HAL
open science

Cross-layer self-diagnosis for services over programmable networks

José Manuel Sánchez Vílchez

To cite this version:

José Manuel Sánchez Vílchez. Cross-layer self-diagnosis for services over programmable networks. Networking and Internet Architecture [cs.NI]. Institut National des Télécommunications, 2016. English. NNT : 2016TELE0012 . tel-01885497

HAL Id: tel-01885497

https://theses.hal.science/tel-01885497v1

Submitted on 2 Oct 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Spécialité : Systèmes Informatiques

Ecole doctorale : Informatique, Télécommunications et Electronique de Paris

Présentée par

José Manuel Sánchez Vílchez

Pour obtenir le grade de
DOCTEUR DE TELECOM SUDPARIS

Auto-diagnostic multi-couche pour services sur réseaux programmables

Soutenue le 7 juillet 2016

Devant le jury composé de :
Directeur de thèse
Noel Crespi, Professeur, Institut Mines Telecom/ Telecom SudParis, France

Encadrante
Imen Grida Ben Yahia, Chef de projet, Autonomic Networking, Orange labs, France

Rapporteurs
Ernesto Damiani, Professeur, Université de Milan, Italie
Xavier Lagrange, Professeur, Telecom Bretagne, France

Examinateurs
Sébastien Tixeuil, Professeur, Université Pierre Et Marie Curie-LIP6, France
Prosper Chemouil, Expert Orange, Réseaux Futures, Orange Labs Networks, France
Laurent Ciavaglia, Directeur senior de projets, Laboratoire Réseaux, Analytique et Securité, Nokia Bell Labs, France
Antonio Manzalini, Senior Manager, IEEE SDN Initiative Co-Chair, Telecom Italia, Italie

N° NNT : 2016TELE0012

Acknowledgements

Firstly, I would like to express my sincere gratitude to my advisor in Orange Labs, Dr. Imen Grida Ben Yahia, for the continuous support of my Ph.D study and related research, for her encouragement, patience, motivation, and immense knowledge. Her guidance has helped me to conduct the research during this thesis and at the last stretch of this thesis. I would like to thank Pr. Noel Crespi, my research director, for the opportunity to enroll in a PhD inside his team, for the follow-up and the interesting exchanges with its research team at Telecom SudParis. I would like also to thank you all those colleagues in its team for their suggestions and advice, especially Angel and Reza.

My sincere thanks go to Bertrand Decocq who provided me an opportunity to join his team inside Orange Labs to do this PhD, but also provided me the opportunity to join his team as a permanent employee inside Orange. Without the precious support of all these aforementioned persons, it would not have been possible to conduct this research.

The last but not least, I would like also to express my deep and sincere gratitude to all my family and friends who have always been there to provide me support in both high and low moments, especially at the beginning of this exciting adventure.

José Manuel Sánchez Vílchez

7th July 2016

Declaration

This thesis manuscript is the result of my own research work and contains the outcome of other researches found in the literature. We properly referenced all these sources at the end of this manuscript. This is the very first time I submit this thesis manuscript for the obtention of the Doctor of Philosophy (PhD) in computer networks by Telecom SudParis.

José Manuel Sánchez Vílchez

7th July 2016

Dedication

To the loving memory of my mother Maria

To my father Manuel and my sister Maria

To my girlfriend Carolina and her marvellous family, Maribel and Salvador

Abstract

Current networks serve millions of mobile customer devices. They encompass heterogeneous equipment, transport and management protocols, and vertical management tools, which are very difficult and costly to integrate. Fault management operations are far from being automated and intelligent, where only around 40% of alarms are redundant only around 1-2% of alarms are correlated at most in a medium-size operational center. This indicates that there is a significant alarm overflow for human administrators, which inherently derives in high OPEX due to the increasing need to employ high-skilled people to perform fault management tasks. In conclusion, the current level of automation in fault management tasks in Telco networks is not at all adequate for programmable networks, which promise a high degree of programmability and flexibility to reduce the time-to-market.

Automation in fault management has become more necessary with the advent of programmable networks, led by SDN (Software-Defined Networking), NFV (Network Functions Virtualization) and the Cloud. Indeed, the implantation of those paradigms has accelerated the convergence between networks and IT realms, which is accelerating the transformation of current networks and leading to a rethinking of network and service management and operations, in particular fault management operations.

This thesis envisages the application of self-healing principles in SDN and NFV combined infrastructures, focuses on self-diagnosis tasks as the main enabler of self-healing. The core of the thesis is to devise a self-diagnosis approach able to diagnose at run-time the dynamic virtualized networking services. This self-diagnosis approach correlates the state of those services with the states of their underlying virtualized resources (VNFs and virtual links) and the underlying network infrastructure state. This approach takes into account the mobility, dynamicity, and sharing of resources in the underlying infrastructure.

Keywords

Autonomics; self-healing; programmable networks; SDN;NFV; fault management; Bayesian Networks; self-modeling; self-diagnosis; alarm correlation; fault-isolation;

Résumé

Les réseaux actuels servent millions de clients mobiles et ils se caractérisent par équipement hétérogène et protocoles de transport et de gestion hétérogènes, et des outils de gestion verticaux, qui sont très difficiles à intégrer dans leur infrastructure. La gestion de pannes est loin d'être automatisée et intelligent, ou un 40 % des alarmes sont redondantes et seulement un 1 ou 2% des alarmes sont corrélées au plus dans un centre opérationnel de taille moyenne. Ça indique qu'il y a un débordement significatif des alarmes vers les administrateurs humains, lequel a comme conséquence un haut OPEX vue la nécessité d'embaucher de personnel expert pour accomplir les tâches de gestion de pannes. Comme conclusion, le niveau actuel d'automatisation dans les tâches de gestion de pannes dans réseaux télécoms n'est pas adéquat du tout pour adresser les réseaux actuels.

L'automatisation de la gestion des pannes devient de plus en plus nécessaire avec l'arrivée des réseaux programmables, lesquels promettent la programmation des ressources et la flexibilité afin de réduire le time-to-market des nouveaux services et la gestion plus efficace. En fait, les paradigmes SDN (Software-Defined Networking), NFV (Network Functions Virtualization) et le Cloud accélèrent la convergence entre les domaines des réseaux et la IT, laquelle accélère de plus en plus la transformation des réseaux télécoms actuels en menant à repenser les opérations de gestion de réseau et des services, en particulier les opérations de gestion de fautes.

NFV vise à déployer les fonctions réseau en hardware banalisée qui soit indépendant des fournisseurs d'équipement afin de réduire le cout d'intégration des nouvelles fonctions réseau dans l'infrastructure de l'opérateur. SDN c'est une nouvelle architecture de réseau qui vise à flexibiliser la connectivité entre ces fonctions réseau virtualisées (VNF), étant basée sur des interfaces ouvertes, une claire séparation entre la couche de control et de donnés, et de l'abstraction.

Les réseaux évoluent vers des réseaux programmables avec l'approche de Software-Defined Networking. Cependant la couche de gestion n'est pas encore définie: la gestion des fautes, la gestion des performances ainsi que le provisioning sont à faire évoluer pour bénéficier de la programmabilité des réseaux avec SDN. Un système d'autodiagnostic, tel comme est proposé dans ces travaux de thèse, est nécessaire pour assurer la continuité des services et la pérennité des réseaux avec SDN. Le but de ces travaux de thèse est d'étudier l'opportunité d'introduire de l'autonomie dans les réseaux de demain afin d'optimiser la gestion des pannes et des dysfonctionnements à travers un système d'autoréparation.

Mots-clés

Autonomics; systems d'auto réparation; réseaux programmables; SDN;NFV; gestion de pannes; réseaux bayésiens; auto modélisation; self-diagnotique; correlation d'alarmes; fault-isolation

Contents

CHAPTER 1 INTRODUCTION 19

1.1 Research context 19

1.2 Software-Defined Networking 19

1.3 Networks Function Virtualization (NFV) 20

1.4 Research problem: Why do we need automation on diagnosis of networking services over programmable infrastructures? 20

1.5 Motivating example 21

1.6 Thesis objectives and principles 23

1.7 Research Questions 23

1.7.1 Research methodology and scientific contributions 24

1.8 Project Contributions and presentations 26

1.9 Thesis organization 27

CHAPTER 2 PROGRAMMABLE NETWORKS AND FAULT MANAGEMENT CHALLENGES 28

2.1 Introduction 28

2.2 Overview of SDN 28

2.2.1 SDN Architecture 28

2.2.2 Forwarding in SDN OpenFlow 29

2.2.3 Types of control in SDN 31

2.2.4 Influence of the type of control 32

2.3 Related work on Fault Management in SDN 36

2.3.1 Fault management solutions for the data plane 36

2.3.2 Fault management solutions for the control plane 36

2.3.3 Fault management solutions for the application plane 38

2.3.4 Fault management solutions for legacy and OpenFlow equipment 39

2.3.5 Fault management solutions including diagnosis 40

2.4 Overview of NFV 40

2.4.1 NFV Architecture 40

2.5 Related work on Fault Management in NFV **44**

2.6 Overview of SDN and NFV combined infrastructures **46**

2.7 Dynamicity of SDN and NFV infrastructures **48**
2.7.1 Dynamicity in SDN 48
2.7.2 Dynamicity in NFV 50

2.8 Conclusion **52**

CHAPTER 3 RELATED WORK ON SELF-HEALING SYSTEMS **54**

3.1 Introduction **54**

3.2 Self-Healing overview **54**
3.2.1 Preliminaries 55
3.2.2 Definition 56
3.2.3 Origins of self-healing 56
3.2.4 Properties of a self-healing system 58
3.2.5 Discussion 59

3.3 Self-Healing architecture **60**
3.3.1 Control-loop architecture 60
3.3.2 Self-healing system state diagram 63
3.3.3 Restoration stages of a self-healing system 65
3.3.1 Discussion 66

3.4 Self-healing mechanisms **68**
3.4.1 Detection mechanisms 68
3.4.2 Diagnosis mechanisms 69
3.4.3 Recovery and remediation mechanisms 70
3.4.4 Discussion 71

3.5 Self-healing algorithms **72**
3.5.1 Algorithm classification per application domain 72
3.5.2 Algorithm classification per objective 74
3.5.3 Algorithm classification per self-healing functional task 76

3.6 Self-Diagnosis algorithms **76**
3.6.1 Data mining algorithms 76
3.6.2 Control-theory 78
3.6.3 Case-Based Reasoning 79
3.6.4 Discussion 80

3.7 Bayesian Networks **81**
3.7.1 Definitions 81
3.7.2 Reasoning in Bayesian Networks 83
3.7.3 Properties of Bayesian Networks 84
3.7.4 Challenges of Bayesian Networks in diagnosis 86
3.7.5 Related work on the generation of the Bayesian Network 87

3.7.6 Related work on Network Diagnosis through Bayesian Networks 92
3.7.7 Topology-Aware and Service-Aware self-diagnosis 103

3.8 Conclusion 104

CHAPTER 4 SELF-DIAGNOSIS ARCHITECTURE FOR PROGRAMMABLE NETWORKS 107

4.1 Introduction 107

4.2 Proposal of a Self-Healing architecture for SDN 107
4.2.1 Position of a self-healing system in the SDN infrastructure 107
4.2.2 Self-Healing architecture for SDN infrastructures 109

4.3 Overall Self-Diagnosis architecture 113

4.4 Self-Modeling module 114
4.4.1 Types of resources modelled 114
4.4.2 Problem formalization 116
4.4.3 Description of resources dependencies through templates 118
4.4.4 Generation of the network dependency graph 121
4.4.5 Generation of the service dependency graph 125

4.5 Exploitation of the service dependency graph for Root Cause Analysis 127

4.6 Conclusion 129

CHAPTER 5 RESULTS AND EVALUATION 130

5.1 Introduction 130

5.2 Topology-Aware Self-Diagnosis Evaluation 130
5.2.1 Generation of the network dependency graph 130
5.2.2 Exploitation of the Network dependency graph for Root Cause Analysis 132
5.2.3 Performance Evaluation 138

5.3 Service-Aware Self-Diagnosis 139
5.3.1 Generation of the services dependency graph 139
5.3.2 Exploitation of the Service dependency graph for Root Cause Analysis 140
5.3.3 Performance Evaluation 144

5.4 Self-Healing framework for video streaming applications over SDN 145
5.4.1 Overall description of the self-healing testbed 146
5.4.2 Implementation 146
5.4.3 Transformation of the network topology into a machine-readable format 147
5.4.4 Construction of the dependency graph 147
5.4.5 Root cause analysis 148
5.4.6 Update of the dependency graph 150
5.4.7 Recovery actions 153

5.5 Conclusion 154

CHAPTER 6 CONCLUSIONS AND FUTURE WORK 156

6.1 Conclusions 156

6.2 Future work 157

6.2.1 Proactive self-diagnosis techniques to avoid service failures 157

6.2.2 Exploration of machine-learning techniques 157

6.2.3 Observability and detection techniques 157

6.2.4 Extension of the fault propagation model 157

List of Figures

Figure 1. Service topology changes in NFV-SDN infrastructures 22
Figure 2. SDN layered architecture by ONF 28
Figure 3. Main components of an OpenFlow switch by ONF 29
Figure 4. SDN forwarding in OpenFlow 31
Figure 5. Example of flow installation: (a) In-band control and (b) out-of-band control 32
Figure 6. Fault propagation differences: (a) In-band control and (b) out-of-band control 33
Figure 7. Degree of centralization: (a) in-band, (b) out-of-band control 34
Figure 8. Control link congestion: (a) in-band, (b) out-of-band control 34
Figure 9. Flow installation delay: (a) in-band, (b) out-of-band 35
Figure 10. Policy Management architecture 38
Figure 11. i-NMCS architecture embedded in the SDN controller 39
Figure 12. NFV definition 40
Figure 13. Functional NFV architecture by ETSI NFV 41
Figure 14. Network service instantiation in the NFV architecture by ETSI NFV 41
Figure 15. (a) VNF NCT, and (b) NFP (blue, red, and yellow) 43
Figure 16. VNFI state transitions by ETSI NFV 44
Figure 17. Different types of implementation of a NFVI node by ETSI NFV 45
Figure 18. Detailed VNF layered architecture by ETSI NFV 45
Figure 19. Networking service deployment in a combined SDN and NFV infrastructure 48
Figure 20. Changes on the network topology in a centralized SDN infrastructure 49
Figure 21. Different network topologies in a distributed SDN infrastructure 49
Figure 22. Changing the type of control to overcome a fault on a control link 50
Figure 23. Traffic restoration in SDN 50
Figure 24. Examples of dynamicity in NFV architectures: (a) scaling-up, (b) scaling-out, and (c) VNFI migration 52
Figure 25. Fault-error-failure chain 55
Figure 26. Quality factors associated to Self-Healing 57
Figure 27. Hierarchy of Self-* properties 58
Figure 28. Comparison of quality factors ensured by a self-healing system and a resilience system 59
Figure 29. Self-Healing challenges from a resilience perspective 60
Figure 30. Autonomic manager 61

Figure 31. Self-Healing system architecture 62

Figure 32. Self-healing state diagram 64

Figure 33. Resiliency state diagram 64

Figure 34. Stages of a Self-Healing system.....65

Figure 35. Similarities between the Self-Healing control-loop (in red) and the D^2R^2+DR control-loop66

Figure 36. Comparison between self-healing and resilience diagram states.....67

Figure 37. Proposed diagram state of a Self-Healing system.....68

Figure 38. Proactive diagnosis 70

Figure 39. Reactive diagnosis.....70

Figure 40. k-NN classification algorithm example.....77

Figure 41. Control theory control-loop 79

Figure 42. Case-Based Reasoning control loop 79

Figure 43. Two different types of case structures: (a) memory-model, (b) category and exemplar model 80

Figure 44. Examples of (a) directed acyclic graph, (b) undirected acyclic graph.....82

Figure 45. Topological order in dependency graphs 82

Figure 46. Belief propagation in chain and tree.....83

Figure 47. QMR-DT dependency graph: (a) N diseases, M symptoms, (b) N=2 diseases, M=1 symptoms 84

Figure 48. Three ways of connecting three variables in a DAG: (a) chain, (b) fork, (c) collider 84

Figure 49. Example of probabilistic dependency graph 85

Figure 50. Diagnosis with Bayesian Networks.....86

Figure 51. Example of possible dependency graphs for a 3-variable problem 88

Figure 52. Cases database example for m=9, n=3 and generated dependency graph.....89

Figure 53. Probabilistic dependency graph extraction from a database 91

Figure 54. Security ontology used by Fenz et al. in (Fenz, 2011) and derived BN example 92

Figure 55. GPON-FTTH network architecture considered.....93

Figure 56. End-to-end service model proposed by Steinder and Sethi.....94

Figure 57. Decomposition of a bridge-to-bridge network topology in a dependency graph 94

Figure 58. Example of VPN and example of calculated dependency graph for the packet loss problem95

Figure 59. Hybrid BN-CBR self-diagnosis approach for VPN proposed by Bennacer et al.....96

Figure 60. Proposed templates: (a) Machine, (b) Application, (c) NbrSet, and (d) Path 98

Figure 61. Sherlock architecture 99

Figure 62. Simplified IP configuration sequence diagram in IMS.....100

Figure 63. Generic model proposed for the IP configuration service 101

Figure 64. BN instance extracted from the generic model for a user User X.....101

Figure 65. Extension of the BN with two BN instances from users User X and User Y.....102

Figure 66. Differences between both types of self-diagnosis: (a) topology-aware, (b) service-aware103

Figure 67. Locations of the self-healing architecture: (a) application, (b) data, (c) control, and (d) management plane.....108

Figure 68. Multi-control loop Self-Healing architecture for SDN 110

Figure 69. Multi-layer Self-diagnosis architecture 113

Figure 70. Detection block that updates the network and services dependency graph 114

Figure 71. Hierarchy class of network resources in programmable networks 116

Figure 72. (a) End-to-end service, (b) Underlying network topology of a virtual link 117

Figure 73. Networking service decomposition in different network segments 117

Figure 74. Zoom on the underlying resources involved in an external virtual link between VNFs 118

Figure 75. Dependency graph of a host: (a) embedding K Applications, (b) embedding K VNFIs 119

Figure 76. Dependency graph of a Controller 120

Figure 77. Dependency graph of a switch 120

Figure 78. Dependency graph of a link 121

Figure 79. JSON data structures provided by: (a) OpenDaylight, (b) Floodlight 121

Figure 80. Definition of network topologies: (a) Tree(D,F), (b) Linear(N) 122

Figure 81. (a) Non–topologically sorted network dependency graph, and (b) topologically sorted network dependency graph 124

Figure 82. Example of Network dependency graph (Q=3 nodes and P=2 links) 125

Figure 83. Generic Network dependency graph (Q nodes and P links) 125

Figure 84. Virtual Resources dependency Graph generation 125

Figure 85. Services dependency graph of one network service 126

Figure 86. Generic Services dependency graph of N network services 127

Figure 87. Root Cause Analysis over the generated services dependency graph of one network service 129

Figure 88. Network Dependency graph of a linear topology (N=2) with out-of-band control 131

Figure 89. Network Dependency graph of a linear topology (N=2) with in-band control 131

Figure 90. Modelled variables in a control link at physical level 132

Figure 91. Network dependency graph of a control link at physical level 132

Figure 92. Root Cause analysis: (a) switch's port up, (b) switch's port down 133

Figure 93. Modelled variables in a control link at physical and logical level 133

Figure 94. Dependency graph of a control link at physical and logical level 134

Figure 95. Root Cause analysis with two different observations of controller's card, switch's port and applications state 134

Figure 96. Root Cause analysis: Case 1, Faulty SDN controller 135

Figure 97. Root Cause analysis: Case 2, simultaneous faulty links at control and data planes 136

Figure 98. Root Cause Analysis on changing CPU conditions in linear L=2 network topology 137

Figure 99. Root Cause Analysis on changing CPU conditions in single network topology 138

Figure 100. Speed as a function of the number of elements 139

Figure 101. Diagnosis of two networking services in different network topologies: (a) tree topology, (b) linear topology 139

Figure 102. RCA strategy on extending the services dependency graph with network services 141

Figure 103. Entropy reduction with the RCA strategy on extending the services dependency graph142

Figure 104. RCA strategy on reducing the network dependency graph 143

Figure 105. Entropy reduction with the RCA strategy on reducing the network dependency graph 144

Figure 106. Example of Network dependency graph (Q=3 nodes and P=2 links) 145

Figure 107. Implementation of the self-healing framework 146

Figure 108. Transformation of the network topology into a machine-readable format.....147

Figure 109. Construction of the dependency graph from templates.....148

Figure 110. Root Cause Identification process.....149

Figure 111. Finer-Granularity of the Root Cause Identification until component level: (a) faulty link, (b) faulty controller 150

Figure 112. Network dependency graph regeneration and update process 151

Figure 113. Case 1: Persistence of the root cause calculation with topological changes 152

Figure 114. Case 2: Update of the root cause calculation with topological changes.....153

Figure 115. Recovery action suggestion according to the calculated root cause 153

Figure 116. Service restoration in two different cases: (a) faulty link, (b) faulty SDN controller application 153

Figure 117. Fault injection and unavailabililty on streaming application due to faulty link.....154

Figure 118. Fault injection and unavailabililty on streaming application due to faulty SDN controller application 154

List of Tables

Table 1. Tag fields defined in OpenFlow 1.0 29
Table 2. Some examples of actions in OpenFlow 1.0 29
Table 3. Some examples of faults in NFV combined infrastructures 45
Table 4. Proposed functional tasks and corresponding mechanisms 68
Table 5. Algorithms classification per application domain 73
Table 6. Self-healing algorithms classification per objective 75
Table 7. Algorithm classification per self-healing task 76
Table 8. Variables considered in a Web server 97
Table 9. Detection actions at each plane 112
Table 10. Recovery actions at each plane 112
Table 11. Types of resources considered per layer 117
Table 12. Topology parameters analyzed by the topology interpreter algorithm 121
Table 13. Output format of Nt and Lt descriptors for Linear (N=5) and Tree(D=2,F=2) topologies ... 122
Table 14. Dependency subgraph instantiation algorithm 123
Table 15. Topological Sorting algorithm 124
Table 16. E_L Edge addition algorithm 124
Table 17. Virtual resources dependency graph generation algorithm 125
Table 18. CPT of a generic component Y in a network resource 127
Table 19. CPTs for the CPU component 128
Table 20. CPTs for the switch port component 128
Table 21. CPTs for the process components 128
Table 22. CPTs for the configuration of a process 128
Table 23. CPU loads at both instants of time 137
Table 24. CPU loads at both instants of time 137
Table 25. Number of vertices (V) as a function of the number of hosts (N_H) 138
Table 26. Affected networking services and underlying physical paths 139
Table 27. VNF forwarding graphs and physical dependencies 140
Table 28. Zoom on the root cause probabilities per host (%) 142
Table 29. Zoom on the root cause probabilities per host (%) 144
Table 30. Cost of extending the services dependency graph 145
Table 31. Vertices generated in several network topologies 145
Table 32. Shape of the supervised network components included in the dependency graph 148

List of accepted papers

- J. Sanchez, I. Grida Ben Yahia, N. Crespi, " THESARD: on The road to resiliencE in SoftwAre-defined networking thRough self-Diagnosis " 2nd IEEE Conference on Network Softwarization, Seoul, Korea, 6-10 June 2016
- J. Sanchez, I. Grida Ben Yahia, N. Crespi, "Self-Modeling based Diagnosis of Services over Programmable Networks," 2nd IEEE Conference on Network Softwarization, Seoul, Korea, 6-10 June 2016.
- J. Sanchez, I. Grida Ben Yahia, N. Crespi, "Self-Modeling Based Diagnosis of Software-Defined Networks," Workshop MISSION 2015 at 1st IEEE Conference on Network Softwarization, London, 13-17 April 2015.
- J. Sanchez, I. Grida Ben Yahia, et. al., "Softwarized 5G networks resiliency with self-healing," 1st Internatio-nal Conference on 5G for Ubiquitous Connectivity (5GU), 2014.
- J. Sanchez, I. Grida Ben Yahia, N. Crespi, "Self-healing Mechanisms for Software Defined Networks". AIMS 2014, 30 June-3rd July 2014.

List of submitted papers

- J. Sanchez, I. Grida Ben Yahia, N. Crespi, "Self-Modeling based Diagnosis of Services over Programmable Networks," International Journal On Network Management (IJNM)

List of abbreviation

Abbreviation	Expansion
ANN	Artificial Neural Networks
API	Application Programming Interface
ARF	Access Relay function
ARQ	Automatic Repeat-reQuest
BN	Bayesian Networks
BRAS	Broadband remote access server
BSS	Business Support Systems
CBN	Causal Bayesian Networks
CBR	Case-Based Reasoning
CLF	Connectivity Session Location and Repository Function
CPE	Customer Premises Equipment
CPT	Conditional Probability Table
CPU	Core Processor Unit
DAG	Directed Acyclic Graph
DHCP	Dynamic Host Configuration Protocol
DNS	Domain Name System
DSLAM	Digital Subscriber Line Access Multiplexer
ECMP	Equal-cost Multi-Path routing
EPC	Evolved Packet Core
FCAPS	Fault, Configuration, Accounting, Performance, Security
FTTH	Fiber To The Home
GPON	Gigabit-capable Passive Optical Networks (GPON)
GUI	Graphical User Interface
HDD	Hard Disk Drive
HMM	Hidden Markov Model
JSON	JavaScript Object Notation
KPI	Key Performance Indicator
LLDP	Link Layer Discovery Protocol
MAC	Media Access Control
MANO	Management And Orchestration
MDT	Mean Downtime
MIMO	Multiple Input Multiple Output
MPLS	Multiprotocol Label Switching
MTTF	Mean Time To Failure
MTTR	Mean time to repair
NACF	Network Access Configuration Function
NBI	Northbound Interface
NFV	Network Function Virtualization
NFVI	Network Function Virtualized Infrastructure
NIC	Network Interface Card
NMS	Network Management System
NOC	Network Operations Center
NSR	Network Service Record
OLT	Optical Line Terminal
ONF	Optical Line Terminal
ONT	Optical Network Terminal
OSS	Operations Support Systems
RAN	Radio Access Network
RCA	Root Cause Analysis
SBC	Session Border Controller
SBI	Southbound Interface
SDN	Software-Defined Networking
SDO	Standards Development Organization
SFC	Service Function Chaining
SIP	Session Initiation Protocol
SISO	Single Input Single Output
SLA	Service Level Agreement
SOM	Self-Organizing Maps
TCP	Transmission Control Protocol

UDP	User Datagram Protocol
UE	User Equipment
VIM	Virtualization Infrastructure Manager
VLAN	Virtual Local Area Network
VLD	Virtual Link Descriptor
VLR	Virtual Link Record
VNF	Virtual Network Function
VNF FG	Virtual Network Function Forwarding Graph
VNF NCT	Virtual Network Function Network Connectivity Topology
VNFC	Virtual Network Function Component
VNFI	Virtual Network Function Instance
VNFM	Virtual Network Function Manager
VNFR	Virtual Network Function Record
VPN	Virtual Private Network
WDM	Wavelength Division Multiplexing

List of variables

Variable	Meaning
N	Number of networking services modelled
M_i	Number of virtual links composing the i-th networking service
N_i	Number of VNFs composing the i-th networking service
P	Number of links in the network topology
Q	Number of nodes in the network topology
X	Number of switches in the network topology
Y	Number of hosts in the network topology
Z	Number of controller in the network topology
$S = \{S_1, \dots, S_X\}$	Set of switches of the network topology
$H = \{H_1, \dots, H_Y\}$	Set of hosts of the network topology
$C = \{C_1, \dots, C_Z\}$	Set of controllers of the network topology
r	Number of control links in the network topology
q	Number of data links in the network topology
$CL = \{CL_1, \dots, CL_r\}$	Set of control links of the network topology
$DL = \{DL_1, \dots, DL_q\}$	Set of data links of the network topology
$VL_{m,n}^{(k)}$	Virtual link between two VNFs *VNF_m and VNF_n*
$\rho_{m,n}^{(k)}$	Physical path between two hosts' NICs
${n_f}^{(k)}$	Number of flows installed to establish$VL_{m,n}^{(k)}$and $\rho_{m,n}^{(k)}$
${n_f}^{(k)}$	Number of intermediate switches composing each virtual link $VL_{m,n}^{(k)}$
${n_l}^{(k)}$	Number of data links composing each virtual link $VL_{m,n}^{(k)}$
GN_i	Dependency sub graph of the i-th network node
GL_i	Dependency sub graph of the i-th network link
E_N	Set of edges inside a node's subgraph
E_L	Set of edges of a link subgraph
E_{VL}	Set of edges between the physical resources and a virtual link
E_S	Set of edges between a virtual resource and a networking service
E_{VNF}	Set of edges between a VNFI and a VNF

Chapter 1 Introduction

1.1 Research context

The advent of programmable networks with SDN (Software-Defined Networking) and NFV (Network Functions Virtualization) is accelerating faster and faster the transformation of current networks towards elastic, on-demand and flexible usage. SDN and NFV are two novel phenomena that are on the wish list of major industrial players (vendors, operators, content providers, software editors) as the means to achieve greater flexibility in managing the network, faster service deployment and provisioning while reducing operational costs. SDN and NFV are thought to be "better together" as considered by the IT and telecommunication industries, due to the incredible synergy coming from the combination of both paradigms. SDN proposes to transition from network configurability to network programmability through network abstractions, open interfaces and the separation of control and data plane. NFV proposes to virtualize network functions with two goals: firstly to remove the vendor lock-in barrier and secondly allow networking services to be flexibly instantiated and scaled according to traffic demands.

However, the flexibility and elasticity in combined SDN and NFV infrastructures is a doubled-edge sword, mainly due to paramount need to rethink network and service management and operations, in especial fault management operations (detection, diagnosis, and recovery). Diagnosis in particular must be sufficiently intelligent, automated and rapid to enable the SDN and NFV promises and fully exploit the advantages from their synergy. Indeed, SDN and NFV are still in a preliminary stage concerning diagnostics aspects, evidenced on the state of the art in both SDN and NFV.

In addition, the dynamicity of virtualized resources in combined SDN and NFV infrastructures complicate the already complex today diagnosis tasks. Networking services are composed of virtualized network functions (VNF) and those VNFs can be migrated across the infrastructure by making an elastic usage of the compute, storage and networking resources.

The high dynamicity of the SDN infrastructure–topological changes at both control and data planes and rapid forwarding changes through flows–becomes even higher when we combine SDN with NFV. This is because the networking services rely on run-time configurable VNFs, which can be scaled, instantiated, deleted, and migrated. This dynamicity urgently calls for an efficient, automated, fast and intelligent diagnosis by automating the modelling the networking services, their virtual resources, and the physical infrastructure.

The aim of this thesis is to provide with a self-diagnosis framework able to diagnose in an automatic manner such dynamic networking services based on fully virtualized network functions over SDN and NFV combined infrastructures.

1.2 Software-Defined Networking

Software Defined Networking (SDN) paves the way towards network programmability by proposing network architecture based on abstraction, open interfaces, and control plane-data plane separation. Many definitions surround the SDN concept, all of them centered on the abstraction and network programmability. ONF (Open

Networking Foundation) defines SDN as *"The physical separation of the network control plane from the forwarding plane, where the control plane controls several devices"* (ONF, 2011).

Network programmability: SDN introduces the programmability of the network behavior through network applications. Programmability is an enabler that can enhance network elements with the ability to change and to accept a new configuration, which modifies their behaviors in response to changes in the network state. In fact, programmability conveyed by SDN can achieve a logical centralized control of the network through the control plane for a group of network elements. The control plane exposes an Application Programming Interface (API), which abstracts the complexity and thus hides unnecessary inner details belonging to underlying layers. The SDN applications program the control plane through APIs. There are several examples of APIs such as the REST API (Representational State Transfer). In fact, each SDN controller defines its own API with its own set of functions and properties.

Abstraction: As defined in the survey on SDN in (Kreutz et al, 2015), there are three types of abstraction in SDN such as forwarding abstraction, distribution abstraction, and specification abstraction. Firstly, **the forwarding abstraction** consists of separating data and control plane. This separation consists in adopting a higher-level namespace and exploiting a logically centralized controller to enforce the network policies by communicating them to generic forwarding hardware (via the southbound interface) in terms of language rather than technology-specific encoding. The separation between control and data planes means that the control plane, which contains the SDN controller, decides on behalf of the data plane resources. From an OpenFlow perspective, the SDN controller decides how to forward packets across the data plane elements by sending a set of flows. Secondly, **the distribution abstraction** consists of embedding logically the intelligence in the control plane. The control plane is composed of one SDN controllers and there are two types of control plane: a distributed control plane, where several SDN controllers take the control of the network, or a centralized control plane, only composed of one SDN controller. In both cases, the SDN controller necessarily becomes a single point of failure as it is in charge of providing with the instructions sent by the network applications to the data plane. Finally, **the specification abstraction** consists of specifying the network behavior from network applications running on the application plane through commands sent to the control plane, which are in turn translated into low-level specific commands delivered to the data plane via to the southbound protocol.

1.3 Networks Function Virtualization (NFV)

NFV is a networking initiative led by Telcos (Guerzoni et al, 2012), to replace Network Functions by virtualized network functions usually implemented as software embedded in commodity hardware (high-volume standard servers, storage and switches).

Nowadays, most network functions are sophisticated, expensive and dedicated customized hardware provided by vendors. Each vendor provides with a different network function with its own set of management tools. Due to the heterogeneity of these network functions is very costly for Telcos to integrate with the rest of their equipment. Network functions can carry out multiple and different types of operations such as firewall, load balancing, cipher, TCP (Transmission Control Protocol) accelerators, concentrators, DNS (Domain Name System), QoE (Quality of Experience) Management, or video optimizers, EPC (Evolved Packet Core), among others.

This approach reduces power consumption, maintenance costs and time to deploy new functions/services. This solution allows us to both remove the vendor lock-in barrier and networking services to be flexibly instantiated and scaled according to network traffic demands at run-time by making an elastic usage of the compute, storage and networking resources.

1.4 Research problem: Why do we need automation on diagnosis of networking services over programmable infrastructures?

SDN propose as main features network programmability, abstraction, and logical centralization. However, those features are going to compel network operations such as FCAPS (Fault, Configuration, Accounting, Performance, Security) to be rethought, especially fault management operations.

The logical centralization of the intelligence inside the control plane makes resilience even more crucial because the entire SDN infrastructure depends on the elements of the control plane, what leads to the paramount need to empower the SDN architecture with resilience properties. Resilience properties are paramount because malfunctions at the control layer propagate to the data layer. As consequence from this centralization, the control plane becomes target of multiple types of attacks, potential resource conflicts when SDN applications allocate resources simultaneously, and the lack of scalability of the SDN controller to handle flow requests. To this end, **fault management approaches are to ensure resilience at the control and the data plane in SDN infrastructures by means of self-healing mechanisms able to detect, diagnose and recover any type of malfunction.**

The thesis focuses on the combination of SDN and NFV, which makes the network infrastructure and resources even more dynamic. The focus of this thesis is self-diagnosis, in charge of automatically finding the origin of a given malfunction by analyzing a set of symptoms, procedure known as root cause analysis. We resort to model-based self-diagnosis approaches, which perform the root cause analysis by first generating a fault propagation model and then exploiting to find the root cause. In SDN and NFV combined infrastructures, this model will be multi-layered covering physical, logical, and virtual resources composing each networking service.

However, the high degree of dynamicity in SDN infrastructures resulted of changes on the network topology, the type of control, and the flow-based forwarding influences the way faults and failures propagate among network resources. **The way faults and failures propagate are explained by dependencies established among SDN resources which depends on how those resources are connected, interact and exchange both control and data information. However, due to the dynamicity of the network topology, type of control and flows, those dependencies change fast and continuously, what makes mandatory to establish a self-modeling methodology to automatically generate in an online manner and update this fault propagation model at run-time to diagnose in an automatic, flexible, and effective way SDN infrastructures.**

On the other hand, in NFV solutions VNFs can be migrated, scaled, duplicated, among other operations, which make network services and their virtual resources highly dynamic and elastic. The SDN infrastructure dynamically allocates the virtual links to connect the VNFs through flows sent to from the control to the data plane and those can be rapidly migrated or modified. The high degree of dynamicity of virtual resources inn combined SDN and NFV infrastructures influences the way faults and failures propagate among the physical, logical, and virtual and services layers. **The way faults and failures propagate are explained by the dependencies established among the networking services and the virtualized resources. However, due to the high dynamicity of the virtual resources, those dependencies change fast and continuously, what makes mandatory to establish a self-modeling methodology to automatically generate in an online manner and update this fault propagation model at run-time to diagnose in an automatic, flexible, and effective way SDN and NFV combined infrastructures.**

1.5 Motivating example

As example, we show in Figure 1 two different end-to-end services delivered in a SDN and NFV combined infrastructure, where each service is composed of a different set of VNFs. In the presence of hardware faults or software faults affecting several VNFs in the blue service, those can be migrated to other physical locations to avoid any outage in the service. This change of location implies to re-establish the virtual links interconnecting the VNFs by sending a request to the SDN controller, which in turn, installs new flows on the nodes to ensure this communication.

In addition, the network topology is already dynamic in the radio access due to several reasons like the connections and disconnections of users to the Access Points (AP), or handovers that transfer users among APs. However, in SDN and NFV, the network topology becomes much more dynamic, especially due to the aforementioned dynamicity of both virtual and network resources.

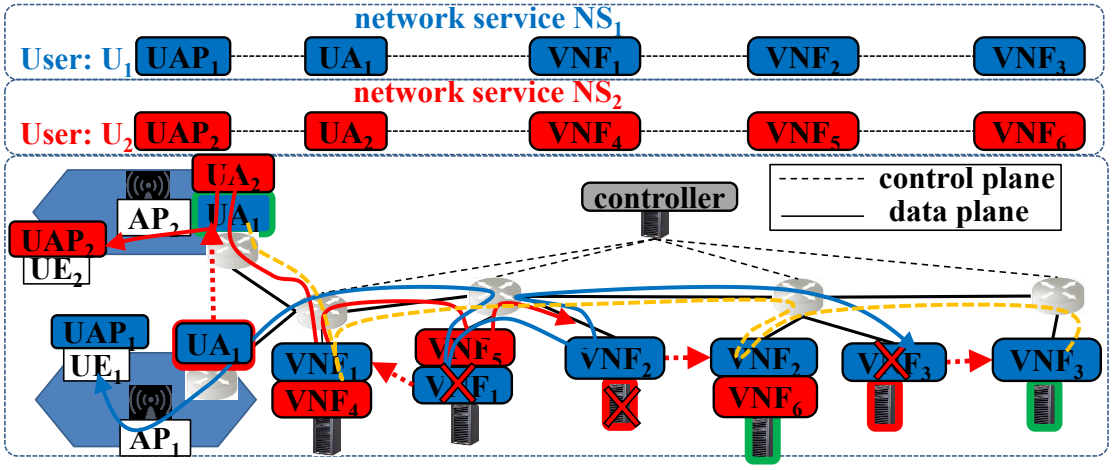


Figure 1. Service topology changes in NFV-SDN infrastructures

Challenge 1: Network dynamicity

In SDN, the network infrastructure becomes dynamic because of the changes in the network topology, composed of the control and data networks. This dynamicity is due to the continuous connections and disconnections of nodes, discovered by the SDN controller at run-time, but also because of changes in the type of control, which change the way the SDN controller interacts with the data plane's elements.

The challenge here is to generate the fault propagation model containing the dependencies among the nodes in the infrastructure in an online and automatic manner. A self-modeling methodology generates this model. This model must contain two components:

- the network topology, to model how SDN resources are connected, and
- the type of control, to model how the SDN controller is connected with the data plane's resources.

Indeed, the network infrastructure and the type of control are dynamic, adding a first dimensionality of dynamicity that a self-diagnosis mechanism must consider to diagnose those infrastructures.

Challenge 2: Virtual resources dynamicity

In NFV, virtual resources composing network services become dynamic because VNFs can be moved, duplicated or migrated to different physical locations and the virtual links connecting the VNFs are dynamically set by the control plane of SDN infrastructure through flows, which can be migrated and modified by the SDN controller at any time.

The challenge here is to generate the fault propagation model containing dependencies of several networking services from its corresponding virtualized resources. A self-modeling methodology generates this model. However, this model must also include the dependencies of those virtualized resources from the physical substrate. Concretely, this fault propagation model must contain the dependencies between a network service and its virtualized resources (VNFs and VLs), the dependencies between VNFs from their physical locations in the network infrastructure (chosen by the VIM (Virtualization Infrastructure Manager)), and the dependencies between the VLs from their physical resources in the network topology (chosen by the SDN controller).

Indeed, virtual links are allocated over the network infrastructure through flows sent by the control plane. The network infrastructure is already dynamic, adding a second dimensionality of dynamicity that a self-diagnosis mechanism must consider to diagnose those infrastructures.

The main challenge comes when both dynamicity dimensions come together in SDN and NFV combined infrastructures, where the network infrastructure may change, but at the same time, the virtual resources deployed over such a dynamic infrastructure.

1.6 Thesis objectives and principles

The aim of this thesis is to define a self-diagnosis framework to ensure resilience of end-to-end network services in SDN and NFV combined infrastructures. However, to diagnose those highly changing and dynamic infrastructures, we need to establish a self-modeling methodology that tracks the dynamic network topology, the type of control, the virtualized network resources involved in the networking services and flows. This self-modeling methodology generates and updates a fault propagation model containing the dependencies among those dynamically deployed resources. We distinguish two levels of dynamicity on combined SDN and NFV infrastructures, and, as a result, two challenges to generate this fault propagation model, coming from the high dynamicity of physical, logical, and virtual resources in SDN and NFV combined infrastructures.

An automated and flexible self-diagnosis requires being as flexible and dynamic as SDN and NFV infrastructures, that is why the self-diagnosis framework conceived in this thesis should be based on a self-modeling approach that generates a fault propagation model. From our perspective, this fault propagation model generation should fulfil the following requirements in order to automate the diagnosis in programmable networks with finer precision:

Automated: The self-diagnosis framework should be a completely automated process where the way faults and failures propagate are automatically generated from the information of the network infrastructure, the services deployed as well as their logical and virtual resources sharing or not resources.

Multi-layered: The self-diagnosis framework should be based on a multi-layered fault propagation model comprising the multi-level dependencies among physical, logical, virtual, and service layers in accordance with the layers in SDN and NFV combined infrastructures.

Updatable: The self-diagnosis framework should be based on an automatically update fault propagation model in accordance with the topological changes, the type of control, the forwarding flows, changes on the locations of the VNFs or in the interconnecting virtual links.

Fast: The self-diagnosis framework should be able to diagnose the root cause in a reasonable time, taking into account that the infrastructure and deployed virtual resources may change in a fast manner.

Extensible: The self-diagnosis framework should be based on a fault propagation model generated in such a way that new discovered elements can be added when the topology changes, or when the networking service adds a new virtual resource.

Fine-grained: The self-diagnosis framework should be based on a fault propagation model that contains not only the dependencies among SDN resources but also the dependencies among the internal resources or components inside each SDN resource such as cards, ports, software, among others to allow the detection of specific root causes inside nodes.

1.7 Research Questions

In this section, we decompose the previous problem statement into four main research questions (RQ), each of them leading to different contributions.

RQ 1: How to conceive a self-healing mechanism for an SDN and NFV combined infrastructure?

Contribution 1: State of the art on SDN and NFV

Contribution 2: State of the art on self-healing systems

Contribution 3: Self-Healing mechanism for SDN and NFV

RQ 2: What mechanisms allow for an automated and dynamic diagnosis of the root cause in multi-layered networks such as SDN and NFV?

Contribution 4: State of the art on diagnosis mechanisms

RQ 3: What self-modeling methodology can we propose to generate the diagnosis model for centralized SDN infrastructures that evolve over time and exploit this model for finding the root cause?

Contribution 5: Definition of multi-layered, fine-granular, machine-readable, extendable templates containing the resources to supervise at physical and logical layers.

Contribution 6: Proposal of a self-modeling approach to generate the diagnosis model for dynamic network topologies

Contribution 7: Conception of a topology-aware self-diagnosis module. This module takes into account the logical and physical resources in dynamic network topologies.

RQ 4: What self-modelling methodology can we propose to generate the diagnosis model of a centralized SDN infrastructure ensuring NFV-based network services where both, the network topology and virtual resources evolve over time, and exploit this model for finding the root cause?

Contribution 8: Extension of the proposed templates to include the virtual and service layers.

Contribution 9: Conception of a self-modeling module that takes as input these extended templates, instantiates them and generates on-the-fly the diagnosis model that includes the physical, logical, and the virtual dependencies of networking services in combined SDN and NFV infrastructures.

Contribution 10: Conception of a service-aware self-diagnosis module. This module takes into account the networking service view and the state of the underlying network resources.

1.7.1 Research methodology and scientific contributions

In this section, we first detail the adopted research methodology and then we describe the scientific contributions. We adopted the following research methodology along the thesis:

To resolve the first research question RQ 1, we first surveyed the state of the art on SDN and NFV. This step is paramount to identify and understand the potential challenges faced in a combined SDN and NFV scenario, in contrast to traditional networks. ***The first contribution*** of this thesis is the identification of the challenges of SDN and NFV infrastructures. The most prominent challenges found were the centralization of the SDN controller and the high dynamicity of the virtualized and physical resources. We also studied self-healing systems, as the identified autonomic mechanism to counter the identified vulnerabilities in SDN and NFV in an automatic and in an intelligent manner. ***The second contribution*** of the thesis is the identification of the architecture, functional blocks, and the techniques used in each functional block of a self-healing system. We also analyzed which inputs could a self-healing mechanism take in a SDN and NFV context, taking into account those infrastructures and their associated challenges. ***The third contribution*** of the thesis is a high-level self-healing architecture to counter the identified challenges of SDN and NFV. This self-healing architecture is based on a multi-layered control-loop. This control-loop includes different recovery procedures to be carried out in the different planes of the SDN architecture. We proposed a set of several metrics in Chapter 3 for SDN infrastructures as input for this self-healing architecture.

To resolve the second question RQ 2, we centered our research on the self-diagnosis task. The ***fourth contribution*** of this thesis is an extensive review of the literature on diagnosis algorithms and the identification of Bayesian Networks algorithm as the most used across different network technologies. This algorithm diagnoses the network by identifying how faults propagate in a given network infrastructure through a fault propagation model or dependency graph. This type of self-diagnosis is model-based because it identifies the root cause by exploring how faults propagate through a diagnosis model.

However, the main limitation is how to automate the generation of the diagnosis model for modelling dynamic networks such as SDN and NFV. As a result, we then centered our efforts on providing with a self-modeling methodology capable of generating but also updating this diagnosis model with perspective to apply it to SDN and NFV combined infrastructures. The ***fifth contribution*** of this thesis is the definition of multi-layered templates to identify what to supervise while taking into account the physical and logical layers. ***As sixth contribution*** in this thesis to address the **Challenge 1,** we propose a topology-aware self-modeling mechanism to automatically model dynamic network topologies and types of control by using a set of finer granularity templates that encompass the dependencies among SDN nodes (physical and logical) as well as smaller sub-components inside those nodes (e.g. CPU, network cards, etc.). Topology-aware self-modeling builds the dependency graph from the network topology and this graph only considers how faults in network resources could affect other network resources. We centered our efforts in understanding how to feed the generated model in the model-based diagnosis engine based on Bayesian Networks. As **seventh contribution**, we propose a topology-aware self-diagnosis module that takes into account the logical and physical resources in dynamic network topologies.

However, the dependency graph generated by our topology-aware self-modeling approach does not model the impact of faults in network resources on services. To this end, we solve the third research question RQ 3 that answers the second **Challenge 2.** ***As eight contribution*** of this thesis, we extended our proposed multi-layered templates to take into account the virtual and services layers. Thanks to these extended templates, we developed a Service-aware self-modeling approach, the ***ninth contribution*** of this thesis, and as an extension of the topology-aware approach to consider how faults in network resources affect each other but also how those resources affect the service layer. As ***tenth and final contribution*** in this thesis, we propose a service-aware self-modelling approach, which allows us to model network services and their dynamic dependencies with the underlying logical and physical resources in combined NFV and SDN infrastructures. A service-aware diagnosis reduces uncertainty by automatically extending or reducing the dependency graph according to the faulty networking service. Moreover, we propose two diagnosis strategies to reduce the high uncertainty obtained in the previous work, which we describe in chapter 4.

In addition, and as proof of concept of this thesis, we implemented a self-healing system to supervise a multicast video streaming service delivered in a dynamic network topology with several clients connected, where the network could be updated and the model regenerated. The self-healing block was located in the management plane. This self-healing framework could detect, diagnose and recover from link failures, traffic transport failures, and application failures by sending several reconfiguration commands to restore the service. This framework was presented at the *Orange research Exhibition* which took place the 1st, 2nd, and 3rd or December.

This thesis has led to five scientific articles, three full papers, a short paper, and demo paper. In addition, we have also submitted a demonstration paper and we plan to write a journal. The references of those accepted articles are the following:

- J. Sanchez, I. Grida Ben Yahia, N. Crespi, " THESARD: on The road to resiliencE in SoftwAre-defined networking thRough self-Diagnosis " 2nd IEEE Conference on Network Softwarization, Seoul, Korea, 6-10 June 2016.

- J. Sanchez, I. Grida Ben Yahia, N. Crespi, "Self-Modeling Based Diagnosis of Software-Defined Networks," Workshop MISSION 2015 at 1st IEEE Conference on Network Softwarization, London, 13-17 April 2015.

- J. Sanchez, I. Grida Ben Yahia, et. al., "Softwarized 5G networks resiliency with self-healing," 1st International Conference on 5G for Ubiquitous Connectivity (5GU), 2014.

- J. Sanchez, I. Grida Ben Yahia, N. Crespi, "Self-Modeling based Diagnosis of Services over Programmable Networks," 2nd IEEE Conference on Network Softwarization, Seoul, Korea, 6-10 June 2016.

- J. Sanchez, I. Grida Ben Yahia, N. Crespi, "Self-healing Mechanisms for Software Defined Networks". AIMS 2014, 30 June-3rd July 2014.

1.8 Project Contributions and presentations

Presentations on conferences

We presented three articles in international conferences:

1) We presented one article at the 1st International Conference on 5G for Ubiquitous Connectivity, which took place 26th and 27th of November in Levi, Finland. The article was entitled "Softwarized 5G Networks Resiliency with Self-healing". The content of this article is available in the annex.

2) We presented one article at the Workshop on Management Issues in Software-defined networks, Software-defined infrastructure and network function virtualization (MISSION 2015) located inside the 1st IEEE International Conference on Network Softwarization, which took place from 13th to 15th of April in London, England. The article was entitled "Self-Modeling based diagnosis of Software-Defined Networks". The content of this article is available in the annex.

3) We presented one article at the 2nd IEEE International Conference on Network Softwarization, which took place from the 6th to 10th of June in Seoul, South Korea. The article is entitled "Self-Modeling based Diagnosis of Services over Programmable Networks". **This article was awarded with the Best Student Paper award.** The content of this article is available in the annex.

We presented three posters in conferences and seminars:

1) We presented a poster at the « Journée des doctorants 2013 », entitled « Mécanismes d'autoréparation pour Software Defined Networking », which took place in Orange premises the 12th of Septembre of 2013.

2) We presented a poster at the Autonomous Infrastructure, Management and Security conference (AIMS2014), which is entitled « Self-healing Mechanisms for Software Defined Networks », which took place from the 30th of June to the 3rd of July in Brno, Czech Republic. The content of its corresponding article is available in the annex.

3) We presented a poster in the 'Journée Cloud 2015', entitled « Topology-Aware Self modeling for SDN-NFV diagnosis », which took place 14th September of 2015 in UPMC LIP6 premises. The program is available at: https://rsd-cloud.lip6.fr/journee.html.

We presented one demonstration at the 2nd IEEE International Conference on Network Softwarization, which took place from the 6th to 10th of June in Seoul, South Korea. The demonstration is entitled "THESARD: on The road to resiliencE in SoftwAre-defined networking thRough self-Diagnosis".The content of this article is available in the annex.

Invited talks

We gave a talk in the seminar 'Séminaire Francilien de Sûreté de Fonctionnement', entitled « Self-Diagnosis and Service Restoration in SDN and NFV infrastructures », which took place the 11th of March 2016 at the Université Scientifique de Versailles (UPVSQ). The program is available at: http://www.lurpa.ens-cachan.fr/version-francaise/manifestations/seminaire-francilien-de-surete-de-fonctionnement/.

Orange Research exhibition

Orange research exhibition is an annual event where the innovative research led inside Orange premises is shown in the shape of innovative platforms, demonstrations, and proofs of concepts in order to give an insight on Orange's current research lines and interests. These demonstrations are the outcome from European projects, French national projects or other types of collaborations. Last edition of the Orange research exhibition took place the 1st, 2nd, 3rd of December 2015 in Paris, where we prepared a demonstration on "Self-Diagnosis for Software-Defined Networks".

Project contributions to European projects

We have contributed to the FP7 European project Univerself during this thesis. This project lasted three years, it started in September 2010 and it ended in August 2013. The aim of this project was to overcome the growing management complexity of future networking systems, and to reduce the barriers that complexity and ossification pose to further growth by realizing autonomics for future networks.

Our concrete contribution to this project was to study autonomic principles, self-* properties, and the autonomic architectures such as the proposed by ETSI GANA.

Website: http://www.univerself-project.eu/

Project contributions to National projects

We have contributed to the French national project ANR REFELXION (REsilient and FLEXible Infrastructure for Open Networking) during this thesis. This project lasts two years, and it is still ongoing, it started in October 2014 and it will end in October 2016. The aim of this project is to bring (i) robustness and flexibility in NFV-SDN architectures, in particular to support critical services, and (ii) dynamicity and efficiency for the provisioning and the chaining of virtualized network functions.

Our concrete contribution to this project network failure diagnosis and fault management for NFV in SDN so that services can keep operating in a seamless way.

Website: http://anr-reflexion.telecom-paristech.fr/

1.9 Thesis organization

We organize this thesis in the following chapters:

The Chapter 2, entitled “programmable networks and fault management challenges”, contains a detailed state-of-the-art on fault management on programmable networks, which is the research context of this thesis. This chapter unveils the issues and challenges when it comes to ensuring fault management. It presents also the related work on fault management for SDN and NFV infrastructures.

Chapter 3, entitled “State of the art on self-healing systems”, contains a detailed state-of-the art on self-healing systems, which is the adopted solution in this thesis This chapter first describes the purpose of self-healing systems, their architecture, their functional tasks, and the techniques used. Later on, this chapter focuses on the self-diagnosis task, with especial emphasis on diagnosis algorithms.

Chapter 4, entitled “Self-Diagnosis architecture for programmable networks”, presents our proposal, a self-diagnosis framework for SDN and NFV combined infrastructures. First, we describe the self-diagnosis architecture and its functional blocks, and then, we describe the different algorithms composing this architecture are explained.

Chapter 5, entitled “Results and evaluation”, shows the results and evaluates performance of the results. We split this chapter in use cases, where each use case shows a different aspect covered by the self-diagnosis framework proposed in chapter 4.

Chapter 6, entitled “Conclusions and future work”, highlights the future work derived from these results and concludes this thesis manuscript.

Chapter 2 Programmable Networks and Fault Management Challenges

2.1 Introduction

We explain in this chapter the concept of programmable networks, the concepts of SDN and NFV and their respective architectures and give an insight on their associated fault management challenges and related work addressing those challenges. It also provides with an insight on the combination of SDN and NFV.

2.2 Overview of SDN

In this section, we describe briefly the SDN architecture, its functional blocks and its modes of operation.

2.2.1 SDN Architecture

SDN architecture is composed of four planes, namely the infrastructure layer (or data plane), the control layer, the application layer and the management plane, as shown in Figure 2

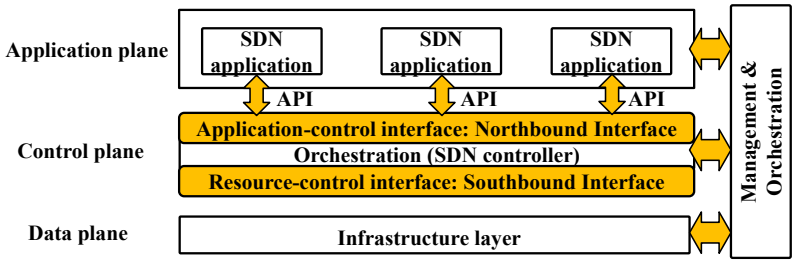


Figure 2. SDN layered architecture by ONF

The data plane is composed of the SDN resources belonging to the infrastructure layer, which are in charge transporting the traffic. The SDN resources located in the data plane are the OpenFlow switches, the hosts/servers acting as traffic sources and sinks and all the control and data links seen in the infrastructure.

The control plane mediates between the data plane and the application plane and it is based on a control software intelligence that relies on a logically centralized abstraction level. The control plane performs the orchestration of resources in the sense that dictates the forwarding rules to the data plane by means of the resource-control interfaces (a.k.a southbound interface). OpenFlow is the de facto southbound protocol to communicate the control plane and data plane (ONF, 2016). The control plane is composed of different SDN controllers, which use the southbound interface to communicate with their underlying OpenFlow switches belonging to the data plane. OpenFlow protocol formalizes the communication between the control plane and the data plane and it is based on the transmission of rules (a.k.a flows) sent by the OpenFlow controller to the switches.

The application plane is composed of SDN applications in charge of programming the network through Application Programming Interfaces (APIs). SDN applications act as clients of aforementioned exposed interfaces by the control plane, named application-control interfaces (a.k.a. northbound interface). The application plane dynamically programs the data plane through these APIs exposed by the control plane.

2.2.2 Forwarding in SDN OpenFlow

An Openflow switch is a forwarding device with one or several flow tables, as shown in Figure 3, where the flow tables are populated with flows entries that are sent by the control plane, concretely the SDN controller. These flow entries consist of match fields, counters, and a set of instructions or actions to apply for each matching incoming packet. Counters gather forwarding statistics, which are grouped per flow table, per flow, per port, or per queue. Those statistics give a precise idea of the percentage of flows matched, for instance.

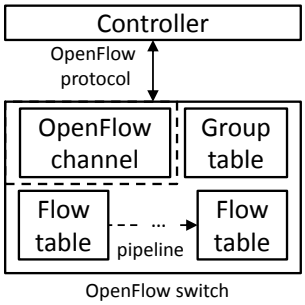


Figure 3. Main components of an OpenFlow switch by ONF

Each flow installed in any OpenFlow switch is composed of a set of tags (a.k.a. matching fields), some of them shown in Table 1. Each installed flow, remains installed inside the switch for a given amount of time, specified by two timers namely, the idle time and hard time. On one hand, the hard time parameter determines for how long the flow remains installed, despite this flow is matched or not. A null value of hard time indicates that the flow will not be deleted in any case, regardless if it matches or not incoming packets. On the other hand, the idle time parameter determines for how long the flow is installed as long it still matches any incoming packet. It no packet matches that flow, it will be deleted after this idle time. A null value of idle time indicates that a non-matching flow will not be deleted in any case.

Table 1. Tag fields defined in OpenFlow 1.0

Ingress port	MAC src	MAC dst	Eth type	VLAN id	VLAN priority	IP src	IP dst	IP protocol	IP ToS bits	TCP/UDP src port	TCP/UDP dst port

When a packet arrives to the OpenFlow switch, it is matched against the installed flows entries in its flow table. If the packet matches any flow, it applies the action associated to that matching flow, otherwise, additional flows are asked to the SDN controller to be installed to know what to do with that unmatched packet. OpenFlow defines several actions, some of them shown in Table 2.

Table 2. Some examples of actions in OpenFlow 1.0

Action	Sub-action	Explanation
FORWARD	ALL	It sends the packet to all the output ports
	IN_PORT	It sends the packet to the incoming port
	CONTROLLER	It sends the packet to the SDN controller
ENQUEUE		It forwards the packet to a given queue attached to an output port. This is mainly to provide with basic QoS
DROP		It drops the incoming packet if the matching rule includes no action

Packet matching is based on priorities, where each flow is installed by the SDN controller with a given priority. For instance, when the packet arrives to the incoming port of a switch, it is matched against the flows and in the case, there are duplicated flows, the matching flow chosen is the one with the highest priority. The maximum value of priority defined in OpenFlow is 665535 (2^16-1 bits), but there are exceptions where this maximum value is exceed, as it will be seen later.

The forwarding behavior of switches can be dictated by the SDN controller in a proactive manner (proactive forwarding), where the flows are installed by the controller in advance, or in a reactive manner a.k.a reactive forwarding, where the controller replies to the queries made by the switches. In proactive forwarding, the SDN controller installs the flows in a proactive manner, before those are asked by the switches. This type of forwarding can be applied when we need to preallocate the path in advance, but this is not always possible. The advantage in this type of forwarding is that the SDN controller does not have to answer any new request and does not add any unknown and uncertain delay like in reactive installation mode. Also, the control plane is less congested than in reactive rule installation mode. There are techniques that preinstall alternatives routes in the switches in advance as a backup in the presence of link faults. In reactive forwarding, the SDN controller installs the flows into the network elements by request of the switches. When a network element does not know what to do with the incoming packets, it sends a copy of the incoming packet to the controller. Then, the SDN controller responds to this request after a given delay by installing the proper flow with the corresponding action to apply to this packet. A conversation between the controller and the switch takes place before installing the rules on the switch. In that dialog, the controller receives a Packet_IN message from the switch and it returns certain amount of Packet_OUT messages in exchange. Reactive means that, when the switches do not know how to forward the packets (they do not match any already installed flow), they send a PACKET_IN to the controller to get the appropriate flow installed.

We show one example of reactive forwarding in Figure 4, which is the most used in SDN infrastructures. The SDN controller is connected to one OpenFlow switch and this one is in turn connected to four different hosts. The OpenFlow switch is running a software OpenFlow client application to communicate to the SDN controller application. Host H_4 (which IP adress is 1.2.3.4) sends a packet towards host H_1 (which IP address is 5.6.7.8) through this intermediate switch. The switch receives this packet and tries to match it with its installed flows, however, none of those flows matches that packet so the switch does not know how to treat this packet and it queries the controller, by sending a Packet_IN notification to request the actions to apply to that incoming packet. In request, the controller sends back a Packet_OUT notification packet and installs a new flow on the switch's flow table to forward all packets coming from host H_4. This flow indicates that the packets coming from host H_4 must be forwarded through the port 1. The rests of the fields in that flow are asterisks, which means that those fields will not be taken into account in the matching of incoming packets i.e. incoming packets can have any value on those fields.

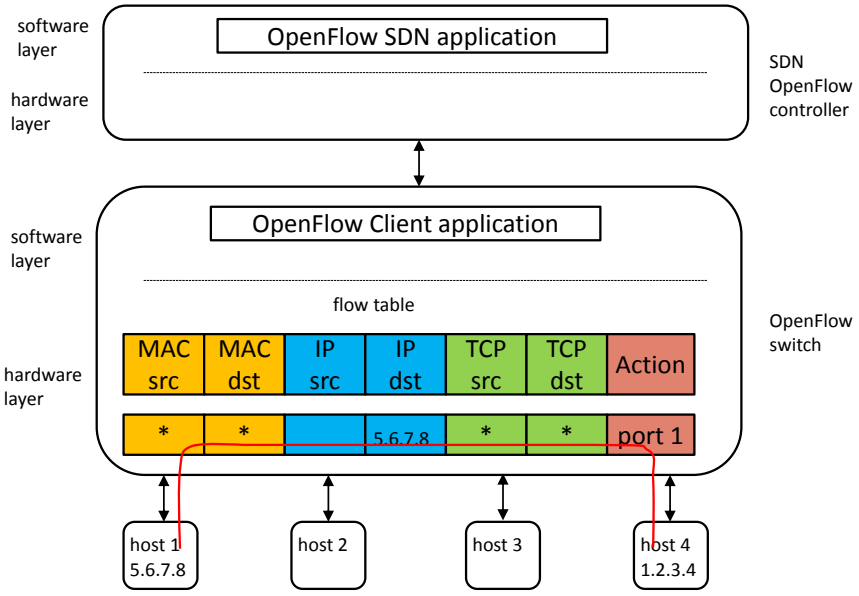


Figure 4. SDN forwarding in OpenFlow

2.2.3 Types of control in SDN

In this section we define how control and data plane interact. This interaction is different in accordance with the type of control led by the control plane. First, we define both types of control and then we analyze the main differences between both types of control.

