping

When receiving multiple VNet requests, it is of the InP best interest to optimize resource allocation in order to reduce congestion, and maximize profitability by enabling more VNets to coexist on the same substrate network. Efficient resource mapping must therefore deal with simultaneously optimizing the constrained placement of nodes and links of a given VNet in the substrate network.

This simultaneous optimization can be formulated as an unsplittable flow problem, known to be *NP-hard* [7, 72] and therefore is only tractable for a small amount of nodes and links. In order to solve this problem, several approaches have been suggested, mostly considering the *offline* version of the problem where the VNet requests are fully known in advance.

In [34], a backtracking method based on subgraph isomorphism was proposed; it considers the online version of the network mapping problem, where the VNet requests are not known in advance, and proposes a single stage approach where nodes and links are mapped simultaneously, taking constraints into consideration at each step of the mapping. Therefore, when a bad mapping decision is detected, a backtrack to the previous valid mapping decision is made, avoiding a costly remap. In order to reduce the search space of the algorithm, upper boundaries for the number of physical hops spanned by a virtual link are defined, as well as the maximum amount of mapping steps, determined by evaluation tests, before considering that the mapping failed.

Other authors, such as [36], define a set of premises about the virtual topology, i.e. the backbone nodes are star-connected and the access-nodes connect to a single backbone node. Based on these premises, an iterative algorithm is run; the backbone nodes are mapped first (arbitrarily in the first iteration), then the access nodes are connected to the closest backbone nodes, their shortest path and link capacities are calculated next and, finally, alternative backbone mappings are evaluated. The best backbone mapping is used in the next iteration. The algorithm terminates when no better solution than the previous one for backbone mapping is found, or when a pre-determined maximum number of iterations is reached.

A distributed algorithm was studied in [26]. Its aim was to reduce the number of messages required for centralized software to have an up-to-date substrate view, to enhance the robustness and scalability of the overall system, and to increase the speed of the VNet mapping, due to the parallel processing. It considers that the virtual topologies can be decomposed in hub-and-spoke clusters, and that they can be mapped in each cluster independently. Therefore, it reduces the complexity of the full virtual network mapping. The *root* substrate node, with the maximum available resources, is considered the hub of the cluster, and becomes responsible for coordinating the mapping of that cluster. Next, the set of substrate nodes able to support the spoke nodes is determined based on shortest path algorithms. In order to map the complete virtual network, the root nodes interact with each other with the intent of making a collective mapping decision.

Zhu and Ammar et al. [72] propose a heuristic and centralized algorithm for dealing with virtual network embedding. Their approach tries to solve an online version of the problem, considering reconfigurations of the existing VNs, when VN requests arrive. In order to further improve the performance of the basic mapping algorithm, a subdivision technique is also explored. The goal of the mapping algorithm is to maintain a low and balanced stress of both nodes and links of the substrate network; with that goal in mind, the algorithm starts by determining each node's stress (number of virtual nodes running on the substrate node) and the links' stress (number of virtual links whose substrate path passes through each substrate link). With these weights determined, the Neighbourhood Resource Availability (NR), that takes into account both the node stress and the local links stress, is calculated for each node. The node with the highest NR is selected as the start node to begin the candidate selection. Next, a set of substrate nodes is determined weighted by their distance to the previously selected substrate node, its node potential is calculated, and in the final step the virtual nodes are mapped. Virtual nodes with more interfaces are assigned substrate nodes with higher NR since virtual nodes with more interfaces are also more likely to setup more virtual links and increase the load on both the substrate node and neighbour links.

Although all these algorithms provide a solution for the virtual network mapping problem, most of them fail to take into consideration that not all virtual nodes are the same. The nodes may have different requirements for CPU, memory and location, and their links may not be constrained only by bandwidth, but also by latency, jitter and loss. The heterogeneity of both virtual and substrate resources is mostly not considered.

Resource and Topology Discovery

A fundamental requirement in order to be able to perform the previously discussed embedding algorithms is to know exactly the existing physical and virtual resources' characteristics, existing topologies (both physical and virtual) and the status of all network elements and links.

The discovery methods should be aware of topologies updates in order to guarantee the consistency of the topology databases and the detection of failures. They should be robust, fast to converge and efficient in gathering and disseminating network information with a reduced footprint in the substrate network.

Regarding physical topology discovery, there are multiple commercial applications that rely on the Layer 3 information to build the physical networks' topology, showing the logical connections between the resources. Hardware providers, such as Cisco [54] and Intel, have developed link layer discovery protocols that strive to provide a more detailed view over the network's elements such as hubs, switches and bridges. However, these tools are of no use when in a heterogeneous, multivendor environment. The IETF recognised this problem and designated a physical topology Management Information Base (MIB) [10], but failed to develop a protocol.

Although the discovery of the physical network topology is essential, the discovery of virtual networks' topology is also required and presents several unaddressed challenges. Because virtual networks are a relatively new concept, and no complete network virtualization tool has been developed so far, there is a general lack of scientific studies regarding virtual topology discovery, although guidelines have been provided by some authors. Some initiatives, like CABO [22] advocate the use of a separate independent discovery plane, and an implementation using distributed algorithms was suggested by [13].

The virtual networks are made of virtual resources laying upon physical resources whose interfaces and links have been configured to establish the virtual links. Therefore the information regarding their topologies is spread-out throughout the physical network. If we consider overlay networks, one will quickly realize the immense similarities between them. Since overlay networks have already been studied extensively, their topology discovery mechanisms are a good starting point for developing a virtual network discovery algorithm.

Overlay topology discovery algorithms have been widely researched in part due to the popularity of Peer-to-Peer (P2P) communities. Due to the distributed nature of P2P, the focus has been on distributed discovery mechanisms. Gossip-based broadcast algorithms, also known as probabilistic broadcast algorithms, are popular in various contexts. They are known for trading reliability guarantees for scalability properties, since they impose a smaller overhead on the network than uncontrolled flooding methods.

T-Man [30, 31] is one of such algorithms. It is gossip based and targets large scale and highly dynamic systems. Assuming random overlay networks with nodes connected through a routed network, the algorithm tries to find each node's neighbours, based on ranking functions that take into account the properties of each node, such as ID and geographic location. In this algorithm, every node maintains information about other nodes, through *partial views*, which are sets of node descriptors. Each node has two threads: an active one, responsible for initiating communications with other nodes; and a passive one that waits for incoming messages. By sharing *views*, each node will build its relevant neighbour table, i.e. its target topology.

One other algorithm has been developed in [35]. In the context of P2P networks, it is proposed a hybrid approach to peer discovery, using a Central Cache for peers not in the local network, i.e. behind some gateway with Network Address Translation (NAT) and multicast, for discovering peers within the same local network. This dual approach combines the benefits of both the centralized and distributed model.

Although none of the studied algorithms is directed specifically at virtual networks, most of the information sharing, propagation and topology building concepts may apply, and will thus lay the foundations for the proposal of a virtual topology discovery algorithm.

Management

From the InP's perspective, management tools must be provided to monitor the substrate and virtual resources. They must make sure that QoS guarantees are being uphold and should also have a way to trigger the reconfiguration of the existing VNs in order to optimize the resource allocation when needed.

Suitable VN management tools should also be provided to VNOs, VNPs and SPs so that their virtual resources can be properly configured and operated.

The management of a virtual network poses some issues, especially if the network spans multiple InPs, since in this case information must be gathered from different entities. Thus, management applications should be developed with a common interface for gathering information and performing operations on the resources. These management tools, besides providing the needed network monitoring and helping to make sure that there are no constraint violations, should also provide accountability data.

Interfacing and InP Interoperability

When requesting a new VN, a SP must, somehow, provide a description of the required VN. Due to the fact that a VN may be created by resorting to several InPs, a common VN description language, such as Extensible Markup Language (XML), should be defined.

A VN embedding situation, spanning multiple InPs, requires communication between them in order to create the so called *folding points*. Just like the requests made by the SP, this communication should also be standardized.

This interoperability also plays an important role in management and topology discovery mechanisms, where the required data will have to be gathered from multiple InPs.

Security

Although theoretically VNs should provide isolation, several problems arise in a virtualization context. The first, and perhaps the most relevant, is the safety of the physical infrastructure. For instance, if DoS attack is performed on a given InP's substrate network, all the hosted virtual networks will be affected. This problem can become even more serious if the substrate network becomes compromised. In this situation, all the virtual networks could also become compromised. A misconfiguration on any substrate resource could jeopardize all networks.

Programmability of network elements, although desired, could also increase the vulnerability if there were security holes in the programming model. Thus some initiatives, such as CABO, propose a controlled programmability scheme, where flexibility is traded-off by security.

Hosted VN should take internal measures to increase their security, using encryption mechanisms for example.

Performance

Despite the many advantages advocated by network virtualization, the VN performance must be comparable to other non-virtualized environments if the business model is to succeed. To that end, performance guarantees must exist.

There is currently no specific tool or benchmark to assess VN performance, studies should be conducted in order to evaluate the gains, or lack thereof, and costs associated with network virtualization. Efficiency, security and overhead are some of the parameters that must be looked into, so that a proper evaluation can be made.

2.3.6 Summary

The concept of network virtualization is not new and many protocols, to some extent, provide a degree of virtualization. It was long ago perceived that the physical infrastructure should not be tied to a specific protocol, and efforts were made to make them independent, e.g. MPLS.

Network virtualization inside the operators' network will provide them with the versatility to custom tailor solutions to their clients, as well as a better resource utilization and, therefore, a reduced TCO. The split of business functions will create a healthy competition environment and will reduce the management complexity at every level, in turn making the companies more agile and capable of responding quickly to shifts in market demands.

Although some virtualization tools exist today, none has been able to set a standard and provide a full network virtualization environment with the required and desired performance and flexibility. Several initiatives have taken place to address this issue. Some, like GENI and CABO, have already terminated and proposed a development framework, while others are still in development, e.g. 4WARD and AKARI. As discussed, network virtualization presents many challenges but the potential benefits are tremendous.

No commercial solution exists today for full operator network virtualization, probably because the core networks are still holding up; but in a near future, problems may arise due to the massive amounts of traffic going through the Internet's backbones and new architectures will need to be deployed. Network virtualization, due to its non-disruptive approach may very well be the solution to this future problem, whether or not it will be successful depends heavily on the companies and their shareholders.

2.4 Network Virtualization Platforms

This section presents some of the existing network virtualization platforms, their main characteristics and architectures. The emphasis will be devoted to the *VNet Management Demonstrator v0.1*, since it is the base of the developed virtualization platform.

2.4.1 GENI

GENI is an Internet Research project, whose overview was provided in 2.3.3. It presents a complex network architecture that can be divided into three main levels: physical substrate, user services and GENI Management Core (GMC).

At the lowest level, the physical substrate may be composed of routers, processors, links or wireless devices. On the top level, the user services provide the user with a set of functionalities designed to make the facilities accessible and to support the researching activities.

The GMC sits in between the physical substrate and the provided user services; its goal is to provide a common-framework that is stable and long-lasting. This placing is in the same conceptual position as the IP protocol, i.e. it is similar to the hourglass model, with the GMC being its waist.

Naming

For each network component, slice, user, or object, GENI defines GENI Global Identifiers (GGIDs) which are unique, unambiguous identifiers, that present authenticity verification.

Components

The *Components* are the main building blocks of GENI. A component may refer to a computer, a programmable router or access point. Each component is expected to provide well-defined remotely accessible interface. It is also expected that these components provide a means for slicing them among several users, either through virtualization or partitioning, granting the user a *sliver* of it. These slivers must be isolated from each other, so that they can be seen as a resource container.

These components support containment; therefore, their behaviour may be restricted, i.e. limits on bandwidth or processor usage may be imposed. In addition, common interfaces for the slivers to access the underlying network are also provided. There may be virtual server, virtual router or virtual switch interfaces.

Slices

A slice may be thought of as a set of slivers spanning several GENI components, where services may run, plus the users that are allowed to use the slivers.

The GMC provides functionalities for creating, deleting or attaining the name of a slice. After defining the slice name, the user must proceed with instantiating the desired slivers, which require an authorization request for each component. After getting the authorization *ticket*, the sliver may be created. Basic functionalities for stopping, starting and destroying slivers are provided.

Aggregates

Aggregates are defined by a set of objects or components that share some common interface. It provides a way for the GENI users to view a collection of components as a single identifiable unit, and to act upon them.

Summary

GENI provides researchers with a common network framework, with basic functionalities, that is highly programmable, and thus, could be used to test network virtualization concepts, algorithms and architectures.

Despite being a huge, mature, and complex project, with multiple workgroups and projects on several research areas, including the previously discussed PlanetLab, GENI does not provide the means for an automatic creation of virtual networks, nor their respective discovery. It simply provides the backbones upon which these mechanisms may be developed.

2.4.2 VNet Management Demonstrator v0.1

The VNet Management Demonstrator is a program developed by Asanga Udugama from the University of Bremen; its aim was to provide a demonstrator capable of showing the creation of the VNets conceptualized in the 4WARD project. The software's implementation, capabilities and issues will be discussed in the following paragraphs. Some of the displayed images were taken from the software's documentation [1].

Architecture

In order to test the proposed functionalities, the testbed in figure 2.16 was considered by the author.

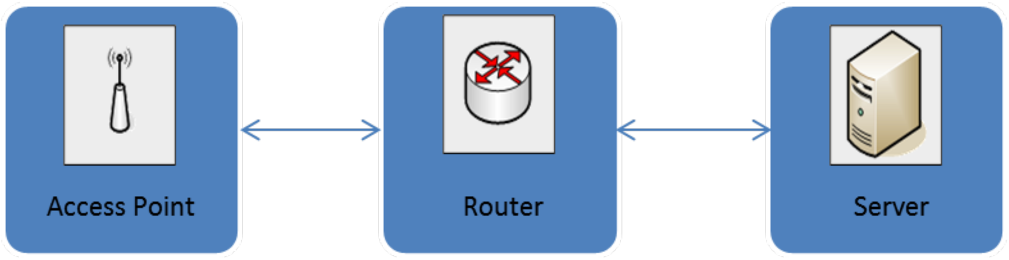


Figure 2.16: VNet Demonstrator v0.1 Testbed

It consisted of a PC playing the role of a router with two interfaces: a server with one interface and an access point connected to the router.

The developed software suite is composed by four main software modules: the Agents, the Manager, the Controller and the Visualizer which are interconnected according to the figure 2.17. The Agents, Manager and Controller were written in C; the Visualizer was written in Java.

Although the pictures' Repository is described as an SQL database in the documentation, the source code made available uses data structures inside the Manager to keep the resources' data.

Agent Modules

The Agents are executed in each physical resource and have the goal of both executing the commands given by the Manager and retrieving local resource information. In the considered testbed, there are three different Agent softwares, specific for each resource: one for the server, one for the router and another one for the access-point. The access-point's Agent runs on the router and controls it remotely.

Local data retrieval is based on statically configured files, i.e. the Agent software parses the configuration file and fills in the data structures containing the interface's configuration as well as other data pertaining to the physical and virtual resources, such as resource type and ID.

As far as command execution is concerned, the Agents are able to perform the execution of three main command types:

- *Resource Commands :*

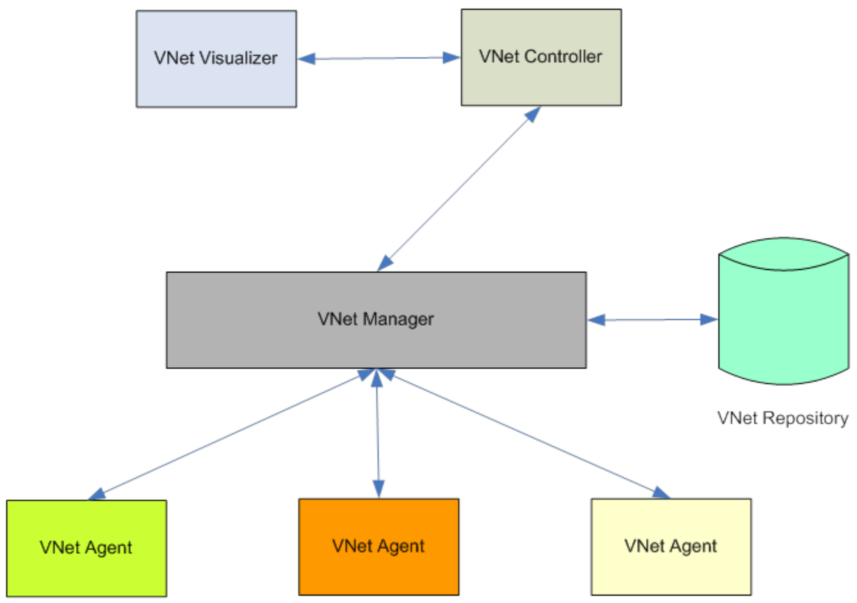


Figure 2.17: VNet Demonstrator v0.1 Architecture

These types of commands allow the bring-up and shutdown of pre-created virtual resources, resorting to scripts;

- *Link Commands :*

  This script-based commands provide the basic configuration of the virtual resource's interfaces; they work by connecting to the virtual machines through Secure Shell (SSH) and executing the pertaining configuration commands;

- *Application Commands :*

  These commands are only supported by the Server Agent and allow the execution of applications. Applications related to virtual machines are executed through SSH.

The Agents are fully threaded: there is a thread to receive commands from the Manager, another one to execute them, and a final one that deals with sending data back to the Manager, as can be seen in figure 2.18.

Despite the fully threaded architecture, the Agents suffer from some performance issues, since no thread signalling is used. Hence, every thread, except the *command receive thread* that blocks waiting for data, performs polling to the linked list containing the commands and sleeps between successive received commands.

Other issues exist with the static nature of this module: every action is preconfigured in scripts, and so are the data gathering mechanisms that are virtually non-existent since all the data gathered comes from a configuration file parse, i.e. all information is statically and manually defined.

The virtual machine creation is also a process that requires some attention. As implemented, the virtual machines must have been previously created and the commands' responsibility is simply to bring them up, which is a situation not desired in a dynamic, unpredictable environment.

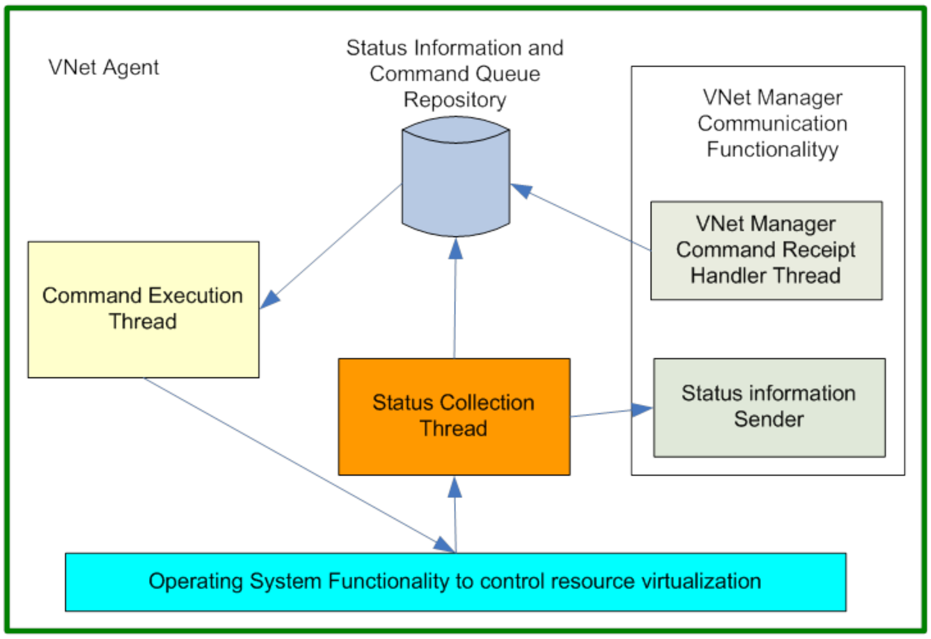

VNet Agent
Status Information and Command Queue Repository
VNet Manager Communication Functionalityy
VNet Manager Command Receipt Handler Thread
Command Execution Thread
Status Collection Thread
Status information Sender
Operating System Functionality to control resource virtualization

Figure 2.18: VNet Demonstrator v0.1 Agent detail

Manager Module

The *Manager*, figure 2.19, is a central entity responsible for handling the Agents running in the physical resources. It can be run on any physical resource, as long as IP connectivity exists to all Agents. Since it is responsible for interacting, not only with the Agents, but also with the Controller, there is some additional complexity when compared to the Agent modules. The program is structured into four main threads:

- *Incoming Agent Connection Thread :*

  This thread is responsible for accepting TCP connections from the Agents. For each new connection, a new thread designed to receive and handle the Agents' messages is created;

- *Incoming Controller Connection Thread :*

  Similarly to the Incoming Agent Connection Thread, this thread accepts incoming connections from the Controller software and launches a handling thread for each new Controller connection;

- *Command Send Thread :*

  This is the thread responsible for sending commands to each connected Agent; it fetches commands supplied by the Controller connection handler thread, identifies the related Agent and sends the proper command.

- *Repository Update Thread :*

  The responsibility of updating the Manager's repository, i.e. the data structure containing the information about all resources and links, belongs to this thread. It receives and processes messages from the Agent connection thread handler.

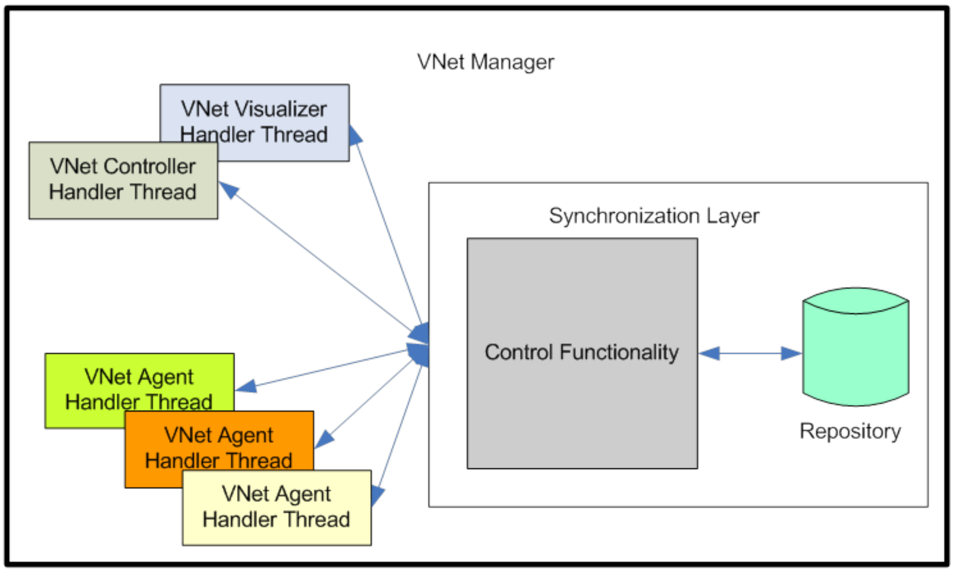

VNet Manager
VNet Visualizer Handler Thread
VNet Controller Handler Thread
Synchronization Layer
Control Functionality
Repository
VNet Agent Handler Thread
VNet Agent Handler Thread
VNet Agent Handler Thread

Figure 2.19: VNet Demonstrator v0.1 Manager detail

Just like the Agent modules, this is also a fully threaded module which has potential for high performance. Nevertheless, since there is no thread synchronization and signalling, all the message queues are polled and therefore significant delays are introduced.

Regarding the repository data structures, they lack versatility since there is one for each Agent type. By having only one type of data structure, which is generic, the repository maintainability should increase.

Controller Module

The Controller module's function is to provide a command line interface for accessing the information from the Manager, and performing the previously described Agent commands. It connects to the Manager via a TCP socket, sends the desired command, which can be a resource information request, performs link configuration, application execution command and virtual resource bring-up, among others, and waits for the Manager's reply if relevant; otherwise, it terminates.

Although the idea of having a command line interface may be interesting for performing operations or displaying bare data, for large networks its usage may become cumbersome. If, for example, a request was performed about information on ten resources and related links, the amount of data provided by this tool would probably make the data analysis difficult and not very intuitive. Besides, as it is implemented, no active connection is maintained with the Manager, thus rendering active monitoring of resources impossible.

Visualizer Module

This module aims to provide a GUI for user interaction. It provides a front-end for the Controller Module; therefore, all the options available for the Controller are also available for the GUI.

It allows the display of multiple VNets, although limited to the previously described topology.

In this module, two main threads exist: a thread responsible for fetching the data from the Controller and another one responsible for displaying it. The Visualizer module works by performing calls to the Controller module, i.e. executing it with arguments, and parsing its output. This Controller dependency, besides introducing overhead, may increase the complexity when in need to add new functionalities or correct software bugs.

Agent-Manager Interaction

This interaction is made using TCP / IP sockets where the manager plays the role of a server, and the Agents behave as clients. When the Agent starts-up, it tries to connect to the Manager, via the preconfigured port number and IP address; when the connection is successful, the previously described threads start-up, and the Agent begins its operation. The lack of a connection management thread to the Manager is a main drawback: when it disconnects, the Agent software terminates.

The exchanged messages share a common format, the fields are delimited by @@ and there is a message terminate sequence: ##. The first field indicates the message type, which can be command, information, or operation return messages.

Manager-Controller Interaction

The interaction is made through TCP / IP sockets, the Controller being the client, and only one operation is performed each time the controller is run.

Just like in the Agent-Manager interaction, the messages exchanged also have their fields delimited by @@, terminate sequence ## and can be command, information, or operation return messages.

Summary

As expected, this demonstrator does indeed provide the functionality of virtual network creation. The problem resides on the limitations of the tool and in the way in which features are provided. Every aspect of it is statically configured and based on specific configuration files and scripts.

Little to none versatility exists nor intelligence on the virtual network embedding. The tool, as it is, is not scalable: it does not predict the existence of more than two interfaces; it is bound to the presented testbed and relies on four software modules when it could do the same tasks with three. Therefore, it increases the number of points-of-failure and reduces the easiness of subsequent modifications.

Despite all the disadvantages, it also provides some potentially good features. The multithreaded nature of the Agents, the Manager and the Visualizer could lead to a versatile and high performance tool. The basis is there, but the implementation failed to take advantage of features like thread signalling or socket interoperability between C and Java for connecting directly the Manager with the Visualizer, for instance.

Chapter 3

Platform Requirements Specification

3.1 Introduction

On the one hand, network virtualization has been previously identified as a possible solution to promote innovation on the currently stagnated Internet; on the other hand, it allows the operators to decrease their networks' costs by increasing the efficiency of resource usage. To that end, this platform's goal is to provide the operators with a network virtualization solution that is easy to use, versatile, and efficient in virtual network embedding, as well as developing and evaluating virtual network mapping and discovery algorithms.

The resulting network virtualization platform provides the necessary functionalities to discover, monitor, map, deploy and manage virtual networks running on top of a substrate network.

By using this tool, network virtualization and its inherent benefits will be made available: resource consolidation, protocol independency, isolation and legacy support are some of the advantages delivered.

The purpose of this chapter is to provide the reader with a high-level overview of the platform. Besides introducing the necessary documentation and specifying the required technical background, a global view of the platform's functionality, features, required environment, intended uses, interfaces, performance and security requirements is provided.

Section 3.2 will provide a global perspective over the platform's main features, intended audience, environment and assumptions. Further details about the platform's features are described in the Section 3.3. The chapter ends with the description of Interface (3.4) and Non-functional requirements (3.5).

3.2 Overall Description

3.2.1 Features

This platform aims to provide four main features. The first one, physical and virtual network and resource discovery, is provided through distributed and centralized algorithms, the latter only for comparison purposes.

In the distributed algorithm, the nodes exchange messages between each other in order to be able to discover both the physical and virtual network topologies, while on the centralized one, a central entity is responsible for determining the existing physical and virtual links based on resource information.

Physical and virtual network and resource monitoring is the second provided feature. The platform accurately and periodically monitors the physical and virtual nodes' static and dynamic characteristics, such as:

- *CPU information :* CPU usage, number of CPUs, operating frequency, brand and model;
- *RAM information :* RAM in use, available RAM and total RAM;
- *Hard Disk Drive (HDD) information :* Existing HDDs, their usage and total size;
- *Static information :* Resource location, by group and Global Positioning System (GPS) coordinates, description and name.
- *Interface information :*
  - Physical Resources: Interface and link status, interface IP configuration, Media Access Control (MAC) address, Maximum Transmission Unit (MTU), used VLANs and link speed;
  - Virtual Resource: Interface and link status, default IP configuration and assigned interface speed.

The third feature is the virtual network deployment and mapping in the physical substrate. It utilizes a simple GUI to design and configure a virtual network. After the proper configuration and topology are defined, a request for creation will be made, and a mapping algorithm is executed. The request is then either approved, and the virtual network will be efficiently embedded on the substrate network, or refused, if it is not possible to embed the VN in the current substrate.

Finally, there is a management feature for the virtual networks. Acting upon a running virtual network is possible, the resources may be suspended, rebooted, started, shutdown or destroyed. Changing the amount of RAM allocated to the virtual resource is also possible.

3.2.2 User Classes

This platform is designed to be used by network administrators with total security clearance and access to the substrate network. The user should be experienced in Linux environments and networking.

3.2.3 Operating Environment

This platform is designed to run on Fedora Core 8 and Debian Lenny Linux distributions with the Xen kernel.

Xen was chosen not only due to its performance superiority when compared to full-virtualization hypervisors, but also because the available testbed does not support hardware virtualization on every node. Besides, since it is open-source and the *libvirt* Application Programming Interface (API) is mature for the Xen kernel, developing software to work with it is made easier.

3.2.4 Constraints

The C programming language was used for programming every module except the GUI, which was programmed in Java.

3.2.5 Assumptions and Dependencies

In order to be able to properly run the software, except the GUI, the following modules must be installed or enabled:

- libvirt;
- libxml2;
- glibtop2;
- bridge-utils;
- 802.1q module;

To properly run the GUI the Java runtime environment should be installed.

3.3 System Features Details

3.3.1 Physical and Virtual Resource and Topology Discovery

Network discovery is a fundamental feature of the software set. By providing means of automatic discovery of both the network resources and links, the network administrator can have a global view of the running networks and respective topologies at a glance, in a simplified manner.

To that end, two possible solutions exist: a centralized and a distributed one.