An SDN infrastructure is generally composed of the several controllers. Each controller's domain is composed of a different network topology. Each network topology is composed of the SDN resources such as hosts, switches and links. There are two types of control in SDN infrastructures, in-band and out-of-band. The control network is composed of the SDN controller, the control links (a.k.a control-to-data links) and the control ports located at the switches (Figure 5 in red) to connect to the SDN controller, whilst the data network is composed of the network elements in the infrastructure layer such as data links, switches and hosts (Figure 5 in blue).

2.2.3.1 In-band control

In in-band control, both control and data network overlap as seen in Figure 5 (a). Links transport both data traffic and control traffic (Figure 5 (a)) so there is not notion of control or data link as both are intertwined. The SDN controller is only directly connected the master switch via the control-to-data interface (S1 in Figure 5 (a)). The rest of switches are *slave switches* (S2, S3 and S4 in Figure 5 (a)) and receive the flows via the master switch S1.

The control traffic is transported in an OpenFlow network, which is implemented by installing flows with higher priorities than the maximum value admissible on the switches at the beginning, where the SDN controller becomes aware of their presence (proactive forwarding). The goal of these flows is to ensure that the rest of flows sent by the SDN controller, which are sent by request of the switches (reactive forwarding), will always reach the switches e.g. the master and the slaves. The priority of these flows proactively installed must be always higher than the maximum admissible value to avoid any modification of these flows. A modification on these flows would have as consequence that the reactive flows would not be installed on the switches. For instance, in Figure 5, those flows ensure that:

- the switch S1 redirects the flows sent by the SDN controller towards the switch S2
- the switch S2 redirects the flows sent by the switch S1 towards the switch S3
- the switch S3 redirects the flows sent by the switch S2 towards the switch S4

Although, in principle, in-band control can be a bit misleading, as control and data traffic is intertwined in the same links despite the principle of separating control and data planes on SDN, the model of control remains the

same as well as the centralized principle, so we can conclude that the type of control is more a question of implementation. However, the type of control influences largely several aspects of a SDN infrastructure.

2.2.3.2 Out-of-band control

In out-of-band, the control network and the data network are clearly separated as seen in Figure 5 (b). Out-of-band control represents better the centralization philosophy of SDN, where control traffic is transported apart from data traffic in a dedicated control network (Figure 5 (b) in red). The SDN controller is connected to all the switches under its control, what we call controller's domain. In this type of control is the most extended in the SDN literature, where all the switches have a specific port to connect to the SDN controller through the control-to-data interface (or control link).

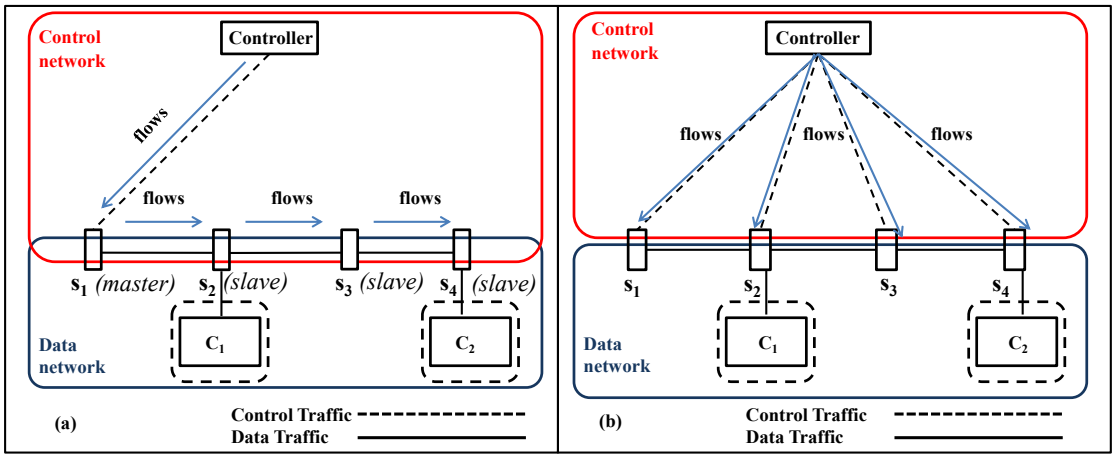


Figure 5. Example of flow installation: (a) In-band control and (b) out-of-band control

2.2.4 Influence of the type of control

We analyze hereafter the influence of both types of control on the following aspects: fault propagation, centralization, control traffic load, and flow installation delay. Among the few works in the literature on SDN that mention the type of control, such as in (Panda et al, 2013), (Sharma et al, 2013), and (Behesti & Zhang, 2012). Those works focus on one type of control, but do not detail their differences a great extent. We focus on centralized SDN infrastructures, composed of one controller, and we detail of the differences in both types of control and its impact on other aspects not treated so far in the state of the art.

In general, it is often said in the literature of SDN that both types of control mostly differ in cost and resilience. On one hand, they differ in terms of cost because in-band control is less expensive to implement than out-of-band control because, in out-of-band control, every switch must be provided with an additional control port to connect to the SDN controller, while in in-band control, only the master switch must be provided with that additional control port. On the other hand, they differ in terms of resilience because in-band control is less resilient than out-of-band control as both types of traffic are separated in out-of-band. On the contrary, the work done by Panda et. al. for distributed SDN infrastructures, call this into question. These authors investigated the enforcement of network policies in distributed SDN infrastructures under out-of-band control. The authors analyzed how the partitioning of the control plane in distributed infrastructures constrained the enforcement of network policies. The partition of the control plane consists of the following: each controller as the view of a given subset of the network topology because a controller can only query the switches in its domain and hence the hosts connected to those switches.

2.2.4.1 Fault propagation

From a fault propagation perspective, in-band and out-of-band are quite different from each other. Faults on links in in-band control affect both control and data traffic, whilst in out-of-band control, links are separated in control and data links, so faults on data links do not affect the control traffic and vice versa.

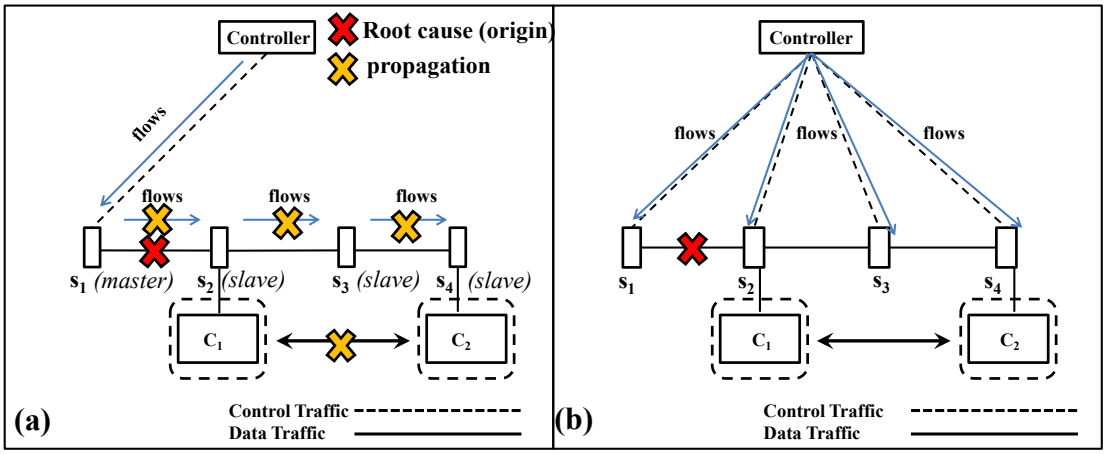


Figure 6. Fault propagation differences: (a) In-band control and (b) out-of-band control

This can be seen by a simple example in a linear network topology with 4 switches (Figure 6), connecting a client (C_1) with a server (C_2), where C_1 is attached to S_2 and C_2 is attached to S_4.

In out-of-band control (Figure 6 (b)), a faulty link between switches S_1 and S_2 will only affect the communication between both switches, but it does not affect the communication between clients C_1 and C_2. Indeed, this fault does not prevent the switches (S_1, S_2, S_3, S_4) from receiving flows through their respective dedicated control links.

However, in in-band control, the same faulty link, has much worse consequences than in out-of-band control, as this links transports both control and data traffic. The first consequence is that none of the slave switches (S_2, S_3, S_4) will be able to receive the corresponding flows and the network will become completely inoperative, because those three switches will not be able to know what to do with the incoming packets from C_1 and C_2, interrupting completely the communication between both clients. This simple example confirms that in-band control is less isolated and less resilient than out-of-band control for the same given network topology. We can conclude that an in-band controlled network has two points of failure, the SDN controller but also the master switch, while in an out-of-band controlled network, the SDN controller is the only point of failure.

2.2.4.2 Centralization

When the degree of centralization of the SDN controller is high, and the SDN controller has to control so many switches, their number of requests to the SDN controller can eventually congest the control links.

We can get an idea on how both types of control impact the scalability of centralized SDN infrastructures by considering the degree of centralization of network elements. The degree of centralization is a metrics that depicts the number of connections seen by each network element in the network topology, including the elements in the control network and in the data network. As example, we consider a linear topology with n network elements (Figure 7), when the hosts attached to the switches are omitted for the sake of clarity.

In in-band control (Figure 7 (a)) the degree of centralization of the SDN controller is always one because the SDN controller is only directly connected to the master switch. This means that the degree of centralization of the SDN controller is independent of the number of elements for any network topology.

In out-of-band control (Figure 7 (b)) the degree of centralization of the SDN controller is n and it changes with the number of switches attached to the SDN controller. We can see that in this network topology example, the degree of centralization of the switches is quite similar; however, this parameter depends on the number of hosts attached to each switch.

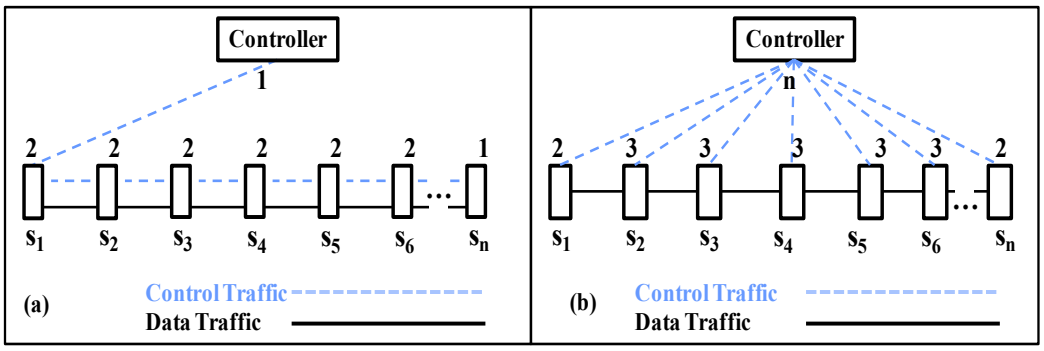


Figure 7. Degree of centralization: (a) in-band, (b) out-of-band control

Basil et. al in (Basil et al, 2015) define a metrics called control session capacity that measures the maximum number of control sessions that the SDN controller can maintain with each SDN node, (or OpenFlow switch). Nevertheless, the control session capacity will not be influenced by the type of control because the number of switches managed by the SDN controller is the same on both types of control. However, this metrics is affected when the number of switches grows.

2.2.4.3 Control traffic load

We first analyze the control traffic load in both types of control. In out-of-band control, the control traffic from each switch is directly sent to the SDN controller through the dedicated control links via the four ports attached to the SDN controller. However, in in-band control, there is only one control link to the SDN controller, so the control link may become congested more easily because it contains the aggregated control traffic from all switches towards the SDN controller. We illustrate these differences in Figure 8, where all switches send one Packet_IN simultaneously to the SDN controller to request a flow.

In out-of-band control, Figure 8 (b), the SDN controller receives four Packet_IN from different control links. As the SDN controller only has four ports, each Packet_IN is receive via each port. However, in in-band control, Figure 8 (a), the SDN controller receives the same amount of Packet_IN packets aggregated in the same control link, which is in turn connected to the only port of the controller, which may risk of congestion

In conclusion, given the same request rate per switch (one Packet_IN), in-band control tends to congest more easily the control link than in out-of-band control, because the number of Packet_IN is four times bigger in in-band control than in out-of-band.

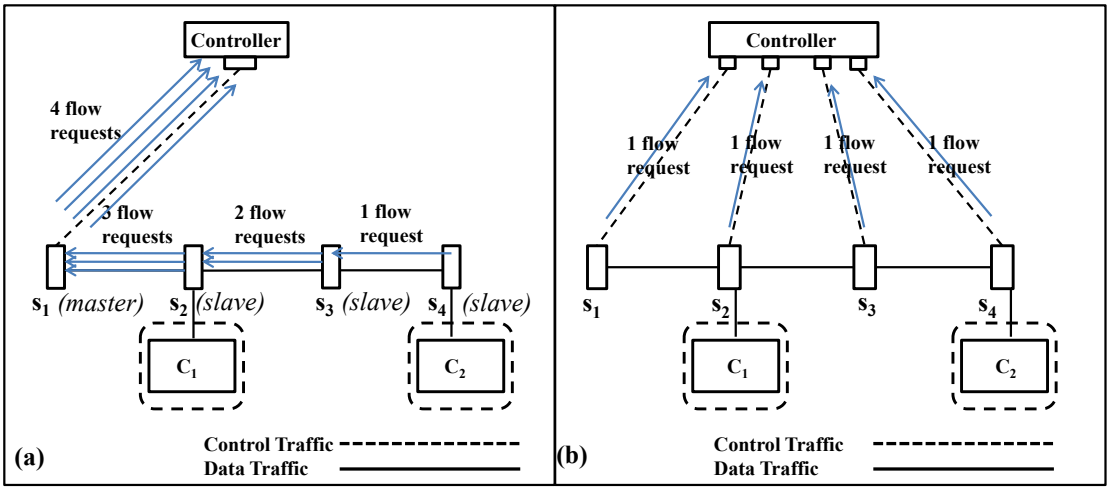


Figure 8. Control link congestion: (a) in-band, (b) out-of-band control

We analyze the traffic load generated by the control traffic on the rest of links in both types of control.

In out-of-band control, the data traffic and control traffic are separated, so the rest of links (called in this type of control data links) will not be charged with any control traffic. However, in in-band control, control and data traffic are intertwined. The control traffic from the slave switches will be retransmitted to the rest of switches until the controller. This can see in the example of Figure 8 (b) where switch S_4 sends a Packet_IN towards the controller, which in turn will traverse first the switches S_3, S_2, S_1 until it reaches the SDN controller.

Given the same request rate per switch (one Packet_IN), in-band control tends to congest more easily the rest of links than in out-of-band control. This phenomenon appears because in in-band control 6 Packet_INs are generated (3+2+1) in the links among the switches, while in the out-of-band control, no control packets are generated (these packet were directly transmitted through the control links).

The control traffic load towards the SDN controller will influence the control session capacity that measured the maximum number of sessions that the SDN controller can maintain. In addition, the forwarding table capacity in the SDN controller will have great importance to determine which level of control traffic congests the SDN controller.

2.2.4.4 Flow installation delay

We analyze here how the type of control could influence the delay of the flows installed by the SDN controller on the switches. The same type of analysis can be done to measure the delay in the other way round, to measure the delay in the flows requests sent by the switches to the controller. We assume that in in-band control the control links have a uniform delay D_C and the rest of links have a uniform delay D. In out-of-band control the control links have a uniform delay D_C and the data links have also a uniform delay D.

In in-band control, the flow installation delay D_c is lower at the master switch because it is directly connected to the controller through the control link. However, this flow delay increases in slave switches (by a factor D per hop) depending on the number of hops of distance to the controller, which depends on the considered network topology. Contrariwise, in out-of-band control, each switch is directly connected to the SDN controller by one control link, which delay is D_c, which does not vary largely.

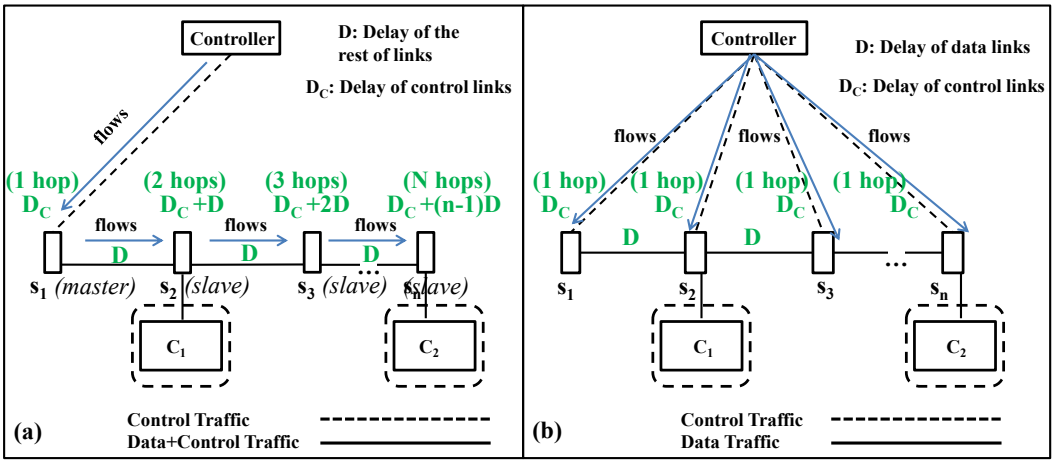


Figure 9. Flow installation delay: (a) in-band, (b) out-of-band

We illustrate these differences in Figure 9, where two linear network topologies with n=25 switches, one under in-band control and its equivalent topology under out-of-band control. We consider a delay on control links D_c=10 ms and in data links D=20 ms. The flow installation delay in in-band control increases with the number of hops separating each switch with the controller. Indeed, if we compare the delay at the nth switch S_n (the most distant element from the SDN controller) in both types of control, it should be close to D_n=(n-1)D+D_c=490 ms in in-band control and D_n=D_c =10 ms in out-of-band control. Nevertheless, the flow installation delay in out-of-band control does not increase and should remain stable around D_c in this example.

The type of control influences in a great extent the flow installation delay. Indeed, this extreme example gives an idea how important is to consider a relatively low number of switches per SDN controller domain, otherwise, the number of controllers should be increased to reduce this flow delay.

Basil et. al in (Basil et al, 2015) define two metrics called reactive path provisioning time and proactive provisioning time, where both are defined as the time taken to the SDN controller to establish a path between a given source and a destination node. It is clear that both reactive and proactive path provisioning times will be influenced by the type of control as seen in this section, as the path provisioning time is the addition of the different flow installation delays.

As a conclusion, due to the differences seen between both types of control in terms and its influence on aspects such as fault propagation, centralization, control traffic load, and flow installation delay, is a fundamental reason to consider the type of control when modelling an SDN infrastructure with the network topology to understand the behavior of an SDN infrastructure.

2.3 Related work on Fault Management in SDN

In this section, we analyze the related work on fault management for SDN. We have classified those fault management challenges and the respective related work in accordance with the different planes of SDN.

In general, we have seen that there is a prominent lack of multi-layer fault management solutions covering all the planes of the SDN architecture e.g. data plane, control plane, and application plane. Indeed, most of the existing fault management solutions for SDN only handle physical faults in the data layer but only a few solutions focus on the control plane, where, although the control traffic only represents less than 1% of the traffic volume, more than 95% of errors are due to the control plane itself.

2.3.1 Fault management solutions for the data plane

Most solutions for fault management in SDN mainly propose traffic engineering recovery solutions that reconfigure the data plane by providing alternative paths to avoid the affected nodes (traffic engineering solutions).

For instance, (Sharma et al, 2013) proposed a fast failure recovery technique exclusively for centralized SDN infrastructures under in-band control. On this article, the authors provide with a traffic restoration scheme that allows the SDN controller to circumvent failures on certain links by proactively sending a set of protection paths that it will utilize in case of failure. Similarly, (Behesti & Zhang, 2012) provide with a resiliency mechanism for in-band controlled SDN infrastructures. This mechanism is based on a resiliency protection metric that allows each switch to protect itself with an alternative path towards the SDN controller.

Turchetti and Duaerte in propose a failure detector, called NFV-FD, which detects faults on the data links of a SDN infrastructure. The NFV-FD block is implemented as a SDN application that communicates with the SDN controller through the REST API so it can detect the data links through an OpenFlow SDN controller. However, the authors assume that the SDN controller does not crash and only focus on faults in the data links, omitting the control links.

Gheorghe et. al. propose SDN-RADAR, a multi-agent distributed network troubleshooting mechanism for SDN that identifies faulty network links impacting user experience. This is a SDN application running on top of the SDN controller, which is running at run-time, and it is intended to support human administrators in charge of performing troubleshooting in SDN.

2.3.2 Fault management solutions for the control plane

The SDN approach logically centralizes all the intelligence in the controller, which becomes a single point of failure. Indeed, the controller, centralized or distributed, performs all the intelligent tasks, which implies problems of security, scalability and resiliency. Indeed, fault management solutions are paramount to maintain the control plane operative in the presence of high control loads towards the control plane, or in the presence of correlated

or uncorrelated faults. In addition, as derived consequence from centralization, SDN architectures suffer from scalability. Scalability can be defined as the capability of the control plane to handle the incoming requests from switches. This means that the control plane, and in particular, the SDN controller, needs to be robust enough facing failures, malfunctions, or attacks. Nevertheless, a few solutions focus on the control plane and especially on the SDN controller in itself.

According to (Basil et al, 2015), three metrics can characterize the scalability of a SDN controller: the control sessions capacity, the network discovery size, and the forwarding table capacity of SDN controller.

In SDN infrastructures, there are three key parameters: the location of the controller(s), the number of SDN controllers deployed, and the number of switches assigned per each controller. These three parameters can have a huge impact on the scalability and performance (e.g. latency of an SDN, the number of needed flows in the switches, or control plane's availability). For instance, when the number of forwarding elements connected to the SDN controller augments the SDN controller risks of becoming congested due to their requests. Similarly, for a given set of switches, the number of SDN controllers may be not enough to deal with all the control traffic generated by those switches. One solution is to distribute the control plane to alleviate the control load towards the control plane, by means of westbound/eastbound API interfaces to ensure the communication among different SDN controllers or clustering techniques, which instantiate several instances of the SDN controller.

(Heller et al, 2012) calculated the number of SDN controllers needed for different network topologies and their optimal location in those topologies. The authors studied 256 different topologies, including linear, ring, hub-and-spoke, tree, and mesh. In their study, the number of needed controllers and their location was calculated to reduce the latency. The authors demonstrated how the latency could be reduced when the number of controllers augmented, and how the latency depends on their position in the network. However, this calculation does not consider if that number and location of controllers is sufficient to ensure fault tolerance. (Yazici et al, 2014) solved a similar problem, by proposing a distributed OpenFlow controller framework and its associated coordination mechanism to augment scalability and reliability under high load in datacenters. This is based on a set of controllers, a.k.a. controller cluster, with are continuously communicating. There is a master node per cluster, which plays the role of controller, but is continuously monitored to detect its failures to replace it immediately by any other node in the cluster.

(Curtis et al, 2011) proposed DevoFlow, a slight modification in the OpenFlow protocol in order to reduce the number of necessary flows installed in the switches. The authors demonstrate that the number of flows can be reduced up to 53 times and the need control messages between the SDN controller and the switches can be reduced up to 42 times. (Li et al, 2014) proposed a secure SDN distributed infrastructure able to resist Byzantine attacks on the SDN controllers and empowered with resiliency. On their proposal, the authors advocate to assign several controllers to each OpenFlow switch. With this controller redundancy per switch, switches are resilient to correlated failures because there are several assigned controllers and, on the condition that there is at least one SDN controller available at any time. The authors also study the controller assignment problem that reduces the number of controllers deployed for a given set of switches.

As a consequence of the logical centralization of the intelligence in the control plane, the SDN controller becomes vulnerable to attacks that as a result, compromise the data plane. For additional information concerning types of vulnerabilities and attacks in SDN, the European Union Agency for Network and Information security provides in (ENISA, 2016) with an extensive threat landscape for SDN.

On one hand, the topology discovery procedure in SDN is based on the OpenFlow Discovery Protocol OFDP. This protocol is based on LLDP (Link Layer Discover Protocol), which is vulnerable to link spoofing attacks because LLDP packets are not authenticated. Link spoofing attacks consist of fabricating LLDP packets and corrupt the network topology information seen by the SDN controller. Nevertheless, there are several countermeasures like the proposed in (Alharbi et al, 2015), which consists of adding a cryptographic MAC to LLDP packets. On the other hand, the SDN controller is vulnerable to DoS and DDoS (distributed) attacks. Those attacks inject huge amount of signaling traffic from one infected host (or several infected hosts in DDoS) to the SDN controller in order to overload it. This overload, together with the centralization of the SDN architecture, causes the control plane to become inoperative, as a result, the data plane.

(Fonseca et al, 2012) proposed an Openflow-based replication mechanism to improve the resilience in SDN that detects abrupt failures on the SDN controller and use replication techniques to transition to a back-up controller. This mechanism could detect failures on the SDN controller and use replication techniques to transition to a back-up controller, but it could also detect DDoS where one attacker host sent packets with random IP source, what forces the SDN controller to install new flows so often that becomes inoperative. Nevertheless, this solution can detect the unresponsiveness of the SDN controller and transition to a back-up controller as soon as this is detected. This technique instantiates replicas of the primary SDN controller and sends the appropriate messages to those replicas when the primary SDN controller fails in order to inform them to take the control of the network. The authors considered the abrupt abortion of the SDN controller and SDN application failures. However, this approach only considers when the SDN controller is compromised, leaving out faults in the rest of components such as the switches, the hosts connected, or any link at the control or data network. However, it only considers when the SDN controller is compromised, leaving out faults in the rest of components in the network.

2.3.3 Fault management solutions for the application plane

Closely related to attacks is the lack of control in the application plane of SDN. As SDN proposes the application plane composed of many SDN applications to take control of the data plane via the SDN controller, the policies sent by the different SDN applications could conflict in the SDN controller (Ma et al, 2014) , (AuYoung et al, 2014) , and (Paladi, 2015). A possible solution is a mediator between the application plane and the control plane (the SDN controller) to check and validate the policies sent by the SDN applications to avoid conflicts.

With this concern, (Ma et al, 2014) and (AuYoung et al, 2014) proposed Athens, a programming framework that detects and resolves dynamic resource conflicts between black-box SDN applications and cloud applications. Given the load on the links in the network as constraint, each SDN application will try to install a set of flows on the switches without taking into account the flows installed by other SDN applications, leading to a resource allocation conflict, where this problem becomes more and more complicated with the augment of the number of SDN applications. The authors propose a configurable coordinator that solves conflicts among SDN applications by automatically assigning a number of votes to each SDN application to decide the priorities of each flow installed and achieve an optimal allocation.

(Paladi, 2015) proposed a framework to manage policies over SDN infrastructures, shown in Figure 10. This framework creates, verifies, and enforces SDN policies as well as controls the access of management applications. This framework is based on two types of policy checkers, one offline policy checker and a real-time policy checker. The real-time is in charge of continuously verifying the incoming policies and tagging them in order to identify to issuing source component. Once those policies are accepted, those are sent towards the SDN controller. The offline policy checker is in charge of conducting periodic and static policy verification in order to ensure isolation, network reachability, and liveness.

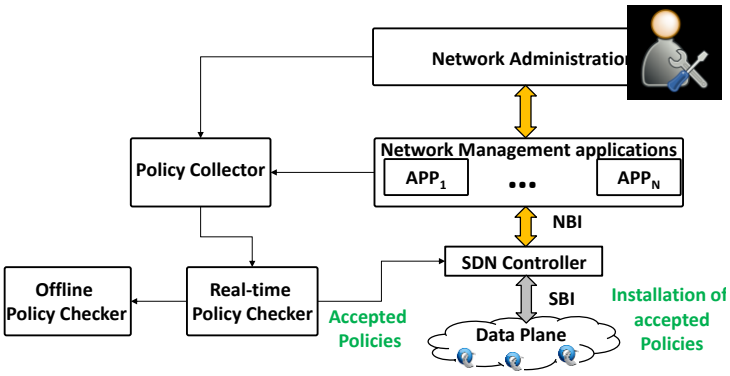


Figure 10. Policy Management architecture

Additionally, the northbound interfaces of the controller must be secured with mechanisms like TLS or SSL to avoid malicious SDN applications to take control of the controller and, by extension, of the data plane. In addition, the mediator entity can ensure that the SDN applications sending policies to the SDN controller are trusted and legitimate to validate their policies before being sent to the SDN controller.

The work from (Fonseca et al, 2012) also detects when an SDN application may lead the SDN controller to a failure state. The authors prove this by creating a SDN application that creates a socket to connect to the secondary controller and waits for a message. When the switch does not receive any confirmation coming from the SDN controller due to the failure on the SDN application, it reconnects to the secondary controller.

2.3.4 Fault management solutions for legacy and OpenFlow equipment

Another constraint in fault management solutions for SDN is that most of them are OpenFlow centric, and so they ignore legacy equipment and non-OpenFlow devices. Hence, several fault-tolerance mechanisms propose solutions for OpenFlow-based equipment and do not seem to be extensible to other equipment such as legacy equipment. Indeed, we identify a lack of fault management frameworks for both OpenFlow-based and non-OpenFlow or legacy equipment—such as programmable eNodeBs (Evolved Node for LTE/UMTS), legacy switches and routers, no matter which southbound protocol they use (e.g. SNMP, NETCONF, etc.).

(Sharma et al, 2013b) proposed a hybrid framework, called i-NMCS (Integrated Network Management and Control System), that includes legacy network management functions as well as SDN based management functions. This framework includes the traditional network management functionalities for topology discovery and fault detection but also the dynamic control based on SDN for provisioning end-to-end flows. The architecture of i-NMCS can be seen in Figure 11. Its main elements are:

- An event manager: it collects events such as new flows or link state changes and updates the provision repository.
- A policy manager: it provides with interfaces to specify network requirements to ensure a given QoS,
- The control decision engine: it is the core of the i-NMCS architecture and it translates the policies from the operator to specific SDN control actions.

The SDN controller integrates several southbound plugins to communicate with several types of equipment such as OpenFlow, legacy switches, SNMP switches, of virtualized equipment.

Its functioning is as it follows. When a user connects to the network, it is authenticated and starts using a VoD service (Video On Demand service) in the enterprise. The OpenFlow switch detects the first packet of that new service and asks the QoS solver for a new flow. The QoS solver is a SDN application running over the SDN controller and computes the required path to connect the user and the streaming service satisfying the QoS level.

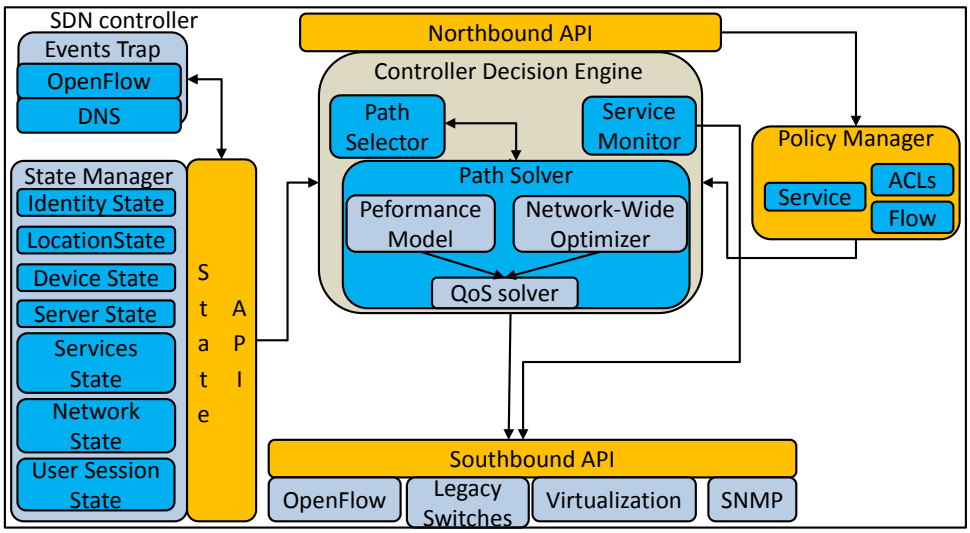


Figure 11. i-NMCS architecture embedded in the SDN controller

2.3.5 Fault management solutions including diagnosis

Most solutions for SDN do not tackle the diagnostic aspects, as exception of some troubleshooting mechanisms. These exception are: NICE (Canini et al, 2012) to test OpenFlow applications, OFRewind (Wundsam et al, 2011) to pinpoint invalid controller actions and packet parsing errors between control and data planes, STS (Scott et al, 2014) to analyze software bugs, NDB (Handigol et al, 2012) to trace packets, or NetSight (Handigol et al, 2014) to detect forwarding loops, among others. Indeed, tracing packets in OpenFlow networks seems to be a hot topic, as indicated by (Georghe et al, 2015) not the diagnostic aspects in itself.

To cite some work that somehow performs some kind of diagnostics on SDN, we can cite the aforementioned multi-agent distributed network troubleshooting mechanism for SDN (SDN-RADAR) proposed in (Georghe et al, 2015) that identifies faulty network links impacting user experience. This is a SDN application running on top of the SDN controller, which is running at run-time, and it is intended to support human administrators in charge of performing troubleshooting in SDN. The output of the troubleshooting approach is a weight which is calculated per link that augments with the probability of that link to be the root cause of the bad quality experienced by the user. This information is given to the human administrator for a deeper analysis on each suspected link.

As a conclusion in this section, diagnostic aspects are not well covered so far in the state of the art on SDN infrastructures, in especial multi-layered diagnostic approaches, what indicates a research line to follow, that is why in this thesis we covered this essential aspect for SDN infrastructures.

2.4 Overview of NFV

In this section, we describe the NFV architecture, with its functional blocks and its mode of operation. We first define the different components of the NFV architecture, following the terminology provided by the ETSI NFV ISG in (ETSI NFV, 2012) and (ETSI NFV, 2014). We will also give shed on the main challenges in NFV-based solutions.

2.4.1 NFV Architecture

A VNF is commonly referred to a virtualized network function, but is formally known as an implementation of a network function susceptible to be deployed in a NFVI. The NFVI infrastructure (a.k.a NFVI) is composed of the virtual and physical computing, network, and storage resources necessary to ensure the functionality of VNFs (Figure 12). The virtual resources of the NFVI rely on a virtualization layer which in turn relies on the physical substrate.

NFV architecture is composed of the following blocks, namely, the NFV infrastructure (NFVI), the NFV Management and Orchestration block (MANO), and the OSS/BSS block, shown in Figure 13. In turn, the MANO block is transversal to the OSS/BSS (Operations Support Systems/ Business Support Systems) and it is composed of the NFV orchestrator, the VNF Manager, and the Virtualized Infrastructure Manager (VIM).

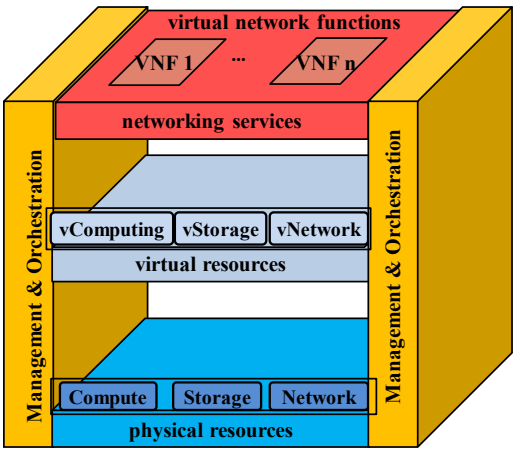


Figure 12. NFV definition

VIM: This block controls and manages the NFVI compute, storage and network resources. This block exposes northbound APIs to manage the virtualized resources inside the NFVI. This block is in charge of creating the underlying virtual links and virtual networks necessary to ensure the VNFFGs of each networking service. It also keeps updated a database with the allocation of virtual resources on the physical substrate.

NFV orchestrator: This block ensures the life cycle of network services, and it orchestrates NFVI resources across several VIMs, manages the policies for the networking services, validates and authorizes NFVI resource requests.

VNF Manager: This block is responsible for handling the life cycle of the VNFIs. It is in charge of managing, modifying, healing, terminating, updating, upgrading, scaling, and migrating the VNFIs.

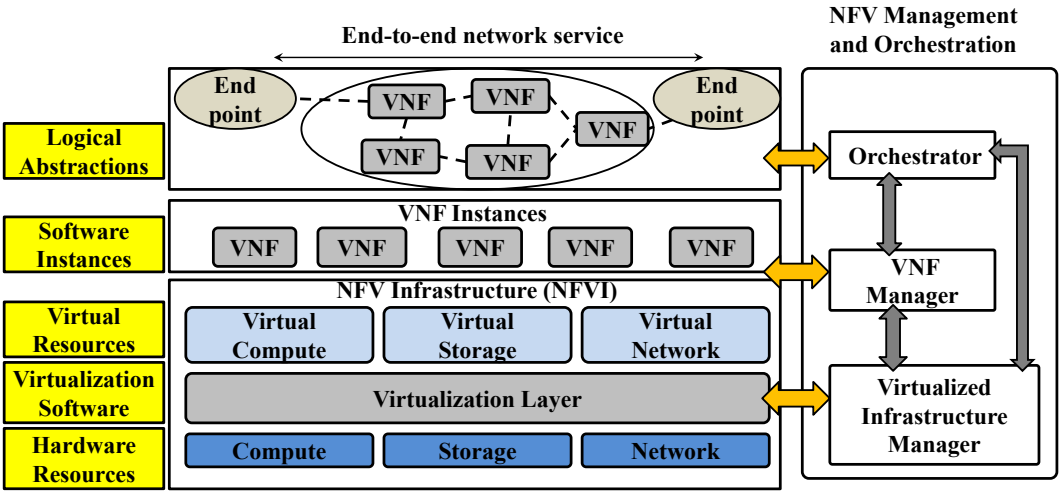


Figure 13. Functional NFV architecture by ETSI NFV

The procedure to instantiate a network service in the NFV architecture is at it follows: First the OSS/BSS sends a service order to the NFV orchestrator which translates it into a resources order, which will be sent as output to the NFVI infrastructure to allocate the necessary resources to instantiate that network service.

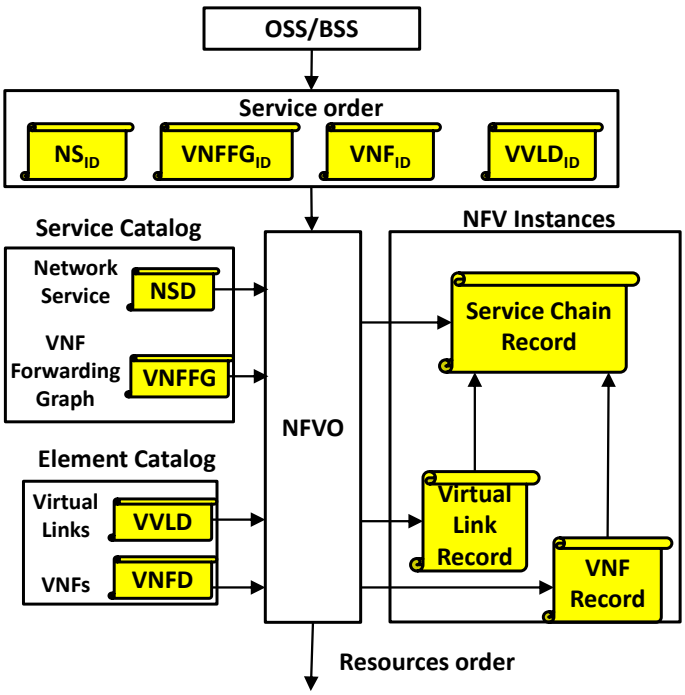


Figure 14. Network service instantiation in the NFV architecture by ETSI NFV

A network service is instantiated in NFV by following these three steps: reception of the service order, instantiation of the VNFs, and instantiation of the virtual links, where this process can be seen in Figure 14.

Reception of the service order: Firstly, the NFVO receives a service order from the OSS/BSS. This service order is a data model, which consists of parameters such as the identifiers of the network service to be instantiated, the identifiers of the VNFFG, the identifiers of its composing VNFs, or the identifiers of the Virtual Links (VL) interconnecting the VNFs.

The NFVO then consults the service catalog where it reads the NSD (network service descriptor). The NSD provides with the VNFs and the Virtual Links (VL) to be instantiated, which are contained in the associated VNFFG (VNF forwarding graph) to that network service. The VNFFG describes how traffic is forwarded among VNFs composing this network service. The NFVO will then instantiate the corresponding VNFs and VLs composing that network service.

VNF instantiation: Secondly, the NFVO consults the element catalog to read the VNF Descriptor (VNFD). The element catalog contains all the on-boarded VNF packages the NFV architecture can provide. The VNFD contains the requirements of each VNF composing the network service such as the VNFC (VNF component) composing that VNF as well as their intra connections, the computer requirements and SLA (Service Level Agreement) parameters. VNFC is usually mapped to a given VM embedded in a physical hardware. The requirements of VMs (required storage, compute parameters, scaling limits, etc.) are given by their respective VDU (Virtual Deployment Unit) descriptors.

The VNF Manager (VNFM) instantiates each VNF. It first reads the VNFD of the VNF to be instantiated, which contains information of the VNFC composing that VNF. The VNFM then gets the number and types of the VNFC to be instantiated. For each VNFC, the VNFM reads the VDU of each VNFC and requests a new VM for that VNFC and network and storage resources to the NFVI. Depending on the number of VNFC composing the VNF, this instantiation process can be more or less complex. For instance, if there are several VNFCs, firstly the different VNFC are instantiated and secondly those are connected to each other. Once those VNFCs are connected, a series of messages are sent among VNFCS in order to find a suitable VNFC to play the role of master and coordinate all the VNFCs (master function). Only the master function communicates with the VNF Manager. After the resources of the VNFC are allocated, the VNF Manager asks to start that VM and it then informs that the VNF is ready for configuration.

VL instantiation: The NFVO consults the element catalog to read the Virtual Links Descriptor (VVLD)**.** The VVLD contains the type of virtual link (point-to-point, point-to-multipoint, etc.), the associated KPIs (Key Performance Indicators) such as QoS, latency, bandwidth), and the network type (hypervisor vSwitch, etc.).

The VIM (Virtualized Infrastructure Manager) allocates the necessary components in the NFVI for each VL. The virtual links connect the VNF in that network service. The VNF NCT (Virtual Network Function Network Connectivity Topology) is a set of VLs to connect the different VNFs. These VNFs are connected through connection points (CP). In Figure 15 (a) is given an example of NCT composed of 4 VNFs linked by three VLs.

On the other hand, the NFP (Network Forwarding Path) is an instance of a given VNF FG (Virtual Network Function Forwarding Graph), to indicate the concrete flows for the traffic among VNFs. A NFP can be defined as and ordered list of CPs traversed to compose a chain of VNFs. There may be several NFP for describing different types of traffic (e.g. media, control, etc.) for the same network service, an example of NFP is given in Figure 15 (b), where those NFP are associated to three different types of communication (media, control, etc.) transmitted over the same VNF NCT of Figure 15 (a).

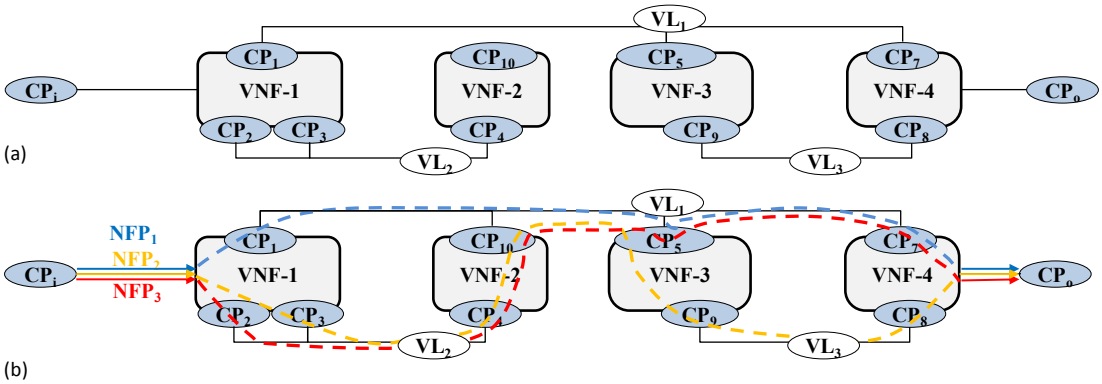


Figure 15. (a) VNF NCT, and (b) NFP (blue, red, and yellow)

Once a VNF is instantiated, it becomes a VNF instance (VNFI), a.k.a as the runtime instantiation of a given VNF. A VNFI is susceptible to be upgraded, update, roll backed, scaled-in/out, scaled-up/down. On one hand, scale-out means adding additional virtualization container for an additional VNFC associated to a given VNFI. The inverse operation is scale-in, and consists of removing that additional virtualization container (VM) associated to a given VNFI. On the other hand, scale-up means increasing the CPU, memory, and storage of a given VM currently supporting a given VNF. The scaling limits supportable by the VM are described in the VNFD. The inverse operation is scale-down, and consists of reducing such parameters. There are three VNF scaling models in NFV:

- Auto-scaling: the VNF Manager triggers the scaling of a given VNF following the VNFD,
- On-demand scaling: launched whether by the VNF instance or the EM without having to request to the VNF Manager,
- Scaling based on a management request: manually triggered by a NOC operation for example.

Each VNFI is characterized by a VNF life-cycle (Figure 16), composed of five states and its corresponding transitions among states. The VNFI life cycle is managed at all times by the VNF Manager. These states are described hereafter:

- **VNFI Null:** the VNF instance does not exist yet and has to be created or instantiated

- **VNFI Instantiated not configured:** the VNF instance exists but is not configured. Once it is configured in transitions to an inactive state.

- **VNFI Instantiated and configured:** the VNFI exists and it is configured, it can be active or inactive.
    - **VNFI Active:** the VNFI is involved in a networking service
    - **VNFI inactive:** the VNFI is not involved in a networking service

- **VNFI Terminated:** the VNFI has been deleted

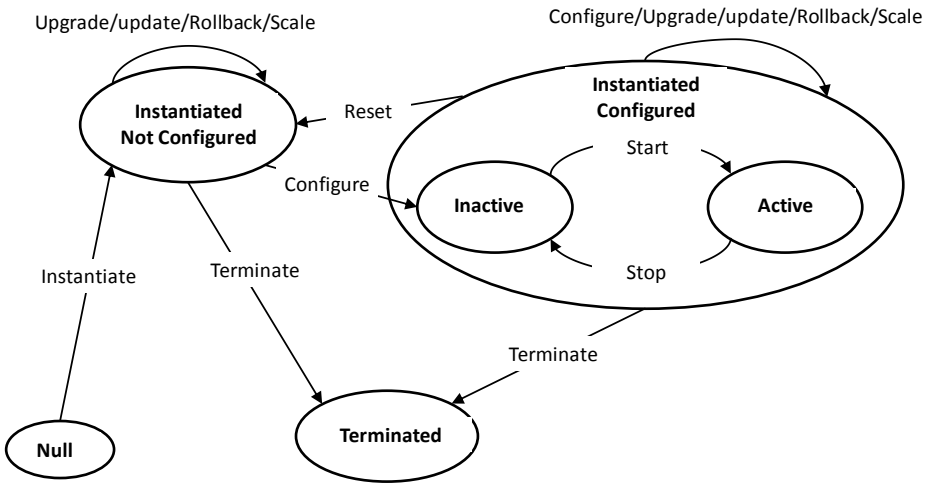


Figure 16. VNFI state transitions by ETSI NFV

2.5 Related work on Fault Management in NFV

We consider in this thesis as fault management mechanisms for NFV as those mechanisms to prevent that any fault in the NFV architecture lead to a failure on the network services. As advantage of NFV approach, network services become more resilient, in terms of both traffic tolerance and redundancy.

On the one hand, network services become more traffic tolerant. This is because sudden increases in the service requests towards the VNFs can be managed whether by replicating the VNF and distributing new VNF instances into multiple physical nodes (scale-out) and distributing the service requests to the rest of nodes, or by assigning more physical resources to the current VNF where the service requests are sent (scale-up). On the other hand, network service can benefit from dynamic redundancy, as VNFIs can be duplicated with redundancy mechanisms at run-time or migrating them on live with pre-emption and regression mechanisms what ensures resilient networking services.

However, networking services defined in NFV have different fault management requirements, not only in terms of SLA and network constraints, but also specific requirements coming from the NFV approach, which will influence the way fault management mechanisms will cope with their respective malfunctions. Indeed, there are two types of networking services, stateful and stateless services. On one hand, stateless services do not require the maintenance of the session parameters to ensure their continuity. As a result, it suffices to migrate the affected VNF to a new physical location containing the VM, if its computing and storage parameters are appropriate. Examples of stateless services are DNS (Domain Name System) and LDAP (Lightweight Directory Access Protocol). On the other hand, stateful services do require the maintenance of the session parameters to ensure their continuity. As a result, it does not suffice to migrate the affected VNF to a new physical location containing the VM, because the state information of the network service must be restored in the new physical location containing the new VM. Stateful services are those based on SIP (Session Initiation Protocol).

In addition, NFV introduces new types of failures as identified by the ETSI NFV group in (ETSI NFV, 2015b) , due to internal composition of the VNFs. Indeed, the NFVI is composed of Network Functions Virtualization Infrastructure Nodes (NFVI nodes). NFVI nodes are the physical devices providing the necessary NFVI Functions for the execution environment of the VNFs. These execution environments are called also virtualization containers or Virtual Machines (VM). ETSI NFV defines four manners to implement a NFVI node. These modes are shown in Figure 17, and explained further here:

- Figure 17 (a): The network function is physical (black-box model)
- Figure 17 (b): The network function is virtualized (VNF) by means of a VM and a hypervisor.
- Figure 17 (c): The hardware is sliced to contain several VNFs.
- Figure 17 (d): The VNF is composed of several VNFC that span across different hosts and VMs.

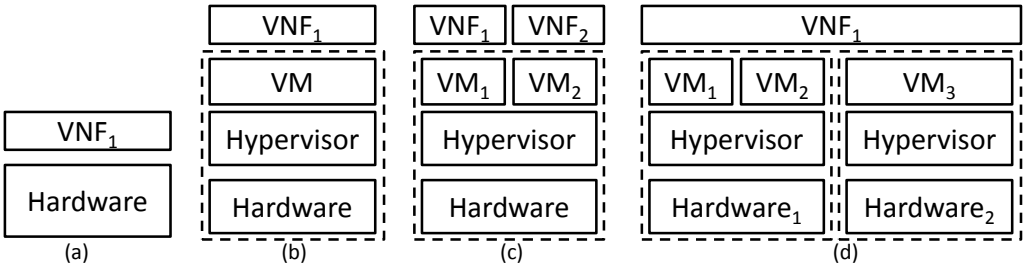


Figure 17. Different types of implementation of a NFVI node by ETSI NFV

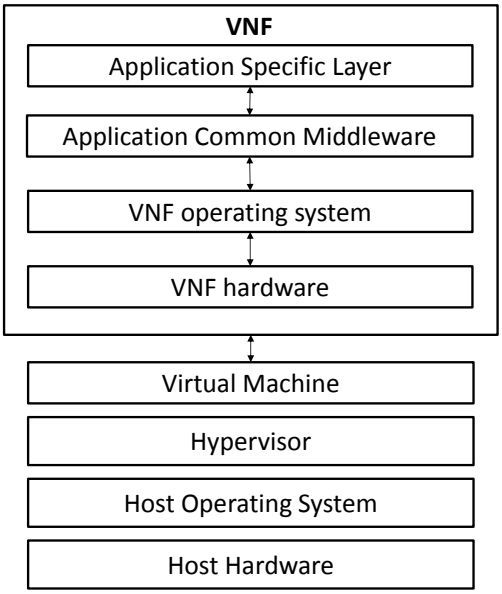


Figure 18. Detailed VNF layered architecture by ETSI NFV

Figure 18 shows the layered architecture of a NFVI node, composed of the VNF and the underlying hardware. The VNF is decomposed in the application specific layer, the application common middleware, the operating system of the VNF and the hardware. We provide in Table 3 with those new types of faults in NFV, classified in energy faults, hardware faults, software faults, and virtual resource faults. Inside each category, we define those typical faults imposed by the NFV technology. The goal of a self-healing system is to avoid that any type of these faults compromises the network infrastructure, supported services and eventually lead to service failures.

Table 3. Some examples of faults in NFV combined infrastructures

Type of fault	Examples
Energy fault	Rack, board
Hardware fault	computing resources faults: disk, memory, CPU network resources faults: NIC, board, bridge
Software fault	VNF application specific fault VNF application common middleware fault VNFC fault VM fault Hypervisor fault Host Operating System Controller kernel flaws Controller Buffer overflows switch buffer overflows
virtual resource fault	virtual network sources faults: virtual links virtual computing resources faults: vNIC, vCPU, vBridges,

In addition, NFVI nodes have a specific layered architecture but also those can be implemented in four different ways, what introduces new types of failures and security threats due to the inclusion of the virtualization layer.

- The additional software introduced in a NFVI node such as the VM, the hypervisor, and the operating system, adds new types of failures and security threats at both a hypervisor level and a virtual machine level.
- The additional layers introduced in the VNF such as Application specific layer, application common middleware, VNF operating system and the VNF hardware introduce different types of faults per layer.
- The slicing of the hardware in order to embed different VNFs, introduces performance impact of among the VNFs embedded if those VNFs are not properly isolated.
- The spreading of VNFs across different VMs embedded in different but remotely connected hardware nodes is vulnerable to link communication failures.

Indeed, NFV-based architectures suffer from several security weaknesses inherited from this virtualization layer. As examples, the Virtual Infrastructure Manager (VIM) is vulnerable to DoS attacks that may prevent users from requesting virtual resources, to VIM intrusion attacks that may free resources, or VIM session hijacking that exploit the weak authentication. For instance, VM side-channel attacks are similar to virtual resource intrusion attacks, what leads to loss of confidentiality and loss of integrity of the virtual resources, unless the VM-to-VM traffic is encapsulated in a secure way to avoid any malicious interceptions. Man-in-the-middle attacks consist of compromising the management channel between the virtual resources and the management system, which will have the same consequences as the VM side-channel attacks and the virtual resource intrusions. While uncontrolled-illegitimate resource requests consist of demanding a high amount of virtual resources, leading to denial of resources to other tenants/slices.

Due to these aforementioned aspects, fault management solutions are necessary for NFV. However, and although there are some management platforms as seen in the survey on NFV in (Mijumbi et al, 2015) such as Cloud4NFV or NetFATE, those platforms do not consider the inclusion of SDN as underlying infrastructure, which is the context of this thesis. There are interesting works concerning fault management in NFV, but the diagnostics aspects are not explore largely, to the best of our knowledge. For instance, the failure detector proposed in (Turchetti & Duarte, 2015) that detects faults on the data links and distributed processes inside a SDN infrastructure, exploits the NFV principles that allow to design, manage and deploy Network Functions in less time than traditional network functions. The failure detection mechanism is based on liveness request messages sent to the monitored process in a periodic manner. If processes do not reply to those messages in a given time interval, the NFV-FD suspects the process is failed, otherwise it states the process is healthy. The NFV-FD is connected to the FDMod block, which discards information of non-interest and analyzes the header information. The NFV-FD failure detector is located at control plane directly communicating to the SDN controller.

(Miyazawa et al, 2015) proposed a fault detection mechanism based on Self-Organized Maps (SOM) to detect failures in NFV-based services. The authors propose a failure model to explain degradations in VNFs such as network congestion and memory leaks. However, SOM parameters are tuned manually and in advance in accordance with the type of failure to detect.

Another important aspect is fault isolation, as identified in (Esteves et al, 2013) and (Chowdhury & Boutaba, 2009) as an open research field, where virtual resources are dynamically mapped over one common physical infrastructure and faults may propagate among networking services. With this concern, (Scholler et al, 2013) propose an information model to ensure a resilient deployment of VNF composing complex services in NFV where redundant components are strategically placed to avoid cascade effects. However, this approach does not ensure resilience in the operational phase. In addition, and as Cloud4NFV or NetFATE platforms, none of these considers SDN as underlying architecture.

2.6 Overview of SDN and NFV combined infrastructures

SDN and NFV are thought to be “better together” by the IT and telecommunication industry in order to exploit their potential benefits. Nevertheless, the way to combine both SDN and NFV architectures is still under discus-

sion, evidenced by the lack of consensus on the position of the SDN elements within the NFV framework. Indeed, there are several manners of combining SDN and NFV, as identified in (ETSI NFV, 2015a), depending on the position of the SDN elements. For instance, there several positions in the NFV architecture for the SDN controller, the SDN applications, and the SDN resources, defined hereafter:

Position of the SDN controller in the NFV architecture: the SDN controller could be considered as part of the VIM, a fully virtualized entity as VNF, part of the NFVI, part of the OSS/BSS or a PNF (Physical Network Function).

Position of the SDN resources in the NFV architecture: the SDN resources (or switches) could be physical switches or routers, virtual switches or routers, a switch or router as a VNF, or a host or server enabled with software (e-switch).

Position of the SDN applications in the NFV architecture: SDN applications use the application control interface (northbound API) of the SDN controller. These SDN applications could be part of a PNF, part of the VIM, virtualized as a VNF, part of an EM (Element Manager) or part of the OSS/BSS.

In this thesis, we will consider that the SDN is the underlying infrastructure ensuring the connectivity among VNFs. The control plane ensures this connectivity among VNFs by establishing virtual links. We consider that both the SDN controller and the SDN applications are part of the VIM. Indeed the SDN controller is managed by the VIM. The SDN resources could be both physical and virtual resources, on the condition that the SDN controller has knowledge those SDN resources in the network topology by means of its topology manager.

In NFV ETSI draft specification (ETSI NFV, 2015a) two types of SDN controller are defined: the infrastructure SDN controller and the tenant SDN controller. On one hand, the SDN tenant controller is located in the EM layer and coordinated with the SDN infrastructure controller. On the other hand, the SDN infrastructure controller is managed by the VIM and it is in charge of ensuring the communication among VNFs mainly providing with connectivity services through the NFVI. This means that the SDN infrastructure controller is in charge of establishing virtual links among the VNFs.

In this thesis, we will focus on the first type of SDN controller, the SDN infrastructure controller and we will refer to it as the SDN controller. In combined SDN and NFV infrastructures, each networking service is composed of a chain of virtualized networking functions (VNFs) connected through virtual links, where the SDN infrastructure controller establishes those virtual links. Figure 19 shows two networking services deployed in a combined SDN and NFV infrastructure. The networking service 1, composed of two VNFs, connects end points C and D (in green) and the networking service 2, composed of three VNFs, connects two end points called A and B (red). Networking service 1 relies on a virtual link $VL_{C,D}$ and the networking service 2 relies on a virtual link $VL_{C,D}$.

First, the NFV MANO block decides where to deploy the VNFs according to the network constraints (nodes capabilities, faults, etc.) and requirements to be fulfilled by the network service specifications (bandwidth, delay, etc.), and it sends the location of the VNFs to the SDN controller, through the orchestration interface, which, in turn, will establish the virtual links connecting the VNFs. It is the SDN controller who decides this path at run-time i.e. it decides the physical network elements on which to allocate the virtual links by establishing flows on the corresponding forwarding elements. The SDN controller and NFV MANO cooperate through the orchestration interface, which is normally the northbound API of the SDN controller.

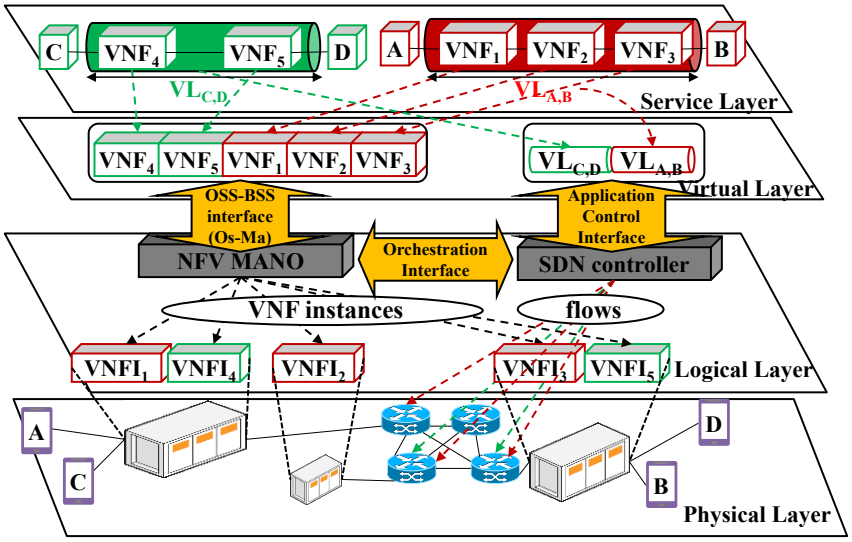


Figure 19. Networking service deployment in a combined SDN and NFV infrastructure

The inherent multi-layer architecture of SDN and NFV, but also the decoupling between virtual and physical layers, pushes towards a multi-layered network architecture, which raises several concerns:

2.7 Dynamicity of SDN and NFV infrastructures

Dynamicity is the main challenge to provide SDN infrastructures with fault management and thus. A fault management mechanism considers this dynamicity by modelling first the dynamic dependencies to reproduce reliably how faults propagate. The dynamicity comes given by the two approaches SDN and NFV.

2.7.1 Dynamicity in SDN

In this section, we discuss those cases deriving in dynamicity of the software-defined infrastructures. We identify three factors: changes on the network topology, changes on the type of control, and changes on the forwarding flows. Indeed, these three factors influence how control and data planes interact, and as a result determine the paths followed by control and data packets.

2.7.1.1 Changes on the network topology

The network topology continuously evolves with those network elements discovered by the SDN controller at run-time. This discovery process usually takes a few milliseconds in an OpenFlow network, where the protocol LLDP (Link Layer Discovery Protocol) ensures the discovery of the newly added switches and hosts.

The dynamicity in the network topology can be seen from two different perspectives: The first case is when the network topology changes over time, where the switches and hosts continuously appear and disappear leading to the fact that the network topology constantly changes as a result, as shown in Figure 20.

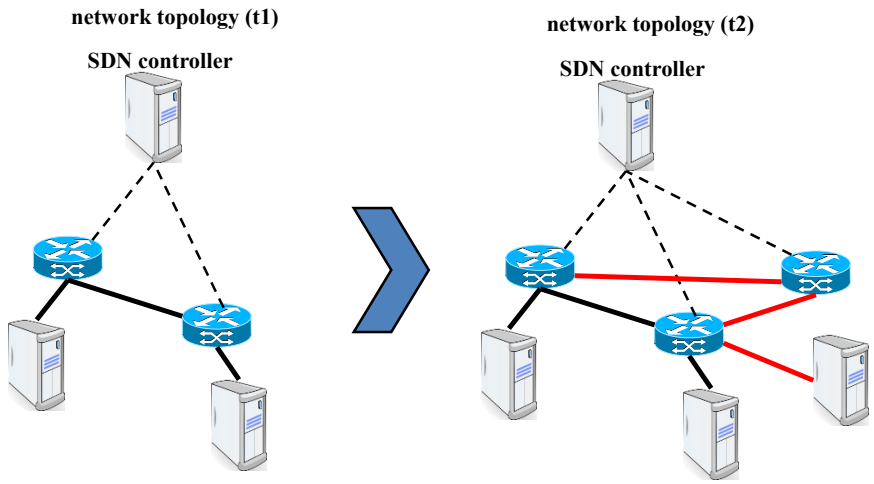


Figure 20. Changes on the network topology in a centralized SDN infrastructure

The second case is when considering the network topology per controller's domain. This case refers to distributed SDN infrastructures, where each SDN controller sees a different network topology so faults propagate in a different way in each controller's domain, as seen in Figure 21.

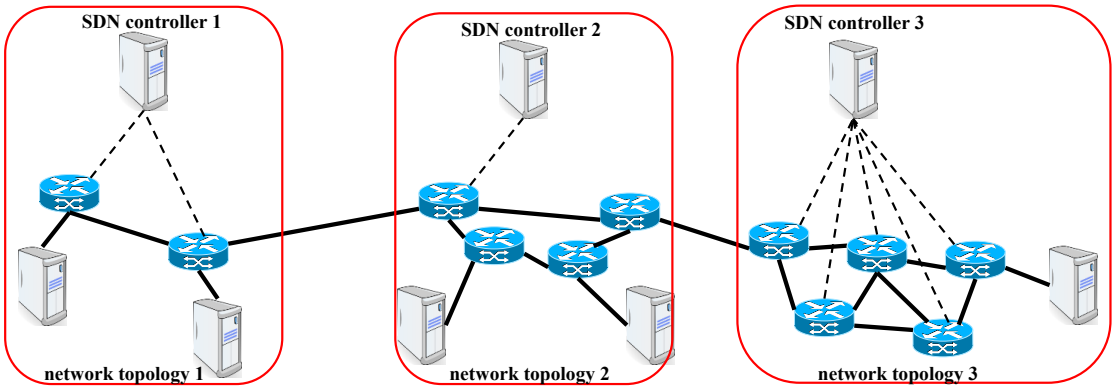


Figure 21. Different network topologies in a distributed SDN infrastructure

2.7.1.2 Changes on the type of control of the SDN infrastructure

The interactions between the control plane and the data plane vary substantially for the same network topology depending on the type of control (in-band or out-of-band), as we shown in previous sections in this chapter how the type of control influenced such different aspects such as fault propagation, centralization, control traffic load, and flow installation delay. In addition, in distributed SDN infrastructures, each SDN controller can control the data plane in a different manner (in-band or out-of-band) so faults propagate differently in each controller's domain due to this factor, as shown in Figure 21.

Additionally, we can imagine the case where the SDN controller is attached to several switches under out-of-band control, and it detects a failure in a control link towards one of those switches, what impedes the SDN controller to install flows on that switch. In this situation, a fault management mechanism could switch the type of control to in-band, and choose as master that switch with which the SDN controller can communicate directly via a dedicated control link. In this way, the SDN controller does not have to use the failing control link to install flows on the switches, and it can send those flows via the master switch. Figure 22 shows this situation with simple example.

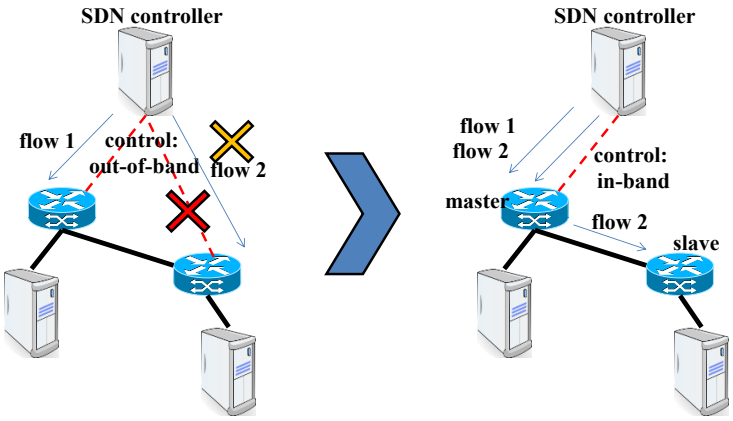


Figure 22. Changing the type of control to overcome a fault on a control link

However, this change in the type of control makes the network topology dynamic, because the interactions established among SDN resources change.

2.7.1.3 Changes on the flows sent by the control plane

The SDN forwarding is based on flows, where the SDN controller establishes paths by installing a set of flows on the switches. Flows can be deleted, modified and installed in a few milliseconds, which makes virtual links and physical paths dependencies change continuously and in a rapid manner.

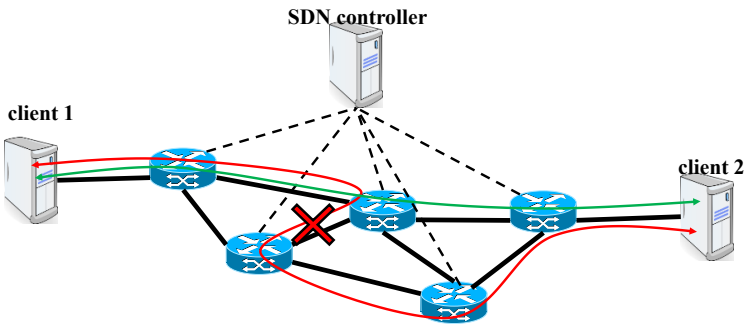


Figure 23. Traffic restoration in SDN

The SDN controller sets a virtual link by sending a set of flows to a selected set of switches and it allocates the physical path between two end points according to a given criterion (end-to-end delay, shortest path, minimum number of hops, etc.). The network elements composing a virtual link can be modified at any time by the SDN controller i.e. by modifying the flows associated to that path, what it means that the dependencies of that virtual link from the physical elements change continuously and in a rapid manner.

As example, we show a typical traffic restoration mechanism in Figure 23, where the SDN controller has already installed a virtual link (in green) to connect 2 clients, client 1 and client 2. However, one physical link fails and then it affects the virtual link. The role of the SDN controller is then to set an alternative path (in red) to ensure the connectivity between both clients. This change in the physical path changes the dependencies of the virtual link from the physical resources.

2.7.2 Dynamicity in NFV

In this section, we discuss those cases deriving in dynamicity of the NFV architecture. We identify four cases: scaling-up/down, scaling-out/in, changes on the VNF locations, and changes on the virtual links.

NFV boosts the flexibility by permitting network services to be composed in an elastic manner at run-time by dynamically chaining VNFs. However, dynamicity is a challenge from a fault management perspective, where network services dependencies become highly dynamic, because the composing VNFs and virtual links can be

moved and migrated across the NFVI, what implies that the dependencies between the network service and VNFs and the dependencies between the VNFs and the NFVI are in a constant change.

2.7.2.1 Scaling-up and scaling-down

In NFV is possible to increase the computing resources granted to VNFs, by means of VNF post-deployment operations such as scaling-up and scaling-down. Scaling-up permits to increase the computing resources of the VM supporting the given VNFIs such as the CPU, memory, or storage size. If the physical resources of the virtual machine used by a VNFI are not enough to guarantee the performance of the networking service, those can be upgraded.

For instance, Figure 24(a) shows how a VNFI is upgraded. This is the only case where the dependencies of the VNFs from underlying physical elements are maintained.

2.7.2.2 Scaling-out and scaling-in

In NFV is possible to increase or decrease the instances of a given VNF, by means of VNF post-deployment operations such as scaling-out and scaling-in. Scaling-out permits to add new instances of a given VNF, by adding additional VMs embedding more VNFC. This allows treating huge users' traffic demands by forwarding the traffic excess to the newly deployed VNFCs.

For instance, Figure 24 (b) shows how an additional VNFC is added, however this change must be notified to the SDN controller in order to establishing a new virtual link to connect that new VNFC to the rest of VNFC composing the VNF.

2.7.2.3 Changes on the VNF locations

VNFIs can be migrated to other physical hardware in the presence of physical faults on links or in the internal hardware. As the VNF location was changed, the virtual links among them will also change. Additionally, the VNFI can be duplicated different physical resources to ensure the resilience of a given service or for load-balancing purposes, so additional virtual links will be allocated. In these two cases, the dependencies of the VNFs from underlying physical elements changes, but also the dependencies of the connecting virtual links from the underlying physical elements in the NFVI.

2.7.2.4 Changes on the virtual links

The virtual links that connect the different VNFs can be modified and set over different physical paths, in the event of faults and failures. Scaling-out permits to add resources instances such as VMs for a given VNF. As a result, this new instance will need to be connected through virtual links, possibly in different physical paths. For instance, in Figure 24, in the presence of a link fault, to ensure the service availability the VNF_2 is migrated. In this case, the virtual link interconnecting the VNFs will change in consequence (from red to green). In this case, as in the previous case, the dependencies of the virtual links change.

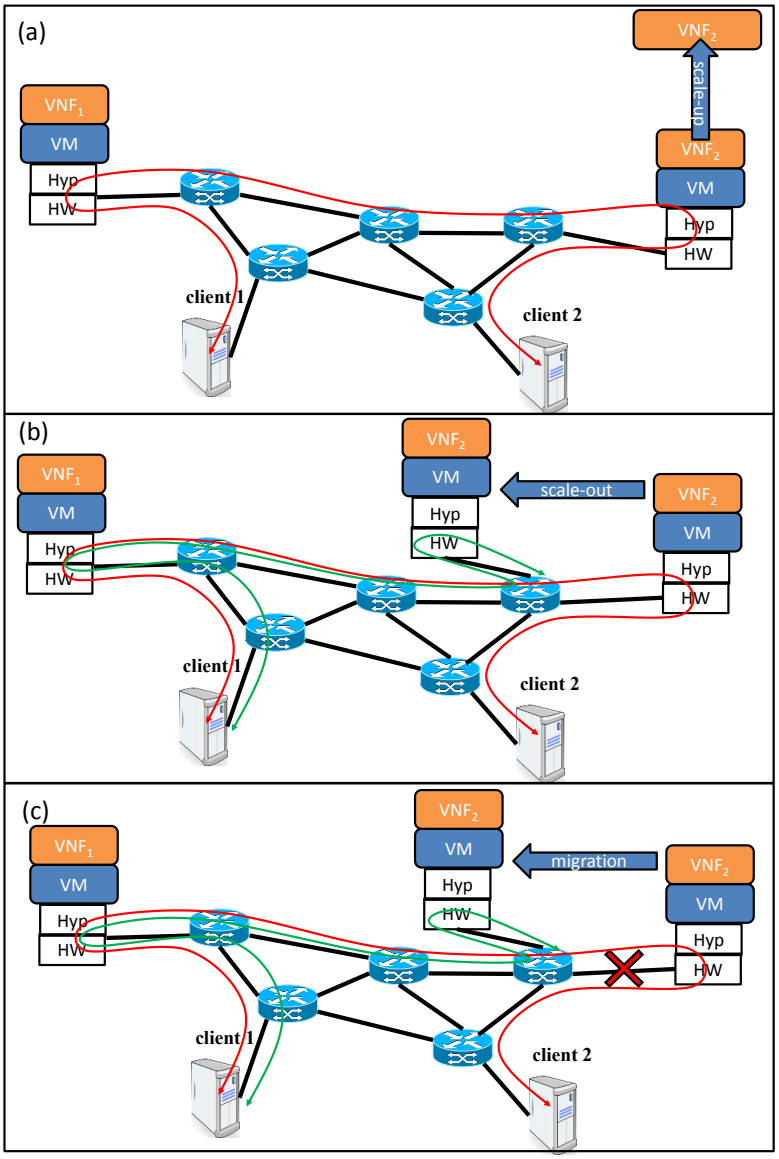


Figure 24. Examples of dynamicity in NFV architectures: (a) scaling-up, (b) scaling-out, and (c) VNFI migration

2.8 Conclusion

This chapter has analyzed both SDN and NFV architectures, functioning modes, and vulnerabilities as well as the related work on fault management in order to identify how to diagnose in an automated and intelligent manager such networks. We identified as main challenge the dynamicity in combined SDN and NFV infrastructures to provide the m with fault management capabilities, where this dynamicity comes given by the changes on the SDN infrastructure (e.g. topological changes, forwarding changes, control changes) but also by the elasticity in the NFV architecture (e.g. VNF scaling, migration, etc.).

Chapter 3 Related Work on Self-Healing Systems

3.1 Introduction

This chapter describes self-healing systems as well as their internal architecture and functional blocks. It also describes and details the origins of self-healing systems, their properties and mission to fulfil in a telecommunications network supporting one or several services. This chapter also compares the control-loop of a self-healing with the control loop of a resilience system as well as their functional tasks and architectures. It describes the most used algorithms in each of these functional blocks as focuses on the self-diagnostics aspects, the core of this thesis manuscript.

In the last part of this chapter, we review the most important self-diagnosis approaches for different network technologies with the aim to understand their limitations and the key learning aspects worth been taken into account in the context of programmable networks.

3.2 Self-Healing overview

Partially and statically automated network and service management characterize Telcos infrastructures and this static automation has reached the limit of its capacity due to the increasing complexity and heterogeneity of current networks. A huge number of operational challenges are inherent in this complex context. On the one hand, recent statistics state in (Wallin, 2012) "operators need to handle millions of alarms per day in each medium-sized NOC". On the other hand, around 40% of alarms are redundant and that the current alarm correlation level is around 1-2% at most. This indicates that there is a significant alarm overflow for human administrators, who are simply not able to handle such amount. The current level of automation in fault management is not at all adequate for future networks. In addition, the quality of most emitted alarms is low due to vendor dependency: each vendor proposes a different alarm interface and provides different documentation.

Fault management is always one of the most in vogue fields of research, due to the increasingly complexity and lack of manageability in all related information technology (IT) environments. Operator's infrastructures are composed of IT systems to provide their services and control the access of users. This issue emphasizes the need to explore the autonomic approaches, as these networks are growing faster and faster caused by the large amount of increasingly more demanding new applications and services. These huge demands in turn cause an unmanageable growth of the complexity and heterogeneity of network and equipment, what impacts efficiency and cost for the operators. In this regard, autonomics is the next generation of management solutions with its self-healing properties.

Network Operations Center (NOC) teams handle fault and service quality degradations within Telcos infrastructures. Their role is to establish the necessary steps to repair and re-establish services for end users. These steps are directory based but those may rely on operational brains that utilize the expertise of NOC teams. The Network Management Systems (NMS) detect the alarms sent by network equipment or their overlying Element Management System (EMS) through the northbound Interface (NBI). The NBI delivers the fault information to the

NMS by means of traps. The NMS aggregates and filters these alarms triggered by different equipment and various network segments. Each alarm generates a trouble ticket that is, considered to be valid, to determine the root cause of that alarm. NOC teams process each ticket separately (or globally if a global error is suspected) and access to the various directory bases to determine the appropriate steps to follow. This approach is error-prone as it relies on manual interaction of operational teams with supervision applications built on to silos and directory bases that are not up to date.

The main reason to empower a network with Self-Healing properties is to reduce the outages and the service downtime. For instance, in an operator infrastructure, applications and data servers are functioning continuously giving service to millions of clients simultaneously, so this downtime is an important parameter to be taken into consideration, due to the fact it represents the overall performance of their services and applications.

3.2.1 Preliminaries

A self-healing system has as main role to prevent, defend, detect, and recovery challenges and faults that may threat the network infrastructure operating services. A self-healing mechanism ensures resilience of a SDN and NFV combined infrastructure, which operates one or virtualized several networking services. The network infrastructure is characterized with some operational parameters depicted by metrics, as the overlying services are characterized by a set of service parameters and KPIs indicating their state.

Along this thesis will use the definitions for active and dormant fault, failure, and error given in (Avizienis et al, 2004). A fault has two states, dormant and active. An active fault may be enabled by challenges or system operations. A challenge, if not prevented by the system defenses, may turn a dormant fault into an external active fault (or equivalently, a system operation may turn a dormant fault into an internal active fault). Faults manifest as errors, where those can pass to the operational state and become failures if not avoided by a handling error system. A fault, or active fault, is “the hypothesized cause of an error” while an error is a deviation between an observed value and a specified correct value, also known as the manifestation of an active fault.

A service failure, also known as failure, implies a deviation of the service from its initial specification, where the functional specification relies on two aspects: the service content, and the service timing. Service content refers to the provision of the correct content regardless of the timing, and service timing refers to the provision of the content in a timely manner regardless of the content. We consider that a self-healing system must ensure the delivery of the service content and the service timing regardless of any type of challenges present in the system.

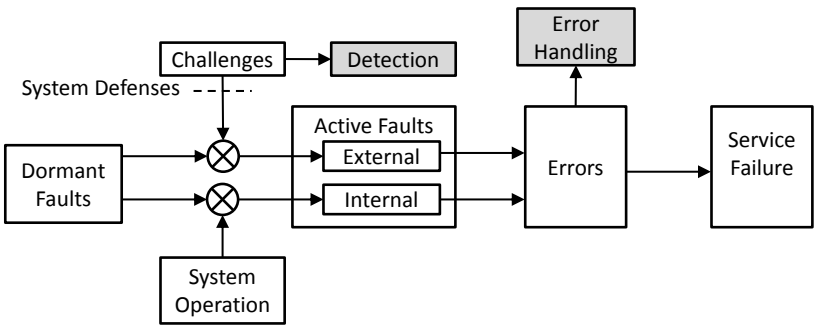


Figure 25. Fault-error-failure chain

One simple example of system operation is a database update, where, by accident, the new version changes the current update repository to a wrong repository, which is not available. However, this is a dormant fault because the symptoms of that wrong repository will not perceived in the network or services exploiting that database as long as no update is required. That dormant fault will turn into an error in the next update (missing repository) and, if the error is not properly handled, it will evolve into a service failure, because the service accessing to that database will experience a database connection error coming from that unsuccessful update.

In the fault management community there is a prominent lack of consensus on what is considered fault and failure. Indeed, in the latest document by ETSI NFV on resiliency requirements for NFV in (ETSI NFV, 2015b), the

authors considered that a fault management system should be renamed as failure management system. Indeed, fault and failure largely differ. According to the English dictionary, a fault is defined as "a problem that may not be obvious and could cause something to fail". (Smith et al, 2011) , (Avizienis et al, 2004), and (Salfner et al, 2010), considered that service failure, often referred to failure has as consequence the aforementioned deviation of the service functional specifications. Indeed, the following examples show this lack of consensus on faults and failures. For instance, in the optical communications field, malfunctions in the links and nodes in the optical infrastructure are considered themselves as failures, although those malfunctions are not impacting service delivered. Similarly, (Tipper, 2013) uses the term failure to describe faults in nodes and links that do not affect services, as well as (Cholda et al, 2007), which considered that faults in links and errors in software are failures but those could not lead directly to service failures and those should be faults in our perspective. We considered that in these aforementioned examples, failures should be defined as faults because the service is not affected.

3.2.2 Definition

There are many different definitions for self-healing systems, also known as self-repairing or self-recovering systems. (Kephart & Chess, 2003) considered that a self-healing system is derived from autonomic systems and defined as "a system that automatically detects diagnoses and repairs localized software and hardware problems". Indeed, a self-healing mechanism is an autonomic-based mechanism that performs the fault-management tasks (fault-detection, fault-isolation, and fault-removal).

On the other hand, (Ganek & Corbi, 2003) defined a Self-Healing system as an organized process of detecting and isolating a defective component, disconnecting it of the system, fixing it, and reintroducing it or even replace it if necessary without any apparent disruption, accomplishing with the target of minimizing the services outages. Contrarily, the ETSI AFI (European Telecommunications Standard Institute-Autonomic network engineering for the self-managing Future Internet) has a rather a proactive vision of a self-healing system: "a system encompassing processes for problem discovery through fault-detection, diagnosis and triggering appropriate actions to prevent disruptions". On the other hand, (Gosh et al, 2007) defined a self-healing system as a system able to perceive that is not performing well and, with or without human intervention, adopt the necessary measures to restore the normal state.

The goal of a self-healing system is to provide a system with resilience. (Sterbenz et al, 2010) define resilience as "the ability of the network to provide and maintain an acceptable level of service in the face of various faults and challenges to normal operation". It is interesting the differentiation the authors made between fault and challenge. One one hand, challenges have an external nature, those can be environmental, natural disasters, or external attacks to the network. On the other hand, faults are inherent to the network and are generated within the network infrastructure. Faults occur inside the system operation processes such as accidents, misconfigurations, operations, or even attacks from inside the network.

3.2.3 Origins of self-healing

As said before, the origins of self-healing come from the autonomic properties defined in (Kephart & Chess, 2003) , known as Self*-properties or self-X properties: Self-Configuring, Self-Optimizing, Self-Protecting and Self-Healing. As a result, Self-healing is one of the aforementioned autonomic self-* properties, and it has autonomic behavior. Autonomics is rooted in the plethora of biological mechanisms found in nature. As an example of biological mechanism, our human body is composed of motile ciliary cells, which have as role to prevent our organism from being infected by keeping the airways clear from mucus, what allows us to breathe normally. Indeed, if the beat frequency of those cilliar cells is not sufficient, they cannot expulse the mucus from our organism, and as consequence, it may compromise our health. The important point is that humans are not even aware when those autonomic biological mechanisms are functioning. This is one cornerstone feature of self-*properties, those are function without being perceived, in an autonomous manner. We present here the mission carried out by each self-* property:

- **Self-Configuring:** This refers to the automatic configuration of the network equipment by means of high-level policies. This implies that, when a new equipment is connected to the network, it should self-advertise by sending its capabilities and the system should configure it automatically as well as providing its capabilities to the rest of network equipment to be aware of this new equipment.

- **Self-Optimizing:** a.k.a. self-tuning or self-adjusting. This refers to the automatic setting of parameters of the equipment installed in the network in order to optimize the global behaviour of the network i.e. end-to-end delay.
- **Self-Protecting:** This refers to the capability of defending the system against correlated failures such as external attacks, massive disasters, or cascading effects. Its role is also to support self-healing systems when those are not able to deal with such problems.
- **Self-Healing:** This refers to the capability of detecting abnormalities, diagnose them and identify the reason for those abnormalities by calculating the root cause, i.e. the element or elements origin of the abnormality. The abnormality may imply a service failure, a simple fault having non-disastrous consequences on the service layer, or even a mismatch in a given operational parameter that could evolve in a service failure.

Each self-* property has strong relationships with certain quality factors, as proposed in (Salehie & Tahvildari, 2009). We focus on the quality factors related to Self-Healing properties, shown in Figure 26, and the focus on this chapter. As defined by Kephart and Chess, the main target of a Self-Healing system is to maximize the availability, survivability, maintainability and reliability of a given system. We define these four properties hereafter:

- **Availability:** The capability of the system to be ready for use. Mathematically defined as the probability of the system or service to operate when needed (Sterbenz et al, 2010)
- **Reliability:** The capability of a service to continue providing its content. Mathematically defined as the probability of the system or service to stay operational in a given time interval T (Sterbenz et al, 2010)

$$Reliability(t) = Prob(no\ failure[0, T])$$

- **Survivability:** The capability of a system to fulfill its mission in a timely manner in the presence of attacks, failures, or accidents (Avizienis et al, 2004)
- **Maintainability:** The capability of a self-healing system to repair and make the system evolve (Avizienis et al, 2004)

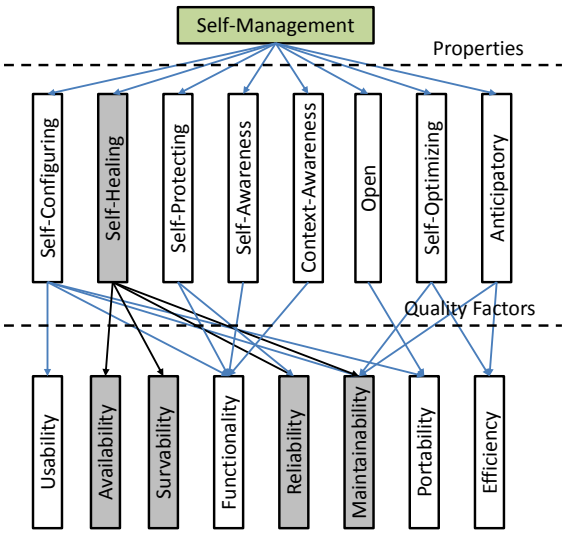


Figure 26. Quality factors associated to Self-Healing

(Salehie & Tahvildari, 2009) go beyond these definitions and classify self-* properties in a three-layered hierarchical structure composed of: general level, major level and primitive level, as shown in (Figure 27). Properties of lower layers are used by the properties located in higher layers to perform more complex autonomic behaviours. For instance, self-healing properties located in a major level may use self-awareness and context-awareness properties located at primitive level to carry out with their healing functions.

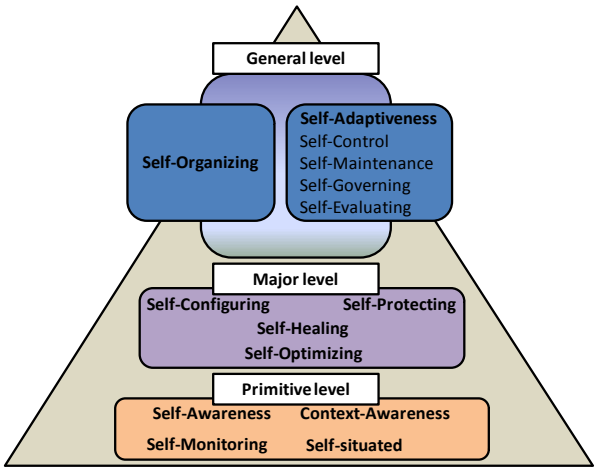


Figure 27. Hierarchy of Self-* properties

General level: This level is composed of self-organizing and self-adaptiveness properties. On one hand, self-organizing is a bottom-up approach where the local spontaneous interactions among elements inside the system give as result an ordered global behaviour. However, the elements are not aware of the emerging global behaviour but only of their local behaviour, as those can only see their neighbours. On the other hand, self-adaptiveness is a top-bottom approach where a global objective is given to the system, which adapts to meet that objective. Self-Adaptiveness is composed of Self-Managing, Self-Governing, Self-Maintenance, Self-Control and Self-Evaluating properties.

Major level: This level is composed of the aforementioned self-* properties (Self-Configuring, Self-Optimizing, Self-Protecting and Self-Healing).

Primitive level: This level is composed of self-monitoring, self-situated, self-awareness and context-awareness properties. Self-awareness means that the system is aware of its own state and its behaviour, while context-awareness means that the system is aware of its surrounding environment.

This hierarchical view provides us with a more accurate idea on the properties of a self-healing system and the metrics that can be used to measure its performance.

3.2.4 Properties of a self-healing system

(Psaier & Dustdar, 2011) state that self-Healing systems inherit their properties from Fault-tolerance, self-stabilizing approaches and survivability. In general, those approaches are not recovery-oriented and are intended to act in a reactive manner and secure the most important systems. Nevertheless, in recovery-oriented approaches the goal is to recover the system and to reestablish the operational state as it was before the disruption.

Fault-tolerance properties: Fault tolerance properties are those identified in fault-tolerant systems. Fault-tolerance is the property of being resilient to uncorrelated faults, as defined by Sterbenz. et. al. Uncorrelated faults mean those faults that do not imply services outages. However, fault-tolerant systems would not be able to prevent service failures in the presence of correlated faults, attacks and massive disasters. The main role of fault-tolerant systems is to handle transient failures and mask permanent failures to return to a valid state by mirroring their operations as a redundancy mechanism. A fault-tolerant system is based on backup components and the appropriate procedures to take over the failing one with no downtime penalties.

Survivability properties: Survivability properties are those identified in survival systems. Sterbenz et. al. define survivability as resilience to correlated faults, the capability of providing a given service without any performance deviation in presence of correlated faults. Survivable systems classify their subsystems as a function of their overall indispensability, in order to maintain only the essential services and turn off the non-essential ones in case of an attack or malicious attempt. Once this threat is removed from the system, non-essential services are recovered again. Survability approaches only contain failing components and secure the most important services, depicting a not optimal but functioning configuration. Fault-tolerance does not imply survivability (i.e. survivability comprises fault-tolerance), but survivability implies fault-tolerance properties.

Self-stabilization properties: Self-stabilization properties are those identified in self-stabilizing systems. Self-stabilizing systems are a non-fault masking approach for fault-tolerant systems based on the definition given in (Dijkstra, 1974) that says that a system is self-stabilizing if and only if it returns to a legitimate state regardless of the state on which it started. As identified by (Arora & Gouda, 1993) and (Psaier & Dustdar, 2011), two main properties characterize self-stabilizing systems: (1) are guaranteed to return to a valid state in a bounded time regardless of the interference (a.k.a convergence), and (2) when it reaches a valid state they attempt to remain at the same legitimate state (a.k.a closure).

3.2.5 Discussion

The quality factors of a self-healing system (availability, survivability, reliability, and maintainability) define the trustworthiness of a self-healing system, in other words, how the network and overlying services managed by the self-healing system are maintainable, reliable, and available. However, a self-healing system is not dependable, because a dependable system must include safety, and integrity properties in addition to the aforementioned properties. Nevertheless, this is the case of a resilience system, where defined that trustworthiness of resilience systems can be measured through dependability, security and performability, where dependability is in turn divided in reliability, maintainability, availability, safety, and integrity. Figure 28 shows how a self-healing ensures reliability, maintainability, and availability but not the rest of properties ensured by a resilience system.

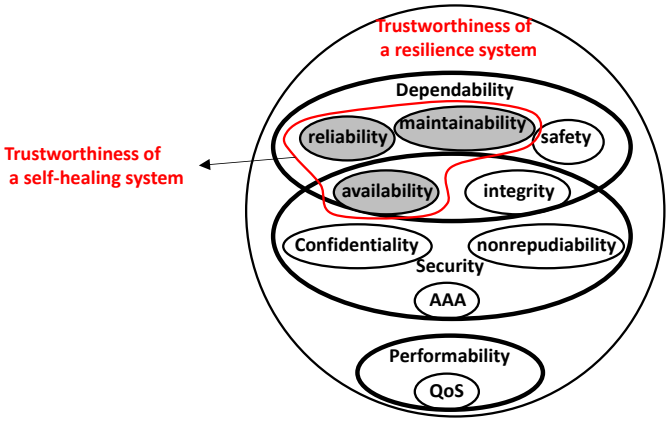


Figure 28. Comparison of quality factors ensured by a self-healing system and a resilience system

Concerning properties of a self-healing system, there are other two types of properties such as traffic tolerance and disruption tolerance. Traffic tolerance is defined in (Caini et al, 2011) and (Khabbaz et al, 2012) as resilience to abnormal traffic conditions. On the other hand, disruption tolerance is defined as resilience to connectivity disruptions due to harsh environmental conditions such as deep space and satellite communications characterized by the difficulty to maintain an end-to-end connection due to the power, battery constraints and high attenuation.

However, a self-healing system does not guarantee traffic tolerance and disruption tolerance properties. Nevertheless, a resilience system, as defined by (Sterbenz et al, 2010), can tolerate all these types of challenges, namely, survivability, fault-tolerance, traffic tolerance and disruption tolerance. It is important to highlight that fault-tolerant systems are resilient to uncorrelated faults and survivable systems are resilient to correlated faults, massive faults and attacks. Figure 29 shows the challenges solved both self-healing systems and resilience systems, where it can be seen how self-healing properties are guaranteed by resilience systems, but not in the other way round.

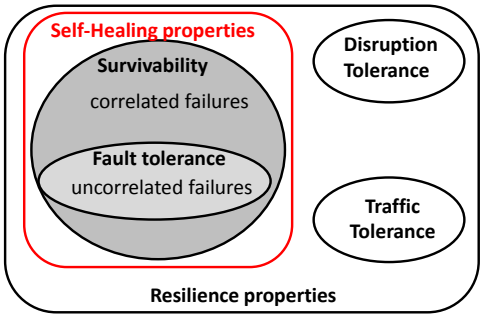


Figure 29. Self-Healing challenges from a resilience perspective

Psaier and Dustdar conclude that a self-healing system encompasses fault-tolerance, self-stabilization, and survivability properties, and if needed, human assistance. However, we could consider additional properties. The conclusions we can draw from the state of the art is that a Self-Healing system:

A self-healing system does not affect performance: The healed system should not perceive that the self-healing system is running.

A self-healing system must be reactive and proactive: It should detect both faults and failures. The definition of proactive in this thesis is treats faults and errors that could evolve into failures (the service is not disrupted at this stage) and reactive treats failures (the service is already disrupted).

A Self-Healing is self-aware: A Self-Healing system relies on Self-Awareness properties, as it is aware of the effects of their actions, by evaluating the impact of its actions on the environment. It perceives that it is not operating in a correct way by monitoring its own state (self-monitoring), and perform the necessary adjustments to remediate the situation.

A Self-Healing is context-aware: It relies on Context-Awareness properties, as it should be aware of what happens by using self-monitoring properties that continuously evaluate the state of its internal elements through sensors. It can evaluate the impact of their actions by sensing the environment before applying those actions and after.

A Self-Healing is recovery-oriented: Its goal is to mitigate the challenges by using a recovery strategy or a remediation strategy whilst the challenge is not mitigated.

A Self-Healing is human-assisted: Its goal is to supervise the self-healing system at all times by human operators and disable it when its response is inaccurate.

A self-healing system is trustworthy: It ensures reliability, maintainability, and availability. However, it is not dependable.

A Self-Healing has resilience properties: It has fault-tolerance, survivability properties and self-stabilizing properties. However, a self-healing system does not guarantee traffic tolerance and disruption tolerance.

3.3 Self-Healing architecture

3.3.1 Control-loop architecture

In this section, we present the architecture of a self-healing system, their corresponding control-loop and their tasks. The architecture of a self-healing system relies on a control-loop, which implies a continuous interaction between a manager system and a managed system. The managed system can be any kind of resource (hardware of software) susceptible to be controlled and affected by a manager. The sensor and actuator of the manager is the physical interface through which it can interact with the managed system.

A control-loop is a continuous and sequential process of an input data received from a sensor. This sequential process consists of a set of chained tasks that process an input, taken by a sensor, and it is sent through the effector. The self-healing system embeds and implements that loop so it manages the data-flow among the

different tasks. This data-flow is a continuous interaction between the manager and the managed system through sensor and effector. From now on, we will refer to manager system as the self-healing system and to the managed system as the healed system.

We first focus on the MAPEK autonomic control loop to derive its particular case of self-healing control-loop and corresponding tasks. Kephart and Chess defined the MAPEK loop (Monitor, Analyse, Plan, Execute and Knowledge). This control-loop is implemented by an autonomic manager. These authors defined the autonomic element, as a combination of the autonomic manager and the managed element. The autonomic manager is an intelligent agent, very similar to the agent concept in Artificial intelligence field, as defined by (Russell & Norvig, 2003) as "those who can perceive their environment and perform actions".

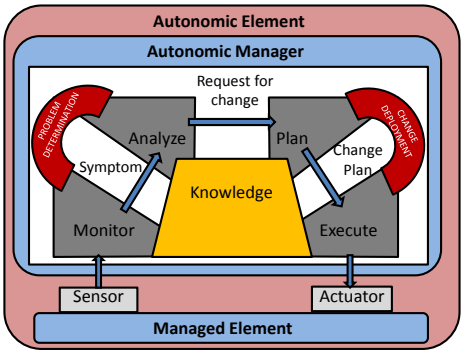


Figure 30. Autonomic manager

The MAPEK control-loop implemented by the autonomic manager is composed of four essential tasks, namely: monitor, analyse, plan, and execute, shown in Figure 30. It can be seen a continuous data-flow of knowledge which is continuously transformed among the different tasks, Knowledge is produced n and consumed by the different tasks of the control-loop. The monitor task acquires data from the environment to be later processed for analysing, the analyse task determines whether the monitored information must follow certain action, the plan task choses an organised set of actions according to the information given by the diagnosis task, the execute task executes the selected plan on the managed element.

Self-Healing control-loop is a particular case of MAPEK control-loop. In fact, the Self-Healing control-loop **monitors** the data captured from the environment by the sensors and obtains data to be **analysed** and processed to **detect** degradations or faults/failures. It **plans** a clear and organized strategy from the diagnosis performed on the analysed symptoms, and finally it suggests one or several actions to be **executed** in the healed system to recover or remediate the abnormality. The Self-healing control loop architecture can be seen in Figure 31.

Depending on the level of autonomics, those actions are directly executed by the self-healing system itself, or those are suggested to a human administrator who will validate after having checked they are appropriate for solving the current problem.

In conclusion, a self-healing control-loop is composed of three main blocks:

Detection: The aim of this block is to maintain the system in stable and healthy conditions to provide optimal performance. Otherwise, if any malfunctioning is detected, the diagnosis task is launched to determine the root cause behind that fault or failure and launch the appropriate recovery actions to solve that specific fault of failure.

Diagnosis: The aim of this block is both understand why a failure or a fault occurred in the network, but also why and how a fault could compromise the system.

Recovery: The aim of this block is to restore the normal operational state by returning back to those service quality levels previous to the disruption.

Figure 31 shows a self-healing system composed of these three blocks detection, diagnosis, and recovery. In this thesis, the healed system is a network element e.g. a router, an OpenFlow switch, a database. The information retrieved from the healed system depends on the type of element, as well as the type of recovery action sent to the healed system.

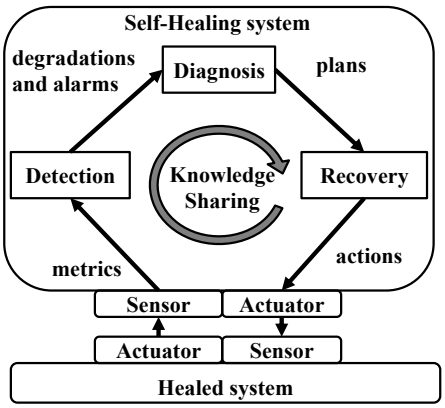


Figure 31. Self-Healing system architecture

A self-healing system receives raw data (e.g. logs or performance measurements) from the healed system. Secondly, these data are analyzed to detect degradations of performance measurements and failures which are usually referred as alarms. The self-healing system can also detect faults. In case of failure, the root cause is diagnosed and a proper recovery algorithm or action is executed to solve the problem. This execution is translated into actions upon the healed system to change its behavior. A self-healing closed-loop receives information from sensors and acts over their elements to be healed with the actuators.

There is a huge diversity in terms of architectures (hierarchical/flat, distributed/centralized, multi-agent/single-agent, decoupled/highly coupled), target context, algorithms used for each self-healing task, nature of the recovery approach (strong or weak adaptation), monitoring granularity (semi-wide or wide-network), monitoring persistence (continuous/adaptive, proactive/reactive) and degree of autonomics. The right choice of implementation may depend on technical constraints or best practices coming from SDOs (Standards Development Organization) recommendations.

For instance, if each healed system requires a dedicated self-healing mechanism we are talking about a distributed self-healing architecture. This is the case where the elements to heal are really different (database, router, switch, logical applications, etc.), what makes different the detection and recovery blocks, due to the difference of data retrieved and recovery actions sent to the healed equipment. This architecture alleviates the load to upper layers and is more scalable. On the other hand, there are cases where centralized self-healing architectures are more appropriate, regardless of the fact that, may be seen as less scalable, as one single self-healing system must heal different equipment almost simultaneously. This is the most suitable architecture when the healed system is the whole network topology seen by a SDN controller.

Depending on whether the self-healing system is embedded inside the healed system (highly coupled) or is externally connected (decoupled), the self-healing mechanism can be reused for monitoring/managing more equipment or not. It is desirable for the self-healing system to be decoupled from the managed system, due to scalability and maintainability issues.

Sterbenz et. al. propose a two-fold continuous process based on two control loops with very different time-scales to ensure resilience. This double control-loop, known as $\boldsymbol{D^2R^2} + \boldsymbol{DR}$, has two strategies: first to heal the network at run-time with short-term measures by a real-time control loop $\boldsymbol{D^2R^2}$ and second to refine and improve the network with medium-term or long-term measures by a background loop $\boldsymbol{DR}$. Those long-term measures may be architectural changes or the inclusion of new mechanisms and algorithms, or new protocols to face new vulnerabilities and challenges. Both control loops are continuously interacting in order to take into account the feedback of the $\boldsymbol{D^2R^2}$ control loop to foresee new updates and upgrades of the resilience system.

$\boldsymbol{D^2R^2}$ **Real-time control-loop:** This loop runs at run-time to solve the urgent and immediate abnormalities in the network. This control-loop is conceived for the operational phase, and it is composed of defend, detect, remediate, as well as recover functional blocks. We consider that this loop must be based on autonomics to react auto-

matically and in time (ms or ns) to the faults and failures occurring in the network. It is composed of the following tasks:

- **Detection of challenges and failures:** This task is similar to the failure detection task of self-healing systems, where the failure is discovered and eliminated. In both cases, this is a reactive approach, where the resilience or the self-healing system just waits for the service to be in a failure state and then triggers the appropriate actions to restore the system.

- **Defense against challenges:** This task is similar to maintenance of health of self-healing systems, where the fault is preventively detected and eliminated. This task has a proactive connotation, where proactive detection approaches have to do with the anticipatory capability, instead of just waiting for the change to occur and react, the system is able to predict from the degradation symptoms and act in consequence to change the behaviour as soon as possible.

- **Remediation during challenges:** This task has as main goal to transition the system bring the system from an unacceptable state, what is considered broken state in a self-healing system. It is normally considered as a rapid recovery solution to minimize the adverse effect of the challenges. This task is not recovery-oriented because it does not imply a removal of the challenge and it coexists with it. Examples of remediation-oriented systems are fault tolerant, self-stabilizing, and fault masking systems.

- **Recovery to normal operations:** This task acts on the root problem, and it is considered in this thesis as equivalent to the recovery functional task given in a self-healing system. This task may imply maintenance actions such repairing actions, mostly human-based (e.g. repairing links, replacing hardware, manual resets, re-establish power supply, etc.) or modification actions such as upgrading hardware, updating software, adaptive mechanisms, or optimization algorithms.

$\boldsymbol{DR}$ **Background control-loop:** This loop refines the network by performing a deeper analysis of the vulnerabilities found and proposes improvements of the network at medium or long-run. This loop implies further analysis and diagnosis to understand why a fault or failure has occurred and propose future refinement and ameliorations of the network. This control-loop is a non-real time loop which should be human-assisted to supervise autonomic systems running at operational stage and refine and upgrade the network.

It is composed of the following tasks:

- **Diagnosis faults:** This task seems to be equivalent to the diagnosis task of a self-healing system. It is intended to perform post-mortem root cause analysis of faults, but it does not seem to cover failures. In this thesis, we focus on diagnosis of both faults and failures.

- **Refinement future behavior:** This task is to take the past events to refine the system. An example of this can take into account the bottlenecks identified by the diagnosis block and propose redundancy mechanisms to alleviate them.

3.3.2 Self-healing system state diagram

First (Gosh et al, 2007) and afterwards (Psaier & Dustdar, 2011) considered that a self-Healing system can be at any of three states: normal, broken and degraded. This diagram can be seen in Figure 32. A normal state is when the system fulfils its conceived mission and meets certain observable and quantifiable quality parameters. A degraded state is there is a deviation on the specified parameters of the system provoking that its mission is not fulfilled and it is thus compromised. The system is *in a faulty state*, i.e. the system is still operational, but does not perform as conceived. A broken state is the managed system is not operational at all. The system is *in a failure or broken state.*

The transitions among states describe possible state sequences traversed by a self-healing system over time. Some transitions are provoked by faults and failures but others are provoked by corrections performed by functional tasks such as recovery actions. Psaier and Dustdar and afterwards Gosh et. al. define six state possible transitions grouped into three functional tasks, which are defined hereafter: maintenance of health, detection of system failure, and system recovery.

Maintenance of health: Its aim is to monitor certain parameters or performance of the system and maintain the operational parameters inside proper limits of operation. Many researchers have been dealing with different strategies such as redundancy components, diversity or with probes to assess its state, for example.

Detection of system failure: Its aim is to discern between a normal state and a degraded state in a precise and rapid manner, a.k.a. *"self and non-self states determination"*, defined by Gosh et. al. This difference between normal and degraded state is known as the fuzzy zone. This depends on a threshold to quantify when the system starts performing at a sub-optimal level. However, the definition of a precise threshold requires knowing in advance the nominal conditions of the system.

System recovery: Its aim is to set the system back to a normal state, regardless of the current state is degraded or even broken. Once the self-healing system is aware is not in a normal state, it has to restore that disruption, and it uses a set of recovery techniques to bring the system into a normal state. Notice that system recovery always restores the system to a normal state, and does not transitions to an intermediate state.

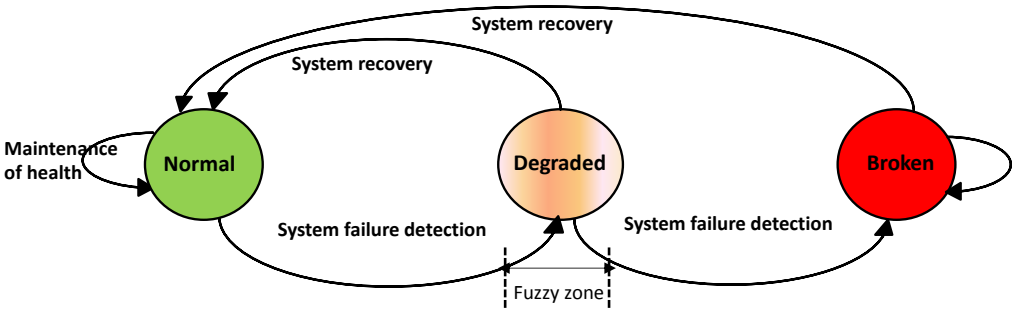


Figure 32. Self-healing state diagram

From a resilience perspective, resilience is tightly coupled with the notion of service as the goal of a resilience system is to avoid any deviation of a service from its functional specification. The operational state is defined by a set of k operational metrics taken from the network $N = \{N_1, \dots, N_k\}$, while the service state is defined by a set of l service parameters $P = \{P_1, \dots, P_l\}$.

The authors define resilience at a given boundary between two adjacent layers, and consider a multi-layer resilience space where resilience of one layer is a pre-requirement for those services acting on upper layers to be resilient. This concept was further explained by (Smith et al, 2011), where the authors explained that a resilient routing needs a resilient topology (e.g. with the appropriate and enough level of redundancy deployed).

Figure 33 shows the state diagram of a resilience system, where the different transitions among states are shown. In the next section, we describe the different tasks responsible for those transitions.

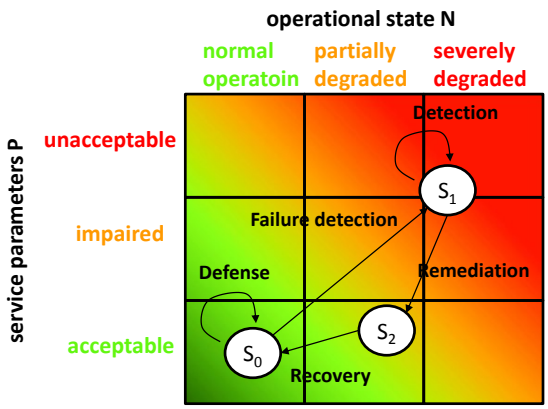


Figure 33. Resiliency state diagram

On one hand, the service state is defined as normal, degraded and unhealthy.

- **Normal state:** This state means that the service is not deviated from its functional specification.
- **Degraded state:** This state means that the service is degraded, and somehow it does not meet its functional specification.
- **Unhealthy state**: This state indicates that the service is severely deviated from this functional specification.

On the other hand, we can see the service parameters states, namely, acceptable, impaired, and unacceptable. These states refer to those KPI or indicators of the different operational parameters involved in a given service.

3.3.3 Restoration stages of a self-healing system

In this section, we describe the restoration stages of a self-healing system, having taken into account the aforementioned diagram state and associated functional tasks and mechanisms from a resilience point of view and a self-healing point of view.

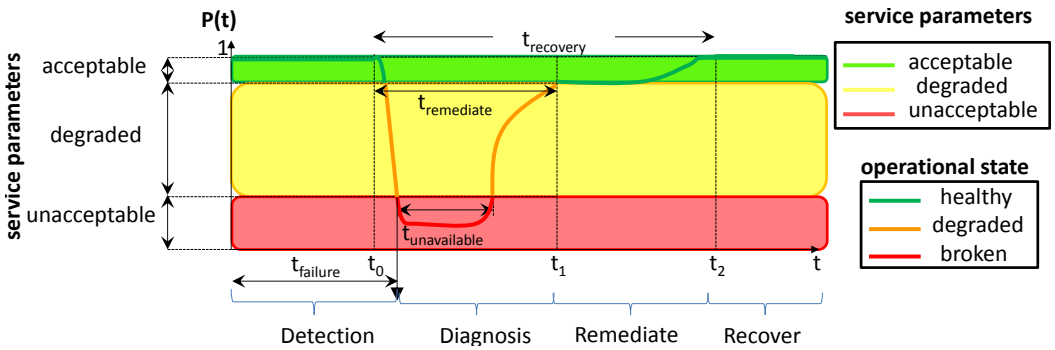


Figure 34. Stages of a Self-Healing system

The state of a given service is described by service parameters, which describe the health and operational state of that service through a set of KPIs (Key Performance Indicators). The service can transition among three states namely, acceptable, degraded, and unacceptable.

- **Acceptable state:** the service is fulfilling its functional specification.
- **Degraded state:** the service is degraded from the functional specification.
- **Unacceptable state:** the service cannot be delivered due to an outage.

The network is composed of different network resources and protocols interacting and cooperating. Each of these resources and protocols are characterized by a set of operational states. This depicts a kind of overall metrics to depict the health of the network infrastructure providing the service. This operational state of the network may traverse three states, healthy (in green), degraded (in orange), and broken (in red).

Figure 34 depicts the performance $P(t)$ of a given service (characterized by service parameters) running in a given network infrastructure (characterized by some operational state). It can be seen the four phases of a resilience system, namely detection, diagnosis, remediation, and recovery.

While the service performance is optimal $P(t) = 1$, the self-healing mechanism is in a maintenance of health surveyance mode to force the network to be in a healthy state and keep the service in an acceptable state. Once the detection mechanism finds a fault, the diagnosis and remediation tasks should find a quickly answer to avoid this fault turns into a failure and affects the service parameters in such a way that the service transitions to a degraded or unacceptable state. $t_{unavailable}$ depicts that amount of time when the service is not ready for use.

The remediation task is to bring the service performance out of the unacceptable state, while the challenge is still in the network. This task is characterized by $t_{remediate}$, the necessary amount of time for the self-healing system to bring the system out form the unacceptable state.

On one hand, the remediation phase does not guarantee an optimal service performance but an acceptable service level, because its aim is to minimize the impact on the service delivery while the challenge is present in the network. On the other hand, the recovery phase does ensure an optimal normal operation and a return of the operational parameters to the healthy state and to the service parameters to an acceptable state.

In this figure, several parameters describe the performance of a self-healing system.

Mean time to repair (MTTR): This is the average time to repair the managed system or service. Recovery time $t_{recovery}$ comprises the remediation time tremediate. The MTTR is defined as an average between trecovery times: $MTTR = \frac{\sum_{i=1}^{N} t_{recovery}^{(i)}}{N}$

Mean Downtime (MDT): This is the total time the system is non operative and cannot fulfill the service. The corrective actions, recovery, remediation, diagnosis and detection are comprised in $t_{unavailable}$. MDT corresponds to the average between tunavailable times $MDT = \frac{\sum_{i=1}^{N} t_{unavailable}^{(i)}}{N}$

Mean Time To Failure (MTTF): This is the average time it takes a system to fail. MTTF corresponds to the average between tunavailable times where tfailure is the instant of time when the system enters in a broken state (red) and the service is in an unacceptable state. $MTTF = \frac{\sum_{i=1}^{N} t_{failure}^{(i)}}{N}$

Mean Time Between Failures (MTTF): This is the average time between two consecutive failures, defined as $MTBF = MTTF + MTTR$

These parameters are tightly related to the aforementioned quality factors optimized by a self-healing system. For instance, Availability is mathematically described as $A = MTTF/MTBF$.

In conclusion to this point, the self-healing mechanism takes as reference the state of the operational parameters of the network infrastructure, while resilience takes as reference the state of the service delivered.

3.3.1 Discussion

We first compare the aforementioned resilience control-loop with the self-healing control-loop and then we compare the self-healing state diagram and the resilience state diagram and we propose a state diagram for a self-healing system that comprises additional functional tasks and mechanisms to ensure fault and failure management in programmable networks.

Interestingly, the resilience double control-loop has some similarities with the self-healing control-loop. Indeed resilience control-loop extends the self-healing blocks, with remediation, defense, or refinement, blocks. The self-healing control-loop is a single autonomic control-loop with detection, diagnosis, and recovery. In contrast, the control-loop seems to be more complete as it takes into account design aspects in the background control-loop and operational aspects in the real-time control-loop.

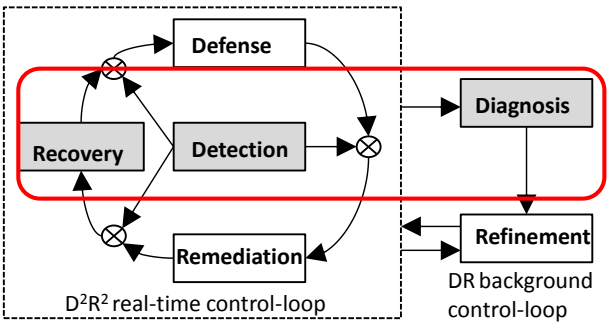


Figure 35. Similarities between the Self-Healing control-loop (in red) and the D^2R^2+DR control-loop

Interestingly, the diagnosis block is not included in the real-time loop, but in the background control-loop as if the diagnosis was considered an offline task, which in self-healing seems to be more at run-time. In this thesis, we advocate by an online diagnosis able to pinpoint the root cause in a rapid manner to launch the remediation or recovery actions based on that root-cause. Figure 35 shows the resilience double-control loop and the self-healing control-loop and its blocks in red.

The main difference is the dimensionality (number of state variables) of both diagram states. This comparison can be seen in Figure 36. Firstly, the self-healing system diagram state is unidimensional and it refers in principle to the states of a generic system, without considering the notion of service. Secondly, the resilience diagram state is bidimensional as it refers to service state and its parameters and to the operational state of the network encompassing the infrastructure and protocols. However, in this thesis, the concept of self-healing is applied to ensure resilience of both network and services deployed, that is why we also show the resilience diagram state that considers both aspects.

In the self-healing diagram state, the transition from broken to broken state does not have any associated functional task. We consider that, once in a broken state, other failures as consequence of the first failure may arise and, if not treated, they could affect more equipment.

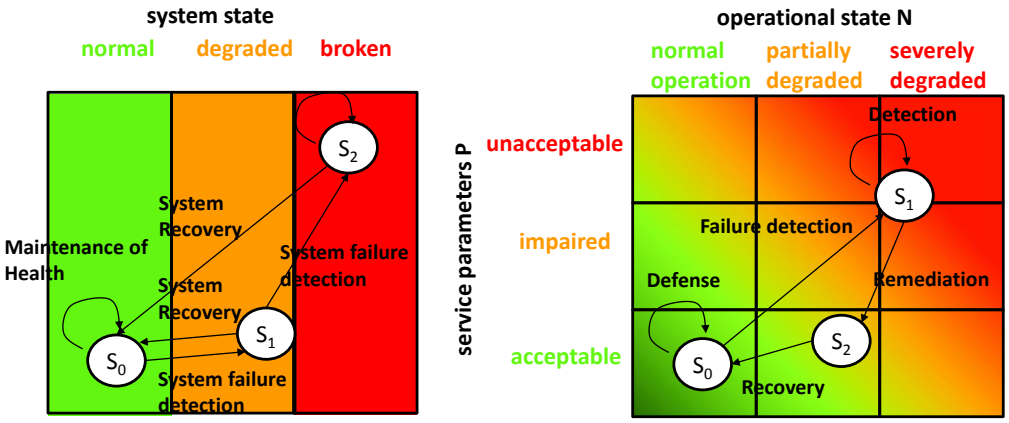


Figure 36. Comparison between self-healing and resilience diagram states

There are only recovery tasks that take the system from a broken state to a healthy state and from a degraded state to a healthy state. However, there is not any remediation task to take the system from a broken state to a degraded state. One example is when the self-healing system does not have more choice than isolating some faulty elements and keep the system in a degraded state while the malfunction is being diagnosed. Remediation task is a temporary solution that returns the system to a degraded state (instead to a broken state) while the abnormality is being restored.

The diagram state always transitions from a healthy state to a broken state by visiting an intermediate degraded state, which in our perspective is not true, as there could be direct transitions from healthy states to broken states. We consider that, depending if it is a fault or a failure, this degraded state is visited or not. **Failures** directly make the self-healing system transition from a healthy state towards a broken state, as the service is affected, without visiting a degraded state. **Faults** make the self-healing system transition from a healthy state towards an intermediate degraded state.

This diagram state does not consider the level of proactivity of each functional task. We consider in this thesis that proactive techniques prevent a failure while reactive ones are launched after the failure has been detected.

As a result, we propose the following self-healing state diagram in Figure 37, where we have extended the aforementioned functional tasks: maintenance of health, system recovery, system remediation, and detection of system failure. In turn, each functional task comprises a set of mechanisms as shown in Table 4.

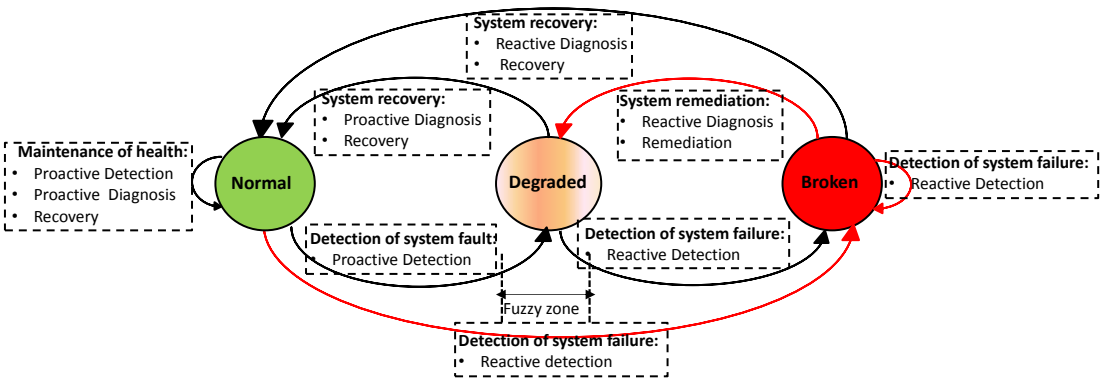


Figure 37. Proposed diagram state of a Self-Healing system

Table 4. Proposed functional tasks and corresponding mechanisms

State transition	Functional Task	Mechanisms used
normal-normal	Maintenance of health	Proactive Detection Proactive Diagnosis Recovery
normal-broken	Detection of system failure	Reactive detection
normal-degraded	Detection of system fault	Proactive detection
degraded-normal	System recovery	Proactive Diagnosis Recovery
degraded-broken	Detection of system failure	Reactive detection
broken-broken	Detection of system failure	Reactive detection
broken-degraded	System remediation	Reactive diagnosis Remediation
broken-normal	System recovery	Reactive diagnosis Recovery

In this proposed state diagram, all the restoration mechanisms are preceded by a type of diagnosis (reactive or proactive) depending if the diagnosis mechanism is diagnosing a failure, a fault, or an error. Table 4 summarizes all the considered state transitions, as well as the functional tasks for each state transition and the mechanisms required to force each transition.

3.4 Self-healing mechanisms

This section reviews the self-healing mechanisms, defined here as those mechanism ensuring the self-healing functional tasks detection, diagnosis, remediation, and recovery.

3.4.1 Detection mechanisms

Two types of proactive detection mechanisms are considered in this section: failure prediction mechanisms and defensive mechanisms.

Failure prediction mechanisms try to anticipate to future service failures by analysing performance metrics of resources and services. Sterbenz et. al. propose to assess when the system is challenged by detecting deviations on service requirements, or by understanding the context of challenge to perform more accurate remediation mechanisms. By contrast, (Cholda et al, 2007) classified the proactive detection mechanisms into signal degradations (e.g. dispersion in fibers, Signal-to-Noise ratio, Bit Error Rate, etc.), or quality degradation (e.g. high delay, low throughput, etc.).

(Sterbenz et al, 2010) and (Salfner et al, 2010) agree on the fact that error detection is key to prevent failures. An error is a deviation, which is monitored and quantified. In this sense, (Salfner et al, 2010) focus on online failure prediction mechanisms to avoid future service failures as consequence to errors or faults. The authors propose four main mechanisms: tracking of failures, symptom monitoring, or error reporting detection, and undetected error auditing. Tracking of past mechanisms failures predict future failures by analyzing past failures. A first approach is to estimate the probability distribution of the time to next failure from the current set of failures. A second approach is co-occurrence mechanisms, where two failures may follow a special or temporal distribution, which may repeat in future. Symptom monitoring mechanisms analyze the errors symptoms such as the amount of free memory in a periodic basis. Error reporting detection mechanisms analyze errors reported by some entity and assess the risk of failure. For instance, this error may be due to a memory violation or an exception thrown by a given application. Undetected error auditing mechanisms search for incorrect states in the system, regardless of the fact those errors impact the system at current time.

Defensive mechanisms try to anticipate to future service failures by preventing the system state from moving outside a normal state. If we consider Sterbenz's perspective, those mechanisms would act in two levels: passive

defensive mechanisms to prevent the challenges from becoming active faults, and active defensive mechanisms to prevent errors from passing to the operational state. On the one hand, active defensive mechanisms are mostly security mechanisms and error handling mechanisms based on cryptographic algorithms, firewalls, or trust boundaries. On the other hand, passive defensive mechanisms would be in the prevention stage according to (Tipper, 2013) and their main goal is to defend the system from threats and challenges to maintain the health of the system by using structural defences that use the redundancy and diversity deployed at network design stage. We consider redundancy and diversity techniques as both passive defensive mechanisms but also as the base of remediation mechanisms. For instance, a protection scheme, a proactive mechanism that pre-computes an alternative path (backup) based on redundancy provided in the network, may be used as defensive mechanism. Examples of protection mechanisms: are Linear Automatic Protection Switching, Self-Protecting Multi-paths, protection rings, redundant trees, resilient routing layers, protection cycles, and p-cycles.

As identified by (Gosh et al, 2007) mechanisms for maintaining redundancy imply the automatic replication of network resources to ensure redundancy to transmit the information. Those techniques are inspired by biological mechanisms like cell division or self-assembling properties. For instance, (Nagpal et al, 2003) propose a programming methodology able to self-assemble complex structures from identically programmed agents that interact among each other locally.

Diversity is about providing redundant alternatives to elude failures provoked by challenges. We cite several types of diversity:

- **Temporal diversity:** It refers to send the same information duplicated in time
- **Spatial diversity:** It refers to send information simultaneously through redundant paths (e.g. 1+1 global/local protection scheme)
- **Information diversity:** It refers to send redundant information but differently coded from the original (e.g. forward error correction, ARQ (Automatic Repeat-request))
- **Implementation/operational diversity:** It refers to implement the same functional behaviour differently to avoid that the same fault causes the same results
- **Medium diversity:** it refers to send information through different media (e.g. optical and wireless accesses).

These mechanisms are reactive because that act upon a service failure, when the system is in a broken state and the service state is inacceptable, therefore its availability is compromised. Their goal is then to react to failures by analysing the alarms indicating failures and then launching the appropriate countermeasures as soon as possible to revert the situation and take the broken state to a normal or, at least, to a degraded temporary state. As examples of such mechanisms, Gosh et. al. identify several mechanisms for detecting failures such as something amiss, which consists of finding something missing that was supposed to be in the system during a normal state, and foreign element notification, where new elements not supposed to be during a normal state are notified. By contrast, (Cholda et al, 2007) considered that the reactive detection mechanisms are those related to the physical layer observed by the symptoms such as loss of light in fibers, loss of signal or modulation, or loss of clock, for instance.

3.4.2 Diagnosis mechanisms

Two types of diagnosis are considered, proactive and reactive mechanisms, both defined hereafter.