In the distributed approach, the nodes exchange messages to discover each other, thus every node, virtual or physical, knows exactly its neighbours. A physical node will know about its physical neighbours and about the logical neighbours of the virtual nodes running on top of it. By combining each node's knowledge, it is possible to build a map of the full topology, both physical and virtual. However, this process needs to be performed with minimal message exchanges and it needs to provide a fast discovery.

Regarding the centralized approach, a central entity is in charge of acquiring all the data pertaining to both physical and virtual resources. After gathering the relevant data, the topology building procedure begins. Since there is no link information, mechanisms similar to brute-force must be used in order to determine physical and virtual links. Thus, this approach has the potential to be computationally intensive on the central node's side. Its response to topologies changes is also computationally heavier than the distributed approach, since any change to any topology may force the full topology recalculation.

3.3.2 Substrate and Virtual Network Monitoring

Resource monitoring is fundamental if one wants to have an accurate view of the virtual and physical networks at a given point in time. The monitoring functions periodically update CPU, RAM, state, HDD and interface information; therefore, it is possible to identify diverse situations, such as failures and high resource usage, where acting on the network may be required.

Monitoring for both physical and virtual link information is provided. For physical links, both link speed and reserved link bandwidth is attained; for virtual links, the only information provided is the reserved bandwidth.

The monitoring functions are periodically performed; the periodicity is large enough so that minor transient states do not trigger too many events (e.g. link information changes when an interface keeps going up and down) but also small enough so that proper action may be quickly performed in the case of resource or link failure, for example. They are able to properly identify failures, configuration changes and load situations. Load should be categorized in levels, e.g. CPU load, which should be evaluated in terms of its time average and classified into a predetermined amount of load levels.

3.3.3 Virtual Network Creation

The virtual network embedding problem is a complex one, where both the resources and links' placement must be optimized. This dual optimization can be computationally heavy; therefore, efficient mechanisms must be developed in order to provide a "good enough" mapping that is not excessively time-consuming. In order to be properly done, i.e. in an efficient and optimized way, it requires an accurate view of the substrate and virtual networks. Both static and dynamic status and features of the substrate network must be known and up-to-date.

The virtual network creation feature requires user interaction. The user shall place the resources via drag-and-drop functions and connect them with links. After the proper network "design" and configuration, the user shall submit the desired virtual network that will be evaluated for feasibility and either approved and implemented or rejected.

In order to properly perform the virtual network deployment, the substrate network's information should be up-to-date; otherwise, less-than-optimal decisions, or even embedding refusals, may arise.

3.3.4 Virtual Network Management

A part of the networks administrator tasks is to manage the existing networks. This platform provides a basic set of management features, using the GUI a resource state may be changed to one of the following:

- *Start :* If a resource is shutdown, start will trigger the boot process;
- *Shutdown :* If a resource is running, but not suspended, it may be properly shutdown;
- *Suspend :* If a resource is running, it may be suspended, its state will be saved and resume will be possible later;
- *Resume :* If a resource is suspended, a resume may take place;
- *Destroy :* The resource and its file system will be deleted.

Besides changing the resource state, the RAM amount may also be changed, even with the virtual node running.

The access to the management features is performed through a menu when right clicking on the resource icon, on the GUI.

The management features are only available for virtual resources.

3.4 Interface Requirements

In this section, the main user interactions will be analysed, and so will the software's communication interfaces and semantics.

3.4.1 Use Cases

Several uses cases, shown if figures 3.1 and 3.2, can be considered. The *Create VNet* action can be performed by a user in order to create a new virtual network or, alternatively, a previously built XML containing the virtual network description can be loaded resorting to the *Load VNet XML* action. After performing any one of these actions, a created or loaded virtual network may be selected and additional actions may be performed to further specify and configure the virtual network. The available actions can be observed in figure 3.1.

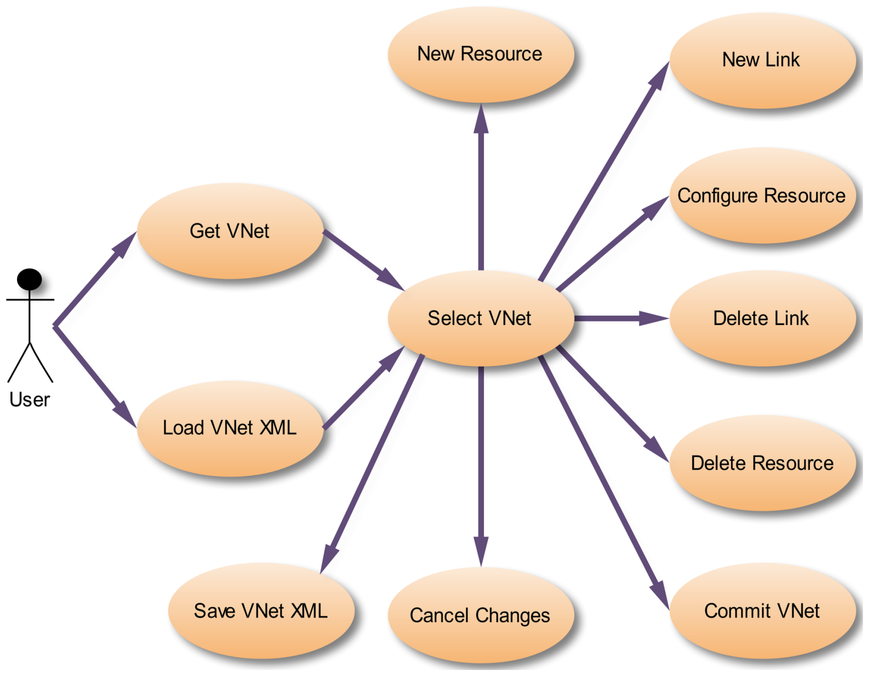


Figure 3.1: Simplified VNet Creation use-cases

In figure 3.2, different use cases are considered. One such example is the *Get VNet* action which triggers a request for a virtual network that can be afterwards managed or monitored. Multiple requests for different virtual networks may be performed. The *Manage VNet* action allows the user to delete the related virtual network, through the *Delete VNet* action, or to modify some parameters of a given resource such as its state and RAM by utilizing the *Modify Resource* action. Monitoring actions are also available through the *Monitor VNet* action that allows the user to *View Resource Properties*.

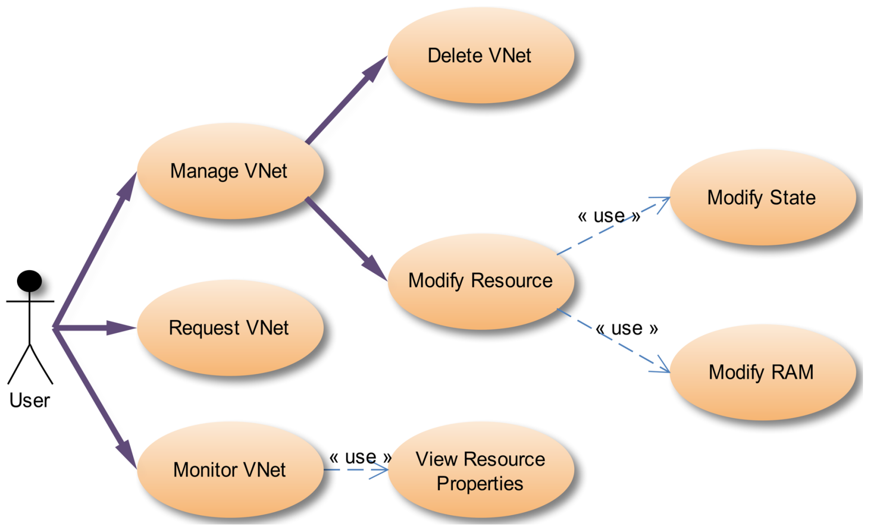


Figure 3.2: Simplified VNet Management and Monitoring use-cases

3.4.2 User Interface

Main Menus

In order to easily allow the user to perform the previously specified actions, a user interface is provided. The user interface provides the so-called *coolbar* with buttons whose functions can be easily identified by their icons. They perform virtual network build tasks, such as:

- New Virtual Router;
- New Virtual Server;
- New Link;
- Delete Link;
- Delete Resource;
- Commit Virtual Network Deployment;
- Cancel Modifications;

Besides these buttons, a menu bar is also provided, in the *Actions* menu the following actions may be performed:

- Create a new virtual network;
- Save the current virtual network to an XML file;
- Load a virtual network from an XML file;
- Quit;

There is also a *Get VNet* menu which will trigger a dropdown menu with the existing VNs. Upon selecting one virtual network, a new tab will appear with the virtual network and subsequent updates to that virtual network will be reflected in the GUI.

The last menu is the *Help* menu, which provides some information about the GUI.

Context Menus

By right-clicking in any one of the resource, a context menu will appear with the related options. The context menus will depend on the resource type and will provide information about the resources' configuration and actions that can be executed.

3.4.3 Software Interfaces

The different modules communicate using a proper message format:

$$ID\ @@\ \ldots \#\#$$

Where *ID* is the message type, a decimal number, @@ is the separator between fields and ## is the message's terminating sequence. Although not very efficient byte-wise, this message format allows for easier debugging.

3.4.4 Communication Interfaces

All communications use TCP sockets except for distributed discovery communications which rely on User Datagram Protocol (UDP) multicast.

3.5 Non-functional Requirements

3.5.1 Performance

The designed platform has a low overhead, and consequently, a small performance impact on the substrate nodes and network. The CPU and RAM usage is kept to a minimum. In the CPU / RAM usage trade-off, higher RAM usage is preferred as opposed to higher CPU usage.

3.5.2 Security

Due to its deployment features, the platform, or a part of it, must be run with elevated privileges, i.e. in root mode, and is therefore potentially dangerous for the system if it becomes compromised or misbehaves.

3.5.3 Software Quality Attributes

Due to its high importance in the operator's network, this software should be robust and reliable. It is designed in an extensible and modular manner; new features may be added without significant changes to the underlying architecture and isolated; independent tests on some of the features may be performed.

3.6 Conclusions

This chapter described the main features desired for the platform, its environment, and constraints. As depicted, the platform shall deliver an easy to use graphical interface, well-defined communication semantics, and a dynamic nature.

The presented used cases reflect the expected interactions of the user with the tool and are believed to be sufficient for monitoring, managing, and creating virtual networks.

The defined platform and library constraints shall guide and set boundaries for the development of the software.

Chapter 4

Architecture & Mechanisms Design

4.1 Introduction

The Architecture & Mechanisms Design chapter shall provide the reader with the necessary insight to properly understand all the mechanisms, modules, databases and existing dependencies of the proposed platform. By providing an in-depth view of the platform's architecture and thorough evaluation of its algorithms, the reader shall become familiar and enlightened about it.

For every desired feature, a detailed description on how to accomplish it will be provided and, in some cases, specific algorithms will be evaluated.

The Chapter starts by decomposing the platform in modules in section 4.2 and describing the data repositories on section 4.3. It proceeds with the description of the modules' dependencies (4.4) and interfaces (4.5). An overview will be given on how to accomplish the previously specified features: Section 4.7 will deal with the virtual network creation feature and related mapping algorithms; section 4.8 will proceed with designing and evaluating the discovery features and mechanisms, while sections 4.9 and 4.10 will define with the monitoring and management aspects, respectively.

4.2 Module Decomposition

This platform, the Network Virtualization System Suite (NVSS), is composed of three modules: the Agent module, the Manager module and the Control Centre module; their hierarchical decomposition can be analysed on figure 4.1. Further details about each one will be given next.

4.2.1 Control Centre module

This module is the user's front-end, i.e. the GUI. It provides the user with graphical and simple to use virtual network creation, management, and monitoring functionalities. Through drag-and-drop mechanisms, the user may design, configure, monitor and manage the desired virtual networks.

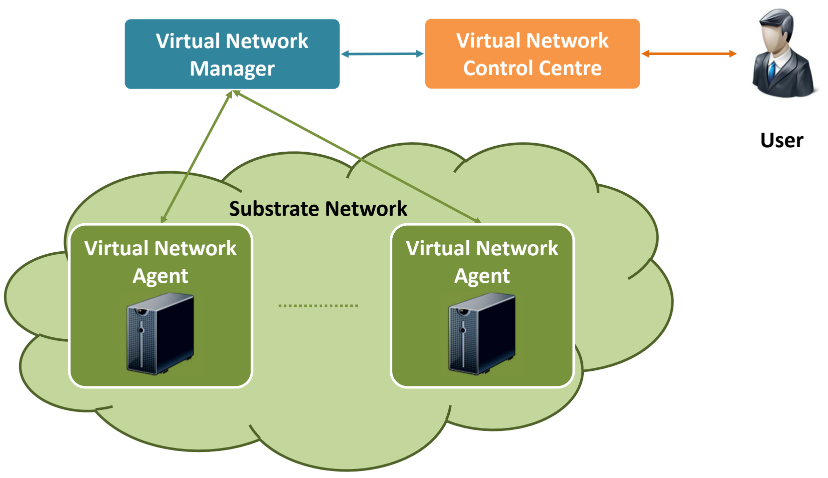


Figure 4.1: Global view of the existing modules.

4.2.2 Manager module

The Manager module's functions are many-fold: it gathers information from the Agents and sends them commands; it also aggregates their information to build the substrate and virtual networks' topologies, and to maintain an up-to-date database containing the resources' static and dynamic information. Furthermore, it is the Manager's job to keep the Control Centre with up-to-date information about its requested virtual networks and to perform its commands, such as changing the state of a resource or mapping and deploying a virtual network request.

4.2.3 Agent module

This module is designed to run on every substrate node in order to act and periodically gather data from it. The Agents send their local resources' information to the Manager, provide discovery functions through a distributed algorithm, and execute resource creation and network configuration requests.

4.3 Data Decomposition

Each one of the described modules has an internal repository where the data pertaining to the virtual networks, resources and links are stored; although similar in function, their implementation is different in each module.

4.3.1 Control Centre Data Decomposition

The Control Centre has two main types or repositories: one relating to the displayed objects that contain data about the graphically shown information, and another one containing the complete information about nodes, links and virtual networks, which is used as a basis for the display objects creation.

4.3.2 Manager Data Decomposition

The Manager needs to store information relating to the existing virtual networks, associated resources and links, and also regarding the connected Agents and Control Centres; therefore, it has three main databases.

4.3.3 Agent Data Decomposition

The Agent has two core databases: the first one stores the information about its local resources, while the second one stores data regarding its virtual or physical neighbours.

4.4 Dependencies

4.4.1 Control Centre Dependencies

The Control Centre requires the Java runtime environment and IP network connectivity. In order to be able to act on the network, the presence of the Manager is required.

4.4.2 Manager Dependencies

The Manager software requires the *libxml2* module. IP network connectivity to the Agents and Control Centres is also required.

4.4.3 Agent Dependencies

The Agents require IP network connectivity to the Manager. To perform their functions, the *libxml2*, *glibtop2*, *libvirt*, *bridge-utils*, *802.1q* and UDP multicast must also be installed and supported.

4.5 Interface Description

4.5.1 User – Control Centre Interface

In figure 4.2 the Virtual Network Control Centre (VNCC) is shown. It contains a main menu with drop-down buttons (e.g. *Actions*) and some Coolbar buttons. The virtual networks are displayed on separate tabs where the resources are drawn and interconnected with the links .

By right-clicking on a resource, a context menu appears with additional functionalities regarding resource information and configuration. These menus and buttons allow the user to perform the previously specified use-cases (figures 3.1 and 3.2).

4.5.2 Manager – Control Centre Interface

The interface between the Manager and the Control Centre is bidirectional, where TCP sockets are used in order to exchange data. The Manager works in server mode, and accepts incoming TCP socket connections on a configurable port number.

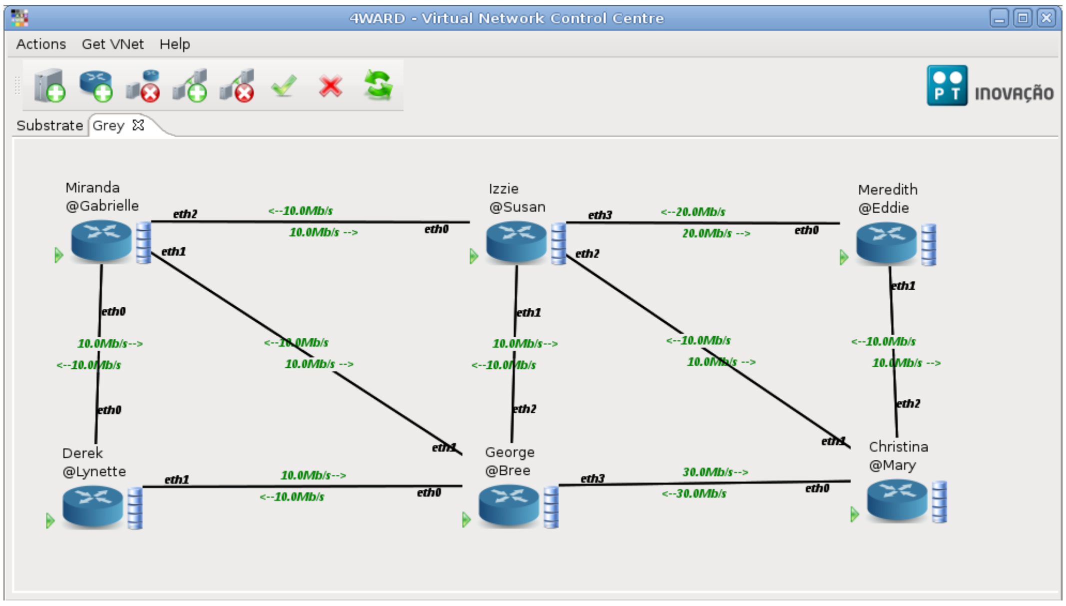


Figure 4.2: Virtual Network Control Center - User Interface.

4.5.3 Agent - Manager Interface

The interaction between Agents and the Manager is also performed using TCP sockets, in a client-server scheme, with the Manager being the server.

4.6 Identification Process

Resource Identification is a fundamental issue: both the Agents and the Control Centres must have a unique designation so that no confusion arises about who is who. The ID allocation is relinquished to the Manager that will provide the connecting Agents and Control Centres a non-utilized ID.

The ID allocation process is simple: upon a successful connection to the Manager, the connected module will request a new ID, if they have not yet received one. Afterwards, the Manager will reply with a unique ID, as clarified in figure 4.3. Unambiguous communication may then take place, both from the Manager to the Agents and Controls Centres, as well as between Agents.

The virtual resource ID allocation is different. After receiving an ID from the Manager, each Agent will build the ID of their local virtual resources based on the Managers' allocated ID; therefore, it is guaranteed that, if the Agent ID is unique, so will the virtual resource's ID be. The ID of the virtual resources has the following format:

Physical_ID@Resource_Name

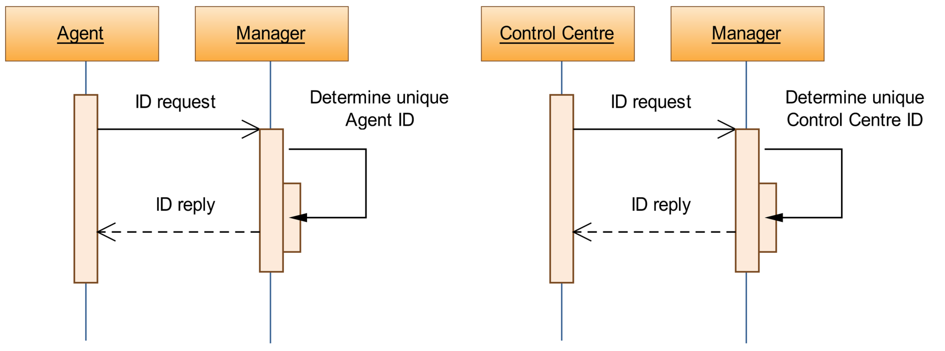


Figure 4.3: Agent and Control Centre ID attribution process.

4.7 Virtual Network Creation

4.7.1 Topology and Configuration

In order to create a new virtual network, the user may either execute the *Create VNet* option or choose the *Load VNet XML* option.

XML was chosen as the description language to store virtual networks due to its portability and the existence of tools to process it in multiple programming languages.

The Control Centre module provides the user with means to draw a new virtual network. By selecting and placing resources on a draw canvas and by connecting them with links, a virtual network may be specified. The placed resources may be configured to suit the user needs; The user may specify the resources' CPU capabilities, RAM amount, location, number of interfaces and also perform network addressing configurations. The following use case diagram illustrates the possible actions (figure 4.4):

The final step in creating a new virtual network is to commit it to the Manager. The Manager will then evaluate the specified virtual network and either accept it or refuse it. The mapping process will be described next.

4.7.2 Virtual Network Mapping

As discussed in Chapter 2, the virtual network embedding problem is a complex one and a trade-off has to be chosen between computation time and embedding optimization.

The virtual network mapping begins when the Manager receives a request for a new virtual network.

The proposed algorithm is based on the one from Zhu and Ammar [72], where a heuristic algorithm tries to optimize link and node placement simultaneously. As defined, the algorithm presents some problems. For instance, the node stress is simply considered to be the number of running virtual machines, and fails to take into consideration the reality of physical nodes, where the CPU load, core count, frequency, and available RAM amount are important factors. Regarding link stress, the algorithm considers the number of virtual links going through a particular physical link, instead of taking into consideration the reserved link's bandwidth or some other metric that takes into account the different load imposed by each virtual link.

One other crucial issue is the fact that it only takes into account perfectly homogenous physical and virtual networks, as is reflected on the single pool of candidates for virtual

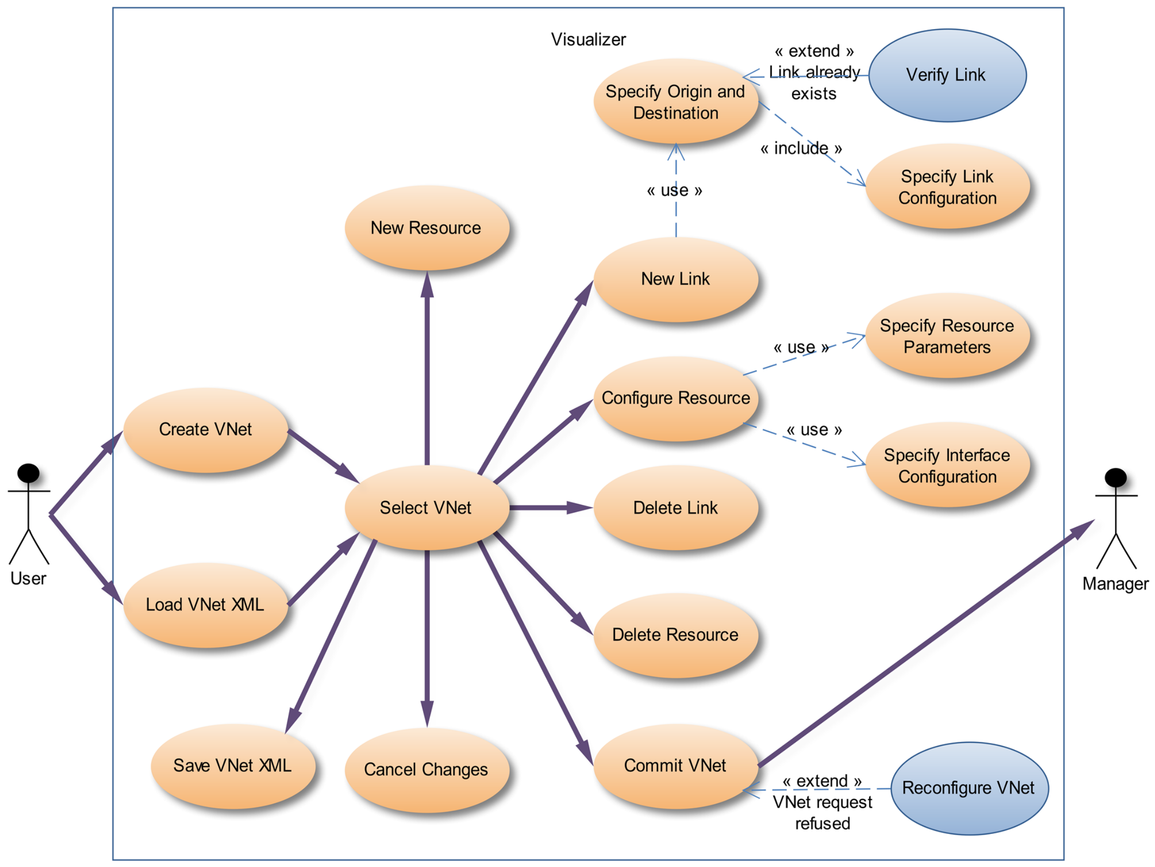


Figure 4.4: VNet Creation use-cases

resources. Node constraints such as location and required specifications are not contemplated, neither the limitations associated with links' bandwidths, which are finite and cannot be over-provisioned.

Taking into consideration the said limitations, the algorithm proposed in this Thesis starts by the determination of a link and node stress factor. Links and nodes with less stress are more prone to accepting new virtual resources.

In order to properly map a virtual network, the detailed and complete view over the substrate network must be present. The complete proposed algorithm will be described next.

The algorithm begins with the determination of Link Stress:

We define $k_j = 0...(L_{V_j} - 1)$ and $i = 0...(L_S - 1)$ where k_j is the link number of a given virtual link belonging to the j^{th} VNet, L_{V_j} is the number of virtual links in the same VNet, i is the link number of a given physical link and L_S is the number of links of the Substrate Network. One can start by establishing that the virtual link stress (S_{LV_j}) of the link k_j belonging to the j^{th} VNet is equal to its allocated bandwidth : $S_{LV_j}(k_j) = BW(k_j)$.

After all virtual links' stresses are determined, the physical links' stresses are calculated:

$S_{LS}(i)$ is the link stress of the i^{th} physical link and is defined as :

$$S_{LS}(i) = \sum_{j=1}^{N_V} \sum_{k=0}^{L_{V_j}-1} ((S_{LV_j}(k_j) | k_j \supseteq i)) \tag{4.1}$$

where N_V is the number of existing VNets. Afterwards, it proceeds with the determination of Node Stress (S_N), which is a combination of the currently available Substrate Node resources and weights active Virtual Machines, free RAM (Free RAM) amount in MB, number of CPUs (N.CPU), CPU frequency in MHz (CPU Freq.) and current CPU Load, which varies between 0 and *N.CPU*. The Node Stress of the i^{th} physical node is:

$$S_{N_i} = \frac{\text{Number of Active VMs}}{\delta + \text{Free RAM} \cdot \text{CPU Freq} \cdot (\text{N.CPU - Load})} \tag{4.2}$$

where δ is a small constant to avoid dividing by 0. The next step is the determination of Node Candidates. For each node, a set of possible physical candidates is determined based on constraints such as location, CPU number, CPU frequency and free RAM amount. After determining the candidates for each virtual node, a sorting algorithm is run that orders the virtual nodes by their number of candidates, so that virtual nodes with fewer candidates will be mapped first.

The algorithm terminates with the final Node Mapping & Path Selection. For each possible candidate v, a Constrained Shortest Path First (CSPF) algorithm to all other candidates (u) of the virtual neighbour nodes is calculated using the previously calculated Link Stresses as weights, and the path cost is stored ($D(v, u)$). The *node potential* is then determined using the formula:

$$\pi(v) = \sum_{u \in V_C} D(v, u) \cdot S_{N_v} \tag{4.3}$$

where V_C is a set containing the candidates of the neighbour virtual nodes.

Upon calculating the *node potential* of all candidates, the candidate with the lowest one is selected.

The algorithm terminates successfully when all the requested virtual nodes are properly mapped, each one on a different physical node chosen from within its candidate set, and the best-constrained paths for each virtual link are determined. For each physical network segment of a virtual link, a unique unutilized VLAN is selected, thus better utilizing the limited amount of available VLANs.

This successful mapping results in a XML, describing the mapped virtual network. By processing and breaking down the mapped XML, individual commands are sent to the proper Agents in order to both create the virtual nodes and set-up the virtual links.

The newly created resources and virtual links will then be automatically discovered by the Agents which will in turn update the Manager and consequently the Control Centre.

4.7.3 Virtual Resource Creation

The virtual node creation plays a fundamental role in the virtual network creation feature, and strongly influences the total time required until the virtual network creation is complete.

In order to provide versatility in the creation of virtual nodes, a template mechanism with hot-templates, i.e. template virtual machines with the file-system already created

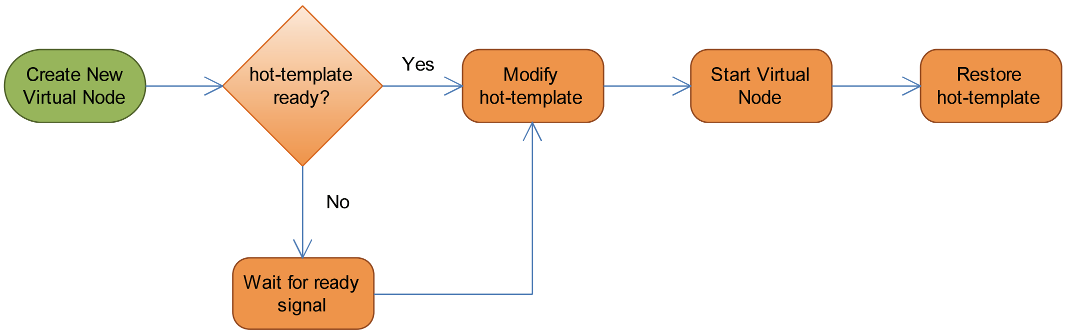


Figure 4.5: Virtual Resource creation.

but not configured, are provided. By having a pool with different standard hot-templates, e.g. a template with the Linux-based XORP [71] and other with a vanilla Debian Lenny distribution, the virtual image management becomes easier on the one hand, and the node creation becomes faster on the other hand.

The creation mechanism, illustrated in figure 4.5, works by having two ready templates per distribution: a static-template, used as a master template, and a hot-template, that will be modified upon the request for creating a new virtual node. After node creation, a cloning of the static-template will take place in order to restore the hot-template. This template cache allows a significant time reduction in the node creation times if the node requests are sparse enough so that the hot-template has enough time to be restored. This is usually the case, since a virtual network will not have two virtual resources on the same physical node, due to the network-mapping algorithm.

Although the proposed mechanism only takes into account one hot-template, it could be further extended to support many hot-templates and mitigate the performance penalty associated with multiple, consecutive, node creation requests.