Proactive diagnosis mechanisms are in charge of identifying faults and errors in the network infrastructure in order to predict and avoid any future service failure and avoid the system to fall into a degraded or a broken state and keep it in a normal state. Those mechanisms are also known as online failure prediction mechanisms, identified by (Salfner et al, 2010). Given a misbehavior in the network, online failure prediction is to predict future failures from the present observations of that misbehavior (fault or error), as shown in Figure 38. A proactive diagnosis mechanism means that it diagnosis in advance to the service failures.

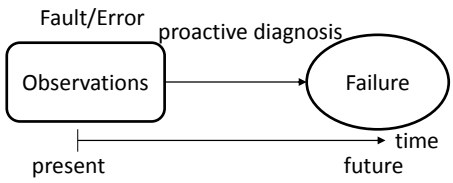


Figure 38. Proactive diagnosis

Reactive diagnosis mechanisms act once the service failure has occurred, as shown in Figure 39. Their goal is to understand which fault caused the given failure observed at present time. Nevertheless, it is important to highlight that the result given by those mechanisms (i.e. the root cause analysis responsible for that failure) can be used to foresee degradations and failures in the future.

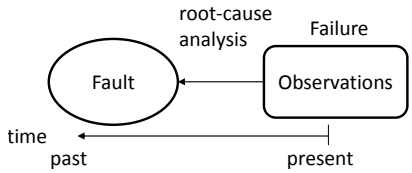


Figure 39. Reactive diagnosis

3.4.3 Recovery and remediation mechanisms

When the self-healing system detects that the service is not available, it triggers the appropriate recovery or remediation mechanisms to restore it as soon as possible. Those recovery actions can be based on redundancy mechanisms that react to failures. For instance, a reactive restoration scheme calculates a path on demand once the service failure occurs. Of course, this reactive mechanism could be launched in the event of a simple fault, but is considered proactive because it is launched before the service failure appears. Example of restoration mechanisms are ECMP (Equal-cost Multi-Path routing), MPLS (Multiprotocol Label Switching), or IP Fast Reroute, among others.
As said before, a recovery mechanism sets the system back to a normal state, while a remediation mechanism sets the system back to a degraded state.

Recovery and remediation mechanisms have diverse particularities according to several criteria, detailed hereafter:

Restoration scope (segment and domain): the network segment to restore can be local (e.g. a link, or a node) or global (e.g. the entire path). This segment can be a single domain (intra-domain) or multiple domains (inter-domains). Recovery and remediation mechanisms are more popular for intra-domains, due to the reluctance of each domain's owner to share information from its domain with others.

Targeted equipment: The type of device will determine the recovery or remediation mechanism used. For instance, we may use rollback and roll forward for databases and redundancy for link failures.

Level of dedication and redundancy of spare resources: Spare resources replace the malfunctioning elements. Those spare resources may be dedicated (e.g. 1:1 one dedicated backup per malfunctioning resource) or shared (e.g. 1:2 one shared backup between two malfunctioning resources).

Layers of intervention: In multi-layer networks, coordination between recovery and remediation procedures is necessary. For instance, SDN and NFV architectures are inherently multi-layered. Three types of coordination are possible: bottom-up, top-down or a hybrid one, combining advantages of the two first.

Duration of abnormality: Transient abnormalities may disappear by themselves and may not be necessary to use recovery mechanisms. However, permanent abnormalities will be always there unless are fixed using recovery mechanisms (e.g. use of maintenance actions).

Type of challenge: The self-helaing system may trigger a recovery or a remediation mechanism according to the type of abnormality (i.e. fault, error or failure).

Strength of the restoration action: Depending on the type of malfunction, the recovery/remediation action can be strong or weak. Strong actions require aggressive and invasive approach (e.g. structural changes in the network, components replacement, redundancy, diversity, etc.). Weak actions are related to changing data parameters of the managed resource to achieve optimization, adjusting certain parameters to meet certain goals (control theory approaches), load balancing mechanisms or even changing the algorithms to achieve these autonomic properties at run-time.

3.4.4 Discussion

Once the failure appears, the self-healing system launches a remediation action to bring the system to an intermediate and temporary state, and, in a parallel way, it launches a post-mortem reactive diagnosis mechanism in order to identify what happened and how to avoid it in the future. Once the self-healing system knows the originating root cause, it can feed it back to a refinement block (such as the shown in the resilience control-loop), able to redesign the network infrastructure to ensure the right level of redundancy to avoid this failure. This redesign is a set of strong actions that redefine and ameliorate the redundancy and diversity of the network, or the conception, implementation and the put in place new mechanisms in the network.

Strong actions will have an impact on the service downtime or performances due to the time taken to perform this action, so those actions should be accompanied of remediation actions that bring the system back to a temporary degraded state while the strong action is being applied. Strong actions are related to maintenance actions, e.g. replacement of the optical fibre to your home. It is more desirable a weak adaptation action instead of a strong action, but it depends on the gravity of the detected failure and it corresponds to the recovery process to decide if it is necessary to go to the extreme of such a solution. In contrast, weak actions are seen here as a tuning or adaptive based on control-theory approaches.

In conclusion, recovery mechanisms and remediation mechanisms differ in the type of mechanism and algorithms or techniques used, but fundamentally, those differ in the final state where the system is after their application. For instance, a recovery mechanism brings the healed system back in a normal state, while a remediation mechanism brings the healed system back in a degraded state.
In general, remediation mechanisms rely on redundancy and diversity, where the root cause of the problem may not be removed but at least is isolated or alleviated. Concretely, a remediation mechanism builds on top of redundancy and diversity by defining a procedure that send the correct policies or orders to activate that redundancy and diversity deployed in the network. For instance, a remediation mechanism may simultaneously transmit information through both the working resources and the spare ones by combining redundancy and diversity (1+1 global/local protection) or it may switch from the malfunctioning resources to the spare resources when working resources fail. Those redundant resources may be dedicated (1:1 global/local protection) or shared (shared M:N global/local protection). In addition, ther are protection schemes that pre-plan and previously allocated those resources to react faster to the fault or restoration schemes that simply calculate the available redundant resources on demand in the presence of a failure. Resource dedication is the most classical redundancy approach - and certainly not the most efficient – and relies on dedicating spare equipment for each resource and re-route the flow to the spare one when a fault occurs. Nevertheless, this a priori fixed resource allocation can be reoriented to a dynamic resource allocation, relying on optimization approaches such as control theory-based, utility-based or reinforcement learning algorithms.

Examples of remediation mechanisms are dynamic re-routing, self-healing rings, node/link/path restoration, node/link/path protection, trunk diversity, multiple homing, or p-cycle protection. In addition, migration based techniques such as pre-emption based process migration, VM migration, or stop-and-copy VM migration to enable dynamic migration of resources in a similar way to a redundancy-based technique, are examples of remediation mechanisms.

For instance, challenges like increases in the traffic load, can be resolved by these remediation or recovery mechanisms: overload mechanisms, load balancing mechanisms, scaling-up strategies, scaling-out strategies or congestion control mechanisms to handle that unexpected amount of traffic. For instance, virtual services or applications may be relocated onto other physical nodes on the fly for this purpose. Remediation mechanisms to deal

with extreme conditions such as deep-space communications and terrestrial networks are based on specific forwarding mechanisms, which are further explained in the surveys led by (Caini et al, 2011) and (Khabbaz et al, 2012). Examples of those forwarding mechanisms are opportunistic forwarding, probabilistic forwarding, encounter-based forwarding, among others. In addition, the authors point out the definition of new application layer protocols or transport layer protocols to face disruption in such harsh environmental conditions.

Alternatively, Gosh et. al. proposed a set of mechanisms that can be used for recovery and remediation, such as the ones briefly explained hereafter:

Redundancy for healing mechanisms: It refers to the capability of the components to replicate in order to replace dead neighbors and achieve the total recreation of the entire structure. This may or not be based on self-organizing or self-assembling approaches such as the works from (Nagpal et al, 2003).

Repair plans for healing mechanisms: It refers to policies that firstly determine the root cause to repair later the problem. Those policies are dictated according to a diagnosis block

Event-based action mechanisms: These techniques are very similar to the previous ones, but the recovery action here is triggered by a given event.

Voting method mechanisms: It refers to a making decision approach that determines if a fault replica in a group of processes has impact on the rest of the processes.

Error handling mechanisms: Recovery can be based on error handling mechanisms that eliminate errors or on fault handling mechanisms to prevent faults from being newly activated. As example of these mechanisms, we have Rollback and roll forward mechanisms that create an image of the system, which is loaded in the event of errors, which ensures that the network is back into an error-free state.

3.5 Self-healing algorithms

This section reviews and classifies the algorithms that could be implemented inside a self-healing system according to three different criteria: per application domain, per objective, and per self-healing functional block (detection, diagnosis, and recovery).

3.5.1 Algorithm classification per application domain

The application domain is defined as the context where the algorithm is applied and it is given by the network technology, the network equipment, the communication protocol, among many others factors. For instance, in this thesis, the application domain is SDN and NFV combined infrastructures. We provide with a possible algorithm classification per application domain in Table 5.

We can observe that both Bayesian Networks and control theory approaches are extensively use in many different contexts and applications domains. On one hand, Bayesian Networks have been used in such different contexts such as:

- IP Multimedia Subsystem (Hounkonnou, 2013),
- VPN ((Bennacer et al, 2012), (Bennacer et al, 2013), and (Bennacer et al, 2015)),
- Software-Defined Networking (Al-Jawad et al, 2015),
- Wireless Sensor Networks (Yunkhao et al, 2010),
- Electrical Power Systems (Mengshoel et al, 2010),

IT systems/ Enterprise Networks ((Bahl & Chandra, 2007), (Zhao, 2008), (Kandula & Mahajan, 2010)), and (Cooper & Herskovits, 1992)

- GPON-FTTH Optical Networks (Tembo et al, 2015), and
- Bridged networks ((Steinder & Sethi, 2004), (Steinder & Sethi, 2002)).

In these contexts, Bayesian Networks is mainly used as a reactive diagnosis algorithm, but the same algorithm can also be used for ensuring QoS when forwarding in SDN. With this concern, (Al-Jawad et al, 2015) propose and describe BaProbSDN, a probabilistic-based QoS routing mechanism for Software Defined Networks. This approach is based on calculating the probability enough bandwidth on links by means of a Bayesian Networks based

algorithm and selecting the forwarding path based on those metrics. On the other hand, control theory approaches have been applied to many different applications domains, as example we can cite contexts such as:

- IT systems ((Storm et al, 2006), (Diao & Hellerstein, 2005), (Parekh et al, 2000), (Parekh et al, 2003), (Diao et al, 2002),),
- cached services ((Lu et al, 2001), and (Lu et al, 2002)),
- multimedia communication networks (Xiaoyuan et al, 2007),
- virtualized resources control (Padala & Hou, 2009),
- software-defined networks and network function virtualization (Akhtar, 2016).

It can be also seen that hybrid techniques are also used to compensate the drawbacks of algorithms when used individually. A first example of hybrid algorithm is the combination of Case-based reasoning with Bayesian Networks, which was proposed by Bennacer et. al. in their works (Bennacer et al, 2012), (Bennacer et al, 2013), and (Bennacer et al, 2015). Case-based reasoning reduces the cost of inference for Bayesian Networks when the number of vertices is high, acting as an auxiliary technique to support Bayesian Networks inference and enhancing the scalability. Another example of hybrid algorithm is the combination of Artificial Neural Networks with digital signal processing techniques the application domain of optical networks such as Internet Protocol/Synchronous Digital Hierarchy (IP/SDH) and Internet Protocol/Wavelength Division Multiplexing (IP/WDM, Wavelength Division Multiplexing) given by (Marilly et al, 2002).

Table 5. Algorithms classification per application domain

Algorithm	Application domain	Reference
Control theory	IT systems/entreprise networks	(Storm et al, 2006), (Diao & Hellerstein, 2005), (Parekh et al, 2000), (Parekh et al, 2003), (Diao et al, 2002),
	Cached services	(Lu et al, 2001), (Lu et al, 2002))
	Multimedia communication networks	(Xiaoyuan et al, 2007),
	Virtualized resources control	(Padala & Hou, 2009),
	SDN and NFV	(Akhtar, 2016)
Bayesian Networks + Case-based reasoning	VPN networks	(Bennacer et al, 2012), (Bennacer et al, 2013), (Bennacer et al, 2015)
Bayesian Networks	IMS (IP Multimedia Subsystems)	(Hounkonnou, 2013)
	SDN	(Al-Jawad et al, 2015)
	Wireless Sensor Networks	(Yunkhao et al, 2010)
	Electrical Power Systems	(Mengshoel et al, 2010)
	IT systems/entreprise networks	(Bahl & Chandra, 2007), (Zhao, 2008), (Kandula & Mahajan, 2010)), (Cooper & Herskovits, 1992)
	Optical Networks (GPON-FTTH)	(Tembo et al, 2015)
	Bridged networks	(Steinder & Sethi, 2004), (Steinder & Sethi, 2002)
Hidden Markov models	Intrusion detection systems	(Univerself, 2013), (Tcholchev et al, 2010)
Clustering	IT systems/entreprise networks	(Vaarandi et al, 2015)
Self-Organizing Maps	IP networks	(Univerself, 2013)
Fuzzy reasoning	IP networks	(Univerself, 2013)

Artificial Neural Networks	Virtual networks	(Univerself, 2013)
Feature vectors	IP networks	(Kimura et al, 2015)
Feature adaptation+ Structural Corresponding Learning	IT systems/entreprise networks	(Zhou et al, 2015)
Support Vector Machines	Electronic Gaming machines	(Butler & Keselj, 2010)
Machine learning techniques	Core networks	(Univerself, 2013)
Game-Theory	Multi-agent systems	(Buchegger & Boudec, 2002), (Hardin, 2002), (Sen & Dutta, 2002)
Reinforcement learning	Web applications in autonomic software systems	(Salehie & Tahvildari, 2009),
Utility-based	Dynamic resource allocation	(Salehie & Tahvildari, 2009),
Architectural difference	Software architecture	(Salehie & Tahvildari, 2009), (Psaier & Dustdar, 2011)
Artificial Neural Networks + Signal Processing Techniques	Optical networks (IP/SDH IP/WDM)	(Marilly et al, 2002)

3.5.2 Algorithm classification per objective

In this section, we classify algorithms according to the objective achieved, shown in Table 6. It can be seen that different algorithms such as support vector machines and artificial neural networks can be used for helping the automation and optimizing the decision making process by using machine learning approaches. However, the focus of this thesis is to automate the diagnostics in programmable networks.

We identify two main research applications of control theory mechanisms: SLA compliance and self-management. **SLA compliance:** With this first concern, (Xiaoyuan et al, 2007) proposed autonomous network architecture for proactive policy business management in multimedia communication networks. This architecture allows for a better QoS / SLA management in service differentiation, by applying certain high-level objectives or policies into more specific commands (known as policy derivation) by means of a control theory mechanism. Another interesting work contributing to SLA enforcement is made by (Parekh et al, 2000), who proposed a control theory algorithm to control the maximum number of users allowed by a Lotus server with the aim of ensuring service level objectives (SLA) concerning database access delay. Also, Lu in (Lu et al, 2001) has applied control theory to differentiated content caching services to achieve performance distribution for resource management, but Lu in has also applied control theory in (Lu et al, 2002) to manage cache resources in a manner that adjusts the quality spacing between classes (QoS differentiation). In all these mechanisms, the control theory approach has been generalized to establish a methodology for designing controllers, by identifying the appropriate targets to be taken into account. Finally, (Akhtar, 2016) propose a Recursive InterNetwork Architecture, a clean-slate network architecture based on a control theory approach that balances the load among the different VNF instances deployed in a SDN infrastructure.

Self-management: Control-theory can also be used as self-adaptive mechanisms to achieve self-management capabilities. For instance, (Parekh et al, 2003) proposed a control theory mechanism to reduce the impact of administrative utilities such as file backups or garbage collection on databases. The authors successfully demonstrate the translation of high-level policies like maximum admissible degradation limit for database performance at an administrative level. This is carried out by a throttling system, which limits the execution of these administrative tasks in order to reduce their impact on database performances. This throttling mechanism regulates the resource consumption of utilities by using self-imposed sleep (SIS). All the aforementioned works on control theory algorithms refer to SISO systems (Single Input Single Output), where only one input is adapted to reduce its deviation to a reference given. Nevertheless, MIMO systems (Multiple Input Multiple Output) are increasing more used for optimizing different inputs. Indeed, there is an important research work on MIMO control-theory mechanisms for optimizing multiple database server parameters such as CPU and memory utilization. For instance, (Storm et al, 2006) proposes a MIMO control-theory algorithm as a tuning model for a DB2 Self-Tuning Manager system for memory allocation purposes. Alternatively, in (Diao & Hellerstein, 2005) they propose a

MIMO control-theory algorithm for load balancing in computing systems, which aim to reduce the service delay and increment the throughput.

Table 6. Self-healing algorithms classification per objective

Algorithm	Objective	Reference
Control theory	Minimizing the impact of administrative utilities on databases workloads	(Parekh et al, 2003)
	SLA enforcement	(Parekh et al, 2000)
	Memory allocation optimization	(Padala & Hou, 2009)
	Load balancing	(Akhtar, 2016)
	Guarantee of QoS differentiation	(Lu et al, 2002), (Lu et al, 2001)
	Proactive policy business management	(Xiaoyuan et al, 2007)
Bayesian Networks + Case-based reasoning	Reactive diagnosis	(Bennacer et al, 2012), (Bennacer et al, 2013), (Bennacer et al, 2015)
Bayesian Networks	Reactive diagnosis	(Hounkonnou, 2013) (Al-Jawad et al, 2015) (Yunkhao et al, 2010) (Mengshoel et al, 2010) (Bahl & Chandra, 2007), (Zhao, 2008), (Kandula & Mahajan, 2010)), (Cooper & Herskovits, 1992) (Tembo et al, 2015) (Steinder & Sethi, 2004), (Steinder & Sethi, 2002)
	Traffic engineering	(Al-Jawad et al, 2015)
	Proactive Diagnosis	(Cooper & Herskovits, 1992)
Hidden Markov models	Anomaly detection (security)	(Univerself, 2013), (Tcholchev et al, 2010)
Clustering	Reactive diagnosis	(Vaarandi et al, 2015)
Self-Organizing Maps	Congestion prediction	(Univerself, 2013)
Feature vectors	Proactive diagnosis	(Univerself, 2013)
Feature adaptation+Structural Corre-sponding Learning	Reactive diagnosis	(Univerself, 2013)
Fuzzy reasoning	QoS degradation identification	(Kimura et al, 2015)
Artificial Neural Networks	Proactive diagnosis	(Zhou et al, 2015)
Support Vector Machines	Proactive diagnosis	(Butler & Keselj, 2010)
Machine learning techniques	QoS degradation identification	(Univerself, 2013)
Game-Theory	Trust model building	(Buchegger & Boudec, 2002), (Hardin, 2002), (Sen & Dutta, 2002)
Reinforcement learning	Decision making	(Salehie & Tahvildari, 2009),
Utility-based	Decision making	(Salehie & Tahvildari, 2009),
Architectural difference	Abnormality detection	(Salehie & Tahvildari, 2009), (Psaier & Dustdar, 2011)
Artificial Neural Networks + Signal Pro-cessing Techniques	Reactive diagnosis	(Marilly et al, 2002)

3.5.3 Algorithm classification per self-healing functional task

In this section, we classify the algorithms according to the realized self-healing functional task, shown in Table 7. In each self-healing functional task, the algorithm has a different objective to achieve, e.g. detect malfunctions, diagnose the root cause or recover the system. We propose a non-exhaustive table where we classify some of the algorithms present in the literature that would fit inside a self-healing system.

As seen in Table 7, some of the algorithms can perform some self-healing functional tasks. For instance, the codebook technique, as an approach that relates faults with symptoms seen in a given application domain, it cannot be used for recovery tasks. Nevertheless, other algorithms such as control theory can be used to recover as they can adapt the system to restore the normal state but also can be used to detect certain deviations from a set of operational parameters by monitoring the deviation in a daily basis.

Bayesian Networks can be used to perform the recovery functional task as shown in the work (Al-Jawad et al, 2015) that calculates the probability of enough bandwidth on links by means of a Bayesian Networks based algorithm. The recovery solution would be based on selecting the best forwarding path based on the probability of enough bandwidth in each underlying link.

Table 7. Algorithm classification per self-healing task

Algorithm	Detection	Diagnosis	Recovery/Remediation
Bayesian Networks	x	x	x
Genetic algorithms		x	
Codebook technique		x	
Hidden Markov models[a]	x	x	
Case-based reasoning		x	x
Control theory	x		x
Fuzzy reasoning	x	x	x
Classification/ Patter recognition	x	x	
Decision trees		x	
Data clustering	x	x	x
Reinforcement learning			x
Utility-based algorithms			x
Policy-based algorithms			x
Artificial Neural Networks[b]	x	x	
Component interaction			x
Process coordination			x
Event-based techniques			x
Architectural difference	x	x	

[a.] Markov chains are framed in Hidden Markov Models (HMM).

[b.] Techniques like Self-Organizing Maps (SOM) are framed in Artificial Neural Networks (ANN)

Hybrid approaches can also be used to extend the functionality that a single algorithm has in itself. For instance, we could combine a Bayesian Networks algorithm with a control theory algorithm to extend it with diagnosis capabilities and recovery capabilities.

3.6 Self-Diagnosis algorithms

This section focuses on self-diagnosis task as a key operation inside a self-healing system. We focus on those aforementioned algorithms used in diagnosis and review the most important taking into account a network diagnosis as general context and programmable networks as application domain.

3.6.1 Data mining algorithms

This section reviews the data mining algorithms reviewed along this thesis. We classify them in supervised techniques or unsupervised techniques.

3.6.1.1 Supervised techniques

These types of algorithms learn how to classify future samples from an input training data set i.e. which is already classified. Each data is a tuple $training_sample = \{input, class\}$. Each input is composed of a set of N features or attributes $input$=$\{x_1, \dots, x_N\}$.These algorithms learn from this training data set the function that will allow to classify new samples $new_sample = \{input\}$.

Classification algorithms

The task of a classification algorithm is to learn the function that allows classifying new observed data into a set of classes. The output of a classification algorithm is the class of the incoming data set.

k-Nearest Neighbor classifier (k-NN) is a popular classification method that classifies each new data sample by a voting method among the k closest training samples. The closest training samples are computed through a distance metrics between the sample to classify and the N the training samples (there are N training samples). Several distance metrics can be defined; here we only show three examples: Euclidean (1), Manhattan (2), or Minkowski (3).

$$\sqrt{\sum_{i=1}^{N}(x_i - y_i)^2} \qquad (1)$$

$$\sum_{i=1}^{N}|x_i - y_i| \qquad (2)$$

$$(\sum_{i=1}^{N}(|x_i - y_i|)^q)^2 \qquad (3)$$

k is an input parameter (an integer). When k has a value k=1 it implies that the current sample is classified only by considering its nearest neighbor, while a high value of k is expected to minimize noise and be more accurate.

We show how this algorithm works with a simple example, shown in Figure 40, where data samples are composed of two features (feature1 and feature2). We have N=3 training samples to classify, and each data sample is classified according to the labels of their k closest neighbors. If we consider k=3, the sample is classified according to three training samples (2 samples of class 2 and one sample of class 1) so the classification algorithm associates that sample to class 2 (c2). Contrarily, if we consider k=1, sample is classified according to 1 training sample (one sample of class 1) so the algorithm associates that sample to class 1 (c1).

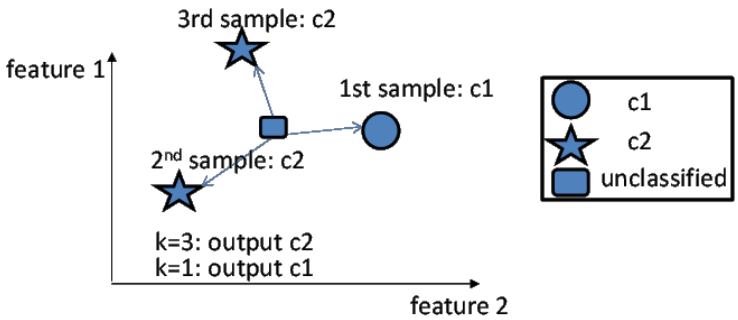


Figure 40. k-NN classification algorithm example

The main drawbacks of this algorithm are two: (1) the computation of the nearest neighbors for each sample due to the computational cost for a large number of samples to classify, and (2) how to tune the k-value for an optimal classification, because this tuning requires analyzing previously the data.

Regression algorithms

A regression algorithm predicts a real number from a set of training data. The output of regression algorithms is indeed a real number rather than a class, where this real number represents the dependent variable to be learned Y.

3.6.1.2 Unsupervised methods

Clustering algorithms

The main task of a clustering algorithm is to partition the incoming observations into different clusters or sets. Those observations are composed of d features (or dimensions). In the context of machine learning, these types of algorithms are referred to as unsupervised learning methods.

The most used algorithm for clustering is k-Means, an iterative algorithm that partitions the incoming observations in k sets $S = \{S_1, \dots, S_k\}$. The goal of k-means is to minimize the distance between the observation points with their respective centroid μ_i, mathematically formalized as:

$$\arg min_s \sum_{i=1}^{k} \sum_{x \in S_i} \|x - c_i\|^2$$

c_i is the cluster center of S_i. The number of clusters k is an input for the k-means algorithm, as well as the d-dimensional n observations. The output of the algorithm is the k clusters, composed of the centroids and the members belonging to each cluster.

- Step 1: the algorithm chooses randomly k of those observations as cluster centers: $c_1, \dots, c_k$
- Step 2: it calculates the distance between the observations and those means
- Step 3:it assigns each observation to its nearest mean, generating k clusters
- Step 4:it updates the centroid of each generated cluster c_i

$$c_i = \frac{1}{|k_i|} \sum_{x_{j \in k}}^{k} x_j$$

- Step 5: if the cluster centers change, repeat from step 2. Otherwise, the algorithm successfully finished.

The main limitation of this approach, but not the only one, is choosing the right value of the number of clusters k. To overcome this limitation, there are algorithms that do not require a priori knowledge of the number of clusters. Those algorithms rely on the relative difference on the observations and tend to work well in those cases where the difference between the clusters is higher than the difference between intra-cluster elements.

3.6.2 Control-theory

Control theory is an interdisciplinary branch in charge of designing adaptive mechanisms. Their aim is to minimize the existing error between the current and the reference input. In modern control theory the target is to design systems which maximize certain objective function over time, what matches very well the artificial intelligence perspective: designing systems that behave optimally. Control theory is then an adaptive mechanism to control the dynamic behavior of a given system, which interacts periodically in a sense-response-action loop, by composing a control loop.

As far as control classical theory is concerned, the negative feedback control loop is the most extended approach to control the dynamic behavior of a system, shown in Figure 41. The term negative comes from the subtraction between the reference input and the measured output coming from the transducer. This subtraction is more known as error signal. This error signal is fed to the controller and adapts its output in order to force a behavior change in the target and in turn to reduce the gap between the desired goal and the current state. This difference is increasingly reduced over time according to certain convergence properties a.k.a SASO properties: Stability, Accuracy, Settling time, and Overshoot, only guaranteed if the whole system (controller+target+transducer) is well designed. Normally, when the number of inputs and outputs to be controlled is one, this system is called SISO (Single Input Single Output), otherwise is called MIMO (Multiple Input Multiple Output).

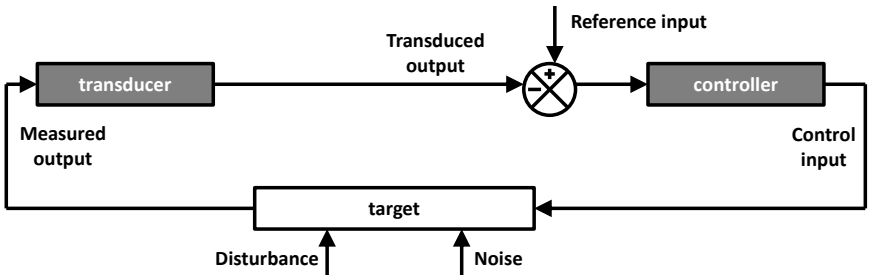


Figure 41. Control theory control-loop

3.6.3 Case-Based Reasoning

Case-based reasoning (CBR) systems are based on the adaptation and application of previously applied solutions to previous problems (gathered as cases in a database) to new problems.

There are several types of reasoning in CBR systems, namely, exemplar-based, instance-based, memory-based, case-based, and analogy-based. For instance, in exemplar-based reasoning, solving a problem is a classification task where an unclassified exemplar is classified in a given class, while case-based reasoning is richer in the sense that cases can be modified and adapted when applied to a different problem.

A CBR is based on four main phases, namely, retrieve, reuse, revise, and retain. The retrieve phase is in charge of receiving the new case and retrieving the most similar cases to the new case from the database (knowledge). The reuse phase is in charge of suggesting a new solution for the current case based on the retrieved cases that are similar to the current case. The revise phase is in charge of revising the suggested solution in order to evaluate possible dysfunctions, and the retain phase is in charge of storing the already revised case for being exploited in future problems.

The granularity of CBR is the case, as each new problem is translated to a case. The general structure of a case may vary as a function of the context application. In fact, the knowledge representation is an open research area as pointed out by (Aamodt & Plaza, 1994). They identify five main open research questions: knowledge representation, retrieval methods, and reuse methods, revise methods, and retain methods.

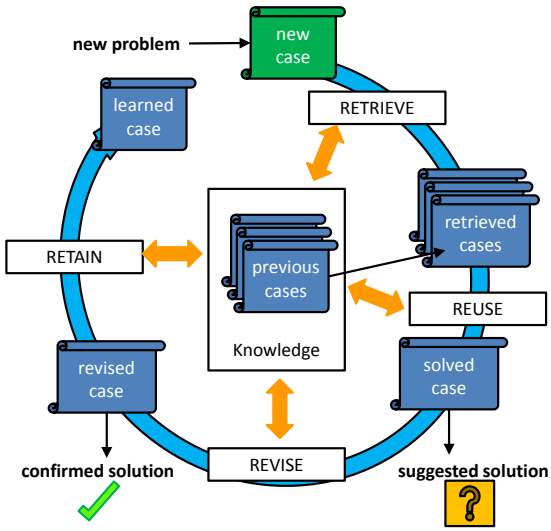


Figure 42. Case-Based Reasoning control loop

CBR has as advantages the capability to learn from experience by adapting old cases to the new problems. However, the CBR performance is not time efficient when it comes to real-time applications. Indeed, performance strongly depends on the structure chosen for the cases that is why is so important to know which characteristics and structure to store per case, and how to efficiently index to optimize the retrieve and reuse phases. Figure 43

shows two possible case structures further detailed by Aamodt et al., the dynamic memory model of Schank and Kolodner (a) and the category-exemplar model of Porter and Bareiss (b).

Memory models organize cases sharing similar properties under a generalized episode. This generalized episode contains norms, cases and indices. Indices, which are composed of indexes names and values, are the necessary features to discriminate among cases in the generalized episode, while norms are common features to the cases indexed in the generalized episode. Category and exemplar models represent concepts in an extensible manner, as we do in our memory. This structure is composed of categories, cases and index pointers, where each case has a given category, and indexes point to a case or category. There are three types of indexes: feature links point from problem descriptors (features) to cases or categories, case links point from categories to their related cases, and difference links point from cases to the neighbor cases only differing in a small number of features.

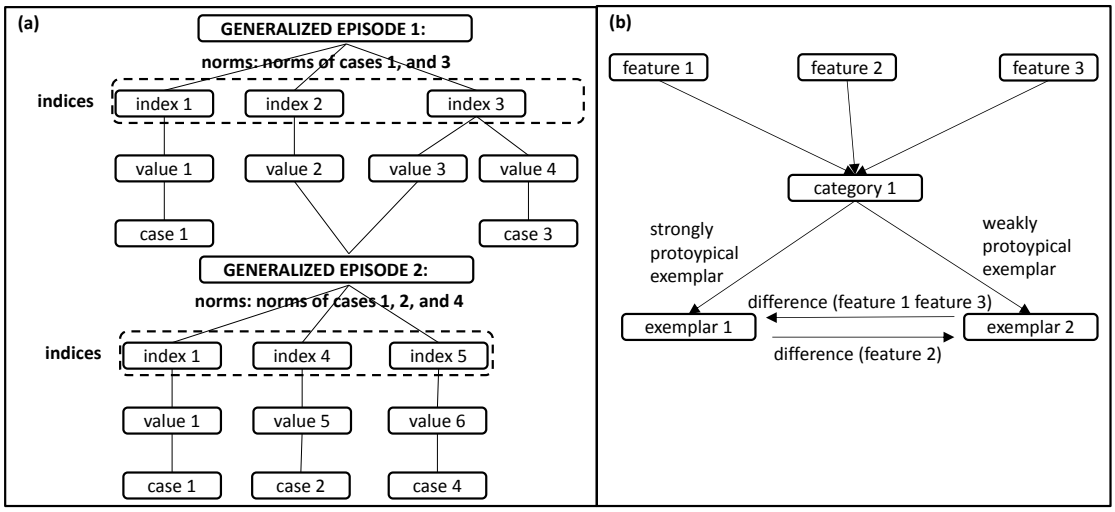


Figure 43. Two different types of case structures: (a) memory-model, (b) category and exemplar model

Additional drawbacks of CBR are dealing with new problems, the need to be specifically designed for the domain application, and its strong dependence from expert knowledge.

This algorithm is interesting to use it as support for other algorithms, where CBR can be used to establish and maintain a database of cases that may be used and adapted for new cases, but each case can be solved by using other different algorithms.

3.6.4 Discussion

We identified the versatility of control theory algorithms to cover quite a broad range of different application domains such as IT systems, cached services, multimedia communication networks, virtualized resources control, or software-defined networks and network function virtualization. However, we identified that control theory approaches, were mainly used for SLA compliance and self-management, but not for network diagnosis, which is the focus of the present thesis.

We also identified that data mining techniques, and more concretely supervised learning based techniques such as artificial neural networks, classification, pattern recognition, decision trees, or data clustering are not the optimal techniques for diagnosing networks. The lack of precedents in using this type of approaches for diagnosing telecommunication networks can be explained by two major factors:

- the own nature of these type of networks, characterized by its complexity, heterogenerity and size,
- the nature of data mining algorithms : their scalability concerns, their need of real-time processing, or their need of high amount of data to train them

-Supervised techniques may need huge amounts of training data to be able to predict accurately from the training data because its prediction capability is linked to the size of the training data. However, in telecommunication networks there may not be enough available training data (cases) for training the algorithms.

-In such dynamic networks where there are so many changes occurring so fast there may not be time to train the algorithm to predict the output with enough accuracy. For instance, ANN (Artificial Neural Networks) approaches for network diagnosis are not common in the existing literature to the best of our knowledge. ANN requires intensive training before being able to relate inputs and outputs in an accurate manner.

BN are very versatile as identified by Irish in (Rish,_), (Steinder & Sethi, 2002) , and (Bennacer et al, 2012), BN can be used to predict, classify, make decisions and diagnose. Indeed, Bayesian Networks are a suitable approach for fulfilling the three self-healing functional tasks i.e. detection, diagnosis, and recovery.

BN are widely and extensively used in many different application contexts such as IP Multimedia Subsystem, Software-Defined Networking, Wireless Sensor Networks, Electrical Power Systems, Enterprise Networks/ IT systems, GPON-FTTH Optical Networks, and Bridged networks. Indeed, BN are used for network diagnosis purposes, which is the goal of this thesis, so in the next section we will elaborate on those types of approaches in order to apply their principles to model and diagnose programmable networks.

3.7 Bayesian Networks

3.7.1 Definitions

A Bayesian Network (BN) is a probabilistic graphical model that models a set of n random variables and their dependencies in a given domain. The BN is a pair (G, p) composed of a Directed Acyclic Graph (DAG) $G = (V, E)$ and p , which can be either a probability distribution or a family of probability distributions indexed by a parameter set Θ defined over the n discrete random variables $\{X_1 \dots X_N\}$. The design of the Bayesian Network concerns both parameters defined in this pair (G, p) and those satisfy the following criteria :

- For each θ belonging to the parameter set Θ, there exist $p(.|\theta): \chi \rightarrow [0,1]$ where $\sum_{\bar{x}} p(\bar{x}|\theta) = 1$
- For each vertex $X_v \in V$ with a parent set composed of n parents, $\pi_v = Parents\ (X_v) = (X_{b_{1(v)}}, \dots, X_{b_{n(v)}})$, there exists a potential $p_{X_v|\pi_v}$ providing the conditional probability distribution of the random variable X_v given its n parent variables $X_{b_{1(v)}}, \dots, X_{b_{n(v)}}$.
- For each parentless vertex $X_v \in V$, there exists a potential denoted by p_{X_v}providing the probability distribution of the discrete random variable X_v. Indeed, this is a particular case where the parent set corresponds to the empty set $\pi_v = \phi$ and therefore $p_{X_v|\pi_v} = p_{X_v}$
- The joint probability function p is then factorized by using the aforementioned potentials $p_{X_v|\pi_v}$ as it follows

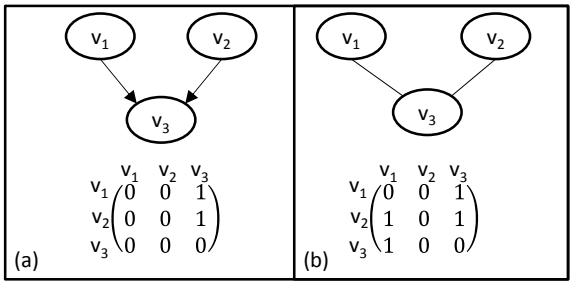
$$p_{X_1,\dots,X_N} = \prod_{v=1}^{N} p_{X_v|\pi_v}$$

The aforementioned potentials $p_{X_v|\pi_v}$ are usually given in the shape of CPT (conditional Probabilty Table). This table is composed of rows defining each possible set of values for the parent set $\pi_v = (X_{b_{1(v)}}, \dots, X_{b_{n(v)}})$ describing the different values of random variable X_v.

The DAG $G = (V, E)$ is composed of a set of N vertices and K edges. On one hand, vertices represent a set of random variables $\{X_1 \dots X_N\}$. Those variables can be continuous or discrete. Discrete random variables have a finite set of mutually exclusive states s. On the other hand, edges $E = \{e_1 \dots e_K\}, K \in NxN$ represent the dependencies among those variables.

Two types of dependency graphs exist. On one hand, undirected dependency graphs are composed of bidirectional edges so that propagation is possible in both senses. On the other hand, directed dependency graphs are

composed of unidirectional edges so that propagation is possible in only one sense. We show a directed acyclic graph and an undirected acyclic graph in .



$$\begin{array}{c|ccc} & v_1 & v_2 & v_3 \\ \hline v_1 & 0 & 0 & 1 \\ v_2 & 0 & 0 & 1 \\ v_3 & 0 & 0 & 0 \end{array} \qquad \begin{array}{c|ccc} & v_1 & v_2 & v_3 \\ \hline v_1 & 0 & 0 & 1 \\ v_2 & 1 & 0 & 1 \\ v_3 & 1 & 0 & 0 \end{array}$$

Figure 44. Examples of (a) directed acyclic graph, (b) undirected acyclic graph

A dependency graph, whether is directed or undirected, is depicted by an adjacency matrix A_{NxN} consisting of a squared matrix of size NxN, where N is the number of vertices of the dependency graph $G = (V, E)$. The values of the adjacency matrix represent the existence of a dependence between two vertices, where '1' means there is a dependence and '0' there is not any dependence between two any vertices. The mathematical expression for each value of the adjacency matrix A_{NxN} is given here:

$$a_{ij} = \begin{cases} 1, if \ \exists \ E_G(v_i, v_j) \\ 0, if \ \nexists \ E_G(v_i, v_j) \end{cases}$$

In the case where the dependency graph is weighted, one scalar value w_{ij} depicts the strength of that edge between two any vertices, given here.

$$a_{ij} = \{w_{ij}, \forall \ E_G(v_i, v_j)$$

Three important characteristics of acyclic graphs can be noticed here:

1) Both directed or undirected graphs have a null diagonal adjacency matrix i.e. $\mathrm{a_{ii}} = 0 \ \forall \ \mathrm{i} \ \in [1, \mathrm{N}]$, as there is no any edge or dependency between one vertex and itself.

2) The adjacency matrix of undirected graphs is symmetric i.e. $\mathrm{a_{ij}} = \mathrm{a_{ji}} \ \forall \ \mathrm{ij} \ \in [1, \mathrm{N}]$, but not in case of directed graphs. shows the adjacency matrices of a directed acyclic graph and an undirected acyclic graph.

3) A directed acyclic dependency graph must be topologically sorted. Indeed, only a directed acyclic graph can be topologically sorted. The topological sort is defined by Skiena in (Skiena, 1990) as a permutation m of the vertices of an acyclic dependency graph such that an edge (i, j) implies that i appears always before j in m . shows a non-topologically ordered dependency graph and a topologically ordered one.

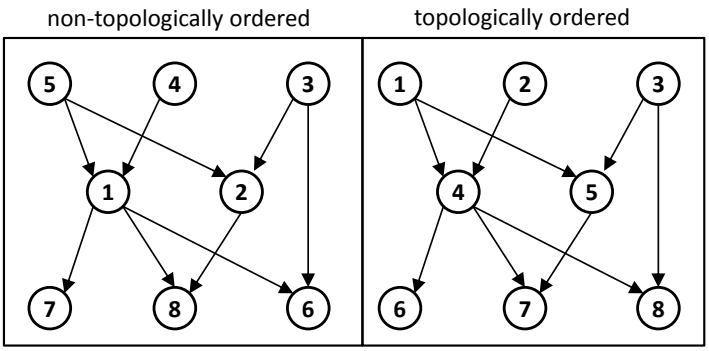


Figure 45. Topological order in dependency graphs

3.7.2 Reasoning in Bayesian Networks

There are three types of queries for a BN, enumerated hereafter:

Probability updating: The observations o are given on some variables and the posterior probability potentials for the rest of variables are calculated.

Most probable configuration: The observations o are given on a set S, and the most probable values (states) on the rest of the variables are computed.

Maximum aposterior hypothesis: The observations o are given on some variables on a set S, and a hypothesis h over a subset of variables, which maximizes the probability $p(h|e)$, is found.

We can classify the vertices of the BN according to their position on the graph such as leaf vertices as those which do not have children, and root vertices as those which do not have parents. We can also classify the vertices according to the information they contain such as observed vertices as those which state is known with probability one, and non-observable vertices V_H as those which state is not known. We call observations to the evidences injected to the BN, where O is the array of observations of value o. Querying the BN means computing the a posteriori probability by injecting the evidences on the observable vertices in that BN.

The posterior probability or belief of a given vertex $X_v \in V$ is defined as $Bel(X_v) = p(X_v = s|o)$ and it is computed by marginalization. This value indicates the overall belief in the statement $X_v = s$ given all the observations o introduced in the BN.This set of observations or evidences can be separated into two complementary subsets: $o_X^-$for those observations coming from the edges towards the parents of X_v and $o_X^+$for those observations coming from the edges towards of the children of X_v.

The posterior probability calculation is based on the Bayes rule, which is as follows $p(H|o) = \frac{p(o|H)p(H)}{p(o)}$, where $p(o)$ is a normalization constant, $p(H)$ is the prior probability, $p(o|H)$ is the likelihood of the evidence, and $p(H|o)$ the posterior probability.

This rule is applied taking into account these two subsets of observations to determine the posterior probability for the random variable X_v as it follows: $Bel(X_v = s) = p(s|o_X^-, o_X^+) = \frac{p(o_X^-|s,o_X^+)p(s|o_X^+)}{p(o_X^-,o_X^+)} = \frac{p(o_X^-|s)p(s|o_X^+)}{p(o_X^-,o_X^+)} = \frac{\mu_v(s)\pi_v(s)}{p(o_X^-,o_X^+)}$

The condicional probabilities $\mu_v(s) = p(o_X^-|s)$ and $\pi_v(s) = p(s|o_X^+)$ are propagated among the neighbours to update the aposterior probability distributions on each variable. In this propagation process, each vertex updates its posterior probabilities by applying the Bayes rule based on messages $\mu_v(s)$ from its children and $\pi_v(s)$ from its parents. This propagation process is shown for a chain Figure 46 (a) and a tree Figure 46 (b).

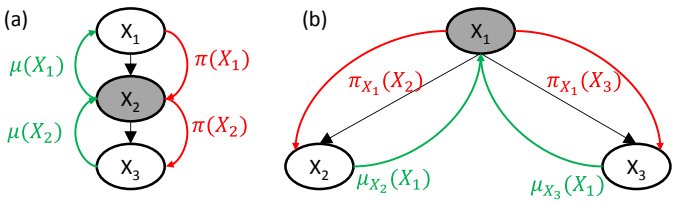


Figure 46. Belief propagation in chain and tree

When BN are used for diagnosis, the type of reasoning is called evidential reasoning or explanation. In this type of reasoning, also known as root cause analysis (RCA), we inject a set of observations for which we need an explanation (the root cause responsible). This explanation is obtained by querying the rest of variables in the BN given those observations. This explanation has an associated value of uncertainty.

One typical example of this type of inference is the QMR-DT model, a decision-theoretic reformulation of the Quick Medical Reference (QMR) model, shown in . Its model is a two-layered dependency graph, composed of N diseases (depicted by N root vertices) and M symptoms (depicted by M leaf vertices). In general, any of the N

diseases can lead to any of the M symptoms (a.k.a findings). The main goal of the diagnosis is to infer which disease is the most probable taking as input the set of M symptoms.

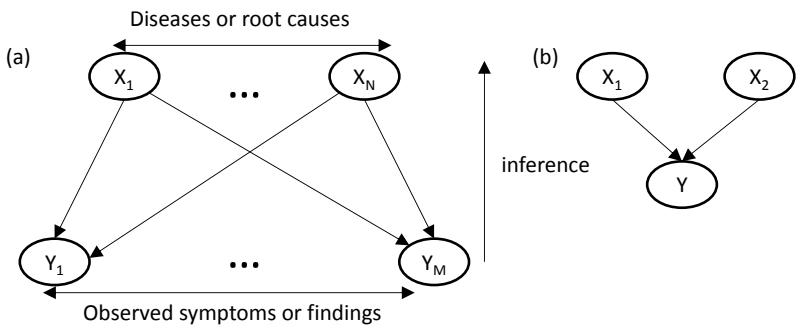


Figure 47. QMR-DT dependency graph: (a) N diseases, M symptoms, (b) N=2 diseases, M=1 symptoms

To reason over this BN, we set the evidences on the state of the symptoms vertices Y_i and the BN algorithm propagates these injected observations through the dependency graph to calculate the a posteriori probability of each root cause vertex X_i. The joint probability describing the diseases and symptoms can be given by the equation:

$$P(X,Y) = P(X|Y)P(Y) = \prod_{i=1}^{M} P(Y_i|X) \prod_{j=1}^{N} P(X_j)$$

where Y is the symptoms vector $Y = \{ Y_1, \dots , Y_M\}$ and X is the diseases vector $X = \{ X_1, \dots , X_M\}$.

3.7.3 Properties of Bayesian Networks

3.7.3.1 Conditional independence property

As said in previous section, any DAG can be decomposed in three types of sub-graphs, where each sub-graph represents a basic way of connecting two variables to a third variable: chain (Figure 48(a)), fork (Figure 48 (b)), and collider (Figure 48 (c)). The conditional independence in these three cases is different.

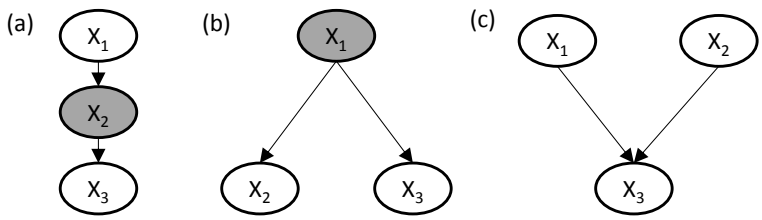


Figure 48. Three ways of connecting three variables in a DAG: (a) chain, (b) fork, (c) collider

For instance, in (a), knowing the state of variable X_2 makes X_1 and X_3 conditionally independent. In (b), knowing the state of X_1 makes X_2 and X_3 conditionally independent. However, in (c) the situation is slightly different, as not knowing the state of X_3 makes X_1 and X_2 independent.

3.7.3.2 Factorization property

In large dependency graphs, where the number of modelled variables is huge, the joint probability distribution is too big to be handled due to the large size of the CPTs defined per variable in the graph.

Nevertheless, thanks to the factorization property of BN, the number of parameters required to describe the joint probability distribution $p_{X_1,\dots,X_N}$ can be drastically reduced by leveraging the conditional relationships among variables. For instance, for a given collection of variables $\{X_1 \dots X_N\}$, the probability distribution can be factorized as $p_{X_1,\dots,X_N} = p_{X_1}\, p_{X_2|X_1} p_{X_3|X_2} \dots p_{X_N|X_{N-1}}$, expressing the joint probability distribution as a product.

This factorization has the benefit of a reduction in the number of parameters to describe the joint probability distribution, which depends on the size of the CPTs. For instance, for a k-state random variable with n parents, k^n parameters are needed to describe its corresponding CPT.

The CPT of parentless vertices is composed of one unique parameter, their a priori distribution p_{X_v}. For binary vertices with n parents, 2^n parameters are needed to describe their CPT.

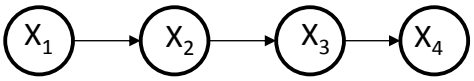


Figure 49. Example of probabilistic dependency graph

As example, we consider the four-vertex chain of , composed of binary variables (k=2). In this chain, X_1 and X_3 are independent on the condition that X_2 is instantiated. The same happens with X_2 and X_4 and and X_3.

The joint distribution $p_{X_1,\ldots,X_N}$ would need 2^4=16 parameters, whilst, thanks to the factorization property of BNs that allows expressing the joint probability distribution as $p_{X_1,X_2,X_3,X_4} = p_{X_1} p_{X_2|X_1} p_{X_3|X_2} p_{X_4|X_3}$, the number of parameters required to describe the joint probability distribution p_{X_1,X_2,X_3,X_4} is drastically reduced. For instance, X_1 is parentless, so it needs 1 parameter (p_{X_1}), while X_2 X_3 and X_4 need 2 parameters, $p_{X_2|X_1}$, $p_{X_3|X_2}$, and $p_{X_4|X_3}$. The joint probability is then described with 7 parameters instead of 16 parameters initially needed.

3.7.3.3 Explaining away property

The explaining away property is a phenomenon that appears when the number of observations introduced in the BN increases.

This property is paramount when diagnosing, because the state of a given variable may be due to multiple possible causes, and we are interested in discarding as many root causes as possible and reducing the diagnosed variables to the minimal set. Indeed, the fact of adding new observations to the BN makes clearer which variables are responsible for that symptom found. The explaining away concept is tightly linked to the uncertainty of the aposterior probability distribution.

We can measure its uncertainty to quantify how the inference engine can discriminate among the different root causes. The uncertainty of a given distribution is quantified via the Shannon entropy (measured in bits). In general, the entropy $H(X)$ of a discrete random variable X with alphabet α and probability mass function $\Pr(X = x)$ is at follows:

$$H(X) = -\sum_{X \in \alpha} \Pr(X = x)\, log \Pr(X = x)$$

Note that $H(X) \geq 0$. In a perfect and doubtless system, $H(X) = 0$.

Entropy depends on the number and quality of the observations added to the BN. The lower entropy, the lower uncertainty and the better the BN engine discriminates among different root causes. Nevertheless, if additional observations *Y* are added in the graph, entropy $H(X)$ is reduced by $G(Y) = H(X) - H(X\,|Y)$, where $H(X\,|Y) = -\sum_{x,y} \Pr(X = x, Y = y)\, log \Pr(X = x, Y = y)$. This phenomenon is known as information gain.

We exemplify the explaining away effect with the following BN, shown in , composed of two diseases (X_1 and X_2) and one observed symptom (Y). We want to determine which of these diseases is the root cause given an observation on Y.

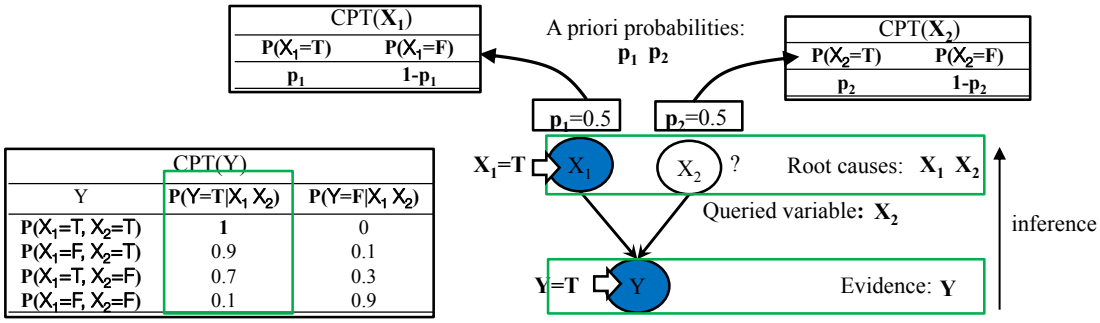


Figure 50. Diagnosis with Bayesian Networks

In this first example, we have as observations that X_1=True and Y=True, and we query the variable X_2 to see how probable is that variable to be the root cause given these observations seen on X_1 and Y (in blue). The a priori probabilities in X_1 and X_2, specified in their corresponding CPTs, are $p_1=p_2=p=0.5$. The equation that allows calculating the aposteriori probability for the disease X_2 is

$$P(X_2 = T|X_1 = T, Y = T) = \frac{P(Y=T|X_1=T,X_2=T)\ P(X_2=T)}{P(Y=T|X_1=T,X_2=T)\ P(X_2=T)+P(Y=T|X_1=T,X_2=F)\ P(X_2=F)} = 0{,}58.$$

We can see in this example that X_1 is more probable than X_2 and this is explained by the CPT of the variable Y, fixed in the green column because $Y = T$, which value is $P(Y|X_1 = T, X_2 = T)=0.7$.

In this second example, we inject on the BN the observations X_1=False and Y=True, and we query the variable X_2 to see how probable is the variable X_2 to be the root cause given these observations on X_1 and Y (in blue). In this case, given that the other posible variable to be the cause X_1 has zero probability (X_1=False), the other posible cause X_2, seems to be the most probable explanation as the explaining away property would predict.

$$P(X_2 = T|X_1 = F, Y = T) = \frac{P(Y=T|X_1=F,X_2=T)\ P(X_2=T)}{P(Y=T|X_1=F,X_2=T)\ P(X_2=T)+P(Y=T|X_1=F,X_2=F)\ P(X_2=F)} = 0{,}9$$

We can see in this second example that the fact that the a posteriori distribution reveals that X_2 is more probable than X_1 and this is explained by the CPT of the variable Y, fixed in the green column due to the fat that $Y = T$, which final value is $P(Y|X_1 = F, X_2 = T)=0.9$.

This property is very important in the root cause analysis procedure to discard some variables with respect to other variables depending on the incremental addition of evidences.

3.7.4 Challenges of Bayesian Networks in diagnosis

When Bayesian Networks (BN) are used for diagnostics or fault-localization, the dependency graph is then used as diagnosis model, which indicates how faults propagate. The Bayesian Network inference engine exploits the probabilistic dependency graph to propagate the probabilities over that graph in a process called the Root Cause Analysis (RCA). However, many other techniques can be used to this perform the RCA such as Occam's Razor or Markov chains.

In this context, two important issues must be solved. Firstly, the generation of the probabilistic dependency graph and secondly, the scalability of the inference process a.k.a the Root Cause Analysis, where the graph is fed with the observations from the network.

Scalability limitations: One first limitation of Bayesian Networks is the scalability due to the need to make the inference over large dependency graphs. This comes as a result of the growth in the number of parents of a given vertex, which leads to an explosion in the number of states of that vertex, impacting on the size of its CPT. Indeed, the size of the CPT grows exponentially according to the value $states^n$. For instance, a discrete binary random variable with one parent has a CPT with size $2^1 = 2$, while the same discrete binary random variable with 5 variables will have a CPT size of $2^5 = 32$. Nevertheless, the scalability of BN-based solutions, as identified by Bennacer their works (Bennacer et al, 2012) , (Bennacer et al, 2013), and (Bennacer et al, 2015), can be allevi-

ated by combining the BN with a CBR based system that reduces the inference process to a great extent. Another approach is the proposed by Hounkonnou et al in (Hounkonnou, 2013), where the dependency graph is extended in such an intelligent way to reduce the inference to the necessary subset of the network to be diagnosed and avoid generate a large dependency graph.

Bayesian Network generation limitations: The second limitation of Bayesian Networks as model-based approach is that it depends on a model that must be generated. However, much more attention is paid to the inference and RCA algorithms in use by assuming that the model is already there, to the detriment of the generation of the probabilistic dependency graph, which in most cases, is manually built from operational team's knowledge. This manual generation is valid for static networks, where networks resources and their interaction tend to be statically fixed, and there is no need to generate and update a model of network and services every other few minutes or seconds. When it comes to diagnosing dynamic network topologies and services as expected with SDN and NFV, the static generation of the dependency graph is not appropriate at all, mainly due to the fact that the model has to be continuously regenerated to encompass those dynamic changes. In our context, SDN and NFV, network topology becomes highly dynamic due to the rapid and flexible programmability of the underlying connections among switches by the SDN controller (up to ms). Networking services defined over SDN and NFV will rely on a dynamic placement and migration of the virtual network functions as well as an elastic usage of the compute, storage and networking resources. Therefore, in combined SDN and NFV environments, the high network dynamicity provided by SDN–topological changes and rapid forwarding changes through flows–becomes even higher with NFV, since the VNF can be scaled, instantiated, deleted, and migrated, as a result, the subsequent changes in the virtual links. In this context, service dependencies from the underlying resources are in a continuous change and need to be managed dynamically.

In this thesis we make a contribution on this second limitation by proposing a self-modeling approach to generate the probabilistic dependency graph in an automatic and online manner. Therefore, in this section, we explore the approaches used to generate the probabilistic dependency graph and we explore how those graphs are applied and exploited in different contexts. In addition, we will explore further different network contexts and compare them to programmable networks in order to identify which elements are worth being applied to diagnose programmable networks.

3.7.5 Related work on the generation of the Bayesian Network

BN are very versatile because those can be used for classification, prediction and diagnosis, the focus of this thesis. Firstly, BNs can be used for clustering and classification problems. In a clustering problem, the input data is partitioned into clusters. This problem, seen from a BN perspective, is to determine which class or cluster maximizes the probability that a given input symptom is consequence of a given root-cause, mathematically formalized as $\max_{class} P(class|data)$. As example of this application, Hamerly and Elkan in (Hamerly & Elkan, 2001) proposed a clustering-based failure predictor for hard disk drive failures. This failure predictor is based on two different Bayesian Methods: first, a naïve Bayes sub model and, secondly, a naïve Bayes classifier trained by an Expectation-Maximization supervised learning method. Secondly, BNs can be used to predict failures by estimating the probability distribution of the next time to failure by using the current probability distribution given by the Bayesian Network algorithm. This is formally defined as $P(symptoms|root-cause)$. As mentioned before, Salfner et. al, showed several examples of BN algorithms that tracks previous failure occurrences and estimates the probability distribution of the next time failures. And thirdly, BNs can be used for diagnosis. Indeed, as it was seen in the previous section this is the most popular application of BNs. Diagnosis consists of calculating the a posteriori probability of those root vertices given the state of those known vertices which state is propagated through the BN. The problem is formalized as finding the a posteriori probability distribution given a set of symptoms $P(root\ cause|symptoms)$.

Bayesian Networks is a suitable approach for network diagnosis, mainly due to the following aspects:

- BN combines the probability theory with the graph theory, what allows exploiting the synergy between both fields in one single algorithm.

- The knowledge representation is compact, reliable, and models randomness, everything together in the shape of probabilistic dependency graph. In contrast to codebook and fault-symptom models, which are bi-layered, a probabilistic dependency graph models intermediary layers and other more complex relationships that can be seen in the application domain of networks.

- BN has training capabilities, not to the extreme of ANN or similar machine learning approaches, but rather on the sense to tune the a priori probabilities and build the dependency graph. However, networks is not the proper environment to train an algorithm because the data may not be available neither the time to train the algorithm.

3.7.5.1 Generation of the Bayesian Network through approaches

In this section, we survey those different approaches that generate the Bayesian Network structure. The generation of the Bayesian Network infrastructure means generating the dependency graph and the conditional probability distributions associated to the dependency graph. In general, the dependency graph is manually generated from an operational team's knowledge, but the core of this thesis is its automatic generation to cope with its high dynamicity that requires generating it an update at run-time.

Indeed, this is a complex mathematical problem, which is found to be a NP-hard. Indeed, the number of possible dependency graphs for a given problem grows in an exponentially with the number of modelled variables n. Indeed, this was already quantified by Robinson in (Robinson, 1977), which defined a recursively-based formula to compute the number of possible dependency graphs for a given number of modelled variables n, shown hereafter:

$$f(n) = \begin{cases} 1, & if\ n = 0 \\ \sum_{i=1}^{n} (-1)^{i+1} C_n^k 2^{i(n-1)} f(n-1), & if\ n > 0 \end{cases}$$

For instance, for a 3-variable problem would have 25 possible dependency graphs and a 6-variable problem would have the astronomic figure of 3,781,503 possible dependency graphs. Figure 51 shows 13 out of the 25 possible combinations of dependency graphs for a 3-variable model.

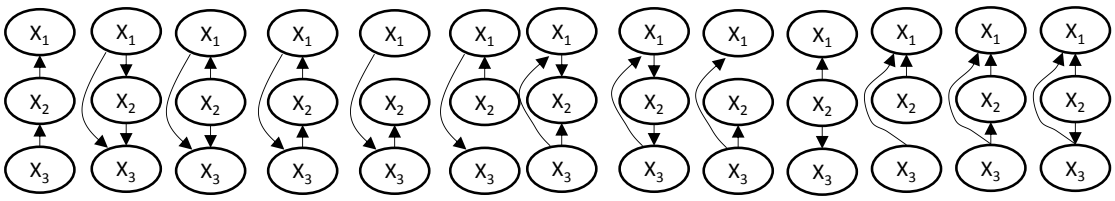


Figure 51. Example of possible dependency graphs for a 3-variable problem

The probabilistic dependency graph can be generated by using different approaches. We identify two types of approaches to learn the network structure: search and score methods and conditional independence testing methods.

The search-and-score method iterates over all the candidate network structures and scores each network structure according to a given criterion or metrics that evaluates the goodness of that network structure. This method receives as input a cases database, defined as a matrix $D_{MxN} \equiv D$ composed of m rows depicting the cases and n columns depicting the corresponding values of each modelled variable X_j for each case.

Each case $C_j\ \forall\ i\ \in [1, m]$ contains the values of the modelled variables included in the Bayesian Network $v_{i,j}$. Figure 52 shows a cases database example for m=9 cases and n=3 variables modelled, as well as the generated dependency graph given as output by a search-and-score method. In this concrete case, the modelled variables are binary, i.e. $v_{i,j}\ \in [0,1]$.