4.7.4 Simulation Results

This subsection 's purpose is to simulate the developed algorithm in conditions as similar as possible to the ones found on real networks, with heterogeneous resources and links.

By testing the algorithm on these “closer-to-reality” conditions, insight and conclusions about the algorithms performance and applicability shall be taken. Regardless of the simulation parameters, the following simulations will all obey to a general procedure, which will be described next.

On the first step, a physical topology was generated using the Waxman random topology generation [69] method, with 30 physical nodes. The recommended parameters, $\alpha = 0.4$ and $\beta = 0.4$, presented some connectivity issues, especially for reduced amounts of nodes, e.g. less than 16 nodes. These settings often caused isolated nodes or clusters where there was not at least one path between every physical node. Hence, after generating the topologies, in the lack of full connectivity, nodes with less links were given additional links until full connectivity was established.

The generated physical nodes were randomly attributed a set of parameters, from a pool of possible ones, such as RAM amount, number of CPUs and CPU frequency. The physical link’s bandwidth was set at a fixed bitrate.

N. CPUs	{1; 2; 3; 4 }
CPU Frequency (GHz)	{2.0 to 3.2 in 0.1 steps }
RAM Memory (MB)	{64; 128; 256; 512 }
Link Bandwidth (Mbps)	{34.368 139.264 }

Table 4.1: Virtual Network Mapping- Virtual Nodes' parameters pool.

N. CPUs	{2; 4; 6; 8}
CPU Frequency (GHz)	{2.0 to 3.2 in 0.2 steps }
RAM Memory (GB)	{2; 4; 6}
Link Bandwidth (Mbps)	{1000}

Table 4.2: Virtual Network Mapping Simulation Scenario 1- Physical Nodes' parameters pool.

Next, virtual networks were generated using the same model, with a varying amount of virtual nodes. After generating the virtual topology, the virtual nodes were also randomly attributed a set of specifications, but in their case, the link bandwidth was also random, from within a pool of possible bandwidths. The virtual nodes' available specification pool can be observed on table 4.1.

The following step finds a solution for the virtual network mapping, using the mapping algorithm. If the mapping succeeds, the virtual nodes are placed on the physical nodes, reducing the amount of available RAM and increasing the physical nodes CPU load by a random amount. The utilization of the physical links will also increase according to the bandwidth utilized by the virtual links.

In order to evaluate the mapping algorithm, two main metrics were considered: the node stress ratio R_N and the link stress ration R_L. From the definition of these two metrics on equations 4.4 and 4.5, one can see that in a perfectly load balanced network, their value should be 1, thus smaller stress ratios indicate better virtual network embedding in the network. On the said equations, N_S and L_S are the set of substrate nodes and links, respectively. A confidence interval of 95% was considered for every result.

$$R_N = \frac{\max_{v \in V_S} S_N(v)}{[\sum_{v \in V_S} S_N(v)]/|V_S|} \tag{4.4}$$

$$R_L = \frac{\max_{v \in V_S} S_L(v)}{[\sum_{v \in L_S} S_L(v)]/|L_S|} \tag{4.5}$$

Simulation Scenario 1

The first simulation scenario assumed that the physical resources had parameters taken from the pools of table 4.2. For each generated physical network, attempts were made to map as many virtual networks as possible. It was considered that a given algorithm could not map any more virtual networks when it failed to embed 10 successive virtual networks.

Two approaches of the developed algorithm were simulated, one that starts the embedding by selecting the virtual nodes with the least amount of physical candidates, and another one that starts the mapping in a random fashion, i.e. without pre-sorting the generated virtual nodes.

The simulation was run 100 times for different virtual network sizes. The number of virtual nodes ranged from 4 to 14 in increments of 2.

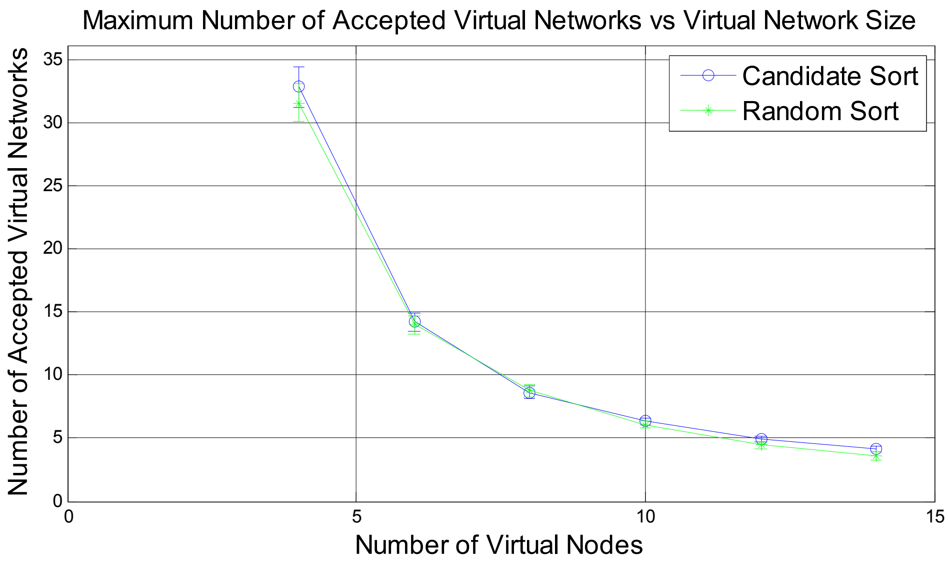


Figure 4.6: Virtual Network Mapping Simulation Scenario 1 - Maximum accepted Virtual Networks.

The results obtained may be observed on figure 4.6. As the virtual networks' size begins to grow, due to the increase in the number of virtual nodes, the number of maximum accepted virtual networks begins to decrease, which was to be expected since larger networks are harder to map due to a higher amount of constraints. Considering the case of small virtual networks, with 4 virtual nodes for example, the mapping algorithms were able to embed about 33 of them, while in the case of virtual networks with twice the virtual nodes, it was only able to embed approximately 8.

The number of maximum accepted virtual network appears to behave similarly to a decaying exponential function, with the increase of the size of the virtual networks.

The mapping algorithm performing a pre-sort of the virtual nodes, considering their amount of physical candidates, consistently shows slightly better results than the random one, particularly for simulations with few or many virtual nodes.

Simulation Scenario 2

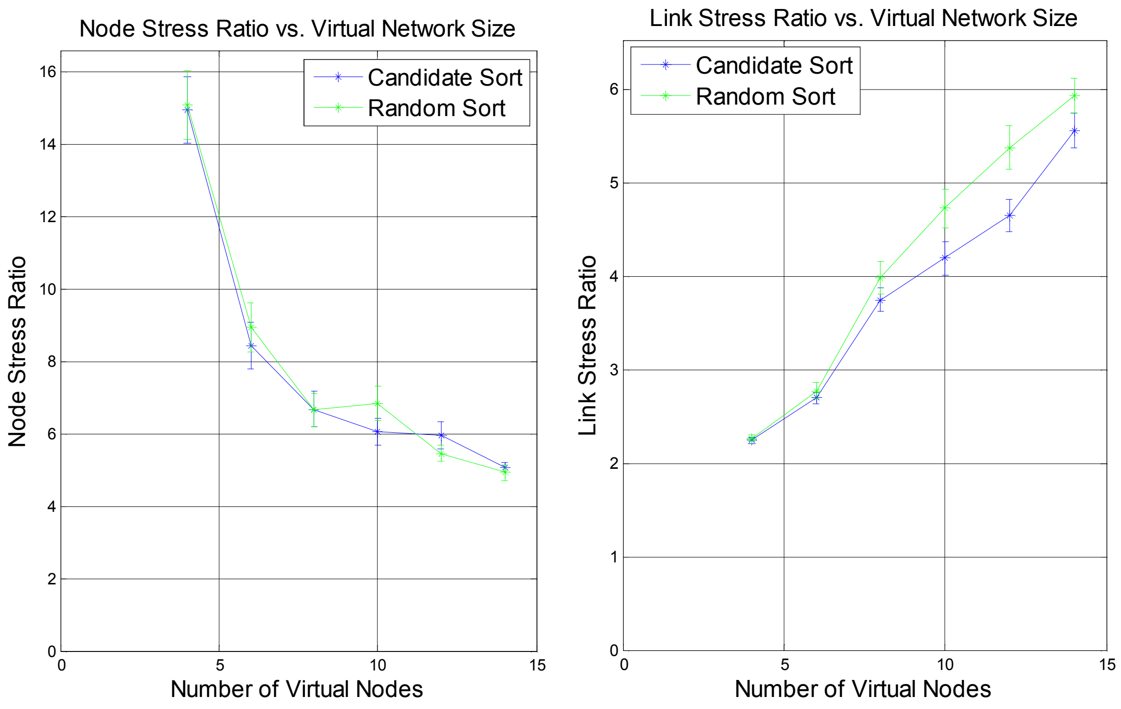
Although the maximum number of accepted virtual networks is important, the load distribution should not be disregarded. To that end, this simulation scenario aims to evaluate the performance of both approaches relating the node and link stress ratios, according to equations 4.4 and 4.5. In order to provide a fair comparison between the two approaches, an embed of 75 % of the previously identified maximum of accepted virtual networks was done, as observed on table 4.3.

The physical nodes' specifications were kept equal to the previous simulation's ones. The simulation was run 300 times for different virtual network sizes. The number of virtual nodes

Number of Virtual Nodes	Number of Embedded Networks
4	25
6	11
8	6
10	5
12	4
14	3

Table 4.3: Virtual Network Mapping Simulation Scenario 2- Number of embedded virtual networks.

ranged from 4 to 14 in increments of 2.



(a) Node Stress Ratio vs. Virtual Network Size (b) Link Stress Ratio vs. Virtual Network Size

Figure 4.7: Virtual Network Mapping Simulation Scenario 2

Starting with the figure 4.7(a), it is possible to state that, as the size of the virtual networks grows, the node stress ratio diminishes, showing that the network's node load is better distributed.

The large node stress ratio differences regarding 4 and 14 nodes virtual networks is mainly due to the way the node stress is calculated. Since the node stress is inversely proportional to the free CPU load, which varies between 0 and N_{CPUs}, one can realize that as the physical nodes become loaded, and their available CPU load tends to 0, the node

N. CPUs	{4; 8; 12; 16}
CPU Frequency (GHz)	{2.0 to 3.2 in 0.2 steps }
RAM Memory (GB)	{4; 8; 12}
Link Bandwidth (Mbps)	{1000}

Table 4.4: Virtual Network Mapping Simulation Scenario 3- Physical Nodes' parameters pool with doubled node capacity.

N. CPUs	{2; 4; 6; 8}
CPU Frequency (GHz)	{2.0 to 3.2 in 0.2 steps }
RAM Memory (GB)	{2; 4; 6}
Link Bandwidth (Mbps)	{2000}

Table 4.5: Virtual Network Mapping Simulation Scenario 3- Physical Nodes' parameters pool with doubled link capacity.

---

stress will tend to infinity.

In the 14-node virtual network embedding scenario, since only 3 virtual networks were embedded, for a total of 42 embedded virtual nodes, it was not very likely that a set of physical nodes got their available CPU load close to 0; thus, their node stress was kept at moderate levels.

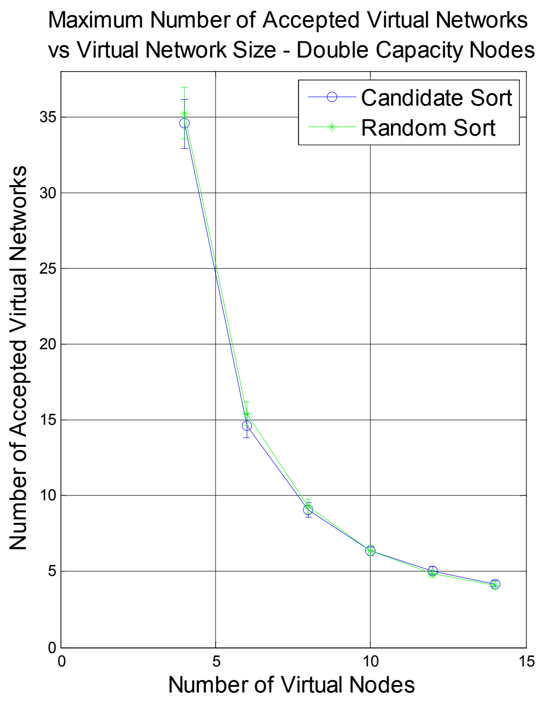
On the other hand, since in the case of 4-node networks, 25 virtual networks with 100 virtual nodes were embedded, it was more likely that some physical nodes, possibly physical nodes with fewer CPUs, got overloaded and that their node stress reflected that overload, thus producing higher node stress ratios. The node stress ratios followed a similar trend, with the pre-sorting approach faring slightly better overall.

Regarding the evolution of the link stress ratio, observed in figure 4.7(b), with the increase of the size of virtual networks, it is possible to note that it shows a growing behaviour. The reason for this behaviour is quite simple. When considering the embedding of smaller virtual networks, the granularity for link placement optimization is high; therefore, it will be easier to better take advantage of physical links with less link stress. In the case of larger and more complex virtual networks, there is less granularity in link placement, it is harder to optimize link placement due to node constraints. Through the attained results, it is possible to state that pre-sorting the nodes leads to lower link stress ratios.

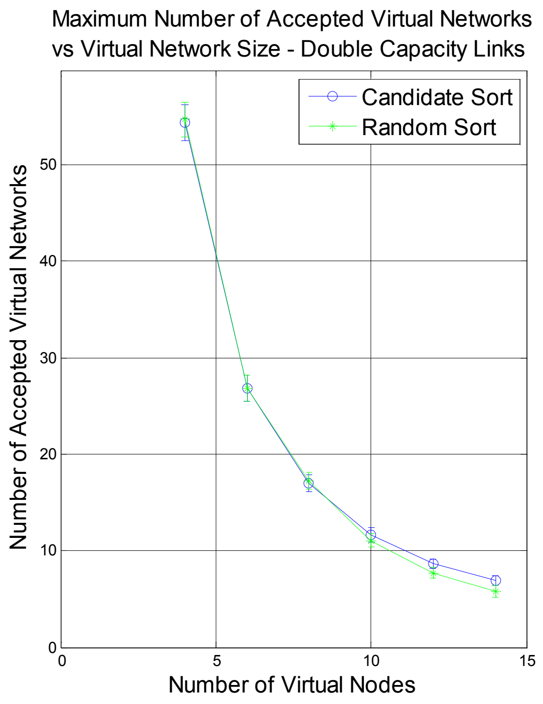
Simulation Scenario 3

In order to assess the impact of both the nodes' and links' capacity on the overall maximum of accepted virtual networks, the tests of the first simulation run were repeated considering two separate situations: the first one where the physical node's capacity was doubled, according to table 4.4, and a second one where the physical links' capacity was doubled, as observed on table 4.5.

On the first case, figure 4.8(a), the capacity of the physical nodes was doubled, only a minor improvement in the number of accepted virtual networks was achieved. In the best-case scenario, for virtual networks composed of 4 virtual nodes, the improvement was limited



(a) Physical Nodes' capacity doubled.



(b) Physical Links' capacity doubled.

Figure 4.8: Virtual Network Mapping Simulation Scenario 3 - Maximum accepted Virtual Networks.

to 2 additional embedded virtual networks. The differences in the mapping algorithm with or without candidate sorting are barely perceptible.

Regarding the doubling of link capacity, the gains realized are notorious (figure 4.8(b)). In fact, the number of accepted virtual networks almost doubled for virtual networks with 4 to 8 virtual nodes, and showed significant improvements for the other virtual networks' sizes.

Considering the accomplished results, it is clear that, for the specified simulation parameters, the main limiting factor for virtual network embedding is the links' capability. Improving the nodes' capacity barely showed any improvements, while increasing the links' bandwidth showed improvements similar to the bandwidth's increase factor.

4.8 Topology Discovery

In order to be able to properly map new virtual networks and allow the user to monitor the existing physical and virtual networks, mechanisms for discovering them are required. To that end, two mechanisms are proposed: a distributed one, that does not require the Manager's interaction, and a centralized one, performed by the Manager after receiving all the resources' information.

4.8.1 Distributed Topology Discovery

The distributed topology discovery, on the one hand, intends to reduce the required processing power on the Manager, while on the other hand, is a step forward towards distributed embedded management in the network elements (the so-called *In-Network Management*). It fosters inter-Agent communication, without depending on the Manager, that could be a key enabler to future distributed functionalities.

The proposed algorithm is based on concepts from both Spanning Tree Protocol (STP) and BGP. The Agents register themselves in a predefined multicast group and, afterwards, exchange messages with each other. The multicast group used is link-local, i.e. Time-To-Live (TTL) of 1, in order to avoid sending the discovery messages to nodes that do not want it and would otherwise need to process the packet before discarding it.

The distributed algorithm for full network discovery relies on the neighbourhood concept, where each physical node knows exactly who its neighbours are, and also who are the virtual neighbours of its local virtual nodes. By aggregating each Agents knowledge, the full topology map may be built, it works pretty much like assembling a puzzle with the pieces numbered, i.e. it is a straightforward process. The concept is demonstrated on picture 4.9.

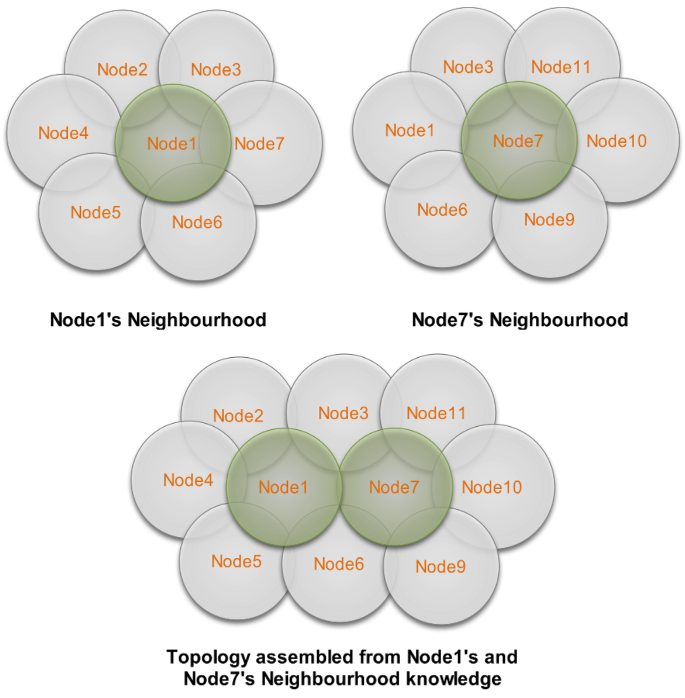


Figure 4.9: Topology Discovery – Assembling neighbourhood knowledge.

In a given network segment, one of the Agents has a special function, the Designated Root (DR) function. This node is responsible for transmitting all the information about its network segment to a new Agent arriving at the network. This approach has its roots on the spanning tree algorithm. It aims to reduce the information exchange of the distributed approach by electing a node to exchange information. This node is responsible for transmitting all the information about its network segment to a new Agent arriving at the network.

Physical Topology Discovery

Upon start-up and periodically, each Agent sends a multicast Hello message through its interfaces. The messages are specific to the interface and indicate their origin interface. This Hello message exchange allows each Agent to know its directly connected neighbours; therefore, by assembling each Agent's knowledge, as depicted on figure 4.9, it is possible to build the full physical topology network map.

Virtual Topology Discovery

The virtual network topology is not simple to achieve, since a virtual link may span through several physical links, thus, message forwarding mechanisms had to be designed.

The Agents exchange information about two kinds of resources: their local resources and the resources advertised by its neighbours that utilize the local physical node as a network hop.

Consider the simple case of figure 4.10, where a virtual link transverses one hop, P2, which has a bridge connecting its *eth0.0500* and *eth1.0800* interfaces, where *eth0.0500* represents an interface using VLAN 500 and *eth1.0800* uses VLAN 800.

Since the bridge provides layer 2 connectivity to the two segments using VLANs, a virtual link exists between P1 and P3 whose terminals are connected to the virtual resources V1 and V3. V1 and V3 have, thus, data link layer connectivity.

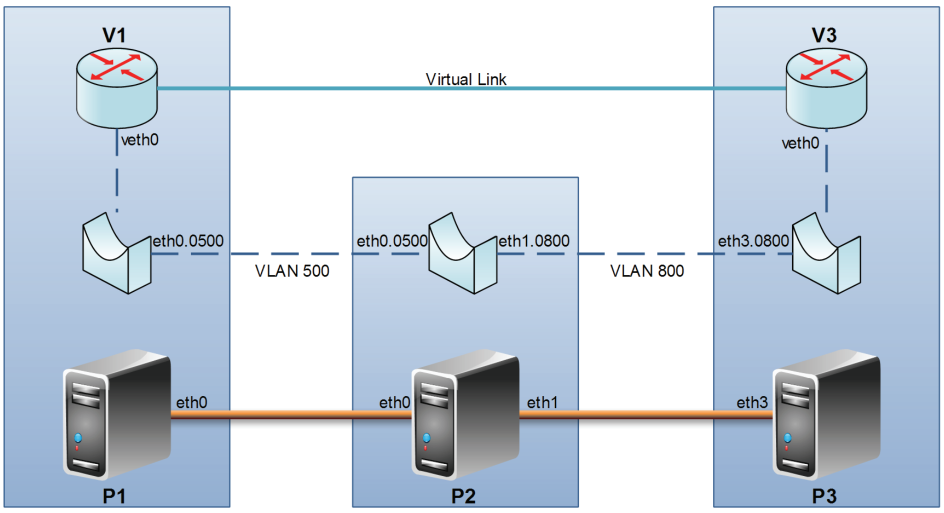


Figure 4.10: Virtual topology discovery example

In order to provide P1 and P3 with knowledge about each other resources, P2 must forward their resource advertisement messages through the proper interface. Considering the resource advertisement message coming from P1 and arriving at eth0.0500, P2 will check its bridge entries, locate a potential output interface, in this case eth1, modify the advertisement message to add a new hop, and then send the message through eth1 which will be received by P3. When P3 receives the message, it knows that resource V1 is located at P1, connected at its local interface eth3, and that the existing virtual link has one physical hop P2. P3 then proceeds to verifying the interface from which the message was sent, i.e. eth1.0800, locates the corresponding eth3.0800 and checks if any local virtual resource, in this case V3, is connected to it through a bridge. After the successful matching, it then discovers that V1 is V3's neighbour through P2 physical node and that the link transverses two physical links. The exact same mechanism also happens in the opposite direction and P1 learns about a virtual link between V1 and V3.

Designated Root

The DR is elected based on the Agents' ID , which contains an integer allocated from the Manager when an Agent starts-up; the Agent with the lowest ID on a network segment is elected the DR and is responsible for sending the networks' information to newcomers.

At start-up, every Agent 's DR is himself, after receiving a message from an Agent with a lower ID, the DR will be updated to reflect the new ID.

The DR role is not allocated *ad eternum*. The DR may crash or be shut down; therefore, mechanisms that trigger a new re-election are required. Each neighbour has an expiration timer that will trigger a new DR election if the current DR fails to communicate within a given time period, i.e. if no Hello message is received.

Pseudo-Code

The pseudo-code displayed on algorithm 1 depicts the overall mechanisms of the developed discovery algorithm. Each interface (IFC_i) has its own neighbour list and runs this pseudo-code. Since every message exchanged has a header similar to the *Hello Message*, the processing for this message type is done every time a message is received; hence, it is not specified on the *switch* clause. Further details about each mechanism will be provided in the following paragraphs.

Bootstraping Mechanism

Bootstraping is a fundamental issue when performing distributed discovery algorithms, it must be quick, efficient and reliable. The proposed bootstrapping mechanism is described in the diagram of figure 4.11. In order to begin the discovery mechanisms: the Agent must have a unique ID given by the Manager and the initial local resource discovery must be completed.

After these initial conditions are met, the discovery algorithm may start and the periodic sending of Hello messages begins. Upon receiving an Hello message, the current DR will identify a new physical neighbour and, since it is the DR, a full update regarding the network segments' associated virtual resources is sent to the arriving Agent.

On the new Agent's side, after receiving any message from a previously existing physical neighbour, an update containing its full knowledge, i.e. only local resources, will be sent. The DR will be afterwards updated based on the IDs of the discovered physical neighbours.

Considering the pseudo-code displayed in algorithm 1, the core of this process is described in lines 3 to 15.

Resource Update Mechanism

There are two main situations where an Agent advertises a resource. The first one is when a new local resource is created; the Agent where the resource resides will send updates through the interfaces related to that particular resource, i.e. interfaces that are bridged to

Algorithm 1: Per-Interface Discovery Algorithm.

```
   input : IFC$_i$
1  DR = My_ID ;
2  repeat Msg_Type = Multicast_Receive(IFC$_i$,Msg_Buffer)
3      NG = GetNeighbour(Msg_Buffer);
4      if NewNeighbour(NG) then
5          AddToNeighbourList(IFC$_i$,NG) ;
6          if DR == My_ID then
7              SendAllKnowledge(IFC$_i$) ;
8          end
9          else
10             ExclusiveUpdateDR(IFC$_i$,NG) ;
11             if DR == My_ID then
12                 SendAllKnowledge(IFC$_i$) ;
13             end
14             UpdateDR(IFC$_i$);
15         end
16     end
17     switch Msg_Type do
18         case Resource Message
19             VNG = GetVirtualNeighbour(Msg_Buffer);
20             if NewVirtualNeighbour(VNG) then
21                 AddtoVirtualNeighbourList(IFC$_i$,VNG);
22             end
23             if size (IFC$_{List}$ = LinkToOtherInterfaces(IFC$_i$,VNG))>0 then
24                 AddToVirtualNeighbourLists((IFC$_{List}$,VNG) ;
25                 SendResourceMessage((IFC$_{List}$,VNG) ;
26             end
27             CheckForVirtualLinksWithLocalResources(VNG);
28         endsw
29         case Delete Message
30             VNG = GetVirtualNeighbour(Msg_Buffer);
31             IFC$_{List}$ = LocateResourceEntriesOnAllInterfaces(VNG);
32             SendDeleteMessageThroughRelevantInterfaces(IFC$_{List}$,VNG) ;
33             RemoveEntriesOnAllInterfaces(IFC$_{List}$,VNG);
34         endsw
35     endsw
36     UpdateLastContactTime(NG);
37 until Terminate Signal;
```

one or more virtual interfaces belonging to that resource. The new resource message will thus only be sent through relevant interfaces.

The second possible situation happens when a resource may be advertised as a consequence of a received resource advertisement, i.e. there is an advertisement forwarding, as shown in figure 4.12. In this case, the physical hop that forwards the resource advertisement appends itself, its input, and output interfaces to the forwarded message, similarly to the BGP protocol. This "path tagging" allows the building of a complete virtual link map, where the physical path, with its multiple link segments, is known.

As can be observed through lines 18 to 28 of algorithm 1, the Agents receiving the new resource advertisement will place an entry on the receiving interface's knowledge database and will locate potential output interfaces for that resource, i.e. the Agents will verify if they are or not a hop for any virtual link belonging to the advertised virtual resource. If they are, they will forward the resource information through the relevant interfaces; if they are not, they will just keep the resource information stored, since it may be needed later.

Besides the forwarding mechanism, a local verification will also be performed in order to assess if any one of the local virtual resources is connected through a virtual link to the

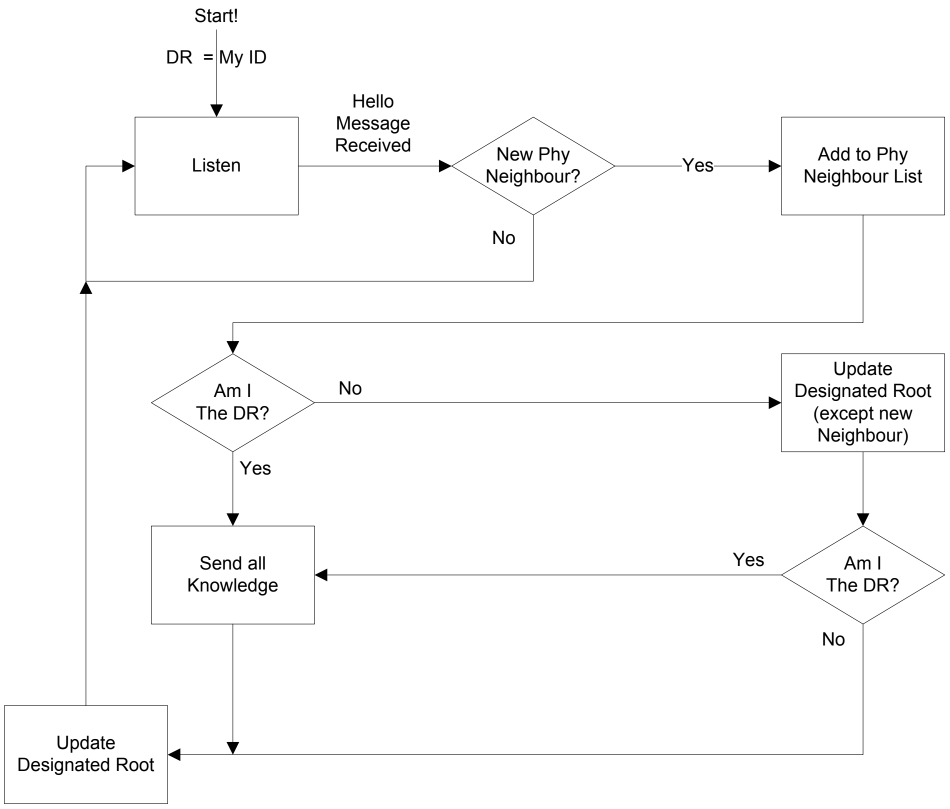


Figure 4.11: Discovery algorithm - Bootstrap diagram

received resource.

Resource Removal Mechanism

One other fundamental part of topology discovery is to be able to delete virtual resources and maintain the consistency in the existing databases. To that end, a mechanism for virtual resource removal also exists, and is illustrated both through figure 4.13 and lines 29 to 34 of algorithm 1. The forwarding mechanisms are similar to the ones of new virtual resource advertisement.

4.8.2 Centralized Topology Discovery

Algorithm Overview

As a comparison base, a centralized topology discovery algorithm was also developed. The algorithm is performed by the Manager upon receiving a pre-determined amount of physical and virtual resources, i.e. the user executing the Manager will have to know beforehand the number of existing resources, both physical and virtual.

Although a button trigger mechanism might have been used, e.g. after pressing a button the Manager would determine the existing topologies with the current resource knowledge, for testing purposes, simply specifying the number of expected resources suffices.

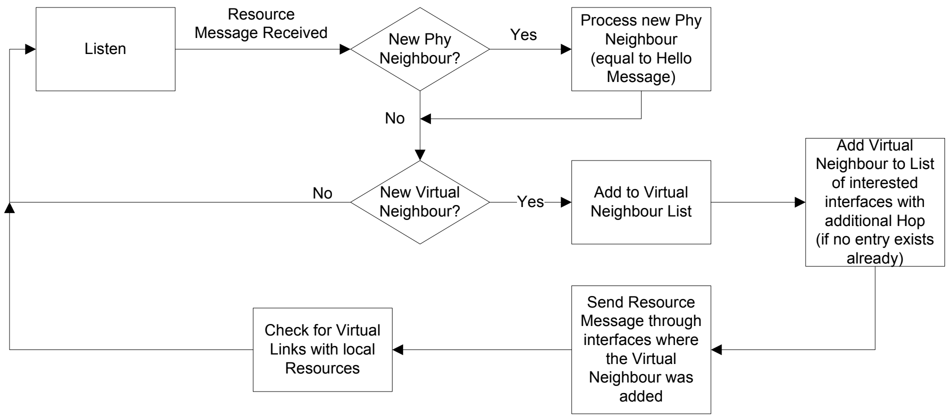

Listen
Resource Message Received
New Phy Neighbour?
Yes
Process new Phy Neighbour (equal to Hello Message)
No
New Virtual Neighbour?
No
Yes
Add to Virtual Neighbour List
Add Virtual Neighbour to List of interested interfaces with additional Hop (if no entry exists already)
Send Resource Message through interfaces where the Virtual Neighbour was added
Check for Virtual Links with local Resources

Figure 4.12: Discovery algorithm – Resource message received

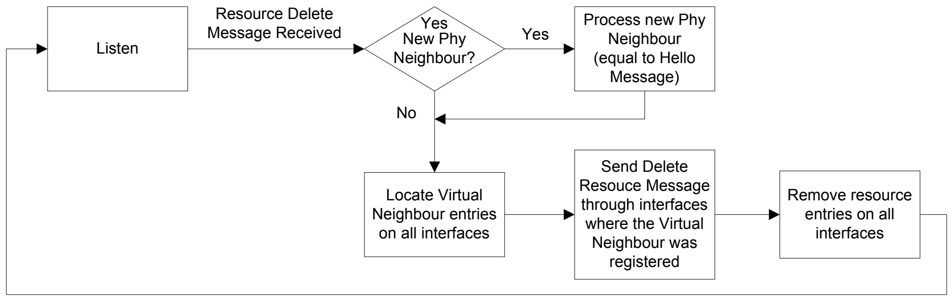

Listen
Resource Delete Message Received
Yes
New Phy Neighbour?
Yes
Process new Phy Neighbour (equal to Hello Message)
No
Locate Virtual Neighbour entries on all interfaces
Send Delete Resouce Message through interfaces where the Virtual Neighbour was registered
Remove resource entries on all interfaces

Figure 4.13: Discovery algorithm – Delete resource message received

To be fair with the distributed resource discovery mechanism, a dynamic approach should have been used, but that would imply a huge amount of CPU processing every time a resource was added or removed from the Manager's resource database, since the changes to the physical or virtual topologies would have to be determined.

Also, implementing such a simple trigger mechanism reduces the amount of modifications required to the Manager's source code, and it is a reasonable approach given the intended use.

Algorithm Design

The algorithm begins with the determination of the physical topology, i.e. by taking the physical resources one by one, checking its interface configuration and determining if any other physical resource has an interface in the same IP network. If any of them does, then it is considered that a physical link exists.

After determining the full physical topology, it is time to determine the existing virtual networks' topology. Once again, the algorithm starts by taking the resources from a given virtual network one by one. Firstly, the physical resource where the virtual resource resides is identified and the algorithm proceeds with determining potential output interfaces; i.e. physical interfaces bridged with the resources' virtual interfaces, and the VLAN associated with each physical interface is determined.

Afterwards, the algorithm locates the physical neighbours connected to each of the iden-

Number of physical nodes	Increment	Number of simulation runs
4 to 50	2	100
60 to 150	10	100
200 to 250	50	50
300 to 350	50	20
400 to 500	100	10

Table 4.6: Distributed discovery - 1^{st} simulation parameters.

tified physical interfaces, by using the already known physical topology map, and checks if they have a sub-interface using the same VLAN. If they do, the next step is to determine the bridge where the sub-interface is connected. After the bridge is found, two situations exist: Either the bridge has a local virtual interface associated to it and a new virtual link has been found, or the bridge has another sub-interface associated and the algorithm must be repeated. The algorithm is, hence, recursive.

When all the virtual networks' topology has been determined, the algorithm terminates.

4.8.3 Simulation Results

In order to assess the scalability of the proposed distributed discovery algorithm, two different tests were made. The first one tested for scalability of the algorithm with the increase of physical nodes, in the presence of a single virtual network spanning half of them; while the second one tested for scalability with the increase of the number of virtual networks.

The physical topology was generated using the Waxman random topology generation [69] method, with the same parameters as the ones used earlier on the mapping algorithm simulation on subsection 4.7.4, i.e. $\alpha = 0.4$ and $\beta = 0.4$, and with full network connectivity guaranteed.

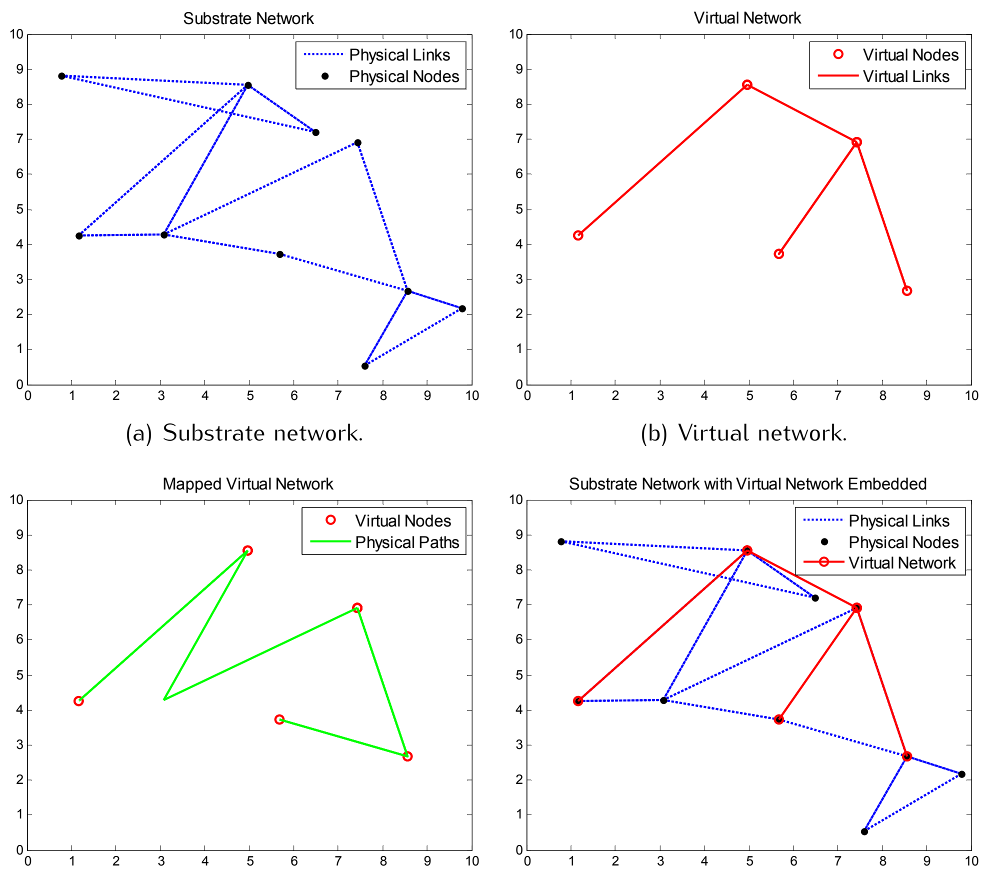
After generating a physical topology, with a given number of nodes, a virtual topology was generated on top of it by randomly selecting half of the physical nodes and creating virtual links. This virtual topology generation used a part of the Waxman method and guaranteed full connectivity of the virtual network. A confidence interval of 95% was considered on every simulation.

Since the virtual links did not match existing physical links most of the times, the Dijkstra algorithm was run in order to get the physical path for each virtual link.

In figure 4.14 one can see an example of a generated physical network, virtual network and the corresponding link mapping.

For comparison purposes, three discovery algorithms were considered: the proposed one, and two others based on uncontrolled and probabilistic flooding, with a flooding probability of 50%.

For the first simulation, a single random virtual network was generated on top of a random physical network. The number of physical nodes varied between 4 and 500 in a non-uniform way. Due to the time required to complete the simulation when in the presence of many nodes, the number of simulations runs for each considered number of physical nodes varied according to table 4.6.



(a) Substrate network.

(b) Virtual network.

(c) Virtual network with mapped links.

(d) Virtual network overlayed on substrate network.

Figure 4.14: Distributed discovery algorithm simulation example.

Considering the graphics relating the first simulation scenario, presented in figure 4.15, it is clear that the proposed algorithm is scalable and that it imposes a much smaller overhead than flooding techniques. Regarding figure 4.15(a), exhibiting the number of exchanged messages, in the case of 500 physical nodes, the difference between the probabilistic flooding algorithm and the proposed algorithm is of about three orders of magnitude.

With respect to the required simulation cycles, the proposed algorithm shows a behaviour similar to that of the flooding algorithms for less than 10 physical nodes. However, for a significant number of nodes, the number of required simulation cycles starts to stabilize on our approach, while in the other cases it continues growing.

The behaviour of the flooding algorithms in respect to simulation cycles is to be expected. Since the network size keeps on growing, so will the number of hops that forward the discovery messages; thus, the number of cycles required for the messages to reach every node is proportional to the number of physical nodes.

In the proposed algorithm, the path crossed by the discovery messages is a previously optimized one; therefore, the number of simulation cycles required for convergence is much smaller. The stagnation in the required number of cycles observed is due to the number of physical hops utilized by the virtual links, being kept approximately constant with the

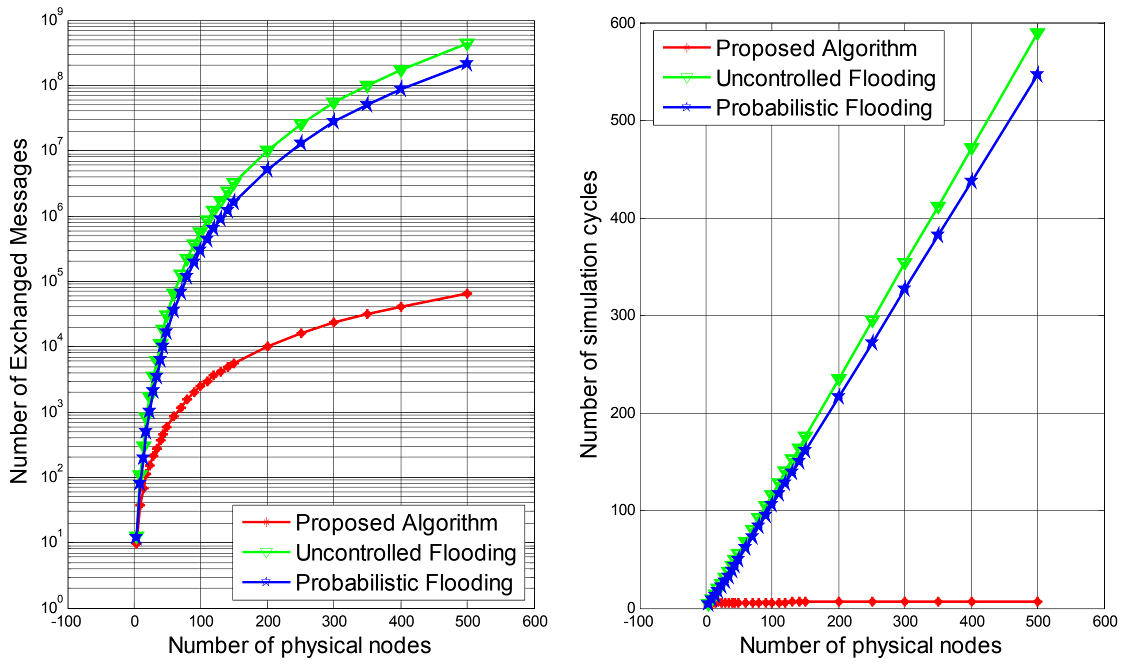


Figure 4.15: Discovery Algorithm Scalability Tests – Number of Physical Nodes.

increase of substrate nodes, since the virtual networks are always created with half of the physical nodes.

Figure 4.16 shows the results obtained with the increase in the number of virtual networks on top of a single substrate network. It is possible to observe that its behaviour follows a linear trend, i.e. that the number of exchanged messages and required simulation cycles are much reduced when compared to the other approaches.

4.9 Substrate and Virtual Network Monitoring

Resource monitoring is fundamental if one wants to have an accurate view of the virtual and physical networks at a given point in time. The monitoring functions periodically update the resources' information, therefore it is possible to identify diverse situations, such as failures and high resource usage, which may require immediate action. Monitoring for both physical and virtual link information is also provided.

To provide proper updated information, every Agent periodically checks its local resources' configuration and status, and reports back to the Manager if any change occurs. These triggered, event-driven updates reduce the overhead traffic on the network. Several parameters are monitored: CPU load (that is classified according to 6 equally distributed levels), RAM, HDD usage, interface and link status, interface bridge attachment and configuration, number of running virtual machines and their state. If a resource crashes or becomes misconfigured, the network administrator will have this information and will be able to take proper actions.

Consider the case of a stable virtual network, i.e. a network with a constant load on its resources and links and no changes on its configuration. The Agent monitoring each of the network's resources will periodically verify their configuration, load, and state. If an

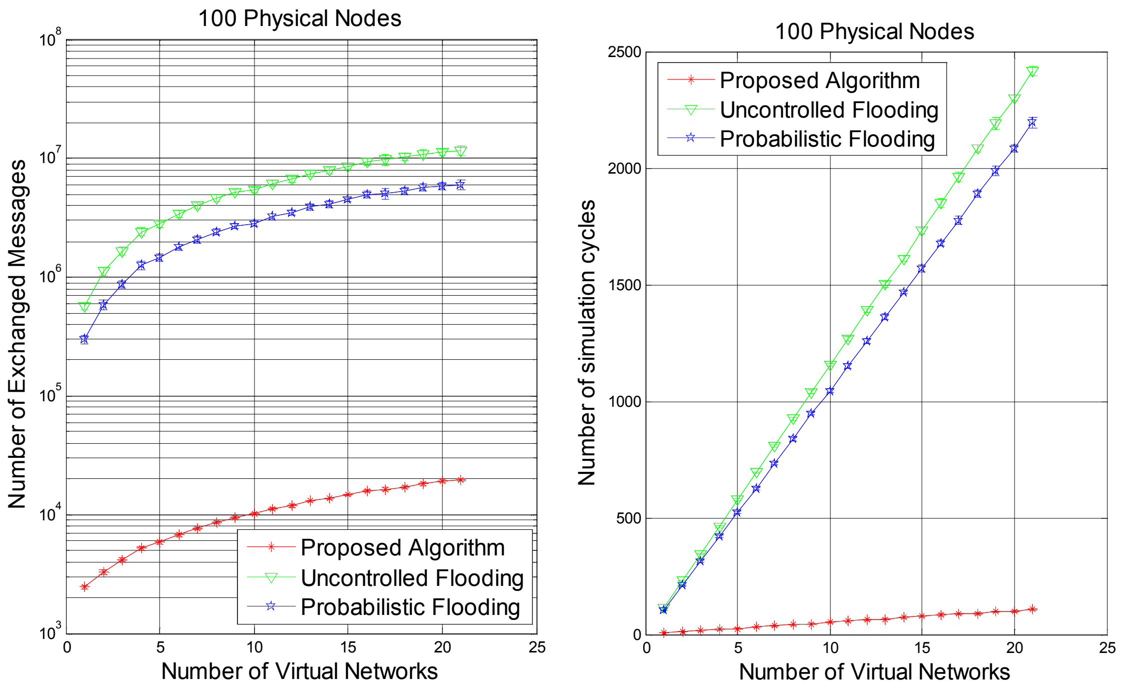


Figure 4.16: Discovery Algorithm Scalability Tests – Number of Virtual Networks.

interface fails, the Agent will realize that the interface state has gone from *up* to *down*, and will therefore send the resource's information to the Manager.

The Manager will then process the resource information, compare it with its local knowledge about the resource and will also realize that the interface state has changed. As a result, a subsequent update will be sent to the Control Centre which will update the interface state from up to down and erase all the existing links from the user's view. Facing *disappearing links* the user will verify the related interface, realize that a problem exists with it and he will take proper action.

A simpler case could be demonstrated with a virtual node changing its state from running to shutdown, or a sudden change in resource load. The Agents, Manager and Control Centre would behave in the same way and the user would quickly identify these situations.

4.10 Virtual Network Management

Just like the previously described monitoring ability, the management feature, i.e. being able to act on the network's resources is also a fundamental one. To that end, some functionalities are provided such as changing the virtual resource state, i.e.: rebooting, shutting down, suspending or powering the resource up, the amount of assigned RAM memory, deleting the resource or even the full virtual network.

The assigned RAM memory may be changed in runtime. This feature can be particularly useful if, for some reason, the resource requires a larger amount of RAM in some time-periods while needing less in others. If less RAM is utilized, it may be made available to other running virtual resources.

The delete feature is available for either single resources or the complete virtual network, and greatly simplifies the administrator work by automating the delete procedures: the virtual machine is removed, its file system is destroyed and the associated bridges and VLANs are freed. The physical machines quickly return to a "clean" state. The virtual network delete request is sent to every Agent on order to allow hop-Agents to clean-up their bridges and release used VLANs.

Other use-cases regarding not only the management but also the monitoring features are exhibited in figure 4.17.

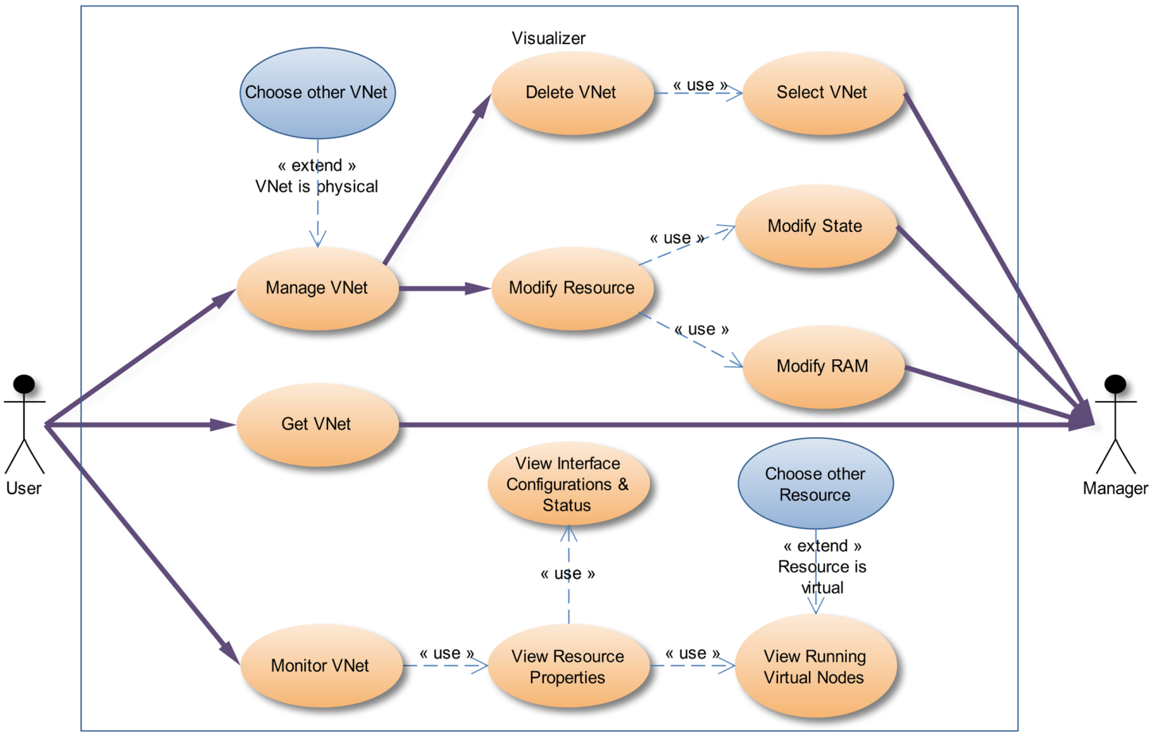


Figure 4.17: VNet Management and Monitoring use-cases

4.11 Conclusions

An analysis of the NVSS software was provided. Its modules, architecture, main databases, communication and user interfaces were described. The Control Centre and its main aspects were described and so were the pertaining use-cases.

Each intended feature was carefully scrutinized. Emphasis was given on topology discovery and embedding algorithms.

Virtual network discovery algorithms, despite being fundamental for a future virtualized network, have not been a major research target in the past few years. The developed distributed discovery algorithm succeeded on providing a simple, fast and low overhead virtual and physical topology discovery algorithm, as proven both by the obtained simulation results and the experimental tests conducted.

A distributed approach to topology discovery was designed and simulated. Considering the attained results, it has been proven that this distributed algorithm is scalable in terms of the number of exchanged messages, required cycles to complete and amount of virtual networks. The results show that the number of exchanged messages can be reduced by about three orders of magnitude when compared to algorithms using uncontrolled or probabilistic flooding. Unlike these flooding algorithms, the number of cycles required for topology discovery remained approximately constant with the increase in the network size.

Since the virtual network embedding problem is a complex one, an heuristics based algorithm was proposed in order to reduce both the embedding complexity and the required computational time. Simulations were made that showed the algorithm's performance considering heterogeneous physical and virtual networks, and conclusions were taken about the impact of the virtual networks' size, and the specifications of both physical nodes and links.

The use-cases and desired functionalities for both monitoring and management features were provided. Several parameters were identified as necessary for proper network monitoring, such as CPU load, available RAM, physical and virtual links' and nodes' states. The management features include changing the state and the RAM amount of the virtual machines in run-time.

Chapter 5

Software Implementation

5.1 Introduction

As result of the previous specification and design chapters, this chapter's purpose is to describe the implementation of the analysed features, functionalities, and algorithms.

In-depth details will be provided about both the composition and the behaviour of every module. Their data structures, threads, main functions, exchanged messages and mechanisms will be discussed.

Some of the main used libraries, APIs and open-source functions will be described first, on 5.2. Then, the Virtual Network Control Center on section 5.3 will be explored, followed by the Virtual Network Manager on section 5.4 and finally the Virtual Network Agent on section 5.5. The chapter will conclude with a brief resume and analysis of the implementation details on section 5.6.

5.2 Auxiliary Functions and Libraries

5.2.1 XML parsing

One of the reasons for working with XML descriptions is the existence of XML APIs for both Java and C, thus these files may be exchanged and processed easily. *JDOM* [27] was used for parsing in Java while *libxml2* [67] was used for C.

5.2.2 popen_noshell

popen_noshell [21] is a C function that presents the same functionality as a *system* or *popen* call but requires far less CPU cycles to execute. This improved speed and lower resource usage made it "the" choice for executing commands from within the Agent software, such as configuring or gathering bridge information and configuring interfaces, among others.

Associated with the unlocked version of *fgets*, command execute and result parsing speeds are greatly increased.

5.2.3 libvirt

This API was described on chapter 2. As stated, it presents a standard interface for interacting with multiple hypervisors, hence its functions were used extensively in the Agent module both when gathering resource information and when creating or acting upon virtual resources.

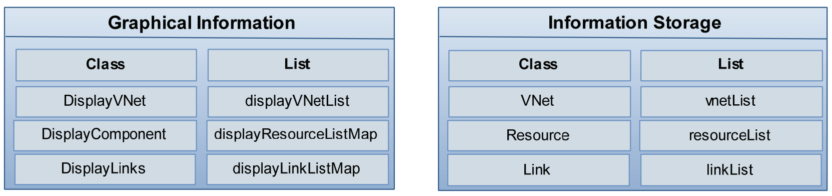
Since a connection to the hypervisor is required on every Agent, the static function *cmdConnect()* was implemented, and so was a synchronization mutex in order to provide mutually exclusive access to the hypervisor. When a function call is performed, the function will verify the state of the hypervisor connection, and try to connect if it is not yet established.

5.3 Virtual Network Control Centre

5.3.1 Databases and Classes

Before going deeper into the Control Centres' database description and details, one concept must be explained: The concept of a hash-map. A hash map presents a one to one relationship between a key and a value. By hashing the key, a hashing function transforms the key into an index of an array where the value is stored.

There are six main databases in the Control Centre modules (5.1), three containing display objects and three other containing the resources', links' and virtual networks' information.



Graphical Information

Class	List
DisplayVNet	displayVNetList
DisplayComponent	displayResourceListMap
DisplayLinks	displayLinkListMap

Information Storage

Class	List
VNet	vnetList
Resource	resourceList
Link	linkList

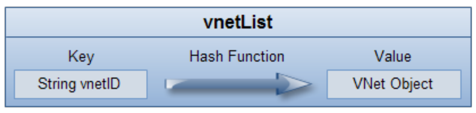
Figure 5.1: Control Centre's Classes and Lists.

The*VNet database*, called *vnetList*, uses a hash map to store its keys and values (figure 5.2(a)). The keys are the unique VNet ID's while the values are VNet Objects, represented on the class diagram of figure 5.2(b).

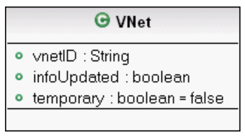
The *Resource database*, implemented as a *resourceList*, is also based on hash maps. Its *Resource* objects store data according to the class diagram of figure 5.3(a). There can be two main types of resources, physical (*PNode*) and virtual (*VResource*), and the latter can also have two types, either virtual node (*VNode*) or virtual router (*VRouter*).

The *Link database* also uses a hash map to store its link objects (figure 5.3(b)).

The remaining three databases store display objects. Starting with the *Display VNet database*, its database is called *displayVNetList* and the hash map is similar to the previous ones: the VNet ID is the key while the value is the DisplayVNet object.

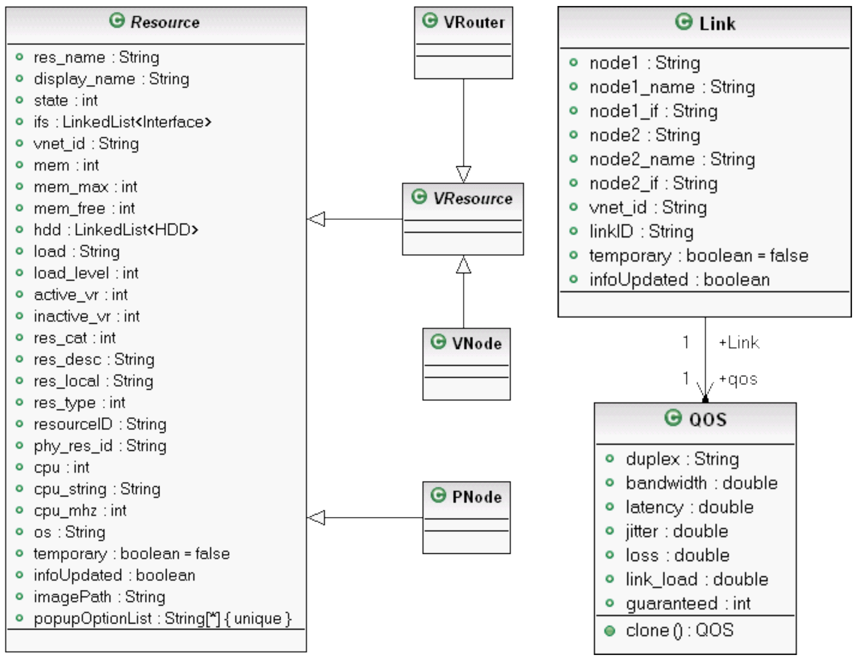


(a) VNet list hash map.



(b) VNet class.

Figure 5.2: Control Centre's VNet list hash map and class.



(a) Resource class. (b) Link class

Figure 5.3: Control Centre's Resource and Link classes.

The *DisplayComponent* database uses a double hash map to store its data, the first key is a VNetID. It leads to another hash map whose key is now the resource ID and the value is a DisplayComponent object.

Finally, the *Display Link database* utilizes a hash map whose key is a VNet ID that leads to a linked list of DisplayLink objects. The Display objects are described by the classes shown in figure 5.4

5.3.2 Module Decomposition

The entry Control Centre class is the *VNetVisualizer* class that is in charge of reading the Control Centre's configuration file, containing the necessary information to connect to the Manager (figure 5.5). Afterwards, a new object is created; the *Controller* initializes the previously described databases and launches the Control Centre's two main threads: The *Model* thread and the *View* thread, as depicted in figure 5.6.

Model thread

This thread is mainly used for listening to the Manager's messages and, aside from the exception of requesting the Manager an ID at start-up, it is a passive thread. Its main function is to process the Managers' messages and update the related databases.