Variables

Cases	X_1	X_2	X_3
C_1	1	0	0
C_2	1	1	1
C_3	0	0	1
C_4	1	1	1
C_5	0	0	0
C_6	0	1	1
C_7	1	1	1
C_8	0	0	0
C_9	1	1	1

Figure 52. Cases database example for m=9, n=3 and generated dependency graph

A search and score method receives as input this database of cases and it provides as output the calculated dependency graph and the conditional probabilities relating those variables. K2 is one of the most popular search and score algorithm for learning the Bayesian Network structure, which is based on a greedy search algorithm proposed by Cooper and Herskovits in (Cooper & Herskovits, 1992). This algorithm considers the following assumptions:

- All the candidate network structures are equally probable
- Modelled variables are ordered
- There is a maximum number of parent vertices per vertex (phi)

K2 algorithm finds the best network structure B_S. It starts with an empty set of parents for each given vertex and it is incrementally adding parents until reaching that maximum number phi. The number of parents for each vertex is increased until the CH score of the resulting network structure is maximized, and the algorithm stops when adding more parents does not improve the CH score. The CH score was defined by Cooper and Herskovits in (Cooper & Herskovits, 1992). This metrics seeks to maximize the probability $P(B_S, D)$, defined as it follows:

$$P(B_S, D)=P(B_S)\prod_{i=1}^{n} g(i,\pi_i) = P(B_S)\prod_{j=1}^{q_i}\frac{(r_i-1)!}{(N_{ij}+r_i-1)!}\prod_{k=1}^{r_i} N_{ijk}!$$

However, as disadvantage, the K2 algorithm does not have to choose that network structure with the highest probability and it is prone to local optima.

The conditional independence testing method is based on by performing statistical tests among modelled variables to determine if those variables have a significant degree of correlation. This means that the behavior of modelled variable X_i can help us determine the behavior of X_j or vice versa. A statistical test is based on an analysis plan, which describes how to use the input data to accept or reject the null hypothesis H_0. If the variables are dependent, the null hypothesis H_0 is true, otherwise, both variables are assumed independent and the alternative hypothesis becomes true. An analysis plan has the following shape:

$$\begin{cases} H_0: X_i, X_j \ dependent \\ H_a: X_i, X_j \ independent \end{cases}$$

This plan depends on two key parameters: the significance level and the test method. The most popular test method is the chi-squared test, usually calculated from a sum of squared errors. Its value can be calculated as $\chi^2 = \sum \frac{(f_O - f_E)^2}{f_E}$, where f_E are the frequency of the expected values, and f_O are the frequency observed values. This algorithm is based on the following steps: (1) Conjecture and Analysis Plan, (2) χ^2value calculation, (3) significance level calculation $(1-p)$ and degree of freedom v, (4) critical value calculation, and (5) comparison between χ^2 and critical value and interpretation of result.

As example, we study the degree of correlation a symptom in the network such as the packet loss and a possible cause, the congestion in a given link. We first conjecture that the observed packet loss is related to congestion (null hypothesis H_0). The following table gathers information of the observed congestion and packet loss.

Observed Data	***Packet Loss***	
Congestion	***Yes***	***No***
Yes	50	25
No	40	45

The observed data is added up per columns and rows to compute the frequencies:

	Software		
Hardware	**Failure**	**No Failure**	**Frequency per line (L)**
Failure	50	25	70+25=**95**
No Failure	40	45	40+45=**85**
Frequency per column (C)	50+40=**90**	25+45=**70**	90+90=**160**

Expected frequencies are calculated based on the previous calculation:

	Software		
Hardware	**Failure**	**No Failure**	**L**
Failure	90*75/160=42.1875	70*75/160=32.8125	70+25=**95**
No Failure	90*85/160=47.8125	70*85/160=37.1875	40+45=**85**
C	50+40=**90**	25+45=**70**	90+90=**160**

The chi-squared calculation is made by using the aforementioned equation $\chi^2 = \sum \frac{(f_O - f_E)^2}{f_E}$, giving as a result:

$$\chi^2 = \frac{(50-42.19)^2}{42.19} + \frac{(25-32.81)^2}{32.81} + \frac{(40-47.81)^2}{47.81} + \frac{(45-37.19)^2}{37.19} = 6.22$$

The degree of freedom is calculated as $v = (\#rows - 1)(\#columns - 1) = 1$, and the significance level is chosen 0.01 this means that the probability that the null hypothesis $H_0$is true becomes $p = 1 - 0.01 = 0.99$. The Critical value of chi square distribution is found in its given table described by the table of pairs (p, v). The critical value found in the table for the pair $(p = 0.99, v = 1)$ is 6.635. Given the fact that $\chi^2 = 6.22 \leq 6.635$, the null hypothesis is accepted and under those conditions on significance level it can be said that packet loss is dependent on the congestion of that given link.

3.7.5.2 Generation of the Bayesian Network from datasets

The BN can be generated from different types of data sets. Here, we survey those types of data sets from which the probabilistic dependency graph has been automatically obtained in the literature.

Relational models like databases can be converted in probabilistic dependency graphs to include the notion of uncertainty and probabilities on it. For instance, S. Singh and T. Graepel in (Singh & Graepel, 2013) proposed a methodology to generate the probabilistic dependency graph containing the information extracted from a rela-

tional model. This approach allows the authors to generate the probabilistic dependency graph from many different types of databases without any manual intervention.

A database or relational model is a set of tables composed of attributes with links to other tables.

In their approach, the authors cover two types of tables: linked tables (i.e. tables which attributes are linked to other tables) and non-linked tables. Figure 53 shows one example of generated dependency graph built from a database composed of three linked tables (User, Movie, and Rating), where the rating table is pointing to User and Movie tables. In the corresponding dependency graph, the interdependencies among tables are translated to arrows in the dependency graph that will connect the different sub-dependency graphs User and Movie, which are in turn composed of sub-dependency graph for each of their components detected in that database.

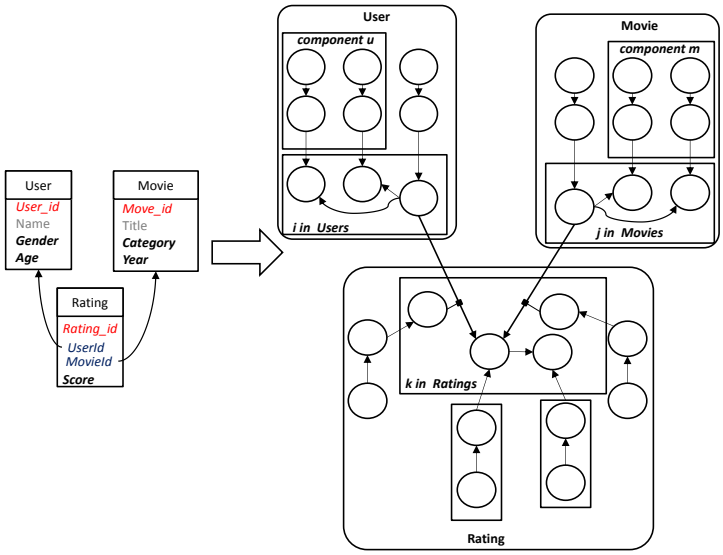


Figure 53. Probabilistic dependency graph extraction from a database

The probabilistic dependency graph can also be obtained from ontologies. In general, ontologies are composed of concepts and their interrelationships with other concepts, but no notion of uncertainty is taken into account in the ontologies, because concepts are related (or not) to other concepts in a binary relationship but there is no notion of the probability that there is a relationship among different concepts.

Messaoud et. al. in (Messaoud et al, 2009b) propose the SemCado (Semantic Causal Discovery) framework, which learns CBN (Causal Bayesian Networks) using prior knowledge extracted from ontologies. Indeed, this work is based on their previous framework named MyCado in (Messaoud et al, 2009a), which was based on a contraint based structure-learning algorithm. The authors exploit the similarities between ontologies and CBNs by following a three-step procedure:

- A concept in the ontology is translated to a random variable and thus a vertex in the CBN
- A semantic causal relation between two concepts in the ontology is translated into a dependency between two vertices in the CBN
- Concept-attribute instances in the ontology are translated to observational data in the CBN

Alternatively, Fenz et. al. in (Fenz, 2011) propose an enhanced Bayesian threat probability determination calculation scheme that enriches the BN with information extracted from the security ontology, shown in Figure 54, in the context of information security risk management for business organizations.

An asset or a given organization requires from a given level of security to overcome vulnerabilities. In the event of a threat, that threat may exploit a vulnerability of the asset (let's imagine the vulnerability of a SDN controller against DoS attacks) and make it vulnerable. The vulnerability has a given severity level. Nevertheless, a control

strategy, able to mitigate that threat should be put in place. However, in this state, the security ontology does not take into account uncertainty, that is why the authors want to enrich the security ontology with the probabilities associated to each of those concepts and interrelationships extracted from the ontology.

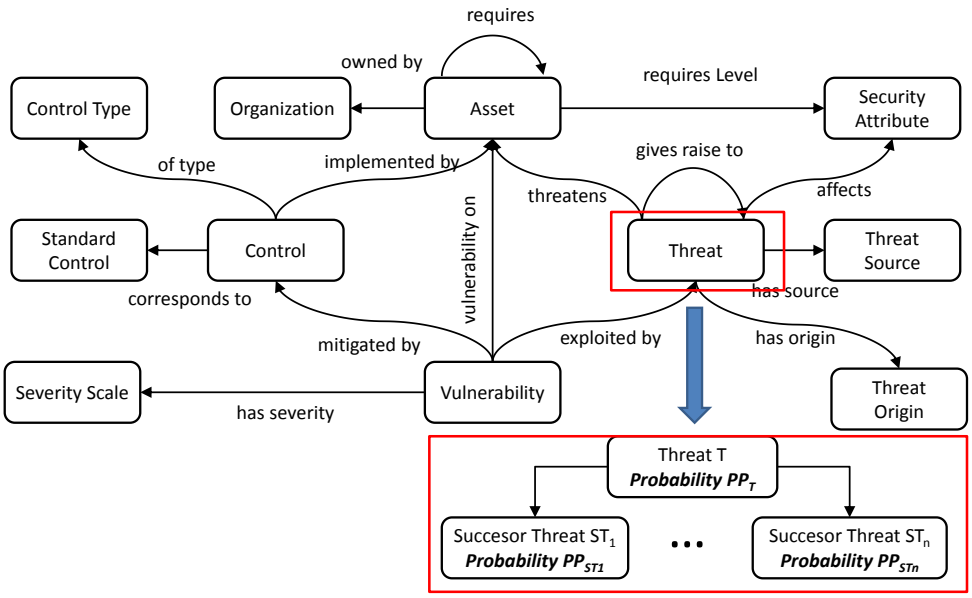


Figure 54. Security ontology used by Fenz et al. in (Fenz, 2011) and derived BN example

Indeed, the authors propose a BN structure which is derived from the security ontology, shown in Figure 54. This structure is compose of a set of n threats with their corresponding probabilities PP, and three probability states e.g. high, medium, and low. The BN is used to determine the threat probability taking into account that threats have predecessor threats (PT) and successor threats (ST).

3.7.6 Related work on Network Diagnosis through Bayesian Networks

In this section, we present the related work on Bayesian Networks applied to different network infrastructures such as bridged networks, VPN, IMS, IT infrastructures, enterprise networks, and optical networks. We discuss the different diagnosis strategies used, the limitations and the key learnings that could be worth being considered for programmable networks.

3.7.6.1 Optical Networks

Tembo et. al. in (Tembo et al, 2015) propose a self-diagnosis approach for GPON-FTTH access networks based on a reconfigurable three-layered dependency graph. The generic model covers two main fault cases, where the fault remains inside a component, not propagating to the rest of the network (local fault propagation) and distributed fault propagation, where faults are spread to the rest of the network. Their proposed generic model covers both local fault propagation and distributed fault propagation.

The first layer models the network topology and the distributed fault propagation among network components, the second layer models the local propagation inside a network component by means of a set of directed dependency graphs interconnected via the first layer, and the layer 3 describes a junction tree relying on the layer 2. This generic model is self-reconfigurable, in the sense that it can automatically learn changes in the network topology and in the local dependencies inside each network component.

The authors apply this generic model to diagnose a FTTH access network based on GPON (Gigabit Passive Optical Network), to model the upstream and the downstream accesses and diagnose loss of communication between the OLT (Optical Line Terminal) and the ONT (Optical Network Terminal), attenuation of the branching fiber between ONTs, and loss downstream communication between OLT and ONT.

In this context, the authors consider three types of layer 2 nodes, root causes, intermediate faults and alarms, where the root causes are the fibers, a faulty ONT or a misconfigured ONT. As advantage, some of the modelled variables included in the generic model such as the power supply of the OLT (AltOLT) take two values (faulty, not faulty) while other modelled variables such as the fibers take up to three values (OK, Attenuated (AT), Broken (BK)).

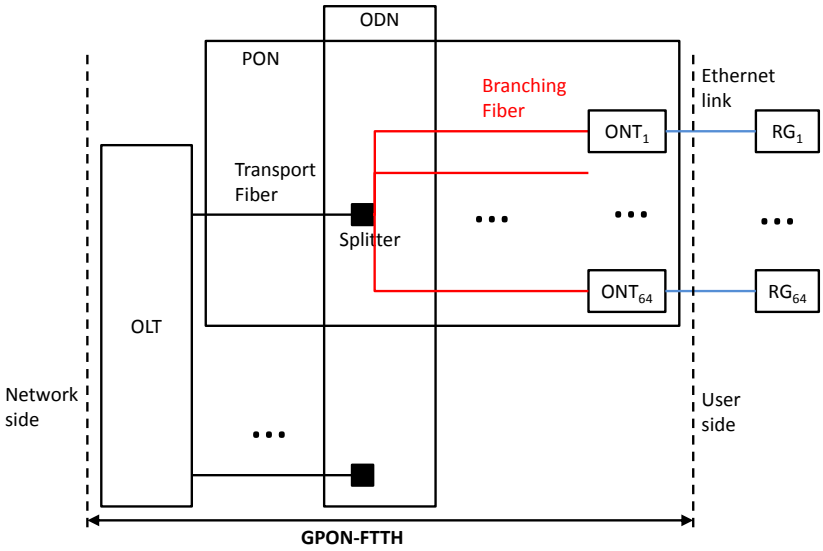


Figure 55. GPON-FTTH network architecture considered

Key learning and limitations: Two main limitations are considered in this work. Firstly, the self-reconfigurability capability of the generic model and the capability to track dynamic changes on the network topology is enunciated in a theoretical manner, but it has not been implemented. Secondly, the granularity of this generic model reaches the node components considered. For instance, they consider ONT (Optical Network Termination), OLT (Optical Line Termination), the transport and branching fibers acting as links or RG (Residential Gateway), but do not consider internal components inside the ONT or the OLT and those dependencies among those internal components, although it has also been enunciated in a theoretical manner that those interactions are comprised in the layer 2 model.

3.7.6.2 Bridged networks

Steinder and Sethi in (Steinder & Sethi, 2002) and (Steinder & Sethi, 2004) proposed a self-diagnosis approach for end-to-end services delivered over bridged networks. The authors propose an end-to-end service model decomposition where the connectivity between two network nodes at layer L+1 to realize a given service composed of two network functions of layer L+1 relies on a set of intermediate nodes (called by the authors as host-to-host services) operating at lower layers. Figure 56 shows this layered model where a service at layer L+1 between two nodes a and c relies on the service layer L which is in turn relying on two network functions of layer L in nodes a and c and so on in a hierarchical manner.

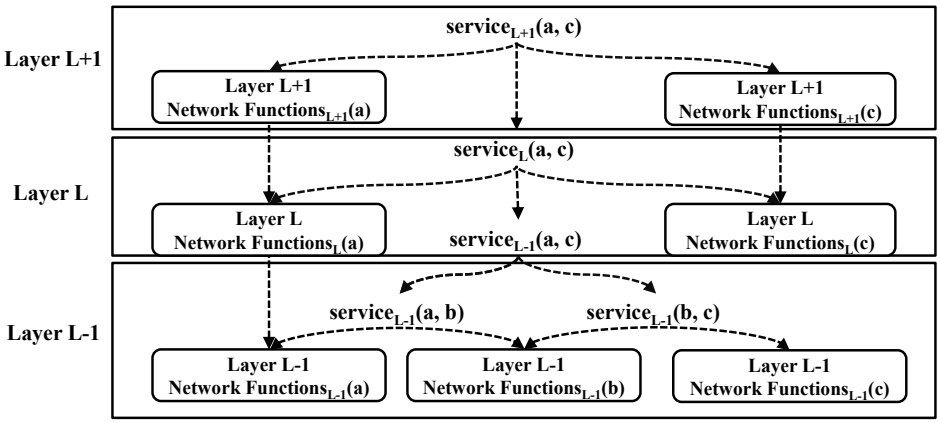


Figure 56. End-to-end service model proposed by Steinder and Sethi

With this service decomposition, the diagnosis model can be built from the network topology by following a divide-and-conquer mechanism that decomposes the end-to-end service in several host-to-host services, each of them relying on a single physical link. Figure 57 shows this decomposition, where the path realizing the end-to-end service is composed of several links, and each link is a possible root cause of a path failure. For instance, the path from hosts H_1 and H_2 which traverses bridges A, B and D, can be divided in host-to-host segments composed of one single physical link. For instance, two link root causes would explain a path failure on the network segment A-D: the links A-B or the link B-D. Thanks to this decomposition of physical paths, the authors can build the probabilistic dependency graph of end-to-end services in an automatic manner.

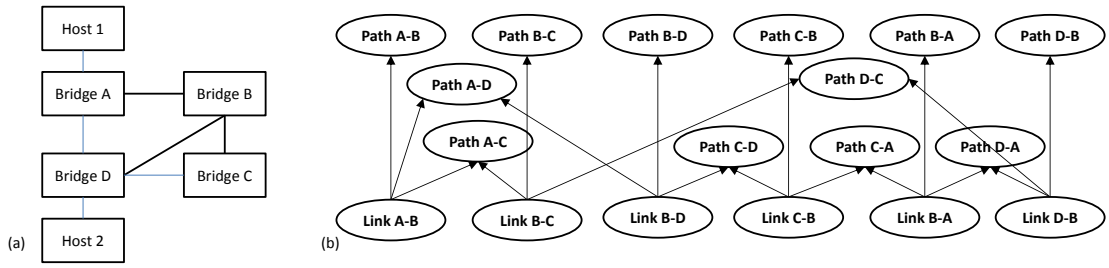


Figure 57. Decomposition of a bridge-to-bridge network topology in a dependency graph

The authors studied the inference to a great extent. Indeed, the authors proposed three different algorithms for the inference process such as the bucket elimination algorithm, an interactive belief updating, and the iterative most probable explanation. Also, the authors compare them in terms of detection rate, false positive rate, prediction capabilities, and maximum network size below 10 seconds of inference time, among other criteria.

Key learnings and limitations: The proposal to decompose a given service at a given layer in simpler services in lower layers will be taken into account for programmable networks, where those are composed of several layers: physical, logical virtual and service layers, as it will be further described in chapter 4.

The decomposition of a given network path in links as root causes is a very interesting approach for modelling programmable networks, which can be also composed of virtual and physical links. However, the authors do not consider as root causes the nodes connecting those different links, which was taken into account on other recent approaches such as (Bennacer et al, 2012), (Bennacer et al, 2013), and (Bennacer et al, 2015), (Kandula & Mahajan, 2010) or, (Hounkonnou, 2013) among other works. We will consider as possible root causes nodes and links in our self-diagnosis approach shown in chapter 4.

However, the authors only diagnose faulty links and do not provide with any methodology to build and update the probabilistic dependency graph with changes in the network topology, although the authors can take the traces of the network topology by using SNMP traps for their proof of concept, do not explicitly tackle this concern.

3.7.6.3 Virtual Private Networks

Bennacer et al. in (Bennacer et al, 2012), (Bennacer et al, 2013), and (Bennacer et al, 2015), proposed a self-diagnosis approach for VPN services. A VPN is shown in Figure 58 where a VPN backbone is connecting to sites (VPN1 and VPN2). There is one egress node CE (Customer Equipment) that connects the clients in between both sites through the VPN backbone. The VPN backbone is composed of PE (Premises equipment) nodes.

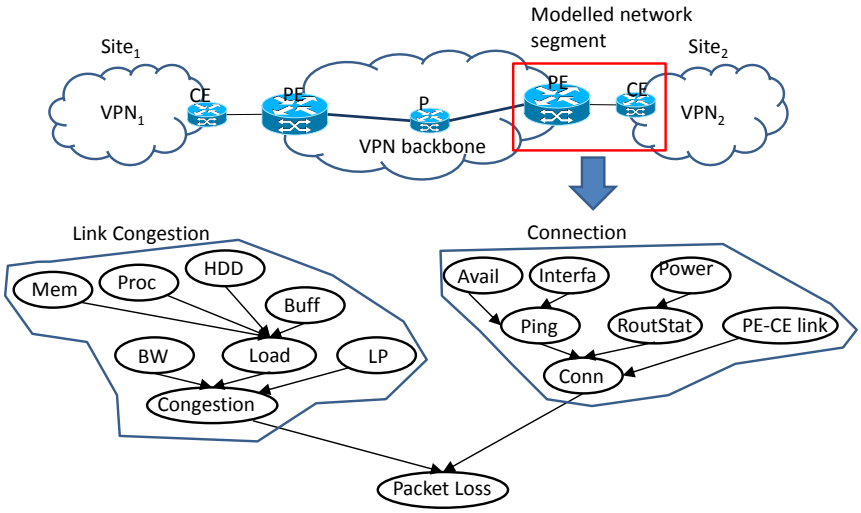


Figure 58. Example of VPN and example of calculated dependency graph for the packet loss problem

In their first work (Bennacer et al, 2012), they proposed a hybrid RCA module based on the combination of Bayesian Networks (BN) with Case-Based reasoning, called BN-CBR hereafter. Thanks to this hybrid approach, the high diagnosis time and the low accuracy provided by the BN algorithm can be improved to a great extent for large networks. The BN algorithm performs diagnosis by exploring the generated dependency graph and this inference process grows exponentially with the size of the network. The authors proposed the following methodology, shown in Figure 59:

Construction of the Bayesian Network through statistical chi-squared tests: Bennacer et al. in (Bennacer et al, 2013), proposed to generate the probabilistic dependency graph by computing the dependencies among the physical symptoms observed in VPN nodes through Chi-squared statistical tests. Thanks to this statistical approach, the probabilistic dependency graph can be built in an automatic manner and be updated with changes in the network topology.

The BN includes the metrics of the network equipment as vertices and the dependencies among the metrics as edges. The self-modeling algorithm starts with no dependencies in the BN and, each time a new equipment vertex in the BN, the self-modeling algorithm computes the dependency of that vertex with the rest of vertices in the BN and, if the chi-squared method considers there is enough statistical dependence, it adds the edge in the BN between both vertices. The approach can update the graph with the addition and removal of vertices by computing the dependencies of the newly added vertices with the rest of vertices in the graph.

The dependency graph includes as modelled variables the metrics associated to each network node in the VPN such as CE, PE, or P. The following metrics are monitored for each network node and link: router status, processor, memory, HDD (Hard Disk Drive), link bandwidth, connection, congestion, buffer, delay, among others.

Let's imagine that the PE-CE link is experiencing packet losses, so the dependency graph proposed (shown in Figure 58) assumes two main root causes explaining packet loss, whether congestion on the link and the connection of both routers (PE and CE). In turn, congestion on the link depends on the link bandwidth, and the load in the routers. The load on the routers depends on metrics such as memory, HDD, buffering, or processing. The connection of the routers depends on their status, which in turn depends on the power supply. Also, the connection depends on the interfaces of the router, which both must be available and must respond to ping.

Expression of a failure as a problem-case: Once the dependency graph is built by using the independence tests in the previous step, the failure occurrence is expressed as a problem-case. The failure occurrence is injected as a problem in the BN, while the failure occurrence is injected as a case in the CBR. This inference approach is based on examining the subset of vertices in the BN that experience discrepancies. First, BN is examined from the faulty vertex, where the problem is detected, and examines those vertices at two hops of distance identifying a subset in the BN, as shown in Figure 59 example, where V_1 and V_3 are at two hops of distance.

Optimizing the inference procedure by using Message Parsing inference: Once is identified that the problem is not similar to any case stored in the CBR, the inference process starts. This inference process is based on Message Parsing, a variant of Bayesian Networks to reason over graphical models, especially poly-trees. As the BN is a DAG, this algorithm only is executed on the potential root cause vertices, excluding those outside the aforementioned subset. In this way, the authors tackle the high computational cost in the inference process and the inherent scalability concerns in BN.

Expressing and saving the result or the RCA as a solution-case: Once the problem is solved by the inference process is stored as a solved case.

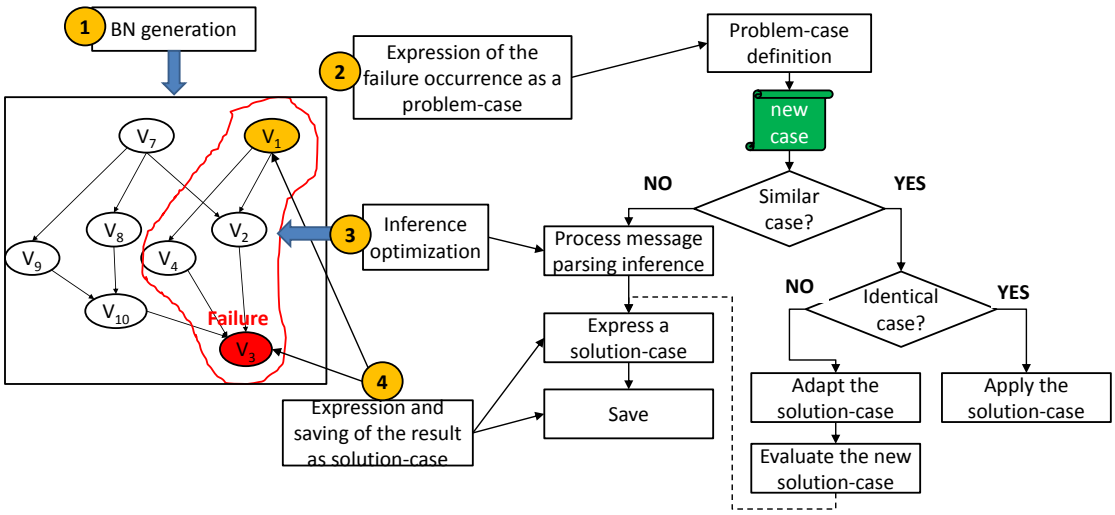


Figure 59. Hybrid BN-CBR self-diagnosis approach for VPN proposed by Bennacer et al.

In (Bennacer et al, 2015), the authors proposed another inference mechanism to improve further the scalability of BN. This mechanism is based on partitioning the BN graph into clusters. This approach duplicates some vertices of the BN in other clusters when those impact several segments in the network. The approach is a recursive inference mechanism. This inference procedure works as it follows:

- It first applies the aforementioned inference based on BN and CBR in the cluster where the fault is identified to identify the most probable root cause
- If that root cause is a duplicated vertex leading to other cluster, the inference is then launched in that new cluster taking that duplicated vertex as evidence
- Once the inference is done over all the clusters, a comparison among the root causes per cluster is made to establish the root cause

Key learnings and limitations: The authors tackle several problems in BN, the scalability, the inference speed, the accuracy, and the automation in the generation of the BN. However, it is not clear how the dependency graph can be updated and regenerated in such a high network as expected in SDN and NFV. As additional limitation, the authors only diagnosed at a physical level, leaving out application and logical resources in thee VPN connections. The granularity of the diagnosis remains at the network node level, only considering links and routers, where smaller sub-components are not considered. Furthermore, the diagnosis is focused on the physical network nodes and is not considering the logical resources (e.g. virtual resources) running over them.

However, the authors chose independence tests as self-modelling approach, where the value of the significance level, which determines if variables are dependent or not, may lead to errors when building the dependency graph. Indeed, the authors chose a value of 0.05, which has a higher degree of tolerated risk than like 0.01, for instance, but it requires less time to compute the dependencies. There is a trade-off between the degree of tolerated risk and the time taken to compute the dependency graph. Other limitation is that the dependency graph built from the VPN always has the same dependencies as it was composed in a modular manner, so those dependencies could be established by means of templates to be assembled according to the network topology.

Bennacer et al. focus not propose any specific approach to deal with uncertainty in any of their works, because in part this issue was solved by reducing the root causes with the information stored in previous problems and cases, and by reducing the subset explored in the BN with their proposed clustering and message parsing inference algorithms.

3.7.6.4 Enterprise networks

Kandula et. al. in (Kandula & Mahajan, 2010) present NetMedic, a self-diagnosis approach for diagnosing enterprise networks based on computing networks by exploiting the information retrieved from the operative systems and applications stored in logs. Firstly, the authors examine carefully the trouble tickets from an enterprise network and identify their symptoms and associated root causes by analysing the logs of the system. The authors then classify the top ten most recurrent faults in enterprise networks in three main categories: how the fault manifests, if the fault impacts an individual application of an entire machine, and what is the identified cause of the fault (root cause). Once the authors have identified the needs of the diagnosis system, they focus on a web server example to identify those variables worth modelling. For instance, the authors identify generic variables for the web servers and application variables, both of them shown in Table 8.

Table 8. Variables considered in a Web server

Generic variables	Application variables
% processor time	current files cached
% user time	connections attempts/sec
input-output data bytes/sec	files sent/sec
thread count	get requests/sec
page faults/sec	put requests/sec
page file bytes	head requests/sec
working set	not found errors/sec

The authors generate in an automatic manner the dependency graph by using a modular approach that connects a set of templates. They define specific templates for each network element in the IT infrastructure such as a machine, an application process, a neighbour set and a path, which are shown in Figure 60. Each template is characterized by several state variables to achieve a detailed diagnosis. For instance, for a web server it includes the aforementioned application and generic variables.

The authors also define weights for the edges in the dependency graph (low, medium, and high) to take into account the dynamic dependencies that depend on how the network resources interact. For instance, a weighted dependency graph includes additional information in the edges that may influence on the propagation of faults despite the CPT are the same. However, adding weights to the dependency graph implies a higher computational cost for the inference because the CPT are bigger in size.

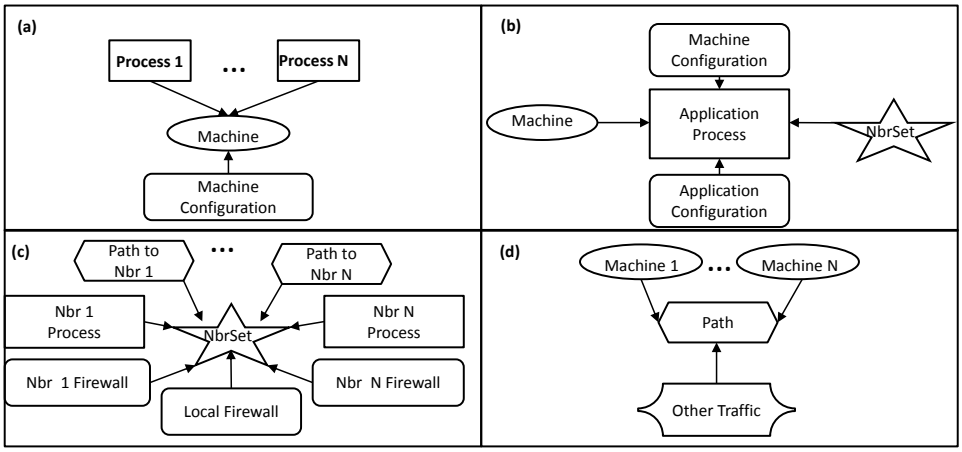


Figure 60. Proposed templates: (a) Machine, (b) Application, (c) NbrSet, and (d) Path

Their self-diagnosis framework can diagnose faults in the machines, but it can also pinpoint those applications as the origin of the malfunction if needed. The authors deal with the trade-off of keeping a minimum level of specificity to diagnose until applications running on machines but without being aware of application specific knowledge. This is because the needed variables to be modelled by a self-diagnosis system would not be reasonable to perform fast diagnosis.

Key learnings and limitations: The identified variables by the authors for the servers can be easily applied to NFV, where the VNFs are embedded in commodity servers. For instance, the application variables will be different in accordance with the specific VNF, while the generic variables will refer to the hosting machine, but taking also into account the VM and hypervisor architecture.

The advantage of adding weights to the dependency graph is a promising idea for programmable networks where there will be even more dynamic dependencies that may depend on the interaction among components that static dependencies with binary values will not model with enough accuracy. For instance, the network topology is dynamic in programmable networks, but the interactions between the components are much more dynamic so a weighted dependency graph could be useful to model the strength of those dynamic interactions. In those cases, the update of the dependency graph would consist of updating only the weights (as a null or a near to zero value implies the removal of the edge).

The idea to model each network component per separate by using a template allows creating a more complex dependency graph by using a divide-and-conquer approach. We propose in the chapter 4 a self-modeling approach that builds the diagnosis model from a set of finer-grained templates, inspired by this approach based on templates. Nevertheless, our templates are extendable and describe the inner dependencies of a given network resource with respect to its inner components, which it has not been done yet in SDN and NFV.

3.7.6.5 IT networks

Bahl et. al. in (Bahl & Chandra, 2007) propose Sherlock, a self-diagnosis algorithm for IT infrastructures based on a self-modeling approach that computes the service dependencies in a given network topology when clients access to some database in the IT infrastructure.

The authors consider a multi-layered model covering both hardware and software root causes in an IT infrastructure. The Sherlock solution is shown in Figure 61, where it can be seen that it identifies and models the service dependencies in a service-level dependency graph, and then it adds information on the network topology on that service-level dependency graph by using the traceroute results. The network topology information is translated into a set of vertices where the root causes vertices are the routers and links for each path traversed by each user.

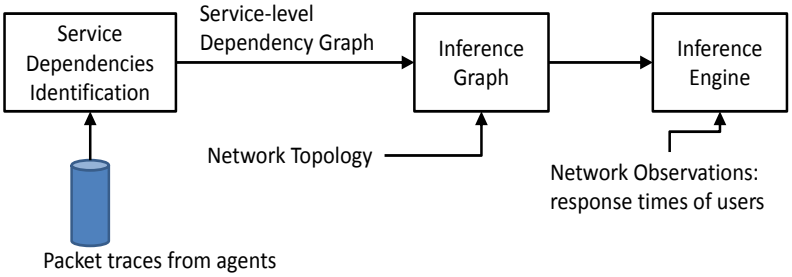


Figure 61. Sherlock architecture

The dependencies between the services and underlying hosts are reported by the agents, which are deployed in the hosts or near the routers, and monitor the packets sent by the users when they access to a given file in a database. The authors consider as root-cause only physical network elements such as routers, client hosts, and webservers that may lead to end-user failures.

Key learnings and limitations: The authors proposed a self-modeling block that automatically builds the model by first discovering the services' dependencies and then adding the network topology dependencies in a multi-level dependency graph. The Sherlock architecture, that first generates a service-level dependency graph and adds the dependencies of the underlying network topology, is identified as a suitable architecture to build a topology-aware and a service-aware multi-layered dependency graph for networking services over SDN and NFV. Nevertheless, our proposed multi-level dependency graph is going up to the subcomponent level and including also logical and virtual layers on that model.

An interesting characteristic of this work is that this multi-layered dependency graph takes into account unmodelled root causes by adding two types of root –cause vertices (always troubled and always down) that model external factors that may lead the user to perceive a degradation.

However, this approach does not consider the software running on the hosts. In SDN and NFV, software running on hosts such as OpenFlow applications on the switches and SDN controllers, or the VNFIs must be modelled, but also their dependencies with the underlying physical network topology. In addition, the authors not consider finer granularity on the root cause, as the granularity considered remains at node level e.g. a computer, a DNS server, a service, any server, a router or an IP link. Other sub-components inside nodes such as interfaces or CPU are not included in the diagnosis. In addition, this approach does not consider that a single host may embed a virtual machine which in turn embeds several virtualized servers, which in programmable networks is already a fact, where a host embeds one hypervisor as the base of one or several VM embedding one or several VNFs.

3.7.6.6 IP Multimedia Subsystem Networks (IMS)

Hounkonnou et al. in (Hounkonnou, 2013) propose a self-diagnosis approach to diagnose the IP Multimedia Subsystem (IMS). The authors deal with two main limitations of model-based self-diagnosis: firstly the generation of the dependency graph and secondly the inference over large dependency graphs. The self-diagnosis approach is based on five main steps, defined hereafter:

- Generation of the generic model (generic BN) describing the resources used by the failing resource
- Locate BN instances of the generic model in the IMS network
- Perform inference in the current BN instance
- If uncertainty is high explore and add other patterns to the current BN
- Repeat extension of the current BN until root cause is diagnosed with enough confidence

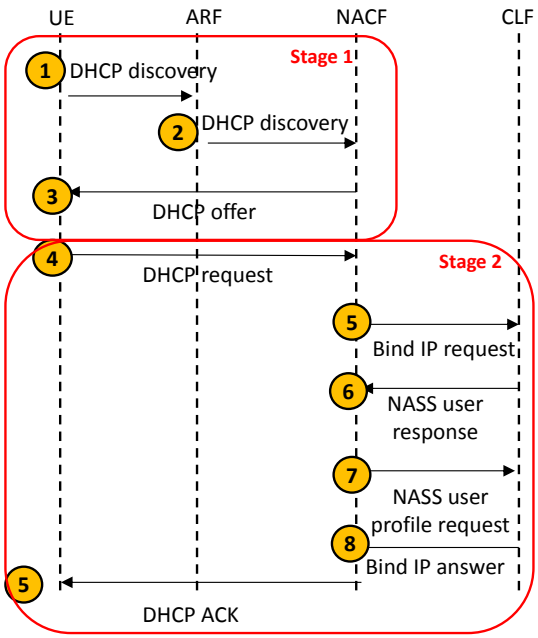


Figure 62. Simplified IP configuration sequence diagram in IMS

We explain the different steps by considering a malfunction in the IP configuration service in IMS affecting a given user. The self-diagnosis approach is to determine which network resource at any of the four IMS layers is responsible for that failure.

Generation of the generic model: First, it generates offline a generic model that is based on the IMS standard sequence traces.

In this concrete case, the authors analyse the traces from the IMS standard of the IP configuration service. This service assigns an IP address to a given UE (User Equipment) by means of DHCP (Dynamic Host Configuration Protocol). Figure 62 shows a simplified diagram of this process, which generally consists of two stages, involving the following entities, the UE, the ARF (Access Relay Function), the NACF (Network Access Configuration Function), and the CLF (Connectivity Location Function).

From these IMS traces, the authors proposed a generic model, which is formalized as a BN (or probabilistic dependency graph) and depicts the dependencies among the different types of resources across the four IMS layers (physical, functional, procedural and service) involved. This generic model is shown in Figure 63, where it can be seen that it does not contain the users, or the instances of the network resources actually involved in the IP configuration service. The generic model depicts how fault may propagate in the IP configuration service when a UE asks for an IP address. Briefly described, if any of the IMS functional blocks or underlying physical resources involved in the IP configuration service fails, the UE will not get any IP.

The generic model contains as vertices the network segments (the first mile, the aggregation and the metro core), the physical nodes, the functional interfaces (UE-ARF, ARF-NACF, UE-NACF, and NACF-CLF), the procedural blocks or stages, and the failing service, which is rather an input vertex.

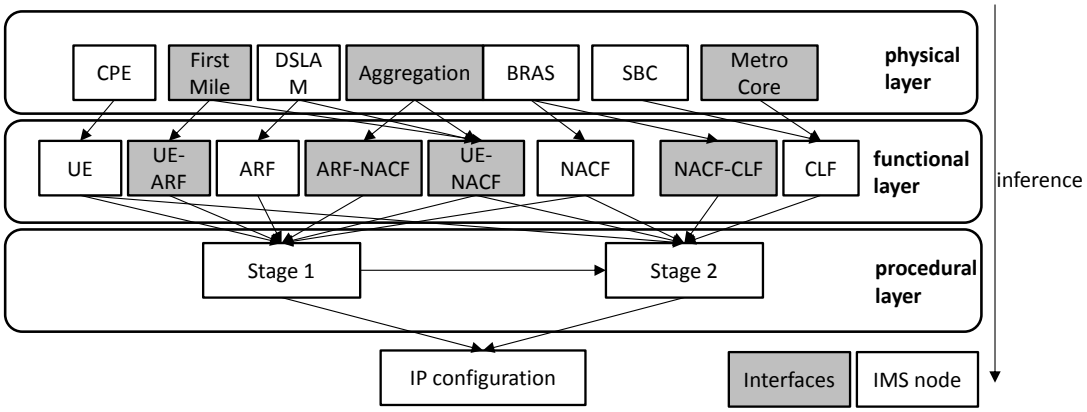


Figure 63. Generic model proposed for the IP configuration service

Locate BN instances of the generic model in the IMS network: The following phase of the self-diagnosis is to identify the users deployed in the network using the affected IP configuration service in IMS. Then, the authors observe the state of the IP configuration service and detect that a given user UserX cannot get the IP. The user instance is retrieved and the BN instance is built from the generic model for the user UserX, as shown in Figure 64. This BN instance is represented by a probabilistic dependency graph comprising the actual dependencies of the current user UserX. This BN instance depicts the actual equipment making possible the connection of the user to the IMS network and performing the IP configuration service.

Perform inference in the current BN instance: Then, the inference is made over the current BN instance, which is composed of one user. This BN instance is fed with a set of observations such as the state of the IP configuration service, which is down (vertices have two states, up and down), and other observations retrieved from the user's equipment, for instance, the state of the CPE (up) or the DSLAM (up).

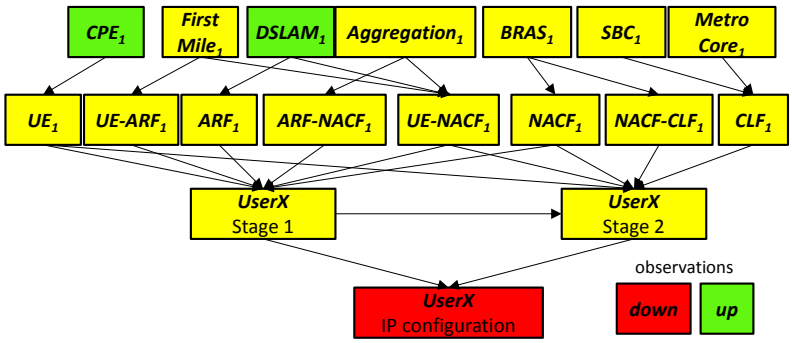


Figure 64. BN instance extracted from the generic model for a user User X

The inference engine is based on BN and starts propagating the observations based on the CPT through the probabilistic dependency graph giving a set of a posteriori probabilities for each network resource at each IMS layer. The authors calculate the entropy of the a posterior distribution to quantify how uncertain the root cause is. However, this root cause may be uncertain because several network resources can be responsible for the failure.

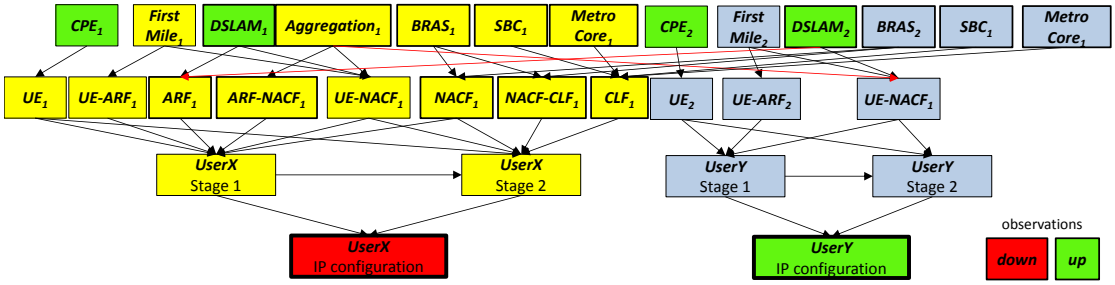


Figure 65. Extension of the BN with two BN instances from users User X and User Y

Extension the current BN with more patterns: If the root cause previously calculated over the current BN is not pinpointed with enough confidence, the authors propose to extend the BN with more users that are sharing resources with the previous user UserX.

This is a novel approach to analyse large networks and reduce the uncertainty based on an intelligent exploration of the user instances connected. The authors proposed to progressively extend the BN by adding those instances of the users sharing resources with the impacted user (User X) and performing the diagnosis over the resulting BN, shown in Figure 65. For instance, the addition of UserY instance to the current BN will imply that the resources shared between both users Userx and UserY are discarded if and only if the UserY is not impacted by the failure on the IP configuration service. This process is iterative, until the root cause is calculated with enough confidence, which is measured in terms of uncertainty.

Key learnings and limitations: Although the authors consider a multi-layered generic model containing the physical network elements, the functional elements, and the procedural elements, the granularity of those elements is limited the internal components inside those blocks are not modelled, as a result, the diagnosis granularity remains at the IMS node level. Alternatively, the generic model is only extracted from the IP configuration service, but the authors did not work on other interesting cases such as the plenty of procedures in IMS such as the IMS registration or IMS call origination.

In addition, the dependencies between the physical infrastructure and the functional blocks shown in the generic model are statically fixed. However, those dependencies may change and this was not taken into account by the authors on their work. In programmable networks, we cannot assume that a given physical node is going always to embed the same VNFI, otherwise this would constraint to a great extent the flexibility of programmable networks what goes against NFV principles.

The authors assume that the network topology remains static during the diagnosis process, so the dependency graph is static. However, this is not true in programmable networks, indeed, we will provide with the capability to regenerate and update the dependency graph even once the diagnosis is already exploiting the dependency graph.

Indeed, in the generic model, the aggregation, metro core and first mile are vertices do not contain finer-granularity such as the actual network segments composing them (until a physical link level). In addition, the DSLAM, the BRAS, and the CPE are static in this work.

However, although the methodology proposed for reducing the uncertainty on the root cause on the malfunctioning service by exploring the clients connected to that service is significantly novel, it seems is not automatic. Indeed, the BN is extended with a few user instances, but it has not been devised how a given user instance is added in an automatic manner to an existing BN and how it can be removed from the BN. The author of this thesis worked on how to extend automatically a given BN with a set of users' instances, where each user instance is described by a set of identifiers of the equipment and the instances of the network used.

3.7.7 Topology-Aware and Service-Aware self-diagnosis

We propose to classify the aforementioned self-diagnosis approaches into topology-aware and service-aware, which are defined here. On one hand, topology-aware self-diagnosis approaches build the dependency graph from the network topology and this graph only considers how faults in network resources could impact other network resources. The impact of faults in network resources on services or clients is not then considered. On the other hand, service-aware self-diagnosis approaches are an extension of topology-aware self-diagnosis approaches in order to take into account the impact of faulty network resources on services as well as the clients using them. The diagnosis can then focus on faults leading to service failures and affecting client experience.

We make this distinction because a self-diagnosis approach needs additional information apart from the observed malfunction to pinpoint the root cause more accurately. A self-diagnosis mechanism receives some information on the context of the fault such as the network segment where the fault occurs or how many services are simultaneously impacted by that fault. This additional information will help to delimit the possible root causes by discarding some network elements because those do not add valuable information.

To exemplify why the service-aware self-diagnosis is more accurate than the topology-aware self-diagnosis and how the service-aware self-diagnosis can discard network elements with this additional information, we show the following example, a faulty link in a network composed of 10 nodes and 20 links. Three types of network components are considered: the CPU of the nodes, the interfaces of the nodes and all the links in the network topology. If we consider a topology-aware self-diagnosis mechanism, it includes in the diagnosis all the network components and their respective observations, but it does not include any additional information. This means that, there are $k_i + 1$ possible root causes per node (the interfaces of each node and its CPU), and 20 additional root causes for links. This yields 66 possible root causes that explain the faulty link. This example is seen in Figure 66(a).

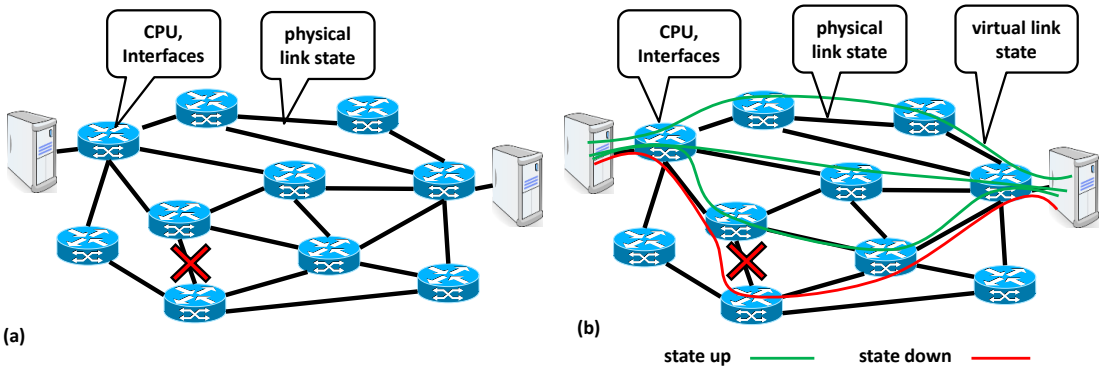


Figure 66. Differences between both types of self-diagnosis: (a) topology-aware, (b) service-aware

Nevertheless, if the self-diagnosis mechanism is service-aware it then considers additional information such as the state of virtual links deployed over that network topology, so the aforementioned root causes will be drastically reduced and thus the uncertainty because it only includes in the diagnosis those network components used by the virtual link which is affected by that faulty link. In this concrete example, the possible root causes will be reduced from 66 to 28, which is a 57.5 % of fewer possible root causes. This example is seen in Figure 66(b). However, the service-aware self-diagnosis mechanism needs to retrieve the dependencies of each virtual link from the network topology, which is even more challenging in virtualized environments such as programmable networks, where those are highly dynamic, as we will present in next chapter.

In conclusion, service-aware correlates additional information to reduce the possible root causes in the network and this approach is necessary for multilayered networks in general, but especially in programmable networks, where information coming from physical, logical, virtual and services layers is correlated to delimit the fault.

3.7.7.1 Topology-Aware self-diagnosis approaches

A first example of topology-aware self-diagnosis mechanism is the proposed by Steinder and Sethi in (Steinder & Sethi, 2002) for end-to-end services over bridged networks. The authors consider as symptoms the loss of con-

nectivity, excessive delay or excessive packet loss, respectively due to broken links, buffer overflows, or transmission link noise, but the authors do not model how those faults, failures and degradations could affect the users over that bridged network.

A second example is the self-diagnosis mechanism proposed in (Bennacer et al, 2012), (Bennacer et al, 2013), and (Bennacer et al, 2015) for VPN. The diagnosis mechanism took into account the metrics of the different nodes such as buffer, delay, processing, or router status, among others. However, this mechanism did not consider the impact of those metrics on the VPN service in itself or the clients using that service.

A third example of topology-aware self-diagnosis mechanism is the proposed by (Tembo et al, 2015) for GPON-FTTH infrastructures, where the diagnosis mechanism took into account attenuations on the fibers, faulty OLTs and ONT, misbehaviors on their configuration, but the authors did not take into consideration how those faults and failures impact the end-users provided with services over the FTTH infrastructure.

3.7.7.2 Service-Aware self-diagnosis approaches

A first example of service-aware self-diagnosis mechanism is the proposed in (Bahl & Chandra, 2007) for IT infrastructures. This mechanism only focuses on faults impacting the clients of the IT infrastructure by taking into account the service response times measured by a set of agents deployed in the IT infrastructure. Indeed, Sherlock is conceived to not report those faults that do not impact the users e.g. if a server has a high CPU usage, Sherlock does not even detect it as long as the users requests are not affected. Service availability is classified in three possible states: up when its response time is normal, down when there is no response or there is an error, or troubled when the response falls outside of normalcy.

A second example of service-aware self-diagnosis mechanism is the one proposed by in (Hounkonnou, 2013) for IMS networks. Indeed, this mechanism is service-aware because it only diagnoses the user affected in a given IMS service by building the corresponding BN of that user and incorporating its observations. If this mechanism analyzes other users is with the only aim to reduce the uncertainty in the root cause for the affected user. Indeed, the strategy to reduce the uncertainty by progressively extending the current BN instance allows adapting the diagnosis in accordance to the observations found in the user of the IMS network.

A third example is the self-diagnosis mechanism proposed by Kandula et al. in (Kandula & Mahajan, 2010), where the self-diagnosis mechanism is to solve those experienced issues of the users of the IT infrastructure. Indeed, the users open the experienced incidences and upload them in a trouble ticket system which are analyzed by the authors to conceive that framework.

A fourth and last example is the work from (Georghe et al, 2015). The authors propose SDN-RADAR, a multi-agent distributed network troubleshooting mechanism for SDN that identifies faulty network links in the data plane impacting user experience. This approach is service-aware because it focuses the diagnosis only on those malfunctions that impact user experience in terms of degradations. This approach is to be further explained in the chapter on fault management on SDN, in chapter 3. However, this approach only focuses on links in the data plane and does not deal with large and dynamic network topologies.

3.8 Conclusion

We propose a multi-layer self-diagnosis framework to diagnose networking services over combined NFV and SDN environments, which to the best of our knowledge has not been tackled before. This framework is composed of a topology-aware and a service-aware self-modeling approach, by diagnosing and correlating physical, logical, virtual and services layers, while considering the dynamic dependencies of the networking services with the underlying virtual, logical and physical resources. As conclusion, we describe here the most important contributions of our proposed self-diagnosis framework with respect to the aforementioned works.

Multi-layer self-diagnosis framework: Our approach utilizes a multi-layered probabilistic dependency graph, like (Bahl & Chandra, 2007) and (Hounkonnou, 2013). In our approach, this dependency graph is adapted to SDN and NFV specificities, and it covers the diagnosis of physical, logical, virtual resources and the corresponding networking services. Contrarily to (Georghe et al, 2015), which only diagnoses faults in the data plane links, our multi-

layer approach diagnoses data plane and control plane, by diagnosing control links, the SDN controller and its internal physical and logical inner components, but also the internal physical and logical components (ports, CPU, applications, VNFs, etc.) inside hosts and switches.

Finer diagnosis granularity: Contrarily to the approaches from Steinder in, (Bennacer et al, 2012), (Bennacer et al, 2013), and (Bennacer et al, 2015), (Bahl & Chandra, 2007) and (Hounkonnou, 2013), which diagnose up to node level, we propose a self-modeling approach that builds the diagnosis model from a set of finer-grained templates, inspired by the diagnosis approach of Kandula et. al. in (Kandula & Mahajan, 2010) for enterprise networks, based on templates. Nevertheless, our templates are extendable and describe the inner dependencies of a given network resource with respect to its inner components, which it has not been done yet in SDN and NFV.

Reduced diagnosis uncertainty: In this thesis we will propose a two-level self-diagnosis approach for diagnosing SDN and NFV combined infrastructures. We first propose a self-diagnosis approach at a topology level (topology-aware self-diagnosis) that generates on-the-fly the probabilistic dependency graph from the network topology, the type of control, and the logical resources running on top of nodes. However, this self-diagnosis approach does not take into account the impact of faulty physical and logical nodes on services. As a result, the diagnosis is focused on the entire network topology and logical resources so the uncertainty on the root cause is high when the diagnosed network topology becomes large.

To solve this issue, we propose to extend the self-diagnosis approach to be service-aware approach, by taking into account the impact of the physical and logical resources on the service layer. This second approach correlates the service view with its underlying network states at virtual, logical, and physical layers, in order to reduce the uncertainty in the root cause. This approach allows us to extend and reduce the multi-layer probabilistic dependency graph in accordance with the diagnosed malfunction to reduce the uncertainty, following a similar strategy to the one proposed by Hounkonnou in (Hounkonnou, 2013).

On-the-fly self-modeling: Contrarily to the approaches from (Bennacer et al, 2012), (Bennacer et al, 2013), and (Bennacer et al, 2015), (Bahl & Chandra, 2007), (Hounkonnou, 2013), (Georghe et al, 2015), and (Tembo et al, 2015) that diagnose static network topologies, we propose a self-modeling approach to diagnose dynamic network topologies and dynamic deployed services based on virtualized network functions. Our self-diagnosis framework is based on a self-modelling approach that discovers the dependencies in a deterministic manner and regenerates the model on-the-fly with changes, unlike the self-modeling approaches proposed by Bennacer and Bahl, which may have false positives as a result of an inappropriate 'significance level' parameter when calculating the dependencies.

Chapter 4 Self-Diagnosis architecture for programmable networks

4.1 Introduction

This chapter presents in details the PhD approach. The first section discusses where to possibly locate a self-healing system inside a SDN architecture. It also motivates the best position to be considered of it. The second section presents the systemic view of a Self-Healing system to ensure the availability of end-to-end services.

The third section of this chapter is to detail a self-diagnosis framework that is relying on multi-layered and finer granular templates for diagnosing the dynamic networking services within an SDN and NFV environment.

The Self-diagnosis framework encompasses:

1) Multi-layered templates definition: these are to identify what to supervise while taking into account the physical, logical, virtual and service layers. These templates are finer granular, extendable and machine-readable.

2) Self-modeling module: It takes as input the previously cited templates, instantiates them and generates on-the-fly the diagnosis model that includes the physical, logical, and the virtual dependencies of networking services.

3) A service-aware root-cause analysis module: It takes into account the networking services' views and their underlying network resources observations within the aforementioned layers.

4.2 Proposal of a Self-Healing architecture for SDN

In this section, we propose a self-healing architecture for SDN infrastructures. We first evaluate the possible locations of a self-healing system and evaluate the drawbacks and advantages of each position.

4.2.1 Position of a self-healing system in the SDN infrastructure

In the following section, we discuss the possible locations of the self-healing system in the SDN infrastructure. We have taken into account the following requirements:

- The information taken as input by the Self-Healing system
- The visibility of a self-healing system of the network topology
- The entities interacting with the Self-Healing system and their interfaces
- The frequency of the interactions with the Self-Healing system
- The time scale of the recovery actions sent by the Self-Healing

- The time scale of the monitored information analyzed by the Self-Healing

In this regard, we identify and discuss four different alternatives for its placement in accordance with the four planes identified in a SDN infrastructure: a) the application plane, b) the data plane, c) the control plane, and d) the management plane.

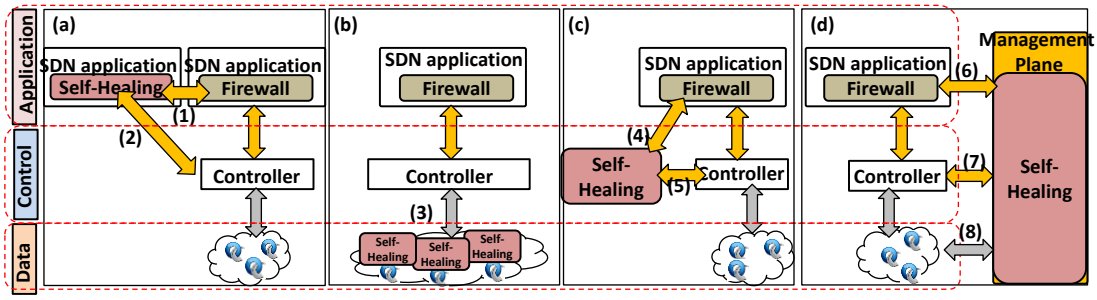


Figure 67. Locations of the self-healing architecture: (a) application, (b) data, (c) control, and (d) management plane

4.2.1.1 In the application plane

In this case, the self-healing system is located in the application plane of SDN. It acts as a SDN application running above the SDN controller, as shown in Figure 67(a).

The self-healing system uses the northbound interface to communicate with the SDN controller (2). Thanks to this interface, the self-healing system can obtain a global view of the network provided by the SDN controller. However, the network topology information is only available if the SDN controller is healthy. The recovery actions and root cause suggestions are directly sent to the SDN controller, which will install flows to avoid the failing elements. However, those types of faults non avoidable by changing the flows, will not be solved.

The self-healing system can also act on the SDN applications via the northbound interface (1) or other type of interface defined by the SDN applications. Thanks to this interfaces the self-healing system can send them orders to reprogram the data plane via the SDN controller. One example is when the self-healing system detects a massive registration of switches in the network topology. Then, the self-healing system can send an order to the Firewall SDN application to block their access to the network by installing on those registered switches blocking flows. However, for a large number of SDN applications, there can be scalability issues.

4.2.1.2 In the data plane

The self-healing system is located in the data plane as a module embedded inside each switch of the network, as shown in Figure 67(b).

The self-healing system is connected to the SDN controller via the southbound interface (3), where the advantages of such a self-healing system is the capability to act fast on the root cause without involving the SDN controller and avoiding unnecessary overhead. However, the distribution of the self-healing module may introduce scalability concerns where several self-healing modules try to contact the SDN controller simultaneously.

In addition, a central self-healing manager should interface with those distributed agents to relax this condition. The self-healing system perceives local faults and failures on that switch and in the neighborhood, but it does not have a network-wide vision. The self-healing system can only apply local recovery actions.

4.2.1.3 In the control plane

The self-healing system is located in the control plane, as shown in Figure 67(c).

The self-healing system can interact with the SDN controller through its northbound interface API (5). It can also interact with it through the westbound or eastbound API interface, if those are defined. Thanks to this interface, the SDN controller is aware of the network topology.

The self-healing system can also interact with the SDN applications via the northbound interface (4) or other type of interface defined by that SDN application. This is necessary when the encountered malfunction must be solved by changing the configuration set by the SDN application.

4.2.1.4 In the management plane

In the management plane, the self-healing system takes a transversal role, as shown in Figure 67(d). This position is very similar to the application plane, but in this case, the self-healing system can interact with the three planes of the SDN infrastructure. The main advantage is the transversality, where the self-healing module will then receive different measurements of the different planes and it will integrate them. The recovery actions will be different for each type of equipment and the considered plane, which will depend on the root cause. The self-healing system interacts with the SDN controller through the northbound API (7) to take profit of its centralization and it network-wide view of the network topology.

Alternatively, the self-healing system can also dialogue with the SDN applications (6) to send orders that reprogram the data plane via the SDN controller, but also, the self-healing system can directly act on the data plane via the southbound protocol (8) for more urgent actions and avoid unnecessary intermediation with the applications or the SDN controller.

In this last architecture, several actions are possible:

- The capability to interact with the SDN controller via the northbound interface and extract the network topology independently from the southbound protocol.
- The capability to send network-wide recovery actions to the SDN or the SDN applications (e.g. modification of a network path)
- The capability to interact with the SDN applications to reprogram the data plane via high-level policies sent through the northbound interface
- The capability to directly interact with the elements inside the data plane to set local recovery actions in a local manner which do not involve the SDN controller (e.g. change the master controller in a given switch)

As conclusion, the location of the self-healing system, the option (d) was chosen, due to the possibility to interact with the data plane, application, and control planes because of its transversal nature.

4.2.2 Self-Healing architecture for SDN infrastructures

This section proposes a self-Healing architeture to ensure the availability of end-to-end services over SDN infrastructures, shown in Figure 68. As presented in the previous section, this framework is located in the transversal management plane of the SDN infrastructure to leverage the logical centralization of the intelligence inside the SDN controller, which has a global view of the network topology. This centralization has the advantage that the self-healing system can take into account the whole network topology state and proceed with more global recovery actions based on this wide view.

The self-healing system acts in the three planes of SDN and in the service plane located on the top of the SDN infrastructure, by taking observations from the network and launching or suggesting recovery actions. It interacts with the SDN architecture by:

Manageming the SDN applications: The self-healing system manages the SDN applications to face malfunctions and changing conditions on the underlying nodes where they are deployed. One example is a threshold alarm on the CPU of a VM hosting the Firewall SDN application, where the self-healing system should redeploy that SDN application on another location.

Programming the data plane: The self-healing system acts on the data plane to set specific configurations on legacy equipment or executes a manual installation that the SDN controller cannot perform by itself.

Programming the data plane through the SDN controller: The self-healing system programs the data plane by means of the SDN controller. In this case, it uses its provded northbound interface to program the data plane in a hardware-abstracted manner (e.g. a SDN application that establishes a given path between two hosts).