After receiving the ID, the Model thread operation, which may be seen on figure 5.7, is straightforward: it has a main "while" loop that blocks on TCP socket *readline* calls. Two successive calls are performed to *readline*, the first one is used to determine the message type, while the second one receives the actual data. It is assumed that the message

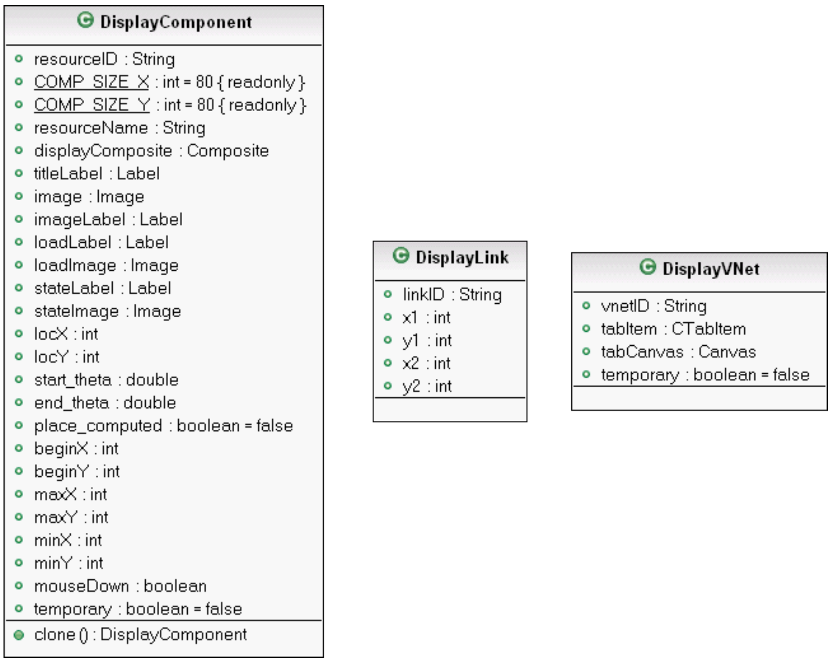


Figure 5.4: Control Centre's Display classes.

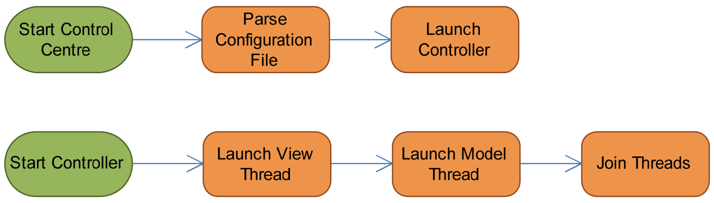


Figure 5.5: Control Centre Start Up.

containing the message type immediately precedes the message with the actual data.

The message types accepted by the Model thread are described in table 5.1.

View thread

The View thread (figure 5.8) is responsible for the graphical interface. It is in charge of handling user interactions, of displaying the networks' topologies and resource information, and of sending requests to the Manager. All the software's functionalities are presented to the user through the View thread.

The view thread uses the Java's Standard Widget Toolkit (SWT) as the main graphical API. At start-up, a global initialization of the graphical interface, i.e. menus and tabs, is performed and a request is automatically made for the substrate network.

The generated GUI can be decomposed into three main sections: the dropdown menu, the *coolbar* menu and the tab canvas.

The dropdown menu, displayed on figure 5.9, provides the user with several functionalities for virtual network management and monitoring: the *Action* menu allows the user to create and delete new virtual networks, to load or save an XML with virtual network description,

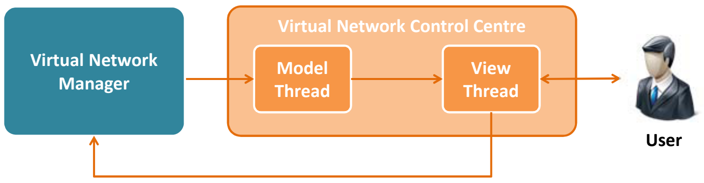


Figure 5.6: Control Centre Module.

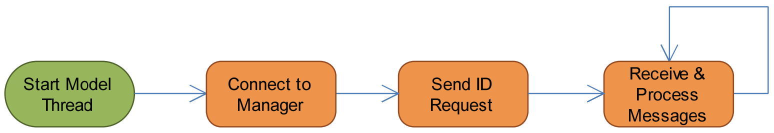


Figure 5.7: Control Centre Model Thread Diagram.

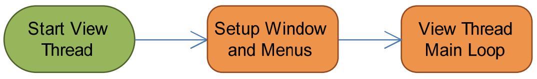


Figure 5.8: Control Centre View Thread Diagram.

and to exit the Control Centre. The *Get VNet* menu provides the user with a list of available VNets that may be chosen for monitoring; selecting a given virtual network based on its ID will create a new tab with the requested virtual network.

The second main section of the GUI is the coolbar shown in figure 5.9 that provides the user with quick access to regular functions when creating a virtual network. These functions include adding a new virtual server or router, removing a virtual resource, adding or removing a virtual link, committing or cancelling the modifications, and refreshing the data of the current opened virtual network tab.

The last section of the GUI is the tab canvas where the virtual networks are drawn, as seen in figure 5.10. By selecting a tab, the View thread will draw the corresponding virtual network. Each resource has additional menus that can be accessed by right clicking on it.

The user actions upon the Control Centre will sometimes require it to send messages to the Manager. The exchanged message types are described on the table 5.2.

The View thread's main loop periodically performs updates on the graphical objects, so that the existing menus and figures always reflect the current networks' status. The diagram of figure 5.11 exemplifies the actions taken.

The loop begins by removing the display objects that correspond to no longer existing content, i.e. resourcesand links of virtual networks that have been removed in the meanwhile. Next, checks are made to see if any new virtual network, link or resource has to be created, and if they do, the display objects are created with the matching information, menus and event-handlers. Subsequently, they are placed on the respective VNet canvas.

The placing function for a given virtual network begins by selecting the node with the highest amount of links and placing it in the centre of the canvas with a total available radial angle of 2π. Afterwards, a recursive function distributes the neighbour nodes in a radially uniform way and attributes them a smaller radial angle. An example result of node placement algorithm is showed in figure 5.12 where one can clearly observe that the node with the highest number of links was placed on the centre of the drawing canvas, and that

Message Type	Description
MV_MSG_USER_INFO	Allows the Manager to send a message that will be displayed to the user.
MA_MSG_ID_REPLY	Manager reply to a Control Centre's ID request. Contains a unique ID.
MA_MSG_VNET_LIST	Contains a list of the Manager's known virtual networks.
MV_MSG_UPDATE_LINK	Contains information about a given link.
MV_MSG_DEL_RESOURCE	Delete command for the specified virtual resource.
MV_MSG_DEL_PHY_RESOURCE	Delete command for the specified physical resource.
MV_MSG_UPDATE_RESOURCE	Contains information about a given resource.

Table 5.1: Manager to Control Centre message types.

Message Type	Description
MV_MSG_NEW_VNET	Create a new virtual network, based on a XML description.
MV_MSG_DEL_VNET	Delete the specified virtual network.
MV_MSG_GET_VNET	Request information and updates for the specified virtual network.
MA_MSG_VNET_LIST	Request a list with the existing virtual networks.
MV_MSG_START_VM	Start the specified virtual node.
MV_MSG_SHUTDOWN_VM	Shutdown the specified virtual node.
MV_MSG_REBOOT_VM	Reboot the specified virtual node.
MV_MSG_PAUSE_VM	Suspend the specified virtual node.
MV_MSG_UNPAUSE_VM	Resume the specified virtual node.
MV_MSG_DESTROY_VM	Delete the specified virtual node.
MV_MSG_SET_MEM_VM	Change the specified virtual node's RAM amount.

Table 5.2: Control Centre to Manager message types.

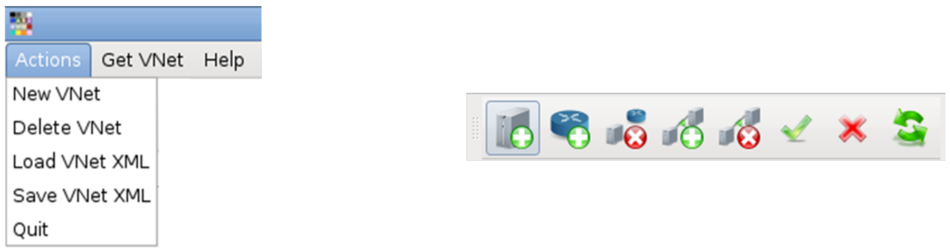


Figure 5.9: Control Centre's Dropdown & Coolbar menus.

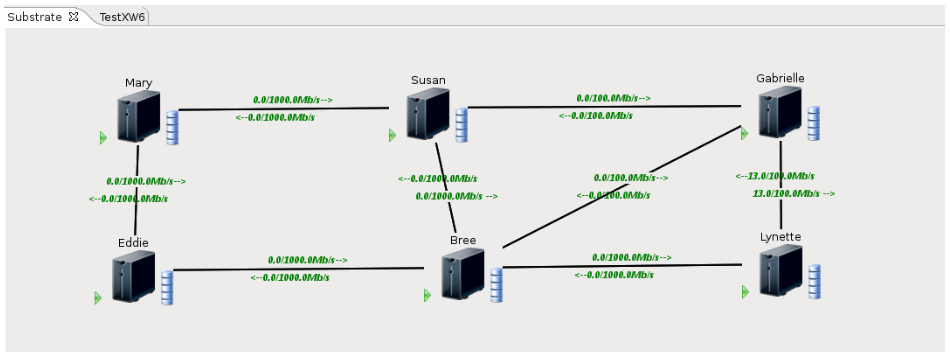


Figure 5.10: Control Centre's Canvas.

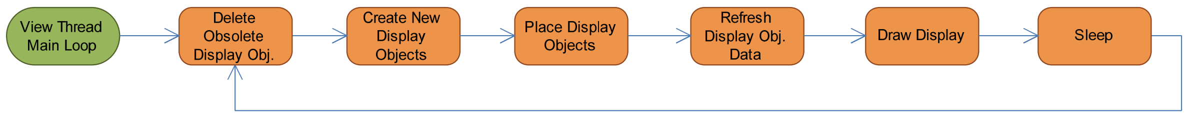

View Thread Main Loop
Delete Obsolete Display Obj.
Create New Display Objects
Place Display Objects
Refresh Display Obj. Data
Draw Display
Sleep

Figure 5.11: Control Centre View Thread main loop Diagram.

its neighbours were radially distributed with a $\frac{\pi}{2}$ radial angle interval.

Then, the refresh function will go through each resource and update its respective *DisplayComponent* object regarding its current state, load image and menu information which may have information that is not up-to-date.

The final step will be to trigger the actual canvas redraw by using the *drawDisplay* function.

After updating the display objects, the thread sleeps 500ms before checking for new updates. The 500ms update interval was chosen as a compromise between keeping the information updated as fast as possible and not having a significant impact on the usage of the computing resources. The chosen update interval is small enough to be barely perceptible by the user.

In the implementation, care is taken to ensure the proper synchronized access to the Control Centre's databases.

5.3.3 Virtual Network Design & Configuration

This subsection will begin by explaining how the Control Centre design features were implemented, and how the designed virtual network is forwarded to the Manager to be mapped.

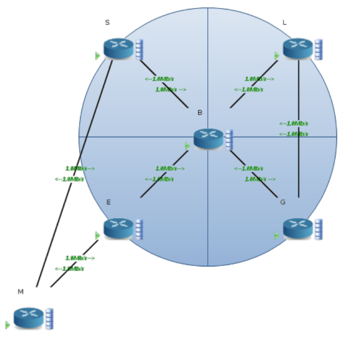


Figure 5.12: Control Centre's Radial resource placement.

As previously specified, the Control Centre uses Java's SWT API to create graphical objects and manage user interactions. The View thread is the one in charge of presenting the interface to the user and handling user-generated events.

When the user selects the *Create new VNet* command, the Control Center's View thread will create a new DisplayVNet object, and the related tab and drawing canvas. The following paragraphs will describe the events necessary to the VNet design and configuration.

Placing New Resources

After the creation of the new VNet tab, the user selects one of the resource buttons on the coolbar that triggers the creation of a temporary Resource and the corresponding DisplayComponent, so that an image of the resource appears on the screen.

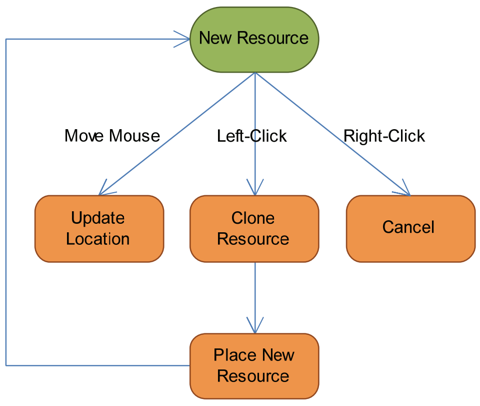


Figure 5.13: Control Center - New Resource Diagram.

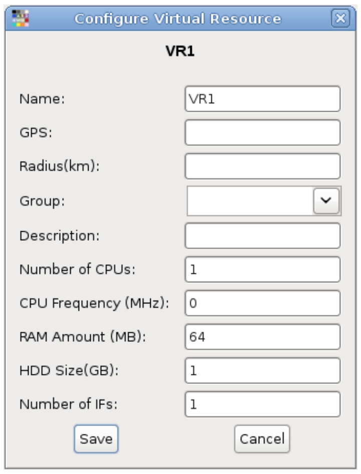
This display component has a set of associated mouse event-handlers, as illustrated by figure 5.13: the move handler causes the icon to change according to the mouse movement, by updating the coordinates of the display component and triggering a redraw of the canvas. The right-click handler will cancel the *Insert Resource* command, delete the temporary object, and reset the coolbar selection. Finally, the left-click handler will clone the temporary object and add it to the resource list of the new VNet, with some default configuration parameters such as 64MB of RAM, 1 CPU and 1 interface.

After the resource placement, the user is able to continue to place resources at will, until it presses the mouse's left-button.

Configuring New Resources

After placing the desired resources, the user may proceed with configuring them. Upon resource placement, event-handlers were added that trigger the opening of a configuration menu when the mouse right-click is pressed, and allow the resource to be moved when left clicking and dragging.

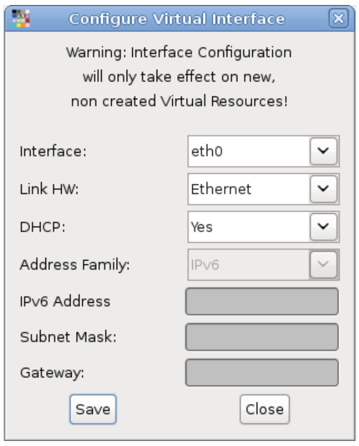
After right-clicking and selecting the *Configure Resource* option, a menu object is created that appears and allows the subsequent modification of the resources configuration, such as location, RAM amount, number of interfaces, etc. The complete menu is shown in figure 5.14(a).


Configure Virtual Resource

VR1

Name:	VR1
GPS:	
Radius(km):	
Group:	
Description:	
Number of CPUs:	1
CPU Frequency (MHz):	0
RAM Amount (MB):	64
HDD Size(GB):	1
Number of IFs:	1

(a) Configure Virtual Resource.


Configure Virtual Interface

Warning: Interface Configuration will only take effect on new, non created Virtual Resources!

Interface:	eth0
Link HW:	Ethernet
DHCP:	Yes
Address Family:	IPv6
IPv6 Address	
Subnet Mask:	
Gateway:	

(b) Configure Virtual Interface.

Figure 5.14: Configuring a new Virtual Resource

One other menu, the *Configure Interfaces* menu on figure 5.14(b), allows the configuration of every virtual resource interface. Only IPv4 and IPv6 were considered, although modifications to support other network protocols are possible and simple. The specified interface configurations will be enforced upon resource creation.

Configuring New Links

One other button on the coolbar allows the placement of new links. By selecting the option, a selection menu object will be created and the user will be prompted with a request to specify both a source and destination for the link through a drop-down menu that lists all the VNet's resources (figure 5.15).

After proper selection, i.e., if no link exists already and if the source is not equal to the destination, a new configuration menu will be created where the link details can be specified, as shown in figure 5.16. When all the desired parameters are configured, the link will be created and drawn in the canvas using Java's class *GC*. A straight line connecting both resources, with each resource's interface specified and the link bandwidth placed on top of the connecting line, symbolizes the link. Additionally, two link entries are added to the VNet's link list, each one representing a link direction.

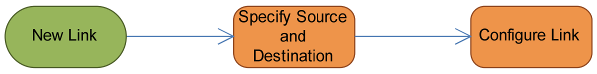
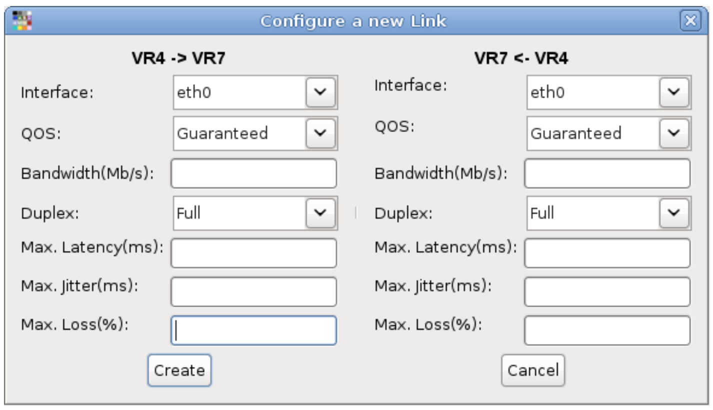


Figure 5.15: Control Center - New Link Diagram.



VR4 -> VR7		VR7 <- VR4	
Interface:	eth0	Interface:	eth0
QOS:	Guaranteed	QOS:	Guaranteed
Bandwidth(Mb/s):		Bandwidth(Mb/s):	
Duplex:	Full	Duplex:	Full
Max. Latency(ms):		Max. Latency(ms):	
Max. Jitter(ms):		Max. Jitter(ms):	
Max. Loss(%):		Max. Loss(%):	

Figure 5.16: Configuring a new Link

Saving and Committing a Virtual Network

After all resources and links are placed and configured, the user has the option to save the design for later use, or commit it to the Manager (figure 5.17).

If the user chooses to save the design, it may access the main menu *Actions* and select the *Save VNet XML* option that will trigger a file browser window to appear. After choosing the desired file name, the *XMLAddNodes* and *XMLAddLinks* methods will build the XML structure and the *XMLOutputter* will be in charge of placing the appropriate data on a *FileOutputStream* object created after specifying the file name.

In order to proceed with the virtual network creation, the user has to select the coolbar button *Commit*, that will trigger the sending of an XML message to Manager, built using a similar process to the *Save VNet XML* command, but writing to a *StringWriter* instead.

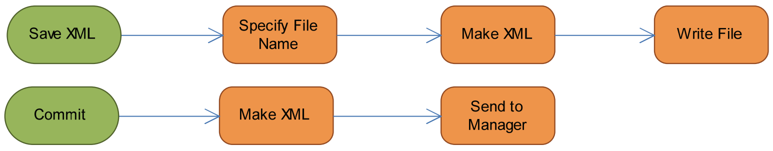


Figure 5.17: Control Center - Commit & Save Diagrams.

The created string may then be sent through a TCP socket. Since the generated XML will have an undetermined size, the message will be sent in several parts, depending on both the message size and on the constant *MSG_BUFFER_SIZE*. The messages will have a field containing the VNet name and the message part number. The last message will have an additional field containing the string *XML_END* to signal the completeness of the XML message.

5.3.4 Virtual Network Monitoring

If the monitoring of a virtual network is desired, the Control Centre provides a main menu button, the *Get VNet* button, that will request a list of available VNets to the Manager and display it (figure 5.18). After selecting the desired virtual network, the Control Centre will register itself as interested in the specified virtual network, so that the Manager will not only send it the current information regarding the requested virtual network, but it will also send subsequent updates to that virtual network.

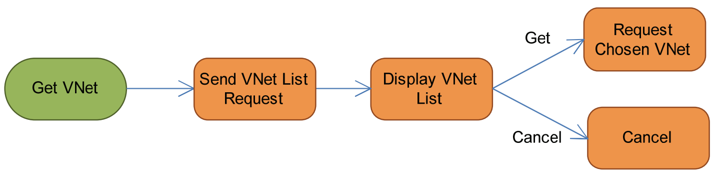


Figure 5.18: Control Center - Get VNet Diagram.

This “per-request” mechanism ensures that the connected Control Centres do not become overloaded with information about virtual networks that are of no interest to them.

After requesting the virtual network, the previously described drawing mechanisms will display it on the proper tab canvas, similar to the one of figure 5.10, where the virtual resources and link can easily be monitored. On figure 5.19 it is possible to see the available load and states for a given virtual resource, which can be monitored almost in real-time.

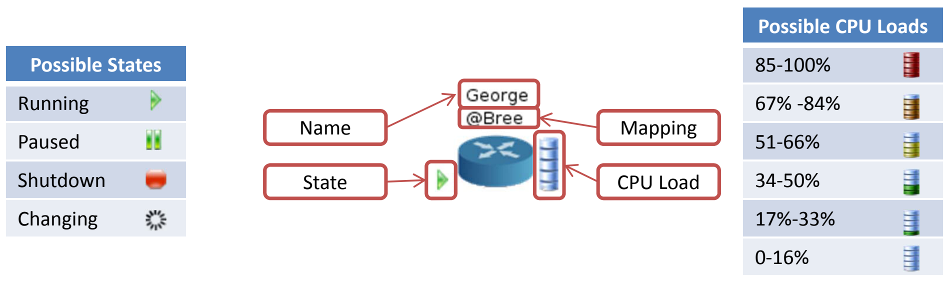


Figure 5.19: Control Center - Virtual Node Monitoring.

5.3.5 Virtual Network Management

The management features includes changing the state of a virtual resource, the RAM amount in run-time, deleting the resource or even the complete virtual network.

The access to the resource management functions is done by right-clicking the virtual node and selecting the desired action. The Control Centre will then use the Manager connection to send a command message containing information about the desired action and the target virtual resource or network.

The message follows the standard format:

CMD_ID @@ Control_Centre_ID @@ Data # #.

The data field depends on the command message type. If, for example, one wants to change the RAM amount, it will contain the resource ID and the target memory amount.

5.4 Virtual Network Manager

5.4.1 Main Databases and Structures

Main Databases

The Manager uses linked lists to store all its data. Regarding the resources', links' and virtual networks' data storage, there are 2 main databases: the *VNet List* is a linked list of pointers to VNet structures, each one containing a linked-list of pointers to *VNet Nodes* which in turn also have a linked list of pointers to related *Links*. This hierarchical architecture optimizes resource and link searches. The full data structure hierarchies can be seen on figure 5.20

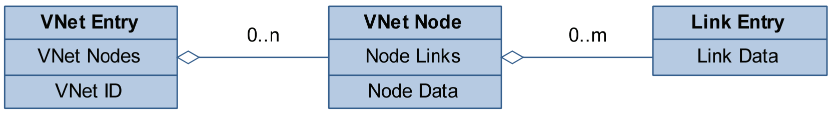


Figure 5.20: Manager's VNet Entry.

The second database is a redundant one, the *Main Resource List* is a linked list that holds pointers to all the resources. This pointer database is kept for compatibility purposes.

Each database has its own associated mutex, used for synchronizing access from multiple threads.

Auxiliary Databases

Since the Manager accepts connections to multiple Control Centres and Agents, the storage of information regarding the connected modules is required. To that end, additional linked lists exist that store the Agents' and Control Centres' information. These connection entries are depicted in figure 5.21.

The connected Agents' data structure contains the necessary socket ID, physical resource ID, information about the Agent's connection handler thread and the time of last contact.

The data structure containing the Control Centres' connection information is a bit more complex. It contains the Control Centre's ID, the requested VNets, a temporary list and an internal message linked list used for sending messages to the respective Control Centre. Thread synchronization and control variables are also included within this data structure.

5.4.2 Module Decomposition

As discussed in the previous chapters, it is the Manager's job to assemble the entire network's information, provide unique IDs to each Agent and Control Centre, and to act on the networks. The subsequent sections will describe each one of the Manager's threads, their roles and interactions. The Manager's start-up sequence is described on the diagram of figure 5.22.

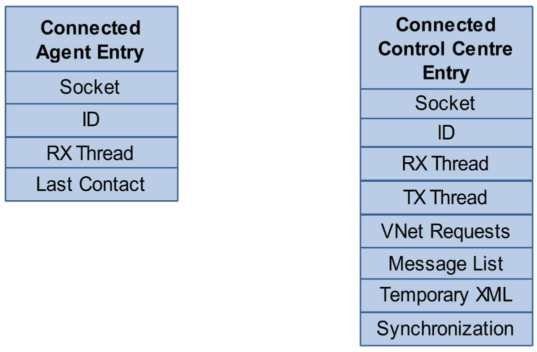
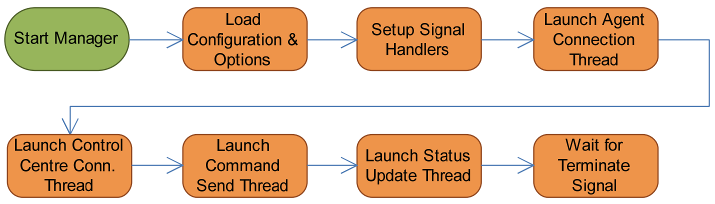


Figure 5.21: Connected Agents and Control Centres Entries.

The Manager is responsible for connecting both multiple Agents and multiple Control Centres. To that end, threads exist that launch additional handler threads for each connected Agent and Control Centre.



Start Manager
Load Configuration & Options
Setup Signal Handlers
Launch Agent Connection Thread
Launch Control Centre Conn. Thread
Launch Command Send Thread
Launch Status Update Thread
Wait for Terminate Signal

Figure 5.22: Manager - Start-Up.

Besides the connection handling threads, there are additional threads used to process the received resources' and pending links' information, update the Manager's databases and trigger Control Centre updates: if a resource or link update is detected in a virtual network previously requested by one or more Control Centres, an update will be sent via the respective Control Centre's TX thread.

Figure 5.23 summarizes the existing threads on the Manager's module.

Agent Connection Accept thread

When an Agent tries to establish a connection with the Manager, this is the thread that will handle the connection request. The thread start-up and Agent accepting process can be described through the diagram of figure 5.24.

If a TCP socket connection is accepted, a new agent connection data structure is created and filled in with the related socket ID, thread information and other additional fields. This socket ID will allow receiving and sending data from and to that Agent.

In order to receive the Agent's messages in a parallel way, an Agent handler thread is launched for each connected Agent; therefore, there will be as many Agent handler threads as Agents. The admissible messages types to be received are described in table 5.3.

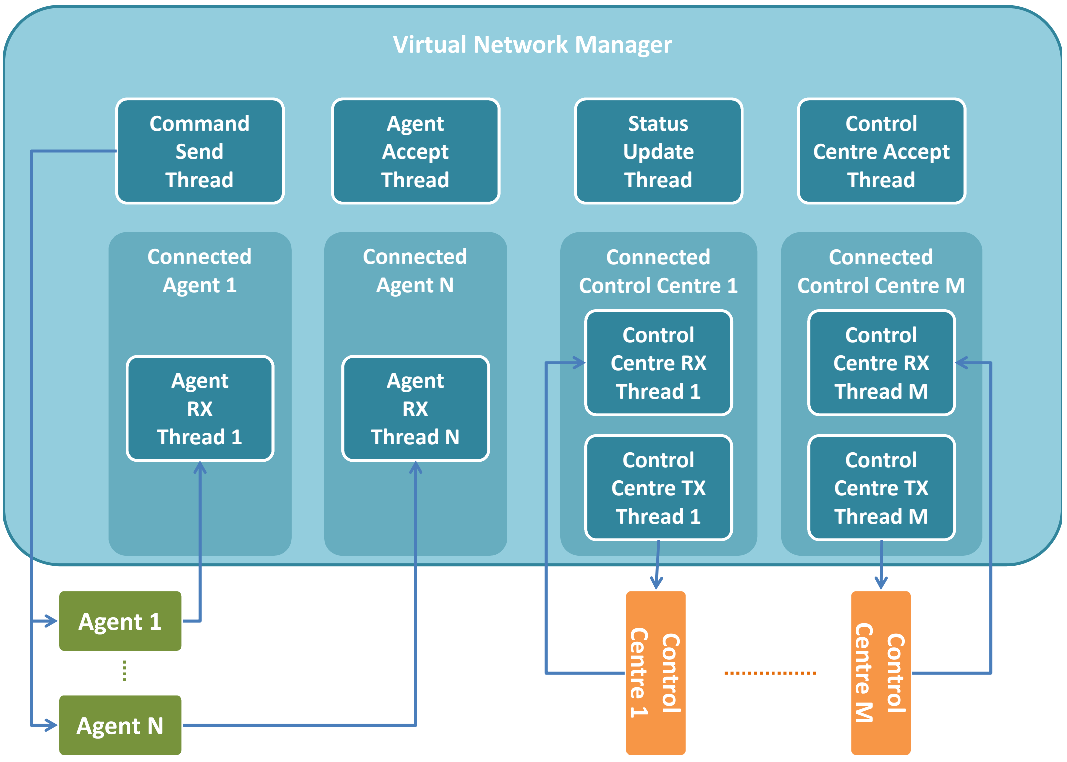


Figure 5.23: Manager Module

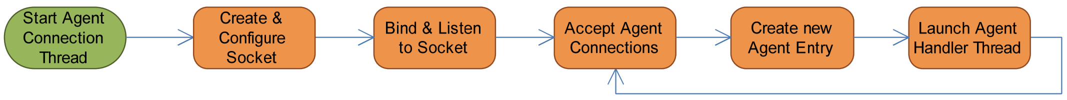

Start Agent Connection Thread
Create & Configure Socket
Bind & Listen to Socket
Accept Agent Connections
Create new Agent Entry
Launch Agent Handler Thread

Figure 5.24: Manager - Agent Connection Accept Thread.

Agent Connection Handler thread

As previous explained, one of these threads is created when a new Agent connects. At start-up, clean-up handlers are registered; next, the thread enters its main loop that blocks waiting for incoming messages and processes the received messages according to their message type.

The clean-up handlers assure that, when the connection is lost to a particular Agent, the utilized blocking socket receive function will generate an error that will trigger the thread exit and the respective cleaning mechanisms.

The Agent will then be removed from the connected Agent's list, the related physical and virtual resources and links will be removed and the Control Centres will be notified of these deletions.

These cleaning mechanisms shall provide the required databases consistency.

Message Type	Description
MA_MSG_ID_REQUEST	Request an ID.
MA_MSG_KEEPALIVE	Used for keep-alive messages.
MA_MSG_TYPE_INFO	Send resource information.
MA_MSG_DEL_RESOURCE	Signal the deletion of a specified resource.
MA_MSG_TYPE_NEIGHBOUR	Inform the Manager about a new neighbour, physical or virtual.

Table 5.3: Agent to Manager message types.

Control Centre Connection Accept thread

Similarly to the Agent Connection Accept thread, this thread accepts Control Centre connections and launches the respective handling threads, but in this case two threads will be launched per connected Control Centre: a Control Centre Connection handler for RX and another one for TX (figure 5.25).

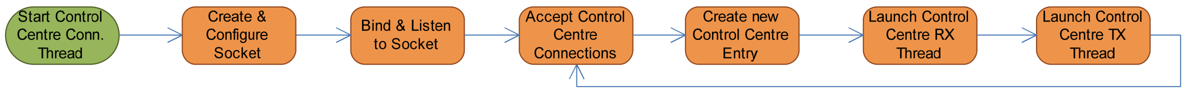

Start Control Centre Conn. Thread
Create & Configure Socket
Bind & Listen to Socket
Accept Control Centre Connections
Create new Control Centre Entry
Launch Control Centre RX Thread
Launch Control Centre TX Thread

Figure 5.25: Manager - Control Centre Connection Accept thread.

Just like in the Agent's case, for each accepted Control Centre connection, a Control Centre connection data structure containing relevant thread control variables and message list is created.

The Control Centre RX thread is similar to the Agent Connection handler one; it will block on a receive call to the connected Control Centre's socket and process the received messages afterwards. It accepts the message types described in table 5.2.

The Control Centre TX thread also blocks waiting for messages on its message list. Upon receiving the signal of a new message available on the list, the message will be processed and sent to the pertaining Control Centre. The available message types were previously described on table 5.1.

Status Update thread

This is the thread that will process the resource information messages received by the Agents' Connection handlers. Every resource message received is processed by this thread that checks for resource updates and for pending links, i.e. link information received for resources that have not yet been inserted on the database. If resource updates are found, the updated resources' information will be sent to the Control Centres that have requested the resources' virtual networks. The full process is depicted in the diagram of figure 5.26.

The status update thread blocks and waits for new resource messages on its message list. The access to the message list is controlled using a mutex. After placing a new message, a signal is sent to the status update thread that will wake up and process it.

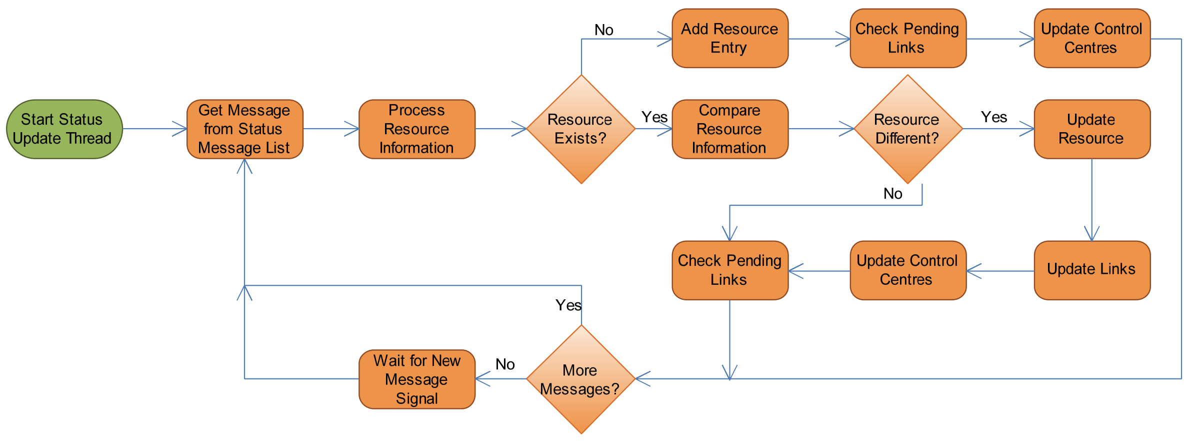


Figure 5.26: Manager - Status Update thread.

Command Send thread

When a thread wishes to send a command to an Agent, this is the thread to whom the request has to be made. It is responsible for sending the desired commands to the Agents. The available message types are described in table 5.4.

Just like the Status Update thread, the Command Send thread is also blocking and sleeps while waiting for new messages. When a new message arrives, it wakes up, processes it, and sends the command to the associated Agent. The thread's behaviour is illustrated in the diagram of figure 5.27.

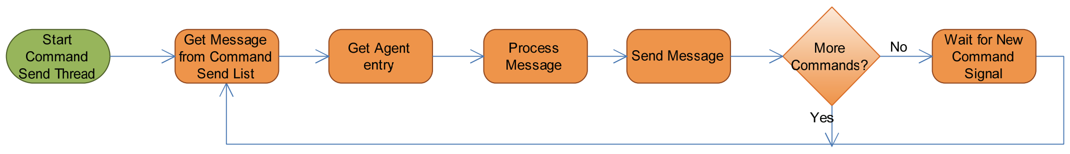

Start Command Send Thread
Get Message from Command Send List
Get Agent entry
Process Message
Send Message
More Commands?
No
Yes
Wait for New Command Signal

Figure 5.27: Manager - Command Send thread.

5.4.3 Virtual Network Mapping

After designing and configuring the virtual network using the Control Centre, the next step is to determine on which physical nodes the virtual nodes should be placed, and how the virtual links should be assigned to physical ones. This problem was presented on 2.3.5 and a solution was developed for it on chapter 4, section 4.7.2. The Manager, thus, has to implement this algorithm and enforce its result.

Receiving the Virtual Network XML

Subsection 5.3.3 ended with the Control Centre sending the fragmented XML message to the Manager, which has, therefore, to perform its reconstruction, as demonstrated in figure 5.28. The multiple messages that compose the virtual network description XML are assumed to be received in the proper order, since we are dealing with a TCP socket.

Message Type	Description
MA_MSG_RESOURCE_UPDATE	Request information about the specified resource.
MA_MSG_FULL_UPDATE	Request information about all the Agent's resources and links.
MA_MSG_ID_REPLY	Reply to an ID request with an unique ID.
MA_MSG_NODE_CREATE	Create the XML specified virtual node.
MA_MSG_BRIDGE_CREATE	Create the XML specified bridge and vlanned interfaces.
MA_MSG_LINK_DELETE	Request a link deletion.
MV_MSG_DEL_VNET	Request a virtual network deletion, i.e. resource and associated links deletion.
MV_MSG_START_VM	Start the specified virtual node.
MV_MSG_SHUTDOWN_VM	Shutdown the specified virtual node.
MV_MSG_REBOOT_VM	Reboot the specified virtual node.
MV_MSG_PAUSE_VM	Suspend the specified virtual node.
MV_MSG_UNPAUSE_VM	Resume the specified virtual node.
MV_MSG_DESTROY_VM	Delete the specified virtual node.
MV_MSG_SET_MEM_VM	Change the specified virtual node's RAM amount.

Table 5.4: Manager to Agent message types.

When receiving the first part of a message containing an XML, or a fragment of it, the Manager will extract the VNet identifier and add a temporary XML message structure to the temporary linked list on the connected Control Center's entry. When receiving subsequent parts of the same XML, identified by the part number and the VNet ID, the Manager will simply append them to the temporary XML. When the last piece is received, the message is removed from the temporary list and the next step, which is the mapping of the virtual network, will begin.

Virtual Network Mapping

The virtual network mapping function, observed in figure 5.29, follows the algorithm specified in 4.8. Firstly, the *map_vnet()* function parses the XML, using *libxml*'s functions, and converts it into a regular data structure, similar to the data structure of the existing virtual networks, i.e. a virtual network entry with a linked list of resources containing the related links.

The function proceeds with locating the physical network entry, from within the existing networks, and follows with a candidate selection for each of the virtual nodes. In this candidate selection, each virtual resource is compared with every physical resource in order to assess which ones satisfy the CPU, memory, HDD, and location constraints. For each virtual node, a linked list is filled with the possible physical candidates.

The algorithm continues with the determination of the physical links' stress. To that end, for every physical link, the amount of allocated bandwidth used by virtual links, is

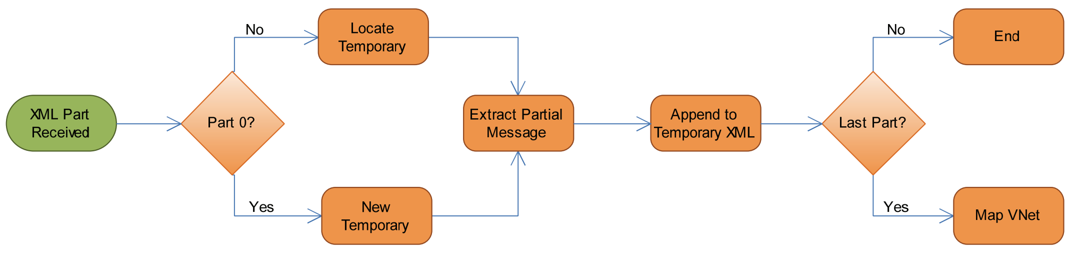


Figure 5.28: Manager - Receive XML Diagram.

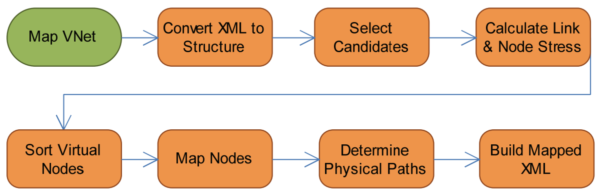


Figure 5.29: Manager - Map Virtual Network Diagram.

determined, thus determining the links' stress. Afterwards, the node stress determination begins and is done for every physical node.

Next, the virtual nodes are sorted considering the number of physical candidates, so that virtual nodes with less possible physical nodes are mapped first. The actual mapping function will then take place. For each virtual node's link, a Constrained Shortest Path First (CSPF) Dijkstra algorithm is run both for every physical candidate as well as for each link's destination virtual node's candidates, so that the node potential factor is determined. For each virtual node, the physical candidate with the lowest node potential, that has not been chosen yet, is selected as the embedding node.

After mapping every virtual node, the same CSPF algorithm is run in order to determine the final physical path for each virtual link. These virtual links may use different VLANs on each physical segment. The determination of the VLAN to be used is based on the information attained from both the origin and destination physical interfaces that compose the physical segment, and choses a non-utilized VLAN on both ends.

The relevant link information such as physical hops and utilized physical interfaces is added to the virtual links' structures.

Finally, by taking the virtual network's structure, which now has the mapping information on it, an XML containing the mapping information is produced and the mapping function terminates.

Virtual Network Committing

The mapped XML produced by the mapping algorithm contains information about every link and node of the virtual network and has, therefore, to be further breakdown on nodes' and bridges' configuration XMLs that will in turn be sent to the respective Agents.

Thus, for each virtual node, an XML file is produced, and for each virtual link, identified by a unique link ID, multiple XML bridge and interface configuration files will be produced, depending on the number of physical hops contained on a virtual link.

The virtual network creation is enforced by sending the XMLs to the appropriate Agents.

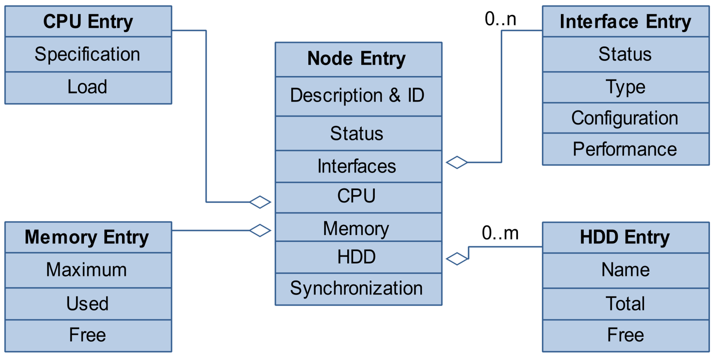
5.5 Virtual Network Agent

5.5.1 Main Databases and Structures

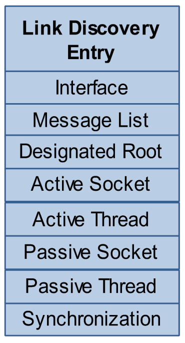
Main Databases

Just like the Manager, the Agents also use linked lists to store their data. There are two main databases: The *Main Resource List* stores the information of all local resources in a linked list, while the *Neighbour List* stores information about the Agent's neighbours, also in a linked list. The neighbour database encompasses both physical and virtual neighbour information.

The access to each database is controlled by individual mutexes.



(a) Node Entry.



(b) Link Discovery Entry.

Figure 5.30: Node and Link Discovery Entries.

As can be seen in figure 5.30(a), each node entry contains information about the resource's CPU, RAM, HDDs and interfaces. There is also additional information about the resource, such as its ID and status. Synchronization variables deal with the concurrent access to the resource, since multiple threads may try to access it at the same time.

Auxiliary Databases

Because the discovery mechanism requires a pair of threads for sending and receiving per interface, a linked list exists containing control data structures. These control structures contain relevant thread control variables and a message list for inter-thread communication, as well as the required data for the discovery algorithm, such as the Designated Root ID, as can be seen on figure 5.30(b).

5.5.2 Module Decomposition

Due to the multitude of tasks performed by the Agents, there are several threads running within the module. A global overview of these threads is shown in figure 5.31 and the Agent's start-up procedure is illustrated on the diagram of figure 5.32. The threads' functionalities will be described on the following paragraphs.

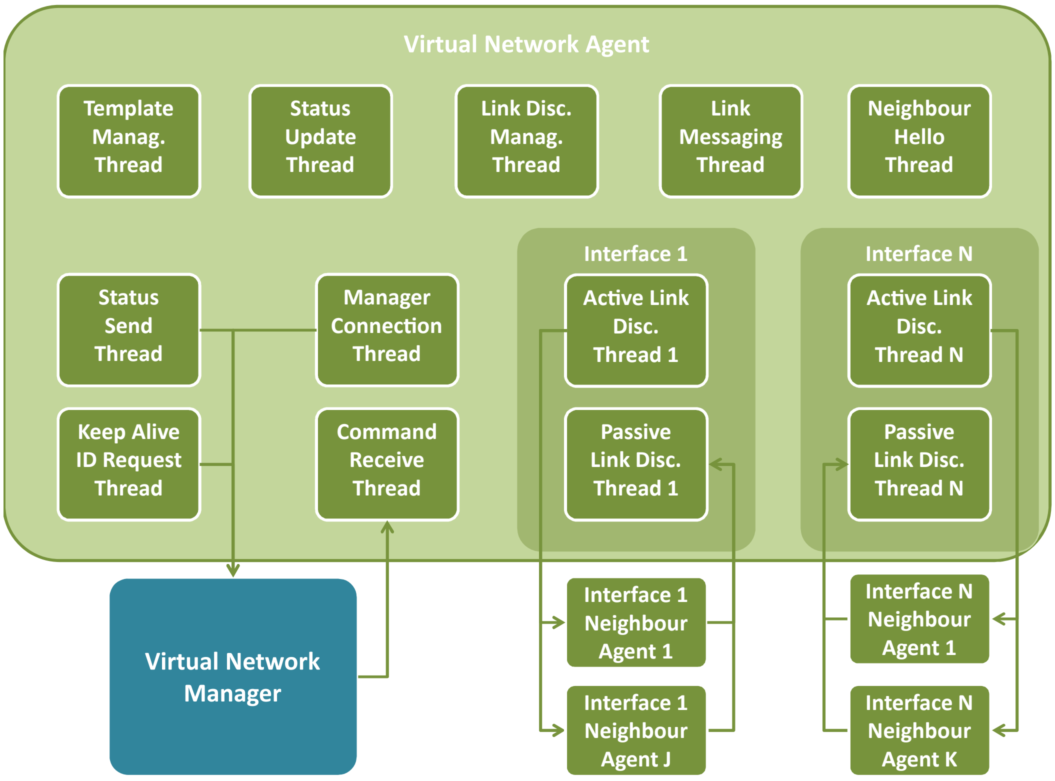

Virtual Network Agent
Template Manag. Thread
Status Update Thread
Link Disc. Manag. Thread
Link Messaging Thread
Neighbour Hello Thread
Status Send Thread
Manager Connection Thread
Keep Alive ID Request Thread
Command Receive Thread
Interface 1
Active Link Disc. Thread 1
Passive Link Disc. Thread 1
Interface N
Active Link Disc. Thread N
Passive Link Disc. Thread N
Virtual Network Manager
Interface 1 Neighbour Agent 1
Interface 1 Neighbour Agent J
Interface N Neighbour Agent 1
Interface N Neighbour Agent K

Figure 5.31: Agent Module

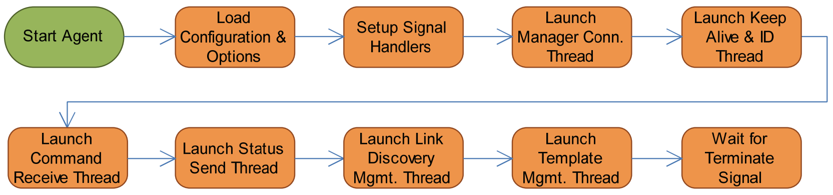

Start Agent
Load Configuration & Options
Setup Signal Handlers
Launch Manager Conn. Thread
Launch Keep Alive & ID Thread
Launch Command Receive Thread
Launch Status Send Thread
Launch Link Discovery Mgmt. Thread
Launch Template Mgmt. Thread
Wait for Terminate Signal

Figure 5.32: Agent - Start-Up diagram.

Manager Connection thread

This thread is in charge of establishing a connection to the Manager and inform the rest of the threads when a connection is established or lost. If there is no Manager connection, it will periodically try to connect until it is successful or the program terminates (figure 5.33).

Upon a successful connection, if the Agent does not have a valid ID, an ID request will be immediately sent in order to speed-up the Manager information update process.

Keep-alive and ID request thread

The Keep-alive and ID request thread periodically checks the Agent's ID (figure 5.34). If the ID is not valid, a request for a valid ID will be sent to the Manager; otherwise, a keep-alive message will be sent. The purpose of the keep-alive messages is to allow the

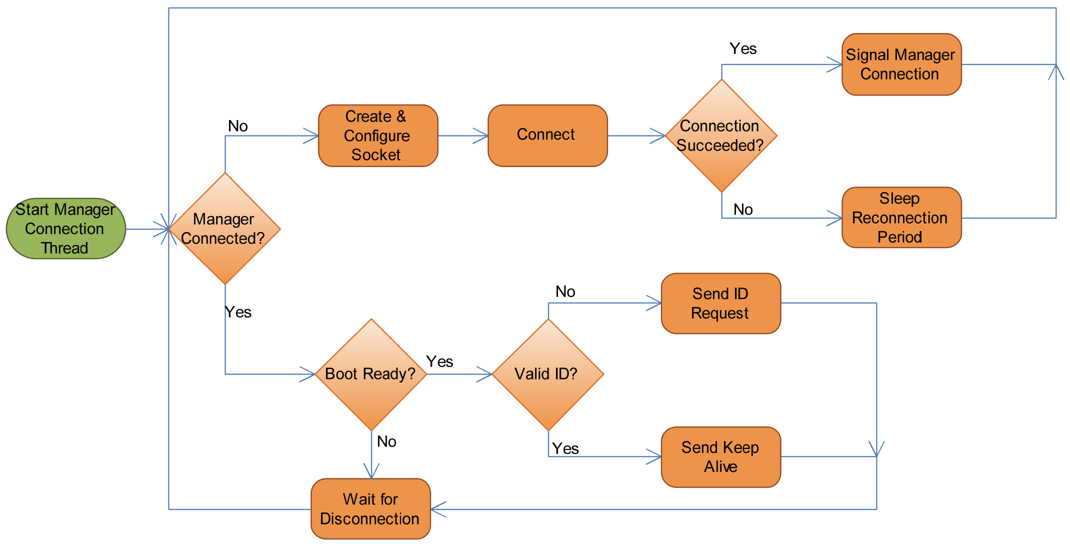


Figure 5.33: Agent - Manager Connection thread diagram.

Manager to identify situations where communication problems with the Agent exist.

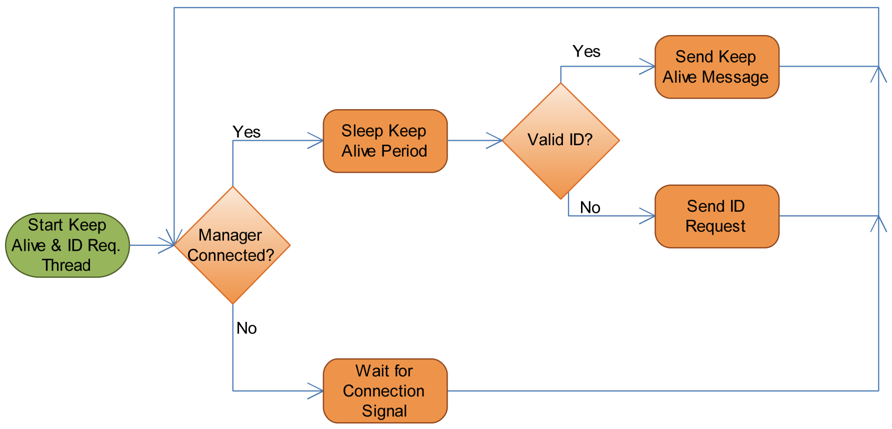


Figure 5.34: Agent - Keep Alive and ID Request thread diagram.

Template Management thread

The Template Management thread has two main functions: the first one is to build a pool of available virtual machine templates and make them ready-to-use, while the second one is to accept incoming virtual machine creation request and commit them. These functions are illustrated on the diagram of figure 5.35.

The Template Management thread waits for node request to arrive on the *Node Request List*. Upon receiving a request for a new virtual node, it checks whether the requested distribution exists or not. If it does, it launches a new thread responsible for node creation and replenishing of the utilized template image.

As a part of the node creation process, this thread will verify the newly created node status, check for virtual links and insert it in the main resource list.

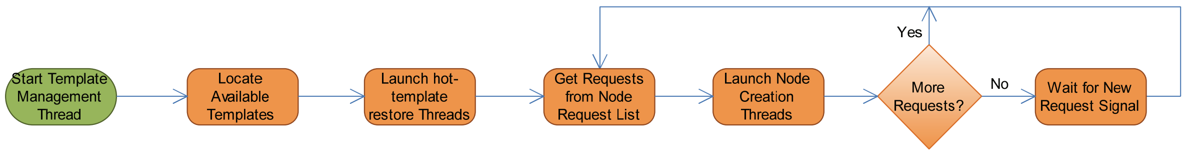

Start Template Management Thread
Locate Available Templates
Launch hot-template restore Threads
Get Requests from Node Request List
Launch Node Creation Threads
More Requests?
Yes
No
Wait for New Request Signal

Figure 5.35: Agent - Template thread diagram.

Status Update thread

This thread performs the resource data gathering and resource update checks. It runs periodically and checks for consistency in the resource database (figure 5.36).

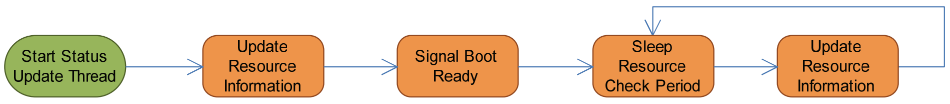

Start Status Update Thread
Update Resource Information
Signal Boot Ready
Sleep Resource Check Period
Update Resource Information

Figure 5.36: Agent - Status Update thread diagram.

The thread relies on a function that checks for local resource updates (figure 5.37). If a resource is created, deleted or modified, it is this thread's function to identify the updates, keep the database updated, and signal threads that may be interested in certain resource changes. It gathers CPU, RAM, HDD, state, and interface data.

Interface changes on the physical resource will trigger a re-evaluation of the link discovery threads. If a change is detected on virtual interfaces' associated bridges, a verification of the consistency in the virtual links databases will take place, and appropriate measures are taken if the resource needs to be advertised.

Regardless of the update detected, the thread will inform the Manager of a resource update, by sending a message to the Status Send thread.

The deletion of no longer existing virtual nodes is not as simple as removing the entry from the local database. The Manager must be warned about the resource deletion and so must the physical neighbours, so that they can remove neighbour entries related to the deleted resource. The deletion process is shown in figure 5.38.

Status Send thread

The outgoing Manager communications go through this thread; it receives message requests from other threads and informs the Manager accordingly.

The requests are made by placing a request structure on the *Status List* and signalling the thread (figure 5.39).

The message types sent to the Manager by this thread are defined on table 5.3.

Command Receive thread

As opposed to the Status Send thread, the Command Receive thread receives the Manager's messages and takes the proper action to execute the requested commands.

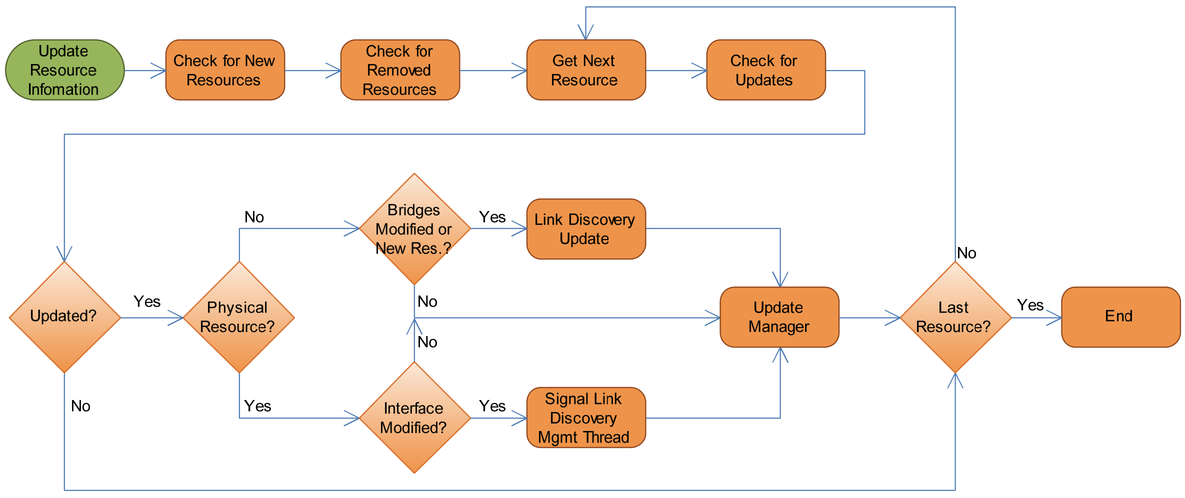


Figure 5.37: Agent - Update Resource Information diagram.

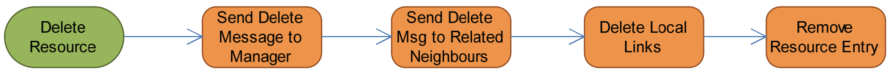

Delete Resource
Send Delete Message to Manager
Send Delete Msg to Related Neighbours
Delete Local Links
Remove Resource Entry

Figure 5.38: Agent - Delete Resource diagram.

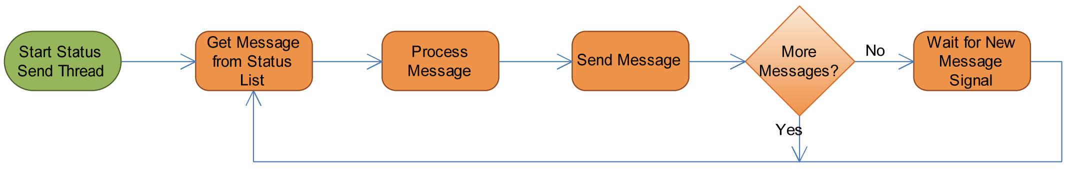


Figure 5.39: Agent - Status Send thread diagram.

It is a blocking thread; therefore, it is only active when there is data to be read from the socket.

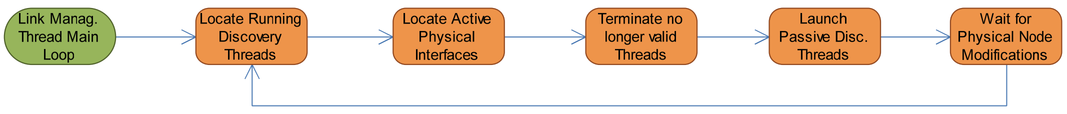
The possible receive commands were previously described on table 5.4.

Link Discovery Management thread

The distributed neighbour link discovery process is managed by this thread. It launches both a passive and active discovery thread per active physical interface; the passive thread is in charge of receiving multicast messages, while the active thread is responsible for sending them. Figures 5.40 and 5.41 illustrate this thread's main procedures.

The Agents implement the algorithm described on the previous chapter, in section 4.8. During runtime, several modifications may happen regarding physical and virtual interface configuration, state, and addition or removal of virtual resources. Due to the highly dynamic nature of production networks, the implemented algorithm must be ready to withstand all these possible events and keep the topology information consistent and updated.

In order for the discovery mechanism to begin, the Agent must have completed the boot process, i.e. all the information about itself and its resources must have been acquired. When this information gathering is complete and the Agent attains a valid ID, the link discovery management thread, that blocks at start-up waiting for the *boot ready* and *valid ID* signals, resumes its execution.

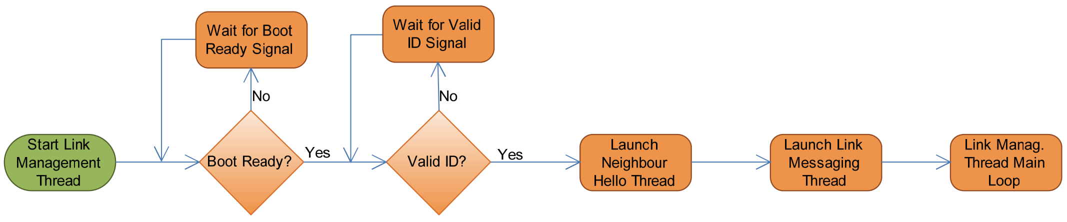


Link Manag. Thread Main Loop
Locate Running Discovery Threads
Locate Active Physical Interfaces
Terminate no longer valid Threads
Launch Passive Disc. Threads
Wait for Physical Node Modifications

Figure 5.40: Agent - Link Management thread diagram.

The Agent will firstly locate the physical node's database entry and afterwards launch two important threads: the *neighbour_hello* and the *link_messaging* threads. The first one periodically sends hello beacons through every interface, while the second one is a proxy thread that enables the sending of messages to each interface's active discovery threads. They will both be described next.

The main loop of the link management thread, figure 5.41, begins by identifying each active and running physical interface, and creates a passive discovery thread per each identified interface that will be in charge of subsequently launching the corresponding active thread. The passive link discovery thread is in charge of receiving multicast messages, while the active link discovery thread is responsible for sending them.