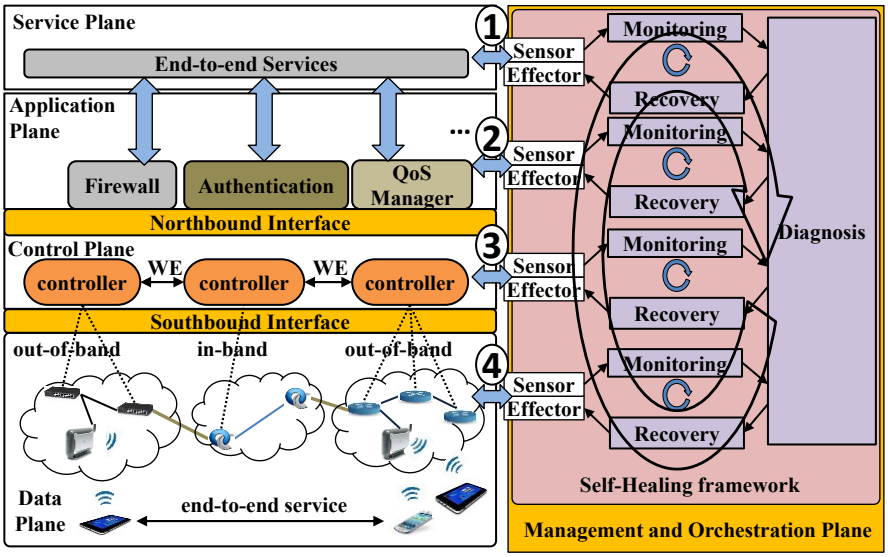


Figure 68. Multi-control loop Self-Healing architecture for SDN

This architecture can be applied to centralized SDN infrastructures, or distributed SDN infrastructures, where several controllers take the control of the different domains. Those SDN controllers are communicating through well define west/eastbound interfaces.

This Self-Healing architecture is composed of a multi-control loop, composed of a control-loop per plane, as shown in Figure 68. , where it can be seen a control-loop per plane with its corresponding sensors and effectors and respective monitoring and recovery blocks. These detectors and recovery blocks must be specific to the plane of the SDN infrastructure. This is due to the type of information retrieved that is specific to the northbound API provided by each type of SDN controller, for instance. The recovery block is also specific to the context or equipment on which the actions are applied.

Each control-loop retrieves symptoms from each plane and correlates them with the symptoms coming from the rest of planes. As the context of this thesis is SDN and NFV combined infrastructures, where the SDN infrastructure ensures the connectivity among the VNFs, this Self-Healing framework can be used to manage the NFV architecture and its related vulnerabilities.

4.2.2.1 Detection actions per plane

This section describes several examples of detection actions per plane, proposed in

Table 9. For instance, in the presence of a network service composed of a set of VNFs, which virtual links are set by a given SDN application that programs the SDN controller, the self-healing system retrieves symptoms from the application plane, the control plane, the data plane and correlates them.

In the presence of a failing element in the underlying physical path of the virtual link connecting two VNFs, the self-healing system can access to the SDN application that set that virtual link. Indeed, it can correlate the state of that virtual link with the state of the physical network elements involved in that virtual link chosen by the SDN controller, which are retrieved from the data and the control planes to pinpoint as responsible that physical link.

Once the diagnosis task clarifies which physical network element is the failing element, the self-healing system sends a as recovery action a forwarding instruction to the SDN application that sets that virtual link to modify the flows to avoid that failing element or root cause.

Table 9. Detection actions at each plane

Plane	Task
Service	Lists the running services Monitors the status of each network service Lists the users using the service Lists the Forwarding Graph of SDN applications for each service
Application	Monitors the state of applications Monitors the allocated path of each application
Control	Monitors the SDN controller(s) Monitors the control links Lists the managed switches by each controller across different domains
Data	Monitors the status of switches Monitors the data links Lists the flows in each switch and their operation mode (standalone or secure) Monitors the status of clients Monitors the status of servers and hosts

4.2.2.2 Recovery actions per plane

We envisage two types of recovery actions for our Self-Healing framework in NFV-SDN based networks: 1) recovery actions that recover the SDN architecture and 2) actions that cooperate with the NFV infrastructure to avoid any service interruption by upgrading, scaling up/down, scaling out/in or migrating the VNFs composing the network services. Table 10 depicts examples of both types of recovery actions on SDN and NFV combined infrastructures.

Specifically, a Self-Healing system can be the intermediate entity between the NFV orchestrator and the VNF Manager to propose dynamic migrations, instantiations or deletions of VNFIs in response to failures on the SDN network.

Table 10. Recovery actions at each plane

Plane	Root cause	Recovery action
Service	End-to-end service misconfiguration	1) Reconfigure end-to-end service 2) Reconfigure involved SDN applications
	Crash of end-to-end service	1) Reinitiate involved SDN applications 2) Restart end-to-end service
Application	Crash of SDN application	1) Restart application 2)Migration to other VM
	Too many requests to SDN application	1) Instantiate other VM to carry out this application 2) Augment memory and CPU of VM
	Misconfiguration of SDN application	Configure SDN application
Control	Control link failure (in-band control)	1) Set standalone mode on affected switches 2) Reconfigure switches to avoid the affected switch
	Control link failure (out-of-band control)	Set standalone mode on affected switches
	Controller failure	Balancing to a secondary controller
Data	Bridge misconfigured	Bridge reconfiguration
	High interference level	Reduction of uplink power on Access Point
	Misconfiguration on client application	Application reconfiguration
	Switch ignores how to reach client	Installation of flow on switch

As example, in an end-to-end service composed of four chained VNFs whose locations may change dynamically across the network, we define the service topology as the subset of the network topology that contains the VNFs used by that service. In the presence of malfunctions that affect the VNFs, they are migrated to other physical locations to avoid the service interruption, what changes the service topology. The SDN controller dynamically reallocates the path to chain the four VNFs.

4.2.2.3 Diagnosis actions per plane

In Figure 68, the diagnosis block is transversal to all the control loops. This is because the diagnosis block is based on a model-based approach, as it was explained in chapter 2. This diagnosis approach is based on a multi-layer dependency graph, automatically generated from the resources detected at each of the planes of the SDN infrastructure. It retrieves the physical resources involved in the data plane and control planes, the logical resources running on those physical resources such as applications and VNFIs, and the virtual resources such as VNFs and their interconnecting virtual links.

4.3 Overall Self-Diagnosis architecture

We describe the self-diagnosis architecture that is based on three modules: a detection module, a self-modeling module, and a RCA module. Self-modeling and RCA modules include a methodology and associated algorithms as well as extensive validation. The detection module is a set of scripts to feed the other modules as the observability techniques are out of the scope of this paper. In Figure 69 we sketch how those modules are related.

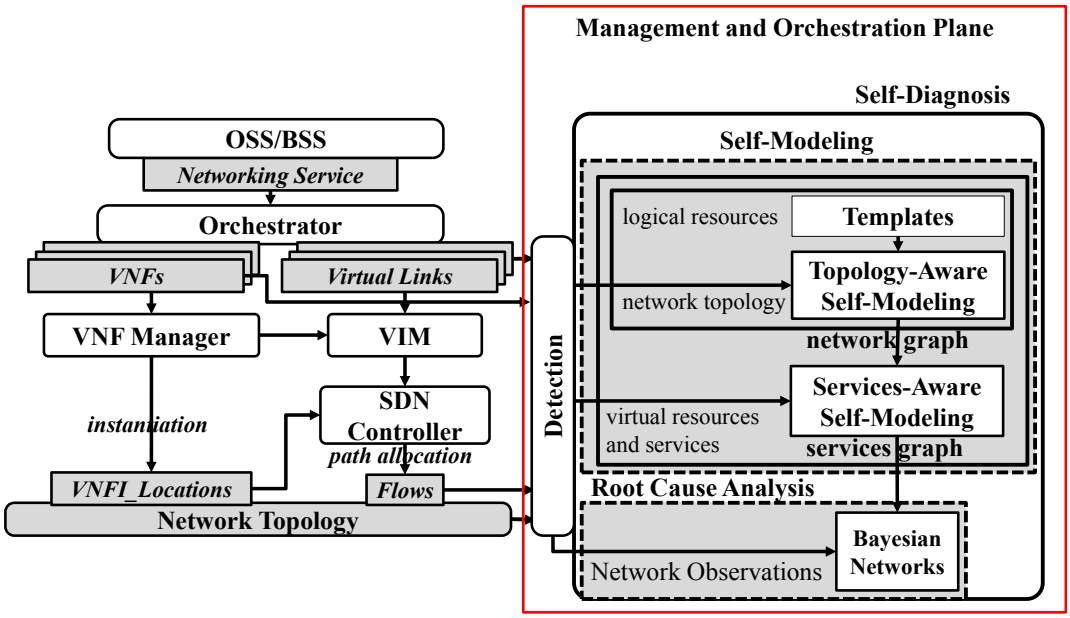


Figure 69. Multi-layer Self-diagnosis architecture

This multi-layer diagnosis system is part of the management and orchestration plane of NFV as it needs to be aware of all the resources coming from these different layers.

Detection module: The detection module builds a view on the networking services and their underlying resources at instant t. It receives the following data: the network topology, including the type of control led by the controller, the logical resources running on networked nodes, the deployed networking services and their respective VNFs and Virtual links, and the flows sent by the SDN controller to establish the physical path to connect the VNFIs.

The detection module keeps the dependency graph updated, by ordering the self-modeling module to regenerate the dependency graph to prevent that the root cause had been calculated based on an outdated model.The self-modeling block tracks changes on the network and service to prevent that the root cause had been calculated based on an outdated model. When the self-modeling module is triggered at tREF, the detector sets as reference the retrieved topology and services present at tREF and the self-modeling module generates the services dependency graph from that reference. From that time on, the detector continuously monitors the topology and services to detect changes and, if so, it orders the self-modeling module to regenerate the services dependency graph and it stops the RCA in case it had already been triggered.

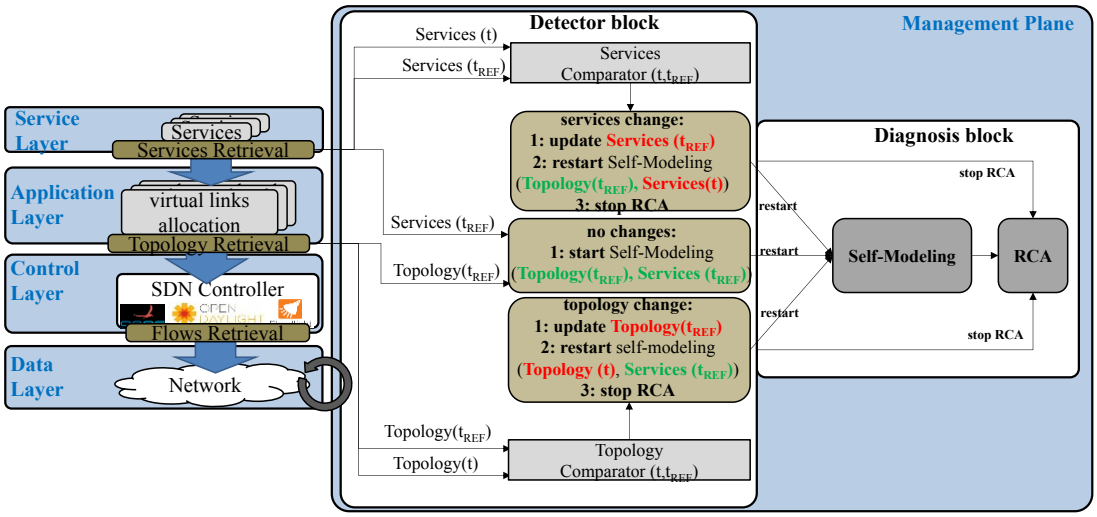


Figure 70. Detection block that updates the network and services dependency graph

Self-Modeling module: The self-modeling module builds the multi-layered dependency graph. It relies on two sub-blocks:

- **Topology-aware self-modeling algorithm:** it generates a first dependency graph from the network topology (physical nodes and links) and logical applications running on the network nodes, hereafter named *network dependency graph.*This self-modeling algorithm classifies the networked elements into a set of fine-grained templates to model their inner dependencies and automatically builds the network dependency graph by assembling these templates.
- **Service-aware self-modeling algorithm:** it generates a second dependency graph, hereafter named *services dependency graph,* by extending the *network dependency graph* with the dependencies of the networking services. The *services dependency graph* contains the dependencies between networking services and virtual resources, and the dependencies of virtual resources from logical and physical resources underneath.

Root Cause Analysis module: The RCA module finds the root cause explaining the service failures by propagating the retrieved network observations through the services dependency graph given as input.

4.4 Self-Modeling module

4.4.1 Types of resources modelled

In this section, we formalize the different types of resources in programmable networks as well as their compositions and properties. We base our model in three types of resources: physical, logical, and virtual running undern a given networking service.

4.4.1.1 Physical resources

The physical resources are those belonging to the infrastructure, nodes and links. There are two types of links, control links and data links.

A link connects two nodes through their interfaces, and the type of node determines to which other nodes it connects to.

A network node is an entity with given capabilities and a given role. We group nodes in three main types: switches, hosts and controllers. Hosts act as source or destination of traffic while switches are intermediate nodes transporting that traffic among host nodes. The controllers, as seen in chapter 2, are those nodes to decide how the traffic is forwarded.

contains: This relationship indicates that the node encompasses internal components. For instance, all the nodes contain a CPU and a set of interfaces.

is_connected_to: This relationship indicates that a given node is connected to a given link. As said before, a link connects two nodes through their interfaces, and the type of node determines to which other nodes it connect to. This means that a control link is always to connect a switch with a controller, whilst a data link can connect two switches, or a switch with a host node.

4.4.1.2 Logical resources

The logical resources are applications running on the network nodes. We grouped them in three types: VNFIs, SDN application controllers, and OpenFlow client applications.

contains: This relationship indicates that the logical resource encompasses internal components. For instance, all the logical resources contain a process (PID) and a configuration related to that process.

is_embedded_in: This relationship indicates that the logical resource is embedded in a given physical resource. For instance, a VNFI is to be embedded in a given host, an OpenFlow client application is to be embedded in an OpenFlow compatible switch, and a SDN controller application is to be embedded inside a controller.

4.4.1.3 Virtual resources

The virtual resources run and are composed of logical resources. We distinguish two types, VNFs and virtual links.

is_connected_to: This relationship indicates that the virtual resource is connected to another virtual resource through its CP (connection points).

is_composed_of: This property is only valid for VNFs and it indicates that the virtual resource is composed of logical resources such as VNFIs).

4.4.1.4 Inter-layer relationships

There are a number of relationships among the different types of network resources that are inherently multi-layer. Those relationships need to be defined in order to enrich the model with inter-layer propagation.

is_supported_by: This property indicates that a given resource relies on another type of resource located in a lower layer. For instance, a virtual resource such a VNF relies on a process (VNFI) that in turn relies on a host where it is embedded.

is_composed_of: This property indicates that a given resource is composed of other resources ubicated at lower layers. For instance, a VNF is composed of VNFIs in the sense that the VNFI is the core process of that VNF. Another example is a virtual link, where it is composed of a set of physical resources (links and nodes).

4.4.1.5 Class hierarchy of network resources

Based on these types of network resources, their relationships and charactersitics defined in previous version we can conceive a hierarchy class grouping all these types of network resources with their relationships among them. This descomposition is similar in shape to the decomposition of network resources done by Hounkonnou for IMS networks in (Hounkonnou,2013). Nevertheless, this decomposition is to formalize all these resources for programmable networks, and beyond that, it reaches a subcomponent granularity, which, as it will be shown later, will be paramount for an accurate diagnosis.

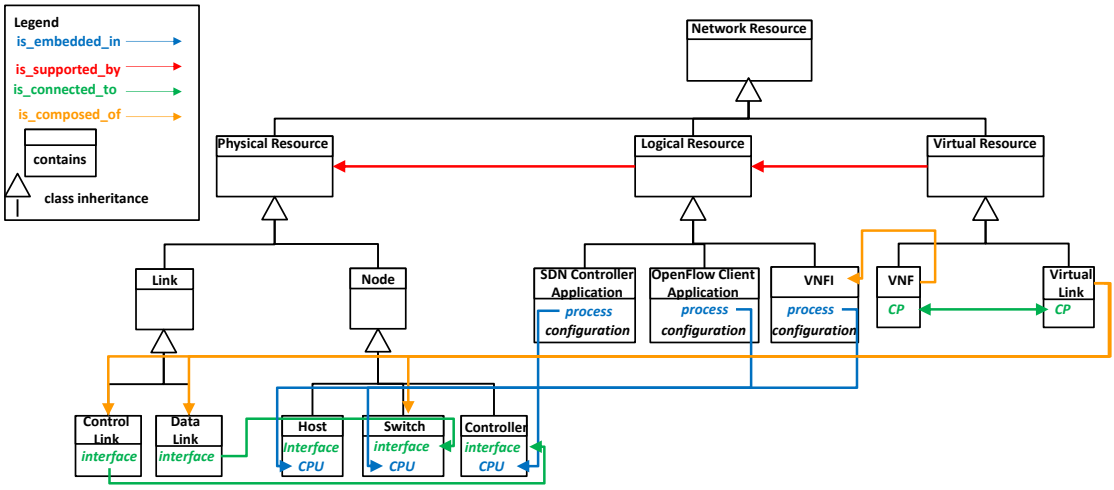


Figure 71. Hierarchy class of network resources in programmable networks

4.4.2 Problem formalization

The objective of this thesis is to diagnose networking service in SDN and NFV combined infrastructures. The aim is to define a self-modeling methodology as support for diagnosis capable of generating the dependency graph of a given networking service, encompassing its dependencies among the aforementioned network resources.

A networking service is composed by a set of VNFs interconnected by a set of virtual links. We formalize the networking services to be modelled as it follows:

- There are *N* networking services deployed over the network infrastructure. Each *networking service*$_i$ $\forall\, i = [1, N]$ is composed of M_i virtual links connecting N_i VNFs.
- The network infrastructure is composed of a set of P links and Q nodes.
- The nodes are a set of switches $S = \{S_1, \dots, S_X\}$, a set of hosts $H = \{H_1, \dots, H_Y\}$, and a set of controllers $C = \{C_1, \dots, C_Z\}$ in such a way that $X + Y + Z = Q$, the total number of nodes in the network topology.
- The nodes are interconnected by a set of control links $CL = \{CL_1, \dots, CL_r\}$ and a set of data links $DL = \{DL_1, \dots, DL_q\}$ in such a way that $r + q = P$, the number of links in the network topology. In in-band infrastructures $r = 1$, whilst in out-of-band infrastructures $r = X$. Control links are only connectable switches with controllers, and data links are connectable to hosts, switches, or both.
- Each virtual link $VL_{m,n}^{(k)} \forall\, k = [1, M_i]$ connects two any VNFs, VNF_m and VNF_n, through their CP (connection points) CP_m and CP_n.
- VNFIs (VNF instances) are logical resources embedded in hosts and send and receive traffic through their hosts' NICs
- A physical path $\rho_{m,n}^{(k)}$ interconnects two any hosts H_m and H_n.
- The SDN controller allocates each *physical path* $\rho_{m,n}^{(k)} \forall\, k = [1, M_i]$ through its ${n_f}^{(k)}$ control links by installing ${n_f}^{(k)}$ *flows* on ${n_f}^{(k)}$ intermediate switches traversing ${n_l}^{(k)}$ data links. This can be seen in Figure 72, for ${n_f}^{(k)}$=4 flows installed on 4 switches and ${n_l}^{(k)}$=5 data links.

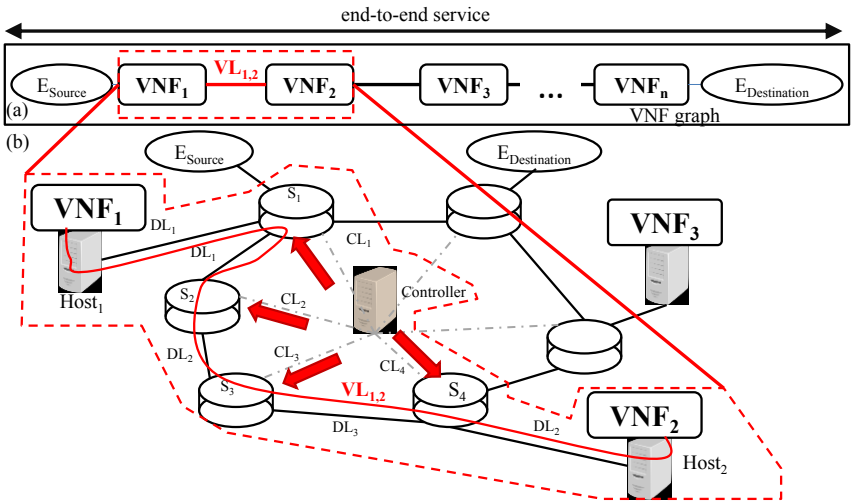


Figure 72. (a) End-to-end service, (b) Underlying network topology of a virtual link

In Figure 72 presents the underlying network topology, where the SDN controller installs flows in order to establish that virtual link interconnecting two VNFs.

We use the 'divide-and-conquer' principle to decompose a given networking service into simpler to model segments (blue, green, and red). This methodological approach is similar to the layered model proposed by (Steinder & Sethi, 2002), but adapted to a SDN and NFV context. The virtual links supporting VNF-to-VNF and PNF-to-VNF communications are established by the SDN controller, which sends flows to a set of switches of the network topology. Figure 73 shows the segments composing a networking service, external and internal virtual links.

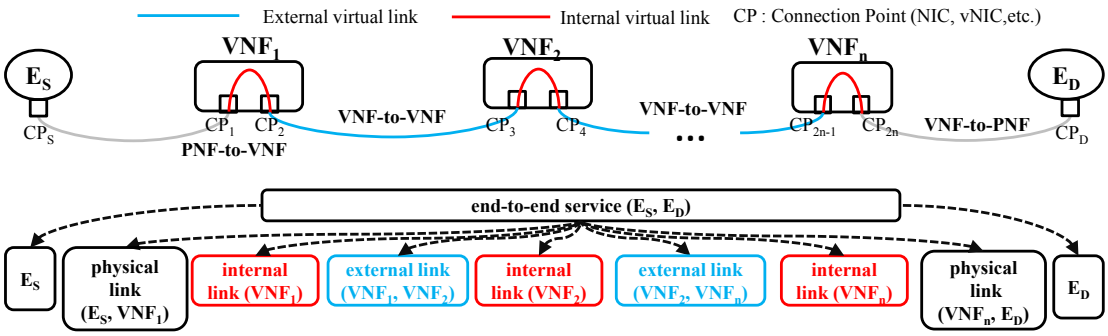


Figure 73. Networking service decomposition in different network segments

We conceive in this thesis a multi-layer diagnosis methodology based on a multi-layered model that includes the supervised resources within the following layers: 1) physical, 2) logical, virtual and 3) networking services. Examples of the supervised resources are given in Table 11. Figure 74 zooms on the components involved in an external virtual link connecting two VNFs ($VNFI_m$ and $VNFI_n$), which are embedded in $host_m$ and $host_n$. Those components are located at virtual, logical, and physical layers.

Table 11. Types of resources considered per layer

Layer	*Resources*
physical	links, switches, hosts, controllers, ports, NICs, CPU
logical	Flows, controller application, OpenFlow application, VNFIs
virtual	virtual links, VNFs
service	VPN, NAT, firewall, streaming, etc.

These physical, logical, and virtual resources can belong to the control plane (in white) or the data plane (in yellow). All the switches of the data plane in this physical path will receive flows from the SDN controller with the following format: {"in_port":$port_x$,"out_port":$port_y$,"src":"CP_i","dst":"CP_j"}.

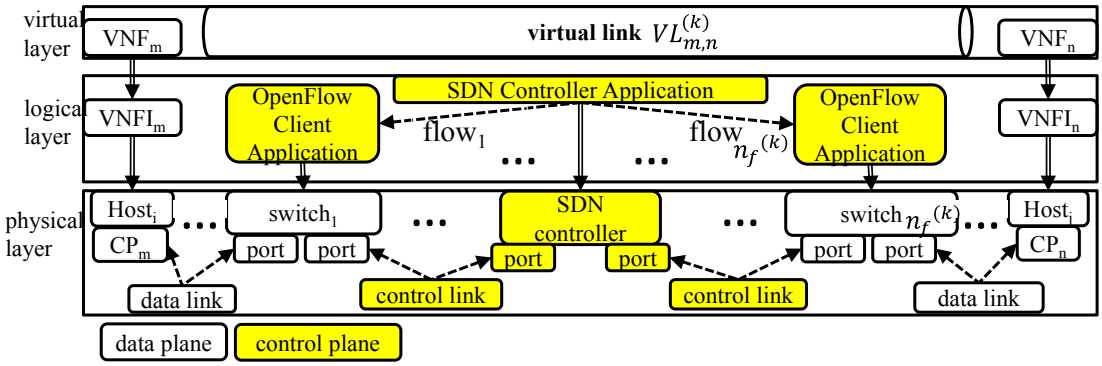


Figure 74. Zoom on the underlying resources involved in an external virtual link between VNFs

4.4.3 Description of resources dependencies through templates

We define a network element as any type of network nodes and links. We propose a template for each network element, so a template for network nodes and another for links. These templates describe the relationships in terms of dependencies between the components inside each network element. The template of a network node is composed of a physical layer and a logical layer, following the TMF Information framework specifications.

- The physical layer encompasses the state of physical resources such as CPU, network cards, and connection ports.
- The logical layer encompasses logical resources such as VNFs or applications running inside each node.

The templates are predefined, but are extensible and adaptable. Each type of network node discovered in the network topology such as controllers, switches, and hosts are characterized by a different set by of dependencies and components so they will require a different template and hence a different dependency graph.

On the templates, those VNFs running on hosts and OpenFlow client applications running on switches are composed of a life-cycle based on three states: instantiated, configured, and active, VNFs have three states: instantiated, configured, and active, following the ETSI NFV GS specification in (ETSI NFV, 2014).

The finer granularity of these templates allows a detailed diagnosis, down to the sub-component level, which is to the best of ou knowledge was not handled in the state of the art and that complements our work in (Sanchez et al, 2015). These templates adapt in accordance with the network topology information.

4.4.3.1 Dependency graph of a Host

The host's dependency graph is predefined, but automatically extensible with the information discovered from each given host in the topology at instant t.

The number of VNFIs, ports and NICs of those dependency graphs are extendable with the VNFIs embedded in each host and the connections found in the topology.

At a physical level, the number of NICs (network interface cards) depends on the number of interfaces discovered in that host, retrieved from the network topology.

At a logical level, hosts may run one or several applications (e.g. Video streaming application) or one or several VNFIs in an NFV environment. If we focus on VNFIs as software running on them, this VNFI must be instantiated, must be configured (the type of VNF and its associated configuration), and must be active (the VNF must be physically reachable to be chained to compose a given networking service). In particular in these templates, the

number of VNFIs added to each host's template corresponds to the number of VNFs detected running on each host, information which is automatically discovered and detected as it will be seen in next sections.

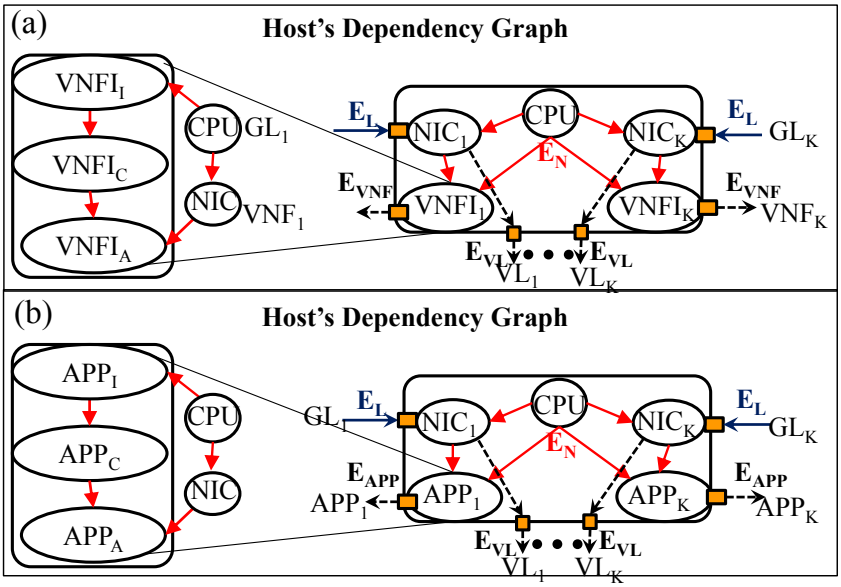


Figure 75. Dependency graph of a host: (a) embedding K Applications, (b) embedding K VNFIs

There are incoming E_L edges coming from the dependency graphs GL_K of the K links connected to that host in the network topology.

As the host can embed different VNFIs, the E_{VNF} outgoing edges represent the impact over the VNFs relying on these instances.

In addition, as the NIC (Network Interface Card) of the host will be involved in a series of virtual links to chain VNFs embedding in it, there are also outgoing edges E_{VL} to represent the impact of faults in the host's NIC on the virtual links established.

VNFIs have three states: instantiated ($VNFI_I$), configured ($VNFI_C$), and active ($VNFI_A$), following the ETSI NFV GS specification.

The K VNFIs embedded in hosts are given by the *VNFI_Locations* variable, which is used by the topology-aware self-modeling algorithm to update the dependency graph of hosts with their corresponding embedded VNFIs (Figure 75).

4.4.3.2 Dependency graph of a SDN Controller

The controller's dependency graph is predefined but the number of ports to communicate with the switches is extensible with the number of ports discovered on the controller. This number of ports depends on the type of control led by the controller (in-band or out-of-band). In in-band control, only one port is added to the graph to connect the controller to the master switch. In out-of-band control, the number of ports added to the graph corresponds to the number of switches managed by the controller found in the topology at instant t.

At a logical level, the controller runs an SDN application (e.g. OpenDaylight, Floodlight, NOX, POX, etc.) which sends the southbound commands to the switches. Ths SDN application can in turn be decomposed in many other inner applications such as the topology manager or similar, bu in those templates only a SDN application is considered. This application must be instantiated (the process is running), must be configured (the type of control carried by the controller and other more advanced commands dependable from the type of controller application), and must be active (the controller must be physically reachable by the switches to send/receive the OpenFlow commands).

There are incoming E_L edges from the n control links, to connect the dependency graph of the controller to the dependency graphs of those n control links seen in the network topology GL_n.

As the SDN controller is allocating at run-time a series of virtual links to chain VNFs composing networking services, there are also outgoing edges E_{VL} to represent the impact of faults in the controller ports on the virtual links established.

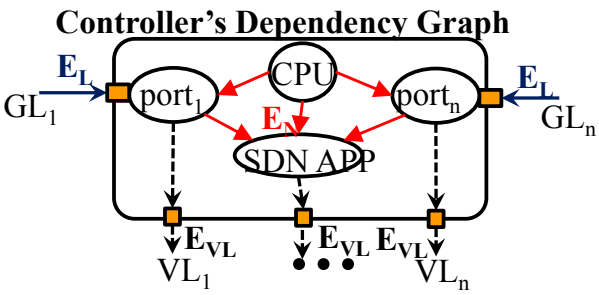


Figure 76. Dependency graph of a Controller

4.4.3.3 Dependency graph of a Switch

The switch's dependency graph is predefined, but the number of ports is extensible with the number of ports discovered per switch.

At a physical level, it contains the number of ports per each switch discovered in the network topology at instant t. The port connecting to the SDN controller is also included.

At a logical level, each switch runs an OpenFlow client application to connect to the SDN controller and interpret the southbound commands received. This application must be launched (the process is running), must be configured (the corresponding controller's IP address must be correctly set), and must be active (the switch must be physically reachable by the controller to send the OpenFlow commands).

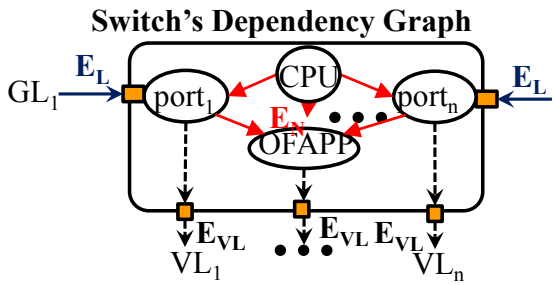


Figure 77. Dependency graph of a switch

There are incoming E_L edges from the control and data links, to connect the dependency graph of the switch to the dependency graphs of those links seen in the network topology.

As the switches compose virtual links to chain VNFs composing networking services, there are also outgoing edges E_{VL} to represent the impact of faults in those switches ports on the virtual links established.

4.4.3.4 Dependency graph of a Link

The dependency graph GL of a network link is simpler and it is composed of the physical layer and one single vertex. Each type of link found in the topology will have this dependency graph GL. This dependency graph has two outgoing edges E_L that will connect to two nodes dependency graphs GN_i.

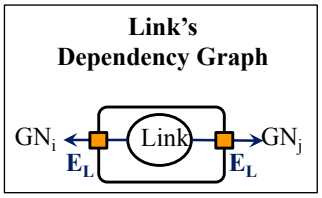


Figure 78. Dependency graph of a link

4.4.4 Generation of the network dependency graph

The network dependency graph is generated by the topology-aware self-modeling approach, described hereafter. The network dependency graph is built from the network topology, which contains the dependencies among the network nodes at a physical level, but also those dependencies of the logical resources running on the switches and the SDN controller and those logical resources running on the network nodes.

The network dependency graph in four steps, described here:

- Step 1: Network topology interpreter
- Step 2: The dependency graph instantiation algorithm
- Step 3: The topological sorting algorithm
- Step 4: The E_L edge addition algorithm

Step 1: Network topology interpreter algorithm

The network topology interpreter retrieves the network topology seen by the SDN controller and generates two machine-readable descriptors that encompass the network nodes and links classified in the following types: controllers, switches, hosts, control links, access links and inter switch links.

This algorithm extracts the network topology from the northbound interface of the SDN controller. The network topology has a JSON (JavaScript Object Notation) data structure format. As example, we show the JSON data structure of a topology composed of one switch connected to the SDN controller. This format depends on the specific controller's northbound API. OpenDaylight and a Floodlight controller provide two different data structures, differing in the number and type of fields and the field names (Figure 79). For example, the MAC (Media Access Control) field can be whether dataLayerAddress or mac and the ip address can be named networkAdress or ipv4, according to the SDN controller used.

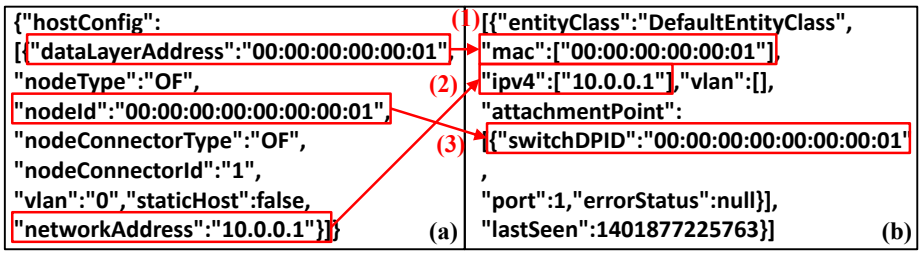


Figure 79. JSON data structures provided by: (a) OpenDaylight, (b) Floodlight

Based on the two data structures provided by two different controllers, this topology interpreter algorithm analyzes and extracts the following parameters per network node shown in Table 12. However, these parameters do not include the control links connecting the controller to the switches. Indeed, the network topology seen by the SDN controller only considers the elements in the data plane. We need to extend these parameters with all these control links to include them in the network dependency graph.

Table 12. Topology parameters analyzed by the topology interpreter algorithm

Network element	*Parameters*	
Host	*MAC address* *IP Address*	

	Access switch	*DPID*	*port*
Switch	*DPID*		
	ports		
Data link	*source switch*	*DPID*	*port*
	destination switch	*DPID*	*port*

As a result, the topology interpreter algorithm provides as output with Nt and Lt:

Nt is the topology descriptor containing the network elements, classified in controllers, control links, switches, data links, and hosts. It contains the links connected to the nodes. Lt is the link descriptor containing the nodes attached to each link.

We describe here two common topologies analyzed along the thesis, the tree topology and the linear topology. The tree topology, Tree(D,F) is shown in (a), and it is a hierarchical topology of D+1 layers, D layers composed of switches and one layer composed of hosts. Each switch in each layer is splitted in F branches. The linear topology, Linear(N), is composed of N host-switch pairs connected by inter switch links.

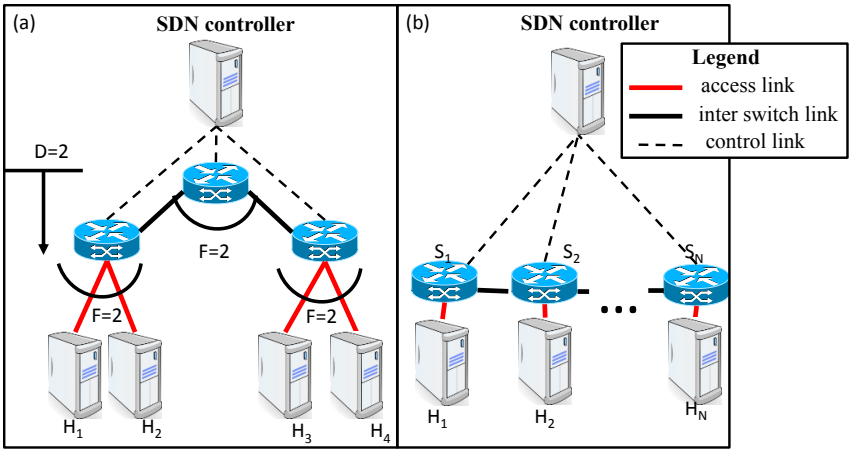


Figure 80. Definition of network topologies: (a) Tree(D,F), (b) Linear(N)

Table 13. Output format of Nt and Lt descriptors for Linear (N=5) and Tree(D=2,F=2) topologies

Topology	*Parameter*	*Nt*	*Lt*
Linear (N=5)	*controllers*	*c0*	*[Ms01, Ms02, Ms03, Ms04, Ms05]*
	control links	*[CL1, CL2, CL3, CL4, CL5]*	*[c0-Ms01, c0-Ms02, c0-Ms03, c0-Ms04, c0-Ms05]*
	switches	*[Ms01, Ms02, Ms03, Ms04, Ms05]*	*[(Ms02, h1), (Ms01, Ms03, h2), (Ms02, Ms04, h3), (Ms03, Ms05, h4), (Ms04, h5)]*
	access links	*[AL1, AL2, AL3, AL4, AL5]*	*[Ms01-h1, Ms02-h2, Ms03-h3, Ms04-h4, Ms05-h5]*
	inter switches links	*[IL1, IL2, IL3, IL4]*	*[Ms01-Ms02, Ms02-Ms03, Ms03-Ms04, Ms04-Ms05]*
	hosts	*[h1, h2, h3, h4, h5]*	*[Ms01, Ms02, Ms03, Ms04, Ms05]*
Tree (D=2,F=2)	*controllers*	*c0*	*[cMs01, Ms02, Ms03]*
	control links	*[CL1, CL2, CL3]*	*[c0-Ms01, c0-Ms02, c0-Ms03]*
	switches	*[Ms01, Ms02, Ms03]*	*[(Ms02, Ms03), (Ms01, h1, h2), (Ms01, h3, h4)]*
	access links	*[AL1, AL2, AL3, AL4]*	*[Ms02-h1, Ms02-h2, Ms03-h3, Ms03-h4]*
	inter switches links	*[IL1, IL2]*	*[Ms01-Ms02, Ms01-Ms03]*
	hosts	*[h1, h2, h3, h4]*	*[Ms02, Ms02, Ms03, Ms03]*

We provide in Table 13 the format required for these descriptors Nt and Lt for a linear (L=5) and tree (D=2, F=2) topology. This algorithm generates these descriptors Nt and Lt from the network topology, and includes in these descriptors the control links (in red).

Step 2: The dependency subgraph instantiation algorithm

This algorithm receives as input the network descriptor. It provides as output the dependency subgraphs of the discovered nodes and links in the network topology at instant t. It follows this methodology for each network element found in the network descriptor:

1) Identifies the type of network element (node or link)
2) Instantiates its corresponding template according to the type of element (GN for nodes or GL for links)
3) Instantiates the dependency subgraph of that network element
4) Appends the instantiated dependency graphs to the *network dependency graph*

Table 14. Dependency subgraph instantiation algorithm

Algorithm: Dependency subgraph instantiation algorithm
IN: Network Descriptor N_t
IN: Templates $\{T_{HOST}, T_{SWITCH}, T_{CONTROLLER}, T_{LINK}\}$
OUT: Network Dependency Graph $NDG(V_{UNSORTED}, E_N)$
for each element in the network descriptor
  inspection of type of element
  if element is of type link
    $TL_i \leftarrow$ instantiation of link template $\{T_{LINK}\}$
    $GL_i \leftarrow$ extract dependency sub graph of template TL_i
    NG$\leftarrow$ append GL_n to global dependency graph
  else
    $TN_n \leftarrow$ instantiation of node template $\{T_{HOST}, T_{SWITCH}, T_{CONTROLLER}\}$
    $GN_n \leftarrow$ extract dependency sub graph of template TN_i
    NG$\leftarrow$ append GN_n to global dependency graph
  end if
end for

The dependency subgraph of nodes is GN_i (V_N, E_N) are different subgraphs for switches, controllers and hosts and the dependency subgraphs of links is GL_i (V_L, E_L) and are the same for control links and data links. These edges E_N (in red) depict the dependencies among components inside the dependency subgraph of a node G_N.

The network dependency graph is composed of these instantiated dependency subgraphs. However, the vertices of the network dependency graph are not topologically sorted yet. We call this set of vertices as $V_{UNSORTED}$.

Step 3: The topological sorting algorithm

This algorithm sorts topologically the vertices of the network dependency graph. It receives as input the network dependency graph with non-topologically ordered vertices ($V_{UNSORTED}$) and it provides as output the same network dependency graph but topologically sorted (V_{SORTED}).
As an example, we present a non-topologically sorted network dependency graph in Figure 81, where the topological order is not respected, as all the instantiated dependency subgraphs contain repeated vertex indexes (e.g. value '1' is repeated). The topological sorting algorithm reorders the vertices to solve this issue and respect the topological order, i.e. parents are always before children. The sorting scheme is shown in Figure 81 (b), where it can be seen that the edges outgoing E_L edges to connect the dependency subgraphs of nodes are not added yet.

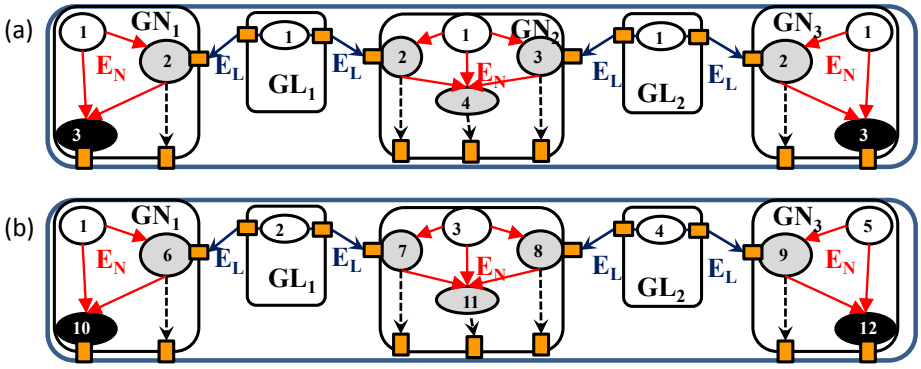


Figure 81. (a) Non–topologically sorted network dependency graph, and (b) topologically sorted network dependency graph

Table 15. Topological Sorting algorithm

Algorithm: Topological Sorting
IN: Network Dependency Graph NG($V_{UNSORTED}$,E_{INTRA})
OUT: Network Dependency Graph NG(V_{SORTED},E_{INTRA})
for each instantiated dependency graph appended to NG
for each layer in template
obtain vertices of appended graph at current layer
sort its vertices topologically
end for
end for

Step 4: The E_L edge addition algorithm

This algorithm adds the dependencies among instantiated dependency subgraphs. It receives as input the link descriptor and the topologically sorted network dependency graph. E_L depicts the dependencies among dependency subgraphs of nodes. An E_L edge connects the dependency subgraph of m-th node GN_m with the dependency subgraph of n-th node GN_n. This edge consists of the following pair: $E_L^{(k)} = (GN_m{}^{(k)}, GN_n{}^{(k)})$. These E_L edges represent the impact of faults in links on nodes interfaces (ports and NICs).

The network dependency graph is built by assembling the dependency graphs GN and GL belonging to the P links and Q nodes found in the network topology, composed of P links and Q nodes.

$$NDG = \bigcup_{k=1}^{P} GL^{(k)}(V_L^{(k)}, E_L^{(k)}) \cup \bigcup_{k=1}^{Q} GN^{(k)}(V_N^{(k)}, E_N^{(k)})$$

Table 16. E_L Edge addition algorithm

E_L Edge addition algorithm
IN: Link Descriptor, Network Dependency Graph G(V_{SORTED},E_N)
OUT: Network Dependency Graph G(V_{SORTED},E_N,E_L)
for each link in Link Descriptor
extract end points attached to link
for each end point in link
E_L←add edge from GL_m[link,link] to GN_n[node,card]
end for
end for

Figure 82 represents an example of the network dependency graph built by the topology-aware self-modeling approach. This network dependency graph is composed by the different subgraphs with their inner E_N edges (in red), and it is assembling these subgraphs through E_L edges (in blue). In both examples, the control links and the SDN controller are not shown for the sake of clarity. Figure 83 shows the generic dependency graph corresponding to an end-to-end path between two hosts, where the control plane's elements have been omitted also for simplicity. It is important to recall the fact that the network dependency graph is composed of the whole network topology not only of end-to-end paths as shown in these two images.

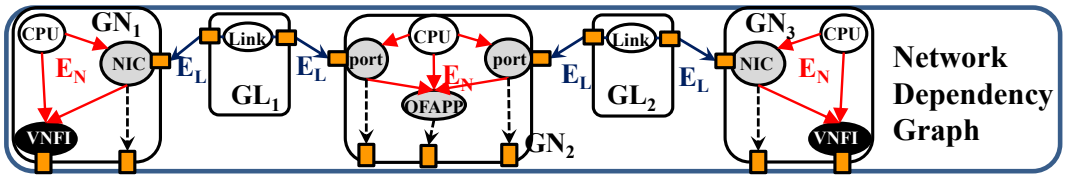


Figure 82. Example of Network dependency graph (Q=3 nodes and P=2 links)

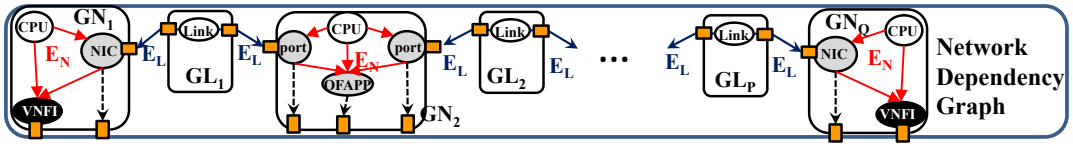


Figure 83. Generic Network dependency graph (Q nodes and P links)

4.4.5 Generation of the service dependency graph

The service dependency graph is generated by the service-aware self-modeling approach, described hereafter. The service dependency graph is an extension of the network dependency graph. The services dependency graph is built in two steps, as described herefter:

- Step 1: Virtual resources dependency graph generation
- Step 2: Service dependency graph generation

Step 1: Virtual resources dependency graph generation

This algorithm creates an auxiliary graph, called virtual resources dependency graph (VRG), containing the discovered networking services and their virtual resources (VNF and virtual links).

Table 17. Virtual resources dependency graph generation algorithm

Virtual resources dependency graph generation algorithm
Input: NSR(Network Service record)
Output: VRG (Virtual Resources Dependency Graph)
nsr←NSR[i] ∀i={1,…,N} //retrieval of NSR of that network service
V(VRG)← V(VRG) U nsr:id
VL←nsr:vlr[j]* ∀j={1,…, M_i} //retrieval of virtual links
V(VRG)← V(VRG) U VL //adds virtual link vertex to virtual layer
E(VRG)← E (VRG) U E_S:=(orig:[VL:id*],dest:[VL:parent_ns*]) // adds E_S edge
VNFR←nsr:vnfr[k]* ∀k={1,…,N_i} //retrieval of VNFs
V(VRG)← V(VRG) U VNF //adds VNF vertex to virtual layer
E(VRG)← E (VRG) U E_S:=(orig:[VNF:id*],dest:[VNF:parent_ns*])//adds E_S edge

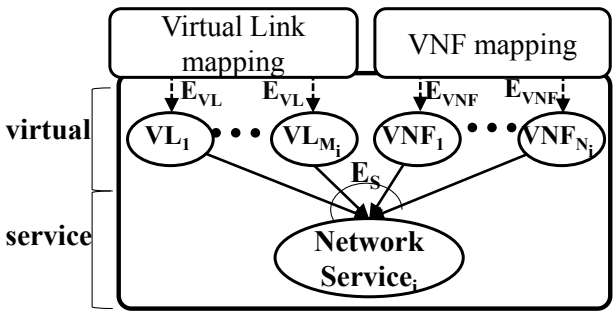


Figure 84. Virtual Resources dependency Graph generation

The *VRG* is composed of a virtual layer and a service layer (Figure 84). For each discovered networking service, both layers are filled as follows:

Service layer: the algorithm adds a *networking service*$_i$ vertex to the *VRG.*

Virtual layer: the algorithm adds M_i virtual links vertices and N_i VNFs vertices. It then adds M_i+N_i E_S edges (in black in Figure 84) from those virtual resources vertices to the *networking service*$_i$ vertex. E_S edges represent the impact of faults in virtual resources on that networking service *networking service*$_i$. If the SDN controller is enabled with the SFC (Service Function Chaining) module such as in OpenDaylight, the VRG could be directly generated from the VNF FG information.

Step 2: Service dependency graph generation

This algorithm connects the *network dependency graph* to the *VRG* and builds the *service dependency graph*. It maps each network service withs its corresponding underlying physical and logical resources.

For each networking service, two mappings are done:

-**VNF mapping:** The VNFI vertices of the hosts in the *network dependency graph* are connected to the VNFs vertices in the *VRG* through edges E_{VNF} (in dash black in Figure 85). E_{VNF} edges represent the impact of faults in the VNFI embedded in hosts on VNFs composing a networking service.

-**Virtual Links mapping:** The physical network resources involved in each virtual link (hosts NICs, switches ports, and OpenFlow client applications inside switches) are connected to their respective virtual links vertices through edges E_{VL} (in dash black in Figure 85). E_{VL} edges represent the impact of faults in physical and logical resources on a virtual link. These network resources are extracted from the *flows*, defined in section IV.

Service dependency graph generation algorithm

Input: NDG, VRG, Flows, VNFI_Locations, NSR(Network Service Record)
Output: SDG (Service Dependency Graph)

```
SDG←NDG ∪ VRG  //initialization
nsr←NSR[i]   ∀i={1,…,N}        //retrieval of networking services
VL←nsr:vlr[j]* ∀j={1,…, Mi}//retrieval of virtual links
VNF←nsr:vnfr[k]* ∀k={1,…,Ni}  //retrieval of VNFs
flowsPerVL←Flows[VL] // retrieval of flows composing virtual links
flow←flowsPerVL[l] ∀l={1,…, ni(j)}
[switch,ports,OFAPP] ←ExtractSwitchInfo(flow) //extracts switch storing flow
[hosts,NICs] ←ExtractHostsInfo(flow)  //extracts hosts connected by that flow
VNFID←VNFI_Locations[hosts] //finds VNFID of VNFI embedded in host
E (SDG)← E(SDG) ∪ EVNF:=(orig:[VNFID],dest:[VNF:id*])  //adds edge
E(SDG)← E(SDG) ∪ EVL:= (orig:[hosts:NIC],dest:[VL:id*])  //adds edge
E(SDG)← E(SDG) ∪ EVL:= (orig:[switch:ports],dest:[VL:id*])  //adds edge
E(SDG)← E(SDG) ∪ EVL:= (orig:[switch:OFAPP],dest:[VL:id*]) //adds edge
```

Figure 85 shows an example of services dependency graph sent to the RCA. This services dependency graph belongs to one networking service (N=1) composed of one virtual link (M_i=1) connecting two VNFs (N_i =2) deployed over a physical path. The *services graph* includes the *network graph* shown in Figure 82.

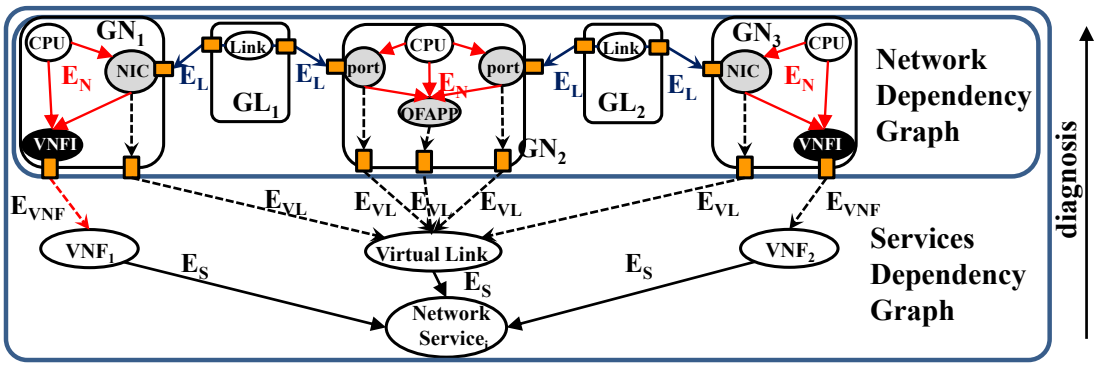


Figure 85. Services dependency graph of one network service

This example is a simplied service dependency graph for one networking service, only composed of a physical path between two VNFIs hosted in two hosts, but it does not show the SDN controller and the control links. The service-aware self-modeling methodology allows to compose this service dependency graph for as many services as found in the M_k virtual links connecting N_k VNFs. This service dependency graph can be extended to include all these services.

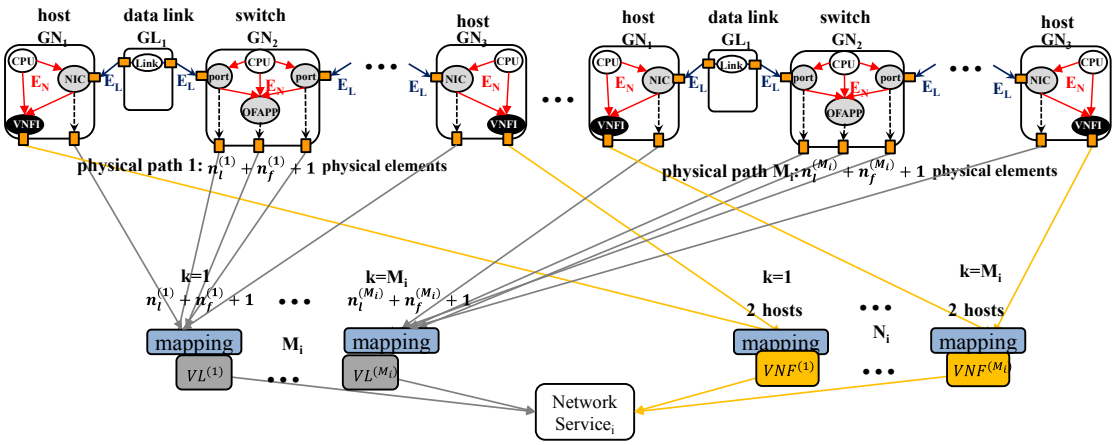


Figure 86. Generic Services dependency graph of N network services

4.5 Exploitation of the service dependency graph for Root Cause Analysis

The root cause analysis approach is based on Bayesian Networks. It receives as input the service dependency graph generated by the self-modeling approach, which includes the network dependency graph. The root cause analysis block reasons over the service dependency graph built from the networking services and their underlying network resources. The service dependency graph depicts how failures in a given networking service are originated in the underlying infrastructure and are propagated.

Each vertex on the service dependency graph represents any subcomponent whithin the physical, logical, and virtual network resources of the SDN infrastructure. For each component within any network resource, p represents the probability of failure for that component. Each vertex is a random variable with two states (up, down).

The CPT for a given subcomponent/resource is given in Table 18 and it is based on the following properties:

- All the network subcomponents/resources can always fail by themselves, with an a priori probability p, regardless of the fact the parent subcomponents/resources function as expected.
- One fault in one network subcomponents/resources immediately propagates to the children network subcomponents/resources and eventually to those networking services depending on that network resource.

This choice is justified by the self-diagnosis approach led by (Hounkonnou,2013) to describe the dependencies among IMS resources and our self-diagnosis framework for SDN (Sanchez et al, 2015) and (Sanchez et al, 2014).

Table 18. CPT of a generic component Y in a network resource

CPT(Y)	Pr(Y='down')	Pr (Y='up')
at least one parent 'down'	1	0
if all parents of Y 'up'	p	1- p

This value p could be different according to the type of subcomponent and resource. For instance p can be different for logical and physical resources and subcomponents.

Physical resources: In CPU vertices, this a priori value p can depend on the measured CPU load at instant t. In SDN controller ports, p can depend on the number of incoming ports of the SDN controller, the more ports the

controller has, the higher probability of failure p, as the SDN controller has more physical connections from the switches and this can induce to congestions and and a high number of flows per second.

Table 19Table 21 shows the CPT of a CPU component inside a network node and Table 20 shows the CPT of a port within a switch node.

Table 19. CPTs for the CPU component

CPT(CPU_C)	P(CPU_C=down)	P(CPU_C=up)
	p=0.1	1-p=0.9

Table 20. CPTs for the switch port component

CPT(port)	P(port=down)	P(port=up)
CPU_C=down & link=down	1	0
CPU_C=up & link=down	1	0
CPU_C=down & link=up	1	0
CPU_C=up & link=up	p=0.1	1-p=0.9

Logical resources: Applications or processes can be better designed (lower value of p), or worse designed (higher value of p). The p value in those applications with the same software and same version should have the same probability, to model bugs in those versions. p can depend on the amount of total memory or CPU consumed by each application, indicating possible degradations. In VNFIs, p can depend on how centralized this instance is at a given time, being p higher. Table 21 shows the CPT of a process running on a network node and Table 22 shows the CPT of the configuration related to that process.

Table 21. CPTs for the process components

CPT(process)	P(process=down)	P(process)
CPU_C=down	1	0
CPU_C =up	p=0.1	1-p=0.9

Table 22. CPTs for the configuration of a process

CPT(configuration)	P(configuration=wrong)	P(configuration)
CPU_C=down & process=down	1	0
CPU_C=down & process=up	1	0
CPU_C=up & process=down	1	0
CPU_C=up & process=up	p=0.1	1-p=0.9

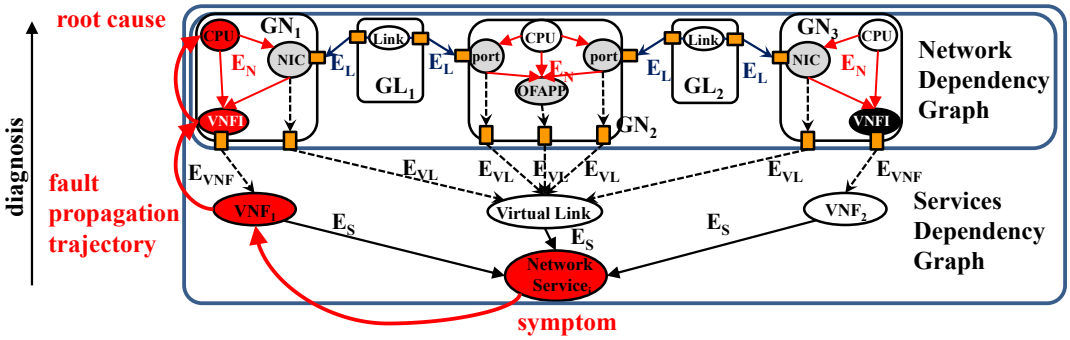


Figure 87. Root Cause Analysis over the generated services dependency graph of one network service

The BN is composed of the physical, logical, and virtual network resources ans inner subcomponents characterized by binary random variables, which indicate their state (‘*down*’ or ‘*up*’) and the edges represent the dependencies among network components.

The root cause analysis works as follows: the RCA starts propagating evidence/symptoms on the network service vertex through the graph based on the CPTs until it reaches the root vertices, yielding a posteriori probability for each vertex. The *services dependency graph* modelling allows the RCA to diagnose dynamic networking services on dynamic network topologies.

Figure 87 shows an example of root cause analyisis over the generated service dependency graph. The RCA engine receives an alarm indicating that the networking service is under failure and introduces this evidence on the graph (red vertex, network service). This evidence is propagated through the graph by updating the aposteriori probabilities by the Bayes rule. In this example, one VNF composing the sevice is failing due to a failure on a physical resource (CPU) which impacts the logical resource VNFI.

4.6 Conclusion

This chapter considers solving two major problems towards self-diagnosis and resilient networks in the context of SDN and NFV: in such context it is needed to define a template or a model that describes the managed elements including physical, virtual infrastructure and other inner details such network cards, or CPU. To fill this gap, we define a template with finer granularity describing the essential managed elements within SDN and NFV. Furthermore, we propose a topology-aware self-modeling diagnosis that builds automatically at runtime the diagnosis model (dependency graph), which answers the challenges of updating the diagnosis model to identify the root causes. Our approach is suitable to any network topology and to any control type in SDN. In addition, it is independent from the controller implementation (e.g. Floodlight or OpenDaylight).

This chapter also extends this topology-aware self-diagnosis towards a multi-layer service-aware self-diagnosis framework capable of diagnosing faults in programmable networks with SDN and NFV, while taking into account the networking service, virtual, logical, and physical layers.

This service-aware self-diagnosis framework diagnoses and correlates two additional layers, virtual and services layer, while considering their dynamic dependencies with the underlying logical and physical resources. The core of the self-diagnosis framework is a self-modeling module that relies on two algorithms to generate on-the-fly and update the diagnosis model from the network topology, logical resources and networking services with a set of adaptable templates.

Chapter 5 Results and Evaluation

5.1 Introduction

In this chapter we present the results conerning our multi-layer self-modeling based diagnosis approach. We first evaluate the topology-aware self-diagnosis approach, which diagnoses based on the network topology and logical resources running on top, and then we focus on evaluation of the service-aware self-modeling approach.

For each of these cases, we evaluate three essential apsects: the generatoin of the model, the explitation of this model to find the root cause, and the performance of the diagnosis, which comprises both the self-modeling methodology and root cause analysis algorihtm.

The last section, presents an implementation of the topology-aware self-diagnosis concept of chapter 4 that enables a self-healing system for streaming applications running over softwarized networks.

5.2 Topology-Aware Self-Diagnosis Evaluation

In this section we evaluate how the network dependency graph is generated and then exploited to find the root cause.

5.2.1 Generation of the network dependency graph

We test our self-modeling based diagnosis in a centralized SDN architecture based on a Floodlight controller. This module runs over the controller for two reasons: (1) to have a global view of the network, and (2) to keep the diagnosis framework independent from any specific southbound protocol. The network topology is obtained via the northbound interface (REST API) trough passive monitoring, to avoid introducing traffic overhead like ping tool. We use Mininet to simulate the SDN network.

First, we prove that our self-modeling algorithm can interpret both the network topology and the control type of SDN (out-of-band and in-band). Next, we study the scalability of this algorithm and finally we validate the diagnosis results and their variation under changing network conditions.

We test the model generation of a linear topology with two switches and a controller with two hosts connected with out-of-band control Figure 88 and in-band control Figure 89.