Start Link Management Thread
Wait for Boot Ready Signal
No
Boot Ready?
Yes
Wait for Valid ID Signal
No
Valid ID?
Yes
Launch Neighbour Hello Thread
Launch Link Messaging Thread
Link Manag. Thread Main Loop

Figure 5.41: Agent - Link Management thread main loop diagram.

After every passive thread is launched, the link management thread will block waiting for modifications relating to the physical node's interface configuration. If any modification is detected, a check will be made in order to determine which discovery threads should be terminated and if any should be created. If, for example, an interface goes down, the associated active and passive threads will be shut down but the data structures will remain intact and ready for the thread restart if the interface goes back up.

Passive Link Discovery Thread

When the passive link discovery thread begins (figure 5.42), the first step is to properly configure a socket to receive multicast messages. The socket creation follows a standard procedure: initially, a socket is created and set to enable port reuse, so that other sockets may receive on the same port number; afterwards, a socket bind to the multicast address is made and, subsequently, the interface is registered on the multicast group.

The resulting socket identifier may now be used to receive messages from the multicast group. Since every multicast message is received on all interfaces, each passive thread performs a message filtering function to determine if the message effectively arrived on its interface. This check is based on the interface's and message's source IP information. If a message is received from a different network than the interface's one, it is discarded.

After creating and configuring the multicast socket, the passive link discovery thread will launch the active link discovery thread, responsible for sending multicast messages.

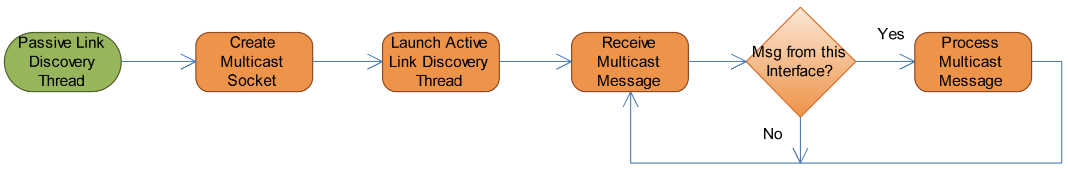

Passive Link Discovery Thread
Create Multicast Socket
Launch Active Link Discovery Thread
Receive Multicast Message
Msg from this Interface?
Yes
Process Multicast Message
No

Figure 5.42: Agent – Passive Link Discovery thread diagram.

Active Link Discovery Thread

As previously explained, this will be the thread in charge of sending multicast messages. At start-up, it creates and configures a socket with the desired multicast address, that may then be used to send multicast messages.

The thread waits for new messages on the link discovery entry's message list. When a message is received, it will process and send the message via the multicast socket. The described behaviour may be analysed in the diagram of figure 5.43.

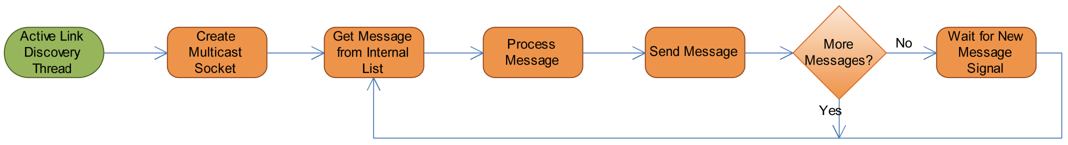

Active Link Discovery Thread
Create Multicast Socket
Get Message from Internal List
Process Message
Send Message
More Messages?
No
Wait for New Message Signal
Yes

Figure 5.43: Agent – Active Link Discovery thread diagram.

Neighbour Hello thread

The main duty of this thread is to periodically send Hello messages to the Agents' neighbours. This thread also serves the purpose of identifying "expired" neighbours, i.e. the neighbours that have not sent any message in a previously pre-determined amount of time.

Expired physical neighbours and related virtual neighbours will be removed from the local neighbour database (figure 5.44).

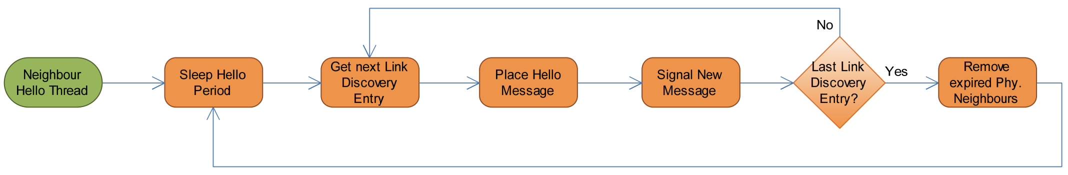

Neighbour Hello Thread
Sleep Hello Period
Get next Link Discovery Entry
Place Hello Message
Signal New Message
Last Link Discovery Entry?
No
Yes
Remove expired Phy. Neighbours

Figure 5.44: Agent – Neighbour Hello thread diagram.

Link Messaging thread

This thread works like a proxy in the way that it provides the other threads with an interface for easily announcing new or deleted resources.

If a local resource is added or removed, this thread will be notified and act accordingly. If a new resource is added, it will check for virtual links, and notifying the relevant link discovery threads of a new resource. If, on the other hand, a resource is removed, it will locate the interfaces previously utilized by the removed resource and send multicast delete messages to the physical neighbours that will then act accordingly.

5.5.3 Resource Data Gathering

The resource data gathering mechanism runs periodically and resorts on several system tools.

The libvirt's *virDomainGetInfo* and *virNodeGetInfo* functions supply information regarding both virtual and physical domain state, allocated RAM memory, number of CPU cores and respective frequency.

Detailed CPU information is gathered from the */proc/cpuinfo* file, and CPU load is computed between successive resource data checks.

The determination of CPU load is different for virtual and physical resources. The physical node's CPU load calculation is performed by gathering the CPU's processing time from the glibtop's *glibtop_get_cpu* call. The difference in total CPU processing time is divided by the total actual bygone amount of time, and a load amount in per cent is determined. The maximum CPU load equals the amount of CPU cores times 100%, i.e. for 8 cores, the maximum load would be 800%.

Determining the amount of free RAM is simpler. For virtual resources, it is simply the difference between the allocated maximum amount of RAM and the currently in use RAM. This info is provided by the *virDomainGetInfo* call. For the physical resource, glibtop provides the *glibtop_get_mem* call, and the total amount of available RAM is considered to be the sum of the free, cached and buffered memory.

Interface configuration information is more difficult to gather. When the number of bridges and interfaces on the system begins to grow, the time required to perform calls to *ifconfig* or *brctl* will increase. Taking into consideration that gathering interface data for a single resource may require several *ifconfig* and *brctl* calls, it is easy to verify that performing multiple successive calls is an unviable option; it presents scalability issues.

In order to reduce the performance penalty of *ifconfig*, the *glibtop* network calls were used. These calls conveniently provide all the interfaces' configuration and statistics on a data structure, thus avoiding the need to perform multiple calls for a single interface.

On the other hand, although *brctl*'s code is public, the efforts made to integrate it with the Agent's software failed, and no library was found that could provide the same information. Hence, a mutex-locked buffer cache was created for it in order to speed up successive calls. A data structure was created to hold the bridges' and associated interfaces' info using linked lists, so that accessing the *brctl* info on this list presents a reduced performance penalty when comparing with accessing it directly through the console command.

This *brctl* cache provides a mean for explicitly requiring a cache update, so that when a function absolutely requires up-to-date bridge information, it may request it.

Detailed data gathering of virtual interfaces' configuration presents some issues. Since the Agent has no means of knowing what is the interface configuration inside the running virtual machines, it has to rely on the XML file saved upon creating the virtual machine; therefore, it only knows the default interface configuration.

The HDD information is attained from the Linux's *df -h* command for the physical resources and from the size of the image file for virtual resources.

5.5.4 Virtual Network Creation

The virtual network creation process may be subdivided into two main phases: the first one is related to the establishment of the virtual links and bridges, which are required for proper virtual machine start-up, while the second phase is the creation and configuration of the virtual nodes.

Bridge & Interface Configuration

All the physical nodes belonging to a virtual link will have a bridge named after the virtual network and the link ID. If, for example, a virtual network exists called *Alpha*, and that virtual network has a virtual link with an ID equal to 10, the physical nodes hosting that virtual link will each have a bridge named *VNetAlpha.010*.

Each virtual link may be composed by several physical links utilizing different VLANs. For every physical link, a non-utilized VLAN number is identified and utilized, resulting in physical sub-interfaces that use the selected VLAN. The previously described bridges will bring the different links, that resort to VLANs, together. The bridge creation process is represented in figure 5.45.

The bridges on physical hops will simple connect two different physical sub-interfaces, while the ones on the edge nodes, i.e. the bridges located on the same physical nodes as the virtual nodes, will bridge the physical interfaces sub-interfaces and the virtual nodes' interfaces associated with that virtual link.

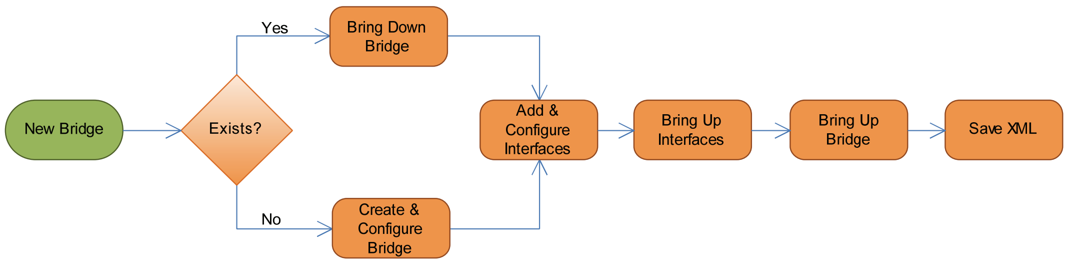


Figure 5.45: Agent - Create new Bridge.

An XML message containing the bridge, interface and QoS information for a given virtual link is sent to the relevant Agent, which will in turn save it on a predefined folder, process it, configure the interfaces, create the bridge if it does not exist already, and associate the interfaces with the bridge.

The bridge configuration process is executed before the nodes' creation, so that when the nodes are created and their status updated, everything will be in place.

Virtual Node Creation

Upon virtual network mapping, each Agent with a resource mapped on it will receive an XML message containing the virtual node's information that will be saved on a predefined folder for future access.

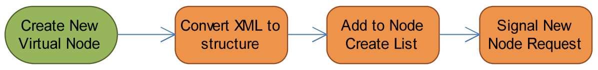


Figure 5.46: Agent - Create Virtual Node Diagram.

The Agent processes the XML message and proceeds with converting it to a resource data structure that will be sent to the Template Management thread for creation (figure 5.46). After proper validation, the node creation process will proceed, according to the diagram of figure 5.47.

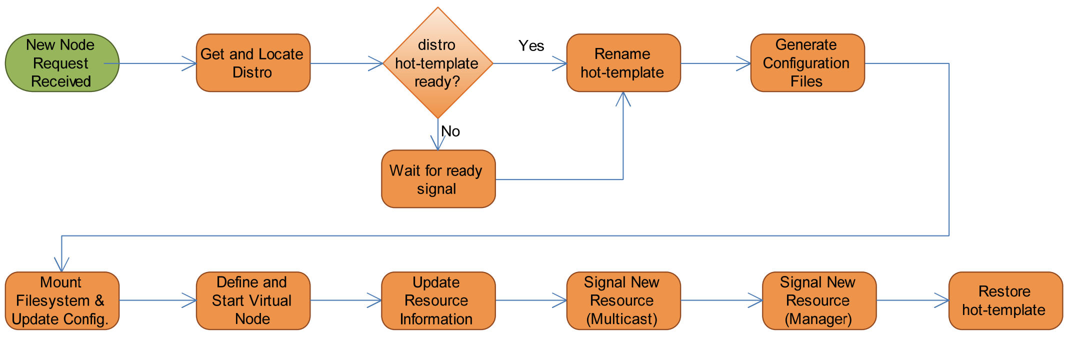

New Node Request Received
Get and Locate Distro
distro hot-template ready?
Yes
No
Rename hot-template
Generate Configuration Files
Wait for ready signal
Mount Filesystem & Update Config.
Define and Start Virtual Node
Update Resource Information
Signal New Resource (Multicast)
Signal New Resource (Manager)
Restore hot-template

Figure 5.47: Agent - Virtual Node Request Diagram.

Firstly, the hot template image folder and XML file will be renamed to match the requested node's name. Afterwards, the template XML file will be modified according to the requested node's data.

The configuration of the virtual interfaces and change of the virtual machines' hostname requires the mounting of the virtual node's file system, and modification of some configuration files, which will depend on whether the environment is Debian of Fedora based.

After properly unmounting the file system, a call to the libvirt's *virDomainDefineXML* function will define the virtual node, and a final call to *virDomainCreate* will effectively create and start it.

Although the virtual machine will take a few seconds to be ready for use, everything is properly configured; therefore the node's information is gathered, added to the main resource list, the Manager is informed about this new resource, and a verification is made to check for the existence of virtual links.

5.6 Conclusion

This chapter presented the implementation of the three that compose the virtualization platform: the Control Centre, the Manager, and the Agent.

The Virtual Network Control Centre was the first module analysed in depth. Since this is the front-end for the software suite, the implemented mechanisms mainly deal with presenting the acquired data to the user, feeding back commands to the Manager, and subsequently to the Agents.

Dynamic virtual network drawing and designing functionalities were implemented. They aim to facilitate the creation of virtual networks. Comprehensive configuration options were provided for configuring both the resources and the links, with several details available for configuration and specification.

Commodity functionalities, such as saving or loading XML files, were made available and so were other “simple” but useful options, such as deleting an entire virtual network at once, or having the ability to monitor different networks at the same time simply by requesting them and opening a new tab. One other interesting feature is the ability to monitor almost in real-time the state and load of both physical and virtual resources. This real-time approach enables network administrators to quickly identify potential issues in the network, for example.

The user interface is simple to use and was made intuitive, in part due to the drag and drop mechanism, simple menus, and the coolbar, that provides a quick short-cut to access the mainly used tools when designing a virtual network.

Next, the underlying mechanisms and threads of the Virtual Network Manager were explored. Besides providing aggregation functionalities for building the networks’ topologies and keeping the Control Centres up-to-date, the Manager is in charge of implementing the essential virtual network embedding algorithm and to send the resulting mapping to the chosen physical nodes. It resembles a gateway in the way that it provides a separation between the user interface, the Control Center, and the Agents running on the substrate network. Its implementation makes the simultaneous operation of multiple users on the substrate network possible, since it accepts an undetermined amount of Control Centres connections.

Finally, the Agent module was analysed. This module is in charge of running on each physical node, and plays an important role, since it is responsible for gathering and detecting changes in resource information, creating and configuring the virtual resources, interfaces and bridges. One other critical feature is the implementation of the distributed discovery algorithm, that provides the fundamental network topology data.

Chapter 6

Tests & Results

6.1 Introduction

The purpose of this chapter is to assess the performance of the developed virtualization platform, regarding the previously developed mechanisms.

The chapter will begin, in section 6.2, by introducing the utilized testbed, resorting to the developed graphical interface to display it. Next, a base virtual network is specified that is used as a reference on the experimental testing performed.

In section 6.3, the performance of the Agents will be evaluated, with respect to the scaling of the time required to gather local resource information. The chapter will proceed in section 6.4, where the time required for performing virtual and physical network topology discovery will be evaluated, considering different start-up situations.

Afterwards, in section 6.5, the performance of the mapping algorithm and of the time required to create new virtual networks, will be examined.

The chapter ends on section 6.6, with a global overview and discussion of the attained results.

6.2 Testbed Description & General Assumptions

The testbed is composed of 6 physical nodes and is connected according to figure 6.1, attained from the developed virtualization platform.

The main specifications of the substrate nodes may be found in table 6.1.

In the following tests, the Manager was running on a separate physical machine, directly connected to the physical node Mary. The created virtual networks were always a replica of the underlying physical network. The virtual nodes were configured with 1 CPU, 64MB of RAM, 1GB of HDD and 1Mbps links. This virtual network is depicted in figure 6.2, and shall uniformly load every physical node and link and be used as a reference.

During the tests, the virtual nodes were idle and so were the physical nodes and links, no other activity or task was being run on the testbed.

The maximum amount of created virtual networks was 40, which corresponds to 40 virtual nodes in each physical node. This limitation is mainly due to the node with the least amount of memory, Gabrielle that would present instability issues with 45 virtual nodes.

The results presented on the following sections always assume a 95% confidence interval.

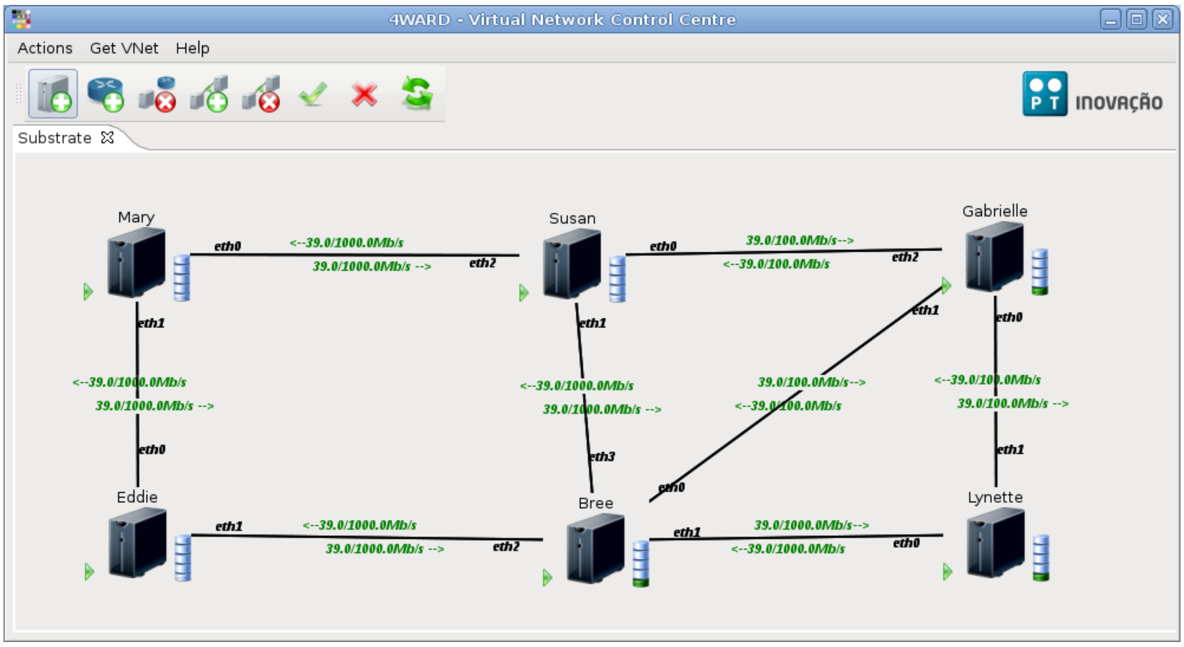


Figure 6.1: Testbed Network.

Node	Susan	Lynette	Gabrielle	Bree	Eddie	Mary
CPU Model	Intel PentiumD 950	Intel PentiumD 950	Intel Core 2 Duo E6400	Intel Xeon E3110	Intel Xeon X3220	Intel Xeon X3330
CPU Freq.	3.40GHz	3.40GHz	2.13GHz	3.00 GHz	2.40GHz	2.66GHz
CPU Cores	2	2	2	2	4	4
CPU Threads	4	4	2	2	4	4
RAM Amount	6GB	6GB	4GB	6GB	6GB	6GB
RAM Freq.	533MHz DDR2	667MHz DDR2	533MHz DDR2	667MHz DDR2	667MHz DDR2	667MHz DDR2

Table 6.1: Testbed specification.

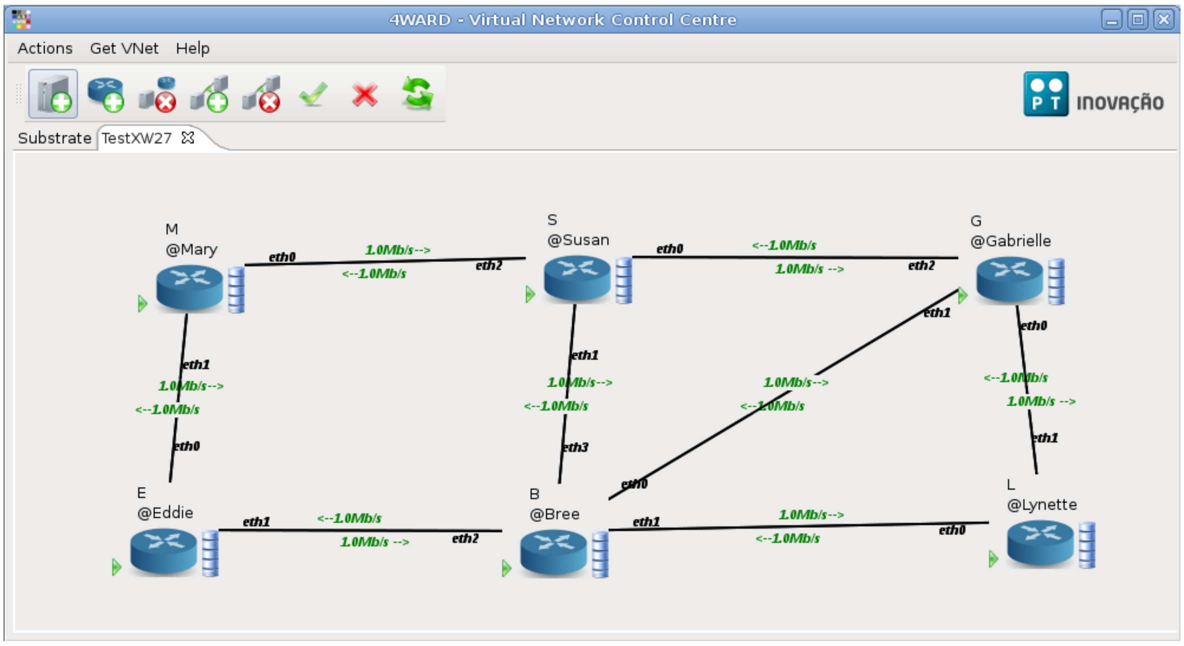


Figure 6.2: Reference Virtual Network.

6.3 Data Gathering

Data gathering is a very important feature if an updated view of the existing networks' status and characteristics is intended. Its performance may be a critical factor: if the data gathering procedures take too long, the reaction to failures or other events may be delayed. This delay may cause severe consequences on the network's performance, as the network administrator may not realize that there is a problem until it is too late.

In order to assess the cold boot and status update time, virtual networks with the same number of nodes as the substrate network were created, as previously described.

6.3.1 Cold Boot

Cold boot is described as the time it takes for each physical node to fully discover and update the information about itself and its virtual nodes, i.e. the time required since the Agents start up until they are ready to perform discovery tasks and send a full update to the Manager.

This test intends to demonstrated the dependency between this start up time, the number of running virtual machines and the capability of the physical nodes.

Methodology

In order to assess the cold boot time, the time difference between the Agent start-up and the end of the first status update call was measured.

This procedure was repeated 10 times for every considered amount of virtual networks.

Results & Discussion

Figure 6.3 exhibits the time required to boot with the increase in the number of existing virtual machines.

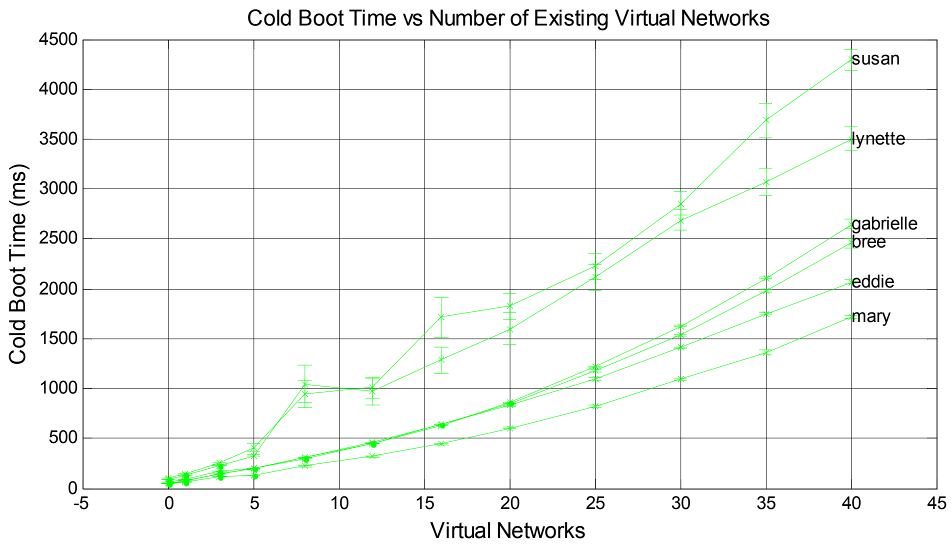


Figure 6.3: Agent Cold boot results.

It is clear that the substrate nodes have very different capabilities, and that the boot time is heavily dependent on the CPU processing power. The physical nodes using the old Intel NetBurst architecture, i.e. Susan and Lynette, perform worse than the ones relying on the more recent Intel Core 2 architecture. The performance disparity between Susan and Lynette seems to be associated with the difference in RAM speed.

6.4 Network Discovery

6.4.1 Cold Network Discovery

The time required since every Agent has booted up until the full physical and virtual network's topologies have been discovered is designated the cold virtual network discovery time.

When every Agent boots up and the Manager is disconnected, the discovery mechanisms are frozen, waiting for a valid ID from the Manager. When the Manager is brought up and quickly allocates an ID to every Agent, the discovery mechanism takes place and every Agent exchanges link discovery messages.

Upon discovering virtual links, the Manager will be updated and will build the virtual and physical networks' topologies. Therefore, the cold virtual network discovery time can also be tough of as the time required for the Manager to have an updated global view since its start-up, considering the situation where no Agent has a valid ID, and thus, the link discovery process has not yet begun.

Methodology

For every virtual network created, the cold discovery time was measured 10 times. In every time, the Manager and Agents were firstly shutdown. Afterwards, every Agent was brought up. When every Agent had finished its cold boot, the Manager was started and began waiting until a predetermined amount of resources and links were received, depending on the number of the currently running virtual networks.

The elapsed time reflects, not only the time required for the Agents to discover its physical and virtual neighbours, but also the time required for transmitting the information to the Manager, and the time required by the Manager to process this information and build the network topologies.

Results & Discussion

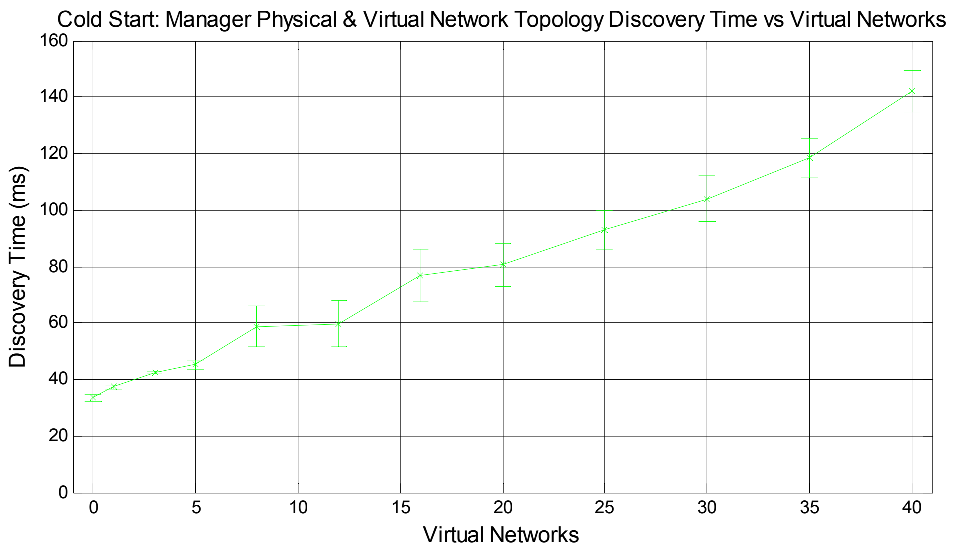


Figure 6.4: Cold network discovery results.

Evaluating figure 6.4, it is possible to notice that the time required for discovery seems to follow a linear trend. Taking into account the results attained from the simulation results of the discovery algorithm, this trend was to be expected.

6.4.2 Hot Network Discovery

This test is similar to the previous one, with the exception that in this case the Agents already have a valid ID and have already exchanged discovery messages with each other; thus, the hot virtual network discovery time reflects the time required for the Agents to send resource and link messages to the Manager and its processing time. In order to provide a comparison base, the centralized network discovery algorithm was also run.

Methodology

In this test, every Agent had already booted up and been given an ID from the Manager; thus, the network discovery had been completed.

The Manager was programmed to terminate its execution upon receiving and processing the expected amount of nodes and links, which was variable according to the number of virtual networks running at a given instant. It had two operation modes: the first one utilized a centralized topology discovery algorithm, while the second one utilized the link information sent by the Agents to the Manager to build the topologies.

A script was created that executed the Manager 100 times in a successive way, both for the centralized and the distributed algorithms, with a 1 second delay between Manager termination and restart.

With the increase of the number of virtual networks, one can evaluate the scaling of the discovery times with the number of existing virtual networks.

Results & Discussion

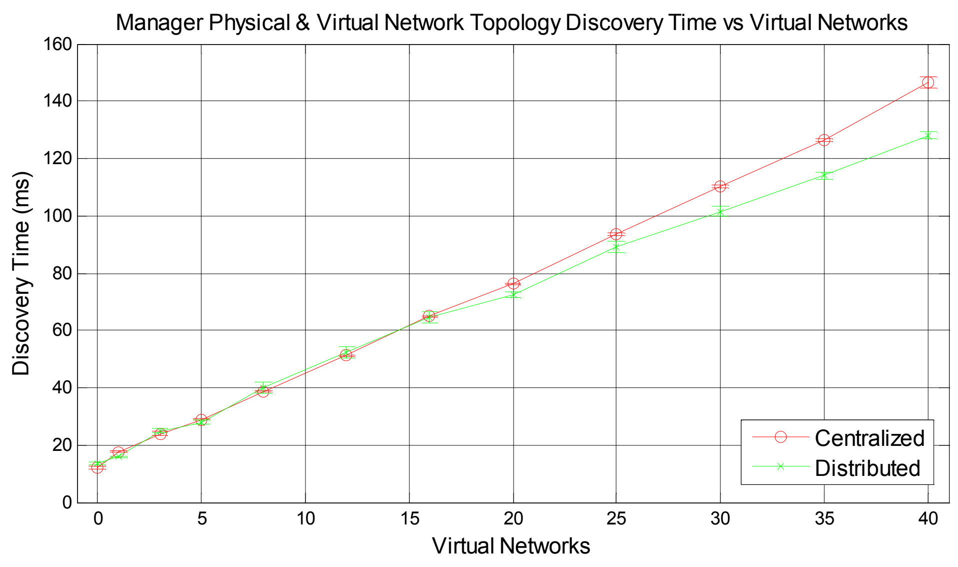


Figure 6.5: Distributed vs. Centralized network discovery results.