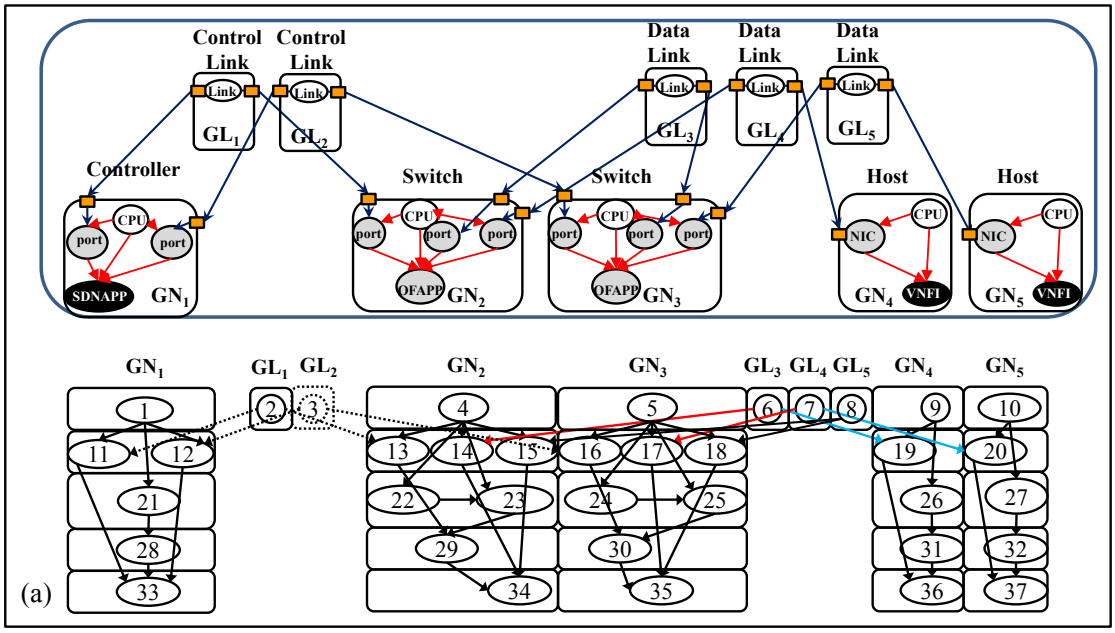


Figure 88. Network Dependency graph of a linear topology (N=2) with out-of-band control

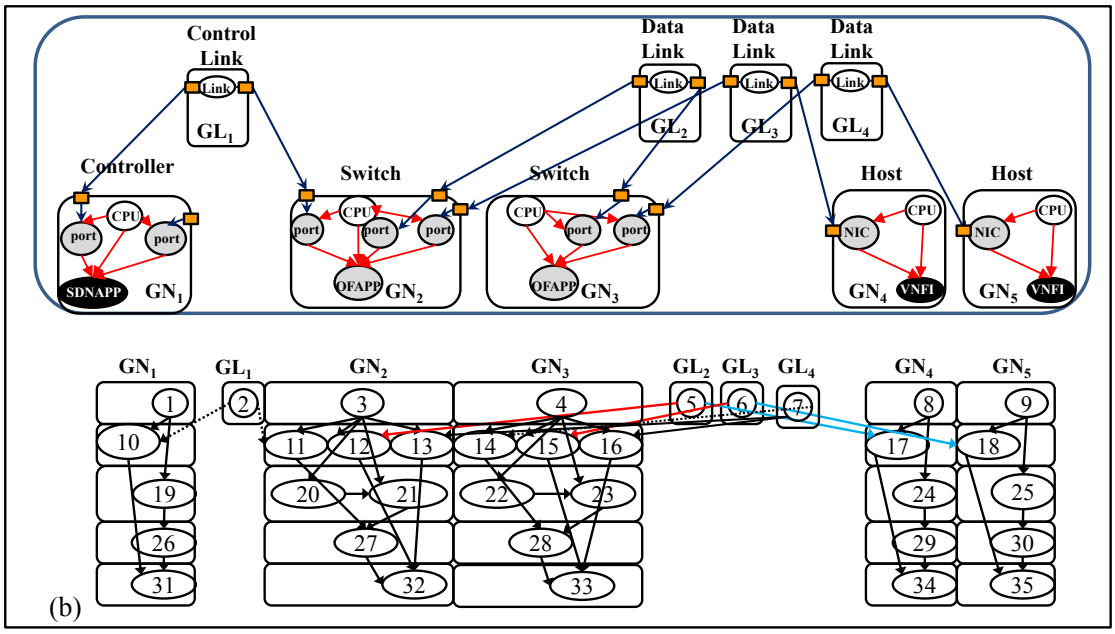


Figure 89. Network Dependency graph of a linear topology (N=2) with in-band control

Figure 88 and Figure 89 show the resulting global dependency graph built by the self-modeling algorithm based on the different templates, topologically ordered.

-The self-modeling algorithm interprets the type of control: In out-of-band control, the self-modeling algorithm instantiates two control links (instances: GL_1, GL_2, vertices: 2, 3), where GL_1 connects both the network card of the master switch (instance: GN_2, vertex: 13) and the network card of the controller (instance: GN_1, vertex: 11). The controller template (GN_1) has two network cards connected to two switches (vertices: 11, 12) through the control link instances GL_1 and GL_2. In in-band control, it only instances one control link (instance: GL_1, vertex: 2) because the controller is only connected to the master switch (GN_2). The controller template then has one network card (vertex: 10), which is connected to the network card of the master switch instance (vertex: 11) through

the control link instance. The other switch is slave (instance: GN_3) and communicates to the controller trough the link IL_1 that connects to the master switch.

-**The self-modeling algorithm interprets the network topology:** For both types for control, it connects both switches through the inter switch link instance (GL_2). It connects both hosts' instances (GN_4 and GN_5) to their respective access links (GL_3 and GL_4).The algorithm can automatically generate ring, star, linear and tree network topologies for different numbers of hosts and switches.

5.2.2 Exploitation of the Network dependency graph for Root Cause Analysis

We analyse in this section how RCA module exploits the network dependency graph generated by the topology-aware self-modeling approach to diagnose logical and physical resources over the SDN infrastructure. We consider a homogeneous failure probability (p), so that all the components inside the network ressources in the dependency graph have the same p=0.1.

5.2.2.1 Diagnosis in SDN infrastructures

We show several examples on how the diagnosis module exploits the network dependency graph to analyze how faults may propagate in the control links between one SDN controller and one OpenFlow switch.

Diagnosis of control-data plane communication at a physical level

We diagnose the communication between an SDN controller and OpenFlow switch at a physical level (Figure 90). The SDN controller is a physical machine with a CPU_C as the base of an SDN controller application like Floodlight to install the OpenFlow rules demanded by the OpenFlow switches. The OpenFlow switches are also physical machines with a CPU_S as the base of the application running on them (OpenFlow client application). Both network entities connect through their respective physical ports to establish the communication.

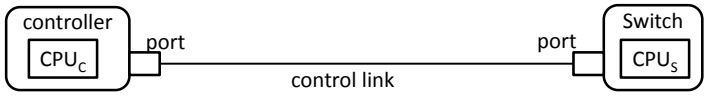


Figure 90. Modelled variables in a control link at physical level

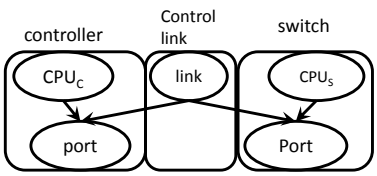


Figure 91. Network dependency graph of a control link at physical level

The network dependency graph of this network link and both SDN nodes is shown in Figure 91, where it can be seen that the control link is linking both dependency graphs, and its fault would propagate to both nodes' ports. The CPTs considered for this example were shown in **Erreur ! Source du renvoi introuvable.**, with a probability of failure p for each network resource (CPU, both ports, and control link) equal to 0.1.

In the first case (a), when we inject as observations to the RCA that the controller's port is down and the switches port is up, the RCA infers that must be an internal problem inside the CPU of the controller. In the second case (b), both ports are down simultaneously, indicating that the link is the most probable root cause. This is the typical connectivity fault between the controller and the switch.

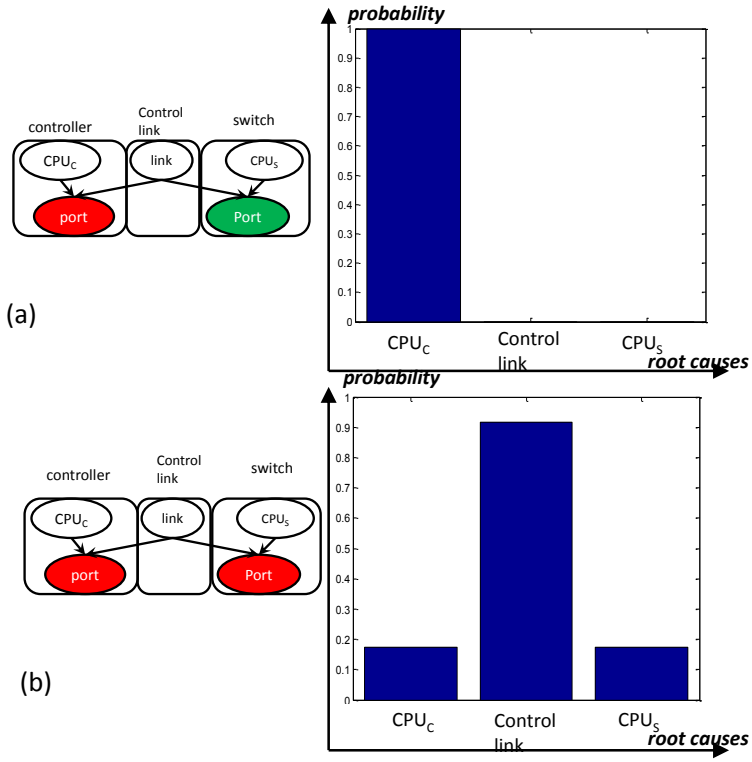


Figure 92. Root Cause analysis: (a) switch's port up, (b) switch's port down

Diagnosis of control-data plane communication at a physical and logical level

We diagnose the communication between an SDN controller and OpenFlow switch at a physical and logical level (Figure 90).

In this case, the network graph, shown in Figure 93, considers also the logical resources running on top of these two SDN nodes, the SDN controller application running on the controller and the OpenFlow client application running on the switch. Both applications have two states: the applications are installed and running, what means that their process identifier would be found in that node, and both applications are connected, what means that they can send or receive information through the corresponding port.

The CPTs considered for this example are shown in**Erreur ! Source du renvoi introuvable.**. The probability of failure p of each network resource (CPU, both ports, and control link) was set to 0.1.

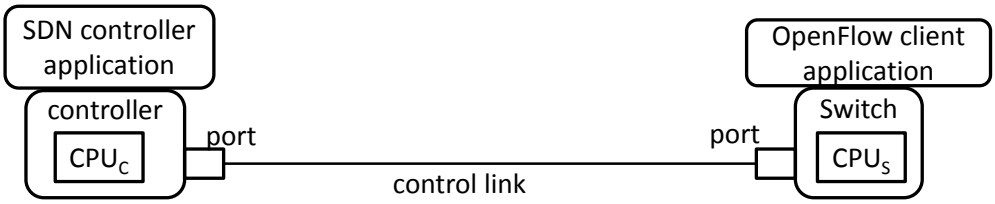


Figure 93. Modelled variables in a control link at physical and logical level

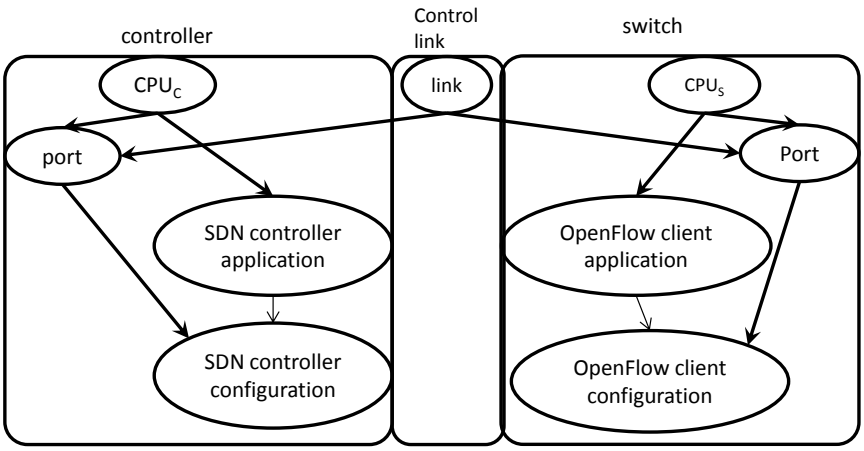


Figure 94. Dependency graph of a control link at physical and logical level

We can see how logical resources running on the nodes add very valuable additional information for the diagnosis. For example, when we launch the RCA taking as observations that the controller's and switch's ports are down, instead of pinpointing the control link as the most probable root cause as in the previous case, both CPU are considered as the most probable root cause (with probability of 0.4) while the control link has as a posteriori probability 0.15. Also, as it is seen in (b), thanks to the addition of the state of the logical resources running on nodes, both CPU components can be discarded from the root cause list and the control link is the root cause, indicating it is a physical fault.

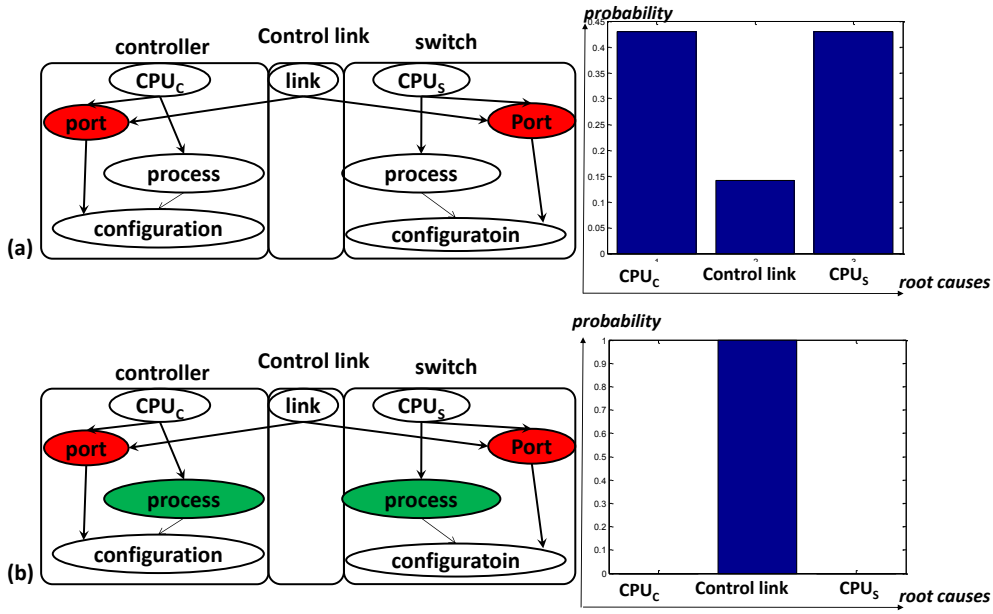


Figure 95. Root Cause analysis with two different observations of controller's card, switch's port and applications state

5.2.2.2 Reactive scenario: diagnosis of faults in the SDN infrastructure

We analyze in this section two cases, we firstly analyze a fault in the SDN controller, and secondly three simultaneous faults in three links at both control and data planes.

Case 1: Fault in the SDN controller

In this first case, the actual root cause is a total shutdown of the SDN controller. The first symptom found in the network is that the SDN controller does not respond to ping. This absence of ping response is understood by the monitoring module as evidence that its interfaces are down, so it informs to the BN module that those controllers' ports are down. The observations with the state of the rest of interfaces and ports of hosts and switches at

both control and data networks are also sent to the BN engine. Those interfaces and ports are shown in Figure 96 in green (healthy interface) and in red (red interface).

The BN engine determines that the most probable root cause is the controller (94.2 %). Thanks to the finer-granularity of these templates proposed, we can zoom on the internal components of the SDN controller (ports, CPU, the Floodlight SDN application, and its associated configuration), on the hosts, and in the switches.

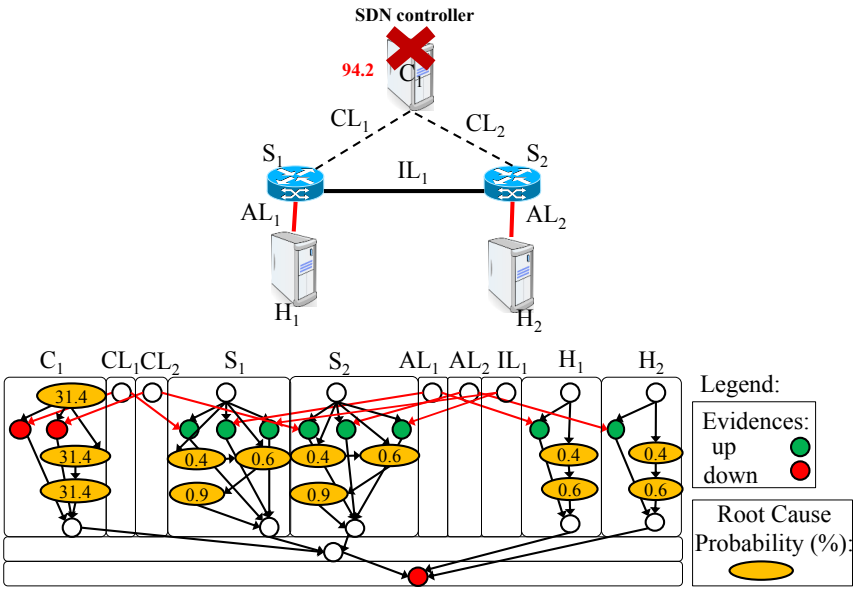


Figure 96. Root Cause analysis: Case 1, Faulty SDN controller

The BN engine determines that the CPU (31.4 %), the Floodlight SDN application (31.4 %) or its configuration (31.4 %) could be the source of the failure. It discards the rest of resources in the rest of network nodes such as switches and hosts (probability below 1 %).

Case 2: Simultaneous faults in control and data links

In this second case, the actual root causes are three: simultaneous link failures in the control link CL_1 and two access links AL_1 and AL_2. As in the previous case, we provided as observations to the BN engine with the state of all the interfaces and ports at both control and data networks. At this case, the SDN controller does respond to ping requests so its interfaces will be up, as seen in Figure 96 in green. However, the SDN controller will not be able to install rules to the switch S_1, as a result, that interface is down, shown in red. In addition, hosts H_1 and H_2 cannot ping their respective switches S_1 and S_2 so their interfaces will be down.

With this information, the BN engine pinpoints those affected links CL_1, AL_1, AL_2 as the most probable root causes (31.1 %), having discarded the second control link CL_2. The SDN controller is almost discarded with a probability of 0.9 % and both switches S_1 and S_2 with 1.8 % as probability of root cause.

Thanks to the finer-granularity of these templates proposed, we can zoom on the internal components of the SDN controller (ports, CPU, the Floodlight SDN application, and its associated configuration), on the hosts, and in the switches.

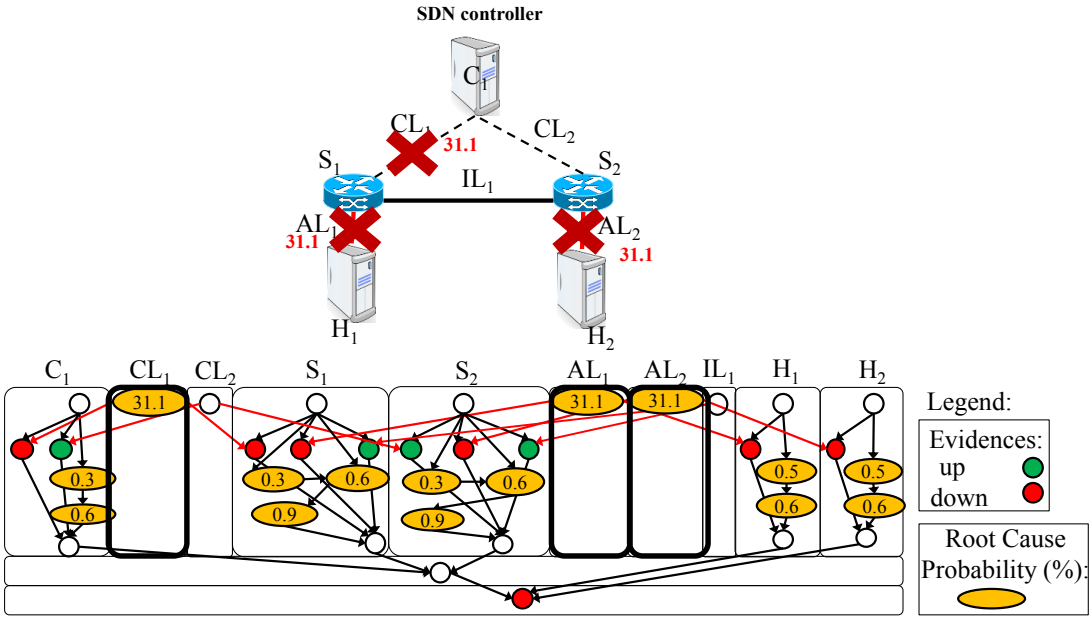


Figure 97. Root Cause analysis: Case 2, simultaneous faulty links at control and data planes

5.2.2.3 Proactive scenario: degradations on the SDN infrastructure

The goal of this section is to show of the RCA can adapt the result of the diagnosis in accordance with the evolution of CPU load in the nodes of the SDN infrastructure. This section also shows how possible future service degradations could be detected in advance by our proposed diagnosis module. We study in this section two different topologies to see how the RCA can receive the network dependency graph of different network topologies.

Linear network topology L=2, out-of-band control

We consider the following centralized SDN infrastructure, composed of a linear network topology L=2 where a service between clients H_1 and H_2 is delivered. In this infrastructure, each network node has a different CPU load. In the event of any degradation on the service, this degradation may be explained by a high CPU load in any of the intermediate nodes involved in the service. In this concrete network, the nodes S_1, S_2, H_1, H_2 and the SDN controller are involved. The SDN controller installs the corresponding flows on both switches S_1 and S_2 in order to connect both clients H_1 and H_2.

The BN engine incorporates observations with the actual load of the CPU on each node, including the CPU load of the SDN controller. The CPU load is included as observation in the CPT table of the CPU vertices in the dependency graph, which will influence the a priori distribution of those vertices, and this, will propagate to the rest of vertices in the dependency graph. As consequence, the calculated root changes as a result of changes on these observations.

At the beginning, the host H_2 is heavily loaded (CPU use 95%) and it is the cause of the service degradation, while the rest of nodes have a normal level of CPU load. In this current situation, the BN engine determines that host H_2 is the most probable root cause (97%) due to this high CPU use, while it discards all the links (with probability of 1 %) as well as the SDN controller (with probability of 7.9 %) as probable causes. Figure 98 shows the a posteriori probability distribution per network node.

Suddenly, the distribution of the CPU load changes, where the CPU load on the host H_2 plummets to a 5 %, while the CPU load on the SDN controller starts raising rapidly until 95% while the rest of nodes have normal level of the CPU load.

Then, the BN engine adapts to the current situation and diagnoses by taking into account the observations with the current CPU load of the network nodes. It then pinpoints the SDN controller as the most probable root cause to explain the degradation on the SDN infrastructure (a transition of root cause probability from 7.9 % to 96.6 %).

Table 23. CPU loads at both instants of time

CPU load	SDN controller C_1	Switch S_1	Switch S_2	Host H_1	Host H_2
Instant t_1	5 %	20 %	40 %	2 %	95%
Instant t_2	95 %	2 %	10 %	35 %	5%

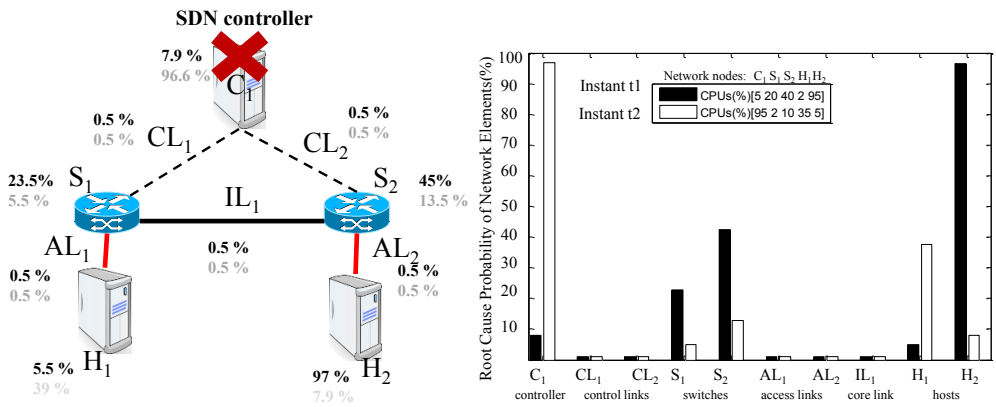


Figure 98. Root Cause Analysis on changing CPU conditions in linear L=2 network topology

Single network topology, out-of-band control

We consider the following centralized SDN infrastructure, composed of a single network topology where a service between clients H_1 and H_2 is delivered. In this concrete network, the nodes S_1, H_1, H_2 and the SDN controller are involved.

At the beginning, there is not service degradation, because any network node is heavily loaded (all the CPU loads are below 30%). In this current situation, the BN engine determines that the switch S_1 is the most probable root cause (27.3%) due to this the fact its CPU load most network is at 30 %, while it discards all the links (with probability of 1 %) as well as the SDN controller (with probability of 13.5 %) as probable causes. Figure 99 shows the a posteriori probability distribution per network node.

Suddenly, the distribution of the CPU load changes, where the CPU load on the switch S_1 plummets from 30% to 10% %, while the CPU load on the SDN controller starts raising rapidly from 15 to a moderate CPU load 65% while the rest of nodes have normal level of the CPU load (5-10%).

Then, the BN engine adapts to the current situation and diagnoses by taking into account the observations with the current CPU load of the network nodes. It then pinpoints the SDN controller as the most probable root cause to explain the degradation on the SDN infrastructure (a transition of root cause probability from 13.5% to 59.2 %).

Table 24. CPU loads at both instants of time

CPU load	SDN controller C_1	Switch S_1	Host H_1	Host H_2
Instant t_1	15 %	30 %	15 %	15%
Instant t_2	65 %	10 %	5 %	5%

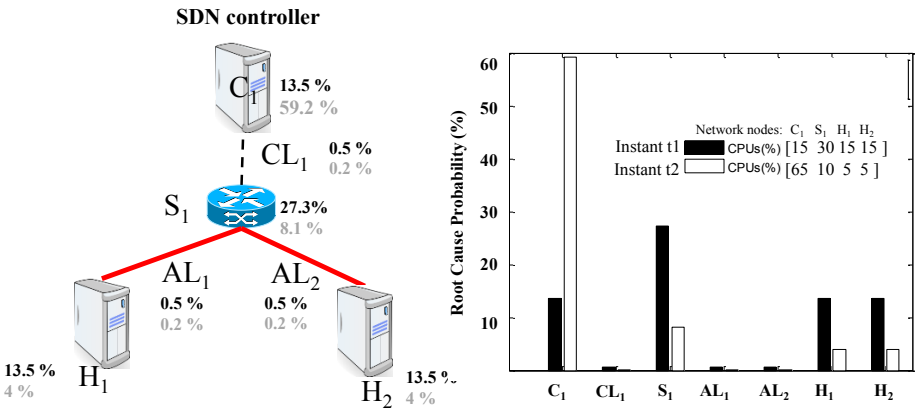


Figure 99. Root Cause Analysis on changing CPU conditions in single network topology

5.2.3 Performance Evaluation

In this section, we evaluate performance of the topology-aware self-diagnosis approach.

Growth in number of generated vertices

We study the growth in number of vertices (*V*) of the network dependency graph for linear and tree topologies for out-of-band control. We analyze both topologies for a varying number of connected hosts (N_{HOSTS}) from 4 up to 256. The number of network elements ($N_{ELEMENTS}$) (nodes and links) is the same for both topologies $N_{ELEMENTS}=3N_{SWITCHES}+2N_{HOSTS}$. The number of vertices in the global dependency graph G is:

$V=V_{CONTROLLER}+V_{SWITCHES}N_{SWITCHES}+V_{HOSTS}N_{HOSTS}+V_{LINK}(2N_{SWITCHES}+N_{HOSTS}-1)$. If we particularize with the values for the aforementioned topology (Figure 88) in out-of-band control: 5 vertices per host template ($V_{HOST}=5$), 8 vertices per switch template ($V_{SWITCHES}=8$), 1 vertex per link template ($V_{LINK}=1$) and 5 vertices per controller template ($V_{CONTROLLER}=5$), this equation becomes $V=5+10N_{SWITCHES}+6N_{HOSTS}$, which explains the linear trend of vertices with the number of hosts described in Table II.

Table 25. Number of vertices (V) as a function of the number of hosts (N_H)

Topology/N_H	4	8	16	32	64	128	256
Tree	62	130	266	538	1082	2170	4346
Linear	72	140	276	548	1036	2180	4356

Speed of self-modeling algorithm

We study the speed of the self-modeling algorithm as a function of the number of network elements ($N_{ELEMENTS}$) to evaluate the impact of the number of network elements in the performance of the algorithms. We launched the self-modeling algorithm for both linear and tree topologies in out-of-band control, ranging from 15 up to 500 network elements. The number of network elements includes the control and data links, the hosts, the switches and the SDN controller.

We averaged the computing time 20 times per network topology to obtain values that are more reliable.

Figure 100 shows the curve of time required by the topology-aware self-modeling algorithm as a function of the number of network elements considered in the network topology. It can be seen an exponential trend in the growth of self-modeling time with the number of network elements for both linear and tree topologies.

Linear topologies scale a little better than tree topologies, but there are not very significant differences between both network topologies in terms of speed. In both cases, the self-modeling time remains less than 30 seconds even when the number of network elements is in 500 elements.

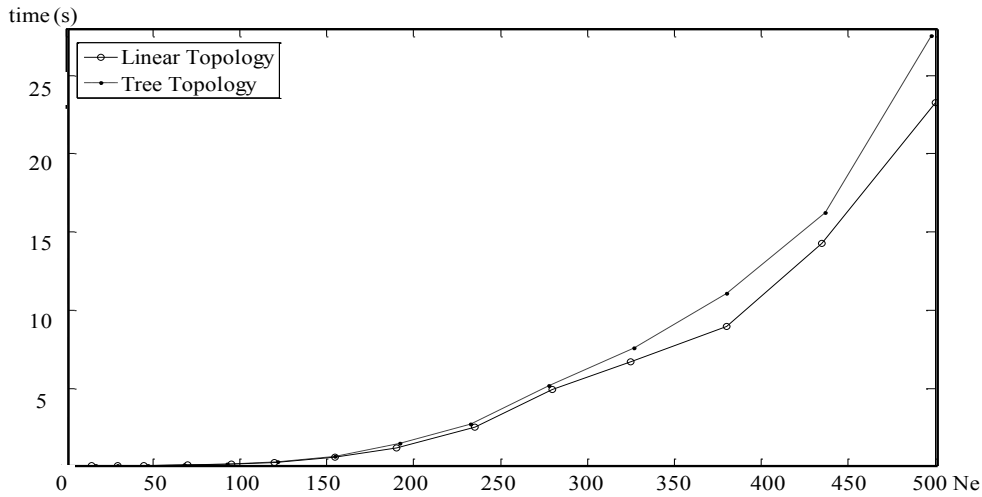


Figure 100. Speed as a function of the number of elements

5.3 Service-Aware Self-Diagnosis

In this section we evaluate how the service dependency graph is generated and then exploited to find the root cause.

5.3.1 Generation of the services dependency graph

In this section, we diagnose two networking services delivered in two different network topologies (Figure 101 (a) and Figure 101 (b)) where we apply the two aforementioned RCA strategies. Each networking service is composed of two VNFs, whose instances $VNFI_{Ai}$ and $VNFI_{Bi}$, are embedded in different hosts. Both VNFIs are connected through a virtual link $VL_{Ai,Bi}$, which is established at run-time by the SDN controller. Table 26 shows the dependencies of each networking service from the underlying physical elements.

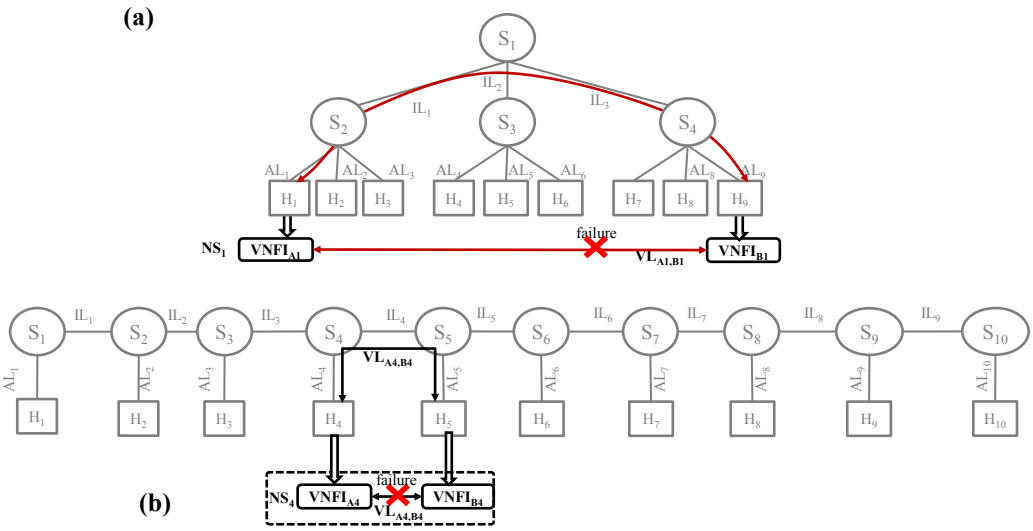


Figure 101. Diagnosis of two networking services in different network topologies: (a) tree topology, (b) linear topology

Table 26. Affected networking services and underlying physical paths

Topology	*Service*	*Virtual Link*	*Host:VNFI*	*Physical path*
Tree **D=2, F=3**	NS_1	$VL_{A1,B1}$	$\mathbf{H_1}$:$VNFI_{A1}$ H_9:$VNFI_{B1}$	DP:[$\mathbf{AL_1,S_2,IL_1,S_1,IL_3,S_4}$,$AL_9$], CP:[$\mathbf{C_0,CL_1,CL_2,CL_3,CL_4,C_1}$]
Linear **L=10**	NS_4	$VL_{A4,B4}$	$\mathbf{H_4}$:$VNFI_{A4}$ H_5:$VNFI_{B4}$	DP:[$\mathbf{AL_4,S_4}$,IL_4, S_5,AL_5], CP:[$\mathbf{C_0,CL_4}$,CL_5,$\mathbf{C_1}$]

*S: switch, IL: inter switch link, CL: control link, AL: access link, C: controller, H:host

DP: Data plane (hosts,switches,datalinks), CP: Control plane (controllers,control links)

First, the self-modeling module generates on-the-fly the *services dependency graph* (that includes the *network dependency graph).* The *services dependency graph* is generated in the three following situations:

Changes on the network topology: The *network dependency graph* is generated for a tree topology (D=2, F=3), in Figure 101 (a), and a linear topology with L ∈ [5,10] Figure 101 (b). In addition, the self-modeling module models changing topologies by discovering new network resources and regenerating the *network dependency graph*, as shown in this video (Sanchez et al, 2016). The *network dependency graph* includes the connections from the SDN controller to the switches, not shown here.

VNF migrations: The self-modeling algorithm generates *the services dependency graph* taking into account both distributions of VNFIs. If VNFIs migrate, the self-modeling regenerates the *services dependency graph* with the new distribution of VNFIs. For instance, in Figure 102 (b), the VNFIs are embedded in different hosts and th eservice dependency graph takes this into consideration to be able to propagated hardware failures on the hosts to their VNFs embedded.

Changes on the virtual links: VNF migrations and topological changes lead to changes on the virtual links connecting them. Table 31 presents the underlying physical resources involved in all the networking services present in each network topology. In both topologies, networking services share some physical network resources such as physical links, switches and part of the control plane. The shared network resources are depicted in bold. This information will be exploited by the RCA to reduce the uncertainty.

Table 27. VNF forwarding graphs and physical dependencies

Topology	*Network Service*	*Virtual Links*	*Host:VNFI*	*Physical Dependencies*
Linear N=10	NS_1	$VL_{A1,B1}$	H_1: VNF_{A1} H_2:VNF_{B1}	**H_1, AL_1, Ms1, IL_1, Ms2,** AL_2, H_2, **CL_1, CL_2, C_1**
	NS_2	$VL_{A2,B2}$	H_1: VNF_{A2} H_3: VNF_{B2}	**H_1, AL_1, Ms1, IL_1, Ms2,** IL_2, **Ms3, AL_3, H_3, CL_1, CL_2, CL_3, C_1**
	NS_3	$VL_{A3,B3}$	H_3: VNF_{A3} H_4: VNF_{B3}	**H_3, AL_3, Ms3,** IL_3, **Ms4, AL_4, H_4, CL_3, CL_4, C_1**
	NS_4	$VL_{A4,B4}$	H_4: VNF_{A4} H_5: VNF_{B4}	**H_4, AL_4, Ms4,** IL_4, Ms5, AL_5, H_5, **CL_4,** CL_5, **C_1**
Tree D=2,F=3	NS_1	$VL_{A1,B1}$	H_1: VNF_{A1} H_9: VNF_{B1}	**H_1, AL_1, Ms2, IL_1, Ms1, IL_3, Ms4,** AL_9, H_9, **CL_1, CL_2, CL_3, CL_4, C_1**
	NS_2	$VL_{A2,B2}$	H_1: VNF_{A2} H_3: VNF_{B2}	**H_1, AL_1, Ms2,** AL_3, H_3, **CL_1, C_1**
	NS_3	$VL_{A3,B3}$	H_1: VNF_{A3} H_6: VNF_{B3}	**H_1, AL_1, Ms2, IL_1, Ms1, IL_2, Ms3,** AL_6, H_6, **CL_1, CL_2, CL_3, C_1**
	NS_4	$VL_{A4,B4}$	H_1: VNF_{A4} H_8: VNF_{B4}	**H_1, AL_1, Ms2, IL_1, Ms1, IL_3, Ms4,** AL_8, H_8, **CL_1, CL_2, CL_3, CL_4, C_1**
	NS_5	$VL_{A5,B5}$	H_4: VNF_{A5} H_7: VNF_{B5}	H_4, AL_4, **Ms3, IL_2, Ms1, IL_3, Ms4,** AL_7, H_7, **CL_1, CL_4, C_1**
	NS_6	$VL_{A6,B6}$	H_2: VNF_{A6} H_5: VNF_{B6}	H_2, AL_2, **Ms2, IL_1, Ms1, IL_2, Ms3,** AL_5, H_5, **CL_1, CL_2, CL_3, C_1**

*S: switch, IL: inter switch link, CL: control link, AL: access link, C: controller, H:host

5.3.2 Exploitation of the Service dependency graph for Root Cause Analysis

In this section, we show how the RCA can adapt and exploit the *services dependency graph* and the *network dependency graph* to efficiently diagnose networking services failures in two different cases. The diagnosis is automated by the on-the-fly generation of both dependency graphs. The RCA calculates the root cause. i.e. the RCA identifies physical, logical and virtual network resources presumed to be the root cause of a given networking service failure. We consider that all the network resources and their internal components have the same a priori probability of fault (p=0.1) in the conducted experiments.

We apply the definition of the entropy $H(X)$ to evaluate the uncertainty of the aposteriori distribution probability $p_{X_1,\ldots,X_N}$ calculated by the RCA based on the injected observations from the network. This calculation is based on the probability of each network resource in the root cause list is down.

$$H = -\sum_{X_i \in N} \Pr(X_i = down)\, log\, \Pr(X_i = down)$$

We propose two RCA strategies to reduce this uncertainty, which effectiveness is proved in this section.

Extension of the *services dependency graph*: we define an RCA strategy that extends the *services dependency graph* to include the dependencies of the healthy networking services that are sharing resources with the affected service. This RCA strategy allows discarding those network resources involved in healthy services.

Reduction of the *network dependency graph*: we define an RCA strategy that reduces the *network dependency graph* to only consider the dependencies of network resources that are involved in the identified faulty networking services, thereby reducing the uncertainty and the diagnosis time.

5.3.2.1 Extension of the services dependency graph

We consider the tree topology (Figure 102), where the services are deployed sequentially i.e. at $t_i=t_0 + (i-1)T$, i=1…N. A failure is injected in service NS_1 and the self-modeling algorithm is launched at $t_1 = t_0$, generating the *services dependency graph* from the affected service NS_1 and the RCA gives a posteriori distribution probability (Figure 104 dark blue bar on the bottom) so spread over all the network resources that no root cause can be clearly identified (entropy: 4.1 bits). The uncertainty can be reduced by adding the healthy services sharing resources with the affected service. Indeed, if the graph is regenerated at $t_2=t_0+T$, when a healthy service NS_2 is deployed (N=2), the root cause becomes less uncertain (entropy: 3.6 bits). Adding the new healthy service NS_2 allows the RCA to discard those shared resources between the affected service NS_1 and NS_2 (S_2, AL_1 and the control plane resources). The RCA module extends the *services dependency graph* to reduce the entropy four times more by including the healthy services as those appear: NS_3 at $t_3=t_0+2T$, NS_4 at $t_4=t_0+3T$, NS_5, at $t_5= t_0+4T$, and NS_6 at $t_6=t_0+5T$. Figure 103 shows how the entropy is reduced from 4.1 (dark blue bar on the bottom), with the only affected service added, to 0.9 bits (brown bar on top), with the affected service and 5 healthy services added.

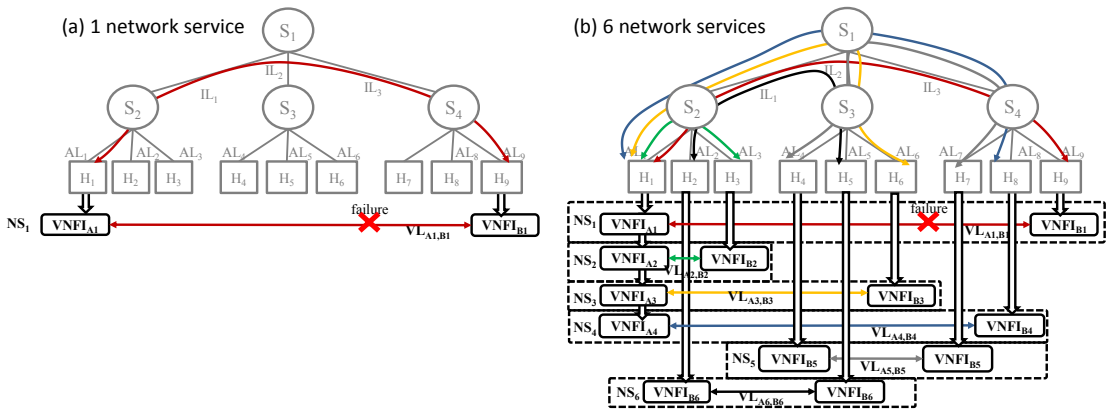


Figure 102. RCA strategy on extending the services dependency graph with network services

In the brown bar probability distribution (Figure 104 on the bottom), the root cause list consists of the hosts H_1 (33%) and H_9 (67%). The rest of hosts are discarded, as those are not involved in the affected service. Our finer granular templates enable a deeper analysis. Not only links and switches are shared among services, but also the CPU and NIC inside hosts. Host H_1 embeds four VNFIs, as a result, those VNFIs share NIC and CPU. Nevertheless, NIC and CPU are immediately discarded when at least one of these VNFIs is involved in a healthy service as it means that NIC and CPU is working fine. Indeed, we see that the most probable explanations (Table 28) are that $VNFI_{A1}$ and $VNFI_{B1}$ are not initiated, configured, or activated (adding up all VNFI states: $VNFI_{A1}$ (33%) and $VNFI_{B1}$ (51%)), which is coherent with the injected failure in NS_1, composed of those VNFs. In addition, the RCA can

discard those VNFIs embedded in H_1 ($VNFI_{A2}$, $VNFI_{A3}$, and $VNFI_{A4}$), because they are not involved in the affected service NS_1.

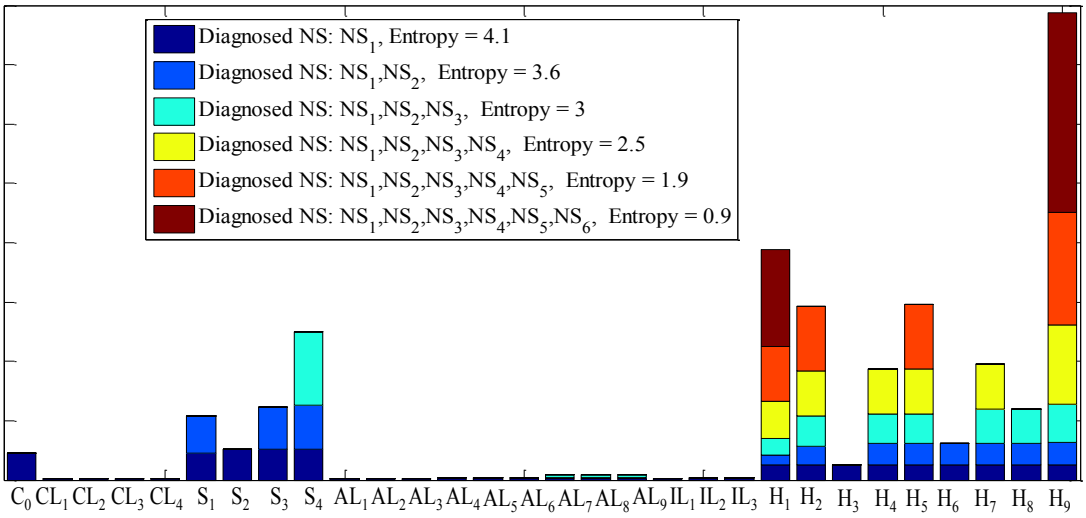


Figure 103. Entropy reduction with the RCA strategy on extending the services dependency graph

Table 28. Zoom on the root cause probabilities per host (%)

Host	*CPU*	*NIC*		*VNFI Not Instantiated*	*VNFI Not Configured*	*VNFI Not Active*
H_1	0	0	$VNFI_{A1}$	6	11	16
			$VNFI_{A2}$	0	0	0
			$VNFI_{A3}$	0	0	0
			$VNFI_{A4}$	0	0	0
H_9	6	11	$VNFI_{B1}$	11	16	24

5.3.2.2 Reduction of the network dependency graph

We first inject a failure in service NS_4 and generate the *network dependency graph* from the network topology of the blue region in Figure 105 and we incrementally reduce the diagnosis region until the optimal one (brown diagnosis region) that gives the lowest uncertainty because the minimum amount of networking resources are diagnosed.

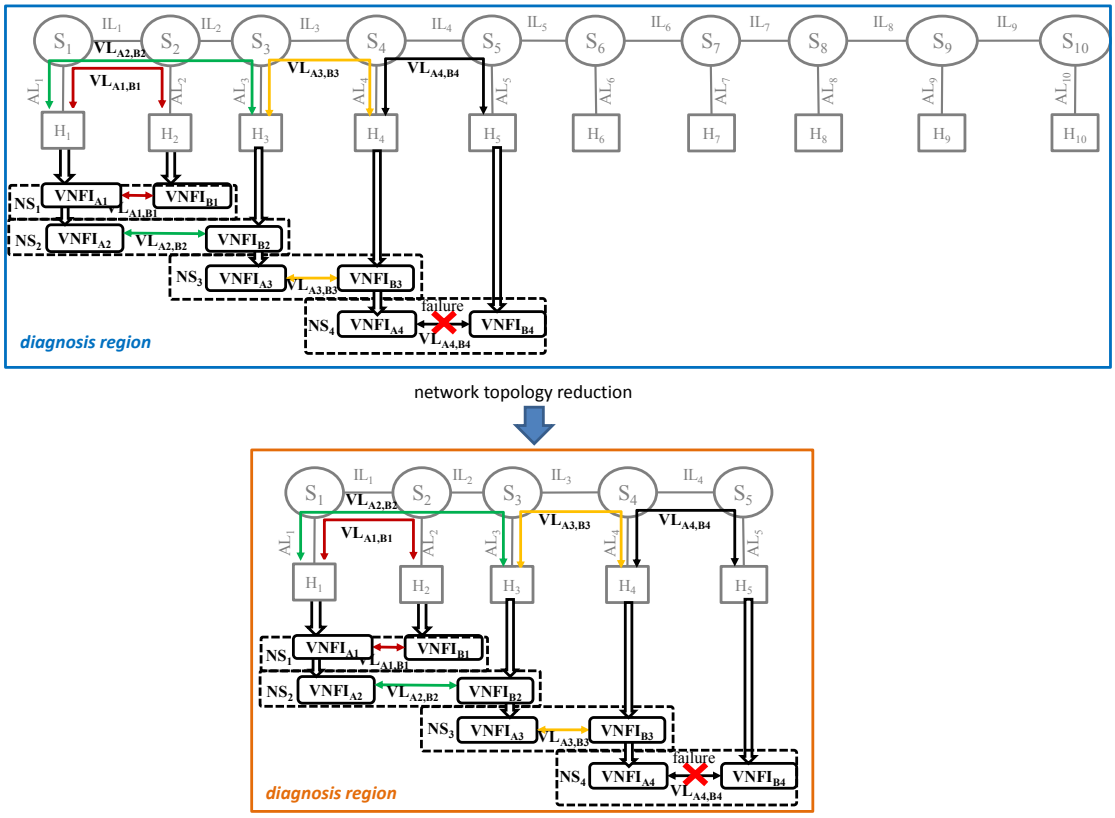


Figure 104. RCA strategy on reducing the network dependency graph

Blue region: the *network dependency graph* is built from a linear topology L=10 and it includes the following services to build the *services dependency graph*:

- (i) the affected service NS_1: the a posteriori distribution probability has as entropy 4.7 bits.
- (ii) the affected service and the healthy services NS_2, NS_3, and NS_4: the a posteriori distribution probability (, Figure 105 dark blue bar on the bottom) has lower entropy (3.9 bits), because the added networking services help discard those network resources involved in them.

In both situations (i) and (ii), the a posteriori distribution is so spread over the existing network resources that no root cause can be identified.

Brown region: the *network dependency graph* is built from a linear topology L=5 and it includes the following services to build the *services dependency graph*:

- (i) the affected service NS_1: the a posterior distribution probability has as entropy 2.2 bits.
- (ii) the affected service and the healthy services NS_2, NS_3, and NS_4: the a posteriori distribution probability (brown bar on top) has lower entropy (1.6 bits) because the added services help discard those network resources involved in them.

Figure 105 shows that the uncertainty on the root cause is reduced when the diagnosis region gets closer to the brown diagnosis region: In situation (i) there is a reduction from 4.7 to 2.2 bits with one service added. In situation (ii) there is a reduction from 3.9 to 1.6 bits with 4 services added (Figure 105 on the bottom). We focus on the situation (ii), where a clear subset of the network–S_5 (48%), AL_5 (3%), H_4 (15%), and H_5 (35%)–is presumed to be the root cause. This result is coherent with the injected failure in NS_4 as its underlying virtual resources, the VNFI embedded in hosts H_4 ($VNFI_{A4}$) and H_5 (VNF_{B4}), are pinpointed as possible root causes. Analogously as in previous section, we can zoom on hosts H_4 and H_5 (Table 29) to obtain the probability of fault in the VNFIs running inside those hosts. The most probable explanation is that those VNFIs embedded on H_4 and H_5 are not initi-

ated, configured, or active (adding up all VNF states: $VNFI_{A4}$ (17%), and $VNFI_{B4}$ (25%)). Contrarily, the hosts embedding VNFIs which are not involved in the affected service (i.e. H_1, H_2, and H_3) are discarded. Furthermore, other VNFIs (e.g. $VNFI_{B3}$) embedded in the hosts presumed to be the root cause (H_4) but not involved in the affected service are discarded. In all regions, those network resources not involved in the affected service NS_4 are discarded (e.g. S_1, S_2, S_3, H_1, H_2, H_3 among others).

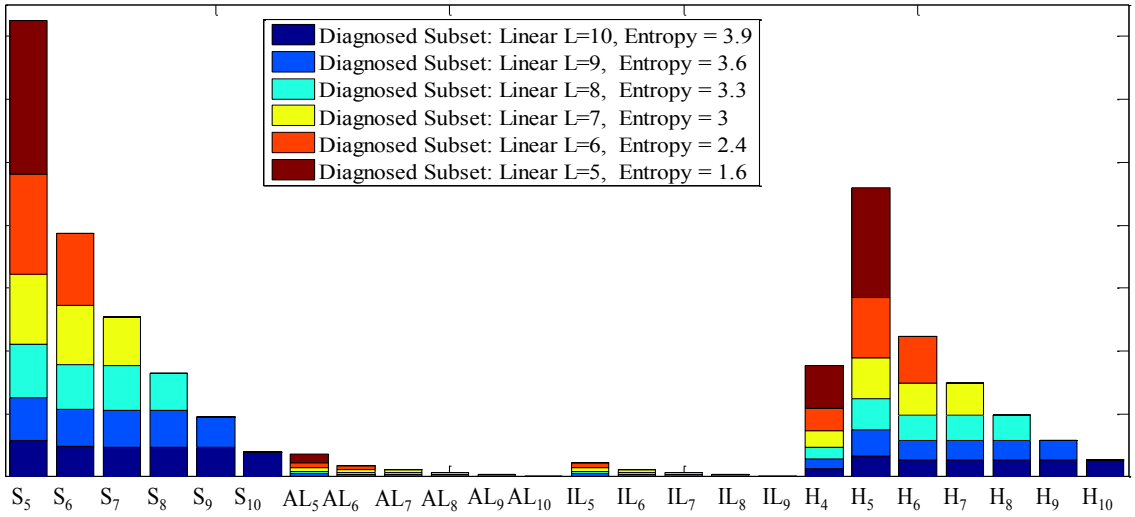


Figure 105. Entropy reduction with the RCA strategy on reducing the network dependency graph

Table 29. Zoom on the root cause probabilities per host (%)

Host	CPU	NIC		VNFI Not Instantiated	VNFI Not Configured	VNFI Not Active
H_4	0	0	$VNFI_{B3}$	0	0	0
			$VNFI_{A4}$	3	5	7
H_5	3	7	$VNFI_{B4}$	5	7	13

5.3.3 Performance Evaluation

We evaluate the performance of both RCA strategies that reduce the uncertainty on the diagnosis of networking services, measured in terms of generated vertices and edges in the dependency graph and diagnosis time.

Case 1, extension of the *services dependency graph*: The RCA strategy that extends the *services dependency graph* in Figure 101 (a), adds a lower number of vertices per service added compared to the number of edges added, as seen in Table 30. This difference is due to the high number of dependencies (E_{VL} edges) of each virtual link from the physical resources (NICs, switches ports, and OpenFlow client applications inside switches). For instance, Figure 106 shows 7 edges (5 E_{VL} and 2 E_{VNF}) and 4 vertices added. These added edges and vertices increase the diagnosis time $t_D = t_{SM} + t_{RCA}$, where t_{SM} is the self-modeling time and t_{RCA} is the RCA time, both averaged 20 times. t_{SM} represents at least 51% of the diagnosis time t_D. When six services are added to the graph, t_{SM} is increased a 57% of t_D with respect to one service added, whilst the t_{RCA} is increased by 72% of t_D, proving that the BN engine inside the RCA scales worse than the self-modeling algorithm in itself.

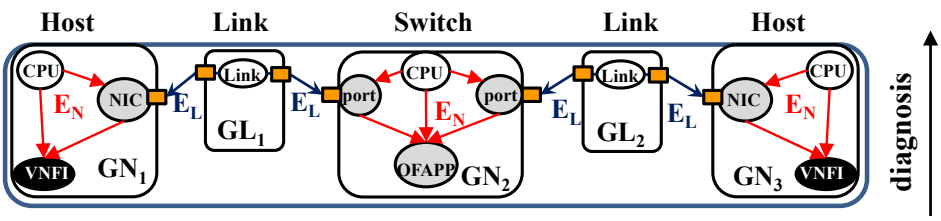


Figure 106. Example of Network dependency graph (Q=3 nodes and P=2 links)

Table 30. Cost of extending the services dependency graph

Services added	*#Vertices*	*#Edges*	t_{RCA}	t_{SM}
NS_1	108	306	1.1	1.4
NS_1, NS_2	115	334	1.2	1.6
NS_1, NS_2, NS_3	122	372	1.6	1.7
NS_1, NS_2, NS_3, NS_4	129	410	1.7	2
NS_1, NS_2, NS_3, NS_4, NS_5	133	440	1.7	2.1
NS_1, NS_2, NS_3, NS_4, NS_5, NS_6	137	470	1.9	2.2

Case 2, reduction of the *network dependency graph*: The RCA strategy that reduces the diagnosis region reduces also the diagnosis time, as the diagnosed network topology is smaller. As example of this reduction, we compare the size of the *services dependency graph* when it is generated from the blue region in (linear topology L=10) to the *services dependency graph* generated from the brown region in Figure 101 (b) (linear topology L=5) resulting from reducing the diagnosis region. The graph includes the 4 networking services (NS_1 … NS_4). The number of vertices is reduced from 196 to 111 vertices while the number of edges is reduced from 592 to 350 edges, and the diagnosis time is almost divided in half, transitioning from 4 to 2.1 seconds (averaged 20 times).

Table 31. Vertices generated in several network topologies

Network Topology	*Type of Control*	*Network Elements*	*Networking services*	*Generated vertices*
Tree (D=2, F=15)	in-band	482	2	1486
	out-of-band	497		1533
	in-band	482	100	1780
	out-of-band	497		1827
Linear (N=100)	in-band	400	2	1409
	out-of-band	499		1707
	in-band	400	100	1703
	out-of-band	499		2001

5.4 Self-Healing framework for video streaming applications over SDN

We include our topology-aware self-diagnosis block inside a self-healing framework to be able to detect, diagnose and recover dynamic SDN infrastructures delivering streaming services. This section evaluates this self-healing framework as a whole when enabled by a topology-aware self-diagnosis approach.

5.4.1 Overall description of the self-healing testbed

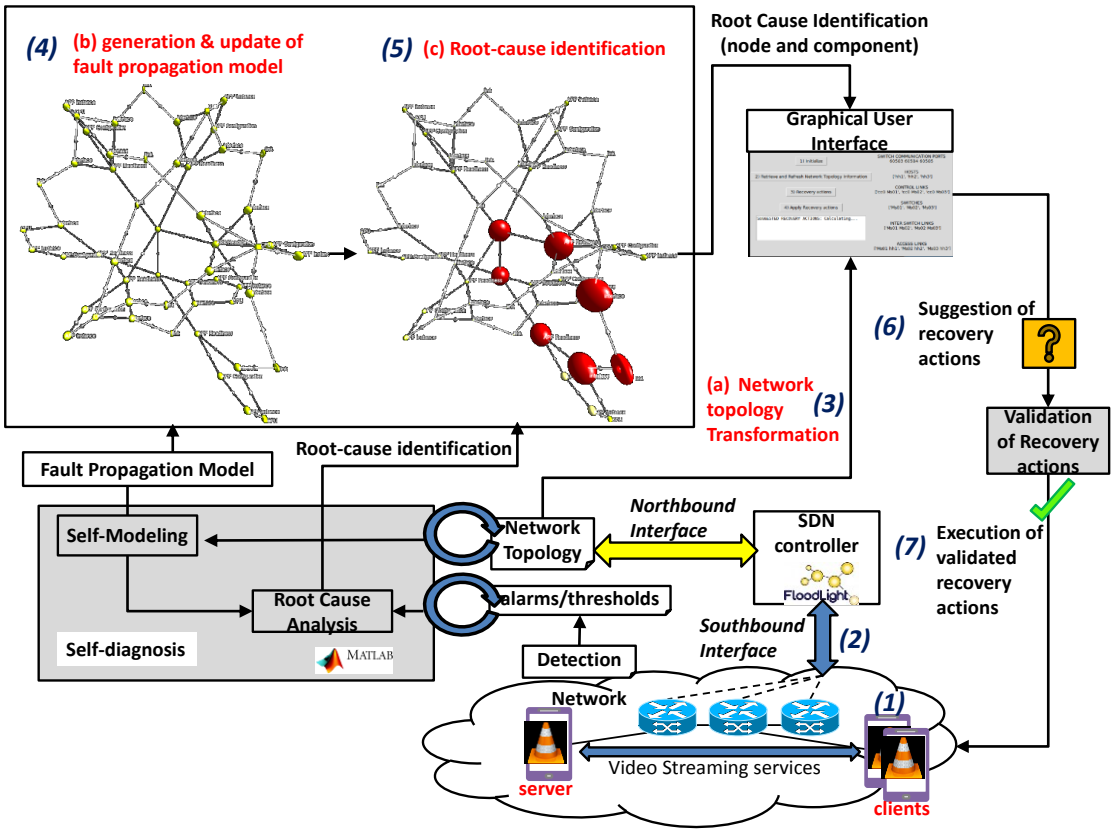


Figure 107. Implementation of the self-healing framework

The workflow of the tesbed is as it follows: A new client arrives and demands the content to the streaming server (1), which starts sending the content. The SDN controller installs the necessary flows on the intermediate OpenFlow switches (2) to allow the streaming packets coming from the content server to be sent to the client and vice versa. Then, the Graphical User Interface (GUI) retrieves the network topology from the SDN controller and classifies the network elements in hosts, switches, links and logical ports, which are shown in the GUI (3). The self-modeling algorithm generates the fault propagation model (4) from this classified list of network elements by instantiating their corresponding templates and assembling their respective dependency graphs, as explained in the previous chapter. If the network topology changes, the self-modeling block regenerates the fault propagation model by incorporating the newly added elements. Once the model is generated, the root cause analysis module will be triggered by the alarms indicating a malfunction in the streaming service and it will update the fault propagation model (5) with the root cause(s). It indicates as root cause a network node but also its internal component (CPU, port, card, application, etc.). Once the root cause is identified, the root cause analysis block suggests a recovery action (6) that will be validated by a human administrator (7) once is proved it re-establishes the streaming service.

5.4.2 Implementation

The self-healing framework is composed of the self-diagnosis block and the SDN infrastructure. The self-diagnosis block includes the self-modeling and the RCA blocks. The self-healing framework is mostly composed of open source software, as exception of MATLAB R2013A used in the self-diagnosis block.

The probabilistic dependency graph depicting the fault propagation model is visualized through the 3-D graph visualizer named UbiGraph in(UbiGraph,-) . A graphical user interface (GUI), to show the network topology with the network elements already classified, is implemented in python with the Qt software library in (Qt, 2015).

This self-healing framework is running on a single physical machine running on a Windows 7 professional 64 bits as OS. Inside this OS we run the self-diagnosis module based on the Bayesian Network algorithm which is based on the Kevin Murphy's Bayesian Networks Toolbox in (Murphy et al., 2001) running in MATLAB R2013A.

The SDN environment is completely virtualized and embedded in a VM (managed by VirtualBox) which is running a Xubuntu OS. The SDN controller is Floodlight, while the SDN infrastructure is emulated through Mininet (Lantz et al, 2010). For instance, in our previous testbed (Sanchez et al, 2014), the SDN controller used was POX (McCauley, 2012). Inside the Mininet emulator, the video streaming application is running on several nodes of that virtualized network topology and sends streaming videos through the SDN infrastructure managed by the SDN controller.

In order to share information among the guest OS (Windows 7) and the host OS (Ubuntu), a shared folder is used, where we embed several descriptors of the network topology extracted from the SDN controller, the logical ports of the OpenFlow switches, among other necessary information to build the probabilistic dependency graph. In addition, there are two files that contain the state of the network and updates in the network topology. When there is a network topology change, an alarm is generated and this information is notified to the Windows OS by changing the state on this file, which is continuously read by the self-modeling algorithm running in MATLAB. Similarly, in the presence of a fault in the network, an alarm is generated and this information is notified to the Windows OS by changing the state on a file, which is continuously read by the RCA algorithm running in MATLAB.

5.4.3 Transformation of the network topology into a machine-readable format

Most open source SDN controllers such as OpenDaylight or Floodlight only provide with a visual description of the network topology which is just not enough for advanced processing like diagnosis. Nevertheless, we developed an algorithm that takes as input the network topology given by the SDN controller (in a JSON format) and generates a machine readable format descriptor containing the information of the network topology at instant t.
First, once the user clicks on the 'Retrieve and Refresh Network Topology Information' button of the GUI, the network topology is retrieved from the SDN controller through the northbound API at instant t, as it was discussed in previous chapters. Then, each network element retrieved is classified in links and network nodes and then each network nodes is classified in switches, controllers or hosts and are shown in their corresponding section in the GUI.
The developed GUI is shown in Figure 108, where the following elements of a linear (N=3) network topology can be found. This GUI shows advanced information of the network topology such as the communication ports to communicate switches with the OpenFlow controller, the hosts identifiers, the switches identifiers, the control links identifiers, the inter switch links identifiers and the access links identifiers.

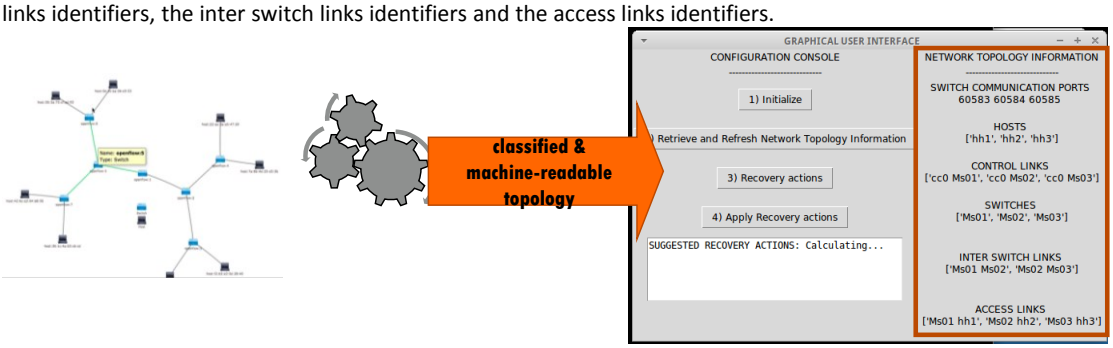


Figure 108. Transformation of the network topology into a machine-readable format

5.4.4 Construction of the dependency graph

Each time the user clicks on the 'Retrieve and Refresh Network Topology Information' button of the GUI, the dependency graph is regenerated with the current information of the network topology given by the controller. The self-modeling algorithm generates the dependency graph first and represents it as a 3-D graph to make easier the interpretation by human operators. This 3-D graph is composed of different types of network components, depicted with different vertices with shapes specified in the following table. The size of each symbol is the same at the beginning, as long as the RCA has not been launched yet.

Table 32. Shape of the supervised network components included in the dependency graph

Symbol	*Type of network component*
ball	application or VNF
cone	switch port or network card interface
torus	link
cube	CPU

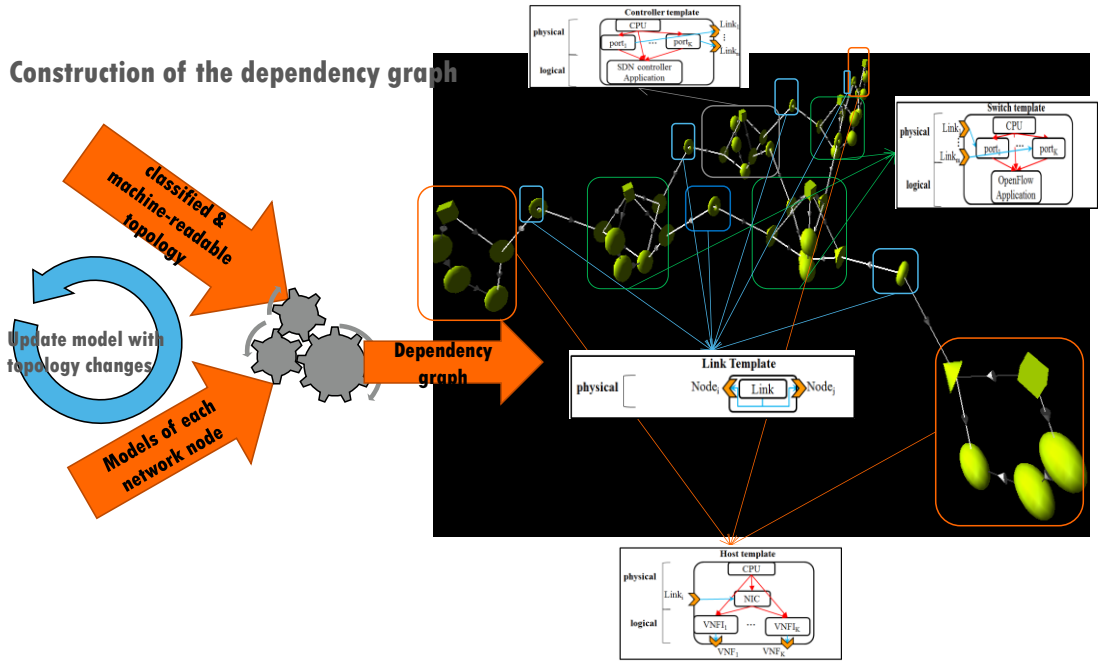


Figure 109. Construction of the dependency graph from templates

5.4.5 Root cause analysis

The RCA block is in charge of identifying the root cause with the Bayesian networks approach to calculate the probability of faulty elements in the current network topology.

Firstly, the network dependency graph is generated by the self-modeling algorithm and is then filled with the observations gathered from the following network components:

- CPU load on network nodes
- state of switches' ports,
- state of SDN controller's ports,
- state of hosts' network cards,
- state of the SDN controller application,
- state of the OpenFlow client applications running on switches,
- state of VNFIs
- state of the video streaming application running on the clients and on server

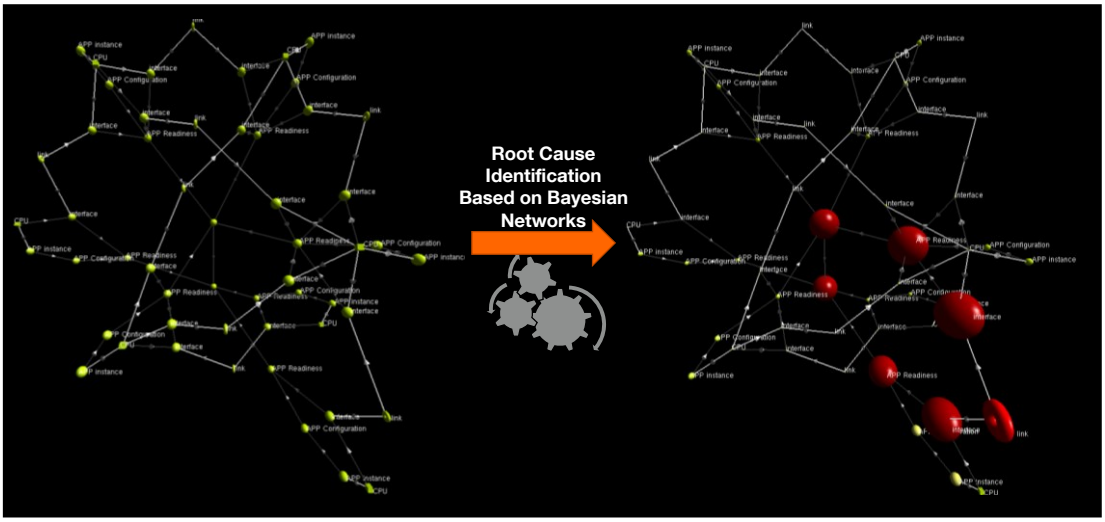


Figure 110. Root Cause Identification process

Secondly, the RCA is triggered and the root cause is calculated by propagating those network observations. As it can be seen in Figure 110, the RCA provides with the same network dependency graph but the vertices are coloured and resized according to their a posteriori probability calculated by the RCA for each network component. For instance, the redder a vertex and the bigger is, the higher its root cause probability is, contrarily, the greener a vertex and the smaller is the lower its a posteriori probability is. Figure 110 shows how the root cause focuses on a subset of the network components, depicted in red and bigger sizes, with respect to the rest of network components, which become smaller and greener.

The most important advantage of our finer-grained templates is that those help identifying the root cause with finer granularity, by identifying faults at a component level.

We describe here two possible faults: one on a data link between a switch and a host (Figure 111 (a)) and one on the SDN controller application (Figure 111 (b)).

In the faulty link between the switch Ms2 and the host H2, the RCA provides with two different resolutions, at a node-level, where the most probable root causes are both nodes and the link interconnecting them, and at component-level, where the most probable root causes are the respective nodes' interfaces.

In the faulty SDN controller application, the RCA provides with two different resolutions also, at a node-level, where the most probable root causes is the SDN controller, and at component-level, where the most probable root cause is its respective application, and its configuration, having completely discarded the physical components of the SDN controller as it responds to ping requests.

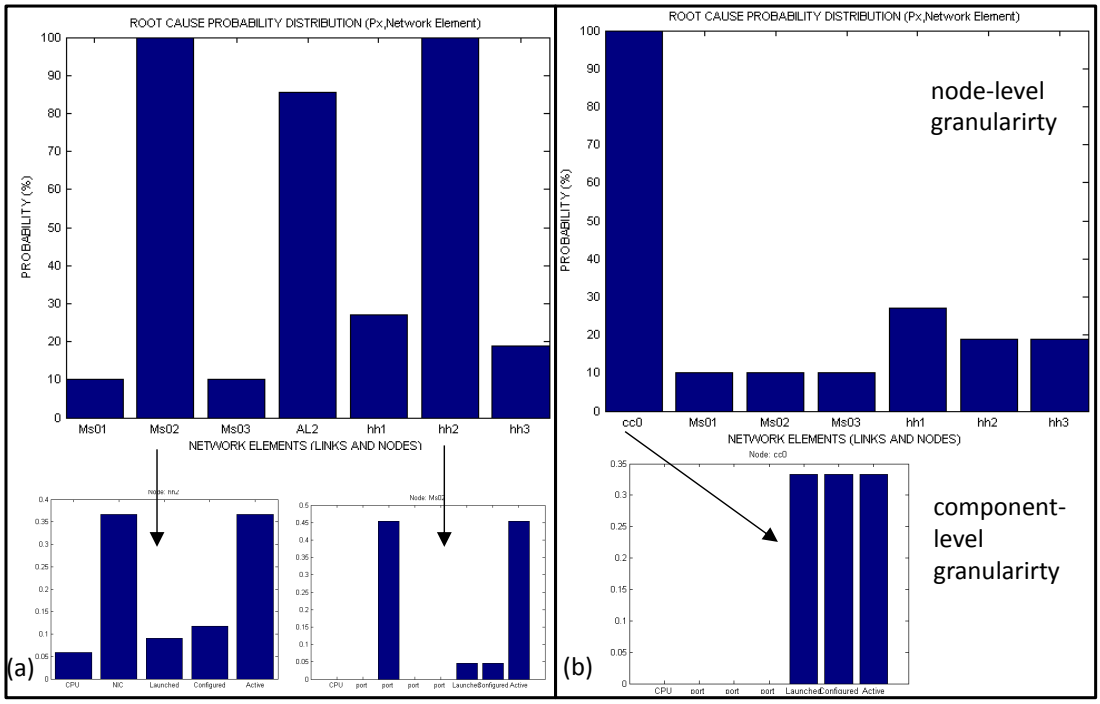


Figure 111. Finer-Granularity of the Root Cause Identification until component level: (a) faulty link, (b) faulty controller

5.4.6 Update of the dependency graph

When there are changes on the network topology, the self-modelling algorithm is triggered to regenerate and update the network dependency graph. This triggering of the self-modeling algorithm is based on a change detector, which utilizes a comparator, previously shown in chapter 4, that stores a previous snapshot of the network topology (reference snapshot) and periodically monitors for network topology changes to be compared with the reference snapshot.

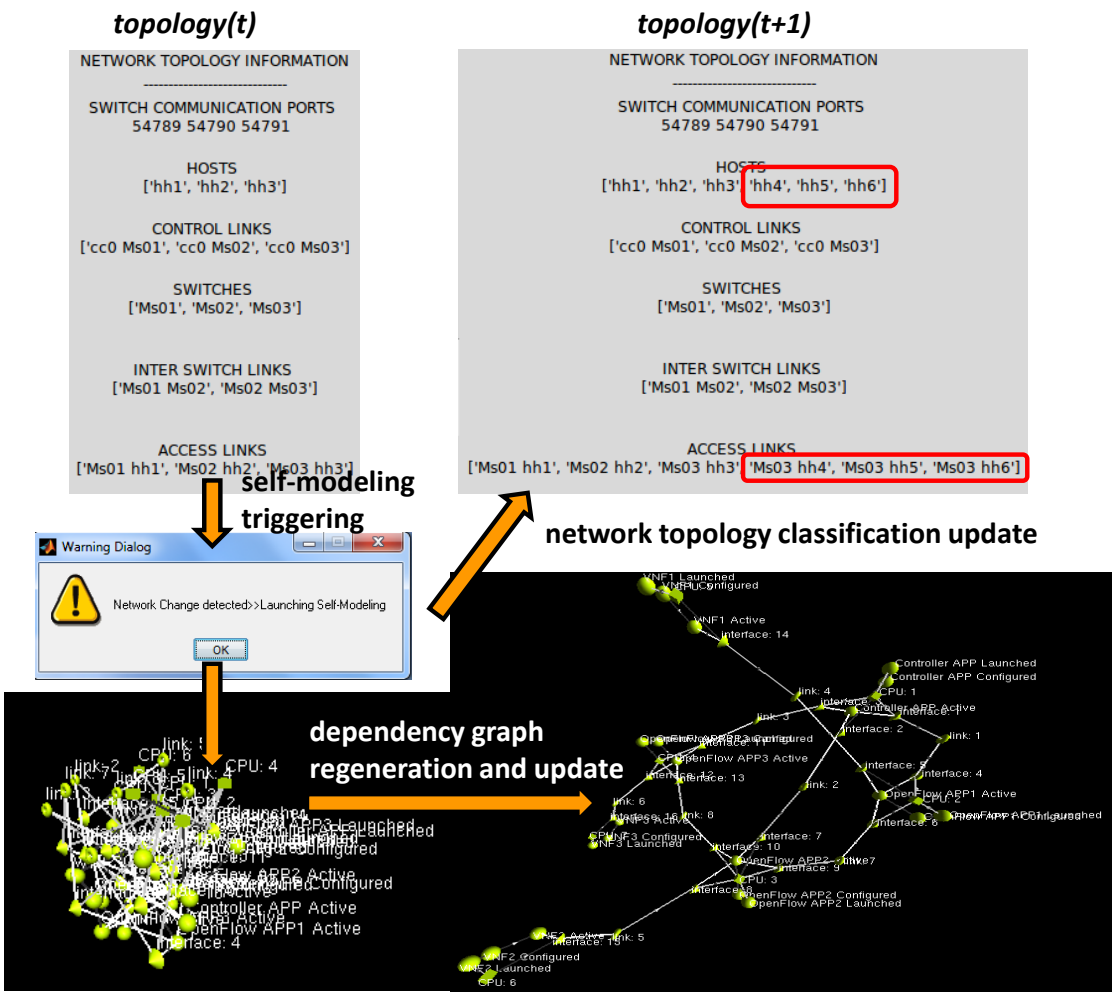


Figure 112. Network dependency graph regeneration and update process

Figure 112 shows the regeneration and update of the network dependency graph.

- Firstly, the network topology at instant t is updated by the SDN controller
- Secondly, the topological information is converted into a machine-readable format file with the updated classified network elements received by the GUI
- Thirdly, the GUI triggers the self-modeling algorithm automatically (shown in Figure 112 in a red square) and it sends this information to the self-modeling block to generate the network dependency graph by showing a message “Network Change Detected> Launching Self-Modeling”.
- Finally, the 3-D network dependency graph is first regenerated and then updated by including those newly added elements in the network topology at instant t+1 found in the machine-readable descriptor

We evaluate here how the RCA module updates the root cause due to topological changes in the network infra-structure with two different use cases. On both use cases, the application consists of a video streaming service where a video is first sent to one single client and then to another new streaming clients that join the streaming service (client 2, client 3, and client 4) with the subsequent topological change.

In the first use case, there is a faulty link which is being diagnosed and a sudden network topology change occurs during the diagnosis. In this case, the network topology is updated and the root cause analysis now includes in the diagnosis those newly added nodes, but the faulty link remains there. The root cause analysis should be consistent enough to pinpoint the same elements as previously to the network change. Now it determines as root causes (root causes: Ms2-switch, AL2-link,hh2-client1), the same as before the change. However, when new

nodes are included in the diagnosis with the same network observations, the uncertainty will be higher as seen in Figure 113 due to the changes in the a posteriori probability distribution.

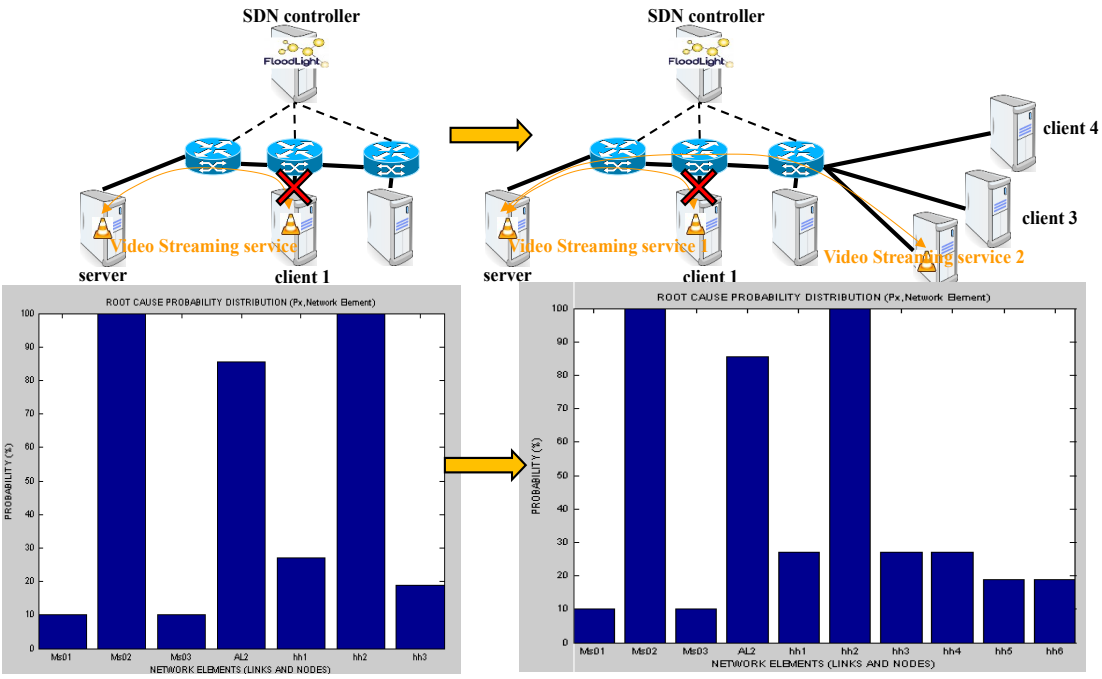


Figure 113. Case 1: Persistence of the root cause calculation with topological changes

In the second use case, there is a faulty link which is being diagnosed and eventually repaired. Later on, a sudden network topology change occurs after the diagnosis was ended, and the self-modeling algorithm regenerates the network dependency graph including the new elements. It is at this time when a different link from the previously diagnosed becomes faulty. In this case, the root cause analysis now includes in the diagnosis those newly added nodes but the network observations are different as the fault is different. The root cause analysis should be updated in two senses: first to include the new elements found in the topology but also to pinpoint the actual root cause (root causes: Ms3-switch, AL6-link, hh6-client2).

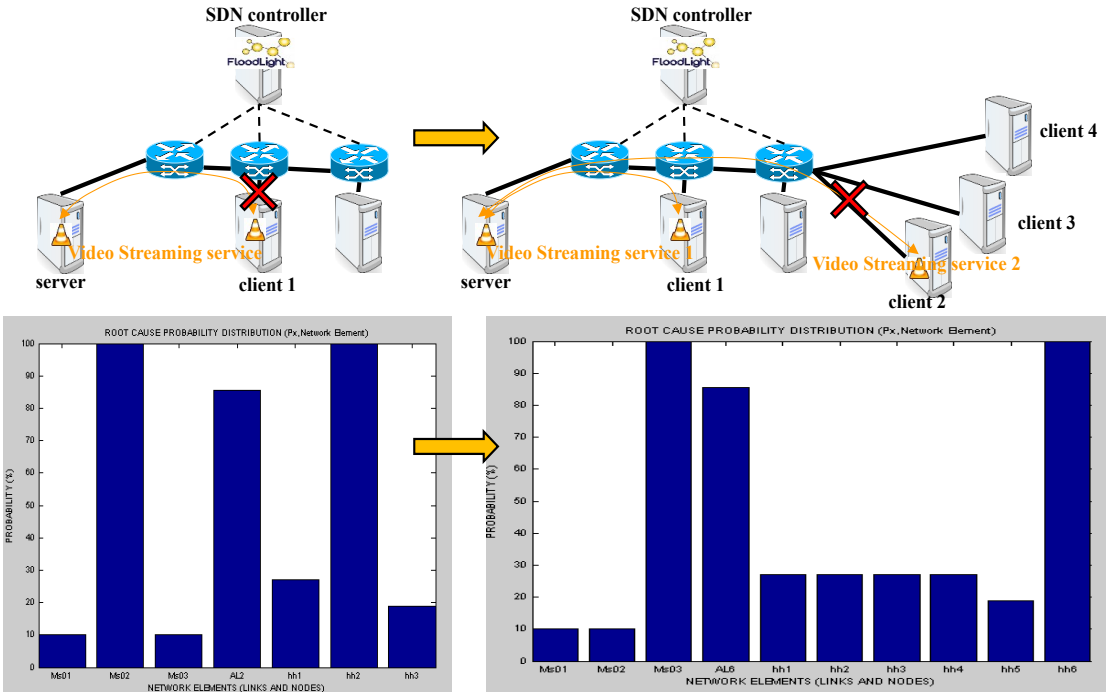


Figure 114. Case 2: Update of the root cause calculation with topological changes

5.4.7 Recovery actions

The recovery block is based on a set of predefined actions to recover the network. Once the RCA gives us the list of root causes and associated probability, the root cause list is sorted according to the probability and the N most probable root causes are selected. Each root cause in this filtered list is associated with a different recovery action. For instance, for recovering a NIC (Network Interface Card), we set the command 'ipconfig eth0 up', or for recovering a link between two network nodes by a forwarding action where this link is avoided or replacing it.

We inject link failures in the network through the Mininet CLI with the command ***link node1 node2 down***. The RCA can suggest recovery actions by setting the command ***link node1 node2 up.***

We inject failures in the SDN controller application by shuting it down, and the RCA suggests as recovery action to launch a new instance of the SDN controller.

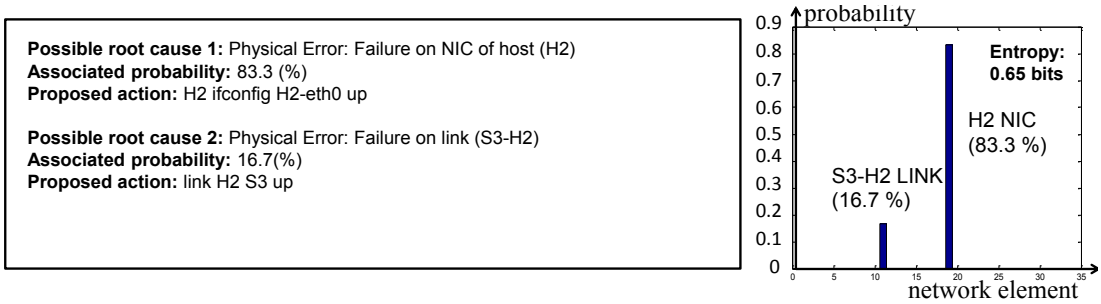


Figure 115. Recovery action suggestion according to the calculated root cause

Once the recovery actions are set into the network, we can measure how long it takes the self-healing module to restore a given failure. We evaluate how fast the re-establishment of the traffic is in the network composed of two clients (Client1 and Client2) and one server delivering that content, which is shown in Figure 116.

We consider two use cases, one faulty link (a) and one faulty SDN controller application (b).

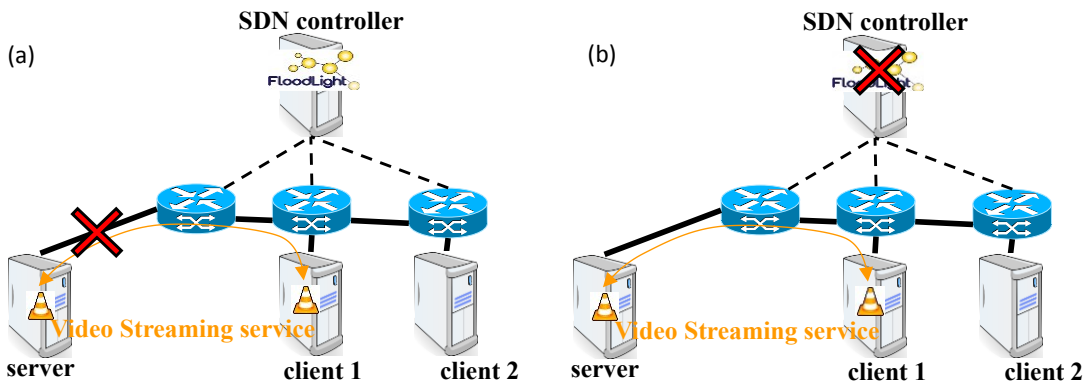


Figure 116. Service restoration in two different cases: (a) faulty link, (b) faulty SDN controller application

Service restoration due to faulty link: Figure 117 shows the traffic (measured in packets) seen by the interface of of the streaming client client1. It can be seen that the first time a fault is injected, it takes 6 seconds the self-healing systemto restablish the service, while it takes 3 seconds, and 1 second afterwards. This value is highly oscilating as it depends on many factors, the time it takes the self-modeling and the RCA algorithms to suggest the action, or the time taken to detect the alarm. The measured window is 314 seconds and the time during which the streaming application is under failure is 6+3+1+1 = 11 seconds, giving an availability factor in this concrete example of (314-11)/314= 96,5.

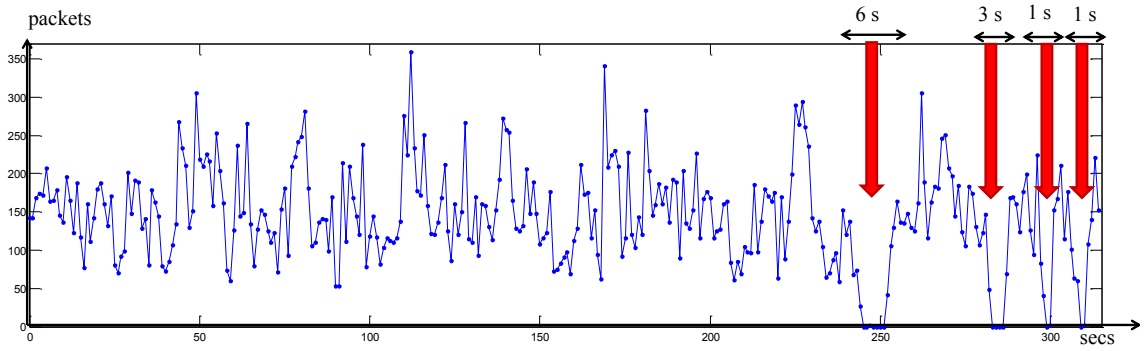


Figure 117. Fault injection and unavailabililty on streaming application due to faulty link

Service restoration due to faulty SDN controller application: Figure 118 shows the traffic (measured in packets) seen by the interface of of the streaming client client1 under a faulty SDN controller application. When we inject a fault in the SDN controller application, it takes 9 seconds the self-healing system to restablish the service. In this case the service is restablished by launching another SDN controller instance. This value is also highly oscilating as it depends on the aforementioned factors but also on the time it takes the detection block to detect that the SDN controller is under failure. The measured window in this case is smaller, 190 seconds and the time during which the streaming application is under failure is 9 seconds, giving an availability factor in this concrete example of (190-9)/190= 95,3.

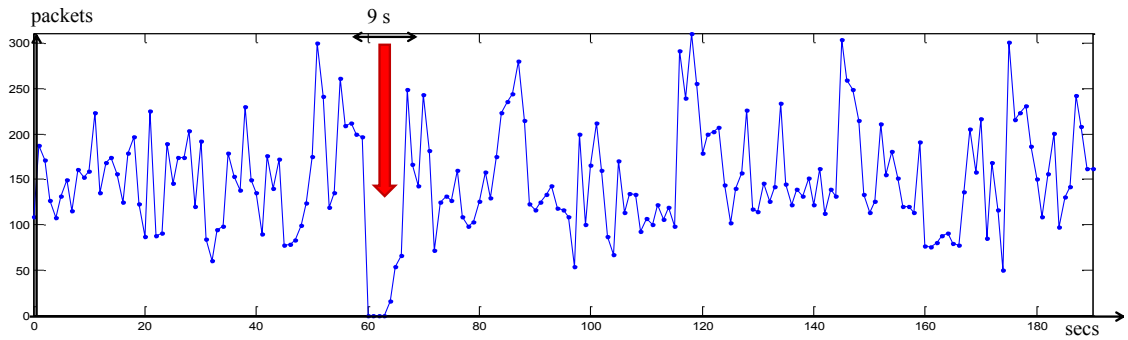


Figure 118. Fault injection and unavailabililty on streaming application due to faulty SDN controller application

Both calculated unavailability values are just given as example, but the important point here is the capability to recover rapidly. We have seen that the self-healing system can provide predefined recovery actions the the entire self-healing loop (detection, self-modeling, RCA, recovery) is under 10 seconds. As conclusion, these recovery times are at the scale of seconds, which affects the unavailability of the streaming application. However, in SDN based infrastructures, which operate at a ms scale, those unavailability values are still so high.

5.5 Conclusion

This chapter evaluated the multi-layer self-diagnosis framework capable of diagnosing faults in programmable networks with SDN and NFV, while taking into account the networking service, virtual, logical, and physical layers.

We evaluated the topology-aware self-diagnosis approach by diagnosing faults in the SDN controller and on both the control and data links. We also evaluated how the RCA can adapt the root cause proposed by taking into account dynamic information such as the CPU load on the nodes in the SDN infrastructure. However, we saw that the uncertainty on the root cause was high due to the finer-granularity of our proposed templates and the consideration of the whole network infrastructure in the diagnosis.

We evaluated the service-aware self-diagnosis approach by diagnosing two network services in two different situations, while applying two different RCA strategies. Both RCA strategies helped reduce the uncertainty on the root cause of our previous topology-aware approach due to the capability to adapt the dependency graph with the appropriate information such as the portion of the network topology or the set of appropriate services.

We evaluated the self-healing framework based on the topology-aware self-diagnosis to show how an automated diagnosis can support self-healing to recover both a streaming service and its underlying SDN infrastructure. However, we also saw that the time taken to the self-healing testbed to recover the streaming application

We found the following limitations on the tests made:

The topologies analyzed in this thesis are Linear and tree topologies with different number of network elements. However, there are far more complex topologies in data centers such as fat-tree or other topologies.

The parameters detected and fed to the self-modelig module are all the elements in the network topology, including the control plan resources. However, no traffic is measured to include in the model the state of a link. The detection is based on the information seen by the SDN controller, but other in a real network many other sources of information are available and should be integrated in this diagnosis module.

The number of controllers in the topology has always been one main controller. However, in real infrasructures there a number of them to ensure the control of the network is maintiained at all times. In this case, the connections between each controller and its set of switches has to be detected and included in the network graph. This has to include the type of control led by each of thes controllers.

The root cause analyzes has analyzed so far mainly unavailability on the networking services or in the underlying infrastructure, but not degradations. Nevertheless, the automated construction of this graph will serve as a base to understand how degradations can propagate in the network and eventually lead to service abnormalities.

In terms of performance, the self-modeling approach takes a long time with respect to the reconfiguration changes in SDN infrastructures, which can reach milliseconds. This is mainly due to the large topologies analysed. In chapter 4 we tried to address this issue by exploring part of the network topology instead of the whole topology. However, a possibility is to only diagnose a subset of the network and the appropriate dependencies with respect to the alarm received by the diagnosis engine.

Chapter 6 Conclusions and Future work

6.1 Conclusions

The chapter 2 describes the context of this thesis, programmable networks. It gives an insight on SDN and NFV, its faut management challenges and related work. This chapter motivates the resilience needs of combined SDN and NFV infrastructures and it describes the research challenges to conceive a self-diagnosis mechanism to cope with the high dynamicity of those infrastructures.

The chapter 3 details self-healing systems, as the autonomic mechanism able to introduce autonomic priniciplies in programmable networks. We survey the state of the art on self-healing systems, the related work on the algorihtms put in place, focusing especially on the self-diagnosis algorithms such as model-bbased approaches like Bayesian Networks, due to its many advantageous properties and use on the network diagnosis context.

The chapter 4 describes the core of this thesis, a cross-layered self-modeling based self-diagnosis approach that automates the diagnosis of programmable networks. This cross-layered self-diagnosis approach takes into account the topological changes in the infrastructure, the type of control led by the control plane, the updates in the forwarding flows, placement of the virtual links and VNFs composing the networking services deployed over the infrastructure. This cross-layered self-diagnosis approach is based on a topology-aware self-modeling approach and a service-aware self-modeling approach. The topology-aware self-modeling approach builds automatically and at runtime the fault propagation model by assembling a set of multi-layered, fine-granular, machine-readable, extendable templates containing the resources to supervise at physical and logical layers. The service-aware self-modeling approach is an extension of the topology-aware self-modeling to diagnose networking services, by including in the diagnosis the virtual and service layers that generates on-the-fly the diagnosis model that includes the physical, logical, and the virtual dependencies of networking services in combined SDN and NFV infrastructures. This cross-layered self-diagnosis approach is suitable to any network topology and any control type in SDN. In addition, it is independent from the controller implementation (e.g. Floodlight or OpenDaylight) because it directly interacts with the SDN controller through its northbound API interface, which introduces a high degree of abstraction with respect to the underlying southbound protocol.

The chapter 5 evaluates the cross-layered self-diagnosis approach. It first shows how it can automatically generate the fault propagation model for several network topologies, types of control and different networking services composed of different chains of VNFs. It also evaluates several faults at the control plane, the data plane, and the networking services deployed over different network topologies. It also evaluates the scalability of the self-diagnosis approach, by evaluating the time taken to generate the fault propagation model as a function of the network topology size and the number of services included in the diagnosis. IT finally evaluates a self-healing mechanism that can suggest predefined recovery actions based on the self-diagnosis approach.

In conclusion, we investigated self-healing properties in order to apply them into programmable networks to provide them with resilience properties so needed in such centralized networks. We proposed a self-diagnosis

framework to diagnose automatically and on-the-fly SDN and NFV combined infrastructures. This framework is based on a self-modeling methodology that generates at runtime a fault propagation graph to describe how faults and failures propagate in such dynamic multi-layer SDN and NFV infrastructure. This model contains the different dependencies among the resources located at physical, logical, virtual and service layers and it is exploited by the root cause analysis to find the root cause. This approach is cornerstone as demonstrated in the thesis to suggest recovery actions based on the calculated root causes at data, control, as ell as application plane.

6.2 Future work

This thesis points out several research directions worth being explored in the future. We structure these research directions in four: proactive self-diagnosis techniques, machine-learning techniques, observability techniques and extension of the current fault propagation model.

6.2.1 Proactive self-diagnosis techniques to avoid service failures

In the thesis we have focused on reactive cases where the faults and failures to find the root causes. However, there are other cases where degradations may end up with the failure of the system or the services deployed. Proactive self-diagnosis mechanisms are an interesting research direction, because those mechanisms are able to evaluate the impact of degradations in network resources such as CPU load and throughput on the VNFs and networking services to predict future failures. It could also include metrics related to SDN and NFV layers to prevent future malfunctions by monitoring degradations and it will integrate the defined metrics for SDN in the self-diagnosis framework to help predicting future degradations due to congestions on the control-to-data interfaces.

6.2.2 Exploration of machine-learning techniques

We identify machine-learning algorithms and techniques to introduce intelligence on the recovery actions suggested by a self-healing system, instead of predefined actions. Those intelligent recovery actions can comprise reconfiguration orders, swapping mechanisms for the controller, alternative forwarding for OpenFlow-enabled switches or load balancing on access points.

We also identify machine-Learning algorithms to automarically learn the template of a new node in the fly, without having to predefine its template. This means to compute the inner dependencies of a newly added node by analyzing its output and input data.

6.2.3 Observability and detection techniques

Another interesting research direction is the conception of intelligent detection techniques to cover two major aspects:

-Automatically adapt the diagnosis to the appropriate and relevant network segment and services affected, instead of considering the whole network topology, which leads as a result to high uncertainty.

-Discern between topological changes and malfunctions. A typical example is when a malfunction occurs in a given link, which can be seen as a topological change (the node and the link attached disappears) and the diagnosis module would include the impacted link in the diagnosis.

6.2.4 Extension of the fault propagation model

The granularity of our diagnosis model considers that the VNFI is embedded in a host, representing the dependencies between the embedding host and the VNFI, as well as it considers the three states of a VNFI (instantiated, configured, and active). However, it could be possible to extend this fault propagation model to make it more accurate and diagnose other malfunctions in additional components such as the hypervisor, VM or other additional layers. Concretely, we identify the following extensions of our fault propagation model:

-Inclusion of the type of the internal components inside a host such as the hypervisor, and the VM(s) composing the VNFI

-Inclusion of the VNFC (VNF components) composing a VNF and the dependencies of a given VNF with the underlying VNFC.

-Inclusion of the electrical network supporting the network infrastructure

-Inclusion of the dynamic interactions between control and data planes that may lead to glitches

Bibliography

(ONF, 2011) ONF, OpenFlow Switch specification 1.1.0 [Online]. Available at: http://archive.openflow.org/documents/openflow-spec-v1.1.0.pdf

(ONF, 2016) ONF, SDN architecture https://www.opennetworking.org/images/stories/downloads/sdn-resources/technical-reports/TR-521_SDN_Architecture_issue_1.1.pdf

(Kephart & Chess, 2003) J. Kephart, D. Chess, The Vision of Autonomic Computing, IEEE Computer Society. 2003

(Salehie & Tahvildari, 2009) M. Salehie, L. Tahvildari, Self-adaptive software: landscape and research challenges. ACM Trans Autonomous Adaptive Systems 4(2):1–42. 2009

(Dijkstra, 1974) EW. Dijkstra, "Self-stabilizing systems in spite of distributed control". Commun ACM 17(11): 643–644.107. 1974

(Sterbenz et al, 2010) J. P. G. Sterbenz, D. Hutchison, et. al. "Resilience and survivability in communication networks: Strategies, principles, and survey of disciplines". Comput. Netw. 54, 8 (June 2010), 1245-1265.

(Psaier & Dustdar, 2011) H. Psaier, S. Dustdar, "A survey on Self-Healing systems: approaches and systems". Computing 91, 1, 43-73. 2011

(Gosh et al, 2007) D. Ghosh, R. Sharman, H. Raghav Rao, and S. Upadhyaya, "Self-Healing systems—survey and synthesis". Decis Support Syst 42(4):2164–2185. 2007

(Russell & Norvig, 2003) S. Russell and P. Norvig, "Artificial Intelligence: A Modern Approach", 2nd ed. Prentice Hall. 2003.

(Dobson & Denazis, 2009) S. Dobson, S. Denazis, S. et. al. "A Survey of Autonomic Communications. ACM Transactions on Autonomous and Adaptive Systems, Vol.1, No.2, Pages 223-259. 2009

(Nagpal et al, 2003) R. Nagpal, A. Kondacs, C. Chang, Programming methodology for biologically-inspired self-assembling systems, AAAI Symposium. 2003.

(Kliger et al, 1997) S. Kliger, S. Yemini, Y. Yemini, D. Ohsie, and S. Stolfo. "A coding approach to event correlation". In Intelligent Network Management (IM). 1997

(Huebscher & McCann, 2008) MC. Huebscher, JA. McCann, "A survey of autonomic computing: degrees, models, and applications". ACM Computing Survey 40(3):1–28. 2008

(Salehie & Tahvildari, 2005) M. Salehie and L. Tahvildari, "Autonomic computing: emerging trends and open problems". SIGSOFT Software Engineering Notes 30(4):1–7. 2005

(Avizienis et al, 2004) A. Avizienis, J.-C. Laprie, B. Randell, C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," in *Dependable and Secure Computing, IEEE Transactions on* , vol.1, no.1, pp.11-33. 2004

(Singh & Graepel, 2013) S. Singh and T. Graepel, "Automated probabilistic modelling for relational data", in CIKM, pages 1497–1500, 2013.

(Kabli et al, 2007) R. Kabli, F. Herrmann, J. McCall, "A Chain-Model Genetic Algorithm for Bayesian Network Structure Learning," GECCO. 2007

(Fenz, 2011) S. Fenz, "An ontology and bayesian-based approach for determining threat probabilities," In Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security, pages 344–354. 2011.

(Rish,_) I. Rish, "A Tutorial on Inference and Learning in Bayesian Networks", IBM T.J. Watson Research Center, [Online], Available at: http://www.ee.columbia.edu/~vittorio/lecture12.pdf

(Skiena, 1990) S. Skiena, "Topological Sorting." §5.4.3 in Implementing Discrete Mathematics: Combinatorics and Graph Theory with Mathematica. Reading, MA: Addison-Wesley, pp. 208-209. 1990.

(Tipper, 2013) D. Tipper, “Resilient Network Design: Challenges and Future Directions”. Telecommunication Systems, 56 (1), pp. 5.

(Cholda et al, 2007) P. Cholda, A. Mykkeltveit, B.E. Helvik, O.J. Wittner, A. Jajszczyk, "A survey of resilience differentiation frameworks in communication networks," Communications Surveys & Tutorials, IEEE , vol.9, no.4, pp.32,55, Fourth Quarter 2007.

(Butler & Keselj, 2010) M. Butler and V. Keselj. In Proceedings of Canadian AI'2010, Ottawa, ON, Canada,, volume LNAI 6085 of Lecture Notes in Computer Science, Springer pp. 366-369. 2010.

(Alharbi et al, 2015) T. Alharbi, M. Portmann and F. Pakzad, "The (in)security of Topology Discovery in Software Defined Networks," Local Computer Networks (LCN), 2015 IEEE 40th Conference on, Clearwater Beach, FL, 2015, pp. 502-505.

(Heller et al, 2012) B. Heller, R. Sherwood, and N. McKeown. “The controller placement problem”. In Proc. 1st workshop on Hot topics in software defined networks, ACM HotSDN ’12, pages 7–12, New York, NY, USA. 2012

(Curtis et al, 2011) A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee. “Devoflow: scaling flow management for high-performance networks”. SIGCOMM Comput. Commun. Rev., 41(4):254–265. 2011.

(Panda et al, 2013) A. Panda, C. Scott, A. Ghodsi, T. Koponen, and S. Shenker, “CAP for Networks”. In Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking, HotSDN. 2013

(Sharma et al, 2013a) S. Sharma, D. Staessens, D. Colle, M. Pickavet, P. Demeester, "Fast failure recovery for in-band OpenFlow networks," in Design of Reliable Communication Networks (DRCN), 2013 9th International Conference on the, vol., no., pp.52-59, 4-7 March 2013

(Behesti & Zhang, 2012) N. Beheshti, Y. Zhang, "Fast failover for control traffic in Software-defined Networks," in Global Communications Conference (GLOBECOM), 2012 IEEE , vol., no., pp.2665-2670. 2012

(Li et al, 2014) H. Li, P. Li, S. Guo and A. Nayak, "Byzantine-Resilient Secure Software-Defined Networks with Multiple Controllers in Cloud," in IEEE Transactions on Cloud Computing, vol. 2, no. 4, pp. 436-447, Oct.-Dec. 1 2014.

(Yazici et al, 2014) V. Yazıcı, M. Oğuz Sunay, Ali Ö. Ercan. “Controlling a Software-Defined Network via Distributed Controllers”. Özyeğin University, Istanbul, Turkey. 2014

(ETSI NFV, 2012) ETSI NFV Group Specification: “Network Functions Virtualisation (NFV); Virtual Network Functions Architecture”, Dec. 2012

(ETSI NFV, 2014) ETSI NFV Group Specification: “Network Functions Virtualisation (NFV); Management And Orchestration”, Dec. 2014

(ETSI NFV, 2015a) ETSI NFV Group Specification Draft: “Network Functions Virtualisation (NFV); Ecosystem; Report on SDN Usage in NFV Architectural Framework”, Sept. 2015.

(ETSI NFV, 2015b) ETSI NFV Group Specification: “Network Functions Virtualisation (NFV); Resiliency Requirements”, Jan. 2015

(Guerzoni et al, 2012) R. Guerzoni et. al., “Network Functions Virtualisation: An Introduction, Benefits, Enablers, Challenges and Call for Action. Introductory white paper,” in SDN and OpenFlow World Congress, June 2012.

(Kreutz et al, 2015) D. Kreutz, F.M.V. Ramos, P. Esteves Verissimo, C. Esteve Rothenberg, S. Azodolmolky, S. Uhlig, "Software-Defined Networking: A Comprehensive Survey," Proceedings of the IEEE , vol.103, no.1, pp.14,76. 2015.

(Ma et al, 2014) Y. Ma, A. AuYoung, S. Banerjee, J. Lee, and P. Sharma, "Automatic resolution of dynamic resource conflicts between SDN applications"

(AuYoung et al, 2014) A. AuYoung, Y. Ma, S. Banerjee, J. Lee., et. al.," Democratic Resolution of Resource Conflicts Between SDN Control Programs", Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies, Sydney, Australia. 2014

(Chickering et al, 1994) D. M. Chickering, D. Geiger, and D. Heckerman. "Learning bayesian networks is np-hard". Technical report, Technical Report MSR-TR-94-17, Microsoft Research Technical Report. 1994.

(Robinson, 1977) R. W. Robinson. "Counting unlabeled acyclic digraphs". In C. H. C. Little, editor, Combinatorial Mathematics V, volume 622 of Lecture Notes in Mathematics, pages 2843, Berlin. 1977.

(Messaoud et al, 2009a) M. B. Messaoud, P. Leray, and N. Ben Amor. "Semcado: a serendipitous strategy for learning causal bayesian networks using ontologies". In Proceedings of the 11th European Conference on Symbolic and Quantitative Approaches to Reasoning with Uncertainty, pages 182–193. 2009

(Messaoud et al, 2009b) M. B. Messaoud, P. Leray, and N. Ben Amor, "Integrating ontological knowledge for iterative causal discovery and visualization". In ECSQARU'09, pages 168-179,hal-00596260,2009

(Friedman et al, 1999) N. Friedman et. al. , "Learning probabilistic relational models", Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI-99), Stockholm, Sweden, August 1999

(Tjoa, 2009) A. M. Tjoa, "Ontology-based generation of Bayesian networks", CISIS 09

(Wallin, 2012) S. Wallin, "Rethinking Network Management: Models, Data-Mining and Self-Learning", Ph.D dissertation, Dept. Comp. Sciencies, Luleå, Sweden, 2012.

(IUT-T, 2001) FCAPS Management Framework: ITU-T Rec. M. 3400, available at: http://www.itu.int/rec/T-REC-M.3400-200002-I

(Ganek & Corbi, 2003) A.G. Ganek and TA. Corbi, "The dawning of the autonomic computing era". IBM Syst J 42(1):5–18. 2003

(Arora & Gouda, 1993) A. Arora and M. Gouda, "Closure and convergence: a foundation of fault-tolerant computing". IEEE Trans Softw Eng 19(11):1015–102. 1993

(Paladi, 2015) N. Paladi, "Towards secure SDN policy management. In: 1st International Workshop on Cloud Security and Data Privacy by Design". 2015

(Steinder & Sethi, 2002) M. Steinder and A. S. Sethi. "End-to-end Service Failure Diagnosis Using Belief Networks". In Network Operations and Management Symposium, NOMS 2002, pages 375-390, 2002

(Bennacer et al, 2012) L. Bennacer, L. Ciavaglia, et.al., "Optimization of fault diagnosis based on the combination of Bayesian Networks and Case-Based Reasoning," in NOMS, 2012 IEEE , vol., no., pp.619,622, 16-20 April 2012.

(Bennacer et al, 2013) L. Bennacer, L. Ciavaglia, et. al. "Scalable and fast root cause analysis using inter cluster inference," in Communications (ICC), 2013 IEEE International Conference on , vol., no., pp.3563-3568, 9-13 June 2013

(Bennacer et al, 2015) L. Bennacer, Y. Amirat, A. Chibani, A. Mellouk, L. Ciavaglia, "Self-Diagnosis Technique for Virtual Private Networks Combining Bayesian Networks and Case-Based Reasoning," in Automation Science and Engineering, IEEE Transactions on , vol.12, no.1, pp.354-366, 2015

(Zhao, 2008) Y. Zhao “Towards Noise-Tolerant Network Service Diagnosis”. EECS Department. Northwestern University. SIGMETRICS 2008

(Steinder & Sethi, 2004) M. Steinder, A.S. Sethi, "Probabilistic fault localization in communication systems using belief networks," in Networking, IEEE/ACM Transactions on , vol.12, no.5, pp.809-822, Oct. 2004

(Mengshoel et al, 2010) O.J. Mengshoel, M. Chavira, K. Cascio, S. Poll, A. Darwiche, S. Uckun, "Probabilistic Model-Based Diagnosis: An Electrical Power System Case Study," in Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on , vol.40, no.5, pp.874-885. 2010

(Tembo et al, 2015) S.R.Tembo, J.L. Courant, S. Vaton, "A 3-layered self-reconfigurable generic model for self-diagnosis of telecommunication networks," in SAI Intelligent Systems Conference (IntelliSys), 2015 , vol., no., pp.25-34, 10-11. 2015

(Yunkhao et al, 2010) L. Yunhao, K. Liu; M. Li, "Passive Diagnosis for Wireless Sensor Networks," in Networking, IEEE/ACM Transactions on , vol.18, no.4, pp.1132-1144, Aug. 2010

(Al-Jawad et al, 2015) A. Al-Jawad, R. Trestian, P. Shah, O. Gemikonakli, "BaProbSDN: A probabilistic-based QoS routing mechanism for Software Defined Networks," in *Network Softwarization (NetSoft), 2015 1st IEEE Conference on* , vol., no., pp.1-5, 13-17 April 2015

(Bahl & Chandra, 2007) P. Bahl, R. Chandra, et. al., “Towards highly reliable enterprise networking services via inference of multi-level dependencies,” in SIGCOMM, 2007.

(Hounkonnou, 2013) C. Hounkonnou, “Active Self-Diagnosis in Telecommunication Networks”. PhD thesis. Université de Rennes 1. July 2013.

(Fonseca et al, 2012) P. Fonseca, R. Bennesby, E. Mota and A. Passito, "A replication component for resilient OpenFlow-based networking," Network Operations and Management Symposium (NOMS), 2012 IEEE , vol., no., pp.933,939, 16-20 April 2012

(Mijumbi et al, 2015) R. Mijumbi, et.al., "Network Function Virtualization: State-of-the-art and Research Challenges," in Communications Surveys & Tutorials, IEEE , vol.PP, no.99, pp.1-1. 2015

(Esteves et al, 2013) R.P. Esteves, L.Z. Granville, R. Boutaba, "On the management of virtual networks," in Communications Magazine, IEEE , vol.51, no.7, pp.80,88, July 2013.

(Chowdhury & Boutaba, 2009) N.M.M.K. Chowdhury, R. Boutaba, "Network virtualization: state of the art and research challenges," in Communications Magazine, IEEE, vol.47, no.7, pp.20-26, July 2009.

(Kandula & Mahajan, 2010) S. Kandula, R. Mahajan, et. al, “Detailed diagnosis in enterprise networks,” in SIGCOMM, 2010.

(Scholler et al, 2013) M. Scholler et. al., "Resilient deployment of virtual network functions," in Ultra Modern Telecommunications and Control Systems and Workshops (ICUMT), 2013 5th International Congress on , vol., no., pp.208-214, 10-13 Sept. 2013

(Miyazawa et al, 2015) M. Miyazawa et.al., "vNMF: Distributed fault detection using clustering approach for network function virtualization," in Integrated Network Management (IM), 2015 IFIP/IEEE International Symposium on , vol., no., pp.640-645, 11-15 May 2015

(Smith et al, 2011) P. Smith, D. Hutchison, J.P.G. Sterbenz, M. Schöller, A. Fessi, M. Karaliopoulos, L. Chidung , B. Plattner, "Network resilience: a systematic approach," Communications Magazine, IEEE , vol.49, no.7, pp.88,97. 2011

(Hamerly & Elkan, 2001) G. Hamerly and C. Elkan. “Bayesian approaches to failure prediction for disk drives”. In Proceedings of the Eighteenth International Conference on Machine Learning. Morgan Kaufmann Publishers Inc., 202–209. 2001

(Salfner et al, 2010) F. Salfner, M. Lenk and M. Malek , "A survey of online failure prediction methods" , ACM Comput. Surv. , vol. 42 , pp.10:1 -10:42, 2010

(Hyojoon et al, 2012) K. Hyojoon et al., “CORONET: Fault tolerance for Software Defined Networks", Network Protocols (ICNP), 2012 20th IEEE International.

(Basil et al, 2015) A. Basil, B. Vengainathan, V. Manral, M. Tassinari, and S. Banks, “Benchmarking Methodology for SDN Controller Performance.” [Online]. Available at: https://tools.ietf.org/html/draft-bhuvan-bmwg-sdn-controller-benchmark-meth-01.

(Sharma et al, 2013b) P. Sharma, S. Banerjee, S. Tandel, S.; et. al., "Enhancing network management frameworks with SDN-like control," in Integrated Network Management (IM 2013), 2013 IFIP/IEEE International Symposium on , vol., no., pp.688-691. 2013

(Scott et al, 2014) C. Scott, A. Wundsam, B. Raghavan, A. Panda, A. Or, J. Lai, E. Huang, et. al. “Troubleshooting blackbox SDN control software with minimal causal sequences”. In SIGCOMM, pages 395–406, August 2014.

(Canini et al, 2012) M. Canini, D. Venzano, et. al., “A NICE way to test OpenFlow applications,” in Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation, ser. NSDI’12. Berkeley, CA, USA: USENIX Association, 2012, pp. 10–10.

(Handigol et al, 2012) N. Handigol, B. Heller, V. Jeyakumar, D. Mazieres, and N. McKeown,“Where is the debugger for my software-defined network?” in Proceedings of the First Workshop on Hot Topics in Software Defined Networks, ser. HotSDN ’12. New York, NY, USA: ACM, 2012, pp. 55–60. 2012

(Wundsam et al, 2011) A. Wundsam, D. Levin, S. Seetharaman, and A. Feldmann, “OFRewind: Enabling Record and Replay Troubleshooting for Networks,” in Proc. 2011 USENIX Conference on USENIX Annual Technical Conference, ser. USENIXATC’11. USENIX Association, 2011, pp. 29–29. 2011

(Handigol et al, 2014) N. Handigol, B. Heller, V. Jeyakumar, D. Mazieres, and N. McKeown,“I know what your packet did last hop: Using packet histories to troubleshoot networks,” in 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14). Seattle, WA: USENIX Association, Apr. 2014, pp. 71–85. 2014

(Turchetti & Duarte, 2015) R.C. Turchetti, E. P. Duarte, "Implementation of Failure Detector Based on Network Function Virtualization," in Dependable Systems and Networks Workshops (DSN-W), 2015 IEEE International Conference on , vol., no., pp.19-25, 22-25 June 2015.

(Caini et al, 2011) C. Caini, H. Cruickshank, S. Farrell, M. Marchese, "Delay- and Disruption-Tolerant Networking (DTN): An Alternative Solution for Future Satellite Networking Applications," in Proceedings of the IEEE , vol.99, no.11, pp.1980-1997. 2011

(Khabbaz et al, 2012) M. J. Khabbaz, C.M. Assi, W.F. Fawaz, "Disruption-Tolerant Networking: A Comprehensive Survey on Recent Developments and Persisting Challenges," in Communications Surveys & Tutorials, IEEE , vol.14, no.2, pp.607-640, Second Quarter 2012

(Buchegger & Boudec, 2002) S. Buchegger and J.-Y. L. Boudec, “Performance analysis of the CONFIDANT protocol”, in Proc. 3rd ACM Int. Symp. Mobile Ad Hoc Netw. Comput., Lausanne, Switzerland, 2002, pp. 226–236.

(Hardin, 2002) R. Hardin, “Trust and trustworthiness”, Trust, the Russel Sage Foundation, vol. IV, p. 234, 2002.

(Sen & Dutta, 2002) S. Sen and P. S. Dutta, “The evolution and stability of cooperative traits”, in Proc. 1st Int. Joint Conf. Autonom. Agents Multi-Agent Syst., 2002, pp. 1114–1120.

(Tcholchev et al, 2010) N. Tcholtchev et al., “Scalable Markov chain Based Algorithm for Fault-Isolation in Autonomic Networks”, GLOBECOM 2010.

(Marilly et al, 2002) E. Marilly, et al., “Alarm correlation for complex telecommunication networks using neural networks and signal processing ”, in IP op. and management, pp 3-7, 2002. DOI: 10.1109/IPOM.2002.1045684.

(Parekh et al, 2000) S. Parekh, N. Gandhi, et. al., Using Control Theory to Achieve Service Level Objectives In Performance Management, October 23, 2000.

(Parekh et al, 2003) S. Parekh, et al., “Managing the performance impact of administrative utilities”, in Self-Managing Dist. Systems, Vol. 2867, pp 130-142, 2003.

(Diao & Hellerstein, 2005) Y. Diao, J. L. Hellerstein, et. al., “A Control Theory Foundation for Self-Managing Computing Systems”, 0733-8716, 2005 IEEE.

(Lu et al, 2002) Y. Lu, T. Abdelzaher, et.al., An Adaptive Control Framework for QoS Guarantees and its Application to differentiated Caching Services, University of Virginia, 0-7803-7426-6/02, 2002 IEEE.

(Lu et al, 2001) Y. Lu, A. Saxena and T. F. Abdelzaher, "Differentiated caching services; a control-theoretical approach," Distributed Computing Systems, 2001. 21st International Conference on., Mesa, AZ, 2001, pp. 615-622.

(Padala & Hou, 2009) P. Padala, K. Hou, et. al., “Automated control of multiple virtualized resources”. In: Proc of EuroSys. 2009

(Storm et al, 2006) A. J. Storm, C. Garcia-Arellano, S. Lightstone, Y. Diao, and M. Surendra, ”Adaptive Self-Tuning Memory in DB2”. In Proc. of VLDB Conf., 2006

(Diao et al, 2002) Y. Diao, N. Gandhi, J. L. Hellerstein, S. Parekh and D. M. Tilbury, "Using MIMO feedback control to enforce policies for interrelated metrics with application to the Apache Web server," Network Operations and Management Symposium, 2002. NOMS 2002. 2002 IEEE/IFIP, 2002, pp. 219-234.

(Xiaoyuan et al, 2007) G. Xiaoyuan, J. Strassner, J. Xie, et al. “Autonomic Multimedia Communications: Where Are We Now? “, Proceeding of the IEEE. 2007

(Akhtar, 2016) N. Akhtar, "Managing NFV using SDN and Control Theory", IEEE/IFIP International Workshop on Management of the Future Internet (ManFI 2016)

(Aamodt & Plaza, 1994) A. Aamodt and E. Plaza (1994), “Case-Based Reasoning: Foundational Issues, Methodological Variations, and System Approaches”. AI Communications. IOS Press, Vol. 7: 1, pp. 39-59.

(ENISA, 2016) European Union Agency For Network And Information Security, “Threat Landscape and Good Practice Guide for Software Defined Networks/5G”, available at: https://www.enisa.europa.eu/activities/risk-management/evolving-threat-environment/enisa-thematic-landscapes/sdn-threat-landscape

(Cooper & Herskovits, 1992) G. Cooper and A. Herskovits, “A Bayesian Method for the induction of probabilistic networks from data”, Machine Learning, 9:309-347, 1992

(Georghe et al, 2015) G. Georghe, T. Avanesov, M.-R. Palattella, T. Engel, and Popoviciu, C., "SDN-RADAR: Network troubleshooting combining user experience and SDN capabilities," in Network Softwarization (NetSoft), 2015 1st IEEE Conference on, vol., no., pp.1-5, 13-17 April 2015.

(McCauley, 2012) M. McCauley, “POX,” 2012. [Online]. Available: http://www.noxrepo.

(Krishnaswamy et al, 2013) U. Krishnaswamy, P. Berde, J. Hart, M. Kobayashi, P. Radoslavov, T. Lindberg, R. Sverdlov, S. Zhang, W. Snow, and G. Parulkar, “ONOS: An Open Source Distributed SDN OS,” 2013. [Online]. Available: http://www.slideshare.net/umeshkrishnaswamy/

(Floodlight, 2012) “Floodlight Is A Java-Based OpenFlow Controller,” 2012. [Online]. Available: http://floodlight.openflowhub.org/

(OpenDaylight, 2013) OpenDaylight, “OpenDaylight: A Linux Foundation Collaborative Project,” 2013. [Online]. Available: http://www.opendaylight.org

(Lantz et al, 2010) B. Lantz, B. Heller, and N. McKeown, “A network in a laptop: rapid prototyping for software-defined networks,” in Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks, ser. Hotnets-IX. New York, NY, USA: ACM, 2010, pp. 19:1–19:6.

(Sanchez et al, 2014) J. Sanchez, I. Grida Ben Yahia, N. Crespi, “Self-healing Mechanisms for Software Defined Networks”. AIMS 2014.

(Sanchez et al, 2015) J. Sanchez, I. Grida Ben Yahia, N. Crespi, “Self-Modeling Based Diagnosis of Software-Defined Networks,” Workshop MISSION 2015 at 1st IEEE Conference on Network Softwarization, London, 13-17 April 2015.

(Sanchez et al, 2016) J. Sanchez, I. Grida Ben Yahia, N. Crespi, “Self-Diagnosis of Networking Services over Programmable Networks,” 2nd IEEE Conference on Network Softwarization, Seoul, Korea, 6-10 June 2016.

(Murphy et al., 2001) Kevin Murphy’s Bayesian Networks Toolbox, MIT AI lab,200 Technology Square,Cambridge. [Online]. Available at: http://www.ai.mit.edu/˜ murphyk/Software/BNT/bnt.html

(Qt, 2015) “Qt software package for Python.” [Online]. Available at: http://www.qt.io/download/

(UbiGraph,-) “UbiGraph 3-D graph representation tool.” [Online]. Available at: http://www.ubietylab.net/ubigraph/

(Sanchez et al, 2016) Topology-Aware Self-Diagnosis framework, [Online]. Available at: https://www.youtube.com/watch?v=xNudu48quRM

(Vaarandi et al, 2015) R. Vaarandi and M. Pihelgas, "LogCluster - A data clustering and pattern mining algorithm for event logs," Network and Service Management (CNSM), 2015 11th International Conference on, Barcelona, 2015, pp. 1-7.

(Kimura et al, 2015) T. Kimura, A. Watanabe, T. Toyono and K. Ishibashi, "Proactive failure detection learning generation patterns of large-scale network logs," Network and Service Management (CNSM), 2015 11th International Conference on, Barcelona, 2015, pp. 8-14.

(Zhou et al, 2015) W. Zhou, T. Li, L. Shwartz and G. Y. Grabarnik, "Recommending ticket resolution using feature adaptation," Network and Service Management (CNSM), 2015 11th International Conference on, Barcelona, 2015, pp. 15-21.

(Univerself, 2013) FP7 UniverSelf Project deliverables. [Online]. Available: http://www.univerself-project.eu/technical-reports.