Comparing the achieved results, displayed in figure 6.5, with the discovery times of the previous subsection, it is possible to state that the discovery process is faster. This is to be expected since these measurements simple incorporate the time required for the Agents to send their neighbour information to the Manager, and the time it takes for the Manager to process or aggregate this information.

By comparing the results of the centralized and distributed approach, one can state that, as the number of virtual networks begins to grow, the distributed approach provides lower discovery times than the centralized one. For 40 virtual networks, the time difference is 20ms, or about 17%.

This is a very important conclusion, as it already shows that distributed approaches need to be supported in future and complex networks.

6.5 Virtual Network Mapping & Creation

6.5.1 Virtual Network Mapping

One of the main reasons of having opted for a heurist approach when designing a mapping algorithm on section 4.7.2 was due to the fact that these heuristic algorithms tend to impose a lighter load on the computing resources and are, thus, faster than the optimal algorithms.

In this section, the performance of the proposed algorithm will be evaluated with the goal of assessing if it presents a viable option in production environments, i.e. if it is fast enough. In spite of the small-scale testbed, some insight should also be gained about the scaling of the algorithm with the increase in the number of existing virtual networks.

Methodology

In order to assess the mapping times, 40 virtual networks, like the ones specified in 6.2 were created, one at the time. The time required for the Manager to process the received unmapped XML and return a mapped one was measured. The tests were repeated 3 times.

Results & Discussion

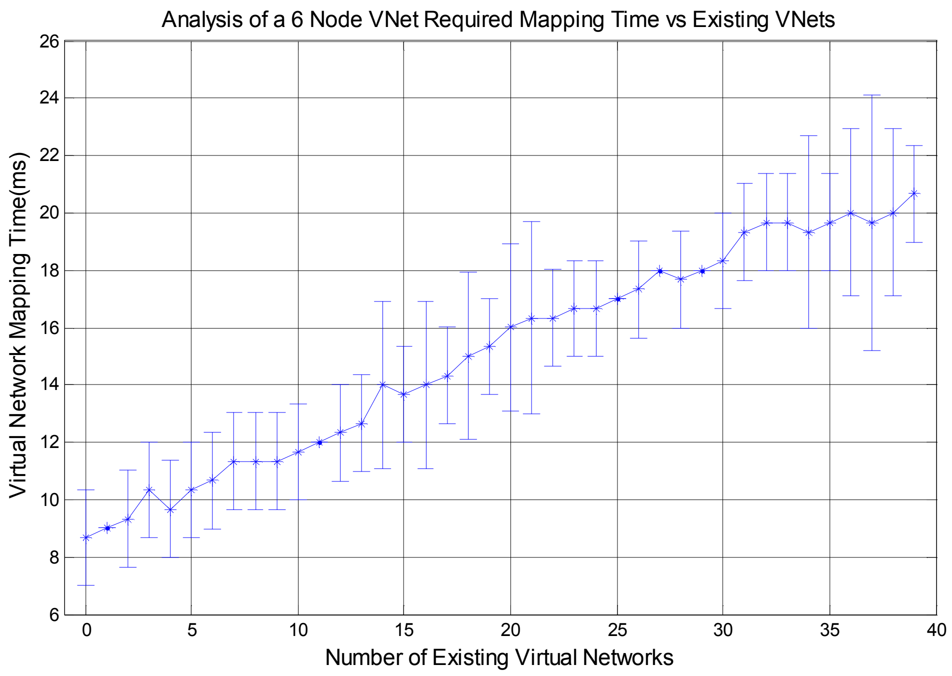


Figure 6.6: Virtual Network Mapping results.

The time required to perform the mapping is shown to increase with the number of existing virtual networks (figure 6.6). Since the mapping procedure only depends on the virtual network to be embedded and on the physical network, it would be expected that the mapping times remained constant.

This is not the case. In order to understand the increase in the required mapping time, one must take into consideration that when performing the mapping, the Manager needs to update the physical links' load, and therefore needs to access each existing virtual network. Thus, for each additional virtual network, the Manager will need more time to calculate the physical links' stress. This increment in needed time is revealed in the attained results, that clearly show a linear scaling with the number of existing virtual networks.

Regarding the absolute mapping times, they remain in the order of low tens of millisecond, which is very good and can be considered real-time. One must, however, take into consideration the lack of complexity in both the embedded and physical networks, that makes the mapping process easier. The considerable deviations on the measured mapping times are probably due to the Manager's need to lock the different resources' mutexes, while performing the mapping.

6.5.2 Virtual Network Creation

Virtual network creation should be as fast as possible, as an operator must be able to respond promptly to every embedding request.

In order to evaluate the time required for creating a virtual network on the available testbed and its scaling with the amount of previously existing virtual networks, several tests were performed.

Methodology

Virtual network creation tests were performed considering that a given amount of virtual networks already existed on the testbed. The amount of previously existing virtual networks was varied between 0, i.e. without virtual networks, and 39.

For each considered point, a virtual network as created and deleted 10 times and the time required for creation was recorded. The created virtual networks are the same as the ones previously specified in 6.2.

The considered creation time encompasses the time required for the Manager to split the mapped XML and send the different command messages to the Agents, as well as the time required for the Agents to report back with updated information about the created resources and links, i.e. the time required to perform the discovery of the created virtual network. The Manager was in charge of measuring these creation times.

Results & Discussion

The results accomplished regarding virtual network creation times, shown in figure 6.7, seem to follow a linear trend with the increase in the amount of existing virtual networks.

It is worth noting that the total creation time, encompassing both node creation and subsequent topology discovery, only depends on the slowest physical node, from the ones chosen to have a virtual node embedded. Considering the physical node's performance estimates attained in section 6.3, one can see that the slowest node, Susan, is about three times slower than the fastest node, Mary.

The demonstrated increase in discovery times is due to the increase in time required to gather resource information. It is worth noting that, when the virtual node is created, the used virtual machine template will be regenerated, imposing a severe strain on the physical node's slow HDD, thus further slowing down the data gathering and subsequent discovery process.

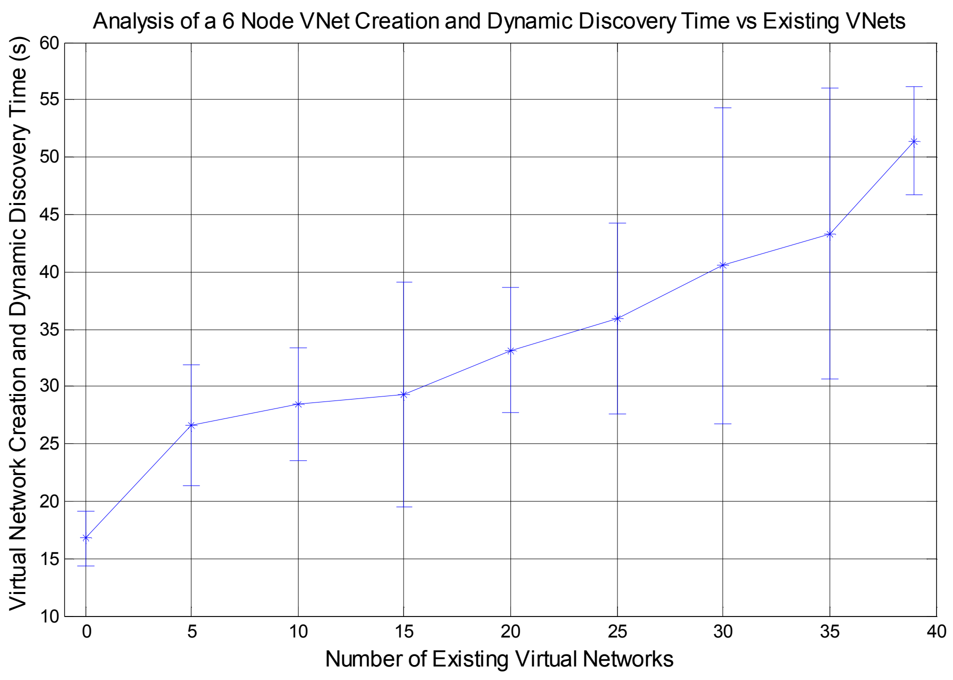


Figure 6.7: Virtual Network Creation results.

Significant deviations were attained when measuring the creation time. Because these measurements depend on every physical node, their respective discovery threads, mutex locks, time required for gathering resource information and also the performance of the hypercalls, large variations were verified.

6.6 Conclusions

This chapter presented the main results achieved with the virtualization platform, regarding the time required for the Agents boot, and the time needed for discovering, creating and mapping virtual networks.

Regarding one of the potential bottlenecks of the Agents, data gathering, it was shown that even for 40 virtual networks, the required amount of time remains within acceptable boundaries, with a worst-case scenario of about 4 seconds.

The implemented discovery algorithm was shown to provide discovery times that should be barely perceptible for the human user, under 200ms. The distributed discovery algorithm performed better than the centralized one with 20 or more existing virtual networks, proving its better scalability.

The final tests, concerning virtual network creation and mapping times, also provided good results. The mapping algorithm was able to quickly map the virtual networks, with mapping results in the order of tens of millisecond. The achieved virtual network creation times, although significantly larger, also remained within acceptable boundaries, having taken less than 1 minute to create a 6 node virtual network even with the substrate network already accommodating 39 virtual networks.

Overall, the attained results demonstrate that the performance and scalability of the tool is very good for about 40 virtual networks, the testing limit.

Chapter 7

Conclusions

7.1 Final Conclusion

Considering the goals set to be achieved, the first conclusion is that they were indeed met.

Initially, the software's constraints, guidelines, and desired features were provided. It served the purpose of being a reference along the design and implementation stages. The platform's requirements, communication semantics, constraints, and the use-cases were established.

The design stage refined the software's architecture (chapter 4), further defined the features, analysed potential issues and presented solutions for virtual network embedding and discovery.

Virtual network discovery algorithms, despite being fundamental for future virtualized networks, have not been a research target in the past few years. In order to fill the existing research gap, a distributed approach for virtual network discovery was proposed, based on concepts from overlay networks, bridging, and routing protocols. An extended analysis and description of the developed algorithm was done. In addition, simulation tests were performed that confirmed the algorithm's feasibility and performance.

The results achieved proved the algorithm's low overhead and scalability properties, concerning both the number of exchanged messages and required simulation cycles. In fact, for physical networks with 500 nodes, the algorithm was able to provide a message overhead that was more than three orders of magnitude lower than flooding-based algorithms performing the same task.

With respect to the scalability tests, with an increasing amount of virtual networks, the tests revealed a linear behaviour, with a much lower overhead penalty when compared to the said flooding mechanisms. These results further clarified the scalability properties of the proposed discovery algorithm.

The experimental virtual network discovery tests have validated the previously accomplished simulation results. It was shown that the discovery algorithm behaves linearly with the increase in the number of existing virtual networks. Even more relevant was the comparison between the distributed algorithm and a centralized approach. This comparison, revealed that performance advantages can be attained with the distributed algorithm when the number of virtual networks to be discovered starts to increase. It is expected that with an additional increase in the complexity of the virtual networks, the performance advantage of the distributed approach will be even more significant. Thus, distributed algorithms will have to be supported on future network virtualization platforms.

Virtual network mapping algorithms, on the other hand, have been studied by several authors, and some solutions have been proposed. Nonetheless, several issues were found that had to be addressed by the developed algorithm, such as the heterogeneity of both physical and virtual resources.

The performance of the proposed mapping algorithm was also tested on a simulation environment. The simulation results showed the behaviour of the mapping algorithm on different scenarios, which considered heterogeneous specifications for physical and virtual networks, as well as virtual networks with different dimensions.

Based on the attained results, it was possible to assess the scaling of the number of accepted virtual networks with their size, the impact of the embedded virtual networks' size on the load distribution on both physical nodes and links, and also the impact of the substrate network characteristics on the number of accepted virtual networks. These results provide guidelines for what should be expected on a real production environment.

With respect to the experimental results, it was shown that the mapping algorithm follows a linear trend with the increase of existing virtual networks. This trend is a good indicative of the mapping algorithm performance, although additional tests on larger and more complex substrate and virtual networks should be performed.

The required time to create virtual networks is a reflex of the performance of both the resource information gathering and virtual node creation mechanisms. The results achieved, show that even with a substrate network running near its limits, with 40 virtual networks embedded, the virtual network creation procedure is still able to provide fast virtual network creation times, i.e. less than 1 minute.

Every module was thoroughly analysed on chapter 5. Throughout this analysis, it was clear that a high performance design approach was taken and that task parallelism was fully explored.

The provided GUI makes the interaction with the user easy and intuitive to the point that designing, monitoring, and managing virtual networks is as easy as using a network simulator. The Manager examination demonstrated its mapping and data aggregation mechanisms, and, finally, the Agents' analysis revealed their main techniques developed for improving the performance of several critical features, such as virtual node creation and data gathering.

All of the attained results demonstrated the scalability and performance properties of the developed platform and respective algorithms. They have also proved that the existence of a single tool to efficiently and quickly instantiate and perform dynamic discovery and monitoring of virtual networks is feasible .

The virtualization platform developed can, therefore, perform the tasks that it was set to achieve: it is able to efficiently and intelligently map virtual networks into substrate networks, to perform discovery tasks, to monitor, and manage virtual networks.

7.2 Future Work

Although many of the desirable and needed features and mechanisms for a network virtualization platform were implemented, other features are also important and should be addressed.

Reconfiguration features could increase the versatility in virtual network management, by allowing on-the-fly addition, removal and reconfiguration of virtual nodes and links. When combined with migration features, several opportunities arise: virtual nodes and links could be reassigned to other locations without disrupting the virtual networks, in order to compensate for physical resources' and/or links' overload; the maintenance of the substrate network could be done without affecting the virtual networks; power savings could be achieved by shutting down unneeded or underused physical resources.

Fault-tolerance mechanisms have to be developed so that the virtual networks' operation is not compromised, even in the case of physical resources' failures.

The security aspects are not to be disregarded. In a substrate network running multiple virtual networks, care must be taken to ensure that access to the virtual networks is done on a secure way. Besides, virtual networks should not pose security risks to other ones running on the same substrate.

The access to virtual networks' information and management capabilities should resort to secure procedures, through authentication for example. In addition, the messages exchanged between each developed module, on the management network, should be made secure.

One other fundamental feature on production environments is the ability to create virtual networks spanning multiple InPs. Therefore, standard communication mechanisms should be developed and so should mapping algorithms that take into account multiple providers and their particularities.

Taking into account the described issues and lack of functionalities, it is clear that there is still a lot of work to be done if this network virtualization platform ever intends to take its place in an operator's network.

Bibliography

[1] 4WARD Consortium: *Virtualisation approach: Evaluation and integration*. Technical report, ICT-4WARD project, Deliverable D3.2., January 2010.

[2] 4WARD Consortium: *Virtualisation approach: Evaluation and integration - update*. Technical report, ICT-4WARD project, Deliverable D3.2.1, June 2010.

[3] Abramson, Darren, Jeff Jackson, Sridhar Muthrasanallur, Gil Neiger, Greg Regnier, Rajesh Sankaran, Ioannis Schoinas, Rich Uhlig, Balaji Vembu, and John Weigert: *Intel Virtualization Technology for directed I/O*. Intel Technology Journal, 10(3):179–192, August 2006, ISSN 1535-766X. http://developer.intel.com/technology/itj/2006/v10i3/2-io/1-abstract.htm.

[4] Adams, Keith and Ole Agesen: *A comparison of software and hardware techniques for x86 virtualization*. SIGOPS Oper. Syst. Rev., 40:2–13, October 2006, ISSN 0163-5980. http://doi.acm.org/10.1145/1168917.1168860.

[5] AMD: *Amd-v™ nested paging*. White paper, July 2008. http://developer.amd.com/assets/NPT-WP-1%201-final-TM.pdf.

[6] AMD: *AMD I/O Virtualization Technology (IOMMU) Specification - R 1.26*. White paper, February 2009. http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/34434.pdf.

[7] Andersen, David G.: *Theoretical approaches to node assignment*. Unpublished Manuscript, December 2002.

[8] Anderson, Thomas, Larry Peterson, Scott Shenker, and Jonathan Turner: *Overcoming the Internet Impasse through Virtualization*. Computer, 38:34–41, April 2005, ISSN 0018-9162. http://portal.acm.org/citation.cfm?id=1058219.1058273.

[9] Barham, Paul, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield: *Xen and the art of virtualization*. SIGOPS Oper. Syst. Rev., 37:164–177, October 2003, ISSN 0163-5980. http://doi.acm.org/10.1145/1165389.945462.

[10] Bierman, A. and K. Jones: *Physical Topology MIB - RFC 2922*, 2000. http://www.faqs.org/rfcs/rfc2922.html.

[11] Cacti: *Cacti*. http://www.cacti.net/.

[12] Carissimi, Alexandre: *Virtualização: da teoria a soluções*. In *Simpósio Brasileiro de Redes de Computadores 2008*, pages 173–207, Rio de Janeiro - Brazil, 2008. Rio de Janeiro - Brazil.

[13] Chowdhury, N. Mosharaf K. and Raouf Boutaba: *A survey of network virtualization*. Computer Networks, 54(5):862–876, April 2010, ISSN 1389-1286. `http://dx.doi.org/10.1016/j.comnet.2009.10.017`.

[14] Cisco: *Cisco*. `http://www.cisco.com/`.

[15] Cisco: *Cisco Nexus 1000V Datasheet*. `http://www.cisco.com/en/US/prod/collateral/switches/ps9441/ps9902/data_sheet_c78-492971.pdf`.

[16] Cisco: *Cisco Visual Networking Index—Forecast and Methodology 2008–2013*. White paper, June 2009. `http://www.cisco.com/en/US/solutions/collateral/ns341/ns525/ns537/ns705/ns827/white_paper_c11-481360.pdf`.

[17] Comeau, Les: *CP-40, The Origin of VM/370*. In *Proceedings of SEAS AM82*, September 1982.

[18] Crocker, S.: *Protocol Notes; RFC 36 - Updated by RFC 39 and 44*, march 1970. `http://tools.ietf.org/html/rfc36`.

[19] Egi, Norbert, Adam Greenhalgh, Mark Handley, Mickael Hoerdt, Felipe Huici, and Laurent Mathy: *Towards high performance virtual routers on commodity hardware*. In *CoNEXT '08: Proceedings of the 2008 ACM CoNEXT Conference*, pages 1–12, New York, NY, USA, 2008. ACM, ISBN 978-1-60558-210-8.

[20] Egi, Norbert, Adam Greenhalgh, Mark Handley, Mickael Hoerdt, Felipe Huici, Laurent Mathy, and Panagiotis Papadimitriou: *The virtual router project*. `http://nrg.cs.ucl.ac.uk/vrouter/`.

[21] Famzah, Ivan Zahariev: *popen_noshell*. `http://code.google.com/p/popen-noshell/`.

[22] Feamster, Nick, Lixin Gao, and Jennifer Rexford: *How to lease the internet in your spare time*. SIGCOMM Comput. Commun. Rev., 37(1):61–64, 2007, ISSN 0146-4833.

[23] GENI: *GENI - Global Environment for Network Innovations*. `http://www.geni.net/`.

[24] Handley, M.: *Why the internet only just works*. BT Technology Journal, 24(3):119–129, 2006, ISSN 1358-3948.

[25] Harrenstien, Ken, Vic White, and Elizabeth Feinler: *Hostnames server - rfc 811*, March 1982. `http://www.faqs.org/rfcs/rfc811.html`.

[26] Houidi, I., W. Louati, and D. Zeghlache: *A Distributed Virtual Network Mapping Algorithm*. In *Communications, 2008. ICC '08. IEEE International Conference on*, pages 5634–5640, 2008. `http://dx.doi.org/10.1109/ICC.2008.1056`.

[27] Hunter, Jason and Brett McLaughlin: *JDOM*. `http://www.jdom.org/`.

[28] Intel: *First the Tick, Now the Tock: Next Generation Intel Microarchitecture (Nehalem).* White paper, April 2008. http://www.intel.com/pressroom/archive/reference/whitepaper_nehalem.pdf.

[29] Jannotti, John, David K. Gifford, Kirk L. Johnson, M. Frans Kaashoek, and James W. O'Toole, Jr.: *Overcast: reliable multicasting with on overlay network.* In *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation - Volume 4*, OSDI'00, pages 14–14, Berkeley, CA, USA, 2000. USENIX Association. http://portal.acm.org/citation.cfm?id=1251229.1251243.

[30] Jelasity, Márk, Alberto Montresor, and Ozalp Babaoglu: *T-Man: Gossip-based fast overlay topology construction.* Comput. Netw., 53(13):2321–2339, 2009, ISSN 1389-1286.

[31] Jelasity, Márk and Ozalp Babaoglu: *T-Man: Gossip-based overlay topology management.* In *In 3rd Int. Workshop on Engineering Self-Organising Applications (ESOA'05)*, pages 1–15. Springer-Verlag, 2005.

[32] Keromytis, Angelos D., Vishal Misra, and Dan 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, ISBN 1-58113-570-X. http://doi.acm.org/10.1145/633025.633032.

[33] Kohlera, E., R. Morris, B. Chen, J. Jahnotti, and M. F. Kasshoek: *The click modular router.* In *ACM Transaction on Computer Systems, vol. 18, no. 3*, pages 263–297. ACM, 2000. http://read.cs.ucla.edu/click/click.

[34] Lischka, Jens and Holger Karl: *A virtual network mapping algorithm based on subgraph isomorphism detection.* In *VISA '09: Proceedings of the 1st ACM workshop on Virtualized infrastructure systems and architectures*, pages 81–88, New York, NY, USA, 2009. ACM, ISBN 978-1-60558-595-6.

[35] Liu, Yu, Guangxi Zhu, and Hao Yin: *A practical hybrid mechanism for peer discovery.* In *Intelligent Signal Processing and Communication Systems, 2007. ISPACS 2007. International Symposium on*, pages 706 –709, nov. 2007.

[36] Lu, Jing and Jonathan Turner: *Efficient mapping of virtual networks onto a shared substrate.* Technical report, Washington University in St. Louis, 2006. http://www.arl.wustl.edu/~{}jl1/research/tech_report_2006.pdf.

[37] Lua, Eng Keong, Jon Crowcroft, Marcelo Pias, Ravi Sharma, and Steven Lim: *A survey and comparison of peer-to-peer overlay network schemes.* IEEE Communications Surveys and Tutorials, 7:72–93, 2005.

[38] MacDonald, Neil: *Neil macdonald's gartner blog.* http://blogs.gartner.com/neil_macdonald/.

[39] Malkhi, Dahlia, Moni Naor, and David Ratajczak: *Viceroy: a scalable and dynamic emulation of the butterfly.* In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, PODC '02, pages 183–192, New York, NY, USA, 2002. ACM, ISBN 1-58113-485-1. http://doi.acm.org/10.1145/571825.571857.

[40] Mannie, E.: *Generalized Multi-Protocol Label Switching (GMPLS) Architecture; RFC 3945*, 2004. http://tools.ietf.org/html/rfc3945.

[41] McKeown, Nick, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner: *OpenFlow: enabling innovation in campus networks*. SIGCOMM Comput. Commun. Rev., 38(2):69–74, 2008. http://portal.acm.org/citation.cfm?id=1355734.1355746.

[42] McQuillan, J., I. Richer, and E. Rosen: *The new routing algorithm for the arpanet*. Communications, IEEE Transactions on, 28(5):711 – 719, may 1980, ISSN 0090-6778.

[43] Menascé, Daniel A.: *Virtualization: Concepts, Applications, and Performance Modeling*, 2005.

[44] Nagios: *Nagios*. http://www.nagios.org/.

[45] Nagle, John: *Congestion control in IP/TCP internetworks*. SIGCOMM Comput. Commun. Rev., 14(4):11–17, 1984, ISSN 0146-4833.

[46] Neiger, Gil, Amy Santoni, Felix Leung, Dion Rodgers, and Rich Uhlig: *Intel Virtualization Technology: Hardware Support for Efficient Processor Virtualization*. Intel Technology Journal, 10(3):167–177, August 2006, ISSN 1535-766X. http://developer.intel.com/technology/itj/2006/v10i3/1-hardware/1-abstract.htm.

[47] Networks, Juniper: *Juniper networks*. http://www.juniper.net/us/en/.

[48] Peterson, Larry, Tom Anderson, David Culler, and Timothy Roscoe: *A blueprint for introducing disruptive technology into the internet*. SIGCOMM Comput. Commun. Rev., 33:59–64, January 2003, ISSN 0146-4833. http://doi.acm.org/10.1145/774763.774772.

[49] PlanetLab: *PlanetLab - An Open Platform for Developing, Deploying, and Accessing Planetary-Scale Services*. http://www.planet-lab.org/.

[50] Popek, Gerald J. and Robert P. Goldberg: *Formal requirements for virtualizable third generation architectures*. Commun. ACM, 17(7):412–421, 1974, ISSN 0001-0782.

[51] Postel, J.: *Computer mail meeting notes - rfc 805*, February 1982. http://www.faqs.org/rfcs/rfc805.html.

[52] Ratnasamy, Sylvia, Paul Francis, Mark Handley, Richard Karp, and Scott Schenker: *A scalable content-addressable network*. SIGCOMM Comput. Commun. Rev., 31:161–172, August 2001, ISSN 0146-4833. http://doi.acm.org/10.1145/964723.383072.

[53] Robin, John Scott and Cynthia E. Irvine: *Analysis of the Intel Pentium's ability to support a secure virtual machine monitor*. In *Proceedings of the 9th conference on USENIX Security Symposium - Volume 9*, pages 10–10, Berkeley, CA, USA, 2000. USENIX Association. http://portal.acm.org/citation.cfm?id=1251306.1251316.

[54] Rodriguez, Sergio R.: *Topology Discovery Using Cisco Discovery Protocol*. CoRR, abs/0907.2121, 2009.

[55] Rose, Robert: *Survey of system virtualization techniques*. Technical report, 2004.

[56] Rosen, E.: *Multi-protocol label switching (mpls) architecture; rfc 3031*, 2001. http://tools.ietf.org/html/rfc3031.

[57] Rowstron, Antony and Peter Druschel: *Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems*. In *Middleware '01: Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, pages 329–350. Springer-Verlag, 2001. http://www.springerlink.com/content/7y5mjjep0hqlctv6.

[58] Schaffrath, Gregor, Christoph Werle, Panagiotis Papadimitriou, Anja Feldmann, Roland Bless, Adam Greenhalgh, Andreas Wundsam, Mario Kind, Olaf Maennel, and Laurent Mathy: *Network virtualization architecture: proposal and initial prototype*. In *VISA '09: Proceedings of the 1st ACM workshop on Virtualized infrastructure systems and architectures*, pages 63–72, New York, NY, USA, 2009. ACM, ISBN 978-1-60558-595-6.

[59] SPEC: *SPEC Virtualization Committee*. http://www.spec.org/specvirtualization/index.html.

[60] Stoica, Ion, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan: *Chord: A scalable peer-to-peer lookup service for internet applications*. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 149–160, New York, NY, USA, 2001. ACM, ISBN 1-58113-411-8.

[61] Su, Zaw Sing and Jon Postel: *Domain naming convention for internet user applications - rfc 819*, August 1982. http://www.faqs.org/rfcs/rfc819.html.

[62] Subramanian, Lakshminarayanan, Ion Stoica, Hari Balakrishnan, and Randy H. Katz: *OverQos: an overlay based architecture for enhancing internet QoS*. In *NSDI'04: Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation*, pages 6–6, Berkeley, CA, USA, 2004. USENIX Association.

[63] Tripathi, S., N. Droux, K. Belgaied, and S. Khare: *Crossbow Virtual Wire: Network in a Box*. In *USENIX LISA '09*. USENIX Association, nov 2009.

[64] Tripathi, Sunay, Nicolas Droux, Thirumalai Srinivasan, and Kais Belgaied: *Crossbow: from hardware virtualized NICs to virtualized networks*. In *VISA '09: Proceedings of the 1st ACM workshop on Virtualized infrastructure systems and architectures*, pages 53–62, New York, NY, USA, 2009. ACM, ISBN 978-1-60558-595-6.

[65] Turner, Jon and David Taylor: *Diversifying the internet*. In *In Proc. IEEE GLOBECOM*, pages 755–760, 2005.

[66] Uhlig, Rich, Gil Neiger, Dion Rodgers, Amy L. Santoni, Fernando C. M. Martins, Andrew V. Anderson, Steven M. Bennett, Alain Kagi, Felix H. Leung, and Larry Smith: *Intel virtualization technology*. Computer, 38(5):48–56, 2005, ISSN 0018-9162.

[67] Veillard, Daniel: *libxml2*. http://www.xmlsoft.org/.

[68] VMware: *Understanding Full Virtualization, Paravirtualization, and Hardware Assist*. White paper, October 2007. `http://www.vmware.com/files/pdf/VMware_paravirtualization.pdf`.

[69] Waxman, B.M.: *Routing of multipoint connections*. Selected Areas in Communications, IEEE Journal on, 6(9):1617 –1622, dec 1988, ISSN 0733-8716.

[70] Whitaker, Andrew, Marianne Shaw, and Steven D. Gribble: *Denali: Lightweight Virtual Machines for Distributed and Networked Applications*. In *In Proceedings of the USENIX Annual Technical Conference*, 2002.

[71] XORP: *XORP - eXtensible Open Router Platform*. `http://www.xorp.org/`.

[72] Zhu, Y. and M. Ammar: *Algorithms for assigning substrate network resources to virtual network components*. In *INFOCOM 2006. 25th IEEE International Conference on Computer Communications. Proceedings*, pages 1–12, 2006.